



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Otimização de algoritmos e escrita de programas eficientes

Bacharel Ciências da Computação

Bruno Rodrigues Faria

Resumo

Este trabalho tem como objetivo aplicar técnicas para escrever programas eficientes em C/C++. Para isso, é proposto um programa, que lê uma matriz de bytes (sem sinal) de um arquivo e a processa, imprimindo o resultado na tela. O programa é executado para matrizes de diferentes tamanhos e tempos de execução são medidos, a fim de determinar a função de custo que melhor se aproxima dos resultados. Em seguida, são propostas cinco alterações sequenciais para reduzir o tempo de execução do programa, documentando as mudanças e o speed-up obtido. Após a última alteração, um novo gráfico de **tempo x tamanho do arquivo** é plotado para determinar se houve mudança na ordem de complexidade do programa. Por fim, são analisados os resultados obtidos. O objetivo final é encontrar uma solução de compromisso entre tempo e memória que ofereça o melhor custo-benefício.

Palavras-chave: Otimização. Análise de Algoritmos. Algoritmos. PAA.

Sumário

1	INTRODUÇÃO	3
2	DESENVOLVIMENTO	3
2.1	Criação de arquivo aleatório	3
2.2	Algoritmo inicial	3
2.2.1	Primeira execução do algoritmo	4
3	MÁQUINA	5
4	MUDANÇAS E RESULTADOS	5
4.1	Alteração 1	6
4.2	Alteração 2	7
4.3	Alteração 3	8
4.4	Alteração 4	9
4.5	Alteração 5	10
5	FUNÇÃO DE CUSTO E COMPLEXIDADE	11
6	COMPARAÇÃO - VERSÃO ORIGINAL E VERSÃO FINAL	11
7	CONCLUSÃO	12

1 INTRODUÇÃO

Escrever programas eficientes é essencial para garantir a boa performance de sistemas computacionais e atender às demandas de usuários cada vez mais exigentes. Além disso, a eficiência de um programa pode impactar diretamente na economia de recursos computacionais, como memória e processamento, que são finitos e têm um custo associado. Por outro lado, encontrar uma solução de compromisso entre tempo e memória pode ser necessário em diversas situações, quando a disponibilidade de recursos é limitada ou quando é preciso atender a requisitos conflitantes. Portanto, é importante desenvolver habilidades para escrever programas eficientes e encontrar soluções de compromisso que maximizem o custo-benefício em cada caso específico.

2 DESENVOLVIMENTO

2.1 Criação de arquivo aleatório

O código gera um arquivo contendo uma matriz de valores aleatórios com um tamanho especificado pelo usuário, utilizando o gerador de números aleatórios da biblioteca padrão do C escrevendo em um arquivo. Esse arquivo será utilizado para testar o desempenho de um programa que realiza operações em matrizes de bytes sem sinal.

2.2 Algoritmo inicial

O código inicial passado pelo professor é um programa C que calcula uma matriz 2D de valores float, dado um arquivo de dados binário, número de linhas e número de colunas como entradas. O código calcula a matriz executando várias operações matemáticas nos elementos do arquivo de dados de entrada.

2.2.1 Primeira execução do algoritmo

Durante a primeira execução do algoritmo, foi constatado que o tempo de execução foi bastante extenso, enquanto o consumo de memória permaneceu baixo como mostrado na **Figura 1**. Com isso, foi possível perceber a variação do tempo de execução de acordo com o tamanho do arquivo utilizado na **Figura 2**. Além disso, foi observado que o algoritmo segue uma complexidade linear, o que pode ser confirmado pela visualização dos gráficos que se aproximam de uma linha quando os valores são mais próximos um do outro.

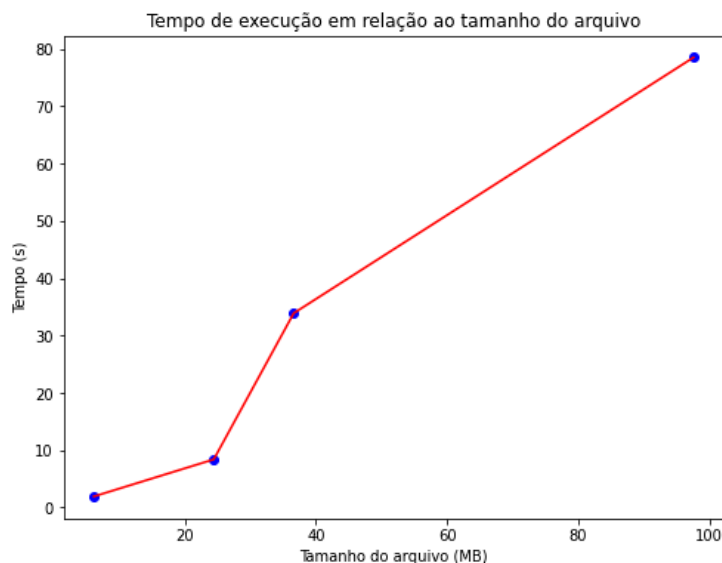


Figura 1 – Análise de tempo x tamanho do arquivo

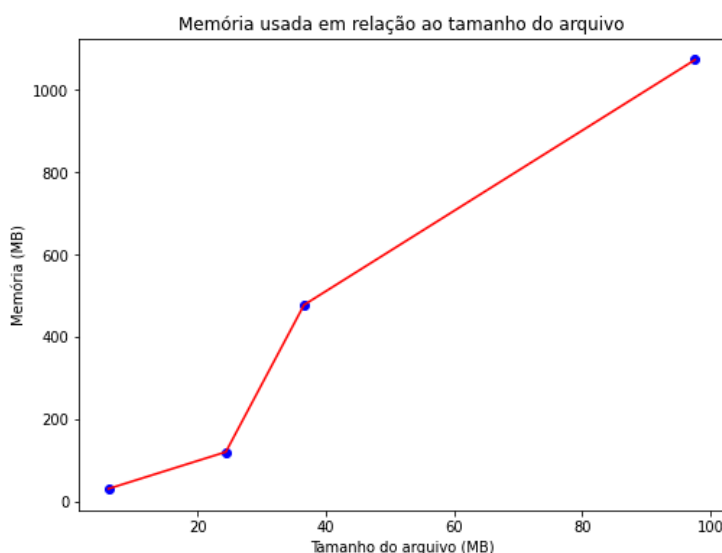


Figura 2 – Análise de memória x tamanho do arquivo

Além disso, pode-se calcular a função de custo para auxiliar na aproximação dos resultados, como mostrado na **Função 1**, juntamente com a complexidade do algoritmo onde m é a quantidade de linhas e n é a quantidade de colunas.

$$f(n) = 512 + 3(m * n), O(m * n) \quad (1)$$

3 MÁQUINA

A fim de fornecer informações mais detalhadas sobre o ambiente experimental utilizado, apresentamos na **Tabela 1** as especificações do computador utilizado nos testes realizados neste estudo. Os dados contidos nesta tabela incluem informações sobre o processador, quantidade de memória RAM, sistema operacional, entre outros.

Especificações	
PROCESSADOR	i5-10400F 2.90GHz
MEMÓRIA RAM	16 GB - 2666mhz - DDR4
SIS. OPERACIONAL	Windows 10
PLACA DE VÍDEO	RX 5500 XT - 8GB
COMPILADOR	MinGW

Tabela 1 – Especificação da máquina usada para testes

4 MUDANÇAS E RESULTADOS

Com isso, foi proposta 5 alterações para o programa que visam reduzir o tempo de execução. Foram documentadas de forma sequencial e foi apresentado o speed-up de ganho feito pela **Função 2** obtido para a versão anterior para a versão mais recente (com a mudança aplicada).

$$Speedup = \frac{tempo_i}{tempo_{i+1}} \quad (2)$$

4.1 Alteração 1

Para a alteração 1, foi feita uma mudança na função DetSinCos, a qual era chamada a quantidade de **colunas * linhas** realizando um calculo repetitivo de seno e cosseno. Para contornar isso, cria-se dois arrays de 256 posições cada para armazenar o cálculo de seno e cosseno para todas as possibilidades de valores especificados no trabalho **0 até 255**. Com isso, quando o programa precisar da valor do seno ou cosseno de um determinado número basta buscar o valor do cálculo na posição do vetor: **seno[numero_procurado]**. Vale a pena ressaltar que como foi utilizado um vetor a utilização da memória foi aumentada, já que criou-se um novo vetor para armazenamento. O speedup obtido para cada um dos arquivos será apresentado na **Tabela 2**.

Speedup	
<i>Tamanho do arquivo</i>	Speedup obtido
6.1 MB	1.586
24.4 MB	1.5
36.6 MB	1.507
97.6 MB	1.506

Tabela 2 – Speedup obtido na alteração 1

Além da tabela de Speedup, pode-se observar a variação do tempo conforme o tamanho do arquivo comparado a versão original do programa, onde o tempo de execução tinha um valor muito mais elevado.

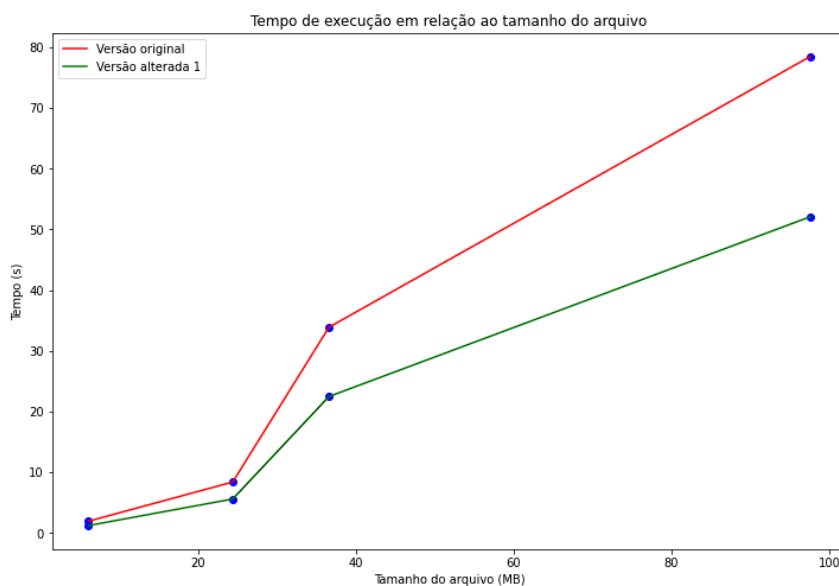


Figura 3 – Comparação dos tempos x tamanho do arquivo

4.2 Alteração 2

A mudança feita no código foi na forma como as frequências dos elementos da matriz são contabilizadas e como os elementos são acessados durante o processamento. No código antigo, utilizava-se dois loopings para acessar as posições da matriz executando a cada posição um cálculo custoso para achar a posição correta na memória ($*(M + j * cols + i)$).

No código novo, o looping foi mudado para apenas um, pois a forma de guardar a matriz é unidimensional (**percorre de 0 até rows*cols - 1**), não sendo mais necessário os cálculos de acesso a posição correta da memória. Então essa mudança tem o benefício de reduzir o número total de operações necessárias, fazendo com que a execução diminua o tempo para ser concluída. Visto as alterações, o Speedup obtido com a mudança pode ser obtido na Tabela 3.

Speedup	
<i>Tamanho do arquivo</i>	Speedup obtido
6.1 MB	1.10
24.4 MB	1.26
36.6 MB	1.27
97.6 MB	1.30

Tabela 3 – Speedup obtido na alteração 2

Além da tabela de Speedup, pode-se observar a variação do tempo conforme o tamanho do arquivo comparado a versão original do programa, onde o tempo de execução tinha um valor mais alto.

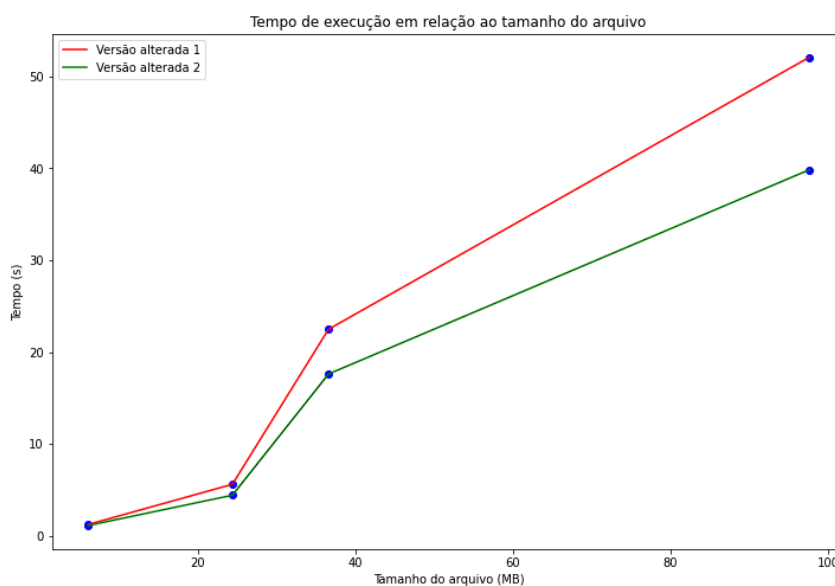


Figura 4 – Comparação dos tempos x tamanho do arquivo

4.3 Alteração 3

Para a alteração 3 foi modificada a forma de leitura do arquivo. No código antigo a leitura do artigo era feita de elemento em elemento, ou seja, tinha-se um looping que percorria de **0 até linhas * colunas - 1**, tendo a cada iteração uma busca no arquivo pela função de leitura **fread()**. Com isso, a alteração proposta foi de alterar a leitura para bloco, retirando o looping necessário para ler elemento por elemento, fazendo isso chama-se a função apenas uma vez passando como bloco a quantidade de **linhas * colunas**.

Speedup	
<i>Tamanho do arquivo</i>	Speedup obtido
6.1 MB	1.23
24.4 MB	1.27
36.6 MB	1.29
97.6 MB	1.28

Tabela 4 – Speedup obtido na alteração 3

Observa-se pela comparação do gráfico que ele é muito parecido com o gráfico da alteração 2. Isso se dá pelo motivo dos Speedups estarem muito próximos, fazendo com que o ganho seja o mesmo para ambas alterações.

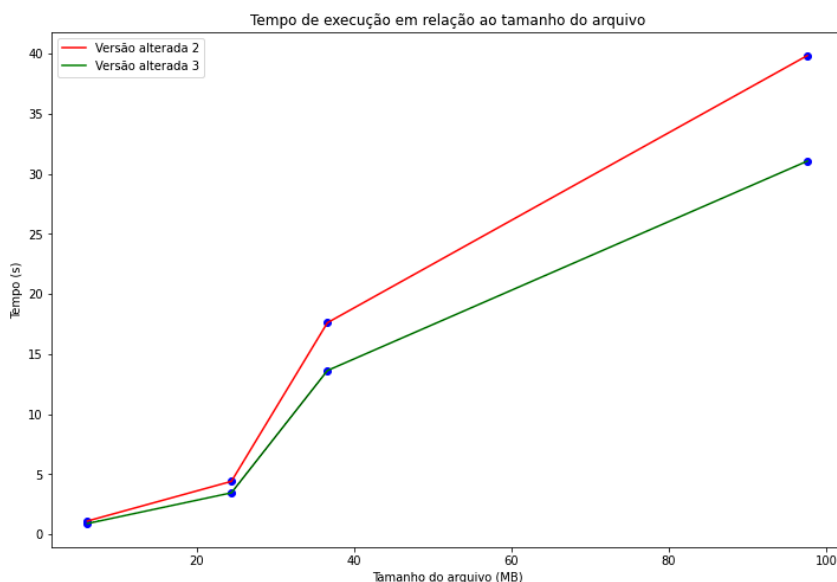


Figura 5 – Comparação dos tempos x tamanho do arquivo

4.4 Alteração 4

A alteração 4 trata totalmente da criação de dois novos arrays para armazenamento prévio de cálculos que estão sendo repetidos. Dentro da função **DetOutPut()** no código antigo faz-se a cada iteração do looping no qual a chamada está inserida um cálculo exponencial adicionando o valor em uma variável, isso é muito inviável, pois faz um cálculo repetido várias vezes, já que temos números repetidos quando as matrizes são grandes.

Para isso criamos dois novos arrays de 256 posições, para armazenar o pré-cálculo das exponenciais para todos elementos. Fazendo com que aumente um pouco o uso da memória mas caia assim a quantidade de tempo para execução, por que será desprezado vários cálculos que não precisavam se repetir.

Speedup	
<i>Tamanho do arquivo</i>	Speedup obtido
6.1 MB	3.75
24.4 MB	3.56
36.6 MB	3.42
97.6 MB	3.54

Tabela 5 – Speedup obtido na alteração 4

Pode-se observar que o Speedup foi muito grande, por se tratar de um calculo que demanda certo custo. Sendo assim, ao olhar o gráfico vê-se uma grande diferença nos tempos da versão antiga para a nova.

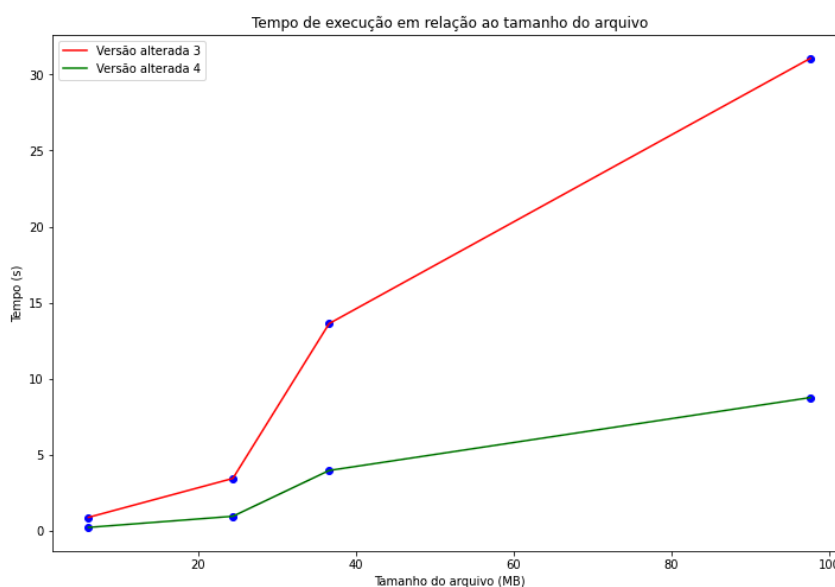


Figura 6 – Comparação dos tempos x tamanho do arquivo

4.5 Alteração 5

Outro cálculo muito custoso que estava acontecendo repetidamente ao caminhar pelo looping é o cálculo de **pow(x, 2)**, o qual faz o número x elevado a 2. Para isso, cria-se mais dois arrays, um para armazenar os valores dos números divisíveis por dois e outro para armazenar os valores dos números não divisíveis. Fazendo com que, faça os cálculos apenas 256 vezes já que é o intervalo de valores diferentes especificado, ganhando assim tempo em troca de memória. Após isso, quando precisa consultar o valor do cálculo apenas faz uma consulta no índice do array, algo bem menos custoso que o próprio cálculo.

Speedup	
<i>Tamanho do arquivo</i>	Speedup obtido
6.1 MB	4.13
24.4 MB	4.18
36.6 MB	4.38
97.6 MB	4.26

Tabela 6 – Speedup obtido na alteração 5

O Speedup dessa alteração foi muito alto, pois quando você abre mão de memória você tem um ganho muito alto em tempo. Essa foi a principal vantagem pela qual armazenar o valor dos cálculos foi uma boa ideia, ganhar tempo de execução para matrizes maiores, já que, no primeiro código a matriz com maior tamanho (97.6 MB) estava executando em 80 segundos e após as alterações o algoritmo passou a executar a mesma matriz no tempo de 2 segundos.

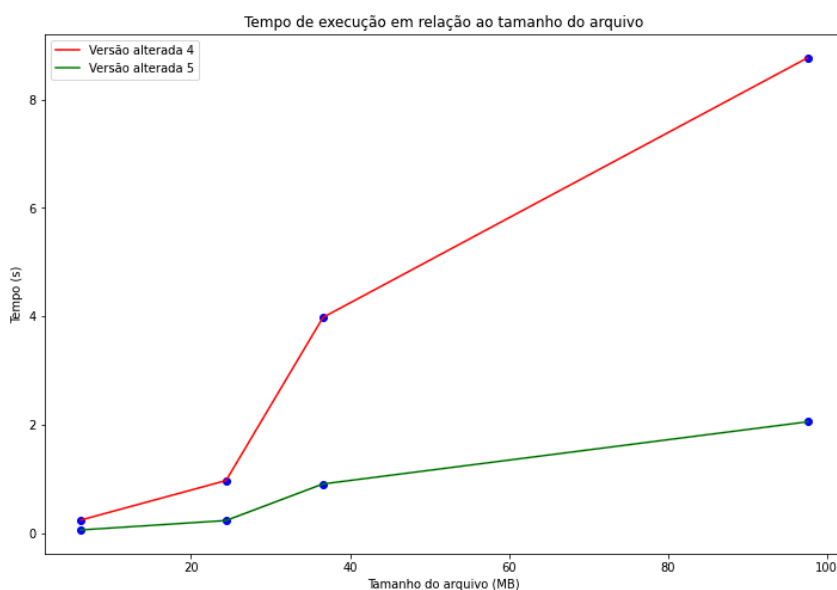


Figura 7 – Comparação dos tempos x tamanho do arquivo

5 FUNÇÃO DE CUSTO E COMPLEXIDADE

Após as alterações do código a função de custo e complexidade está sendo apresentada pela **Equação 3**, pode-se ver que a função de custo foi diferente da função de custo apresentada na **Seção da primeira execução do algoritmo**, mostrando que a quantidade de operações custosas foi muito menor, tendo como resultado o aumento da memória usada e a diminuição do tempo de execução. Porém a complexidade continuou a mesma, visto que o algoritmo precisa acessar todos os elementos presentes na memória, ou seja, a quantidade total de elementos se da pela quantidade de colunas multiplicado pela quantidade de linhas.

$$f(n) = 1024 + 2(m * n), O(m * n) \quad (3)$$

6 COMPARAÇÃO - VERSÃO ORIGINAL E VERSÃO FINAL

Um programa em C original é usado para processar uma matriz de dados lidos de um arquivo de bytes, utilizando funções matemáticas da biblioteca padrão, como sin, cos, exp e log. Ele contém várias funções que calculam e atribuem valores a variáveis. A versão modificada do código utiliza uma tabela pré-computada para calcular as funções trigonométricas sin e cos, bem como outra tabela para armazenar os valores de saída das funções DetOutput em vetores de valores ímpares e pares. Além disso, houve uma mudança na forma de armazenamento da matriz e nas operações de acesso à memória e leitura do arquivo para melhorar o tempo de execução. Com isso, o novo código sacrifica o uso de memória para obter um ganho de desempenho. A **Tabela 7** mostra o Speedup obtido com a versão modificada do programa em relação à original.

Speedup	
<i>Tamanho do arquivo</i>	Speedup obtido
6.1 MB	33.44
24.4 MB	36.20
36.6 MB	37.36
97.6 MB	38.21

Tabela 7 – Speedup total obtido

Por mais que a complexidade não tenha alterado, o número de operações críticas realizadas pelo algoritmo foi reduzida drasticamente, fazendo com que o algoritmo ficasse mais rápido por conta dessas operações ocuparem uma quantidade de $(m * n)$ várias vezes e no algoritmo modificado acontece na maioria 256 vezes, que é a quantidade de elementos possíveis como explicado anteriormente nas modificações. Pode-se observar o consumo de memória e o tempo de execução na **Figura 6**

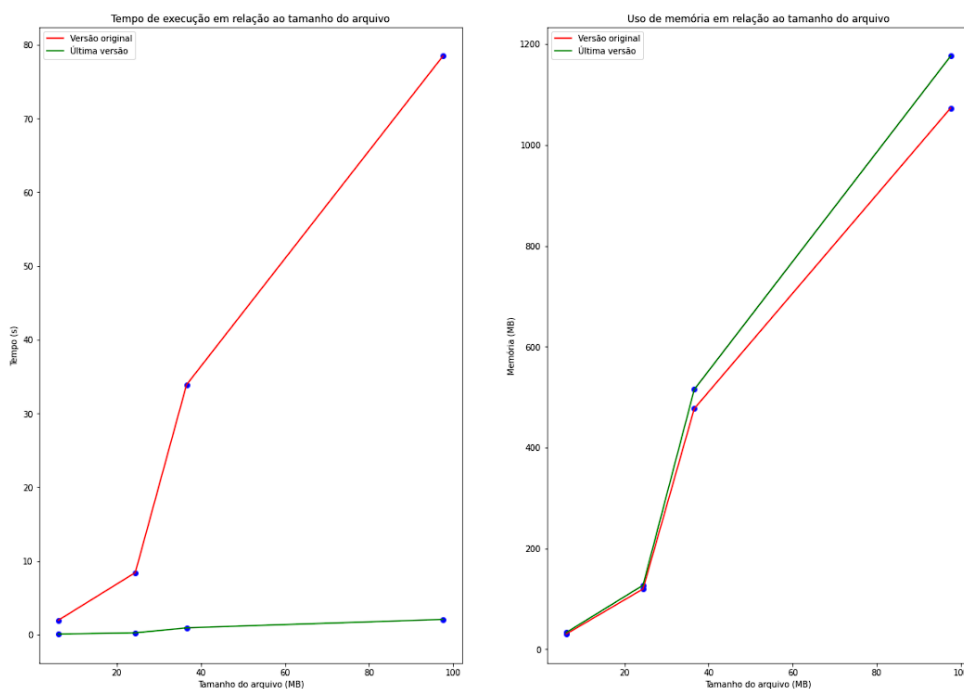


Figura 8 – Comparação - Original X Final

7 CONCLUSÃO

Após todas as alterações propostas, pode-se concluir que a melhor solução está presente na alteração **5**, onde o tempo ficou mais curto para a execução, após analisar o gráfico e o consumo de memória a alteração **5** é a solução com melhor custo/benefício, já que utiliza um pouco mais de memória mas não gasta o tempo que gastava antes para executar.

Porém, cabe analisar para saber se em algum dos casos tem uma solução de compromisso entre tempo e memória. A alteração **3** tem uma solução de compromisso, podemos confirmar isso pelo tempo de execução e memória já que fica o meio termo das duas (pior e melhor), não pesando para o lado da memória nem para o lado do tempo de execução, como mostra os gráficos da **Figura 9**.

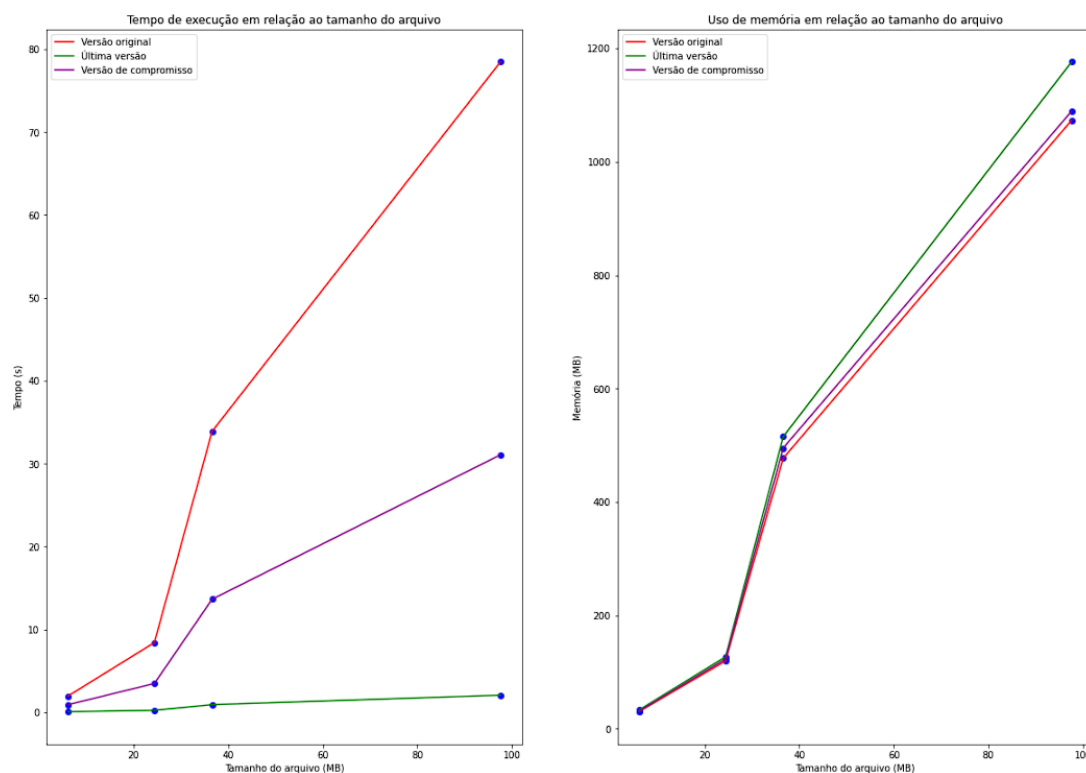


Figura 9 – Análise de solução de compromisso

No entanto, é importante notar que, em algumas situações, uma solução de compromisso pode não ser a melhor opção e pode levar a resultados insatisfatórios para todas as partes envolvidas. Nesses casos, outras abordagens, como a cooperação ou a busca por uma solução de ganha-ganha, podem ser mais eficazes, neste caso a solução que seria de maior interesse para o trabalho seria a que executa o programa no menor tempo de execução.