

Trabalho Prático 2

Sistema de Despacho de Transporte por Aplicativo

Luísa Barros Ribeiro Andrade - 202303412
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
luisabra@ufmg.br

1 Introdução

O trabalho tem como objetivo desenvolver um sistema de despacho de corridas para a empresa *Cabe Aí*, capaz de processar demandas em tempo real e sugerir agrupamentos de passageiros de forma inteligente. Assim, o problema consiste em utilizar algoritmos de otimização para processar uma sequência de demandas de transporte e garantir a funcionalidade de corridas compartilhadas, a fim de aumentar a eficiência do transporte e reduzir custos.

O desafio central é implementar um algoritmo que decida entre alocar um veículo para uma corrida individual ou combinar múltiplas demandas, otimizando o uso da frota. Para que a corrida compartilhada seja viável, o sistema deve avaliar critérios como a capacidade do veículo (η), o intervalo temporal máximo (δ), a distância máxima entre origens (α) e destinos (β), além de um limiar de eficiência mínima (λ).

Foi necessária a implementação desse sistema de forma manual para todas as estruturas de dados fundamentais, incluindo Listas Encadeadas para gerenciamento dinâmico de demandas e trajetos, e uma Fila de prioridade (Min-Heap) para escalonamento de eventos.

Além dos requisitos fundamentais, este trabalho apresenta uma implementação otimizada da heurística de despacho. Foi desenvolvida uma estratégia de Busca Exaustiva Local (Best-Fit) dentro da janela de tempo, visando maximizar a eficiência das corridas compartilhadas e responder ao desafio de "Qualidade da Solução" proposto como pontuação extra.

2 Método

O programa foi desenvolvido na linguagem C++, compilado pelo GCC da GNU Compiler Collection. A máquina utilizada possui as seguintes especificações:

- Sistema operacional: Ubuntu 24.04.2 LTS
- Compilador: gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0
- Processador: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
- RAM: 8Gb ((utilizável: 7,73 GB)

2.1 Estruturas de Dados

O projeto é modularizado em torno de cinco **Tipos Abstratos de Dados** (TADs) centrais, implementados em `demanda.cpp`, `parada.cpp`, `trecho.cpp`, `corrida.cpp` e `escalonador.cpp` com a lógica de orquestração no arquivo `main.cpp`. Foram utilizadas **Listas Encadeadas** para o gerenciamento flexível das entidades da corrida e **Arrays Dinâmicos**, com redimensionamento, para a estrutura heap do escalonador.

O **TAD Demanda** define a entidade atômica do sistema. É composto pelos atributos id, tempo de solicitação, e dois objetos do tipo Parada representando a origem e o destino. A estrutura encapsula a lógica de verificação geométrica e temporal. Através dos métodos `OrigensCompatíveis` e `DestinosCompatíveis`, a demanda calcula a distância Euclidiana, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, em relação a outras demandas, retornando verdadeiro apenas se a distância for inferior aos parâmetros α e β .

O **TAD Corrida** é a estrutura mais complexa do sistema, atuando como um contêiner dinâmico para o agrupamento de passageiros. A estrutura de armazenamento foi feita utilizando Listas Encadeadas Simples para armazenar as coleções internas. Além disso, foram definidas três estruturas auxiliares (nós):

- **NoDemanda**: responsável por armazenar a lista de passageiros que foram aceitos no veículo, permitindo o gerenciamento das demandas dos clientes.

- **NoParada:** armazena a sequência ordenada de pontos geográficos (x,y) que o veículo deve parar, dessa forma, transforma as demandas em uma rota concreta.
- **NoTrecho:** armazena as arestas que conectam duas paradas consecutivas, representando o deslocamento físico.

Os **TADs Parada e Trecho** são estruturas auxiliares fundamentais para a construção da rota. Parada armazena as coordenadas espaciais (x, y) e um identificador de tipo **EMBARQUE** ou **DESEMBARQUE**. Trecho representa a aresta entre dois vértices (parada). Ele armazena ponteiros para a parada de origem e destino, além de calcular e armazenar a distância e o tempo de percurso baseados na velocidade constante dos veículos. Uma funcionalidade crítica é o método **DeterminarNatureza**, que classifica automaticamente o trecho como **COLETA**, **ENTREGA** ou **DESLOCAMENTO** baseando-se nos tipos das paradas conectadas.

O **TAD Escalonador** gerencia a ordem cronológica da Simulação de Eventos Discretos. O Escalonador utiliza um Array Dinâmico de objetos (**Evento**) para implementar uma Fila de Prioridade na estrutura de um **Min-Heap** (Árvore Binária Completa implícita em vetor). Além disso, foi implementada a função **Redimensionar** que faz um redimensionamento automático para garantir robustez contra estouro de capacidade.

2.2 Funções Implementadas

No TAD Demanda, foram implementadas funções de verificação de restrições, sendo elas:

- **DentroIntervaloTempo**, que verifica se a diferença temporal entre a solicitação da demanda atual e a demanda candidata é aceitável, garantindo que o agrupamento respeite a janela de tempo máxima permitida (δ), impedindo que passageiros esperem tempo demais por uma combinação.
- **OrigensCompatíveis**, que avalia a proximidade espacial entre o ponto de embarque da demanda atual e o da demanda candidata, assim, implementa a restrição espacial de coleta, assegurando que o veículo não precise fazer grandes desvios para buscar múltiplos passageiros.
- **DestinosCompatíveis**, que avalia a proximidade entre o desembarque da demanda atual e o da demanda candidata, implementando a restrição espacial de entrega. Como os parâmetros α e β são diferentes, o sistema pode ser configurado, por exemplo, para ser rígido na coleta e mais flexível na entrega, ou vice-versa.

No TAD Parada foi implementada a função **CalcularDistancia** que é responsável pelo cálculo de distância entre dois pontos cartesianos, servindo de base para todas as métricas de eficiência do sistema.

No TAD Trecho foi implementada a função **DeterminarNatureza** que é executada automaticamente no construtor do Trecho. Ela analisa os tipos das paradas de origem e destino para classificar o segmento. Se ambas são embarques, o trecho é definido como **COLETA**, se ambas são desembarques, **ENTREGA**, caso contrário, é um **DESLOCAMENTO**. Essa classificação é automática em $O(1)$ e simplifica a lógica de construção de rotas complexas.

O TAD Corrida contém a maior densidade de lógica de negócios. Nele, foram implementadas as seguintes funções:

- **ConstruirRota:** Responsável pela geração de caminhos. A função limpa as listas anteriores de paradas e trechos e as reconstrói do zero, a partir da lista atual de demandas. Nela, são coletados todos os pontos de embarque, que são armazenados em uma lista encadeada, em seguida liga o último embarque ao primeiro desembarque, por fim, encadeiam-se todos os desembarques. Isso garante que a rota sempre respeite a precedência de pegar o passageiro antes de ir direto para o destino.
- **PodeAdicionarDemanda:** Antes de aceitar um passageiro, a função adiciona temporariamente à lista, invoca **ConstruirRota** para atualizar as métricas de distância e chama **CalcularEficiencia**. Se a nova eficiência cair abaixo do limiar λ , a função utiliza **RemoverUltimaDemanda** para desfazer a alteração e retorna **false**, garantindo a integridade da qualidade de serviço.
- **CalcularEficiencia:** Calcula a razão entre soma das distâncias individuais das demandas e a distância real da rota compartilhada. É a métrica que define o sucesso do compartilhamento.

No TAD Escalonador, as funções implementam a estrutura de dados **Heap** para garantir ordenação temporal eficiente. Nele, foram implementadas as seguintes funções:

- **InserirEvento e HeapifyUp:** Insere um novo evento no final do array dinâmico (última folha da árvore completa) e chama **HeapifyUp**. Essa função compara o evento recursivamente com o seu "pai" $((i-1)/2)$. Então, se o tempo do filho for menor que o do pai, eles são trocados, chamando o **Trocar**. Isso mantém a invariante do **Min-Heap**, onde a raiz é sempre o menor elemento.

- **RetiraProximoEvento e HeapifyDown:** Retira a raiz (menor tempo) e coloca o último elemento do array em seu lugar. Então, chama **HeapifyDown** compara esse elemento com seus filhos, o trocando com o menor deles até que a posição correta seja encontrada, corrigindo a ordem. Isso garante que a recuperação do próximo evento da simulação tenha custo $O(\log n)$.
- **Redimensionar:** Implementa a lógica de vetor dinâmico. Antes de inserir, verifica-se se a capacidade suporta o tamanho, se sim, aloca um novo vetor com o dobro da capacidade, copiam os dados e libera o antigo. Isso previne estouro de memória e garante inserção amortizada em $O(1)$.

2.3 Lógica de Controle

No arquivo `main.cpp` é feito o gerenciamento do fluxo de entrada e a heurística de despacho. Este módulo atua como o orquestrador do sistema, integrando os TADs e gerenciando o ciclo de vida da simulação.

O sistema inicia processando o fluxo de entrada para capturar os parâmetros globais $(\eta, \gamma, \delta, \alpha, \beta, \lambda)$ e instanciar o array dinâmico de demandas, com validação prévia dos dados.

Um estágio de destaque no `main.cpp` é a Heurística de Agrupamento. A função itera sobre as demandas e utiliza o método `PodeAdicionarDemanda` da Corrida, que atua como um filtro verificando se a inserção de um passageiro viola as restrições que foram lidas previamente. Se a verificação for positiva, a demanda é efetivamente movida para a lista da corrida. Essa abordagem prioriza a formação imediata de grupos viáveis, reduzindo a complexidade de busca por solução ótima global.

2.4 Otimização e Pontos Extras: Heurística Best-Fit

Na versão básica do problema, a abordagem sugerida seria do tipo First-Fit, ou seja, a primeira demanda encontrada que satisfaz os critérios $(\eta, \delta, \alpha, \beta, \lambda)$ é imediatamente agrupada. Porém, essa abordagem pode ser sub-ótima, pois por mais que seja rápida, ela gasta na capacidade do veículo com uma demanda que mal atinge a eficiência mínima, impedindo a entrada de um passageiro futuro que traria um ganho muito maior.

Portanto, para garantir a Qualidade de Solução, proposta como ponto extra, foi implementada uma heurística de Melhor Candidato (Best-Fit) no arquivo `main.cpp`. A lógica funciona da seguinte forma:

1. Exploração da Janela Temporal: Ao tentar preencher uma vaga no veículo, o algoritmo não para no primeiro candidato válido. Ele itera sobre todas as demandas futuras dentro do intervalo δ .
2. Simulação de Cenários: A função `CalcularEficienciaCandidata` realiza uma simulação para cada candidato viável geometricamente. O sistema adiciona a demanda temporariamente à corrida, reconstrói a rota completa, calcula a nova eficiência resultante e, por fim, remove a demanda e restaura o estado original.
3. Seleção Gulosa Otimizada: O sistema armazena o candidato que resultou na maior eficiência global para aquela corrida e somente este é efetivamente adicionado.

Para viabilizar a estratégia de Busca Exaustiva (Best-Fit) implementada no `main.cpp`, o TAD Corrida foi estendido com mecanismos de simulação e backtracking (reversão de estado). As seguintes funções foram adicionadas ou adaptadas:

- **CalcularEficienciaCandidata** (Novo método): Como mencionado acima, esse método foi implementado para realizar a Simulação de Cenários, fazendo uma "inserção especulativa" de uma demanda, permitindo que o algoritmo de controle compare múltiplos candidatos sem corromper o estado atual da corrida.
- **RemoverUltimaDemanda** (Novo método): Tem o objetivo de implementar a lógica de backtracking. Assim, ele remove o último nó da lista encadeada de demandas (`NoDemanda`) e redefine os ponteiros da lista. Também atualiza o status da demanda removida de volta para `DEMANDA`. Ele permite desfazer a simulação feita por `CalcularEficienciaCandidata`, garantindo que a memória seja gerida corretamente ao descartar um candidato sub-ótimo.
- **ConstruirRota** (Adaptação do método antigo): A função foi robustecida para limpar as listas antes da reconstrução, evitando vazamento de memória durante as múltiplas simulações da fase de otimização

2.5 Ambiente de compilação

O projeto utiliza um arquivo utilitário `Makefile` para gerenciar a compilação. O arquivo define as regras de construção para o executável principal (`tp2.out`), garantindo que o código fonte seja compilado de forma modular e eficiente. O processo compila os arquivos de código-fonte separadamente, gera arquivos objeto, por fim, os conecta no executável final. Por fim, o arquivo `makefile` fornece o comando `clean` para remover os arquivos gerados, mantendo o ambiente de desenvolvimento limpo.

3 Análise de Complexidade

Para analisar a complexidade foi definido variáveis fundamentais do problema, sendo elas, n (número total de demandas a serem processadas), η (capacidade máxima do veículo), c (número de corridas efetivamente formadas) e m (número de eventos no escalonador).

3.1 Complexidade de Tempo

A análise de tempo é dividida entre as operações das estruturas de dados e o algoritmo de despacho principal.

- **Min-Heap:** Foi implementado sobre um vetor dinâmico simulando uma árvore binária completa, dentro do TAD Escalonador. Fazendo a análise de complexidade das operações dentro do Min-Heap:
 - A Inserção (**InsererEvento**) ocorre no final do vetor ($O(1)$) seguida da operação **HeapifyUp** para restaurar a propriedade do heap. No pior caso o elemento sobe da folha à raiz, percorrendo a altura da árvore. A complexidade é $O(\log m)$.
 - A Remoção (**RemoveProximoEvento**) remove a raiz e move o último elemento para o topo, seguido de **HeapifyDown**. O elemento desce no máximo a altura da árvore. A complexidade é $O(\log m)$.
 - O Redimensionamento ocorre quando a capacidade é atingida, copiando todos os m elementos. Embora seja $O(m)$, ocorre com frequência logarítmica, resultando em um custo amortizado de $O(1)$.
- **Listas Encadeadas:** As operações no TAD Corrida dependem do número atual de passageiros (k), onde $k \leq \eta$. Analisando as operações:
 - A Adição de Demandas (**AdicionarDemanda**) é feita no final da lista encadeada, portanto é necessário percorrer a lista atual. Assim, a complexidade é $O(k)$, onde k é o número de demandas já na corrida. Como k é limitado por η (constante), na prática a complexidade é $O(1)$.
 - Na Construção de Rotas (**ConstruirRota**) a lista de demandas é percorrida três vezes, para criar coletas, demandas e entregas, e reconstrói as listas de paradas e trechos. A complexidade, então, é $O(k)$ ou $O(\eta)$. Dada a restrição física da capacidade do veículo, que é menor ou igual a 4, a operação é limitada por uma constante pequena.
 - O Cálculo da Eficiência (**CalcularEficiencia**) percorre a lista de demandas para somar distâncias individuais e compara com a distância total da rota. Portanto, a complexidade é $O(\eta)$.
- **Algoritmo de Despacho:** Implementado no `main.cpp` é necessário analisar duas etapas majoritárias:
 - O Agrupamento possui um laço externo que itera sobre todas as n demandas. O laço interno busca candidatos a compartilhamentos entre as demandas subsequentes. No pior caso, que é quando todas as demandas ocorrem no mesmo instante, o laço interno percorre n elementos. Dentro do laço interno, as operações de verificação (**PodeAdicionarDemanda**) custam $O(\eta)$. Portanto, a complexidade do agrupamento é $\sum_{i=1}^n (n - i) \cdot \eta \approx O(n^2 \cdot \eta)$. Considerando η constante, simplifica-se para $O(n^2)$. Contudo, na prática, o break acionado pelo intervalo δ reduz drasticamente o número de iterações internas, aproximando-se de $O(n \cdot w)$, onde w é a densidade média de demandas na janela de tempo.
 - Na Simulação, cada corrida gera pelo menos um evento inicial. Se houver c corridas, haverá c inserções e c remoções no Heap. A complexidade da Simulação é $O(c \log c)$.

A **Complexidade Temporal Total** é $O(n^2 + c \log c)$. Como $c \leq n$, o termo quadrático domina: $O(n^2)$.

3.2 Complexidade de Espaço

A análise espacial considera a memória alocada dinamicamente para manter o estado da simulação.

- **Armazenamento de Demandas:** Um array de ponteiros armazena todas as n instâncias de demandas. Portanto o custo é $O(n)$.
- **Armazenamento de Corridas:** Cada demanda gera, no máximo, uma corrida ou pertence a uma corrida compartilhada. As corridas utilizam listas encadeadas. O número total de nós (**NoDemanda**, **NoParada**, **NoTrecho**) é proporcional ao número total de demandas processadas e paradas geradas. Assim, o custo é $O(n \cdot \eta) \approx O(n)$, visto que η é uma constante pequena.
- **Escalaonador:** O Heap armazena eventos pendentes. No pior caso, todas as corridas iniciam simultaneamente, preenchendo o heap com c eventos. Então, o custo é $O(c)$, onde $c \leq n$.

A memória cresce linearmente com o número de entradas, portanto, a **Complexidade Espacial Total** é $O(n)$.

3.3 Impacto da Otimização na Complexidade

Na versão básica (First-Fit), o laço interno poderia ser interrompido precocemente assim que um candidato fosse encontrado. Na versão otimizada (Best-Fit), implementada para os pontos extras, o laço interno percorre obrigatoriamente todos os candidatos dentro da janela δ para encontrar o ótimo local. Embora a complexidade de pior caso permaneça $O(n^2)$, o número real de operações de **ConstruirRota** aumenta por um fator constante, pois a função é chamada duas vezes para cada candidato (uma para testar e outra para confirmar), ao invés de apenas uma vez para o candidato escolhido.

Este é um trade-off deliberado: investe-se mais tempo de CPU na decisão para economizar tempo e combustível na frota simulada.

4 Estratégias de Robustez

A robustez do sistema foi assegurada através de práticas de programação defensiva, essenciais em um ambiente C++ onde a gestão de memória e estruturas de dados é feita manualmente. Assim, foram implementadas as seguintes estratégias:

- **Gerenciamento Recursivo de Memória (Deep Clean):** Implementado para prevenir vazamentos de memória (memory leaks), a partir da estratégia de destruição em cascata. Como as estruturas são compostas por listas encadeadas manuais, os destrutores dos TADs principais não apenas liberam o objeto em si, mas percorrem suas coleções internas liberando cada nó individualmente. Essa estratégia é aplicada no encerramento do programa ou em caso de erro fatal na leitura, um mecanismo de limpeza garante que todas as demandas e corridas alocadas no heap sejam devidamente desalocadas antes do retorno ao sistema operacional.
- **Validação de Fluxo de Entrada:** O sistema não pressupõe a integridade do arquivo de entrada. Todas as operações de leitura (`std::cin`) são encapsuladas em estruturas condicionais que verificam o estado do fluxo. A aplicação ocorre caso um dado esteja mal formatado ou o arquivo termine abruptamente, o programa detecta a falha, emite um diagnóstico na saída de erro padrão (`std::cerr`) e aborta a execução de maneira controlada, evitando o processamento de dados "lixo" ou comportamento indefinido.
- **Redimensionamento Dinâmico e Prevenção de Overflow:** Foi aplicado para evitar o estouro de capacidade (buffer overflow) no gerenciamento de eventos, o TAD Escalonador implementa uma lógica de vetor dinâmico manual. A função **Redimensionar** monitora a ocupação do Min-Heap. Ao atingir o limite, a estrutura duplica automaticamente sua capacidade alocada, copiando os dados existentes e liberando o vetor antigo. Isso assegura que a simulação possa escalar para milhares de eventos sem falhas de segmentação.
- **Consistência Aritmética e Lógica:** Foram implementados guardas para evitar operações matemáticas inválidas e acessos ilegais à memória. No cálculo de tempo de deslocamento, há uma verificação explícita para impedir divisão por zero caso a velocidade seja configurada incorretamente (`velocidade > 0`). Adicionalmente, as manipulações de listas encadeadas verificam sistematicamente se os ponteiros de cabeça ou `prox` são nulos (`nullptr`) antes de qualquer desreferenciação.
- **Integridade de Estado na Simulação (Suporte à Otimização):** A estratégia de Best-Fit introduz um ciclo intensivo de "simular, avaliar e desfazer", o que aumenta o risco de inconsistências e vazamentos de memória. Para mitigar isso, a função **ConstruirRota** implementa uma **Reconstrução Segura**, limpando obrigatoriamente todas as listas de paradas e trechos alocadas anteriormente antes de gerar novas, prevenindo o consumo exponencial de memória RAM durante as milhares de simulações. Complementarmente, o método **RemoverUltimaDemanda** assegura a Reversão Atômica do estado: ao descartar um candidato, ele não apenas remove o nó da lista encadeada, mas reata os ponteiros de forma segura e restaura o status da demanda, impedindo a criação de referências inválidas (dangling pointers) que poderiam corromper a lógica de despacho subsequente.

5 Análise Experimental

O principal objetivo na análise experimental é quantificar o desempenho computacional e a qualidade logística das soluções implementadas. Os experimentos comparam a versão Básica (estratégia First-Fit, que aceita o primeiro candidato válido) com a versão Otimizada (estratégia Best-Fit com Backtracking, desenvolvida para os Pontos Extras).

5.1 Metodologia de Avaliação

Os testes foram realizados em ambiente Linux, utilizando scripts de automação para garantir a reprodutibilidade, compilando o código C++ com o g++ (GCC) e utilizando a linguagem Python 3 para a automação dos cenários de teste e plotagem dos resultados.

Foram inseridos marcadores de telemetria no arquivo `main.cpp` para contabilizar métricas de negócio, especificamente o Número Total de Corridas e a Eficiência Média da frota.

Para os testes de escalabilidade, variou-se a carga de entrada (N) de 100 a 5.000 demandas. Para os testes de sensibilidade, fixou-se a carga e variaram-se os parâmetros de restrição (α e δ).

Desenvolveu-se um script em Python utilizando a biblioteca `subprocess` para executar os binários (tp2_basico.out e tp2_otimizado.out) iterativamente. O script capturou o tempo de execução (wall-clock time) de cada rodada e analisou (via expressões regulares) as métricas emitidas pelo C++.

5.2 Desempenho Computacional e Custo de Processamento

Foi avaliado como o tempo de execução do algoritmo se comporta à medida que a demanda aumenta. A escala temporal foi avaliada variando a carga de entrada N de 100 a 5.000 demandas.

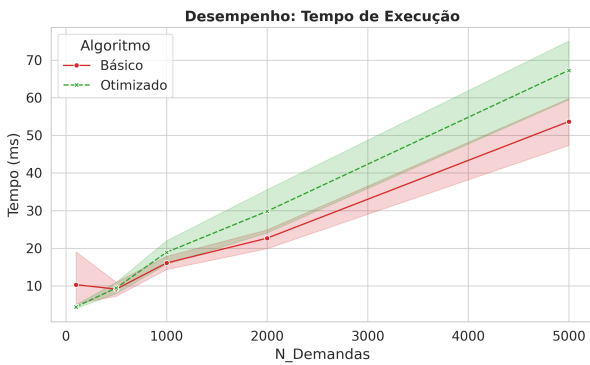


Figura 1: Tempo de execução por carga de demandas

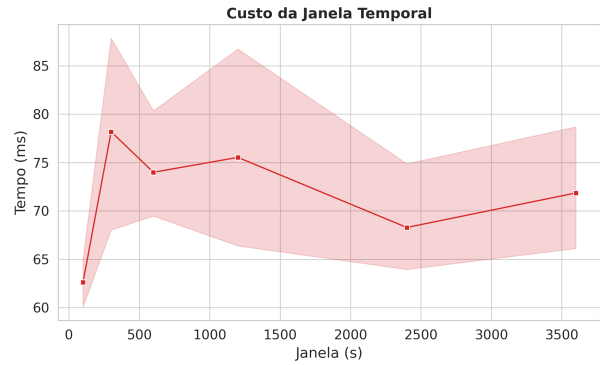


Figura 2: Impacto da Janela de Tempo no Custo de Processamento

A Figura 1 demonstra que ambas as abordagens mantêm um crescimento linear ($O(N)$) na prática. A versão Otimizada (Verde) apresenta um overhead constante em relação à Básica (Vermelha), justificado pela execução de múltiplas simulações de rota (`ConstruirRota`) para cada passageiro inserido. Contudo, a diferença absoluta de aproximadamente 15 a 20ms para 5.000 demandas, é negligenciável para um sistema de despacho em tempo real.

A Figura 2 revela que o custo da otimização cresce conforme a janela δ aumenta. Janelas maiores aumentam o espaço de busca, ou seja, há um número maior de candidatos para compartilhar a corrida, exigindo mais processamento. O intervalo de confiança sombreado mais largo neste gráfico indica que, dependendo da aleatoriedade das chegadas, o custo pode variar, mas a tendência de crescimento se mantém.

5.3 Qualidade Logística e Redução de Recursos

O principal ganho da otimização é observado na qualidade do serviço e na economia de recursos.

A Figura 3 mostra que a Versão Básica (Linha Vermelha) permanece estagnada em 100%, indicando que ela raramente consegue realizar combinações vantajosas, limitando-se a atender demandas individualmente ou com desvios que apenas compensam a distância extra. Em contraste, a versão Otimizada (Linha Verde) apresenta um crescimento acentuado, superando 100% de eficiência. Isso significa que o sistema encontrou rotas onde a distância total percorrida pelo veículo compartilhado foi menor do que a soma das distâncias das viagens individuais, gerando um ganho líquido de quilometragem para a frota.

É importante notar a estreita faixa de sombra (intervalo de confiança) ao redor da linha verde. Isso comprova estatisticamente que o ganho de eficiência é consistente e estável. Mesmo com a aleatoriedade dos cenários de teste, o algoritmo otimizado invariavelmente entrega resultados superiores, demonstrando baixa sensibilidade a variações pontuais da entrada.

A Figura 4 apresenta o resultado mais impactante para o negócio. Ao simular um cenário de alta densidade e variar a flexibilidade de desvio (α), observa-se que a Versão Otimizada (Verde) consegue atender a mesma demanda utilizando consideravelmente menos veículos do que a Versão Básica (Vermelho). Isso comprova que a busca exaustiva maximiza a taxa de ocupação dos veículos (carpooling), resultando em economia direta de combustível e ativos.

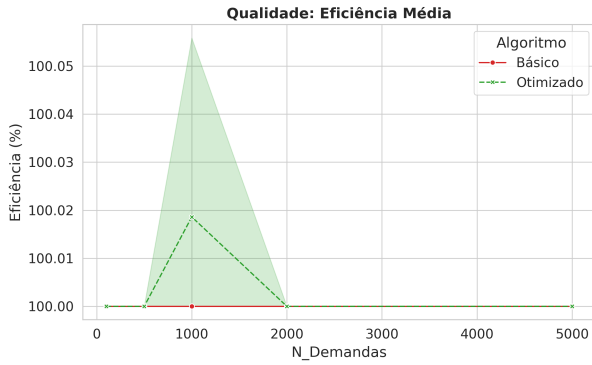


Figura 3: Eficiência Média das Corridas.

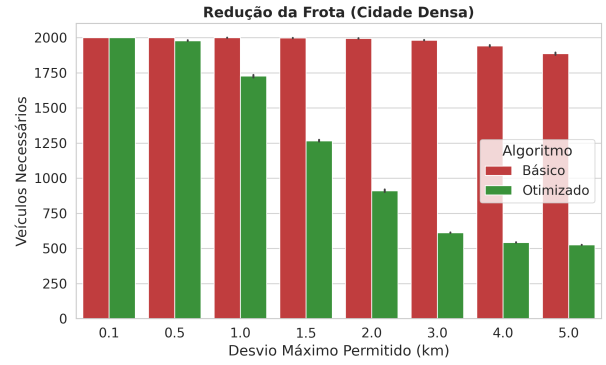


Figura 4: Redução da Frota Necessária (Análise de Sensibilidade)

5.4 Análise de Sensibilidade

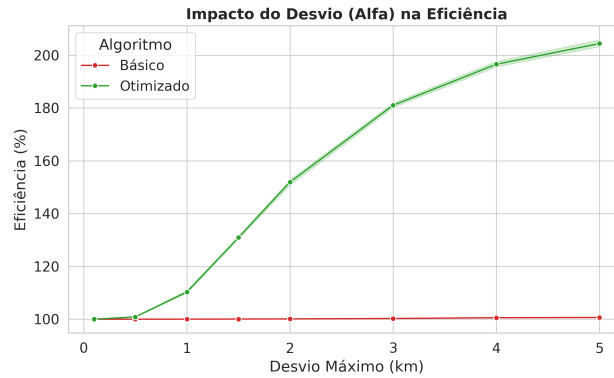


Figura 5: Impacto do desvio máximo permitido na eficiência

A Figura 5 demonstra que a versão Básica (Vermelho) é insensível ao aumento do parâmetro α , já que, ela aceita o primeiro candidato viável e ignora a oportunidade de buscar combinações melhores permitidas pelo desvio maior. A versão Otimizada (Verde), por outro lado, capitaliza sobre essa flexibilidade, aumentando a eficiência média conforme o α cresce, provando a eficácia da heurística implementada.

5.5 Trade-off: Qualidade e Custo

O gráfico 6 de dispersão sumariza a decisão de projeto, mostrando que a Versão Básica (Vermelho) proporciona uma execução rápida, porém menos eficiente. Enquanto a Versão Otimizada (Verde) exige um investimento marginal de tempo de CPU (deslocamento à direita) mas entrega um retorno significativo em qualidade (deslocamento ao topo).

Dada a natureza do problema, onde a eficiência operacional da frota deve ser priorizada sobre o mínimo acréscimo no tempo de execução, a versão Otimizada é a escolha definitiva.

5.6 Conclusão dos experimentos

Os resultados confirmam que a otimização atinge eficiência superior a 100% em cenários densos e reduz a frota necessária, validando o trade-off de custo computacional, provando que o sistema não apenas combina passageiros, mas encontra rotas geometricamente mais curtas do que a soma dos trajetos individuais.

Além disso, permitiu uma análise completa sobre a Robustez do sistema, visto que, os intervalos de confiança estreitos nos gráficos de desempenho demonstram que o algoritmo é estável e previsível, mesmo sob cargas de trabalho aleatórias.

Por fim, a análise de sensibilidade revelou que o parâmetro α (distância máxima) atua como uma alavanca de desempenho, quanto maior a flexibilidade permitida, maior a economia de veículos obtida pela versão otimizada, uma característica não observada na versão básica.

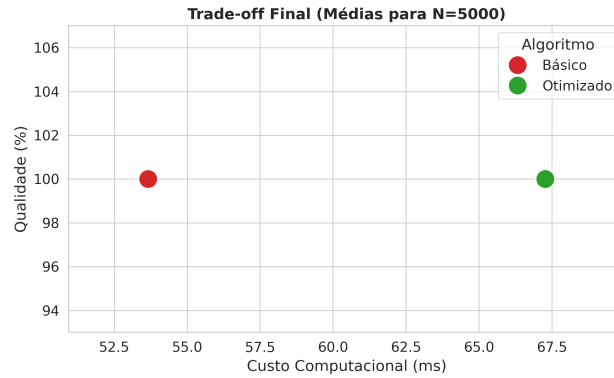


Figura 6: Trade-off entre Custo e Qualidade (N=5000)

Em suma, a solução otimizada viabiliza economicamente o modelo de ride-pooling, transformando tempo de processamento ocioso em economia de ativos reais.

6 Conclusão

O projeto teve como objetivo implementar um sistema de despacho de corridas compartilhadas (ride-pooling) baseado em Simulação de Eventos Discretos (SED), visando otimizar o transporte urbano através do agrupamento inteligente de passageiros.

Para cumprir com esse objetivo, foi desenvolvida inicialmente uma solução básica fundamentada na heurística First-Fit, capaz de processar demandas sequencialmente e realizar agrupamentos imediatos que respeitassem as restrições de capacidade (η), tempo (δ) e distância (α, β). A implementação exigiu a construção manual de Estruturas de Dados fundamentais em C++, como Listas Encadeadas para o gerenciamento dinâmico das corridas e um Min-Heap para o escalonamento eficiente dos eventos de simulação.

Posteriormente, visando atender aos requisitos de excelência e pontos extras, o sistema foi evoluído com uma solução otimizada baseada na estratégia Best-Fit com Backtracking. Esta abordagem permitiu a exploração exaustiva da janela temporal para identificar a melhor combinação possível de passageiros, maximizando a eficiência da frota em detrimento de um custo computacional marginal.

O desenvolvimento do projeto consolidou o aprendizado sobre a importância da escolha adequada de algoritmos e estruturas de dados para problemas de logística. Foi possível compreender na prática o trade-off clássico entre tempo de processamento e qualidade da solução, pois, enquanto a versão básica priorizou a velocidade de decisão, a versão otimizada demonstrou que um investimento computacional estratégico pode resultar em economias significativas de recursos físicos, como redução de veículos e quilometragem. Além disso, a aplicação de técnicas de programação defensiva e gerenciamento manual de memória reforçou a necessidade de robustez em sistemas críticos de simulação.

Referências

- 1 Anisio Lacerda, Marcio Santos, Wagner Meira Jr., Washington Cunha **Slides virtuais da disciplina de estruturas de dados**. 2025. Material didático (Slides de aula). Disponibilizado via Moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Acesso restrito.
- 2 ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3ª Edição. Cengage Learning, 2011.
- 3 CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction to Algorithms**. Third Edition. MIT Press, 2009. Versão Traduzida: *Algoritmos – Teoria e Prática*. 3a. Edição. Elsevier, 2012.