Trabalho Prático 2

Sistema de Despacho de Transporte por Aplicativo

Samuel Higino Rocha de Almeida - 2024066083

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG - Brasil

samuelhralmeida@gmail.com

1. Introdução

O problema proposto consiste em simular um sistema de corridas compartilhadas por aplicativo. O programa deve recebe demandas de corrida ordenadas por uma marcação de tempo, agrupa demandas que são elegíveis a uma corrida compartilhada e simula o andamento das corridas, gerando estatísticas sobre a simulação ao final da execução. Para solucionar isso, foi implementado um conjunto de classes que opera em prol de formar grupos com demandas que são elegíveis para uma corrida compartilhada e programar uma corrida para cada grupo.

A seção **2** trata da **implementação** do programa, descreve como o código está organizado, quais estruturas de dados foram utilizadas e quais tipos abstratos de dados foram criados para o programa, bem como apresenta especificações sobre o ambiente onde o código foi desenvolvido e testado. Caso necessário, também serão descritos os funcionamentos de métodos de estruturas de dados ou TADs.

A seção **3** foca em **analisar a complexidade** das funções implementadas nas classes do programa, utilizando notação assintótica.

A seção **4** descreve as **estratégias utilizadas para robustez** de código e técnicas de programação defensiva utilizadas para manter o código estável.

A seção **5** trata-se da **análise experimental**, apresenta como os testes foram feitos e quais foram os resultados.

A seção **6** compõe uma **conclusão** para o projeto, incluindo um resumo do que foi apresentado ao longo da documentação, observações adicionais, problemas enfrentados durante o desenvolvimento e os ganhos de conhecimento ao final do trabalho.

Finalmente, a seção **7** apresenta as **referências bibliográficas** utilizadas para o desenvolvimento do projeto.

2. Implementação

2.1. Organização

A pasta principal do projeto (TP) está organizada da seguinte maneira:

— **TP**
|— **bin**: diretório para onde vai o arquivo output do programa, tp2.out;
|— **include**: arquivos de cabeçalho (.hpp) dos TADs implementados;
|— **obj**: diretório para onde vão os arquivos de objeto (.o) da compilação;
|— **src**: arquivos de código .cpp, incluindo implementação das classes e o main;
Makefile

2.2. Funcionamento

O programa começa solicitando os parâmetros de simulação, que são:

- Eta (η): capacidade máxima dos veículos;
- Gama (γ): velocidade dos veículos;
- Delta (δ): intervalo máximo de tempo permitido entre demandas para uma corrida compartilhada;
- Alpha (α): distância máxima entre origens de demandas permitida para uma corrida compartilhada;
- Beta (β): distância máxima entre destinos de demandas permitida para uma corrida compartilhada;
- Lambda (λ): eficiência mínima de uma corrida compartilhada, dada pela soma dos deslocamentos de cada demanda sobre a distância total da corrida compartilhada.

Logo depois, o programa inicializa um Manager com os parâmetros de simulação, solicita a quantidade de demandas que serão oferecidas pela simulação e as informações das demandas uma a uma. Cada demanda é criada usando o método CreateDemand do Manager, e, depois que todas as demandas são coletadas, o programa inicia a simulação e imprime as estatísticas das corridas.

2.3. Estrutura de Dados e TADs

2.3.1. Tipos Abstratos de Dados

1. **Point2D**

Esta classe foi implementada para representar um ponto 2D no espaço, com coordenadas x e y. Foram implementados somente métodos para obter cada coordenada e calcular a distância até outro ponto. Também é possível copiar um ponto para outro.

2. **Demand**

Esta classe foi implementada para representar as demandas por corrida no aplicativo. Cada demanda possui um ponto de origem, um ponto de destino, um ID e o tempo quando foi solicitada. Os métodos incluem acesso aos atributos do objeto, calcular a distância entre origens e destinos com outra demanda e calcular o deslocamento da demanda.

3. **Stop**

Esta classe foi implementada para representar uma parada de uma corrida. Ela possui um ponto, um ID – o mesmo da demanda associada – e um tipo, que pode ser de embarque ou desembarque. Os métodos permitem ter acesso aos atributos e calcular a distância até outra parada.

4. **Segment**

Esta classe foi implementada para representar um segmento ou trecho de uma corrida, entre duas paradas. A classe possui um ponteiro para duas paradas, além de um comprimento/distância, um tipo (coleta, deslocamento ou entrega) e um marcador se o trecho foi completo. Seus métodos permitem ter acesso aos atributos e marcar o segmento como completo.

5. **Ride**

Esta classe foi implementada para representar a corrida em si. Ela armazena um conjunto de paradas e um de segmentos, contadores para monitorar a quantidade de paradas e segmentos, marcações se a corrida está em execução e se foi concluída, além de sua distância total, eficiência, tempo de começo e de fim e duração total.

As corridas são construídas com base em um grupo de demandas (próximo item) e, com base nas demandas, calcula a ordem das paradas, a distância e a eficiência da corrida. No construtor também é passado a eficiência mínima e, caso o critério não seja obedecido, uma exceção *low_efficiency* é lançada e a construção da corrida é interrompida.

Seus métodos incluem acesso a atributos, assinalar início e conclusão da corrida, calcular sua duração e imprimir as coordenadas das paradas. A impressão é feita na ordem em que as paradas são visitadas.

6. **DemandGroup**

Esta classe foi implementada para representar um grupo de demandas elegíveis para uma corrida compartilhada juntas. O grupo funciona como uma pilha – mais detalhes em 2.3.2 – e possui um tamanho máximo passado como parâmetro no construtor.

7. **Event e EventScaler**

Estas classes funcionam em conjunto para formar o escalonador. O evento é composto apenas de uma marca de tempo, um identificador e um tipo (início de corrida ou fim de corrida). Já o escalonador em si funciona como um min-heap e armazena os eventos organizando-os por tempo – mais detalhes em 2.3.2.

8. **Manager**

Esta é a classe principal que reúne quase todos os processos de funcionamento do programa. Ela possui como atributos os parâmetros de simulação, vários grupos de demanda e corridas, além de um escalonador e uma marca de tempo de simulação. Esta classe tem duas funções principais: a de criar uma demanda (CreateDemand) e de iniciar a simulação (StartSimulation).

A função CreateDemand inicializa uma nova demanda com os parâmetros fornecidos e tenta inseri-la no grupo mais recentemente criado, verificando se ela é compatível por todos os critérios com as outras demandas do grupo. Se não for, a função cria um novo grupo de demandas e insere a nova demanda lá. Caso isso ocorra, também é criada uma corrida para o grupo anterior. Durante a criação de uma corrida, todos os eventos associados a ela já são agendados no escalonador.

Já a função StartSimulation inicia a recuperação dos eventos agendados em ordem cronológica, até que não hajam mais eventos restantes. Se o evento for de uma conclusão de corrida, são impressas as estatísticas da corrida associada em uma linha, no formato especificado pelo enunciado.

2.3.2. Estruturas de Dados

Neste trabalho foram usadas duas principais estruturas de dados: uma pilha e um min-heap. A classe DemandGroup possui o funcionamento de uma pilha e EventScaler de um min-heap.

Começando pela pilha, o funcionamento é mais simples. O grupo de demandas é inicializado como um vetor de demandas e mantém um contador para monitorar quantas demandas tem o grupo. O método para inserir insere uma demanda na primeira posição disponível do vetor e o método de retirar uma demanda sempre retira a mais recente inserida, ou seja, a de maior posição no vetor. Esta classe permite também o acesso a qualquer elemento da pilha, apesar de isso não ser unanimidade da estrutura de dados, pois o acesso é importante para o projeto.

Já o min-heap, o escalonador, simula uma árvore binária em um vetor de eventos. A característica de min-heap faz com que os sucessores de um nó da árvore seja sempre maior ou igual ao próprio nó e, neste caso, o critério para saber se um evento é maior é pelo tempo. A classe conta com dois métodos fundamentais que restauram a propriedade de min-heap em qualquer ponto da árvore, um que restaura a partir de um nó para cima (HeapifyUp) e outro para baixo (HeapifyDown). Na inserção de um novo evento, ele é inicialmente colocado na posição de maior valor do vetor (em uma folha da árvore) e é chamada a função HeapifyUp para restaurar a propriedade de min-heap a partir do novo evento. Já a recuperação do evento (ou remoção) ocorre sempre na raíz da árvore – o evento de menor tempo. Para a remoção deste evento, o evento na posição de maior valor do vetor é movida para a raíz no lugar dele e é chamada a função HeapifyDown para reorganizar o min-heap.

2.3. Ambiente de programação e testes

O código foi desenvolvido no Visual Studio Code, usando um subsistema Linux para Windows 11 com WSL Ubuntu. Além disso, o código foi desenvolvido na linguagem C++11 e compilado pelo compilador G++ da GNU Compiler Collection integrado ao Visual Studio Code.

Dados de processador e memória da máquina usada para desenvolvimento e testes:

- Processador Intel Core i5-12500H (2.50GHz) de 12ª geração
- RAM 8 GB DDR5 4800 MHz

3. Análise de complexidade

Nesta seção, chamaremos de n o número de demandas solicitadas em uma execução do programa.

3.1. Tempo

Para a análise assintótica de complexidade de tempo, vamos olhar para as funções MakeDemand e StartSimulation do Manager, uma vez que essas são as duas principais funções chamadas pelo main.cpp. Serão mencionadas apenas as complexidades de funções que sejam relevantes para a análise – funções com complexidade $\Theta(1)$, por exemplo, não afetam a análise de complexidade e serão omitidas.

A primeira dessas, MakeDemand, possui um melhor caso de complexidade $\Omega(1)$, que é o caso trivial onde a primeira demanda é inserida no primeiro grupo e não precisa haver nenhuma checagem de compatibilidade. Já no pior caso, caso todas as primeiras $(n - 1)$ demandas sejam elegíveis para uma corrida compartilhada e o parâmetro η é igual a n, a n-ésima demanda poderá ser checada nos critérios de distância (α e β) com todas as outras demandas, que resultará em uma complexidade $O(n)$. Adicionalmente, caso esta demanda seja compatível com todas pelos critérios de distância, a checagem de eficiência (λ) também será $O(n)$, uma vez que chama o construtor da classe Ride que possui complexidade $O(n)$ no pior caso – já que esta tem que criar todas as paradas e segmentos da corrida, que têm número proporcional a n. A criação de um novo grupo e inserção de uma demanda em um grupo não afetam a complexidade de tempo (por serem $\Theta(1)$). No fim, a complexidade total de uma chamada da função, no pior caso, é:

$$O(n) + O(n) = O(n)$$

A segunda função, StartSimulation, possui um loop while que roda de acordo com a quantidade de eventos agendados, que por sua vez é proporcional à quantidade de corridas, que é limitada superiormente por n – no pior caso, todas as corridas são individuais. A recuperação de um evento no escalonador custa $O(log\, n)$, em virtude do número de eventos ser, por consequência, limitado por uma função $O(n)$. A cada vez que um evento recuperado representa a conclusão de uma corrida, é realizada a impressão de suas estatísticas, que incluem as coordenadas dos pontos de parada. Os número de pontos de parada é sempre

proporcional a n, uma vez que, por cada demanda ter origem e destino, a quantidade total de paradas, independente do número de corridas, será $2n$. Portanto, pelo fato de que o crescimento da quantidade de eventos e de paradas não ser independente, a complexidade da função será:

$$O(n) \bullet O(log\ n) = O(n\ log\ n)$$

Por fim, vamos analisar a complexidade total de tempo do programa. No main.cpp, a função MakeDemand roda n vezes, uma para cada demanda. Sendo assim, a complexidade desta parte do programa é $n \bullet O(n) = O(n^2)$, e a complexidade total do programa é a soma desta complexidade com a da função StartSimulation, que é chamada 1 vez:

$$O(n^2) + O(n\ log\ n) = O(n^2)$$

Portanto, a complexidade total de tempo do programa é $O(n^2)$.

3.2. Espaço

Já para a análise assintótica de complexidade de espaço, vale ressaltar que as classes Point2D, Demand, Event, Stop, Segment e Scaler ocupam $\Theta(1)$ de espaço. Dessas, apenas Scaler ocupa espaço extra temporário em algumas funções, para restaurar a propriedade de heap, equivalente a $O(log\ n)$. Cada objeto de Ride ocupa $O(n)$ de espaço, proporcional à quantidade de demandas contidas no DemandGroup passado em seu construtor. Por sua vez, DemandGroup ocupa $O(\eta)$ de de espaço, proporcional à capacidade máxima dos veículos de uma simulação.

Sabendo disso, para a análise completa, temos que analisar a complexidade de espaço de Manager, que é a única classe instanciada no main.cpp. No pior caso, a classe terá n grupos de demandas e n corridas armazenadas, além de outros atributos que somam $\Theta(1)$. Isso resulta em uma complexidade de:

$$O(n) + O(\eta n) = O(n\eta)$$

Nenhum método de Manager usa ou chama outra função que usa espaço extra com complexidade maior que ou semelhante a $O(\eta n)$, o máximo que teremos é $O(log\ n)$ vinda de Scaler. Portanto, a complexidade total de espaço do programa no pior caso é $O(\eta n)$.

4. Estratégias de Robustez

O principal meio de tornar o código robusto foi por meio do tratamento de exceções da biblioteca stdexcept. Em várias partes do código, principalmente em métodos de classe, são lançadas exceções como *runtime_error*, *out_of_range* e *logic_error* em erros comuns, como tentar acessar posições inacessíveis de vetores, recuperar evento quando o escalonador está vazio, entre outros.

Foi criada uma exceção exclusiva para este trabalho, chamada *low_efficiency*, que herda de *runtime_error*. Esta exceção é lançada quando, na criação de uma corrida, a

eficiência mínima não é atingida. A classe **Manager** é a responsável pela maioria das detecções e tratamento de exceções.

5. Análise experimental

5.1. Ferramentas para testes

Para fazer os testes, foram gerados 3 arquivos diferentes com demandas aleatórias. As demandas tinham IDs sequenciais (0, 1, …) e as coordenadas tanto de suas origens quanto de seus destinos foram completamente aleatorizadas para serem pontos em $\{(x,\ y) \mid x \in [0, 100], y \in [0, 100]\}$. Dessa forma, não haveria nenhum viés para avaliação.

Para a medição de tempo, foi utilizada a biblioteca chrono do C++, e a medição de memória foi implementada diretamente no código. Todas as classes mantém conta de quanto espaço na memória ela ocupa e o Manager também mantém a contagem de quanto de memória extra foi utilizada e qual foi o pico global de uso. Algumas variáveis, como iteradores, foram ignoradas para a avaliação da memória por serem desprezíveis.

5.2. Objetivos

O objetivo dos testes foi observar o comportamento do sistema à medida que um dos parâmetros de simulação variava. Os resultados foram escolhidos com base no que parecia ser mais relevante ou interessante. Especificamente, para cada teste, foram coletados os seguintes dados:

- Maior uso global de memória
- Distribuição de corridas por número de passageiros
- Eficiência média das corridas

5.3. Resultados

Na análise do tempo de execução, foi observado um crescimento coerente com a análise assintótica, o tempo cresce em proporção maior do que linear em relação ao número de demandas. Nos 3 arquivos de entrada, a quantidade de demandas era de 50, 100 e 200. Foram feitos 10 experimentos com cada arquivo, variando o parâmetro λ de 0.1 até 0.82 e depois foi feita a média do tempo de execução dos experimentos. Os resultados foram os seguintes:

- 50 demandas: 252.5 ms em média
- 100 demandas: 593.2 ms em média
- 200 demandas: 1559.8 ms em média

Vale lembrar que esses valores na prática não servem bem para uma análise quantitativa, mas para uma análise qualitativa. O tempo mostrado pelos testes na verdade é muito maior do que na prática pois o código dos testes teve de monitorar várias nuances a mais do que o programa padrão.

O primeiro resultado interessante de se observar é quando comparamos o uso de memória do programa à medida em que o parâmetro lambda cresce. No gráfico 5.1, podemos observar o crescimento do máximo global de memória usada pelo programa à medida em que o parâmetro lambda (λ) cresce em duas simulações diferentes, uma com 200 demandas (barras em azul) e outra com 100 demandas (barras em vermelho).

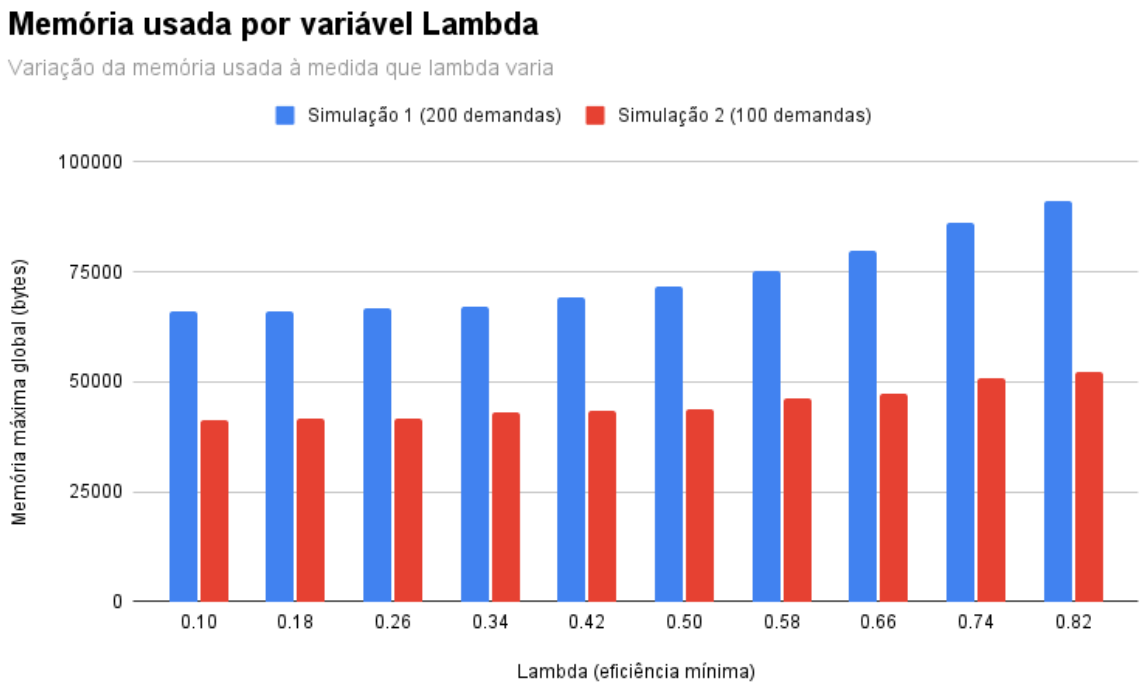

Gráfico 5.1

Em ambas as simulações os outros parâmetros permaneceram constantes e apenas λ varia. A memória cresce junto com λ, pois quanto maior seja λ, mais corridas individuais são geradas ao invés de corridas compartilhadas. Isso pode ser observado no gráfico 5.2, que mostra a variação da quantidade de cada tipo de corrida (de 1 a 4 passageiros atendidos) de acordo com o parâmetro λ na simulação de 200 demandas.

Observe como a quantidade de corridas individuais cresce rapidamente e os outros tipos de corrida tendem a decrescer. Logicamente, quanto maior o λ, menor a taxa de compatibilidade entre demandas e maior será o número de corridas individuais. É interessante reparar também que o número de corridas duplas não varia tão bruscamente quanto as outras, isso porque muitas corridas que eram triplas ou quádruplas em simulações com λ baixo têm mais chance de se tornar corridas duplas quando λ cresce.

Outro resultado interessante observado é a correlação entre a eficiência média das corridas e o máximo de memória global usada. É possível ver pelo gráfico 5.3 que essa correlação é direta. Isso ocorre pois as corridas compartilhadas tendem a ter uma eficiência menor que 1, já que, como as demandas são aleatórias, a chance de duas demandas serem próximas o suficiente em origens e destinos e ao mesmo tempo terem deslocamento grande é

muito baixa. Por isso, quanto mais próximo de 1 for a eficiência média, mais corridas individuais provavelmente são geradas, e mais memória ocupada.

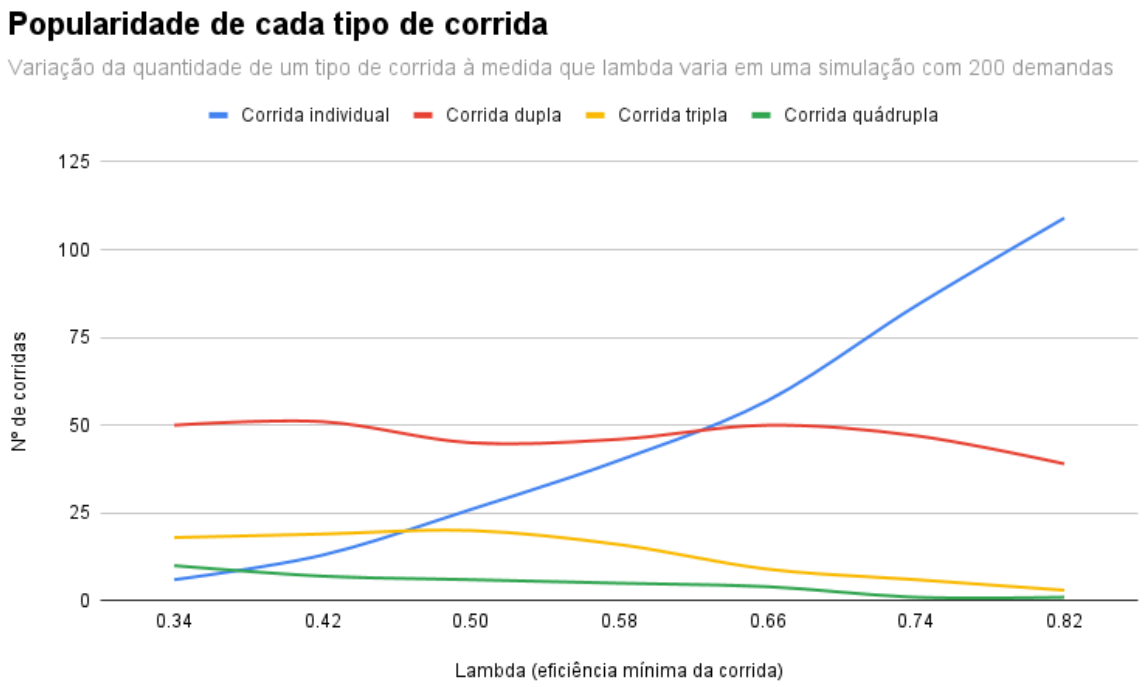

Gráfico 5.2

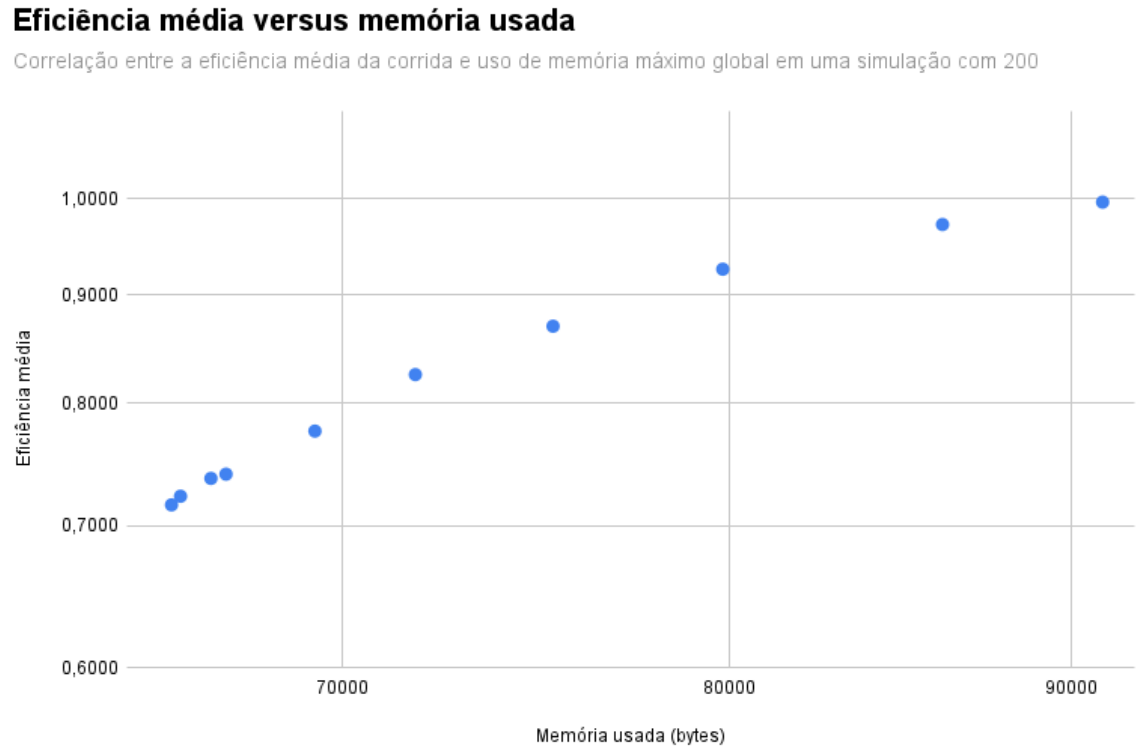

Gráfico 5.3

6. Conclusão

Durante a produção deste trabalho, ficaram evidentes dois objetivos principais: praticar a implementação de diferentes estruturas de dados e produzir uma análise experimental que relacionasse os parâmetros de simulação com outras variáveis. Os principais problemas enfrentados foram justamente a decisão por quais estruturas de dados utilizar e como produzir uma análise experimental relevante.

Com relação ao primeiro desses problemas, a decisão pelo min-heap foi tanto pela recomendação do enunciado tanto pelo desejo de praticar uma estrutura que é raramente usada em um contexto casual. No geral, a implementação, apesar de ter tido seus problemas, foi um bom aprendizado para entender até mesmo as árvores binárias no geral. Já a implementação da pilha surgiu naturalmente com o problema proposto pelo enunciado, já que se encaixava perfeitamente às necessidades do código.

Já ao segundo problema, foi a parte mais difícil de todo o projeto. Como são muitos parâmetros possíveis de avaliar e muitas correlações possíveis de se observar, a decisão de escolher apenas algumas foi difícil de avaliar. A opção final acabou sendo pelos experimentos em que o parâmetro lambda (λ) variava, já que produzia os resultados mais interessantes. Na variação dos outros parâmetros, o único aspecto interessante de se observar era a variação da própria eficiência das corridas.

Ao final do projeto, o uso das diferentes estruturas de dados se mostrou finalmente ser útil em contextos reais, algo que é difícil apresentar durante as aulas. Também ficou claro que uma análise experimental boa pode revelar nuances ocultas do seu projeto. No geral, foi um projeto enriquecedor que exigiu colocar todos os conhecimentos das aulas em prática.

7. Referências

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - *Introduction to Algorithms (4th Edition)*

2. Geeks for Geeks - *Min-Heap in C++*; disponível em: https://www.geeksforgeeks.org/cpp/min-heap-in-cpp/