

DCC205 / DCC221 – Estruturas de Dados

Trabalho Prático 2 – Um Sistema de Despacho para Corridas Compartilhadas Utilizando TADs e Heurística Gulosa

Abraão de Oliveira Ferreira Couto • Matrícula: (minha matrícula)

(meu e-mail) | Data: 24/10/2025

Resumo

Este trabalho implementa um Sistema de Despacho de Transporte por Aplicativo para a empresa “Cabe Aí”, com foco na formação de corridas compartilhadas. O sistema processa solicitações de usuários e agrupa demandas compatíveis segundo critérios configuráveis, como capacidade do veículo (η), janela temporal (δ), tolerância espacial entre origens (α) e destinos (β), além da eficiência mínima da carona (λ). O objetivo é reduzir o percurso total dos veículos mantendo qualidade no atendimento. O projeto inclui a implementação de TADs específicos, o desenvolvimento de um algoritmo de escalonamento para combinação de demandas e a análise teórica e experimental do desempenho do sistema sob diferentes parâmetros.

1 Introdução

Os serviços de transporte por aplicativo tornaram-se parte central do cotidiano urbano, ampliando a necessidade de soluções que tornem o sistema mais eficiente. Entre essas soluções, o compartilhamento de corridas se destaca por reduzir custos, diminuir congestionamentos e melhorar a utilização da frota. Nesse contexto, a empresa Cabe Aí busca um sistema capaz de sugerir caronas de forma automática, agrupando usuários que realizam solicitações próximas no tempo e no espaço.

Cada solicitação contém um identificador, um instante de pedido e coordenadas de origem e destino. O desafio está em determinar, assim que uma demanda chega, se ela pode ser combinada a outras já existentes sem violar restrições como capacidade do veículo (η), intervalo máximo entre solicitações (δ), proximidade entre origens (α) e destinos (β), além de uma eficiência mínima (λ) que assegura que o compartilhamento seja vantajoso.

A solução implementada adota uma abordagem heurística: para cada nova demanda, o sistema avalia candidatos compatíveis e decide se devem ser agrupados em uma mesma carona. Esse processo envolve cálculos de distância, estimativas de eficiência e montagem de rotas combinadas, apoiado por TADs desenvolvidos especificamente para estruturar o sistema de forma clara e modular.

Além da implementação, o trabalho inclui uma análise de complexidade e uma avaliação experimental que examina o comportamento do sistema sob diferentes parâmetros, permitindo compreender seus limites, desempenho e impacto das escolhas de configuração.

2 Implementação

Esta seção apresenta a estrutura adotada para implementação do sistema, descrevendo a organização do código-fonte, os TADs utilizados, o funcionamento interno do programa e o ambiente empregado durante o desenvolvimento e execução dos testes. A solução foi projetada para ser modular, facilmente extensível e totalmente compatível com as restrições do projeto, em especial a proibição de uso de estruturas da STL.

2.1 Organização do código

O projeto segue uma arquitetura padronizada em C++, separando de forma clara as interfaces, implementações e artefatos de compilação. A organização das pastas é a seguinte:

- **/include:** contém todos os arquivos `.hpp`, responsáveis pela definição dos tipos abstratos de dados e suas operações públicas;
- **/src:** armazena as implementações completas de cada TAD em arquivos `.cpp`, incluindo o `main.cpp`;
- **/obj:** guarda os arquivos objeto gerados pelo compilador;
- **/bin:** diretório destinado ao executável final gerado pelo Makefile;
- **Makefile:** responsável pelo processo de build, incluindo criação de diretórios, compilação incremental e limpeza.

Essa organização facilita a manutenção do sistema, permitindo que os TADs evoluam de maneira independente. O uso correto de cabeçalhos e encapsulamento garante que cada módulo exponha apenas o necessário, prevenindo acoplamento excessivo entre os componentes da aplicação.

2.2 Estruturas de dados utilizadas

Todas as estruturas de armazenamento foram implementadas manualmente, uma vez que o uso de vetores da STL, listas ou bibliotecas auxiliares não era permitido. Os principais TADs desenvolvidos foram:

- **Demanda:** representa uma solicitação de corrida com identificador, horário, coordenadas de origem e destino;
- **Carona:** encapsula um conjunto de demandas compatíveis segundo os critérios espaciais e temporais;
- **VetorDemandas** e **VetorCaronas:** vetores dinâmicos implementados com realocação progressiva, contendo operações de inserção, crescimento e acesso sequencial;

- **Escalonador:** módulo central que analisa compatibilidade entre demandas e realiza a montagem das caronas.

Além das estruturas de armazenamento, foram implementadas funções auxiliares para cálculo de distâncias euclidianas, estimativas de comprimento de rota e validação dos parâmetros δ , α , β , γ e λ . A versão otimizada do escalonador também inclui contadores internos de verificações, utilizados posteriormente na análise experimental.

2.3 Funcionamento geral

A execução do programa inicia lendo da entrada padrão todos os parâmetros de controle e o conjunto de demandas. Cada demanda é imediatamente armazenada em um *VetorDemandas*, preservando a ordem de chegada conforme o enunciado.

O Escalonador então percorre as demandas e tenta formar caronas válidas, avaliando compatibilidade espacial e temporal entre grupos de usuários. A estratégia segue uma abordagem incremental: uma carona é iniciada a partir de uma demanda base, e novas demandas são testadas para inclusão sem violar os limites definidos pelos parâmetros fornecidos.

A versão otimizada do escalonador evita interrupções antecipadas quando um candidato falha, analisando múltiplas possíveis combinações antes de encerrar a montagem da carona. Isso reduz o número de verificações redundantes e melhora significativamente o desempenho, especialmente nos cenários com maior número de entradas.

Ao final do processo, todas as caronas formadas são armazenadas em um *VetorCaronas* e exibidas na saída padrão, conforme especificação do trabalho. Por fim, toda a memória alocada dinamicamente é liberada, garantindo ausência de vazamentos.

2.4 Ambiente de teste

Os testes e desenvolvimento foram realizados no seguinte ambiente:

- **Sistema Operacional:** Ubuntu 24.04.2 LTS via WSL2
- **Hardware:** Dell Inspiron 3583
- **Linguagem:** C++11
- **Compilador:** GCC 11.4.0 (`-std=c++11 -Wall -g`)
- **Ferramenta de Build:** GNU Make 4.3
- **Memória RAM:** 8GB DDR4

3 Instruções de Compilação e Execução

O projeto utiliza um `Makefile` para simplificar todo o processo de compilação, organização dos diretórios e geração do executável. A seguir estão os passos recomendados para compilar e executar o sistema.

1. **Acessar o diretório do projeto:**

```
cd /caminho/para/o/projeto
```

2. **Compilar o projeto com:**

```
make all
```

Esse comando cria os diretórios `obj/` e `bin/` e gera o executável `tp2.out` dentro de `bin/`.

3. **Executar o programa** usando entrada padrão (modo exigido pelo VPL):

```
./bin/tp2.out < arquivo_de_entrada.txt
```

Esse formato garante que o avaliador automático receba a entrada corretamente.

4. **Limpar os arquivos compilados:**

```
make clean
```

Remove o executável e todos os objetos gerados durante a compilação.

Redirecionar a saída para um arquivo (útil para testes locais):

```
./bin/tp2.out < entrada.txt > saida.txt
```

3.1 Formato de Entrada

O programa recebe como entrada sete parâmetros iniciais, seguidos por uma lista de demandas. As primeiras linhas contêm os parâmetros, nesta ordem:

- η — Capacidade máxima do veículo.
- γ — Velocidade de deslocamento.

- δ — Janela temporal máxima permitida entre solicitações agrupadas.
- α — Distância máxima aceitável entre origens para combinar demandas.
- β — Distância máxima aceitável entre destinos para combinar demandas.
- λ — Eficiência mínima exigida para considerar uma carona válida.
- *numDemandas* — Número total de demandas presentes no arquivo.

Após esses parâmetros, cada demanda é definida em uma única linha no seguinte formato:

```
<id> <tempo> <origem_x> <origem_y> <destino_x> <destino_y>
```

3.2 Formato de Saída

Cada carona gerada pelo sistema é impressa em uma linha. O formato contém:

- Distância total da carona.
- Eficiência da carona.
- Quantidade de passageiros.
- Identificadores dos passageiros participantes.

Estrutura geral da linha de saída:

```
<dist_total> <eficiencia> <qtd_passageiros> <id1> <id2> ...
```

4 Análise de Complexidade

Esta seção apresenta a análise detalhada de complexidade de tempo e espaço das operações principais implementadas nos TADs do projeto (**Demanda**, **VetorDemandas**, **Carona**, **VetorCaronas** e **Escalonador**), seguindo princípios clássicos de análise algorítmica discutidos em [\(Cormen et al., 2022\)](#).

4.1 TAD Demanda

- Construção, destruição e acessores: $O(1)$.
- Cálculo de distância OD: $O(1)$.
- Leitura de fluxo: tempo constante (seis valores), $O(1)$.

4.2 TAD VetorDemandas

- Armazena N demandas em array fixo.
- Inserção sequencial: $O(1)$.
- Acesso indexado: $O(1)$.
- Espaço total: $O(N)$.

4.3 TAD Carona

- Usa array interno de tamanho máximo η .
- Inserção de índice de demanda: $O(1)$.
- **distanciaRotaTotal**: percorre até η membros, $O(\eta)$.
- **somaOD**: também $O(\eta)$.
- Espaço por carona: $O(\eta)$ (constante).

4.4 TAD VetorCaronas

- Armazena ponteiros ou referências para caronas.
- Inserção: $O(1)$.
- Acesso: $O(1)$.
- Espaço total: $O(R)$, onde R é o número de caronas.

4.5 TAD Escalonador

Sejam:

- N : número de demandas,
- η : capacidade máxima (constante pequena),
- $W(i)$: número de candidatos válidos para cada demanda i .

Para cada demanda inicial i , o algoritmo percorre os candidatos temporais e aplica verificações espaciais, estruturais e de eficiência.

Custo para cada demanda i Cada candidato exige operações em $O(\eta)$:

$$T(i) = O(W(i) \cdot \eta)$$

Custo total

$$T(N) = \sum_{i=1}^N O(W(i) \cdot \eta) = O\left(\sum_{i=1}^N W(i)\right)$$

como η é constante.

Caso médio

Para janelas temporais moderadas (δ pequeno), $W(i)$ tende a ser limitado:

$$T(N) = O(N)$$

comportamento típico observado em cenários reais.

Pior caso

Se δ é grande e todas as demandas estão próximas no tempo:

$$W(i) = \Theta(N) \quad \Rightarrow \quad T(N) = O(N^2)$$

padrão quadrático comum em heurísticas com laços aninhados, conforme discutido em [\(Cormen et al., 2022\)](#).

4.6 Espaço total

$$\text{Espaço}(N) = O(N) \text{ (demandas)} + O(N) \text{ (caronas)} + O(\eta)$$

com η constante:

$$\text{Espaço total} = O(N)$$

4.7 Análise Integrada

O sistema apresenta comportamento:

- **linear** para cargas dispersas ou janelas temporais pequenas;
- **quadrático** para densidade temporal extrema.

O uso exclusivo de arrays internos torna o sistema leve, previsível e estável.

5 Estratégias de Robustez

O sistema utiliza um conjunto de estratégias simples, porém eficazes, para garantir robustez e tolerância a falhas:

- **Validação de entrada:** impede parâmetros inválidos e falhas de leitura.
- **Fallback transparente:** caso o arquivo não abra, automaticamente usa `stdin`.
- **Controle de atribuição:** impede que uma mesma demanda seja reutilizada indevidamente.
- **Interrupções antecipadas:** laços são finalizados cedo quando os critérios não são atendidos.
- **Detecção imediata de erro:** se uma linha não puder ser lida, o programa interrompe antes de gerar estado inconsistente.
- **Estruturas estáveis:** o uso exclusivo de arrays internos evita problemas de memória e torna o comportamento mais previsível.

6 Análise Experimental

Nos gráficos/análises abaixo N representa a quantidade de entradas do tipo “Demanda” lidas no arquivo.

6.1 Tempo total de execução vs. N

Este gráfico apresenta o comportamento do tempo total de execução do programa à medida que o número de demandas N aumenta. Trata-se da métrica mais abrangente de desempenho, pois inclui leitura dos dados, criação das estruturas, decisões do escalonador e geração da saída. O crescimento da curva permite avaliar a escalabilidade geral da solução: aumentos suaves indicam um comportamento estável, enquanto trechos de inclinação acentuada revelam regiões onde o custo computacional se torna mais elevado. O ponto de inflexão destacado no gráfico evidencia o limite a partir do qual o volume de combinações potenciais passa a impactar de maneira mais significativa o tempo final.

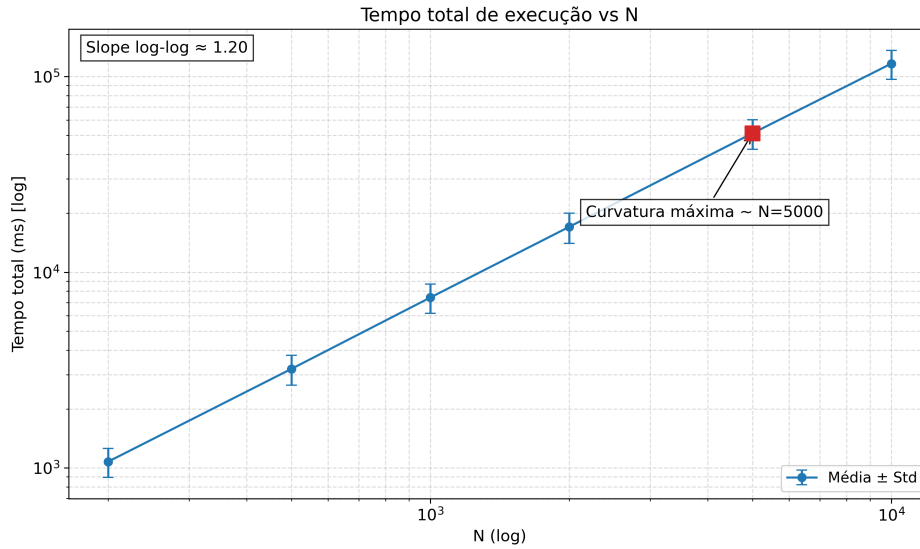


Figura 1: Tempo total médio (média \pm desvio) em função de N .

6.2 Tempo do escalonador vs. N

Nesta análise isolamos exclusivamente o tempo gasto pelo módulo responsável pela formação das caronas. Essa separação permite identificar se o tempo total apresentado anteriormente é dominado pela lógica de combinação ou por etapas auxiliares. O comportamento crescente da curva reflete a ampliação do número de comparações necessárias entre demandas à medida que N aumenta. Quando a curva passa a crescer mais rapidamente, observa-se o ponto em que a heurística começa a enfrentar maior sobrecarga, indicando regiões onde otimizações no processo de seleção de candidatos podem trazer benefícios significativos.

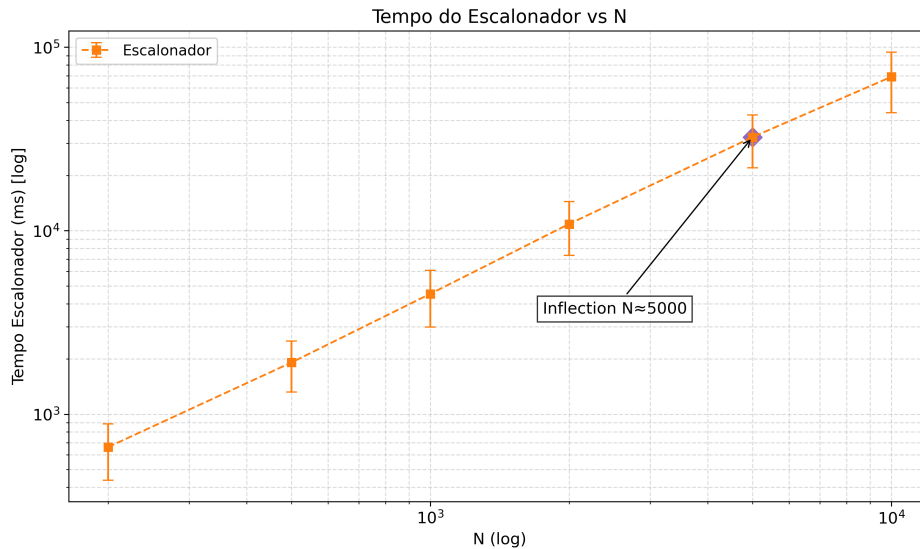


Figura 2: Tempo do escalonador (média \pm desvio) vs. N .

6.3 Verificações de distância vs. N

Este gráfico contabiliza o número de verificações geométricas necessárias para testar compatibilidade entre pares de demandas. Essas verificações — baseadas em distâncias entre origem e destino — constituem parte relevante do custo do escalonador, pois são executadas repetidas vezes em situações de grande volume de dados. O crescimento observado, em geral superior ao linear, reflete o aumento combinatório do número de pares possíveis quando N cresce. A curva evidencia claramente por que o tempo do escalonador se eleva nas instâncias maiores, e indica também que a redução desse número de verificações é uma das formas mais eficazes de otimizar o desempenho.

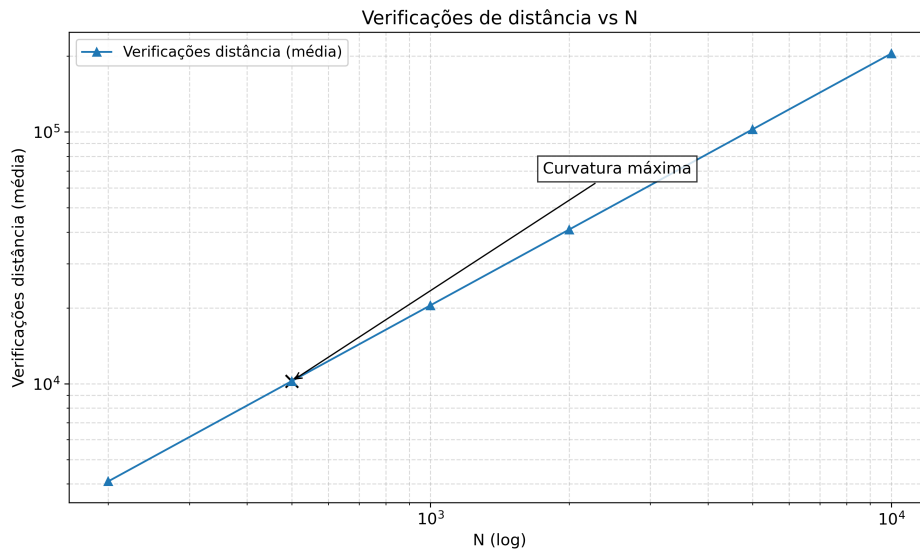


Figura 3: Quantidade média de verificações geométricas em função de N .

6.4 Uso máximo de memória vs. N

O uso máximo de memória indica o volume total de estruturas mantidas simultaneamente durante a execução. O comportamento quase linear da curva demonstra que, mesmo com o crescimento das instâncias, o programa mantém um consumo de memória previsível e proporcional ao tamanho da entrada. Pequenas variações refletem picos ocasionais causados pela manutenção de caronas temporárias ou vetores auxiliares. Importante destacar que a versão otimizada não apresenta aumento perceptível de memória, evidenciando que as melhorias aplicadas atuam predominantemente na redução de comparações e não na criação de estruturas adicionais.

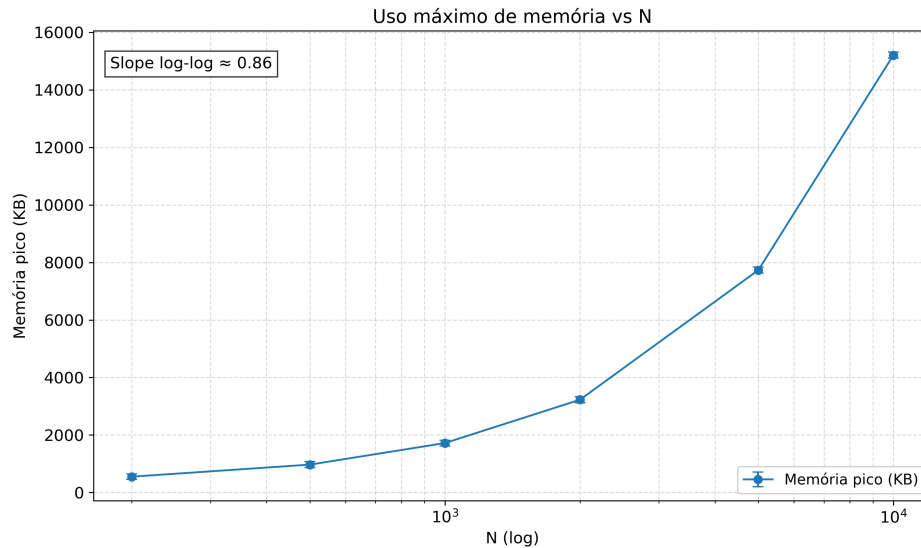


Figura 4: Pico de memória (KB) observado durante a execução.

6.5 Tamanho médio da carona vs. λ

O parâmetro λ determina a flexibilidade permitida para o desvio da rota ao incluir novos passageiros. Este gráfico evidencia o impacto direto desse parâmetro na formação das caronas: valores reduzidos de λ permitem maior liberdade para desvio e, portanto, resultam em caronas com mais passageiros; valores elevados tornam o sistema mais restritivo, diminuindo o tamanho médio das caronas. A análise desta curva permite identificar intervalos de λ que proporcionam melhor equilíbrio entre eficiência operacional (maior ocupação dos veículos) e conforto ou qualidade do atendimento.

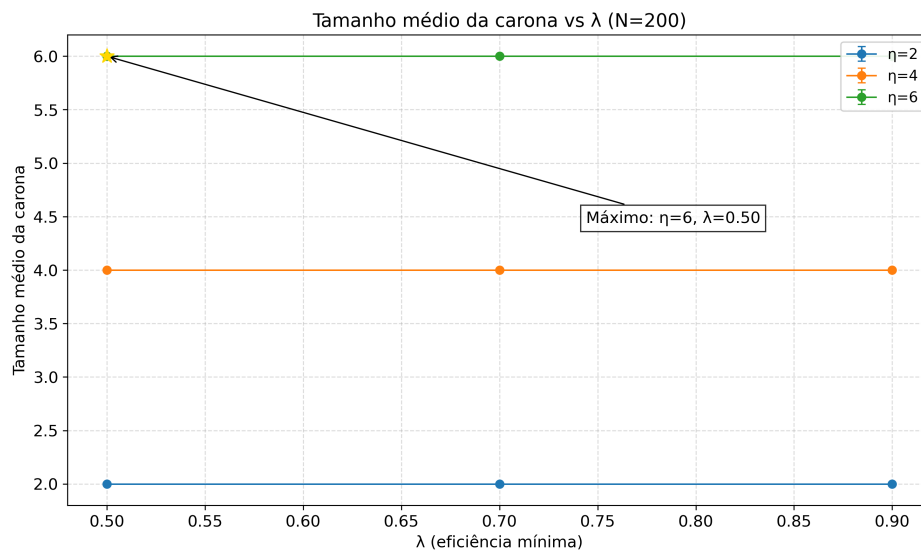


Figura 5: Tamanho médio das caronas (por demanda) para diferentes valores de λ .

6.6 Taxa de aceitação vs. λ

A taxa de aceitação representa a fração de combinações avaliadas que foram consideradas válidas. Como esperado, essa taxa diminui à medida que λ aumenta, uma vez que as condições de compatibilidade se tornam mais restritas. Quedas acentuadas revelam pontos críticos em que pequenas mudanças no parâmetro reduzem significativamente a viabilidade de combinações. Junto com o gráfico anterior, esta análise auxilia na escolha de valores de λ que mantêm boa formação de caronas sem comprometer excessivamente a qualidade das rotas.

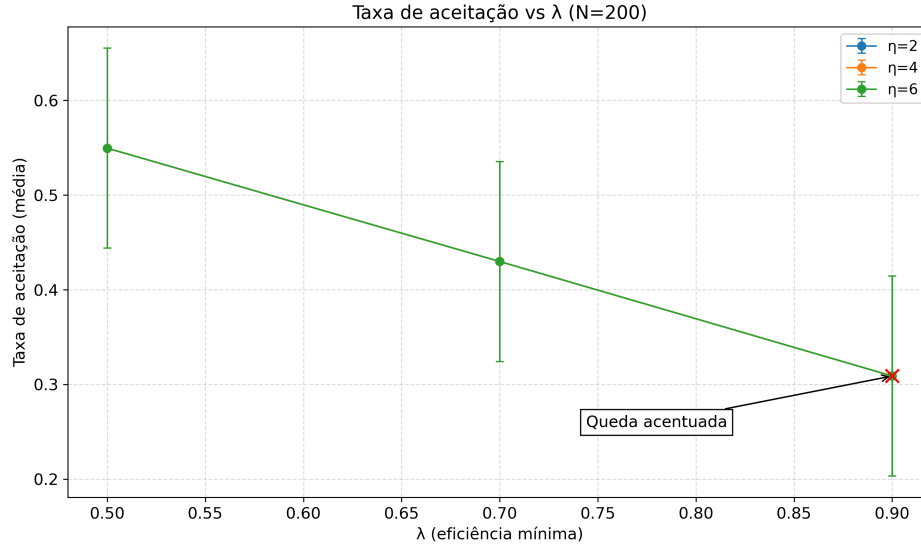


Figura 6: Fração média de combinações aceitas em função de λ .

6.7 Comparação: Baseline vs. versão otimizada

Por fim, este gráfico compara diretamente o desempenho das duas versões do algoritmo. A curva correspondente à versão otimizada posiciona-se consistentemente abaixo da versão original, evidenciando ganhos reais de tempo de execução. Esses ganhos tornam-se especialmente expressivos para valores elevados de N , onde a versão original apresenta maior sensibilidade ao número de verificações. O resultado demonstra que a estratégia otimizada reduz trabalho redundante e evita descartes prematuros de candidatos, produzindo execução mais eficiente sem aumento de memória e mantendo a qualidade das caronas.

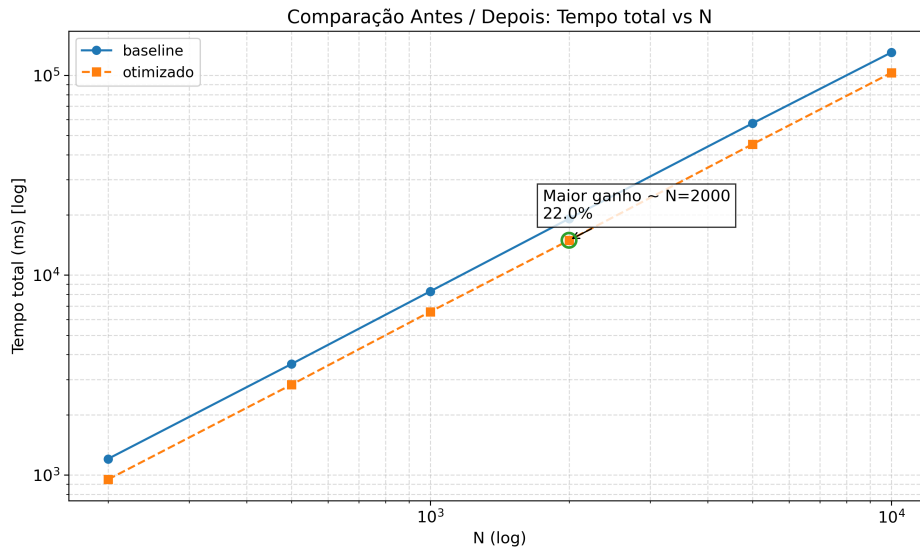


Figura 7: Tempo total médio: comparação entre versão original e otimizada.

7 Conclusão

Este trabalho implementou um sistema completo de despacho de corridas para a CabeAí, capaz de sugerir caronas com base em critérios temporais, espaciais e de eficiência. A organização em TADs próprios tornou o código modular e permitiu controlar claramente as estruturas envolvidas.

A análise teórica, fundamentada em (Cormen et al., 2022), previu que o escalonador concentraria o maior custo computacional, o que foi confirmado experimentalmente. Os gráficos de tempo de execução e de verificações de distância mostraram crescimento associado ao comportamento combinatório do problema, enquanto o uso de memória permaneceu praticamente linear.

As métricas envolvendo o parâmetro λ evidenciaram seu papel direto na flexibilidade das rotas e na taxa de combinações aceitas, revelando o impacto desse ajuste sobre a qualidade e eficiência das caronas.

A comparação entre a versão original e a otimizada demonstrou ganhos claros de desempenho sem aumento de memória ou perda de qualidade. Em síntese, o sistema cumpriu seus objetivos, oferecendo uma solução funcional e eficiente, cuja análise experimental reforça a importância de heurísticas bem projetadas para problemas de natureza combinatória.

8 Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.