Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Trabalho Prático 3
Consultas ao Sistema Lógico

João Pedro Moreira Smolinski
2024023996
jp.smolinski05@gmail.com

Conteúdo

1 Introdução

Este documento detalha a implementação e a análise do sistema de "Consultas ao Sistema Lógico", desenvolvido como parte do Trabalho Prático 3 da disciplina de Estruturas de Dados (DCC205-2025/1). O projeto visa criar um sistema capaz de processar um fluxo constante e contínuo de eventos logísticos e responder em tempo real a diferentes consultas. Tais consultas podem ser o histórico completo de um pacote em específico ou o estado atual de todos os pacotes associados a um determinado cliente.

O principal desafio do trabalho reside na necessidade de processar um arquivo único contendo tanto eventos quanto consultas, ordenados cronologicamente, e gerar respostas eficientes sem a possibilidade de ler a entrada múltiplas vezes. Para isso, a solução precisa construir e manter índices dinâmicos que permitam acesso rápido às informações de pacotes e clientes, garantindo que as consultas sejam respondidas com base no estado do sistema no exato momento em que são feitas.

A solução proposta é um sistema em C++ que processa o fluxo de dados em uma única passagem. Para otimizar as buscas, foram implementadas árvores AVL como estruturas de indexação, que mapeiam identificadores de pacotes e nomes de clientes aos seus respectivos dados. Os eventos são armazenados em um arranjo (Array) dinâmico que serve como a fonte central de informações. Este relatório abordará em detalhes as estruturas de dados criadas, a metodologia de implementação, a análise de complexidade computacional e as estratégias de robustez adotadas.

2 Método

Para a implementação, optou-se pelo uso da linguagem C++. A fácil modularização proposta pela linguagem, bem como a abstração e o encapsulamento proporcionados pela programação orientada a objetos, foram fatores que colaboraram com essa escolha. Além disso, a linguagem possui um sistema de gerenciamento de erros lógicos e exceções que foi muito bem aproveitado durante o desenvolvimento do projeto.

Em adição, vale notar que o projeto foi feito usando o VS Code como IDE. Seu compilador é o GNU Compiler Collection (G++) versão gcc 6.3.0. O código foi testado em um ambiente com:

- **Processador -** Intel(R) Core(TM) i3-10100F CPU @ 3.60GHz.
- **Memória -** 16.0 GB.
- **Sistema Operacional -** Windows 10 22H2 e Ubuntu 24.04.1

2.1 Instruções

O projeto e seus arquivos foram organizados em pastas de forma que o arquivo Makefile consiga executar todas as compilações e builds necessárias para a execução.

Caso seja ambiente Linux ou semelhante, deve-se usar o comando *make all* na pasta raiz do projeto. Em seguida, executar *./bin/tp2.out arquivo_de_entrada* para rodar o programa, substituindo arquivo_de_entrada pelo nome do arquivo a ser usado. Se não for passado nenhum parâmetro adicional, a entrada deve ser pelo terminal.

Caso seja um ambiente Windows, deve-se usar o comando *make all_windows* na pasta raiz do projeto. Então executar *./bin/tp2.exe arquivo_de_entrada*, substituindo arquivo_de_entrada pelo arquivo de entrada. De forma semelhante ao ambiente Linux, pode ser executado sem passar nenhum parâmetro adicional, assim a entrada passa a ser o terminal.

2.2 Fluxo de Execução

O sistema é projetado para ler um fluxo de dados da entrada padrão selecionada. Cada linha é processada sequencialmente e é separada em três tipos:

- Evento de Pacote,
- Consulta por Pacote,
- Consulta por Cliente.

Para cada tipo diferente de linha, é chamado um método dedicado a processá-la da melhor forma.

2.2.1 Processamento de Evento

Quando uma linha de evento é lida, logo em seguida o tipo de evento e identificador do pacote associado são lidos a partir dela. Após isso, de acordo com o tipo de evento, o restante dos parâmetros vai ser lido. Essa lógica segue a proposta de que eventos de registro informam remetente, destinatário, origem e destino; eventos de armazenamento, remoção e rearmazenamento informam destino e seção em que o pacote é armazenado; eventos de transporte informam origem e destino do transporte e eventos de chegada informam somente o destino. Nesse

momento, o evento é criado e adicionado no fim do arranjo dinâmico, preservando a ordem cronológica crescente do mesmo.

Após isso, é necessário atualizar o índice dos pacotes. Como citado na introdução, são usadas duas árvores AVL como estruturas de indexação. Em uma delas, a chave de acesso aos valores é o identificador do pacote, e o valor associado é um arranjo dinâmico contendo o índice de todos os eventos daquele pacote no arranjo central. A atualização é feita adicionando o índice do evento atual ao fim do arranjo daquele pacote.

Por fim, caso o evento seja de registro, uma lógica semelhante é feita nos índices dos clientes. A segunda árvore AVL usa o nome do cliente como chave de acesso e o valor associado é um arranjo dinâmico com o número identificador de todos os pacotes relacionados a aquele cliente. A atualização é feita adicionando o identificador do pacote registrado ao fim do arranjo daquele cliente.

2.2.2 Processamento de Consultas por Pacote

Após a leitura de uma linha de consulta por pacote, o identificador do pacote a ser consultado também é lido. Então, na árvore AVL dos pacotes, o arranjo relacionado ao identificador da entrada é pego para ser impresso. O arranjo é percorrido de forma linear até que termine ou o tempo relacionado ao evento seja após o tempo da consulta requisitada. Finalmente, na saída é possível rever a consulta feita, em seguida o número de eventos associados a tal consulta e, por fim, cada evento relacionado ao pacote, um em cada linha e em ordem cronológica.

2.2.3 Processamento de Consultar por Cliente

Ao ler uma linha de consulta por cliente, lê-se também o nome do cliente. Após isso, retorna o arranjo de identificadores de pacotes relacionado a esse cliente, por meio de um acesso à árvore AVL dos clientes. Para cada pacote, seu evento de registro e evento mais recente são buscados de forma similar a uma consulta por pacote, mas ignorando os eventos intermediários. Então todos esses eventos são reordenados a fim de manter a ordem cronológica. Após a ordenação, a saída esperada contém a consulta feita, o número de linhas que conterão na saída e os eventos de registro e mais recentes de cada pacote relacionado ao cliente, também um por linha e em ordem cronológica.

2.3 Estruturas de Dados

Para a execução do projeto, foram pensados alguns TADs que facilitassem a implementação. Foram criadas diversas classes com interfaces próprias que suprissem as necessidades do problema:

- **LogicSystem**: Classe central que orquestra todo o processamento. É responsável por ler a entrada, direcionar cada linha para o método de processamento correto e gerenciar as estruturas de dados principais.
- **Event**: Representa um único evento logístico. Armazena todos os seus atributos, como o tempo em que é feito, tipo, identificador de pacote associado, remetente, destinatário, origem, destino e seção de armazenamento. Possui métodos de comparação, com base no tempo e identificador, que possibilita uma ordenação linear.
- **DynamicArray**: Implementação de um vetor dinâmico. É usado como a estrutura de armazenamento principal para todos os objetos Event e também para as listas de índices e identificadores de pacotes dentro dos nós da AVL.
- **AVL**: Implementação de uma Árvore AVL genérica. É a estrutura de dados principal para os índices do sistema, garantindo buscas, inserções e atualizações com baixa complexidade. Como dito anteriormente foram instanciadas duas árvores, uma para pacotes e outra para clientes.
- **Node**: Classe que representa um nó na árvore AVL, armazenando a chave, o valor e os ponteiros para os filhos.

Além das estruturas acima citadas, foi criado um enumerador auxiliar:

- **EventType-** Usado para definir o tipo do evento. Composto por RG - evento de registro de pacote; AR - evento de armazenamento de pacote em uma certa seção de um certo armazém; RM - evento de remoção de pacote de uma determinada seção; UR - evento de rearmazenamento de pacote (não foi possível transportar); TR - evento de transporte de pacote até outro armazém; EN - evento de entrega de pacote em seu destino e NOT_VALID - tipo "nulo" de evento (usado para eventos não inicializados completamente).

2.4 Funções

As funções e métodos de classes foram desenvolvidos visando à simplificação da leitura e do fluxo de execução:

- **LogicSystem**
  - **LogicSystem-** Inicializa a classe do sistema lógico e gerenciamento.

  - ~**LogicSystem-** Destrutor padrão do sistema.
  - **Run-** Pode receber um arquivo de entrada ou não, caso não receba usa o terminal como entrada. Inicia o fluxo de eventos e encerra assim que todas as linhas forem lidas
  - **ProccessLine-** Recebe uma linha da entrada, pega o tempo e tipo de linha e chama a função que lida com o tipo específico.
  - **ProccessEvent-** Recebe uma linha de evento, pega os atributos restantes da entrada, cria e armazena o evento e atualiza os valores necessários das árvores AVL's.
  - **ProccessPackage-** Recebe uma linha de consulta por identificador de pacote, pega o histórico de eventos relacionados ao pacote e imprime conforme esperado pela saída.
  - **ProccessClient-** Recebe uma linha de consulta por cliente e pega os pacotes relacionados ao cliente. Então, em ordem cronológica, para cada pacote imprime seu evento de registro e evento mais recente.

- **Event**
  - **Event-** Constrói o evento com base nos parâmetros passados, tempo, tipo e identificador ou cria um evento vazio.
  - ~**Event-** Destrói o evento.
  - **Print-** Imprime o evento respeitando a formatação por tipo.
  - **GetTime-** Retorna o tempo do evento.
  - **GetType-** Retorna o tipo de evento.
  - **GetId-** Retorna o identificador do pacote do evento.
  - **SetOrigin-** Define a origem do pacote do evento.
  - **SetDestiny-** Define o destino do pacote do evento.
  - **SetDivision-** Define a seção do pacote do evento.
  - **GetSender-** Retorna o remetente do pacote do evento.
  - **SetSender-** Define o remetente do pacote do evento.
  - **GetReceiver-** Retorna o destinatário do pacote do evento.
  - **SetReceiver-** Define o destinatário do pacote do evento.
  - **StringToType-** Traduz uma String em um tipo de evento.
  - **operator=-** Sobrecarga do operador de atribuição, copia os valores de outro evento no atual.
  - **operator<-** Sobrecarga do operador "menor que", compara o tempo e identificador de dois eventos.
  - **operator>-** Sobrecarga do operador "maior que", compara o tempo e identificador de dois eventos.

- **DynamicArray**
  - **DynamicArray-** Constrói um arranjo dinâmico e inicia seus atributos.
  - ~**DynamicArray-** Destrói o arranjo dinâmico desalocando a memória.
  - **Push-** Insere um novo valor ao fim do arranjo, chamando a função de aumentar se necessário.
  - **GetSize-** Retorna o tamanho do arranjo.
  - **Sort-** Ordena o arranjo, chamando a orndenação por Quicksort.
  - **operator[]-** Sobrecarga do operador de acesso por índice, retorna o valor da posição recebida.
  - **Resize-** Aumenta o tamanho máximo do arranjo, realocando memória e transferindo os valores antigos.
  - **QuicksortR-** Implementação recursiva do algoritmo de ordenação Quicksort.
  - **Partition-** Algoritmo de particionamento do vetor com base em um pivô calculado com a mediana dos valores extremos e central.
  - **Swap-** Troca o valor de duas variáveis.
  - **Median-** Retorna a mediana de três números.

- **AVL**
  - **AVL-** Construtor padrão inicia a raiz como nula.
  - ~**AVL-** Destrutor padrão que chama o destrutor recursivo a partir da raiz.
  - **Insert-** Recebe chave e valor associado e chama a inserção recursiva a partir da raiz.
  - **Find-** Recebe chave e chama a busca recursiva a partir da raiz.

  - **RotateRight-** Rotaciona um nó para a direita, realocando seus nós filhos. Isso transforma o antigo nó filho à esquerda em nó pai.
  - **RotateLeft-** Rotaciona um nó para a esquerda, realocando seus nós filhos. Isso transforma o antigo nó filho à direita em nó pai.
  - **InsertR-** Insere de forma recursiva uma chave associada a um valor na árvore. Segue a lógica de que valores a esquerda são menores que o centro e valores a direita são maiores. Verifica o balanceamento da árvore e chama as rotações que forem necessárias.
  - **FindR-** Busca de forma recursiva uma chave na árvore. Se achar retorna o apontador para o nó, caso contrário retorna um apontador nulo
  - **DeleteR-** Deleção recursiva de uma sub-árvore, deleta a sub-árvore à esquerda, deleta a sub-árvore à direita e por fim deleta a si mesmo.

- **Node**
  - **Node-** Construtor que inicia o nó com chave e valor associado.
  - **∼Node-** Destrutor padrão do nó.
  - **GetHeight-** Retorna a altura do nó.
  - **UpdateHeight-** Atualiza a altura do nó.
  - **GetBalance-** Retorna ao balanceamento do nó.
  - **Max-** Retorna o maior de dois inteiros.
  - **GetRight-** Retorna o nó a direita.
  - **SetRight-** Define o nó a direita.
  - **GetLeft-** Retorna o nó a esquerda.
  - **SetLeft-** Define o nó a esquerda.
  - **GetValue-** Retorna o valor associado ao nó.
  - **SetValue-** Define o valor associado ao nó.
  - **GetKey-** Retorna a chave associada ao nó.

3 Análise de Complexidade

Para fins de sintetização, serão citadas nessa seção somente funções cuja complexidade de tempo ou espaço não são constantes ($O(1)$). Dessa forma o valor mínimo de custo é logarítmico e caso a função não seja citada abaixo, assuma que seu custo é $O(1)$. Além disso, serão usadas as variáveis:

- E**-** Número de eventos.
- P**-** Número de pacotes.
- C**-** Número de clientes.
- Q**-** Número de consultas.

Vale lembrar que o número de pacotes é no máximo o número de eventos, assim como o número de clientes é no máximo duas vezes o número de pacotes. Dessa forma, $P \in O(E)$ e $C \in O(P) \in O(E)$. Além disso a entrada tem tamanho $O(E + Q)$

3.1 Tempo

- **AVL::InsertR-** Adiciona um novo nó a partir de um nó. Cada sub-árvore AVL tem profundidade máxima $O(log(n))$, sendo n o tamanho relativo a sub-árvore, esse é o tamanho máximo da recursão e custo de tempo de inserção. Todas as possíveis rotações subsequentes tem custo $O(1)$.
- **AVL::Insert-** Chama a função AVL::InsertR a partir da raiz da árvore, custo $O(log(n))$ sendo n o tamanho total da árvore AVL.
- **AVL::FindR-** Procura uma chave específica em algum nó da AVL. Como a AVL tem profundidade $O(log(n))$ esse é o custo da operação (e tamanho máximo da pilha de recursão).
- **AVL::Find-** Chama a função AVL::FindR a partir da raiz da árvore. Custo $O(log(n))$ sendo n o tamanho da AVL.
- **AVL::DeleteR-** É chamada recursivamente até atingir um nó vazio. Como a AVL tem altura máxima $O(log(n))$ sendo n o número de nós, esse é o tamanho máximo de pilha. Porém, essa função deleta todos os nós de uma sub-árvore, dessa forma tendo complexidade $O(n)$.

- **AVL::∼AVL-** Chama AVL::DeleteR a partir da raiz. Custo $O(n)$ sendo n o tamanho da AVL.
- **DynamicArray::Resize-** Cria um novo arranjo de tamanho maior e transfere todos os valores do antigo para o novo. Custo $O(n)$ sendo n o tamanho do arranjo antigo.
- **DynamicArray::Push-** No pior caso deve chamar DynamicArray::Resize, tendo custo $O(n)$. Apesar disso tem custo amortizado baixo, aproximadamente $O(1)$, já que se chamar um DynamicArray::Resize com custo n, as próximas n chamadas vão ter custo $O(1)$ (considerando que o tamanho do arranjo dobra).
- **DynamicArray::Partition-** No pior caso, percorre o arranjo todo, custo $O(n)$.
- **DynamicArray::QuicksortR-** Usando a mediana de três valores, o algoritmo de Quicksort recursivo tem custo $O(n \times log(n))$.
- **DynamicArray::Sort-** Chama a função DynamicArray::QuicksortR, que custa $O(n \times log(n))$.
- **LogicSystem::ProccessEvent-** Chama DynamicArray::Push no arranjo de eventos, pior caso $O(E)$, custo amortizado $O(1)$. Então procura o identificador do pacote na AVL de pacotes, custo $O(log(P))$, adiciona o índice ao fim do arranjo (pior caso $O(E)$, custo amortizado $O(1)$) e insere na AVL, custo $O(log(P))$. Caso seja um evento de registro, para a AVL de clientes pega o remetente e o destinatário, adiciona o identificador do pacote e insere novamente na AVL. Busca e inserção tem custo $O(log(C))$ e adicionar o identificador tem pior caso $O(P)$ e custo amortizado $O(1)$. Dessa forma, o pior caso é aquele que aumenta todos os arranjos, totalizando $O(E + log(P) + log(C) + P) \in O(E)$. O custo amortizado é $O(log(P) + log(C))$.
- **LogicSystem::ProccessPackage-** Puxa da AVL de pacotes o arranjo de eventos, custo $O(log(P))$. Então itera sobre os eventos do pacote para a impressão. Custo $O(E_p)$, sendo E_p os eventos relacionados a um pacote específico. Vale lembrar que $E_p \in O(E)$, sendo E o número de eventos totais. Nesse caso a complexidade temporal da função seria $O(E_p + log(P))$, como $P \in O(E)$, o pior caso é $O(E)$.
- **LogicSystem::ProccessClient-** Puxa da AVL de clientes o arranjo de indices de pacotes, custo $O(log(C))$. Para cada pacote procura na AVL de pacotes os índices dos eventos, pegando o primeiro e último, custo $O(P_c \times log(P))$, sendo P_c o número de pacotes de dado cliente. Por fim, ordena o arranjo de eventos (que vai conter dois eventos por pacote), custo $O(P_c \times log(P_c))$, então imprime os eventos, custo $O(P_c)$. Vale lembrar que $P_c \in O(P)$, pois um cliente pode estar relacionado a todos os pacotes. Dessa forma, a complexidade total fica $O(log(C) + P \times log(P) + P \times log(P) + P) \in O(P \times log(P))$, considerando que $C \in O(P)$.
- **LogicSystem::ProccessLine-** Confere o tipo de linha e chama LogicSystem::ProccessEvent, ou LogicSystem::ProccessPackage ou LogicSystem::ProccessClient. Complexidade $O(E + E + P \times log(P))$, como $P \in O(E)$, a complexidade temporal total é $O(E \times log(E))$.
- **LogicSystem::Run-** Inicia o laço principal de leitura da entrada, para cada linha lida chama LogicSystem::ProccessLine. Considerando que a entrada tem N linhas, sua complexidade total é $O(N \times E \times log(E))$.
- **LogicSystem::∼LogicSystem-** Destrói o arranjo de eventos $O(1)$, a AVL de clientes e seus arranjos, $O(C)$ e por fim destrói a AVL de pacotes e seus arranjos, $O(P)$. Complexidade temporal total $O(C + P) \in O(P)$, já que $C \in O(P)$.

3.2 Espaço

- **AVL-** Todas as funções recursivas da classe AVL tem tamanho máximo de pilha $O(log(n))$, sendo n o número de nós. Isso acontece pois a AVL é uma árvore com alto grau de balanceamento, logo sua altura cresce em ordem logarítmica ao número de nós. As funções com essa complexidade espacial são **AVL::InsertR**, **AVL::Insert**, **AVL::FindR**, **AVL::Find**, **AVL::DeleteR** e **AVL::∼AVL**.
- **DynamicArray::Resize-** Cria um arranjo de tamanho proporcional ao tamanho atual (duas vezes). Dessa forma tem complexidade $O(n)$, sendo n o tamanho antigo do arranjo.
- **DynamicArray::Push-** Como pode chamar DynamicArray::Resize, tem pior caso espacial $O(n)$, mas é amortizado para aproximadamente $O(1)$, já que após uma chamada de custo n haverão n outras chamadas de custo $O(1)$.
- **DynamicArray::QuicksortR-** O tamanho de pilha máximo do algorítmo de Quicksort recursivo com mediana de três ainda é $O(n)$, com n sendo o tamanho do arranjo.
- **DynamicArray::Sort-** Como chama DynamicArray::QuicksortR, tem custo espacial $O(n)$, com n sendo o tamanho do arranjo.
- **LogicSystem::ProccessEvent-** Chama DynamicArray::Push para o arranjo de eventos, $O(E)$ no pior caso, $O(1)$ amortizado. Nas AVL's de pacote e cliente chama AVL::Find e posteriormente AVL::Insert, custo $O(log(P))$ e $O(log(C))$ para pacotes e clientes. Por fim, chama DynamicArray::Push para os arranjos dos clientes e pacote, custo $O(P)$ e $O(E)$ respectivamente, custo amortizado $O(1)$ para ambos. Dessa forma o custo espacial total de pior caso é $O(E + log(P) + log(C) + P + E) \in O(E)$, já que $C \in O(P)$ e $P \in O(E)$. Amortizado é $O(log(P) + log(C))$

- **LogicSystem::ProccessPackage-** Chama AVL::Find na AVL de pacotes, $O(log(P))$. A impressão dos eventos é feita diretamente, sem o uso de outro arranjo.
- **LogicSystem::ProccessClient-** Chama AVL::Find na AVL de cliente, $O(log(C))$. Para cada pacote encontrado chama AVL::Find na AVL de pacotes, $O(log(P))$. Além disso cria, usa e ordena um arranjo dinâmico dos eventos a serem impressos, custo $O(P_c)$, já que para cada pacote vai haver no máximo dois eventos (registro e mais recente), P_c sendo o número de pacotes de um certo cliente. $P_c \in O(P)$, pois um cliente pode ter relação com todos os pacotes e $C \in O(P)$ pelo motivo inverso, já que cada pacote pode estar atrelado a um cliente diferente. Dessa maneira, a complexidade total é $O(P)$.
- **LogicSystem::ProccessLine-** Chama LogicSystem::ProccessClient, ou LogicSystem::ProccessPackage ou LogicSystem::ProccessEvent. Complexidade espacial total $O(E + log(P) + P)$, como $P \in O(E)$, a complexidade fica $O(E)$.
- **LogicSystem::Run-** Chama LogicSystem::ProccessLine para cada linha lida. Considerando a execução total da entrada, ao fim haverá um arranjo dinâmico de tamanho $O(E)$, e duas AVL's, uma para clientes e outra para pacotes, com tamanhos $O(C)$ e $O(P)$ respectivamente. Somando o conteúdo de todos os arranjos dos nós de cada AVL, na AVL de clientes temos $O(P)$, já que cada pacote vai estar ligado a dois clientes e na AVL de pacotes temos $O(E)$ já que cada evento vai estar ligado a um pacote. Ou seja, na execução total do método temos complexidade espacial $O(E)$, já que $C \in O(P)$ e $P \in O(E)$.

3.3 Total

Dessa forma, a complexidade de tempo e espaço total de execução do programa tem a seguinte análise. Considerando funções em série e em paralelo, a *main* vai chamar primeiramente LogicSystem::LogicSystem, então LogicSystem::Run e por fim LogicSystem::∼LogicSystem. Ou seja, tem complexidade total temporal $O(N \times E \times log(E) + P) \in O(N \times E \times log(E))$ e complexidade espacial $O(E)$.

4 Estratégia de Robustez

Os pontos em que mais podem ocorrer erros no código são durante o acesso a índices indevidos, bem como na alocação e liberação de memória dinâmica utilizada. Tendo isso em vista, foram implementados testes que lidam com essas questões.

- **Índices inexistentes -** Caso um evento se relacione com um pacote ou cliente ainda não existente, seus índices eram criados e adicionados automaticamente na AVL respectiva. Além disso, caso uma consulta fosse feita a um cliente ou pacote inexistente, ela era abortada e retornava tamanho de log 0. Dessa forma a integridade dos dados e acesso a memória era mantida.
- **Construtores e Destrutores -** Os construtores são feitos de forma a inicializar todas as variáveis sem nenhum tipo de lixo. Os destrutores tem como objetivo evitar vazamentos de memórias. Para garantir que todo nó da AVL seja excluido da maneira correta, a própria classe deletava o conteúdo do nó (que pode incluir ponteiros) antes de deletar o mesmo.
- **Encapsulamento -** O encapsulamento proposto pela programação orientada a objetos do C++ permitiu maior modularização e menor margem para erros de acesso indevido a memória e a variáveis.
- **Exceções gerais -** O programa é feito para que qualquer acesso indevido lance exceções que são capturadas nos locais corretos. Se intratáveis, as exceções encerram a execução do programa, para evitar maiores complicações.

5 Análise Experimental

A análise experimental realizada tem como base testar como o sistema se comporta em diferentes cenários. Esses cenários são variações da quantidade de clientes, número de pacotes, quantidade de eventos por pacote e quantidade de consultas. As entradas de teste foram geradas automaticamente por um gerador de entradas criado na linguagem Python. Todos os dados são armazenados em arquivos CSV e seus valores possíveis para cada variação são:

- **Clientes -** entre 10 e 500.
- **Pacotes -** entre 100 e 5000.
- **Eventos por pacote -** entre 5 e 500.
- **Número de Consultas -** entre 100 e 5000.

Para os testes, também é feita uma verificação de forma a garantir que cada teste é único e não se repetirá. Após isso, foi usado um outro código em Python que executasse todas as entradas e armazenasse seus dados.

5.1 Clientes

A primeira categoria de testes varia a quantidade de clientes registrados, indo de 10 a 500 clientes. O teste foi feito de forma a mudar unicamente esse parâmetro, ou seja, o número de eventos era mantido. No gráfico abaixo (Figura 1) é possível conferir a relação entre o número de clientes e o tempo de execução do programa.

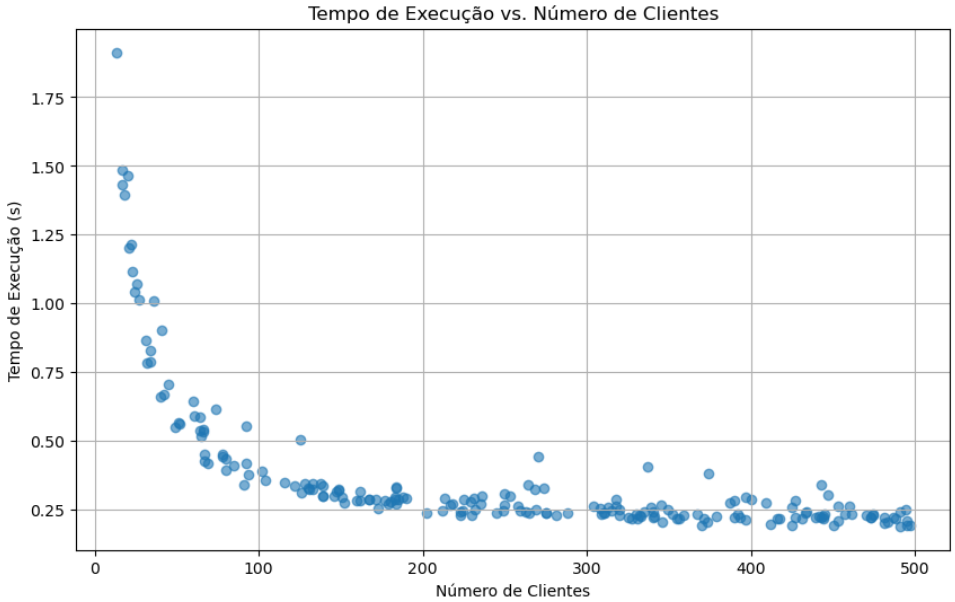

Figura 1: Relação entre clientes e tempo de execução

A princípio o resultado parece contraintuitivo, já que o gráfico indica que quanto mais clientes menor o tempo de execução. Isso acontece pela natureza do experimento, como o número de clientes é o único parâmetro que muda, com menos clientes os pacotes ficam muito concentrados e as consultas de cliente ficam mais caras. Isso alivia quando há mais clientes pois o único preço que aumenta com isso é o custo logarítmico de retornar o índice do cliente da AVL. Enquanto o preço de iterar sobre os pacotes do cliente, $O(P_c)$ diminui bastante. Concluindo, o gráfico revela o comportamento esperado em relação a densidade de pacotes por cliente.

5.2 Pacotes

Nesta etapa, o parâmetro de entrada mutável é somente o número de pacotes. Procurou-se não mudar tanto a quantidade de eventos e linhas de entrada pois essas influenciaram também de forma crescente no tempo de execução (como previsto na análise de complexidade). É possível verificar no gráfico abaixo (Figura 2) como o crescimento do número de pacotes afeta temporalmente o sistema.

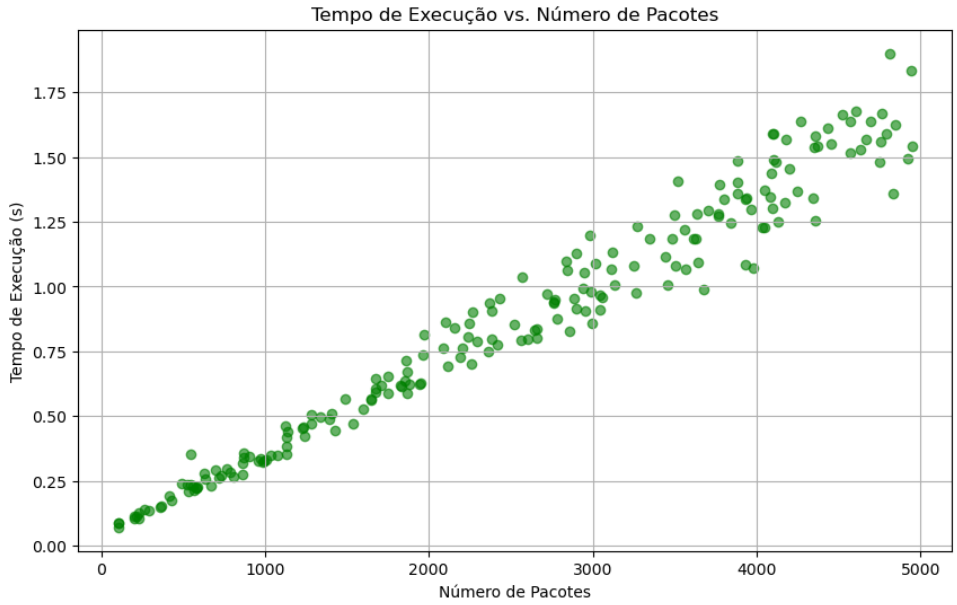

Figura 2: Relação entre pacotes e tempo de execução

Como pode ser visto, há uma forte relação linear positiva entre as duas grandezas. Tal relação era prevista na análise de complexidade, já que diversas funções e métodos chamados tem custo $O(P)$, incluindo a inserção de pacotes em arranjos de clientes e as próprias consultas de cliente. Nesse caso, a inserção em arranjos de clientes fica ainda mais frequente já que mais pacotes resulta em mais registros.

5.3 Eventos por pacote

A terceira etapa da análise experimental consiste em alterar variáveis que definem o número de eventos por pacote. Vale ressaltar que nesse teste o número total de eventos totais também cresce consideravelmente, para evitar que seja muito semelhante ao teste de pacotes (somente um teste de densidade). Para que isso acontecesse, o número de consultas foi reduzido, de forma que o tamanho da entrada seja também o total de eventos.

Na Figura 3, localizada abaixo, é possível conferir a relação entre o número total de eventos e o tempo de execução.

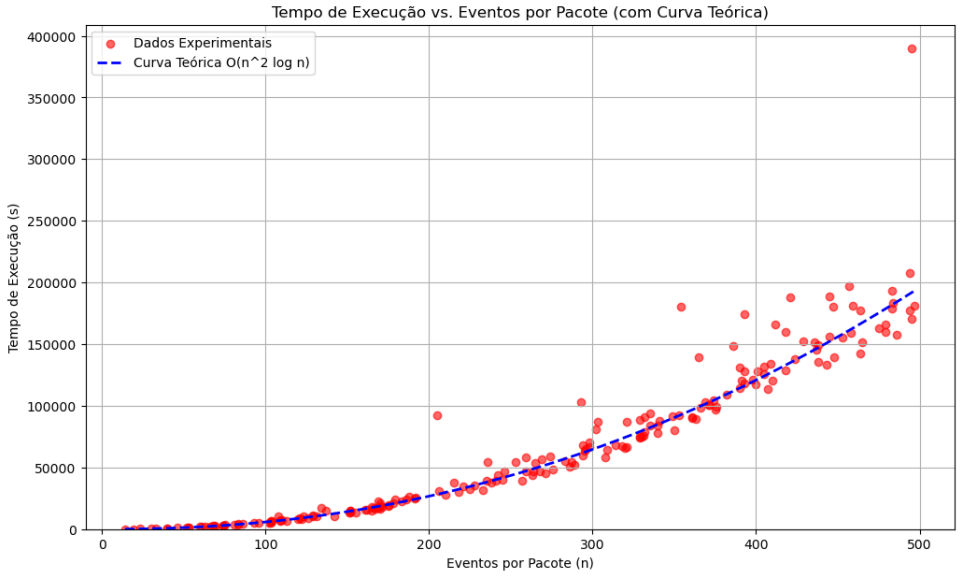

Figura 3: Relação entre o total de eventos e os parâmetros de entrada

Além dos pontos do gráfico foi impressa uma curva $N^2 \times log(N)$ teórica para verificar a aproximação com os dados. É interessantíssimo ver que eles se alinham quase perfeitamente ao esperado. Como visto na análise de

complexidade, o custo temporal total do sistema é $O(N \times E \times log(E))$, mas quando N, que é o tamanho da entrada, se aproxima de E assume $O(E^2 \times log(E))$. Ou seja, assim como previsto, um aumento no número de eventos provoca o aumento aproximado previsto e observado por ambas as análises, de complexidade e experimental.

5.4 Número de Consultas

Por fim, será alterado o número de consultas no arquivo de entrada. Tal análise possibilita uma abstração para um sistema real onde o número de consultas é mais frequente do que o número de eventos em si. Na figura 4 abaixo, é possível conferir a relação da quantidade de consultas e o tempo de execução do sistema.

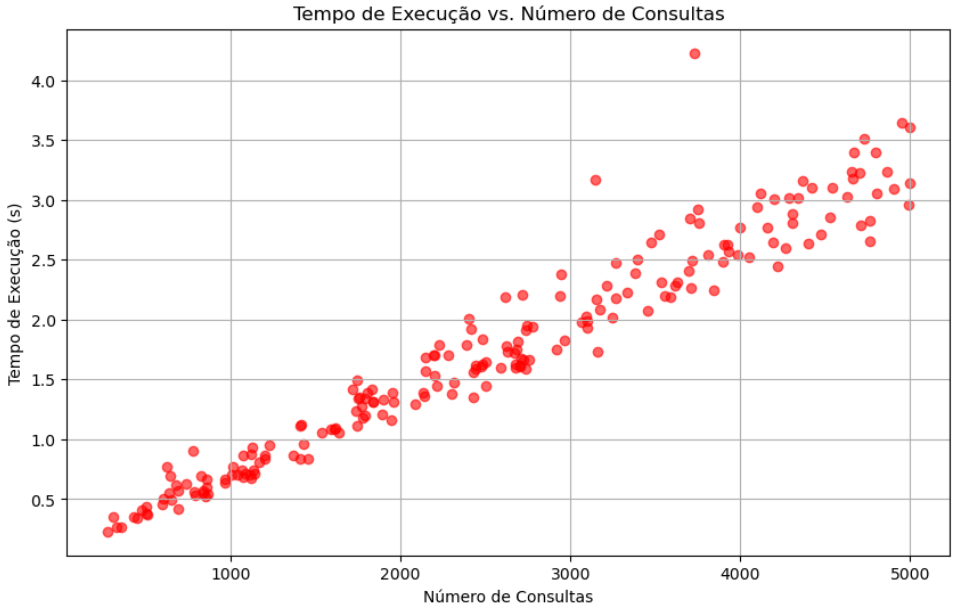

Figura 4: Relação entre o número de consultas e tempo de execução

É perceptível a grande relação linear positiva entre as duas grandezas relacionadas. Isso também era previsto na análise de complexidade, pois mantendo o número de eventos fixos, a complexidade temporal cresce de forma linear em relação ao tamanho da entrada. Relembrando a complexidade temporal total $O(N \times E \times log(E))$, onde neste teste somente N se altera. Isso indica também que em um sistema onde o número de consultas é muito maior e frequente do que o número de eventos, cada consulta vai ter tempo de execução semelhante e será dominante na complexidade total.

6 Conclusão

Destarte, o problema propunha desenvolver um sistema de consultas logísticas eficiente e robusto. Para resolvê-lo foi necessário implementar diversos conceitos cruciais de estruturas de dados eficientes, como a aplicação de árvores AVL para simular índices dinâmicos, arranjos dinâmicos e buscas otimizadas.

Vale comentar que esse trabalho e suas análises posteriores instigam o aluno a pesquisar e aprender mais sobre algoritmos e estruturas eficientes para cada tipo de pesquisa. A escolha entre otimizações espaciais ou temporais é importantíssima para o desenvolvimento e manutenção de projetos mais complexos de forma consistente.

7 Bibliografia

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011

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

Especificação do Trabalho Prático 1 - Ordenador Universal DCC/ICEx/UFMG.

Anisio Lacerda, Wagner Meira Jr., Washington Cunha, Lucas N Ferreira. Estruturas de Dados, Slides da disciplina DCC205 - Estruturas de Dados, Departamento de Ciência da Computação, ICEx/UFMG.

8 Extra

Para uma implementação além do que era previsto no enunciado, foi pensada a disposição um tipo de consulta que retorna todas as movimentações de pacotes dentro de um armazém em um intervalo dado de tempo.

8.1 Adaptação

Para adaptar o programa a esse cenário, foi necessário implementar um novo tipo de linha. Este tipo recebe, além do tempo em que será executado, o identificador AM que representa uma consulta em armazém, seguido pelo tempo de início do intervalo, tempo final do intervalo e identificador do armazém a ser consultado. Além disso foi criada a função LogicSystem::ProccessStorage, sendo a função que vai lidar com essa consulta, chamada na LogicSystem::ProccessLine.

Essa função recebe a linha de entrada, pega o tempos de início e fim, além do identificador do armazém e imprime, assim como as outras consultas, a consulta a ser executada. Após isso cria um arranjo dinâmico que vai conter todos os eventos a serem impressos.

A decisão de projeto mais importante tomada foi a da não criação de uma terceira AVL para armazéns. Tal estrutura ocuparia um espaço $O(E + A)$, sendo E o número de eventos e A o de armazéns, como $A \in O(E)$ a complexidade espacial ficaria $O(E)$. Apesar disso não mudar a complexidade espacial total do programa, em casos de muitos eventos ainda representaria uma grande parcela de memória a mais sendo usada. Dessa forma, como a procura era por um intervalo específico e o arranjo dinâmico de eventos já é ordenado cronologicamente, optou-se pela busca e consulta a partir do próprio.

Isso é feito usando uma busca binária para procurar o primeiro evento dentro do intervalo requisitado. Após isso, itera-se sobre o arranjo de eventos procurando eventos do armazém específico até atingir um evento fora do intervalo ou o fim do arranjo.

Armazéns considerados "movimentados" por um evento são armazéns de origem em eventos de transporte e armazéns de destino em eventos de armazenamento, remoção de armazém, rearmazenamento e entrega.

Após a filtragem de todos os eventos, é imprimido a quantidade de linhas de saída da consulta e cada evento filtrado.

8.2 Complexidade

O próximo passo com o novo algoritmo é verificar sua complexidade, pois pode acabar alterando a complexidade total do projeto.

- **Complexidade Temporal-** A busca binária para encontrar o primeiro evento dentro do intervalo tem custo $O(log(E))$, já que sempre corta o arranjo pela metade. A partir disso, a filtragem de eventos tem custo $O(E_t)$, sendo E_t a quantidade de eventos totais no intervalo de tempo. Como o intervalo pode conter todos os evento, seu custo é $O(E)$. Após isso os eventos filtrados, que também tem complexidade $O(E)$ são impressos. Dessa forma a complexidade temporal total da nova aplicação é $O(E + log(E)) \in O(E)$.
- **Complexidade Espacial-** É criado um arranjo dinâmico de tamanho $O(E_a)$ sendo E_a o número de eventos do armazém requisitado no intervalo especificado. Como todos os eventos podem ser em um mesmo armazém dentro do intervalo $E_a \in O(E)$. Dessa forma a complexidade espacial total da aplicação é $O(E)$.

Como é possível notar, a complexidade da nova consulta é de mesma magnitude das consultas anteriores. Onde a consulta por cliente tem custo espacial e temporal $O(P) \in O(E)$ já que $P \in O(E)$. Porém, é importante notar que as consultas anteriores não iteravam sobre eventos que não seriam impressos. Na nova consulta, mesmo que o armazém não tenha nenhum evento no intervalo, será iterado sobre todos os eventos do intervalo. A princípio isso parece um problema, mas é o $tradeoff$ por um menor custo de memória usada. Este tipo de decisão é comum no ramo do desenvolvimento de sistemas pois muitas vezes é necessário trocar espaço por tempo ou vice-versa.

8.3 Exemplo de Uso

Por fim, segue um exemplo de entrada e saída esperado usando a nova consulta proposta.

Listing 1: Exemplo de arquivo de entrada.

```
0000001 EV RG 003 JOAO MARIA 000 003
0000002 EV AR 003 000 002
0000006 EV RG 000 ESTER JULIANA 001 000
0000006 EV RG 006 LUISA ANDRE 003 002
0000006 EV AR 006 003 002
0000009 EV AR 000 001 002
0000012 EV RG 005 BERNARDO LUISA 002 003
0000018 EV AR 005 002 003
```

```
0000040 EV RG 001 DAVI ANDRE 001 002
0000045 EV AR 001 001 002
0000060 EV RG 006 JOAO JOSE 000 003
0000068 EV AR 006 000 002
0000080 EV RG 002 PEDRO LUISA 000 003
0000080 EV RG 004 LUCIA RICARDO 003 002
0000092 EV AR 004 003 002
0000093 EV AR 002 000 002
0000102 EV RM 001 001 002
0000102 EV RM 002 000 002
0000102 EV RM 004 003 002
0000102 EV RM 005 002 003
0000102 EV TR 005 002 003
0000103 EV RM 000 001 002
0000103 EV RM 006 003 002
0000103 EV RM 006 000 002
0000103 EV TR 000 001 002
0000103 EV TR 001 001 002
0000103 EV TR 004 003 002
0000103 EV TR 006 003 002
0000104 EV RM 003 000 002
0000104 EV TR 003 000 002
0000104 EV TR 006 000 002
0000104 EV UR 002 000 002
0000122 EV EN 005 003
0000123 EV AR 000 002 000
0000123 EV EN 001 002
0000123 EV EN 004 002
0000123 EV EN 006 002
0000124 EV AR 003 002 003
0000124 EV AR 006 002 003
0000202 EV RM 000 002 000
0000202 EV RM 002 000 002
0000202 EV RM 006 002 003
0000202 EV TR 000 002 000
0000202 EV TR 002 000 002
0000203 EV RM 003 002 003
0000203 EV TR 003 002 003
0000203 EV TR 006 002 003
0000222 EV AR 002 002 003
0000222 EV EN 000 000
0000223 EV EN 003 003
0000223 EV EN 006 003
0000302 EV RM 002 002 003
0000302 EV TR 002 002 003
0000322 EV EN 002 003
0000323 AM 001 200 001
```

Listing 2: Exemplo de arquivo de saída.

```
000323 AM 000001 000200 001
6
0000009 EV AR 000 001 002
0000045 EV AR 001 001 002
0000102 EV RM 001 001 002
0000103 EV RM 000 001 002
0000103 EV TR 000 001 002
0000103 EV TR 001 001 002
```

Como é possível ver, na saída há todas as movimentações do armazém 001 no período dado. Vale lembrar que eventos de registro não são inclusos, já que esses não são contados como "movimentações de armazém".