Trabalho Prático de Estruturas de Dados: Sistema de Escalonamento Logístico

Vinicius de Alcantara Garrido
Departamento de Ciência da Computação – UFMG
viniciusalcantara115@ufmg.br

Junho de 2025

1. Introdução

Esta documentação lida com o problema de um sistema de escalonamento para uma empresa de logística, que envolve o gerenciamento de pacotes em múltiplos armazéns e o transporte entre eles. O sistema opera com base em uma simulação de eventos discretos, onde cada evento (chegada de pacote, transporte) ocorre em um instante de tempo específico.

O objetivo deste trabalho é implementar um simulador capaz de processar uma série de eventos logísticos a partir de um arquivo de entrada, gerenciando o estado dos pacotes e armazéns e gerando um log detalhado das operações. A simulação deve respeitar as regras de negócio, como a capacidade dos armazéns e veículos de transporte, e a política de manuseio de pacotes (LIFO - Last-In, First-Out).

Para resolver o problema, foi desenvolvida uma solução em C++ que utiliza uma arquitetura orientada a objetos e estruturas de dados customizadas. A abordagem central é um laço de simulação que processa eventos ordenados por tempo, utilizando uma fila de prioridade. As estruturas `Pilha` e `VetorDinamico` foram implementadas para gerenciar o armazenamento e a manipulação dos pacotes.

2. Método

2.1. Estruturas de Dados e Classes

- **`Simulacao`**: Classe central que orquestra a simulação, contendo o grafo da rede, os armazéns e a fila de eventos.
- **`Armazem`**: Contém múltiplas seções de saída (uma para cada destino), onde cada seção é uma pilha (`Pilha<Pacote>`) para garantir a política LIFO.
- **`Evento`**: Classe base para os eventos. Os tipos são `CHEGADA` e `TRANSPORTE_PACOTES`.

- `FilaDePrioridade<Evento*>`: Um heap que ordena os eventos, garantindo que o próximo a ser processado seja sempre o de maior prioridade.

2.2. Lógica da Simulação

O núcleo da simulação é um laço que executa os seguintes passos:

1. **Carregamento (`Simulacao::carregar_dados`)**: Lê o arquivo de entrada para inicializar o grafo, armazéns, pacotes e eventos iniciais.
2. **Execução (`Simulacao::executar`)**: Em cada iteração, o evento com maior prioridade é removido da fila e processado de acordo com seu tipo:
   - `processar_evento_chegada`: Um pacote chega e é empilhado na seção correspondente ao seu destino.
   - `processar_evento_transporte`: Aciona a lógica de transporte, que envolve roteamento (via BFS), seleção de pacotes e agendamento de novos eventos.

3. Análise de Complexidade

A complexidade do simulador é determinada pela interação entre as estruturas de dados, a lógica de processamento de eventos e os algoritmos de grafos para roteamento.

Seja `E` o número total de eventos na simulação, `N` o número de pacotes, `A` o número de armazéns, `R` o número de rotas no grafo, e `C` a capacidade de uma seção de armazém.

3.1. Estruturas de Dados e Algoritmos

- `FilaDePrioridade` **(Heap)**: Armazena os eventos. As operações de inserção (`insere_evento`) e remoção (`proximo_evento`) têm complexidade **O(log E)**.
- `Pilha` **e** `VetorDinamico`: Usadas nos armazéns. Operações de inserção e remoção no final (`empilha`, `desempilha`) têm custo **O(1)** amortizado.
- **Grafo e Busca em Largura (BFS)**: O sistema de rotas é modelado como um grafo. Para determinar o próximo armazém na rota de um pacote, uma busca em largura (BFS) é executada. A complexidade da BFS é **O(A + R)**.

3.2. Lógica de Processamento de Eventos

O motor da simulação é um laço que processa eventos até que a fila de prioridade esteja vazia.

- `processar_evento_chegada`: Um pacote chega e é empilhado no armazém. Esta é uma operação de complexidade **O(1)**.
- `processar_evento_transporte`: Este é o evento mais complexo.

  - **Roteamento**: Determina a rota para os pacotes, potencialmente envolvendo a execução da BFS, com custo **O(A + R)**.
  - **Manuseio de Pacotes**: Desempilha, seleciona e reempilha pacotes, com um custo proporcional à capacidade da seção, **O(C)**.
  - **Criação de Novos Eventos**: Agenda novos eventos de chegada e, possivelmente, de transporte, inserindo-os na fila de prioridade com custo **O(log E)**.
  - A complexidade combinada de um único evento de transporte é **O(A + R + C + log E)**.

3.3. Complexidade Geral

- **Tempo**: O laço principal executa `E` vezes. A complexidade de cada iteração é dominada pelo evento de transporte. Portanto, a complexidade de tempo total do simulador é **O(E * (A + R + C + log E))**.
- **Espaço**: A memória é consumida por três componentes principais:
  - A fila de eventos: **O(E)**
  - O armazenamento de pacotes nos armazéns: **O(N)**
  - A representação do grafo (matriz de adjacência): **O(A²)**
  - A complexidade de espaço total é **O(N + E + A²)**.

4. Estratégias de Robustez

Para garantir a correção e a estabilidade do simulador, foram adotadas as seguintes estratégias de programação defensiva:

- **Gerenciamento de Memória**: A classe `Pilha` (e seu `VetorDinamico` subjacente) implementa um construtor de cópia profunda e um operador de atribuição. Isso previne erros de *shallow copy*, onde múltiplas pilhas poderiam compartilhar ou corromper os mesmos dados, um bug crítico que foi identificado e corrigido durante o desenvolvimento.
- **Execução Determinística**: O sistema de chaves de prioridade para eventos garante que, para uma mesma entrada, a saída seja sempre idêntica. A chave (`long long`) ordena os eventos por tempo, tipo e IDs de entidade, eliminando ambiguidades e tornando a simulação totalmente reprodutível e testável.
- **Validação de Entradas**: Embora a validação robusta de arquivos de entrada não tenha sido o foco principal, o script de teste `testar_saida.sh` verifica a existência dos arquivos de entrada e referência antes da execução, prevenindo erros de tempo de execução por arquivos ausentes.

5. Análise Experimental

Para avaliar o desempenho prático da implementação, foi conduzida uma análise experimental medindo o tempo de simulação e o número de rearmazenamentos em

função da variação de parâmetros-chave do sistema. A metodologia empregada consistiu em:

1. **Configuração de Parâmetros Base**: Foram estabelecidos valores de referência para todos os parâmetros do sistema, mantendo-os constantes durante cada análise individual.
2. **Variação de Parâmetros**: Para cada análise, um único parâmetro foi variado enquanto os demais permaneciam em seus valores base.
3. **Automação**: Um script em Python foi desenvolvido para executar automaticamente as simulações, variando os parâmetros e gerando entradas aleatórias para cada simulação.
4. **Coleta de Dados**: A saída do algoritmo foi analisada a partir do mesmo script python para extrair:
   - Tempo total de simulação (obtido do último evento processado)
   - Número total de rearmazenamentos (contabilizados na saída do programa)

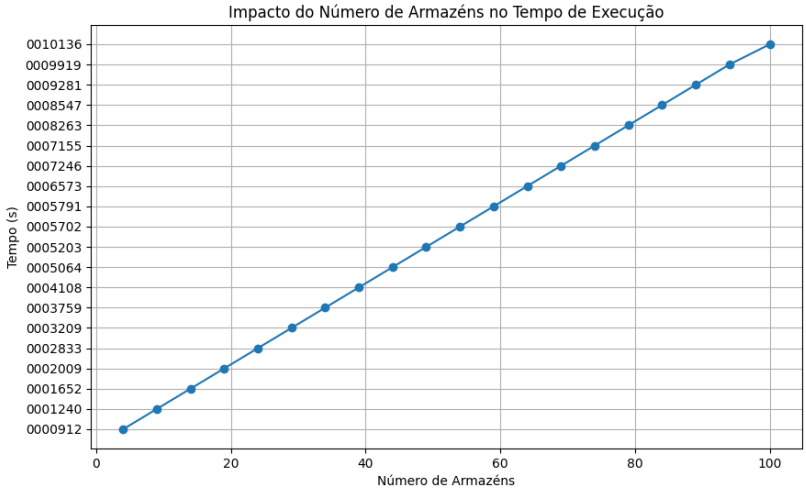
Parâmetros Base Utilizados

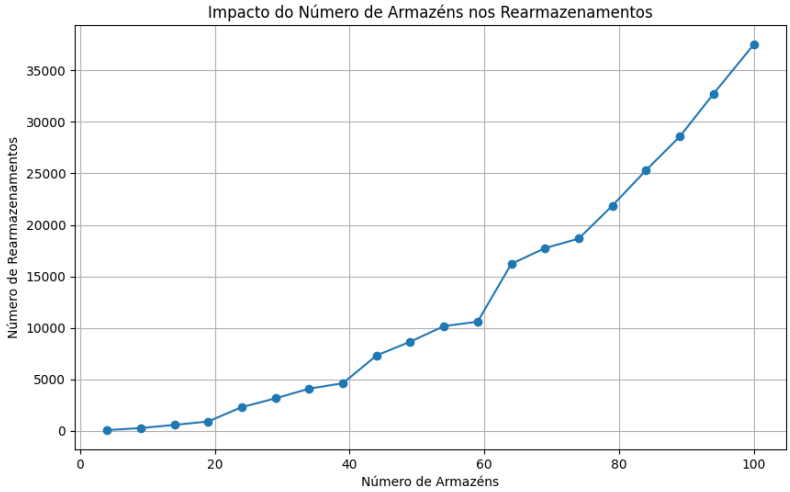
Parâmetro	Valor
Capacidade de Transporte	2
Latência de Transporte	20
Intervalo de Transporte	100
Custo de Remoção	1
Número de Armazéns	4
Número de Pacotes	50

5.1. Análise do Número de Armazéns

ID	Armazéns	Tempo (s)	Rearmazenamentos
0	4	0.841	84
1	9	1.246	305
2	14	1.619	516
3	19	2.113	840
4	24	2.639	2,353
5	29	3.196	2,963
6	34	3.655	3,990
7	39	4.026	4,653
8	44	4.854	7,387
9	49	5.135	8,585
10	54	5.699	10,759
11	59	5.843	11,330
12	64	7.099	15,322
13	69	7.074	16,582

ID	Armazéns	Tempo (s)	Rearmazenamentos
14	74	7.290	18,230
15	79	7.733	22,200
16	84	8.535	24,252
17	89	9.114	28,681
18	94	9.505	30,339
19	100	10.022	36,210



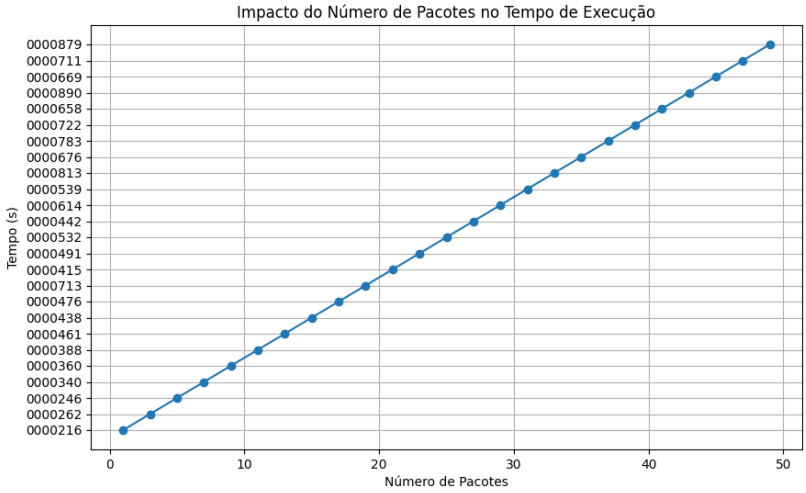


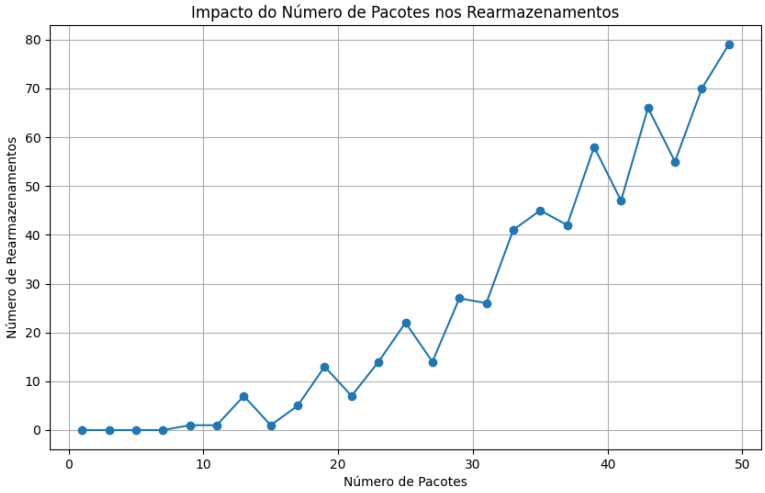
Conclusão: O tempo de simulação aumenta de forma aproximadamente linear

com o número de armazéns, enquanto o número de rearmazenamentos cresce exponencialmente, indicando maior complexidade logística.

5.2. Análise do Número de Pacotes

ID	Pacotes	Tempo (s)	Rearmazenamentos
0	1	0.216	0
1	3	0.262	0
2	5	0.246	0
3	7	0.340	0
4	9	0.360	1
5	11	0.388	1
6	13	0.461	7
7	15	0.438	1
8	17	0.476	5
9	19	0.713	13
10	21	0.415	7
11	23	0.491	14
12	25	0.532	22
13	27	0.442	14
14	29	0.614	27
15	31	0.539	26
16	33	0.813	41
17	35	0.676	45
18	37	0.783	42
19	39	0.722	58
20	41	0.658	47
21	43	0.890	66
22	45	0.669	55
23	47	0.711	70
24	49	0.879	79



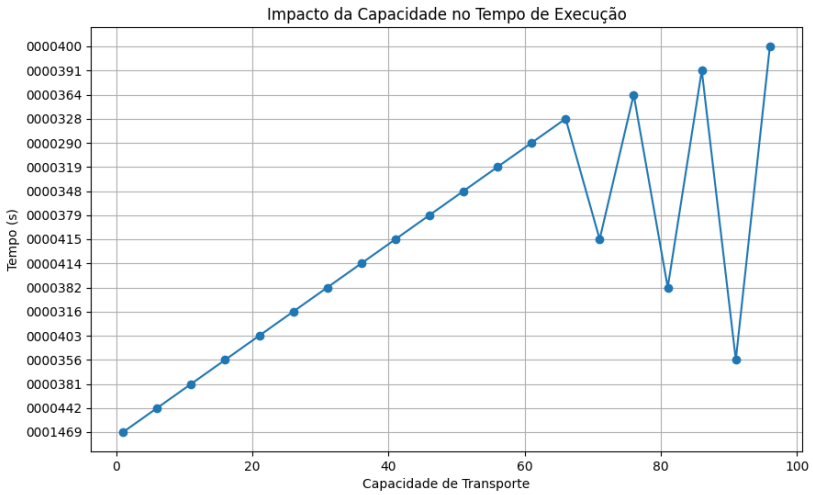


Conclusão: O aumento do número de pacotes leva a um crescimento acentuado em ambos os indicadores, devido à maior competição por recursos. No caso, o tempo de simulação aumenta linearmente, enquanto o número de rearmazenamentos cresce de forma exponencial.

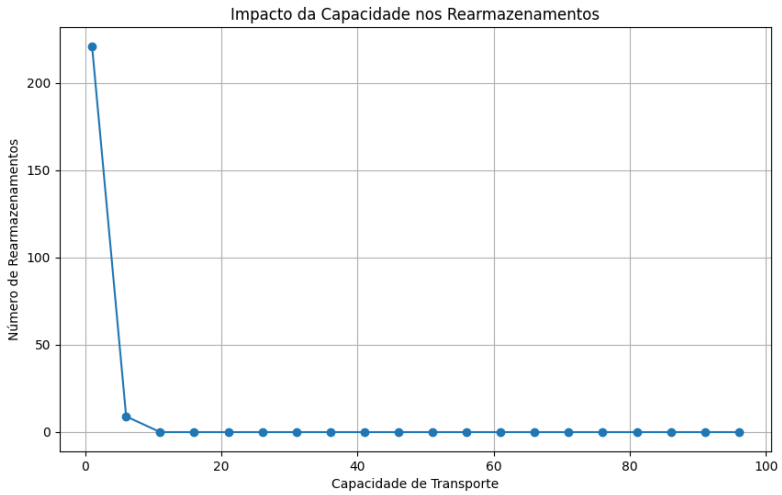
5.3. Análise da Capacidade de Transporte

ID	Capacidade	Tempo (s)	Rearmazenamentos
0	1	1.644	202

ID	Capacidade	Tempo (s)	Rearmazenamentos
1	6	0.477	16
2	11	0.507	3
3	16	0.376	0
4	21	0.388	0
5	26	0.336	0
6	31	0.307	0
7	36	0.372	0
8	41	0.354	0
9	46	0.314	0
10	51	0.366	0
11	56	0.379	0
12	61	0.409	0
13	66	0.390	0
14	71	0.344	0
15	76	0.309	0
16	81	0.407	0
17	86	0.370	0
18	91	0.399	0
19	96	0.344	0



obs: O eixo y(Tempo(s)) está invertido.



Conclusão: Aumentar a capacidade de transporte reduz drasticamente o número de rearmazenamentos e o tempo de simulação já que são necessarias menos viagens para transportar o mesmo número de pacotes.

6. Conclusões

Este trabalho resultou no desenvolvimento de um simulador funcional para um sistema de escalonamento logístico. A utilização de uma simulação baseada em eventos, juntamente com estruturas de dados eficientes, permitiu modelar o problema de forma precisa e determinística.

O principal aprendizado foi a importância de um gerenciamento de memória cuidadoso em C++, especialmente a necessidade de cópias profundas para garantir a integridade dos dados entre objetos. A depuração de um bug de *shallow copy* na classe `Pilha` reforçou a necessidade de testes rigorosos e de uma compreensão clara do ciclo de vida dos objetos. Além disso, a implementação do sistema de prioridade de eventos foi fundamental para garantir a reprodutibilidade dos resultados, uma característica essencial para qualquer simulador confiável.

7. Referências

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