Trabalho Prático 2 - Armazéns de Hanoi

João Henrique Alves Martins - 2024013885

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

jalvesmartins16@ufmg.br

1 Introdução

Os sistemas de escalonamento são ferramentas essenciais para organizar a ordem de realização de tarefas em várias esferas. Esses sistemas otimizam esses processos ao priorizar certas tarefas em relação à outras, de acordo com algum critério específico.

O problema proposto nesse trabalho foi a implementação de um sistema de escalonamento de transporte de pacotes, utilizando armazéns com pilhas, inspiradas nas famosas Torres de Hanoi. Dada a topologia dos armazéns, o programa deve ler um número x de pacotes e, baseado em alguns critérios diferentes, como tempo armazenado e tipo de evento, escalonar todos os eventos de transporte desses pacotes, até que encontrem seu destino final, visando eliminar pacotes que ficam armazenados por mais tempo que o esperado. A simulação é feita de forma discreta, portanto computa apenas os tempos onde de fato ocorreram algum evento.

Essa documentação tem como objetivo esclarecer todas as características do programa que foi implementado. Além da introdução, esse documento conta com detalhes das decisões de implementação (Seção 2), análise de complexidade dos algoritmos usados (Seção 3), estratégias de robustez utilizadas no desenvolvimento (Seção 4), análise experimental (Seção 5) e conclusões finais do trabalho (Seção 6), além da bibliografia utilizada (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, graph, list, heap, queue, stack, node, package, warehouse, transport, event, scheduler.

2.1 Estruturas

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

- **Stack :** O TAD Stack é o ponto central da implementação, pois simula exatamente as Torres de Hanoi ao forçar o desempilhamento de todos os pacotes para mover o último. A classe é uma pilha de nós de pacotes e funciona exatamente como uma pilha convencional. Devido ao número variado de pacotes, decidi implementar utilizando ponteiros. Além disso, acresci um id

para todas as pilhas, para facilitar a identificação dessas pilhas quando estiverem contidas nas seções.

- **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.

- **Queue :** O TAD Queue também é uma fila convencional. Ela foi utilizada para podermos utilizar o algoritmo BFS (Breadth First Search), usado no grafo de armazéns para o calculo das rotas dos pacotes. Sendo assim, é uma fila que comporta apenas inteiros, que no nosso sistema representam os ids dos armazéns. Também foi implementada usando nós encadeados, devido ao fator de variabilidade do tamanho do grafo e das quantidades de armazéns conectados.

- **Package :** O TAD Package representa os pacotes em uma classe simples, que armazenam suas informações básicas e a sua rota, armazenada em uma lista de inteiros. O pacote também contém o atributo status que, por ser dispensável para esse TP, não teve sua lógica implementada. Mesmo assim, o atributo foi adicionado para futuras implementações mais robustas.

- **Warehouse :** O TAD Warehouse representa os armazéns em uma classe que contém um id e uma lista de pilhas de pacotes. Essa lista, chamada de "sessions", representa as diferentes seções que um armazém possui. Cada ligação entre dois armazéns gera uma nova pilha na lista, correspondente ao armazém conectado.

- **Graph :** O TAD Graph representa toda a topologia dos armazéns, em forma de grafo. Cada armazém é representado como um nó do grafo, que contém um id, um armazém, uma lista encadeada de arestas e um ponteiro para o próximo nó. As conexões entre um armazém e o outro são exatamente as arestas do grafo. Apesar da leitura do grafo ser feita através de uma matriz de adjacência, optei por implementá-lo usando a lista de adjacência e nós de armazéns. Essa implementação facilita completamente o processo de transporte e manuseio do grafo, visto que cada nó contém o próprio endereço do armazém.

- **Events :** O TAD Events é uma classe que representa os eventos que ocorrem na nossa simulação discreta e alimentam o nosso escalonador. Os eventos podem ser de dois tipos diferentes, sendo eles transporte e armazenamento, e guarda os atributos importantes para a simulação, como o pacote e os armazéns envolvidos. O atributo mais importante é a chave do evento, que determina a ordem em que os eventos são realizados. Essa chave contém o tempo o evento e outras informações sobre o transporte.

- **Heap :** O TAD Heap é uma classe que implementa um minheap de eventos. Um minheap é uma estrutura que mantem dados ordenados de forma crescente, ou seja, os eventos retirados do minheap são sempre os "menores". Na implementação do sistema, os eventos são ordenados pela chave, que possui o tempo nos dígitos mais significantes e posteriormente os armazéns e tipo de evento. Essencialmente, essa classe é a base do nosso escalonador, visto que é o TAD responsável pela ordenação dos eventos. A implementação do minheap foi feita com um vetor estático, de tamanho máximo = número de pacotes x número de armazéns, que é o número máximo de eventos. Optei por essa implementação por ser a que eu tenho mais familiaridade, visto que foi implementada dessa forma ao longo do curso de Estruturas de Dados.

- **Scheduler :** O TAD Scheduler é o nosso escalonador de eventos. Ele possui apenas um Heap e funções de adição e remoção de eventos. É apenas mais uma camada de encapsulamento das estruturas de dados.

- **Transport :** O TAD Transport é a peça final do sistema. Essa estrutura é responsável por todo o mecanismo de transporte. Ele realiza a manipulação do tempo, calculo de rotas (BFS), execução dos eventos e escalonamento dos eventos. Além de armazenar todas as estatísticas de transporte, como tempo atual, intervalo entre transportes, capacidade do transporte e pacotes entregues, o TAD cria todos os eventos e registra todas as estatísticas dos transportes.

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.

O programa recebe na entrada todas as variáveis globais, sendo elas as estatísticas do transporte (intervalo, capacidade, duração, custo de remoção do pacote) e a topologia do Grafo, através de uma matriz de adjacência. O programa instancia o grafo e cria todos os armazéns e seções (Vértices e arestas), através das funções **readNodes()** e **readEdges()**. Também instancia o Transporte e o Escalonador.

Após isso, recebe os dados de todos os pacotes (Tempo de chegada, id, origem e destino). Durante a leitura de cada pacote, o sistema cria o pacote com os dados de entrada e calcula a sua rota, usando BFS na função **calculateRoute()**. A BFS é um algoritmo de busca em grafos, usada para encontrar o caminho mais curto entre dois nós (Nesse caso, armazéns). Por fim, essa rota é armazenada em uma lista contida dentro do pacote e, com a rota calculada, a chegada dos pacotes no primeiro armazém é escalonada através da função **scheduleEvent()**, que é responsável por escalonar todos os eventos.

Depois de escalonar todas as chegadas, o sistema chama a função **createTransports()** para escalonar o primeiro transporte entre todas as arestas.

Com os eventos de chegada e transporte escalonados, a simulação começa e o sistema entra no loop de execução de eventos. Enquanto o número de pacotes entregues for menor que o número de pacotes, a função **executeEvent()** é chamada.

Essa função faz todo o tratamento dos eventos. Primeiramente, ela retira o próximo evento do Scheduler, e em seguida executa o evento de forma adaptável, variando em relação ao tipo.

Caso o evento seja do tipo 1 (Armazenamento), a função verifica se o pacote chegou no seu destino final (Compara o armazém destino do pacote com o armazém em que ele esta sendo armazenado) e, caso tenha chegado, imprime a chegada, deleta o pacote e incrementa o contador de pacotes entregues. Caso contrário, a função apenas armazena o pacote no devido armazém, através da função **storePackage()**, sempre na seção do próximo destino. A sessão e todos esses parâmetros acima são passados diretamente no evento, facilitando o processo de execução.

Caso o tipo do evento seja 2 (Transporte), a função **transportPackages()** é chamada. Essa função cria uma pilha auxiliar e desempilha todos os pacotes. Logo em seguida, ela escalona a chegada dos pacotes alvo no armazém destino desse evento, garantindo que cada um seja armazenado na seção correspondente ao próximo armazém da sua rota. Esses pacotes são os últimos capacidade-do-transporte pacotes da pilha original, e por construção, são os pacotes armazenados a mais tempo. Os pacotes restantes são reempilhados na pilha original. Para cada um desses passos, o transporte atualiza e imprime as estatísticas. Por fim, após fazer o transporte, a função escalona o próximo transporte para essa aresta, incrementando o tempo de acordo com o intervalo determinado na entrada.

Dessa forma, o programa segue o ciclo: executa evento - escalona novo evento, até que todos os pacotes sejam entregues. Quando essa condição é atingida, o loop se encerra e o programa termina. Todas as estatísticas são impressas no terminal.

Todo esse fluxo se apoia no funcionamento adequado das Estruturas de Dados implementadas para representar cada entidade. O funcionamento e análise de complexidade das funções usadas internamente pelo sistema são detalhadas na próxima seção.

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 esta 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 Pilha

- **push() :** Um pacote é empilhado em *O(1)*, adicionando o pacote no topo. Em relação a espaço, é *O(1)*, pois não usa espaço extra.
- **pop() :** Um pacote é retirado em *O(1)*, retirando o pacote no topo. Em relação a espaço, é *O(1)*, pois não usa espaço extra.

3.2 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.

3.3 Fila

- **enqueue() :** Assim como a pilha, a fila tem um ponteiro no início e fim. Portanto, enfileiramos um item no fim da lista em *O(1)*. Também é *O(1)* em espaço.
- **dequeue() :** Desenfileirar um item segue a mesma regra. *O(1)* em tempo e espaço.

3.4 Grafo

- **readNodes() :** A função de leitura dos nós do grafo chama a função **insertNode()**, que intera sobre um loop de tamanho n, inserindo nós. Portanto, as duas funções são *O(n)* para tempo e espaço.
- **readEdges() :** Para adicionar as arestas, a função chama a **insertEdge()** para todas as casas com o inteiro 1 na matriz de adjacência. Portanto, como percorre a matriz toda, é $O(n^2)$ em tempo e espaço (No pior caso, pode chegar a criar n^2 novas arestas).
- **findWHouseNode() :** Para encontrar um nó na lista encadeada, precisamos percorrer a lista toda. Portanto, a complexidade de tempo é *O(n)* em tempo e *O(1)* em espaço.

3.5 Heap

Essa seção também se aplica ao Scheduler, visto que ele apenas encapsula o TAD Heap.

- **Heap() :** A função de criação do Heap recebe um maxsize, que na nossa implementação é o número de pacotes x número de armazéns. Portanto, a complexidade de espaço é *O(n * k)*, ou $O(n^2)$. Já a complexidade de tempo é constante (*O(1)*).

- **insert() :** Para remover um item do nosso minheap, criamos um novo evento e adicionamos ao fim em *O(1)*. Após isso, precisamos verificar se o heap manteve sua ordem, e caso tenha sido alterada, precisamos reordená-lo. Assim, chamamos a função **bottomHeapfy()**, que é responsável por esse processo. Essa função tem complexidade de tempo *O(log n)*, pois percorre no máximo a altura da árvore. Portanto, a complexidade da insersção é *O(log n)* em tempo e *O(1)* em espaço.

- **remove() :** Para remover um nó do heap seguimos a mesma lógica supracitada. Removemos o primeiro item em tempo constante e depois chamamos a função **topHeapfy()**, que percorre no máximo a altura da árove, reordenando o nosso heap. Portanto, a complexidade da remoção é *O(log n)* em tempo e *O(1)* em espaço.

- **isEmpty() :** Para verificar se o heap está vazio, basta consultar seu tamanho, feito em tempo constante. Portanto, a complexidade é *O(1)* em tempo e *O(1)* em espaço.

- **getters() :** Como o heap foi implementado como vetor, conseguimos acessar as posições de antecessor e sucessores em tempo constante. Portanto, a complexidade é *O(1)* em tempo e *O(1)* em espaço.

3.6 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 **setRoute()**, que é*O(n)* para espaço e tempo, pois copia uma lista de n inteiros para o atributo route.

3.7 Warehouse

- **storePackage() :** Para empilhar um pacote em uma seção específica, precisamos percorrer a lista de pilhas até encontrar a seção desejada, e depois empilhar o pacote na pilha. Portanto, a complexidade de tempo é *O(n)*, e de espaço é constante.

- **addSession() :** Adicionar uma nova seção na lista de pilhas é uma operação constante, pois adicionamos essa nova pilha ao fim da lista encadeada. Portanto, *O(1)* para ambas complexidades.

3.8 Transport

Esse TAD é o detentor das funções responsáveis pela simulação, portanto, é a seção mais importante.

- **calculateRoute() :** A função de cálculo de rotas utiliza do BFS para pesquisa no grafo. Esse algoritmo, implementado com o uso da **queue**, percorre todos os nós do grafo, visitando cada um deles apenas uma vez, e depois visitando suas conexões. Como a nossa implementação foi feita utilizando a lista de adjacência, a complexidade de tempo é proporcional ao número de vértices + suas arestas, ou seja, *O(n + a)*. Para espaço, como aloca vetores de tamanho n+1 posições para os pais dos nós e para marcar as visitas em cada nó, também é *O(n)*.

- **createTransports() :** Essa função é responsável por escalonar os primeiros transportes para todas as conexões entre armazéns, ou arestas do grafo. Para isso, ela itera sobre todos os nós e depois sobre todas as arestas do grafo, escalonando o transporte para cada uma delas. Sendo assim, no pior caso é $O(n^2 * log\ n)$ para tempo, sendo n o número de vértices, em casos de grafos completos, e *O(1)* para espaço, pois não aloca memória extra.

- **transportPackages() :** Essa função é uma das mais importantes do sistema. Para fazer o transporte entre dois armazéns, a função primeiro cria uma pilha auxiliar, de complexidade espacial *O(n)*. Após isso, a função desempilha todos os pacotes da pilha em tempo *O(n)*, sendo n o número de pacotes na pilha original. Depois, esses n pacotes ou são transportados ou

rearmazenados na pilha original, totalizando uma operação *O(n + k log n)*, sendo k o número de pacotes transportados, devido ao escalonamento desses eventos no heap. Portanto, essa função é aproximadamente *O(n log n)* para tempo e *O(n)* para espaço.

- **executeEvent() :** Essa função também é uma das mais importantes do sistema. Para executar um evento, a função tem 2 possíveis abordagens no switch-case:

  Caso o evento seja de chegada, a função tem pior caso em *O(n)*, quando o pacote ainda não chegou no destino final e precisa ser armazenado. A função **storePackage()** é chamada.

  Caso o evento seja de transporte, a função **transportPackages()** é invocada, totalizando uma complexidade de *O(n log n)* em tempo e *O(n)*espaço. Após o transporte, o próximo transporte também é escalonado em *O(log n)*.

  Portanto, no pior caso, a função é *O(n + k+1 log n)* em tempo, ou seja, *O(n log n)*, e possúi pior caso de complexidade de espaço em *O(n)*.

3.9 Events

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

3.10 Main

Com todas as complexidades computadas, podemos analisar a complexidade do nosso sistema como um todo. Em relação a espaço, nosso sistema aloca os pacotes em *O(p)*, um grafo de armazéns e suas pilhas, com pior caso em $O(n^2)$, para grafos completos e um Heap de complexidade $O(n^2)$. Portanto é $O(2 * n^2 + p)$, sendo p o número de pacotes.

Já para o tempo, precisamos somar as complexidades das principais funções. Após a topologia ser determinada, em $O(n^2)$, a complexidade é composta especialmente pelas seguintes operações: Leitura de pacotes e cálculo das rotas, feita em *O(p * n+a)*, ou seja, $O(n^2)$. Depois, criamos os transportes em $O(n^2 * log\ n)$ no pior caso e, para a simulação, temos uma complexidade de *O(w * (p + k+1 log n)*, sendo w o número de execuções, e o resto o custo de remover, executar e escalonar o próximo evento. A complexidade de tempo do nosso sistema é aproximadamente $O(n^2 + (p * n + a) + (n^2 * log\ n) + w * (p + k+1\ log\ n))$.

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; Verificação de tamanho ou posições válido para listas, pilhas e para o heap, evitando acessos inadequados ou inexistentes; Casos default nos swithc-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.

5.1 Número de pacotes x Tempo de execução

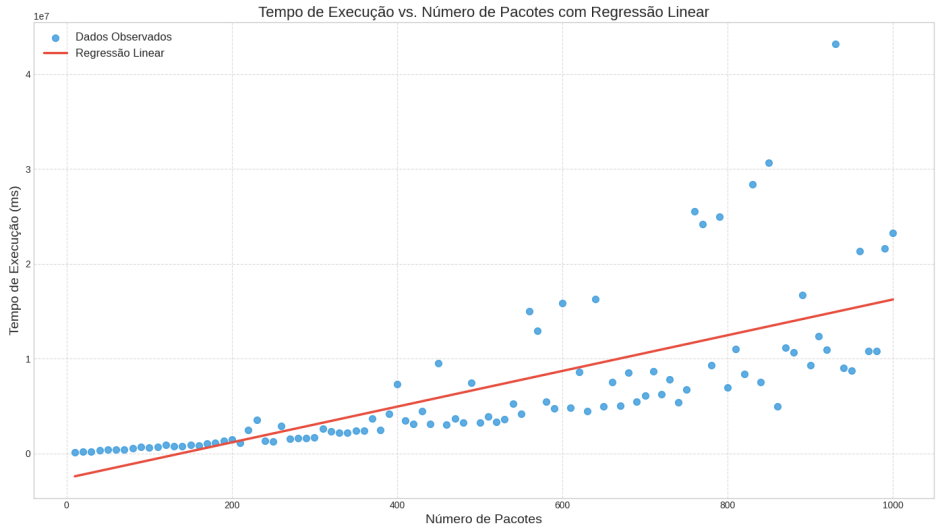

Figura 1: Quantidade de pacotes x Tempo de execução, com regressão linear.

No gráfico acima, é apresentada a relação entre o número de pacotes e o tempo de execução do algoritmo, com a topologia e capacidade de transporte constantes. A curva de crescimento da função mostra que o tempo de execução cresce de forma linear, conforme o número de pacotes aumenta.

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

Analisando a função de complexidade da nossa main, é perceptível que com o número de armazéns constante, os elementos que envolvem a variável p, para pacotes, não são os de complexidade mais elevada. Isso ocorre pois o cálculo da rota vai ter custo constante para cada pacote, o que evita o pior caso dessa função, onde o tamanho do grafo é equivalente ao número de pacotes, tornando o cálculo $O(n^2)$.

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

5.2 Número de armazéns x Tempo de execução

No gráfico abaixo, é apresentada a relação entre o número de armazéns e o tempo de execução do algoritmo, com número de pacotes e capacidade de transporte constantes. A curva de crescimento da função mostra que o tempo de execução, semelhantemente à variação dos pacotes, cresce de forma linear, conforme o número de armazéns aumenta.

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

Mais uma vez, fica nítido que a entidade responsável por puxar a complexidade do nosso programa não são os armazéns.

É importante ressaltar que a esparsidade da matriz de adjacência não foi um elemento explorado nesse experimento. Caso houvessemos aumentando o número de arestas do grafo, provavelmente teriamos um aumento considerável e não-linear no tempo de execução, visto que atingiriamos o pior caso de $O(n^2 * log\ n)$ para escalonar os eventos de transporte, sendo cada evento representante de uma aresta do nosso grafo K(n) (Grafo completo, representado pela letra K, de n nós).

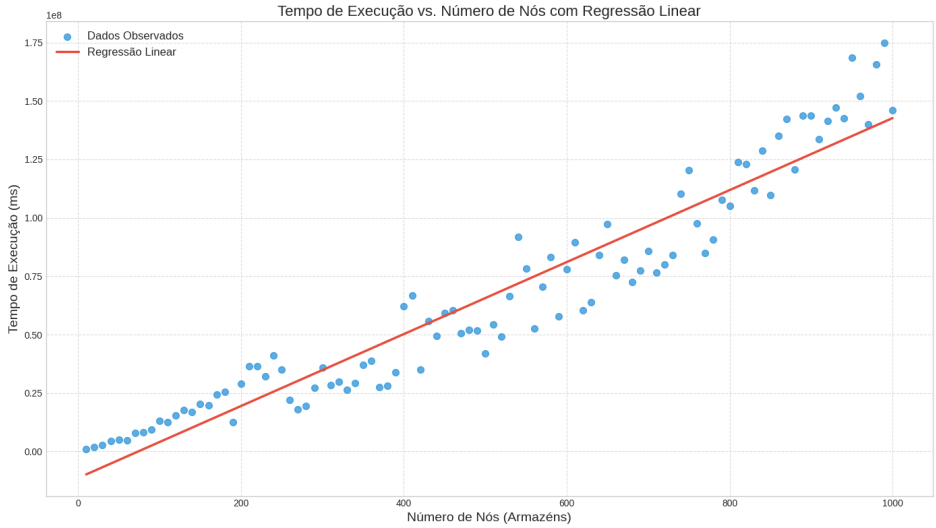

Figura 2: Quantidade de armazéns (Nós do grafo) x Tempo de execução, com regressão linear.

5.3 Capacidade de transporte x Tempo de execução

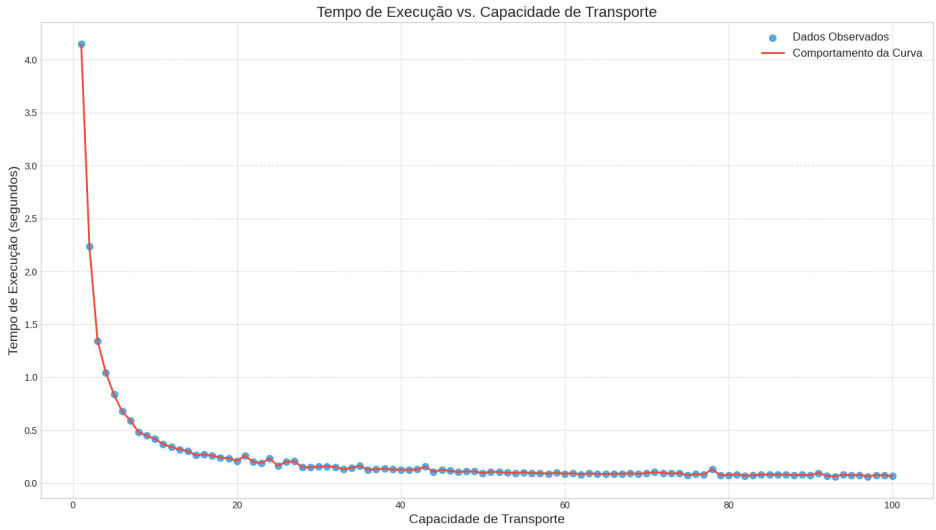

Figura 3: Capacidade de transporte x Tempo de execução.

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

Como esperado, devido à análise de complexidade, a estrutura responsável pela maior complexidade no nosso sistema é o Heap, nosso escalonador de eventos.

A capacidade de transporte é a variável responsável por aumentar significativaemente o número de inserções e remoções de eventos no escalonador, visto que com uma capacidade de transporte muito pequena, o sistema é obrigado a escalonar e realizar muito mais eventos comparado à uma capacidade maior. No caso extremo, com capacidade = 1, o sistema precisa inserir 1 evento de transporte e chegada para cada pacote, explodindo o tempo de execução em sistemas com muitos pacotes.

Para esse experimento, usei 10.000 pacotes e 4 armazéns, variando a capacidade de 1 até 100. O resultado é compatível com a complexidade esperada de diversas remoções e inserções de eventos em um minheap, de *O(n log n)*, observada na função **executeEvent()**.

Esse resultado, junto aos outros 2, comprovam nossa teoria inicial, baseada na análise de complexidade. O nosso escalonador de transportes é mais afetado pelo número de eventos que ele tem que realizar, e não pelo número de pacotes ou armazéns que compõem sua topologia. Esse comportamento simula a realidade de transportes na vida real, onde vemos cada vez mais o uso de containers e navios transportadores, potencializando a capacidade de transporte, mesmo que os destinos sejam muitos ou muito distantes.

6 Conclusões

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

Os resultados da análise experimental mostram a importância de sistemas que otimizem a ordenação e realização de eventos, assim como o implementado nesse trabalho. É evidente que as diferentes configurações dos armazéns ou do transporte levam à resultados totalmente diferentes, mostrando exatamente a necessidade desse tipo de sistema.

Nesse sentido, a eficácia do sistema implementado foi provada nas análises 5.1 e 5.2, mostrando que se tivermos um transportador de capacidade suficiente, nosso sistema será capaz de lidar com esses transportes de forma linear, mesmo com muitos pacotes ou armazéns.

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 lidando com tantas estruturas diferentes. Cada escolha impacta diretamente o funcionamento do produto final.

O desafio de entender a forma com que todas essas estruturas se conectariam para forma o sistema foi a parte mais interessante e desafiadora do trabalho. Sinto que depois desse TP, tenho um conhecimento mais amplo sobre os diferentes TADs e estou muito mais pronto para lidar com desafios mais complexos de computação. Mais uma vez, compreender a proposta foi um desafio, mas depois de concatenar as ideias, consegui dar seguimento tranquilamente com o projeto. O tamanho do projeto foi outra dificuldade.

Seguindo o aprendizado com o TP1, 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 variáveis em inglês e formatadas em snake-case.

Em geral, aprendi muito com esse trabalho e, apesar das dificuldades, fico extremamente satisfeito com o resultado.

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 1 de Estrutura de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.