Trabalho Prático 2 de Estruturas de Dados - Sistema de Escalonamento Logístico

Lara Amélia Maia de Freitas

2024005912

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte – MG – Brazil

laraameliamf@gmail.com

1. Introdução

O problema proposto foi a implementação do Sistema de Escalonamento Logístico da empresa "Armazéns Hanói", inspirado no funcionamento da Torre de Hanói. Assim, o principal objetivo foi registrar corretamente as chegadas de pacotes a armazéns, obter rotas ótimas para cada pacote, e acompanhar o seu trânsito até a chegada ao destino final na rede de armazéns da empresa. Para tanto, foram empregadas diversas estruturas de dados que atuaram como representantes de componentes do sistema real de armazéns, além de ser implementada uma simulação discreta de eventos representando o funcionamento real do sistema logístico.

Esta documentação tem como fim descrever a implementação do sistema (o que será feito nas seções 2, "Método" e 4, "estratégias de robustez"), além de analisar o seu funcionamento e custo (o que será feito nas seções 3, "análise de complexidade", 5, "análise experimental" e 6, "conclusões"). A seção 7 conta com a bibliografia utilizada no desenvolvimento do trabalho.

2. Método

O sistema foi inteiramente desenvolvido em linguagem C++ e compilado com o compilador g++ da GNU Compiler Collection, especificamente na versão c++11. O código está organizado em, além do Makefile, mais 17 arquivos, sendo 8 deles do tipo .hpp (headers dos TAD's utilizados na solução) e 9 do tipo .cpp (implementações dos headers e da main, que define o fluxo principal do programa). Para solucionar o problema, foram definidas 5 classes próprias do sistema (representam seus elementos do mundo real), além de classes auxiliares para a execução de suas funções.

A classe Pacote representa um Pacote real que é inserido e deve ser acompanhado ao longo de sua movimentação no sistema. Conta, basicamente, com atributos que fornecem informações sobre o pacote (como armazém de origem e de destino, identificador único, horário de postagem, dentre outros), construtores e métodos getters e setters. Também tem como atributo uma lista encadeada que armazena a rota a ser percorrida pelo pacote no sistema de armazéns. Em geral, o Pacote mais é manipulado por outros métodos e classes do que tem funções em si, além do fator representativo de um pacote real. Um método importante na classe é o getProximoRota(), que percorre a lista encadeada representante da rota do pacote e retorna o próximo vértice presente na rota com base no vértice atual em que o pacote está. Caso não haja próximo (ou seja, o pacote já chegou ao fim de sua rota), o método retorna -1.

A classe Armazém representa um armazém no sistema. Ela é acompanhada por uma classe "auxiliar" Secao, que representa as seções existentes no armazém. Cada seção conta com duas pilhas (uma principal onde os pacotes são armazenados e uma auxiliar utilizada em transportes), além de um identificador para o destino ao qual ela se refere. Além disso, também há atributos com informações acerca do armazém em si, como seu identificador no grafo e o custo de remoção em seções. Além dos getters, setters e construtores, os principais métodos da classe são: armazenaPacote (recebe um ponteiro para pacote e a posição no vetor de seções do armazém em que a seção com o destino desejado se encontra, e armazena o pacote na pilha principal da seção associada), esvaziaPrincipal (recebe o índice no vetor de seções em que a seção desejada se encontra e desempilha um pacote da pilha principal, o empilhando na pilha auxiliar da seção), carregaTransporte (desempilha um pacote da pilha auxiliar na seção desejada), retornaPrincipal (desempilha pacotes da pilha auxiliar e os empilha na pilha principal da seção associada) e encontraSecao (encontra o índice no vetor seções da seção com o destino especificado como parâmetro). Além disso, há métodos para checagem de tamanho das pilhas no armazém (tamSecaoAux, checaVaziaAux, checaVazia e tamSecaoPrincipal). As pilhas utilizadas nas seções também constituem uma das estruturas "auxiliares" do sistema. Foram implementadas utilizando ponteiros e contam com os métodos típicos desse tipo de estrutura de dados: Empilha (insere um novo elemento, nesse caso, ponteiros para pacotes, na pilha), Desempilha (remove um elemento do topo da pilha), Limpa (que remove as estruturas usadas pelo TAD) e Vazia (checa se a pilha está vazia com base em seu tamanho). A classe Pilha é acompanhada pela classe TipoNo, que representa os nós utilizados nela.

A classe Transporte representa os caminhos entre os armazéns sendo, na prática, um grafo implementado utilizando lista de adjacência. Além da estrutura do gráfico em si, o principal método

da classe é o buscaLargura, que realiza uma busca em largura entre 2 vértices passados como parâmetro (origem e destino) para obter o menor caminho possível entre eles. Para essa classe e esse método em específico, também foi implementada uma classe Fila. Foi utilizada uma implementação com ponteiros para que o seu tamanho pudesse ser variável, e os métodos presentes são os típicos de uma fila: desenfileira (retira um elemento do início da fila), enfileira (insere um elemento no final da fila), Vazia (utilizada para checar se a fila está ou não vazia com base em seu tamanho atual) e Limpa (que faz a limpeza das estruturas empregadas na Fila). Além disso, para a Fila também há uma classe auxiliar TipoCelula, que representa os nós da Fila.

A classe Evento encapsula dados de possíveis ações a serem tomadas pelo sistema, em especial ações de Pacote (armazenamento de pacotes em um armazém) e Transporte (que representam a movimentação entre armazéns). Além de construtores, destrutor e métodos getters e setters para a classe, os principais métodos são construirChavePacote (que constrói chaves para eventos de tipo 1 conforme especificações), construirChaveTransporte (que constrói chaves para eventos de tipo 2 conforme especificações) e os métodos decodificadores (decodficarTipoEvento, que extrai o tipo evento da chave; decodificarTempoEvento, que obtém o tempo de ocorrência do evento; decodificarIdPacote, que extrai a ID do Pacote associado ao evento no caso de eventos de tipo 1; decodificarOrigemTransporte, que obtém a origem do transporte representado pelo Evento e decodificarDestinoTransporte; que obtém o destino do transporte associado). Além disso, o Evento também armazena informações associadas - tempo, chave, tipo, origem, destino e ponteiro para o pacote associado.

A classe Escalonador realiza, de fato, a lógica do escalonamento logístico da empresa. Para a sua implementação foi utilizada uma estrutura de minHeap, sendo ponteiros para Eventos os elementos inseridos e removidos da estrutura. Assim, o fator de comparação utilizado para construção e remoção no Heap é a chave dos eventos. A estrutura de Heap foi implementada com o uso de um vetor, e a classe conta com os métodos e atributos típicos: método para inserção (Inserir, que recebe um ponteiro para Evento), remoção (Remover, que remove um elemento do Heap que é, por construção, o de menor chave), métodos para encontrar elementos no Heap (GetAncestral, GetSucessorEsq e GetSucessorDir) e para checar se o Heap está vazio (Vazio).

As demais estruturas auxiliares implementadas para o sistema são Lista Encadeada e Lista de Adjacência, acompanhadas por seus respectivos nós, Node para Lista Encadeada e NodeAdj para a Lista de Adjacência. Ambas são implementadas utilizando ponteiros e contam com as operações típicas de seus tipos de TAD.

O fluxo do programa, que define a funcionalidade do sistema logístico em si, é implementado pela main, definida no arquivo mainv2.cpp. Nessa parte do sistema, temos a leitura dos dados da entrada (que definem os parâmetros do sistema logístico e fornecem informações sobre a rede de armazéns e sobre os pacotes a serem transportados) e o loop com a simulação de eventos. Inicialmente, são criados os primeiros eventos de transporte entre cada par de armazéns vizinhos e os eventos de chegada/postagem dos pacotes lidos em seus armazéns de origem. Os TAD's representativos de elementos do mundo real também são criados e populados de acordo com a entrada. Enquanto o escalonador não estiver vazio, o que indica que existem eventos, ou algum pacote não tiver sido entregue ainda, realizamos as seguintes ações: se o evento é de transporte, realizamos as ações necessárias (remover da pilha principal, retirar e retornar da auxiliar - para essas ações, realizamos loops de acordo o tamanho das pilhas e capacidade de transporte) e marcamos um novo evento de transporte para o par origem-destino associado ao evento; se o evento é de pacote (ou seja, está ocorrendo a chegada de um pacote a um armazém), checamos se o pacote está chegando ao destino final (se for o caso, o marcamos como entregue) e, se não for o caso, o colocamos no armazém correto na seção correspondente ao próximo armazém em sua rota.

2.1. Instruções para compilação e execução

Antes de tudo, é necessário que já esteja instalado e disponível um compilador para linguagem C++. Recomenda-se o compilador g++ da GNU Compiler Collection, especificamente na versão c++11, que foi utilizado em todo o desenvolvimento do projeto.

1. Abra o terminal e vá até o diretório raiz do projeto, com cd + <nome do diretório>

2. Digite o comando ‘make’ ou ‘make all’, que compila os arquivos e gera o executável ‘tp2.out’, armazenado na pasta bin.
3. Após a compilação, para executar o programa, digite ‘./bin/tp1.out <nome do arquivo de entrada>’. Caso o arquivo esteja armazenado em um diretório específico, inclua o caminho para ele antes do nome, como ‘diretorio/<nome do arquivo>’.
4. Para limpar os arquivos armazenados nas pastas bin e obj, use o comando ‘make clean’.
5. **Exemplo de entrada para execução:** ./bin/tp2.out teste1.txt

3. Análise de Complexidade

A análise de complexidade será realizada por classes, além de também conter uma seção para métodos “generalistas”, como getters, setters e construtores.

1. **Métodos getters e setters:** esse tipo de método está presente em todas as classes/TAD’s implementados. Como realizam somente operações de custo constante (atribuições para setters e retorno para getters), a sua complexidade de tempo é $O(1)$. Além disso, como as variáveis com as quais operamos já existem ao chamar esses métodos, não é necessária memória auxiliar e a complexidade de espaço é $O(1)$.
2. **Construtores e Destrutores:** também estão presentes na grande maioria das classes. Assim como os getters e setters, executam somente operações de custo constante (como atribuições, alocação e desalocação de memória) sobre objetos já existentes. Logo, as complexidades tanto de espaço quanto de tempo são $O(1)$.
3. **Classe Pilha:** como utilizamos uma implementação com ponteiros para a pilha, as operações principais do TAD apenas movimentam os ponteiros. Assim, os métodos Empilha e Desempilha apresentam complexidade de tempo $O(1)$. O método Vazia também apresenta complexidade de tempo $O(1)$, já que apenas realiza uma comparação e um retorno. Já o método Limpa(), como tem de iterar pela Pilha, apresenta complexidade $O(n)$, com n o tamanho da Pilha, que é, no máximo, a quantidade total de pacotes no sistema. Em uma análise geral, a complexidade de espaço ao operar com a Pilha é $O(n)$, já que conta com n ponteiros para Pacote em sua estrutura.
4. **Classe Fila:** similarmente à Pilha, utilizamos uma implementação com ponteiros, de forma que os métodos Enfileira e Desenfileira apresentam, ambos, complexidade de tempo $O(1)$, assim como o método Vazia. A Fila é utilizada unicamente para performar a busca em largura sobre os vértices do grafo. Dessa forma, a complexidade de tempo do método Limpa é $O(v)$, em que v é o número de vértices no grafo, o que corresponde ao número de armazéns do sistema. Em uma análise do uso da classe como um todo, temos complexidade de espaço $O(v)$ - o pior caso ocorre quando todos os vértices têm de ser inseridos na fila e o melhor caso ocorre para a rota trivial, em que origem e destino são os mesmos e o custo é $O(1)$.
5. **Classe Lista Encadeada:** os métodos posicionaEm, posicionaAntes e inserePosicao tem de iterar sobre a lista até uma determinada posição. Se o tamanho da Lista é k, temos, para as três, complexidade de tempo $O(k)$ - no pior caso, iteramos por toda a lista e, no melhor caso, por apenas um item. Seguindo a mesma lógica, os métodos Limpa e pesquisa também apresentam complexidade $O(k)$. Em uma análise geral, a complexidade de espaço gerada pela classe é também $O(k)$, uma vez que armazenamos k nodes da Lista.
6. **Classe Lista de Adjacência:** é utilizada na implementação do grafo representado pela classe Transporte. Seja v a quantidade vértices no grafo/armazéns no sistema e e a quantidade de adjacências (ou seja, arestas) no grafo. Assim como na Lista Encadeada, os métodos posicionaEmAdj e posicionaAntesAdj apresentam complexidade $O(v)$ - tem de iterar sobre v elementos no pior caso e sobre um só elemento no melhor caso, com complexidade $O(v)$. A complexidade de espaço para o seu uso, no geral, é $O(v + e)$ - temos uma lista de tamanho v para os vértices e, outras listas de tamanho total v representando as adjacências. Os métodos insereVert, insereAresta e pesquisaVert também têm complexidade de tempo $O(v)$. Além

disso, o método Limpa possui complexidade de tempo $O(v + e)$, já que tem de percorrer a lista de vértices e as listas de arestas.

7. **Classe Armazem:** o TAD Armazem tem como principais funções armazenar informações sobre armazéns e operar sobre as respectivas seções, o que corresponde a operar sobre pilhas. A complexidade de espaço da classe é de $O(n + nroAdj)$, em que n é a quantidade de pacotes inseridos no sistema (corresponde ao tamanho máximo possível para uma Pilha) e nroAdj é o número de vizinhos que um determinado armazém possui. Sendo v mais uma vez o número de armazéns no sistema (e vértices no grafo), a complexidade de espaço é $O(n * v)$. Os principais métodos da classe, carregaTransporte, retornaPrincipal, esvaziaPrincipal, armazenaPacote, e tamSecaoPrincipal, tamSecaoAux, checaVazia e checaVaziaPrincipal, basicamente chamam métodos da classe Pilha (empilha, desempilha, vazia, todos de complexidade $O(1)$) em uma seção de interesse, de forma que possuem complexidade de tempo $O(1)$. O método encontraSecao, por sua vez, tem de iterar pelo vetor de seções da classe, de tamanho máximo v (número total de armazéns), em busca de uma seção com um certo destino, produzindo uma complexidade de tempo $O(v)$.
8. **Classe Pacote:** o método da classe que extrapola os já citados (getters e setters) é getProximoRota, que obtém o próximo vértice na rota de transporte do pacote, que é representada por uma Lista Encadeada. Se o tamanho da rota do pacote é L, temos uma complexidade de tempo $O(L)$. Como L pode ser no máximo v-1, com v o número de armazéns no sistema, temos $O(v)$. A complexidade de espaço do uso de um TAD Pacote é também $O(v)$, vide o armazenamento da rota em sua estrutura.
9. **Classe Evento:** os métodos da classe Evento são getters, setters ou lidam com a geração e decodificação de chaves para diferentes tipos de eventos. O processamento das strings é in-place e não requer que a string seja percorrida, de forma que os métodos construirChavePacote, construirChaveTransporte, decodificarTipoEvento, decodificarTempoEvento, decodificarIdPacote, decodificarOrigemTransporte, decodificarDestinoTransporte todos apresentam complexidade tanto de tempo quanto de espaço $O(1)$.
10. **Classe Escalonador:** a classe Escalonador foi implementada como um minHeap que tem como elementos inseridos ponteiros para Eventos. O critério de comparação utilizado na estrutura do Heap é a chave identificadora dos eventos. A estrutura do Heap possui um tamanho máximo fixo sobre o qual operamos e é independente de fatores da entrada, com complexidade de espaço $O(1)$. Seja m o tamanho real do Heap, ou seja, a quantidade de eventos nele presentes em um dado momento. Por construção da estrutura de dados Heap, o método Inserir possui complexidade de tempo $O(logm)$. O mesmo ocorre para o método Remover - apesar de o custo de remoção da raiz ser $O(1)$, o custo vai a logm com as operações de heapify subsequentes. Os métodos para encontrar elementos no Heap (GetAncestral, GetSucessorEsq e GetSucessorDir) apenas realizam operações aritméticas, com complexidade de tempo e espaço $O(1)$.
11. **Análise geral do sistema:** a complexidade de espaço do sistema como um todo é determinada pelo armazenamento da estrutura do grafo, pela quantidade de eventos no escalonador, número de pacotes e de armazéns com seções. Seja v o número de armazéns (vértices no grafo), e o número de arestas no grafo (conexões entre armazéns) e n o número de pacotes inseridos inicialmente. A estrutura do grafo tem complexidade $O(v + e)$. Para o escalonador, no pior caso, teremos uma complexidade $O(n + e)$ - quando há, simultaneamente, eventos ativos para cada pacote e eventos de transporte para cada conexão possível entre armazéns. Para cada pacote, também armazenamos uma Lista Encadeada de tamanho máximo v (temos de passar por todos os armazéns no sistema), gerando uma complexidade de pior caso de $O(n * v)$. Por fim, para um armazém, o número máximo de seções possíveis é v-1 (o armazém se conecta com cada um dos outros armazéns), e cada seção pode ter, no máximo, n pacotes (todos os pacotes do sistema simultaneamente em uma única seção), e temos v

armazéns. Assim, geramos uma complexidade de pior caso de $O(n + v^2)$. Com isso, como analisamos o pior caso para cada um dos TAD's em que ele existe, chegamos a uma complexidade de espaço total de $O(n * v + v^2 + e)$. Para a complexidade de tempo do programa principal, temos: leitura dos dados de entrada com custo $O(v^2)$ para a leitura da matriz de adjacência do grafo; leitura de pacotes com complexidade $O(n)$ e cálculo de rotas via busca em largura com complexidade $O(n * (v + e))$; inicialização dos primeiros eventos de chegada e de transporte com complexidade $O((n + e) * logm)$, em que m é o tamanho do heap - inserimos n+e eventos com custo logm para inserção e construção do heap. Além disso, para o loop principal da simulação, temos: retirada de eventos do heap com custo $O(logm)$; processamento de eventos - $O(n * v)$ no pior caso para os eventos de pacote (no pior caso temos custo $O(v)$ para o método encontraSecao e getProximoRota, gerando custo n*v se cada pacote tem de passar por cada armazém) e $O(e * t)$ para transportes, em que t é o número total de vezes em que o sistema agenda eventos de transporte. Com isso, tem-se uma complexidade de tempo, no pior caso, de $O(v^2 + e + n + n * (v + e) + (p * v + e * t) * logm)$.

4. Estratégias de Robustez

Para garantir a robustez do sistema, a principal estratégia empregada foi o lançamento de exceções, utilizando o método de try-throw-catch disponibilizado pela linguagem C++. Em especial, foi empregado em main.cpp para indicar possíveis erros na leitura de dados do arquivo de entrada (por exemplo, se as informações não estavam no formato esperado e, portanto, geraram problemas na leitura), ou mesmo caso o arquivo de entrada necessário não tivesse sido passado como parâmetro de execução. Assim, caso ocorram erros nessa parte do sistema, eles são devidamente identificados, não são propagados e não geram comportamentos inesperados no sistema posteriormente. Além disso, também foi utilizado tratamento de exceções em métodos que tratam de ponteiros (como ao acessar os ponteiros head de lista encadeada, pilha ou fila) e alocação dinâmica de memória, visando a identificar momentos em que poderia ocorrer acesso indevido a regiões de memória. Também foi empregada em métodos em que poderia ocorrer busca em posições de memória inválidas, como em encontraSecao, caso a seção desejada não estivesse presente no vetor de seções do armazém, ou em Fila ou Pilha, caso tentássemos remover itens enquanto a estrutura se encontra vazia.

Por fim, também foi realizada a modularização das classes, mantendo os seus atributos privados e, portanto, inacessíveis a alterações diretas e possivelmente maliciosas por parte dos usuários.

5. Análise Experimental

5.1 Método e Resultados

Para realizar a análise experimental, inicialmente foram fixadas algumas configurações padrão para o sistema, de forma que variamos um dos parâmetros por vez para analisar como cada um deles impacta o seu funcionamento. Os resultados analisados foram o tempo de simulação, que representa o tempo de término da simulação no relógio do sistema, e o tempo de execução, que representa o tempo real de execução do programa, em segundos. Para obter os tempos de execução, como esse fator apresenta diferentes resultados em diferentes execuções, foi utilizado como valor final a média do tempo obtido em 10 execuções do sistema. As entradas para o sistema foram geradas pelo código fornecido pelos professores.

O primeiro fator analisado foi o número de armazéns. Na Tabela 1, apresentamos os resultados obtidos com a variação do número de armazéns no sistema. Para facilitar a visualização dos dados, também apresentamos os Gráficos 1 e 2, com as relações entre esse fator, o tempo de execução e o de simulação.

nro. pacotes	nro. armazéns	latência transp.	intervalo transp.	capacidade transp.	tempo simulaçã	tempo execução
100	25	20	100	2	2124	0,03738256
100	50	20	100	2	2031	0,06142654
100	75	20	100	2	2622	0,07681778
100	100	20	100	2	2730	0,08114992
100	125	20	100	2	3027	0,09713028
100	150	20	100	2	2523	0,10145
100	175	20	100	2	3623	0,134619
100	200	20	100	2	3923	0,1452652

Tabela 1. Resultados com variação no número de armazéns

Gráfico 1. Armazéns e execução

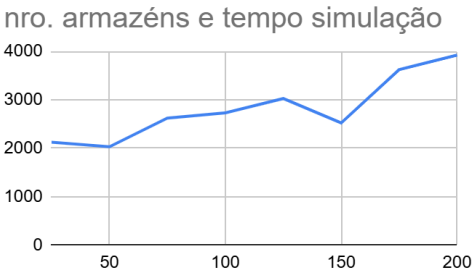


Gráfico 2. Armazéns e simulação.

Em seguida, foi variada a quantidade de pacotes inseridos no sistema. Os resultados obtidos são apresentados na Tabela 2 e nos gráficos 3 e 4, com a relação entre esse fator e os tempos de execução e simulação, respectivamente.

nro. pacotes	nro. armazéns	latência transp.	intervalo transp.	capacidade transp.	tempo simulação	tempo execução
25	20	20	100	2	1025	0,010207466
50	20	20	100	2	1125	0,01509644
75	20	20	100	2	1525	0,0244378
100	20	20	100	2	1925	0,03650872
125	20	20	100	2	2225	0,07012498
150	20	20	100	2	2525	0,0780543
175	20	20	100	2	3025	0,10432072
200	20	20	100	2	3425	0,11537868

Tabela 2. Resultados obtidos com a variação da quantidade de pacotes no sistema.

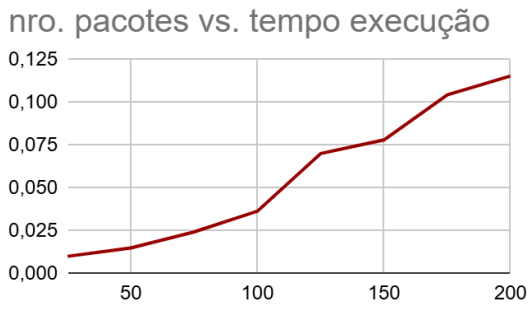


Gráfico 3. Pacotes e execução.

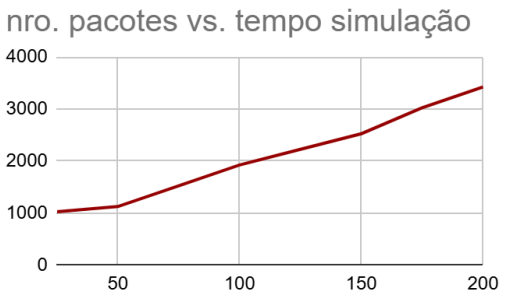


Gráfico 4. Pacotes e simulação.

Por fim, para observar alterações na quantidade de rearmazenamentos e constrição do sistema em geral, variamos separadamente os parâmetros de latência de transporte, intervalo entre transportes e capacidade de transporte. Os resultados são apresentados na Tabela 3 e nos

gráficos 5, 6, 7, 8, 9 e 10, que representam as relações de cada parâmetro com os fatores tempo de execução e tempo de simulação.

nro. pacotes	nro. armazens	latência transp.	intervalo tarnsp	capacidade transp.	tempo simulação	tempo execução
50	20	20	50	2	625	0,013980352
50	20	20	100	2	1125	0,01405528
50	20	20	150	2	1675	0,01686312
50	20	20	200	2	2225	0,01840662
50	20	20	100	1	1725	0,0208245
50	20	20	100	5	926	0,016033929
50	20	20	100	10	926	0,0162397
50	20	20	100	15	926	0,01222203
50	20	20	100	2	1125	0,013249108
50	20	80	100	2	1185	0,016623878
50	20	160	100	2	1965	0,020635
50	20	640	100	2	6345	0,02367074

Tabela 3. Resultados obtidos com a variação de parâmetros relacionados ao número de rearmazenamentos.

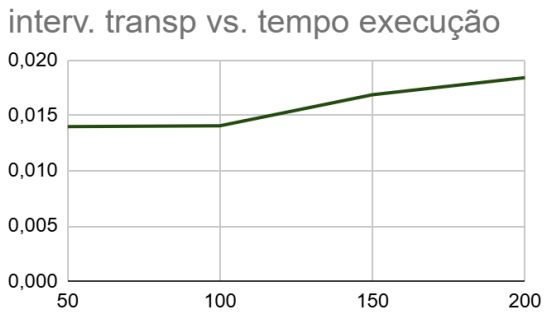


Gráfico 5. interv. transp. e execução.

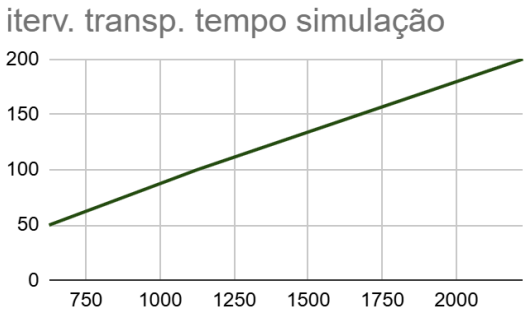


Gráfico 6. interv. transp. e simulação.

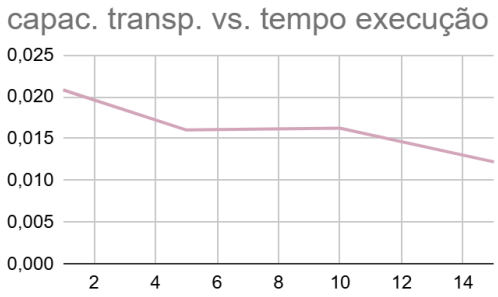


Gráfico 7. capac. transp. e execução.

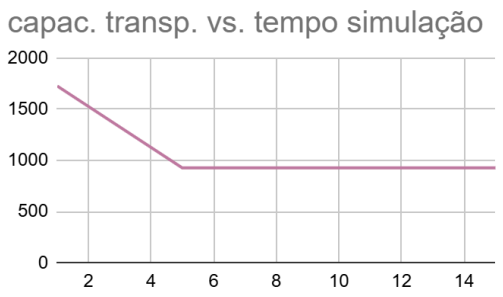
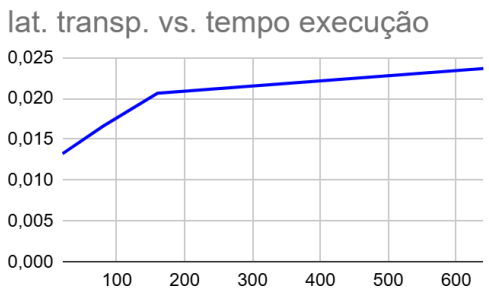


Gráfico 8. capac. transp. e simulação.



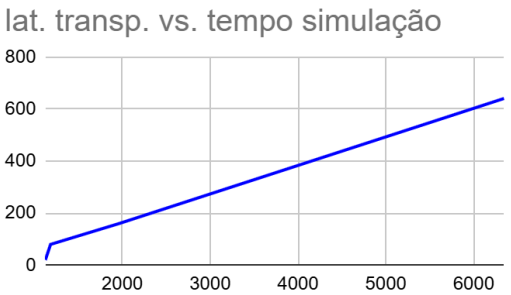


Gráfico 9 e 10. lat. transp., execução e simulação.

.

5.2. Conclusões

Temos que o aumento no número de armazéns no sistema gera aumento no tempo de execução. Este resultado é esperado uma vez que, com uma quantidade maior de armazéns (e, portanto, de vértices no grafo), há mais possibilidades de rotas para pacotes (o que aumenta o custo de operações como a busca em largura sobre o grafo) e, dependendo da topografia do grafo, essas rotas podem se tornar mais longas, o que gera um tempo de permanência maior para os pacotes do sistema e a necessidade por mais eventos, tanto de chegada quanto de transporte. Apesar disso, é interessante notar que o comportamento do tempo de simulação em relação a esse fator é mais anômalo, podendo diminuir com o seu aumento. Esse resultado também é esperado, já que o funcionamento do sistema depende de uma combinação de eventos, em que a quantidade de armazéns tem impacto menos direto do que fatores como capacidade e latência de transporte, por exemplo.

Para o número de pacotes, notamos que os tempos de execução e de simulação aumentam à medida que essa quantidade cresce. Este resultado também é correto, uma vez que a presença de mais pacotes gera mais operações de todos os tipos: temos de aplicar mais buscas em largura para determinar rotas, há geração de uma quantidade maior de eventos (de transporte e de chegada), mais movimentações de itens (tanto no escalonador/heap quanto nas estruturas de pilha e fila), além de que o sistema tende a ficar mais "congestionado" (em termos de possíveis rearmazenamentos e distribuição de pacotes em armazéns em geral). Interessante notar que o comportamento do tempo de simulação é quase linear.

Para o intervalo entre transportes, notamos que o seu impacto sobre o tempo de execução não é tão grande, mas gera grandes impactos sobre o tempo de simulação, com comportamento quase linear. Esse resultado faz sentido, uma vez que as operações em que o utilizamos são basicamente aritméticas para determinação do relógio do sistema, de forma que não são impactantes sobre a execução mas determinam diretamente como o tempo corre na simulação em si.

Para a capacidade de transporte, notamos que, como esperado, quanto maior o seu valor, menores são os tempos de execução e de simulação. Isso ocorre porque podemos transportar mais pacotes por vez, reduzindo o congestionamento do sistema (há menos rearmazenamentos e possivelmente menos pacotes por armazém). Apesar disso, notamos que há um "ponto de saturação" para esse valor: o aumento gera redução no tempo de simulação mas, a partir de um certo ponto, ocorre uma estabilização nos valores, o que é pertinente - a quantidade de pacotes permanece constante e a capacidade de transporte pode se tornar superior ao valor máximo de pacotes por seção em um armazém, de forma que, independente desse valor, o transporte passa o ocorrer da mesma forma.

Por fim, notamos que, para a latência de transporte, o impacto é grande sobre o tempo de simulação e discreto para o tempo de execução. Assim como para o custo de transporte, esse resultado também faz sentido - as operações realizadas com esse parâmetro são basicamente aritméticas e devem ter pouco impacto no custo de execução, mas determinam diretamente como o tempo corre dentro da simulação.

6. Conclusões

Este trabalho lidou com o problema de implementar o Sistema de Escalonamento Logístico da empresa "Armazéns Hanói", cujo sistema de armazenamento é inspirado na Torre de Hanói. Os principais objetivos a serem atingidos com o sistema implementado foram: armazenar e manipular pacotes no sistema (registrando informações sobre sua chegada e movimentação entre entidades do sistema), definir rotas ótimas para cada pacote (utilizando a técnica de busca em largura em grafo) e desenvolver a lógica para definição do sistema de escalonamento.

Durante o desenvolvimento do trabalho, foi possível exercitar, em especial, conceitos relacionados a estruturas de dados, especialmente para as principais estruturas utilizadas: listas encadeadas, pilhas, filas e heap. Foi possível praticar a implementação dessas estruturas em um contexto mais próximo do mundo real, uma vez que os elementos armazenados e manipulados por elas são, em geral, os demais TAD's implementados, que são estruturas mais

complexas e representativas de elementos reais do sistema (como pacotes, armazéns e eventos).

Importantes desafios foram superados ao longo do processo, como: compreensão e implementação de uma simulação discreta de eventos, que era um conceito até então estrangeiro; modelagem de TAD's e estruturas de dados de acordo com as necessidades específicas do sistema e as estruturas reais que queríamos representar; e elaboração de estratégias para realização da análise experimental.

7. Bibliografia

1. Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.

2. Meira Jr, Wagner. e Lacerda, Anisio. (2025). Aulas e slides da disciplina Estruturas de Dados. DCC, UFMG.

3. Meira Jr, Wagner. e Lacerda, Anisio. (2025). Práticas Avaliativas 2 e 3 da disciplina Estruturas de Dados. DCC, UFMG.

4. Especificação do Trabalho Prático 2, Sistema de Escalonamento Logístico. DCC, UFMG.

5. Filho, Paulo Feofiloff. "Busca em Largura (BFS)." *Algoritmos para Grafos*. Instituto de Matemática e Estatística - USP. Accessed June 22, 2025. https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html.