Trabalho Prático 3 - Consultas ao Sistema Logístico

João Henrique Alves Martins

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

jalvesmartins16@ufmg.br

1 Introdução

A gestão eficiente de informações é um pilar fundamental para o sucesso de operações logísticas complexas. A capacidade de consultar rapidamente o histórico e o estado atual de objetos em uma base de dados é uma ferramenta importantíssima para a otimização de processos em várias esferas, contribuindo para a diminuição do tempo necessário para realizar essas tarefas.

O presente trabalho surge como uma evolução do sistema logístico desenvolvido para os Armazéns Hanoi. "Satisfeitos com o simulador de escalonamento", a empresa agora demanda a implementação de um sistema de consultas robusto e eficiente. O objetivo do trabalho é processar um fluxo contínuo de eventos logísticos e, em meio a eles, responder a duas categorias principais de consulta em tempo real: o histórico completo de um pacote específico (PC) e um resumo dos pacotes associados a um determinado cliente (CL), seja como remetente ou destinatário.

O desafio central reside na estruturação dos dados de forma a permitir que essas consultas sejam executadas com baixa latência, mesmo com um volume crescente de eventos. Para isso, o sistema deve ser construído sobre uma base de eventos processados cronologicamente a partir de um único arquivo de entrada, utilizando índices otimizados para acelerar o acesso às informações de pacotes e clientes, garantindo que as respostas sejam geradas de forma eficiente e precisa no momento em que a consulta é recebida.

Esta documentação tem como objetivo esclarecer todas as decisões de projeto e características do sistema de consultas implementado. Além desta introdução, o documento detalha as decisões de implementação, com foco nas estruturas de dados escolhidas (Seção 2), realiza uma análise de complexidade dos algoritmos (Seção 3), descreve as estratégias de robustez utilizadas no desenvolvimento (Seção 4), apresenta uma análise experimental do desempenho (Seção 5) e, por fim, expõe as conclusões do trabalho (Seção 6), seguida da bibliografia (Seção 7).

2 Método

O programa foi implementado na linguagem C++ e compilado pelo compilador G++ (GNU Compiler Collection). Essa linguagem foi escolhida, ao invés do C, devido às características do problema. Como o programa é robusto, a implementação de classes é adequada. Foram usados o Sistema Operacional Linux (Ubuntu) e o processador o 11th Gen Intel Core i5-1135G7×8. A máquina possui 8GB de RAM.

O código fonte foi modularizado da seguinte forma (cpp e hpp para a maioria): main, Event, Client, Package, System, List, HashMap.

2.1 Estruturas

As estruturas implementadas e sua função dentro do sistema são detalhadas abaixo.

- **List :** O TAD List é uma lista encadeada convencional. Como no sistema precisamos de listas que armazenam diferentes dados, ela foi implementada usando templates. Como ela é encadeada, facilita a insersão e remoção no início da lista, que são funções frequentemente utilizadas no sistema.
- **HashMap :** O TAD HashMap é um HashMap convencional, implementado com o uso de templates. Optei por essa implementação pois foi necessário implementar 2 índices diferentes, um para pacotes e o outro para clientes. O HashMap utiliza a função std::hash padrão, e armazena qualquer tipo de objeto. Optei pelo uso do encadeamento separado para a resolução de colisões, com uma taxa de 0.75 para o LOAD-FACTOR de rehash. Dessa forma, foi possível otimizar os tempos de busca. Escolhi o HashMap ao invés de outras estruturas de dados devido ao seu tempo *O(1)* de busca e implementação mais simples. O contraponto dessa escolha é o uso maior de memória.
- **Package :** O TAD Package representa os pacotes em uma classe simples, que armazena o seu id (Usado como chave no HashMap) e os índices correspondentes aos seus eventos no array de eventos, armazenados em uma lista de inteiros. Essa implementação é o primeiro passo para a nossa otimização de pesquisa, visto que o pacote contém esses índices necessários para a sua consulta como atributo.
- **Events :** O TAD Events é uma classe que representa os eventos que ocorrem na nossa pesquisa e alimentam o nosso "banco de dados". Os eventos podem ser de seis tipos diferentes, e cada evento armazena todas as informações necessárias para a sua pesquisa e impressão. O evento contém o seu tempo, o tipo de evento, o id do pacote correspondente ao evento, remetente e destinatário, armazéns de origem e destino. Optei por armazenar os eventos em um vetor estático, simplificando a pesquisa e eliminando o uso de ponteiros para os eventos.
- **Client :** O TAD Client é a classe que representa os clientes. Nossa classe é simples, e armazena o nome do cliente (Usada como chave para pesquisa no HashMap) e uma lista de ponteiros para os pacotes que contém essse cliente como remetente ou destinatário. Esse é o segundo elemento que simplifica a pesquisa, pois ao armazenar os ponteiros para esses pacotes, precisamos armazenar os índices apenas no pacote e acessá-los através desse ponteiro, eliminando a duplicação de dados.
- **System :** O TAD System é a peça final do sistema. Essa estrutura é responsável por todo o mecanismo de pesquisa e armazenamento dos dados. Ele contém os dois índices (HashMap de clientes e de pacotes), além do array de eventos. Essa classe é responsável por processar e armazenar todos os eventos no array, além de fazer as impressões das pesquisas. Como ele processa os eventos, é responsável por atualizar os índices quando um novo evento é adicionado.

2.2 Fluxo do programa

Após a apresentação das estruturas de dados, segue a descrição do fluxo de funcionamento do programa e as principais funções do sistema.

Após a abertura do arquivo de entrada, o nosso System é instanciado e a entrada de eventos começa. Optei por fazer as consultas de forma síncrona com a entrada dos eventos, simplificando e otimizando nosso processo de pesquisa.

Entramos no loop principal do programa, que faz a leitura dos eventos. Enquanto houverem eventtimes para serem lidos, o nosso loop faz a leitura adaptável dos eventos, dependendo das tags de consulta. Em um switch case, o loop alterna entre EV: Registro de evento, CL: Consulta de cliente, PC: Consulta de pacotes.

Para os eventos, caso o evento seja de resgistro (RG), o sistema cria um evento novo de forma estática e o aloca na primeira posição disponível no vetor de eventos. Após isso, ele atualiza os índices

com a função **processRG()**, que cria um pacote novo e insere ele no HashMap de pacotes, através da função **insert()**. Além disso, essa função atualiza ou cria um novo cliente para o destinatário e remetente no Hash de clientes. A função verifica se o cliente já existe e, caso já exista ela adiciona o ponteiro do novo pacote na lista de pacotes daquele cliente. Caso não exista, cria o cliente e também adiciona o novo pacote na lista.

Para outros eventos, como o pacote ja foi criado e o cliente já esta apontando para o pacote, precisamos apenas atualizar a lista de eventos do pacote. Portanto, chamamos a função **processOtherPackEvent()** e adicionamos o índice do novo evento no respectivo pacote.

Já para as consultas PC, que retornam o histórico do pacote, fazemos a leitura do id e chamamos a função **getValue()** do Hash de pacotes, que retorna o ponteiro para o endereço do nosso pacote. Depois, apenas chamamos a função **printEvent()** para cada índice do nosso vetor que esta presente na lista do respectivo pacote. Importante ressaltar que a função é adaptável para cada tipo de evento, que possui características diferentes.

Por fim, para as consultas CL, que retornam o primeiro e último evento de todos os pacotes de um cliente, chamamos a **getValue()** com o nome do cliente, que retorna o ponteiro para o endereço do cliente, e por fim chamamos a função **getClientEvents()**, que retorna uma lista com todos os índices necessários de forma ordenada. Essa função acessa cada endereço de pacote na lista original do cliente e, para cada um, insere de forma ordenada, através da função **insertSorted()**, o primeiro e último índice da lista de índices do pacote. Foi aplicada uma otimização para evitar os piores casos dessa inserção. Primeiro inserimos os maiores eventos e depois os de registro, que já estão ordenados e portanto são adicionados em *O(1)*. Por fim, com a lista de índices dos eventos do cliente pronta, chamamos a função **printEvent()** para cada índice. No fim das entradas válidas, o programa sai do loop while e se encerra.

2.3 Instruções de compilação

Após extrair o conteúdo do .zip e adicionar o arquivo txt com a entrada, entre no diretório extraído e execute o comando: make run FILE=nome-arquivo-entrada.txt. Esse comando compilaŕa e fará a ligação de todos os arquivos, além de executar o programa com o arquivo de entrada como parâmetro de entrada.

3 Análise de Complexidade

A análise de complexidade está divida baseada nas classes que contém a função. A análise foi feita em cima das funções mais importantes do sistema, ignorando as funções auxiliares *O(1)*, como getters e setters.

3.1 Lista

- **pushFront() e pushBack() :** A complexidade para adicionar um item, tanto no início, quanto no final da fila é *O(1)*, visto que temos ponteiros head e tail. Em relação a espaço, é *O(1)*, pois não usa espaço extra além do novo nó.
- **popFront() e popBack() :** Assim como adicionar, complexidade para remover um item, tanto no início, quanto no final da fila é *O(1)*, visto que temos ponteiros head e tail. Em relação a espaço, é *O(1)*, pois não usa espaço extra.
- **front() e back() :** As funções que acessam os itens do inínio e fim da lista, também são *O(1)* em tempo e espaço, devido aos motivos supracitados.
- **insertSorted() :** A complexidade para adicionar um item de forma ordenada em uma lista tem pior caso em *O(n)* e melhor caso em *O(1)*. É *O(1)* para espaço.

3.2 HashMap

- **insert() :** Para inserir um item no nosso Hash, como otimizamos a colizão e o rehash, em média gastamos *O(1)* de espaço e tempo. No pior caso, temos uma inserção onde o bucket já esta próximo da lotação máxima e é necessário percorrer todos os nós do bucket externo em *O(n)*.
- **getValue():** Novamente, no melhor caso e no caso médio é *O(1)* para tempo e espaço, pois acessa diretamente o bucket desejado através da função hash. No pior caso é *O(n)* para tempo.
- **contains() :** No melhor caso e no caso médio é *O(1)* para tempo e espaço, pois acessa diretamente o bucket desejado através da função hash. No pior caso é *O(n)* para tempo.
- **rehash() :** Esta é a função que mais consome espaço extra. Ela aloca uma new-table de tamanho 2k, sendo k a capacidade estabelecida anteriormente, enquanto a old-table (de tamanho k) ainda existe na memória. O espaço auxiliar é, portanto, proporcional à nova capacidade e a complexidade é *O(2k + n)*. Já para tempo é *O(n)*, pois insere novamente todos os elementos do Hash antigo.
- **getBucketIndex() :** *O(1)* para tempo e espaço. Apenas retorna o módulo da chave pelo tamanho da tabela.
- **clearTable() e deleteAllValues() :** *O(1)* para espaço. Percorre todos os elementos da tabela, portanto é *O(k + n)*.

3.3 Package

Todas as funções do TAD que representa o pacote são constantes em tempo e em espaço. As funções implementadas são apenas **getters** e **setters** para os atributos do pacote. A única especial é a **addEvent()**, que é*O(1)* para espaço e tempo, pois adiciona um índice no fim da lista de eventos.

3.4 Client

- **getClientEvents() :** Para montar a lista de eventos de um cliente precisamos inserir n elementos de forma ordenada em uma lista. Portanto no pior caso, a função é $O(n^2)$ de tempo e *O(n)* de espaço. Porém, como fizemos uma otimização para evitar esse pior caso, essa complexidade é evitada.

  Além dessa, as funções do TAD que representa o cliente são constantes em tempo e em espaço. As funções implementadas são apenas **getters** e **setters** para os atributos da classe.

3.5 Event

O TAD Events possúi apenas **getters** e **setters** para seus atributos, portanto é constante em todas as suas funções.

3.6 System

- **processRG() :** Para processar um evento de registro, em média usamos *O(1)* de tempo e espaço, pois fazemos um número de operações fixos nos HashMaps. Porém, caso a insersão seja degradada ao longo do tempo, podemos ter *O(c + p)* em tempo, sendo c e p o número de clientes e pacotes no seu respectivo Hash.
- **processOtherPackEvent() :** Para processar um evento de outro tipo, apenas adicionamos o novo índice a lista de índices do pacote. Portanto, a função é *O(1)* para tempo e espaço.
- **printEvent() :** Para imprimir um evento, apenas fazemos algumas verificações constantes para imprimir da forma correta. Portanto, *O(1)* em tempo e espaço.

3.7 Main

Com todas as complexidades computadas, podemos analisar a complexidade do nosso sistema como um todo. Em relação a espaço, nosso sistema aloca dois HashMaps em *O(k + n)*, com k sofrendo alterações ao longo das insersões, e um vetor de tamanho 200 para os eventos. Portanto gastamos aproximadamente *O(2(k + n))*, com k variável.

Já para o tempo, precisamos somar as complexidades das principais funções. Em geral podemos afirmar que nosso sitema é bem performático, pois as consultas nos HashMaps são extremamente eficientes e otimizam nossas pesquisas para as consultas. Em geral: as insersões de eventos vão ser feitas em *O(1)*, podendo chegar em *O(n)* para os piores casos; As consultas são feitas em *O(1)* em média, e apenas para montar a lista de eventos do cliente podemos chegar em $O(n^2)$, sendo n o número de pacotes daquele cliente. As impressões são feitas em *O(1)*. Diante disso, juntamente a análise experimental, é possível avaliar muito positivamente nosso sistema de consulta.

4 Estratégias de Robustez

O sistema contém diversos mecanismos de segurança, contra falhas e entradas equivocadas, baseados nos conceitos da programação defensiva. Os principais gatilhos são: Verificação de existência do arquivo de entrada; Verificação do sucesso da abertura do arquivo; Proteção de retornos do tipo nullptr nos HashMaps; Casos default nos switch-cases, protegendo contra casos atípicos; Liberação da memória de todos os vetores dinamicamente alocados, verificada através do uso do valgrind.

Todas essas verificações, em caso de falha, retornam 1 (Código que indica falha na execução do programa) ou retornam nulo. A implementação desses gatilhos é essencial para a identificação de possíveis falhas e para a proteção do sistema.

5 Análise Experimental

A análise experimental foi feita com a obtenção de dados utilizando a biblioteca "chrono"em C++.

As diferentes entradas foram geradas com scripts em python, e os resultados foram armazenados em um .csv. Os resultados foram plotados utilizando a biblioteca matplot.lib. Para todos os exemplos, a variável em análise variou entre 50 e 2000 unidades.

5.1 Número de pacotes x Tempo de execução

No gráfico abaixo, é apresentada a relação entre o número de pacotes e o tempo de execução do algoritmo, com o número de eventos e clientes se mantendo constante. O resultado da análise é o melhor possível, visto que na medida que aumentamos o número de pacotes, o tempo de execução se mantém constante.

A relação constante entre tempo de execução e pacotes do sistema é uma das vias de demonstração da eficiência do nosso sistema de pesquisa, que lida muito bem com quantidades maiores de pacotes.

Nesse sentido, fica nítida o upside de escolher o HashMap com encadeamento separado. Essa estrutura de dados permite uma consulta extremamente rápida, em tempo constante. O downside seria o uso de memória, que deve ser elevado para permitir a velocidade das consultas. Essa relação memória x tempo de computação foi uma das minhas decisões de implementação no projeto. Optei por priorizar a velocidade das pesquisas, visto que é o ponto central do trabalho.

É importante ressaltar que alguns outliers nos nossos dados são esperados, devido à configurações da máquina ou situações específicas de execução.

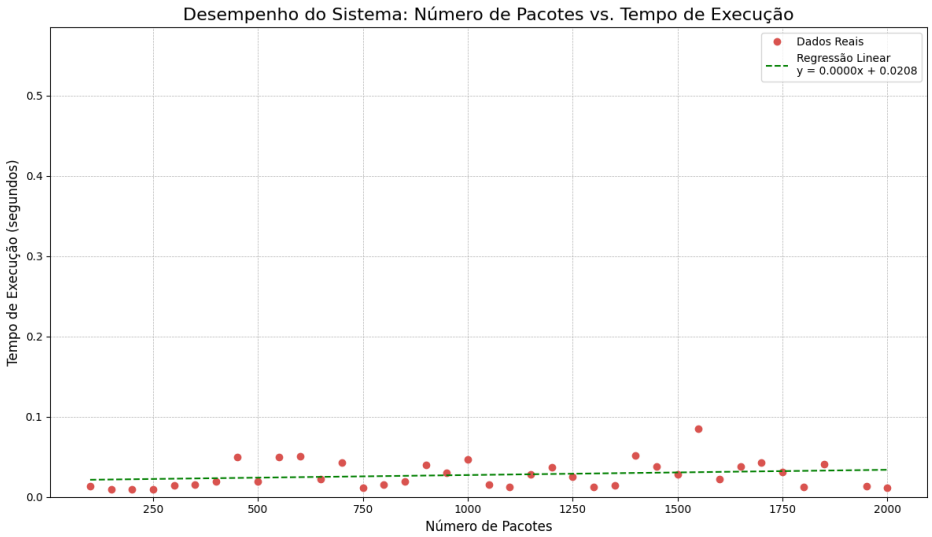

Figura 1: Quantidade de pacotes x Tempo de execução, com regressão linear.

5.2 Número de clientes x Tempo de execução

No gráfico abaixo, é apresentada a relação entre o número de clientes e o tempo de execução do algoritmo, com número de pacotes e eventos constantes. A curva de crescimento da função mostra que o tempo de execução, semelhantemente à variação dos pacotes, é constante conforme o número de clientes aumenta.

A relação constante entre tempo de execução e armazéns do sistema é mais uma demonstração da eficiência do nosso sistema de pesquisa, que lida muito bem com muitos clientes.

Essa relação é meio obvia, visto que os clientes também são armazenados em um HashMap. Portanto, a análise anterior também serve para essa seção.

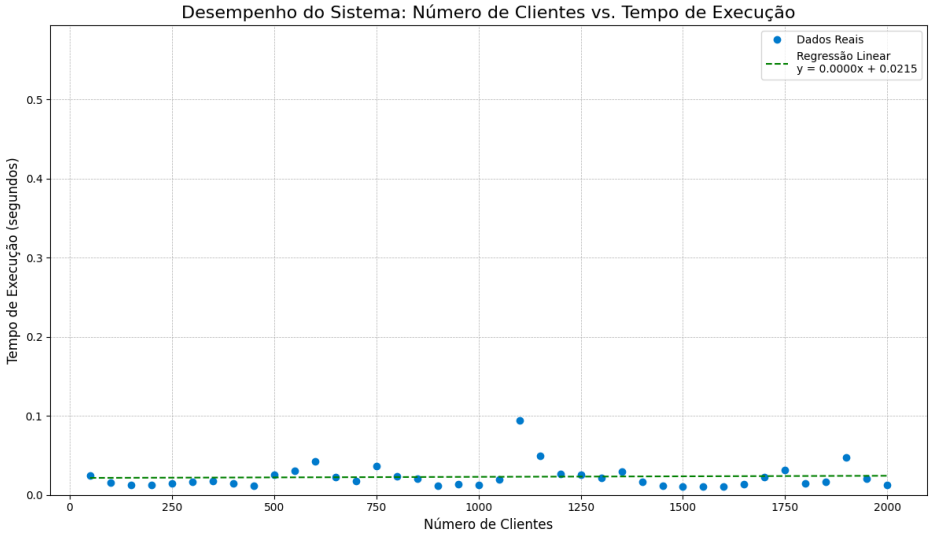

Figura 2: Quantidade de clientes x Tempo de execução, com regressão linear.

5.3 Quantidade de eventos x Tempo de execução

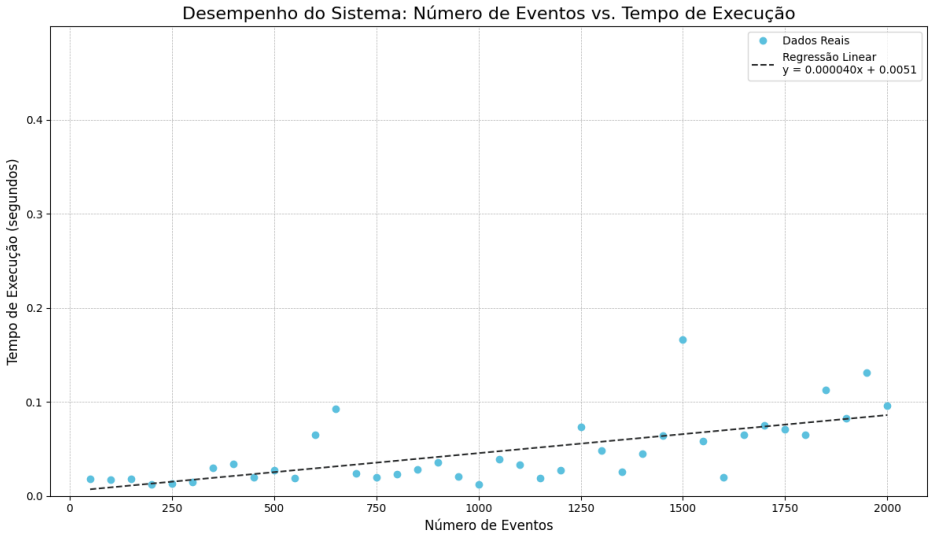

Figura 3: Quantidade de eventos x Tempo de execução.

Por fim, no gráfico acima, temos o experimento relacionado à curva de complexidade em relação ao número de eventos e o tempo de execução.

Como esperado, devido à análise de complexidade, com o aumento de eventos, temos uma curva de crescimento linear em relação ao tempo.

O número de eventos não é algo encapsulado dentro de uma estrutura de otimização, como o HashMap, portanto, esse aumento é esperado devido as operações de processamento dos eventos.

Esse resultado, junto aos outros 2, comprovam nossa teoria inicial, baseada na análise de complexidade. O nosso sistema de pesquisa é mais afetado pelo número de eventos que ele tem que realizar, e não pelo número de pacotes ou clientes que compõem sua topologia. Esse comportamento comprova a eficiência das estruturas de dados escolhidas.

6 Conclusões

Nesse trabalho, utilizando os conhecimentos obtidos nas aulas de Estruturas de Dados, foi possível implementar e analisar um sistema de pesquisa de eventos de transporte, visando otimizar a busca desses eventos.

Os resultados da análise experimental mostram a importância de sistemas que otimizem a pesquisa de objetos armazenados em uma base de dados, assim como o implementado nesse trabalho. Para bases muito extensas, é inviável utilizar sistemas de pesquisa não-otimizados.

Nesse sentido, a eficácia do sistema implementado foi provada nas análises 5.1 e 5.2, mostrando que se tivermos um HashMap de tamanho suficiente, nosso sistema será capaz de lidar com essas consultas de forma constante, mesmo com muitos pacotes ou clientes.

Em conclusão, a realização do trabalho evidenciou mais uma vez o impacto das decisões de implementação e do uso de diferentes estruturas de dados no resultado final do projeto, especialmente tento a liberdade para escolher entre estruturas diferentes. Cada escolha impacta diretamente o funcionamento do produto final.

Esse TP, apesar de ser mais simples que os últimos dois, também trouxe suas dificuldades. Escolher entre o HashMap e outras estruturas que otimizam pesquisas, como a Árove AVL, foi um dos principais desafios enfrentados. No fim, a análise experimental mostrou que fiz uma boa escolha.

Seguindo o aprendizado com o TP1 e TP2, utilizei boas práticas de programação e usei o GitHub para versionamento do código. Esse processo me ajudou muito na depuração e construção do projeto. Também, fiquei mais satisfeito com o padrão de código que utilizei, usando novamente variáveis em inglês e formatadas em snake-case.

Como último trabalho, fico muito satisfeito com o resultado que obtive e com todo o caminho que traçei ao longo da disciplina de Estruturas de Dados. Todos os trabalhos contribuiram de forma significativa para o meu aprendizado.

7 Referências

Meira, W. and Lacerda, A. (2025). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Meira, W. (2025). Instruções para a realização do Trabalho Prático 3 de Estrutura de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.