Trabalho Prático 2 - Sistema de Escalonamento Logístico

Nome: Miguel Bertussi Carneiro Moreira
Matrícula: 2024005483

1. Introdução

A empresa vietnamita Armazéns Hanoi, parte do grupo que inventou a Torre de Hanoi, está automatizando o seu antigo sistema logístico de entrega de pacotes, que recebe pacotes e os distribui entre os diversos armazéns até que cheguem ao seu destino. O novo sistema a ser implementado deve ser capaz de transportar, armazenar e acompanhar os pacotes durante estas entregas.

O problema claramente tem parte de sua estrutura baseada em um grafo inicial de armazéns, onde cada armazém pode ser visto como um vértice e a ligação entre dois armazéns distintos uma aresta. A fim de calcular a menor rota possível, um algoritmo de **busca em largura** (**BFS**) será implementado neste grafo. Além do grafo, temos que definir alguma ordem para os possíveis eventos de transporte e armazenamento, então, um **heap** será implementado a fim de atingir este objetivo, funcionando como uma **fila de prioridade**.

A coordenação entre estas estruturas e algoritmos possibilitará a otimização do sistema de transporte de pacotes, e durante a execução, contêineres auxiliares como **vector** e **stack** serão de grande importância para o registro e armazenamento do estado dos pacotes a todo momento.

2. Implementação

O código foi inteiramente projetado e implementado na linguagem C++, visando principalmente a modularização através das classes e o aspecto multi-uso do **template**. A estrutura foi feita em cinco arquivos de estruturas de dados (**vector**, **queue**, **heap**, **stack**, **eventandparameters**) e quatro de domínio (**package**, **warehouse**, **scheduler**, **simulator**). Nesta seção, descreveremos as classes e seus respectivos métodos.

2.1 Vector

A classe **vector** é uma implementação própria de um vetor dinâmico genérico, que pode armazenar elementos de qualquer tipo **T**. Ela é semelhante a **std::vector**, permitindo inserção dinâmica de elementos, acesso por índice e cópia profunda entre vetores.

Atributos

- **T* data**: *array* simples do tipo **T**.
- **int _capacity**: capacity atual alocada do vetor.
- **int _size**: quantidade atual de elementos armazenados no vetor.

Métodos e Operadores Sobrecarregados

- **vector():** inicializa o **vector** com **_capacity** igual a um e **_size** zero.
- **vector(int capacity)**: inicializa o **vector** com a **capacity** desejada e **_size** zero.
- **vector(const vector& other)**: construtor de cópia que aloca nova memória e copia todos os elementos do **vector** original.

- **~vector()**: destrutor responsável por desalocar a memória utilizada pelo **vector**.
- **void push_back(const T& element):** insere elemento no final do **vector**. Caso o **vector** esteja cheio, **expande** sua **_capacity** antes da inserção.
- **void clear():** reseta o **_size** do **vector** para 0, mantendo a **_capacity**. Os elementos são perdidos, mas a memória permanece alocada.
- **int size() const:** retorna quanto elementos estão no **vector**, **_size**.
- **void expand():** dobra a **_capacity** do **vector** alocando nova memória, copiando os elementos antigos e liberando a memória anterior.
- **T& operator[](int index)**: retorna uma referência ao elemento da posição **index**, permite leitura e escrita.
- **const T& operator[](int index) const**: versão constante do operador acima, permite apenas leitura.
- **vector<T>& operator=(const vector<T>& outro)**: realiza cópia profunda dos elementos do **vector outro**, substituindo os dados do **vector** atual.

2.2 Stack

A classe **stack** implementa uma estrutura de dados de pilha (último a entrar, primeiro a sair - LIFO) genérica, utilizando a classe **vector** como base para o armazenamento dos dados.

Atributos

- **vector<T> data**: **vector** que armazena os elementos da pilha.

Métodos e Operadores Sobrecarregados

- **stack()**: construtor padrão que cria uma pilha vazia.
- **stack(const stack<T>& other)**: construtor de cópia que cria uma nova pilha com os mesmos elementos da pilha fornecida.
- **~stack()**: destrutor padrão, que delega a liberação de memória para o destrutor do **vector** interno.
- **void push(const T& element)**: insere um elemento no topo da pilha.
- **void pop()**: remove o elemento do topo da pilha.
- **T& top()**: retorna uma referência ao elemento no topo da pilha.
- **bool empty() const**: verifica se a pilha está vazia.
- **int size() const**: retorna a quantidade de elementos na pilha.
- **stack<T>& operator=(const stack<T>& other)**: permite a atribuição de uma pilha a outra, realizando uma cópia completa dos elementos.

2.3 Queue

A classe queue implementa uma estrutura de dados de fila (first-in, first-out - FIFO) genérica. Ela utiliza a classe **vector** como estrutura de dados para armazenar seus elementos.

Atributos

- **vector<T> data**: **vector** utilizado internamente para armazenar os elementos da fila.

- **int start**: armazena o index de começo da fila.

Métodos

- **queue()**: construtor padrão que inicializa uma fila vazia.
- **~queue()**: destrutor padrão, a desalocação da memória é gerenciada pelo destrutor da classe **vector**.
- **void rearrange()**: método privado auxiliar que desloca todos os elementos do vector uma posição para a esquerda.
- **void push(const T& element)**: adiciona um elemento ao final da fila.
- **void pop()**: remove o primeiro elemento da fila, dando incrementando **start** em um.
- **const T& front()**: retorna uma referência constante ao primeiro elemento da fila.
- **bool empty() const**: verifica se a fila está vazia, retornando **true** se **_size** de **data** for 0.

2.4 Heap

A classe **heap** implementa uma estrutura de dados de ***min heap*** genérica. Utilizando de um **vector** para realizar o armazenamento de dados.

Atributos

- **vector<T> data**: **vector** usado para armazenar os elementos do **heap**.

Métodos

- **heap()**: construtor padrão que inicializa um heap vazio.
- **~heap()**: destrutor padrão, a memória é gerenciada pelo **vector data** de atributos.
- **void push(T element)**: insere um novo elemento no heap e o reposiciona usando **heapifyUp** para manter a propriedade de min heap.
- **void pop()**: remove o elemento da raiz, substitui pelo último elemento do heap e reestrutura a árvore usando **heapifyDown** para restaurar a propriedade de **min heap**.
- **const T& top() const**: retorna uma referência constante do elemento na raiz do **heap**.
- **bool empty() const**: verifica se o heap está vazio.
- **void heapifyUp(int index)**: método privado que move um elemento na árvore até que a propriedade de min heap seja satisfeita.
- **void heapifyDown(int index)**: método privado que move um elemento "para baixo" na árvore até que a propriedade de min heap seja satisfeita.
- **int getAncestor(int index)**: método privado que retorna o índice do nó pai.
- **int getSuccessorRight(int index)**: método privado que retorna o índice do filho à direita.
- **int getSuccessorLeft(int index)**: método privado que retorna o índice do filho à esquerda..

2.5 Event and Parameters

Este arquivo armazena dois tipos de dados distintos, **events** e **parameters**, os **events** serão gerados durante a execução do código, enquanto os **parameters** são lidos na entrada da main. Aqui, como os únicos métodos são os construtores,

2.5.1 Event

Atributos

.int time (instante de tempo do evento), **.type** (tipo do evento, 1 para armazenamento e 2 para transporte), **.packageID** (ID do **package** a ser transportado), **.origin** (armazém de partida), **.destination** (armazém de chegada).

.long long key (chave única do evento).

Métodos e Operadores Sobrecarregados

- **event(int _time, int _type, int _packageID)**: inicializa os atributos de nomes semelhantes aos parâmetros recebidos, associado a eventos de **armazenamento**.
- **event(int _time, int _type, int _origin, int _destination)**: inicializa os atributos de nomes semelhantes aos parâmetros recebidos, associado a eventos de **transporte**.
- **event()**: construtor padrão.
- **bool operator<(const event& other)**: retorna **key** < **other.key**.

2.5.2 Parameters

Atributos

.int transportCapacity (quantidade de pacotes que podem ser transportados de uma vez), **.int transportLatency** (duração do transporte entre dois armazéns), **.int transportInterval** (intervalo de tempo entre transportes), **.int removalCost** (custo de remoção de um pacote de um armazém), **.int numWarehouses** (número de armazéns), **.vector<vector<int>> warehouseSchema** (matriz de adjacência do grafo dos armazéns) , **.int numPackages** (número de pacotes a serem transportados)**.**

Métodos

- **parameters(int _transportCapacity, _transportLatency, _transportInterval, …)**: construtor padrão que inicializa todos os membros com os valores fornecidos (na mesma ordem da declaração dos atributos).

2.6 Algorithms

Este é um arquivo .hpp que possui funções genéricas auxiliares para o resto do código.

Funções

- **void swap(T& a, T& b)**: troca o valor de duas variáveis entre elas.
- **stack<T> BFS(vector<vector<T>>& graph, T originNode, T destinationNode)**: busca em largura no **graph**, que retorna na **stack** o menor caminho entre dois *nodes*.
- **int smallestElement(vector<T>& data, int& a, int& b, int& c)**: retorna qual das três posições **a**, **b**, e **c** está armazenando o menor elemento em **data**.

2.7 Package

A classe **package** encapsula todos os dados referentes aos pacotes a serem transportados, possibilitando o rastreio destas informações e atualização delas.

Atributos

- **int ID**: identificador único do pacote.
- **int origin**: armazém de origem do pacote.
- **int destination**: armazém de destino final do pacote.
- **int arrivalTime**: tempo em que o pacote chegou ao sistema (ao seu armazém de origem).
- **int timeStored**: tempo total que o pacote passou armazenado.
- **int timeTransported**: tempo total que o pacote passou em trânsito.
- **stack<int> route**: pilha que armazena a rota do pacote, o topo é a **origin**.
- **int session**: seção do armazém em que o pacote está armazenado.

Métodos e Operadores Sobrecarregados

- **package()**: construtor padrão.
- **package(int _ID, int _origin, int _destination, int _arrivalTime)**: construtor que inicializa um pacote com seus dados essenciais.
- **package(const package& other)**: construtor de cópia.
- **~package()**: destrutor padrão.
- **void advanceRoute()**: remove o topo da pilha de rota, avançando a posição do pacote.
- **void adjustSession()**: pega o próximo elemento da rota, caso não haja, **session** vira -1.
- **void setRoute(stack<int>& _route)**: recebe a rota calculada e a insere em **route**.
- **int getID()**, **getArrivalTime()**, **getOrigin()**, **getDestination()**, **getSession(), getTop(), getTimeTransported(), getTimeStored()**: métodos get para acessar os atributos privados do pacote.
- **bool isRouteEmpty() const**: verifica se a pilha de rota está vazia ou não.
- **package& operator=(const package& other)**: atribui os dados de um pacote a outro.

2.8 Warehouse

A classe **warehouse** é a responsável por armazenar os pacotes nas devidas seções, que são formadas por **pilhas** de pacotes.

Atributos

- **vector<stack<package>> sessions**: um **vector** que possui **stack**’**s** de **packages**, onde a seção que o pacote está, é o destino dele (não necessariamente o destino final).

Métodos

- **void addSession()**: adiciona uma nova seção (uma pilha de pacotes vazia) ao armazém.
- **void storePackage(package& pac)**: armazena um pacote na sessão correspondente ao atributo **session** do pacote **pac**.
- **package& removePackageFor(int destination)**: procura por pacotes na seção **destination**, os desempilha e retorna uma referência ao pacote removido.
- b**ool hasPackageFor(int destination)**:verifica se a pilha da seção **destination** está vazia.

2.9 Scheduler

Esta classe representa o coração do loop principal, o **escalonador**, gerenciando a fila de eventos pela ordem de inserção no **heap**.

Atributos

- **heap<event> eventScheduler**: um ***min heap*** que armazena objetos do tipo **event** junto à sobrecarga do operador < em **event** garantem a prioridade e o funcionamento adequados.

Métodos

- **void insertEvent(const event& _event)**: Insere um novo evento na fila de prioridade (o heap). O heap se reorganiza automaticamente para manter a ordem dos eventos.
- **event removeEvent()**: Remove e retorna o evento que está no topo do heap. Este é o próximo evento a ser processado na simulação.
- **bool empty() const**: Verifica se a fila de eventos está vazia. Retorna true se não houver mais eventos a serem processados.

2.10 Simulator

Esta classe é o motor central da simulação. É uma classe apenas de métodos, todos dados utilizados foram pré-preparados antes de começar a execução de suas funções, utilizando de um **scheduler** para gerenciar os eventos e coordenando as interações entre os **warehouses** e **packages**, além dos **parameters** da main.

(**all = scheduler& eventScheduler, vector<warehouse>& warehouses, vector<package>& packages, parameters& param).**

Métodos

- **void calculateAllRoutes(vector<package>& packages, parameters& param)**: faz o cálculo da rota de cada pacote de **packages** utilizando **BFS** e armazena esta no pacote.
- **int nextTranspTime(int time, parameters& param)**: calcula o próximo tempo de transporte agendado.
- **void keyGenerator(event& currentEvent)**: gera a **key** única do **event**.
- **void scheduleInitialStorageEvents(scheduler& eventScheduler, vector<package>& packages)**: agenda os eventos iniciais de armazenamento de pacotes nos seus armazéns de origem.
- **void processStorageEvent(event& currentEvent, all)**: processa um evento de armazenamento, colocando o pacote no armazém correto e avançando a rota, e agenda o próximo evento de transporte do pacote caso ele não tenha chegado em seu **destination**.
- **void processTransportEvent(event& currentEvent, all (-packages))**: processa um evento de transporte, movendo pacotes de um armazém de origem para um de destino e agendando os próximos eventos de armazenamento e de transporte se necessário.
- **void preSimulation(scheduler& eventScheduler, all)**: executa todas as configurações iniciais da simulação, como calcular rotas e agendar os primeiros eventos.
- **void runSimulation(scheduler& eventScheduler, all)**: Inicia e executa o loop principal da simulação, processando eventos da fila do scheduler até que ela esteja vazia.

3. Análise de Complexidade

Nesta seção, a complexidade de cada algoritmo será definida de forma precisa.

3.1 Vector, Stack, Queue

O funcionamento do **vector** permite apenas inserção no fim do vetor, o que custa em média O(1) tanto de tempo quanto de espaço, pois é apenas um acesso à índice. O pior caso ocorre quando o vetor está cheio e a expansão é necessária, neste caso o custo passa a ser O(n) espacial e temporalmente, pois um vetor com o dobro da capacidade é criado e todos os elementos passam do vetor menor para o maior, mas, como a expansão foi definida exponencialmente, este custo é amortizado e é, em média, O(1), pois a memória já estará alocada e apenas insere-se o elemento ao fim do vetor, o que é custo constante. A **stack** tem o custo médio O(1) para remoção e inserção, já que ambos acontecem apenas no topo, o pior caso ocorre quando a pilha está cheia e é necessário expandi-la, o que é O(n) temporal e espacialmente. A **queue** que foi implementada tem custo médio de inserção O(1), pior caso quando está cheia e precisa expandir O(n), e de remoção O(1), pois podemos apenas inserir normalmente no vetor e o contador **start** permite custo constante de remoção, basta somar um ao seu valor.

3.2 Heap e Scheduler

Como visto em aulas, o **heap** implementa uma árvore binária em um vetor, então as operações sobre ele são O(log(n)) temporalmente, para manter a propriedade base do **heap**, no caso de inserção e remoção, já que ele vai percorrer apenas um caminho da árvore. Quanto à complexidade espacial, ela será O(1), porque o **heap** não utiliza nenhum tipo de estrutura adicional. A complexidade do **scheduler** é a mesma do **heap**, já que seu único atributo é um **heap**.

3.3 Event and Parameters

Todas as operações são constantes, apenas atribuições. O(1) de tempo e espaço.

3.4 Algorithms

O custo de **swap** e **smallestElement** é constante, o primeiro apenas troca o valor de duas variáveis e o segundo faz acessos por índice a um vetor, ambos O(1) para tempo e espaço.

A BFS possui custo distinto, já que iremos percorrer, no pior caso O(V), onde V é o número de vértices e O(E), onde E é o número de arestas, todos os vértices do grafo, resultando em O(V + E) temporalmente. Espacialmente, temos que o pior caso é quando acessamos os V vértices, então temos O(V) para a **pilha**.

3.5 Warehouse

O custo para armazenar e remover pacotes é o custo de inserção e remoção da **stack**, O(1), com pior caso O(n), e isso vale tanto temporal quanto espacialmente. O custo de checar se

há um pacote ou não é sempre constante, apenas checa se a pilha está vazia, portanto O(1). Já o custo de adicionar seção, é o custo médio de adicionar elementos no **vector**, O(1) amortizado.

3.6 Simulator

Q = número de eventos, **P** = número de pacotes, **V** = número de armazéns, **C** = capacidade de transporte, **E** = número de arestas.

A complexidade do método **calculateAllRoutes(...)** é facilmente obtida com as que nós já conhecemos, sendo P o número de **pacotes**, teremos O(P * (V + E)), já que a rota é calculada para cada pacote. Para o método **scheduleInitialStorageEvents(...)**, para cada pacote P, iremos criar um evento associado, O(1), e inseri-lo no ***min heap*** **eventScheduler**, logo temos (P * log(P)).

Já para o método **processStorageEvent(...)**, temos O(1) para achar o **pacote** e O(1) para testar se a seção do armazém está vazia ou não, caso não esteja, agendamos o evento e inserimos no ***min heap*** o que custa O(log(Q)), onde Q é a quantidade de eventos no momento da inserção.

Para o método **processTransportEvent(...)**, temos o loop inicial (que se não acontecer, nada vai), que desempilha todos os P **pacotes** guardados em uma seção específica de algum **armazém**, O(P) temporal e espacialmente. Depois, para o C (capacidade de transporte) primeiros pacotes, iremos agendar eventos de **armazenamento** e vamos inseri-los no ***min heap***, então teremos O(C * log(Q)), e o último loop possui C-P iterações O(1), portanto, ao fim da função teremos O(P + C * log(Q)) temporalmente e O(P) espacialmente.

Agora que calculamos as complexidades auxiliares, podemos calcular os dois métodos principais, **preSimulation(...)** e **runSimulation(...)**. Para o primeiro, teremos **calculateAllRoutes(...) + scheduleInitialStoreEvents(...)** = O(P * (V + E) + P * log(P)). Já para o segundo, podemos considerar como pior caso todos eventos de **transporte**, já que ele é mais caro que o de **armazenamento**, então teremos O(Q * (P + C * log(Q)).

3.7 Complexidade Geral

Considerando o pior caso, temos o seguinte: todos os pacotes tem a mesma rota, a rota é a maior possível e cada transporte é feito de um em um, então teremos que percorrer V - 1 **armazéns** em V * P eventos. Para calcular esta complexidade, iremos somar a complexidade de **preSimulation(...)** e **runSimulation(...)** nos piores casos, além do custo de criar o grafo e de criar os pacotes, que são $O(V^2)$ e O(P):

$O(P * (V + E) + P * \log(P))$ e $O(V * P * (P + \log(P)))$.

$= O(P * V + P * E + P * \log(P))$ e $O(V * P^2 + V * P * \log(P)) = O(V * P^2)$.

$= O(V^2 + P * (V + E) + P * \log(P) + V * P^2) = O(V^2 + P * (V + E) + V * P^2)$. -> temporalmente.

Espacialmente, teremos $O(V^2 + P + P) = O(V^2 + P)$. -> (grafos, pacotes e eventos).

4. Análise de Robustez

Todos os códigos que envolvem alocação de memória dinâmica foram reforçados com estruturas ***try-catch*** genéricas para lançar mensagens de erro durante a execução, mas não

interrompê-la, porque no contexto deste código, a estabilidade é preferível, já que em um caso real, mesmo que o sistema fique lento no primeiro momento, o pacote deve chegar ao destino, então precisamos manter a execução mesmo que signifique vazar memória. O sistema implementado dos ***try-catch*** não irá tratar a exceção, apenas sinalizar onde ocorreu o erro e qual foi ele, por meio do ***fprintf(stderr, ...)***.

Os blocos ***try-catch*** também foram inseridos em métodos para sinalizar erros distintos de *leaks*, como algum acesso a uma posição inexistente, por exemplo. Quanto à robustez do **vector**, na alocação de memória sua memória, no caso de não possuir size inicial, é iniciado com **_capacity** igual a um, o mínimo para ter memória alocada, para evitar *leaks*. Como pode ser visto na imagem, após rodar o teste do *Valgrind* para achar *leaks*, usando o teste 3 como exemplo por ser mais pesado, nenhum foi achado:

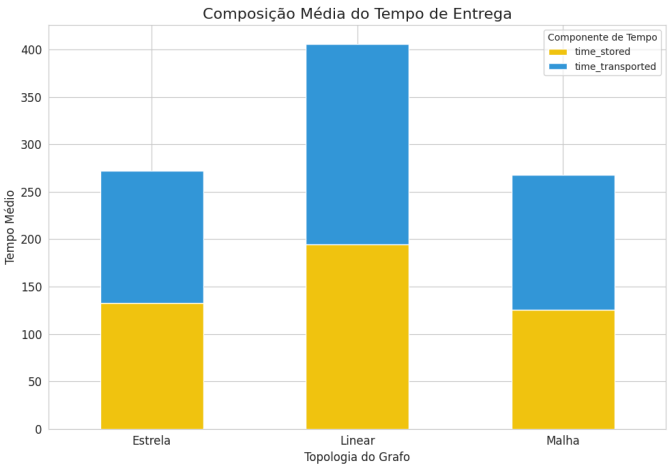
```
==913==
==913== HEAP SUMMARY:
==913==     in use at exit: 0 bytes in 0 blocks
==913==   total heap usage: 2,479 allocs, 2,479 frees, 160,668 bytes allocated
==913==
==913== All heap blocks were freed -- no leaks are possible
==913==
==913== For lists of detected and suppressed errors, rerun with: -s
==913== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

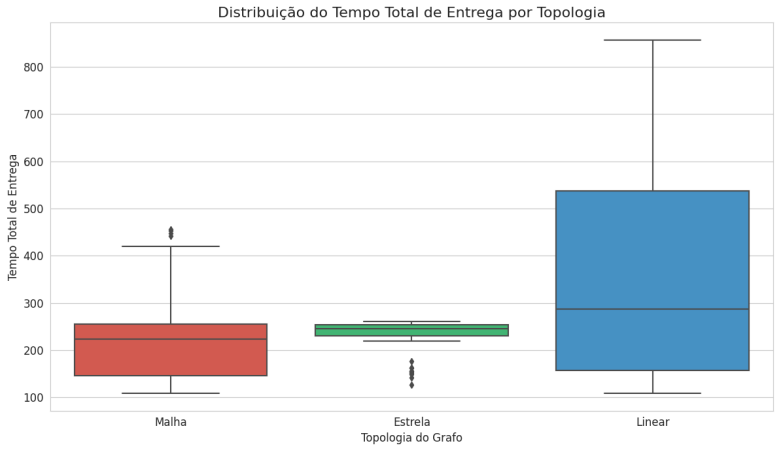
5. Análise Experimental

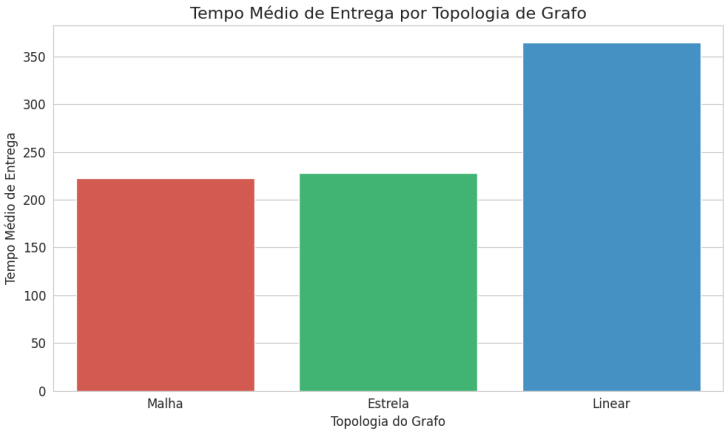
5.1 Elaboração

A análise experimental será feita testando diferentes configurações de topologias de mapas de armazéns, mais especificamente malha, estrela e linear, e utilizando, inicialmente, 100 pacotes que serão distribuídos em 10 armazéns, a fim de analisar como os tempos de armazenamento e transporte totais se comportam nessas condições.

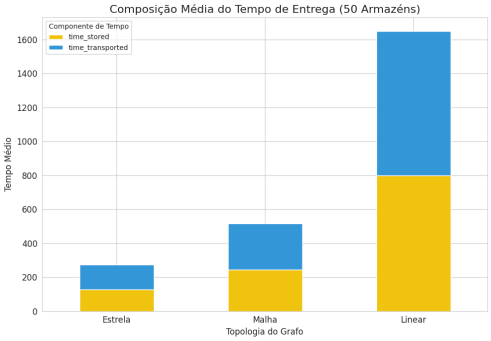
5.2 Resultados e Análise

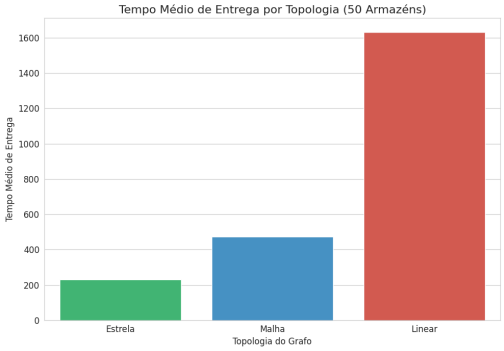






Estes resultados demonstram que a composição do esquema de armazéns é de suma importância para a eficiência do sistema, evidenciando a clara vantagem que configurações de malha e estrela possuem sobre um esquema linear, que foi usado como teste discrepante. A ideia principal é evidenciar que uma rede logística com alta conectividade otimiza muito os tempos de entrega. Testando agora com 50 armazéns e 250 pacotes, temos:





Estes novos resultados evidenciam a escalabilidade de um modelo altamente conexo como o de estrela, e a não escalabilidade do modelo linear, enquanto o modelo de malha continua sendo equilibrado. Portanto, concluímos que a disposição dos armazéns é um fator **crucial** para a velocidade de entrega dos pacotes e quanto mais conexões, mais rápido.

6. Conclusão

O problema tratado neste trabalho, foi o de implementar a **automatização do sistema de entregas** da empresa “Armazéns de Hanoi”, de mesma origem dos criadores da “Torre de Hanoi”. As diversas estruturas de dados utilizadas para resolver o problema se mostraram muito úteis e eficientes, principalmente o **heap** e o **vector**, já que ao utilizá-las conseguimos eficientemente ordenar os eventos e armazenar os dados de forma segura e expansível, além das **pilhas** para simular o armazenamento dos **pacotes.**

Em geral, o trabalho proveu uma boa noção de como a implementação do sistema em cenários variados pode ter uma grande influência na execução do código, expondo seus pontos fortes e possíveis melhorias, principalmente na disposição dos armazéns e na escolha da rota.

7. Bibliografia

Lacerda, A. and Meira JR, W.(2024). Slides da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Documentação Extra

8.1 Algorithms

Foram incluídas estruturas que permitem a ponderação de arestas com o objetivo de utilizar o algoritmo de **Dijkstra**, a fim de simular um cenário mais real e complexo, em que nem sempre a melhor rota será a mais curta, como é o caso do trânsito na cidade, caminhos mais longos com custos mais baixos podem ser preferíveis a caminhos curtos curtos com custos mais altos. Pensando nisto, foram implementadas duas estruturas genéricas e flexíveis:

- **edge**: armazena um vizinho e o peso, então representa uma aresta ponderada.
- **nodeDistance**: armazena um **node** e a distância deste até a origem, além de ter o operador sobrecarregado para que possa ser utilizado corretamente no ***min heap***.

Com estas estruturas, podemos implementar o algoritmo:

- **stack<T> dijkstra(vector<vector<edge<T, W>>>& graphic, T originNode, T destinationNode):** o algoritmo irá achar o menor caminho entre **originNode** e **destinationNode** a partir da propriedade do ***min heap*** para fazer escolhas ótimas quanto à distância total percorrida desde a origem. Enquanto o ***min heap*** não estiver vazio, ele irá remover o **node** com a menor distância e checar se o **node** atual é o **destinationNode**, se for, o caminho foi encontrado, se não, para cada **node** vizinho do atual, o algoritmo calcula o custo de chegar ao vizinho a partir do atual até achar o de menor custo, aí a distância é atualizada e o vizinho é inserido novamente no ***min heap*** com a nova prioridade, e o loop continua até chegar no **destinationNode**. No fim, uma **stack** é retornada com a rota ideal.

9. Complexidades Extras

As estruturas **edge** e **nodeDistance** não implementam métodos, são apenas containers de dados, então sua complexidade é sempre O(1). O algoritmo de Dijkstra, da forma que foi implementado, tem no seu pior caso a passagem por todos os vértices e arestas. O loop do **heap** irá passar no máximo uma vez por cada vértice V, então, pelo custo do **heap**, já temos O(V * log(V)), para a inserção e atualização do **heap**, teremos que fazer isto para todas as arestas do grafo, o que custaria O(E * log(V)), então teríamos, temporalmente, a inicialização de todos os vértices, O(V) + O(V * log(V)) + O (E * log(V)) = O(E * V * $\log^2$(V)). Espacialmente, teremos que alocar, no pior caso, O(V) para o **vector** de pais, O(V) para o de distâncias e O(V) para o ***min-heap***, além dos O(V^2 + E) do próprio grafo (leitura da matriz de adjacência no arquivo), então, O(V^2 + E).

10. Considerações Finais

O algoritmo foi implementado apenas no arquivo **algorithms**, porque o sistema de ponderação de arestas do grafo não foi implementado, no entanto, ele seria de grande ajuda para casos em que há trânsito de pacotes em uma mesma rota, para conseguir distribuí-los mais eficientemente, embora seu custo seja superior ao da BFS. Perderíamos em complexidade mas ganharíamos tempo com o transporte dos pacotes.