Trabalho Prático 2

Sistema de escalonamento logístico e simulador de eventos discretos de transporte

Mateus Antinossi Cordeiro Queiroz – 2024023767

Departamento de Ciência da Computação (DCC)

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

mateusacq@ufmg.br

1. Introdução

O problema proposto foi a implementação de um programa que, munido de algoritmos e estruturas de dados, lesse informações de pacotes, armazéns e transportes que uma empresa administrava com um sistema caracterizado por "Hanoi" (empilhamento de pacotes). Também foi necessário escalonar eventos que descreviam as movimentações dos pacotes ao longo do tempo, cabendo, aqui, a simulação de eventos discretos de transporte.

Esta documentação se subdivide nas seguintes seções:

- **2. Método:** seção que trata da implementação: estruturas de dados, tipos abstratos de dados e métodos implementados;
- **3. Análise de Complexidade:** seção que analisa a complexidade de tempo e de espaço dos elementos apresentados na seção 2, utilizando notação assintótica;
- **4. Estratégias de Robustez:** seção onde são discutidas a programação defensiva e a capacidade do código de resistir a falhas e prosseguir com a execução;
- **5. Análise Experimental:** avaliação de diferentes dimensões que impactam o custo computacional da execução do código.
- **6. Conclusão.**

2. Método

O programa foi desenvolvido em C++, usando o paradigma de orientação a objetos para representar os elementos logísticos: pacotes, armazéns, transportes e eventos. As classes principais são: **Simulador**, **Evento**, **Pacote**, **Armazém**, além das estruturas de suporte: Grafo, Heap, Lista, Fila e Pilha.

A leitura da entrada é feita via linha de comando:
./tp2.out <arquivo_de_entrada.txt>

2.1. TAD Simulador:

Basicamente, desejava-se receber dados sobre a topologia dos armazéns (rotas possíveis entre eles) que uma empresa possuía. Em cada um deles, pacotes são armazenados. Dados sobre esses pacotes também são fornecidos no arquivo de entrada. Baseando-se nisso, um simulador de eventos discretos poderia escalonar eventos que representariam o transporte de cada um desses pacotes entre os armazéns até a chegada deles em seu destino.

No entanto, há alguns detalhes importantes: o transporte possuía uma capacidade máxima. Em cada seção dos armazéns, os pacotes eram desempilhados e empilhados, na sequência de remoção, em uma pilha auxiliar. Os últimos que sobrassem, após atingir a capacidade máxima do transporte, seriam imediatamente rearmazenados (reempilhados) na mesma seção. Isso é o armazenamento Hanoi.

Para a simulação de eventos discretos, algumas variáveis eram importantes: o intervalo entre transportes, o tempo que pacotes ficavam em transporte (latência) e o tempo de chegada do pacote no armazém. Um relógio era usado para rastrear esses eventos.

Os dados dos pacotes e dos armazéns eram obtidos da com o método “leArquivo(in)”.

Vetores que armazenavam os pacotes e os armazéns foram alocados nessa classe também. Não somente, estruturas de dados que representavam a topologia e o escalonador de eventos foram alocadas dinamicamente, para ter um tamanho variável.

2.2. TAD Evento:

Essa classe representava um evento que poderia ser de dois tipos: 1 (pacote) e 2 (transporte). Se fosse um evento do tipo 2, o objeto receberia o tempo em que o evento iria começar, o armazém de saída e o armazém de chegada do transporte feito. Se fosse do tipo 1, armazenaria o id do armazém onde um pacote estava chegando (“origem”) e o destino do pacote, bem como o tempo em que isso ocorreria e o ID do pacote. No simulador de eventos essas informações eram úteis para saber se um pacote estava chegando no seu armazém de origem, em um intermediário ou no seu destino.

Cada evento gera uma chave única em string, usada como critério de comparação e obtenção da maior prioridade no heap, garantindo a ordenação.

Dígito	1	2	3	4	5	6	7	8	9	10	11	12	13
Pacote	Tempo						Pacote						Tipo
Transporte	Tempo						Origem			Destino			Tipo

Figura 4. Especificação das chaves da fila de prioridade

Exemplos de chave:

0003220000021 – Uma chave do tipo 1 (pacote)

0004010020012 – Uma chave de evento do tipo 2 (transporte)

2.3. Pacote:

Essa classe possuía atributos relacionados ao pacote (id do armazém de origem e do de destino, id do pacote, rota que percorreria para chegar ao destino, tempo que chegou na origem) e estatísticas que para análise. Possui métodos de acesso (getters e setters).

2.4. Armazém:

Classe com atributos que representassem um armazém (seu id, o número de pacotes ali armazenados e o número de armazéns da empresa, que também correspondia ao número de seções no armazém). Também possuía uma pilha para cada seção, onde eram armazenados os pacotes, que foi alocada dinamicamente como um vetor da estrutura de dados pilha, chamado “seções”. Inclui métodos para armazenar, recuperar e verificar pacotes, que manipulavam essa pilha com o auxílio dos seus métodos internos.

2.5. Grafo:

Representava a topologia dos armazéns, com as rotas que existiam entre cada um deles. A estrutura do grafo em si era representada, em sua classe, com uma lista de adjacência. Foi feito assim para que o algoritmo de busca em largura (BFS) pudesse ser implementado, juntamente do uso das estruturas de dados lista encadeada, pilha e fila.

Possuía métodos internos de verificações de segurança, alguns getters, verificadores e funções de adicionar e remover arestas que montavam o grafo na função leGrafo().

Exemplo de topologia:

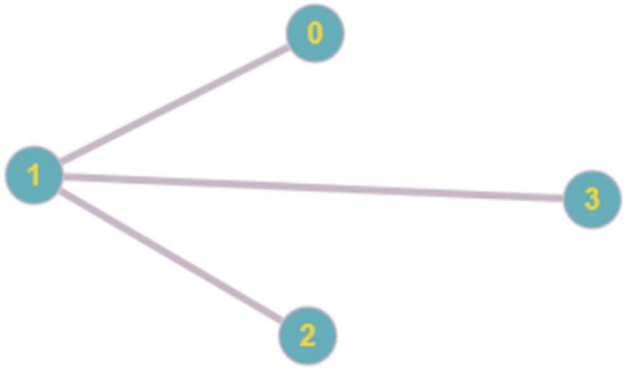

Figura 3. Grafo da topologia

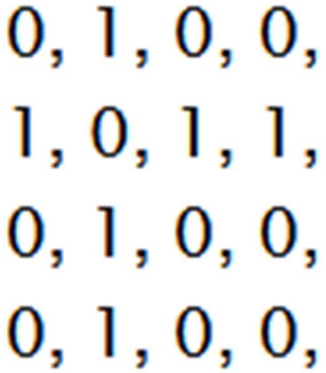

Figura 4. Matriz de adjacência auxiliar

2.6. Heap:

Fila de prioridade que organiza os eventos pelo tempo, implementada como um **min-heap**. O heap possuía atributos onde eram definidos o tamanho e a capacidade dele, juntamente de um vetor de elementos do tipo Evento, que representava a estrutura. Possuía métodos de inserção e remoção da raiz (InsereEvento() e RetiraProximoEvento()) e métodos internos que garantiam suas propriedades de Heap (ordenação de pais menores que filhos), são eles: **GetAncestral(), GetSucessorDir(), GetSucessorEsq(), HeapifyPorBaixo() e HeapifyPorCima()** (recursivos).

2.7. Lista:

Implementação da estrutura de dados lista encadeada com um nó cabeça sentinela.

Possui apontadores para a primeira célula (célula apontada pelo nó cabeça), e para a última. Conta com getters e setters para a primeira posição e para um item em alguma posição passada no método e com os métodos:

- **Posiciona() e PosicionaAnterior():** métodos internos usados para obter um apontador para uma célula em alguma posição.
- **Insere e remove Início, final ou Posição:** métodos para inserir ou remover células no início (primeira célula após o nó cabeça), no final (apontado por "último") ou em uma posição passada na função.

2.8. Fila:

Uma estrutura de dados semelhante à lista, porém, com uma lógica de inserção e remoção diferentes. Aqui, não é possível remover em posições definidas (como o final ou o início. Com sua lógica de remoção FIFO (First-in-first-out), o método realiza:

- **Enfileira():** insere uma célula ao final da fila;
- **Desenfileira():** remove a primeira célula da fila.

2.9. Pilha:

Análoga à fila, armazena células encadeadas, mas, tem lógica de remoção LIFO (Last-in-first-out). Seus principais métodos são:

- **Empilha():** insere um elemento no topo da pilha.
- **Desempilha():** remove o elemento no topo.

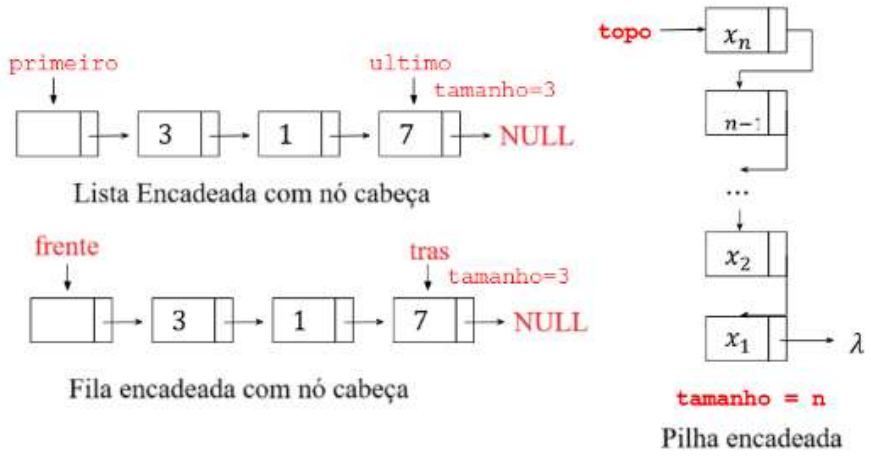

Figura 6. Esquemas ilustrativos das estruturas lista, fila e pilha

2.10. main()

Lê o nome do arquivo de entrada via argumentos da linha de comando e inicia a simulação chamando o simulador.

3. Análise de Complexidade

Cada um dos métodos descritos tem suas particularidades e suas operações próprias, que contribuem para o funcionamento do programa como um todo. Por isso, cabe, nessa seção, uma análise da complexidade de tempo e de espaço deles, formalizada pela notação assintótica.

3.1 Tempo

3.1.1 O(1)

Getters, setters, inserção/remoção simples nas estruturas (pilha, fila, lista, heap raiz) são métodos com complexidade de tempo O(1).

3.1.4 O(n)

Considerando "n" o número de elementos na estrutura: os métodos que limpam as estruturas de dados: métodos de limpeza, destrutores, impressores de estruturas, o método de pesquisa na lista, a verificação de arestas no grafo e a execução da BFS (caso percorra todos os vértices) possuem complexidade O(n).

3.1.5 $O(\log_2 n)$:

As funções de "Heapify" e o método de inserção no Heap possuem, no pior caso, complexidade $O(\log_2 n)$, que seria o caso em que percorreriam a altura do heap, caminho este que possui comprimento $\log_2 n$.

3.1.6 $O(n^2)$:

A leitura da matriz de adjacência e, no pior caso, a leitura geral do arquivo de entrada, considerando o tamanho da matriz ou o número de pacotes eram métodos com complexidade de tempo $O(n^2)$.

3.1.7 Função principal: Simulador::simulação()

Considerando n = número de pacotes, m = número de armazéns, pode-se analisar a complexidade do coração do programa:

Para o agendamento inicial de eventos, temos: O(n) para as chegadas de pacotes, O(E) para os transportes entre cada par de armazéns conectados.

Considerando Q o número de eventos ativos na fila de prioridade, inserção e remoção de eventos têm complexidade $O(\log_2 Q)$. Em um caso extremo (todos os pacotes passam por todos os armazéns), Q = n x m, logo, a complexidade ficaria $O(\log_2(n \times m))$.

Processamento de eventos: Cada evento tipo 2 (transporte) pode exigir varredura de uma seção inteira, gerando custo **O(n)**.

3.1.8 Complexidade de tempo total:

Inicialização: $O(n + E)$ Caso médio: $O(n \times \log_2 Q)$ Pior caso: $O(n \times m \times \log_2(n \times m))$

3.2 Espaço

Usando as mesmas variáveis da seção 3.1.6 e ainda na função principal (simulação):

3.2.1. Fila de Prioridade

Este Heap alocado com tamanho fixo NUM_MAX_EVENTOS possui complexidade de espaço O(NUM_MAX_EVENTOS), desconsiderada para a complexidade total.

3.2.2. Vetor de Armazéns

$O(m^2 + n)$, m^2 pelas seções, n pelo total de pacotes armazenados em todos os armazéns.

3.2.3 Vetor de Pacotes

Cada pacote guarda sua rota, com até m posições: O(n x m).

3.2.4 Grafo de topologia

O(m + E) (listas de adjacência + vértices).

3.2.5 Matriz de adjacência auxiliar

$O(m^2)$, para ler conexões entre todos os m armazéns.

3.2.4 Complexidade de espaço total

$$O(m^2 + n) + O(n \times m) + O(m + E) + O(n^2) = O(n \times m + m^2 + E)$$

No pior caso (grafo denso, $E \sim m^2$), a complexidade fica: $O(n \times m + m^2)$

A menos que n seja pequeno em relação a m^2, termo n x m predomina se n é grande.

3.3 Conclusão

A análise confirma que a decisão de fazer um sistema de escalonamento logístico, buscando a maior eficiência do algoritmo, é válida: foi possível implementar um programa eficiente, com complexidade de tempo total $O(n \times m \times \log_2(n \times m))$ no pior caso, porém com uma complexidade de espaço $O(n \times m + m^2)$ maior, graças ao trade off entre uso de espaço e eficiência no tempo de execução.

4. Estratégias de Robustez

Para garantir robustez e evitar falhas durante a execução, foram aplicadas as seguintes estratégias:

4.1. Validação da entrada

O programa garante que a linha de comando correta é usada na sua execução pelo terminal, sem comprometer o resto da execução, através de uma verificação no começo do main().

4.2. Monitoramento e depuração

Nas estruturas de dados, há o lançamento de erros com o método std::cerr para notificar comportamentos inesperados dos métodos. Mensagens de depuração acompanham operações e movimentações importantes do programa.

Essa estratégia facilita a depuração do programa e o tratamento de falhas, contribuindo, também, para a robustez do código.

4.3. Controle de memória

Estratégias que prevenissem segmentation fault foram usadas com a ideia de controlar memória. Uma delas envolvia a verificação dos valores passados em getters e setters, para ver se estavam dentro dos limites comportados pela estrutura.

Além disso, todas as classes com alocação dinâmica possuem construtor de cópia, destrutor e sobrecarga do operador de atribuição, respeitando a Rule of Three e prevenindo shallow copies e double free.

5. Análise Experimental

3 experimentos analisaram variações em dimensões da entrada. Para tanto, um gerador de cargas de trabalho aleatória, com os parâmetros de interesse controlados, foi usado, junto de uma base com os dados retornados e de gráficos que os exibissem.

Experimento I

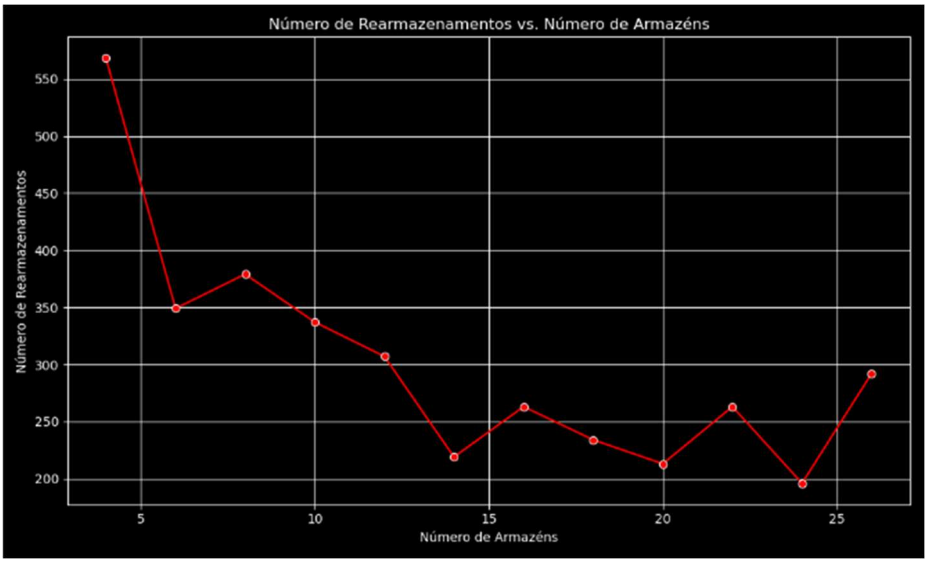

Figura 7. Gráfico que relaciona parâmetros com o aumento do número de armazéns

Analisando o gráfico da figura 7 (ao lado), observa-se que o aumento do número de armazéns impacta na diminuição do número de operações de rearmazenamento de pacotes, feitos quando pacotes de uma seção não pudessem ser enviados para transporte. Essa é uma conclusão razoável, uma vez que mais armazéns são mais opções para transportar os pacotes, como que “diluindo” a distribuição dos itens.

Experimento II

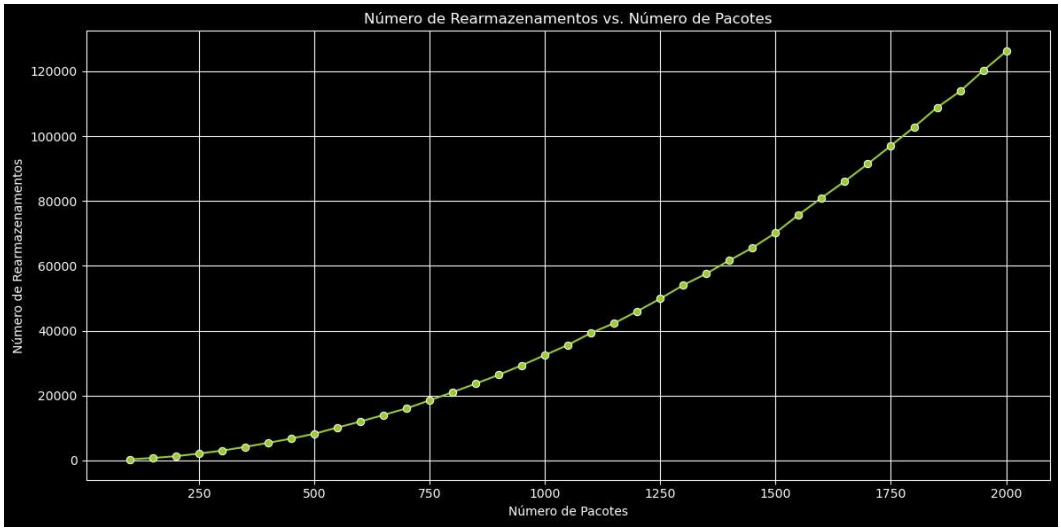

Figura 8. Gráfico com efeito da variação do número de pacotes

Aqui, o aumento do número de pacotes (mantendo número de armazéns fixo) causou aumento no número de rearmazenamentos, o que mostra que a dimensão de número de pacotes é mais capaz de gerar um gargalo no sistema.

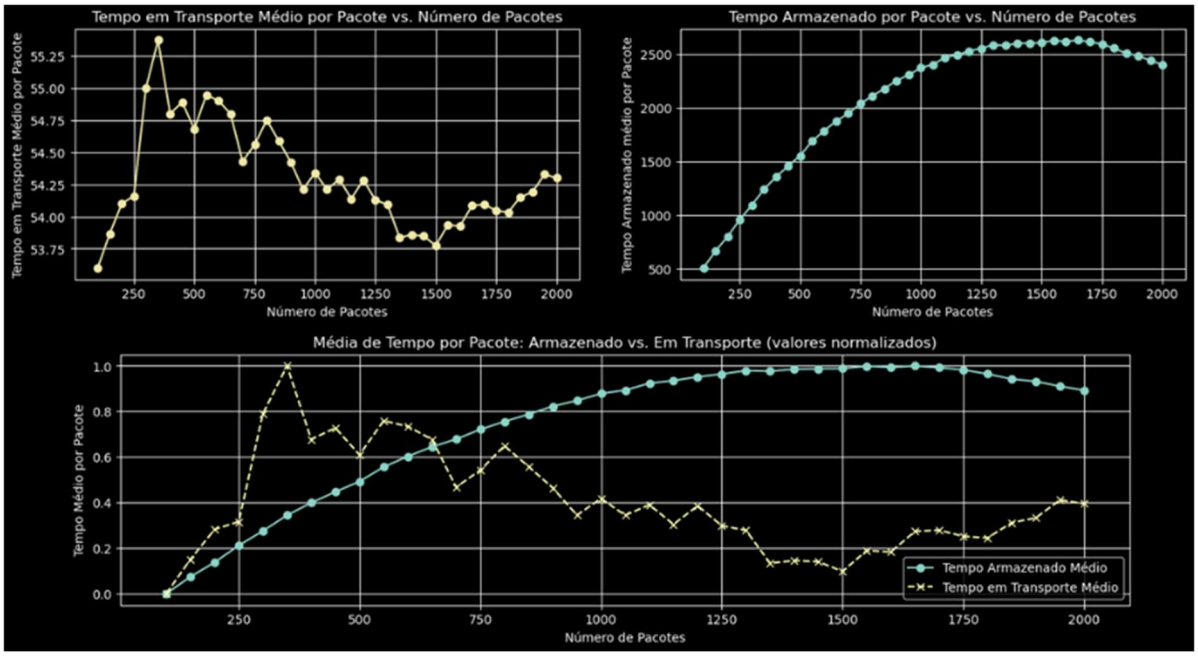

Figura 9. Influência da variação do número de pacotes em estatísticas do programa

Nesses gráficos compara-se os tempos em transporte (decrescente) e armazenado (crescente) médios obtidos na medida que o número de pacotes aumentava. Enquanto uma cresce, a outra decresce. Aqui é possível visualizar melhor a possibilidade de gerar gargalo com muitos pacotes na carga de trabalho.

Experimento III

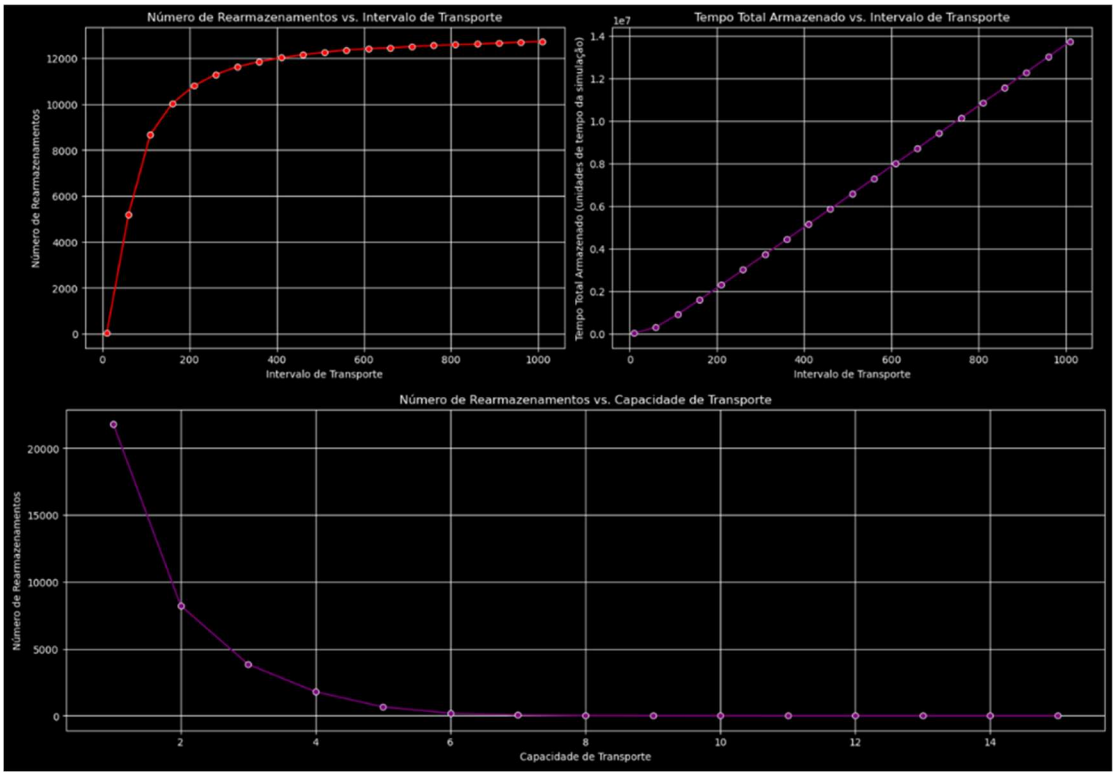

Figura 11. Gráficos da dimensão de frequência de transporte

A frequência de transporte e a capacidade afetam diretamente o número de rearmazenamentos e o tempo armazenado total dos pacotes.

Adicional: Tempo de execução

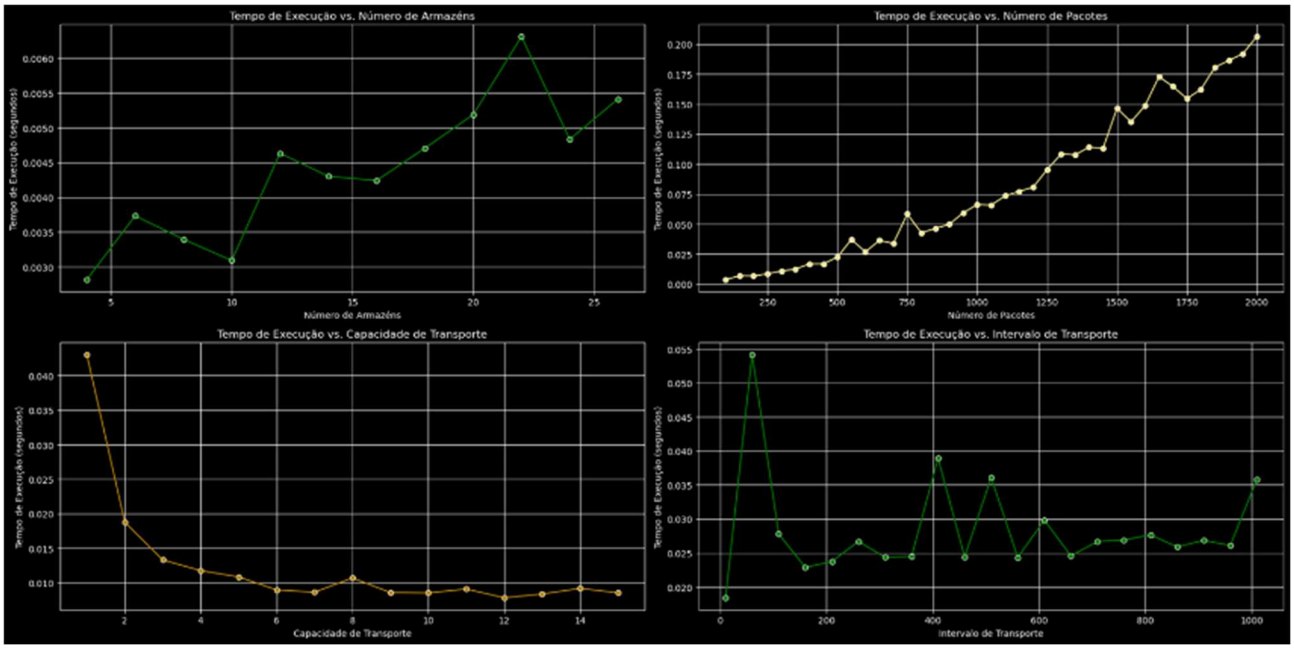

Figura 12. Gráficos do tempo de execução vs núm. de armazéns, de pacotes, capacidade de transporte e intervalo, respectivamente.

O tempo de execução cresce com o número de armazéns e pacotes e reduz com o aumento da capacidade de transporte.

6. Conclusão

O simulador mostrou-se eficiente, com bom balanceamento entre custo temporal e uso de memória. A arquitetura com eventos discretos e o uso de TADs adequados garantiu robustez e escalabilidade. Pela análise experimental, algumas dimensões especiais, com mais capacidade de congestionar o sistema, puderam ser identificadas na carga de trabalho, principalmente o número de pacotes a ser gerenciado.

Bibliografia:

Meira, W. e Lacerda, A. (2025). Slides virtuais da disciplina de estruturas de dados. Chaimowicz, L. and Prates, R. (2020). Aulas virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte.

Chaimowicz, L. and Macharet D. (2024). Slides virtuais da disciplina de Programação e Desenvolvimento de Software 2.

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: Seção 22.2: Breadth-First-Search.

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011: Capítulo 3: Estruturas de dados básicas; Capítulo 7: Algoritmos em Grafos.