Prof. Wagner Meira Jr. DCC / ICEx /UFMG

DCC205 -Estrutura de Dados 2023.2

Trabalho Prático 2 – Verificar Coloração Gulosa em Grafo e Algoritmos de Ordenação

Lucca Alvarenga de Magalhães Pinto – lucca.alvarenga@dcc.ufmg.br

Matrícula: 2021036736

1 INTRODUÇÃO

O trabalho consiste na implementação de um programa para analisar a coloração de um grafo e verificar se é uma coloração válida. Se estiver de acordo, deve-se printar os vértices de forma crescente em relação as cores, caso contrário deve-se printar 0. Em caso que estiver certo, o grafo é ordenado pelo algoritmo de ordenação escolhido pela entrada padrão, podendo ser um dos 7 métodos: *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quicksort*, *Mergesorft*, *Heapsort*, *Método feito pelo aluno*. O objetivo é a analisar a performance dos algoritmos na ordenação dos vértices.

Assim, dado um grafo G com vértices numerados por inteiros positivos maiores que 0, a coloração será considerada correta quando:

- Um vértice não estiver conectado com outro de mesma cor.
- Um vértice de cor C estiver conectado com pelo menos um vértice com cada uma das cores menores que C. Por exemplo, um vértice de cor 3 deve estar conectado com outros de cores 1 e 2. No caso de um vértice ter cor 1, ele sempre estará certo.

A ordenação dos vértices considera a ordem das cores, mas no caso em as cores forem iguais deve-se ordenar de acordo com o rótulo dos vértices. Por exemplo, no caso do grafo descrito pela tabela abaixo:

Rótulo	0	1	2	3	4
Cor	1	3	2	1	2

O grafo ordenado ficaria da forma: 0 3 2 4 1. Logo, a estabilidade dos métodos de ordenação não tem impacto sobre o resultado, uma vez que vértices de cores iguais sempre vão possuir rótulos diferentes (rótulos são únicos).

2 MÉTODO

2.1 Configurações da máquina

O código foi executado nas seguintes configurações de ambiente:

- Sistema operacional: Ubuntu 20.04.6 LTS (Focal Fossa)
- Linguagem de programação: C++
- Compilador: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
- Processador: 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz
- Memória: 15Gb

2.2 Estrutura de dados

Para resolução do problema, o grafo foi implementado como uma classe que possui uma "*ListaAdjacencia*" de vértices, ou seja, uma lista encadeada de vértices, em que cada vértice é um "*Node_Adj*" que possui uma lista encadeada interna dos vizinhos que está conectando, uma chave referente a identificação do vértice, e uma cor associada a ele.

Foi adotado a implementação de uma lista encadeada de forma a alocar dinamicamente recursos conforme necessário para armazenar o grafo, evitando a necessidade de um bloco fixo de memória.

2.2.1 Verificar Coloração

Na classe Grafo a função "*EhColoracaoGulosa*" percorre cada vértice na "*ListaAdjacencia*" e primeiro verifica se a cor atribuída ao vértice atual é diferente da cor de seus vizinhos. Se existir dois vértices conectados com cores iguais a função retorna 'false', indicando que a coloração não é válida. Caso contrário, verifica para todo vértice de cor maior que 1 se na sua lista de vizinhos existe vértices com todas as cores menores que a dele. Se uma cor menor que a atual não for encontrada na lista de vizinhos do vértice a função retorna 'false', indicando que ele não está conectado com uma cor menor que a dele.

2.2.1 Ordenação

Depois de verificar que a coloração dos grafos está correta, ocorre a ordenação dos vértices de forma crescente, primeiro em relação a cor e depois em relação a identificação. As implementações dos métodos clássicos de ordenação foram feitas com base nos slides disponibilizados pelo professor, mas manipulando elementos do tipo "*Node_Adj*" com funções da classe "*ListaAdjacencia*".

As principais funções utilizadas para trocar os itens foram "*ListaAdjacencia::trocar_nos*", "*ListaAdjacencia::trocar_nos_das_posicoes*" e "*ListaAdjacencia::recebe*". As operações realizadas pelas primeiras duas funções são feitas através da manipulação de ponteiros, em que ocorre a reorganização dos ponteiros que apontam para os nós na lista encadeada. Já a função "*recebe*" copia os atributos do nó passado por parâmetro para o nó encontrado na posição i. Além disso, a função "*ListaAdjacencia::get_elemento_em*" foi usada para acessar um nó específico na lista encadeada com base em sua posição.

3 ANÁLISE DE COMPLEXIDADE

Abaixo é identificando a complexidades das principais funções do projeto que foram usadas para avaliar a coloração e ordenar os vértices do grafo:

1. **Classe ListaAdjacencia**

1.1. get_elemento_em:

Complexidade de Tempo: O loop while percorre a lista até atingir a posição desejada, e o número de iterações é determinado pelo valor de "*indice*". Portanto, a complexidade de tempo é linear em relação à posição do nó na lista.

Complexidade de Espaço: A função utiliza uma quantidade constante de espaço, independentemente do tamanho total da lista. Ela apenas usa o ponteiro temporário temp. Portanto, a complexidade de espaço é O(1).

1.2. trocar_nos:

Complexidade de Tempo: A função trocar_nos percorre a lista encadeada duas vezes para encontrar os nós com chaves especificadas (chave1 e chave2). O pior caso ocorre quando a lista é percorrida completamente nas duas buscas, resultando em uma complexidade de tempo de O(N), onde N é o número de elementos na lista. As operações de troca de ponteiros e manipulação subsequente são operações de tempo constante (O(1)).

Complexidade de Espaço: A função utiliza apenas ponteiros temporários e variáveis locais, cujo espaço é constante, independentemente do tamanho da lista. Portanto, a complexidade de espaço é O(1).

1.3. trocar_nos_das_posicoes:

Mesma complexidade de tempo e de espaço que a função "*trocar_nos*".

1.4. recebe:

Mesma complexidade de tempo e de espaço que a função "*trocar_nos*".

2. **Classe Grafo**

2.1. EhColoracaoGulosa:

A função percorre todos os vértices do grafo uma vez (laço externo enquanto temp != nullptr). Para cada vértice, há um loop interno que verifica as cores dos vizinhos. Portanto, a complexidade de tempo é O(V + E), onde V é o número de vértices e E é o número de arestas no grafo.

2.2. BubbleSort:

Complexidade de Tempo:

Loop Externo (do-while): O loop externo executa enquanto houver trocas realizadas durante a passagem pela lista. No pior caso, a complexidade de tempo é O(V * E), onde V é o número de vértices e E é o número de arestas.

Loop Interno (while): O loop interno percorre os vértices, comparando e trocando se necessário. No pior caso, o loop interno tem uma complexidade de tempo de O(V), onde V é o número de vértices.

Portanto, a complexidade total de tempo é O(V * E) no pior caso.

Complexidade de Espaço:

A função utiliza uma quantidade constante de espaço adicional, independentemente do tamanho do grafo. As variáveis ocupam uma quantidade fixa de espaço. Além disso, as funções chamadas "*vertices.get_primeiro_elemento()*" e "*vertices.trocar_nos()*" não têm complexidade de espaço significativa. A complexidade de espaço é O(1).

2.3. SelectionSort:

Complexidade de Tempo:

Loop Externo (for): O loop externo percorre a lista de vértices uma vez, realizando comparações e trocas. No pior caso, a complexidade de tempo é O(V^2), onde V é o número de vértices.

Loop Interno (for): O loop interno, aninhado ao loop externo, também percorre a lista para encontrar o vértice com a menor cor. No pior caso, a complexidade de tempo é O(V), onde V é o número de vértices.

Portanto, a complexidade total de tempo é O(V^2) no pior caso.

Complexidade de Espaço:

A função utiliza uma quantidade constante de espaço adicional, pois as variáveis utilizadas (tamanho, indiceMenor, i, j, temp, menor, chave_i, chave_indiceMenor) são todas variáveis locais e ocupam uma quantidade fixa de espaço. Além disso, as funções chamadas "*vertices.get_size()*", "*vertices.get_elemento_em()*", "*vertices.trocar_nos()*" não têm complexidade de espaço significativa. A complexidade de espaço é O(1).

2.4. InsertionSort:

Complexidade de Tempo:

Loop Externo (for): O loop externo percorre a lista de vértices uma vez, realizando comparações e movendo elementos para inseri-los na posição correta. No pior caso, a complexidade de tempo é O(V^2), onde V é o número de vértices.

Loop Interno (while): O loop interno, aninhado ao loop externo, realiza comparações e trocas na lista para posicionar corretamente o elemento atual. No pior caso, a complexidade de tempo é O(V), onde V é o número de vértices.

Portanto, a complexidade total de tempo é O(V^2) no pior caso.

Complexidade de Espaço:

A função utiliza uma quantidade constante de espaço adicional, pois as variáveis utilizadas (i, j, aux, tamanho_vertices, temp) são todas variáveis locais e ocupam uma quantidade fixa de espaço. Além disso, as funções chamadas "*vertices.get_size()*", "*vertices.get_elemento_em()*", "*vertices.trocar_nos()*" não têm complexidade de espaço significativa.

Portanto, a complexidade de espaço é O(1).

2.5. QuickSort:

Complexidade de Tempo:

O algoritmo Quicksort possui as seguintes funções:

- Grafo::Particao(int Esq, int Dir, int *i, int *j):

  Realiza a partição da lista, escolhendo um pivô e rearranjando os elementos em duas partições.
  No pior caso, a função executa em O(n), onde n é o número de elementos na lista.

- Grafo::Ordena(int Esq, int Dir):

  Chama recursivamente a função Particao e realiza a ordenação das partições.
  No pior caso, o algoritmo tem uma complexidade de O(n^2), mas em média é O(n log n).

- Grafo::Quicksort():

  Inicia o processo de ordenação chamando a função Ordena.
  O tempo total é determinado pela complexidade da função Ordena, que é O(n log n) em média.

Portanto, a complexidade de tempo médio do Quicksort é O(n log n), mas no pior caso pode ser O(n^2).

Complexidade de Espaço:

A complexidade de espaço do Quicksort é determinada principalmente pela recursão. Cada chamada recursiva consome espaço na pilha de chamadas. No pior caso, a profundidade da pilha pode ser igual ao número de elementos na lista, sendo O(N).

2.6. Mergesort:

Complexidade de Tempo:

O algoritmo Mergesort possui as seguintes funções:

- Grafo::Merge(int esq, int meio, int dir):

  Mescla duas sub-listas ordenadas (da esquerda e da direita) em uma única lista ordenada. O tempo de execução é proporcional ao número de elementos nas sub-listas, ou seja, O(n), onde n é o número total de elementos nas sub-listas.

- Grafo::Mergesort(int esq, int dir):

Divide recursivamente a lista em sub-listas menores até que cada sub-lista contenha apenas um elemento.
Em seguida, chama a função Merge para mesclar as sub-listas.
O tempo total é determinado pela complexidade da função Merge e pelo número de divisões realizadas.
A complexidade de tempo do Mergesort é O(n log n).

- Grafo::Mergesort():

  Inicia o processo de ordenação chamando a função Mergesort.
  O tempo total é determinado pela complexidade da função Mergesort.
  A complexidade de tempo do Mergesort é O(n log n).

Portanto, a complexidade de tempo médio do Mergesort é O(n log n).

Complexidade de Espaço:

O Mergesort requer espaço adicional para armazenar as sub-listas temporárias durante a mesclagem. No pior caso, é necessário espaço adicional equivalente ao tamanho total da lista. A complexidade de espaço do Mergesort é O(n).

2.7. Heapsort:

Complexidade de Tempo:

O algoritmo Heapsort consiste nas seguintes funções:

- Grafo::Heapsort():
  - Cria uma instância de Heap com o tamanho desejado.
  - Insere as tuplas no heap.
  - Remove e imprime as tuplas do heap em ordem.
  - O tempo de execução é dominado pelas operações de inserção e remoção no heap.

- Heap::Inserir(Tupla t):
  - Insere uma tupla no heap e reorganiza a estrutura para manter a propriedade do heap.
  - A complexidade de tempo é O(log n), onde n é o número de elementos no heap.

- Heap::Remover():
  - Remove o elemento de maior prioridade (raiz) do heap e reorganiza a estrutura para mantê-lo um heap válido.
  - A complexidade de tempo é O(log n), onde n é o número de elementos no heap.

Portanto, a complexidade de tempo total do Heapsort é O(n log n), sendo eficiente para grandes conjuntos de dados.

Complexidade de Espaço:

O Heapsort requer espaço adicional para armazenar as tuplas no heap. No pior caso, é necessário espaço adicional equivalente ao tamanho total da lista. A complexidade de espaço do Heapsort é O(n).

2.8. Meu_metodo_de_ordenacao:

Complexidade de Tempo:

- Utiliza um algoritmo de ordenação bidirecional para ordenar os elementos da lista.
- O laço externo (while) executa até que a lista esteja totalmente ordenada ou que os índices esq e dir se encontrem.
- O laço interno (for) move o maior elemento para o final da lista e o menor elemento para o início alternadamente.
- A complexidade de tempo depende do desempenho do algoritmo de ordenação interno.
- O pior caso do é O(n^2) quando a lista está reversamente ordenada, e o melhor caso é O(n) quando a lista já está ordenada.
- A função “trocar_nos” tem complexidade O(n)

Complexidade de Espaço: É um algoritmo de ordenação in-place, ou seja, não requer espaço adicional significativo além de algumas variáveis auxiliares. A complexidade de espaço é O(1).

4 ESTRATÉGIA DE ROBUSTEZ

No meu projeto, tomei várias medidas para garantir a qualidade do código e a identificação de erros. Aqui estão algumas das principais práticas que adotei:

Tratamento de Exceções: Implementei exceções em funções que podem encontrar erros durante a execução, como verificar nas funções de inserir um elemento nas listas se a posição dada como parâmetro é maior que o tamanho da lista. Isso torna o código mais robusto e permite a identificação e tratamento adequado de problemas, caso ocorram. Além disso, todos as estruturas de dados utilizadas para armazenar informações possuem destrutores próprios para não permitir vazamento de memória.

Testes com Doctest: Criei testes para as classes ListaAdjacencia e ListaEncadeada usando o framework Doctest. Isso incluiu testes de unidade para as funções e métodos, bem como testes de integração para garantir que as classes funcionassem bem juntas. Os testes ajudaram a identificar problemas no código e a verificar se o código estava produzindo os resultados esperados.

Verificação de Memória com Valgrind: Usei a ferramenta Valgrind para verificar a consistência da memória no código. Isso ajudou a garantir que o código não tivesse problemas de gerenciamento de memória que podiam causar falhas ou problemas de desempenho. No final, não foram identificados nenhum erro relacionado a alocação de memória

Apesar disso, deve-se destacar que o programa foi feito com base no enunciado do trabalho que pressupõe que as entradas de dados serão corretas.

5 ANÁLISE EXPERIMENTAL

Para avaliar e comprovar a complexidade de tempo dos diferentes tipos de métodos de ordenação usados no projeto, foram plotados gráficos de tempo de

execução por caso de teste. O número de arestas em cada grafo foi estabelecido com base na densidade do grafo.

A densidade de um grafo é uma medida que indica a proximidade entre o número real de arestas em um grafo e o número máximo possível de arestas em um grafo desse tamanho. Em termos mais simples, a densidade fornece uma medida de quão "cheio" está um grafo em relação ao número total de possíveis conexões entre seus vértices.

A fórmula para calcular a densidade de um grafo não direcionado é dada por:

$$D = \frac{2E}{V(V-1)}$$

onde:

- D é a densidade do grafo,
- E é o número de arestas no grafo,
- V é o número de vértices no grafo.

Essa fórmula é derivada do fato de que em um grafo não direcionado, o número máximo de arestas é $V(V-1)/2$ (grafo completo), e a densidade é a razão entre o número real de arestas (E) e o número máximo possível. O fator 2 no numerador é usado porque cada aresta é contada duas vezes em um grafo não direcionado.

Dessa forma, ao aumentar o número de vértices do grafo e variar o número de arestas com base na densidade há certo ajuste no nível de conectividade entre os elementos, auxiliando na visualização da complexidade dos algoritmos.

O tempo de execução de cada método foi medido pelo uso da biblioteca “Chrono”, e os resultados foram armazenados num arquivo .txt que foi gerado o gráfico por meio de um notebook em Pyhton com uso das bibliotecas “Pandas” e “Matplotlib”. Com o uso do gerador de casos de teste providenciado pelo professor, inicialmente foram criados 20 grafos, variando o número de vértices de 100 a 2000 (foi utilizado um número de vértices reduzido devido ao longo tempo de execução). Esse primeiro experimento é ilustrado na Figura 1 do Apêndice-Figuras. No grupo de algoritmos com piores desempenhos ficou “SelectionSort” e “Meu Método”, no grupo de desempenho médio ficou “Insertionsort” e “Bubblesort”, já o grupo de melhor desempenho foi “Mergesort”, “Quicksort” e “Heapsort”.

Ao analisar o gráfico é notável o impacto da estrutura de dados adotada no tempo de execução de cada algoritmo. A escolha de fazer o grafo como uma ListaAdjacencia impactou principalmente o desempenho de algoritmos que dependem

de acessos sequenciais. Além disso, as várias chamadas da função “*get_elemento_em*” que possui complexidade de tempo O(n), e da função “*troca_nos*” aumentou ainda mais o tempo de execução dos algoritmos, de forma que o “Insertionsort” teve maior complexidade de tempo que o “Bubblesort”.

Algoritmos mais eficientes, como “Mergesort”, “Quicksort” e “Heapsort”, tiveram um desempenho muito superior como o esperado, pois suas estratégias de divisão e conquista minimizam a dependência da natureza encadeada ou sequencial da lista.

Como forma de verificar o impacto da implementação do grafo por meio da ListaAdjacencia e das funções adotadas para retirada e troca de itens, foi criado uma classe Array de tamanho fixo, para ver qual seria a diferença de tempo na ordenação com o seu uso. Assim, o segundo experimento uso essa classe para armazenar e ordenar 29 grafos, variando o número de vértices de 1000 a 30000 (foi utilizado um número de vértices extenso devido ao curto tempo de execução). Esse segundo experimento é ilustrado na Figura 2 do Apêndice-Figuras. No grupo de algoritmos com piores desempenhos ficou “Bubblesort”, “Selection Sort” e “Meu Método”, no grupo de desempenho médio ficou “Insertionsort”, já o grupo de melhor desempenho foi o mesmo “Mergesort”, “Quicksort” e “Heapsort”.

Na figura 2, comparando os algoritmos percebe-se a diferença principalmente no “Insertionsort” que passou a ter uma complexidade de tempo menor que a do “Bubblesort” que aumentou bastante o tempo de execução. Além disso, é evidente a melhora na performance dos algoritmos de ordenação visto que dessa maneira as operações de troca de elementos e acesso a um elemento específicos são constantes (O(1)).

6 CONCLUSÃO

Portanto, o programa criado de acordo com as especificações propostas conseguiu identificar a cloração do grafo e ordenar ele de forma correta, com uso de uma lista encadeada. Contudo, foi verificado que a implementação dos métodos da lista de maneira que a troca de elementos e o acesso a um elemento específicos possuem complexidade O(n) aumentou bastante o tempo de execução dos algoritmos de ordenação. Assim sendo, é importante avaliar o contexto da aplicação de forma generalizada e traçar um planejamento inicial mais elaborado com o objetivo de escolher a melhor opção de algoritmo para atender aos requisitos do problema. Em particular, como no projeto foi estabelecido que seria passado o tamanho do grafo previamente, logo implementar ele por meio de Array de tamanho fixo oferece maior facilidade em manipular os elementos armazenados.

BIBLIOGRAFIAS

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Flavio Vinicius Del Padre. PDS2 - UFMG. Disponível em: https://flaviovdf.io/pds2-ufmg/. Acesso em: 01/11/2023.

GeeksforGeeks. Sorting Algorithms. Disponível em: https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/. Acesso em: 01/11/2023.

BACKES, P. G. Aula 06 - Ordenação. Universidade Federal de Uberlândia, Faculdade de Computação. Disponível em: <https://www.facom.ufu.br/~backes/facom39401/Aula06-Ordenacao.pdf\>. Acesso em: 01/11/2023.

Professor Mario. Algoritmos Gulosos. Maio 14, 2020. Disponível em: <https://yewtu.be/watch?v=Qyv20alPqec\>. Acesso em: 01/11/2023.

APÊNDICE – INSTRUÇÕS PARA COMPILAÇÃO

Para executar o código do trabalho deve ser feito os seguintes passos. Na pasta raiz do projeto execute o comando ‘make al’ para realizar a compilação do programa. Para rodar o arquivo executável gerado acesse a pasta ‘bin’ do projeto que contém o executável ‘tp2.out’. Para apagar o executável na pasta ‘bin’ e os arquivos ‘.o’ gerados na compilação, digite ‘make clean’.

1. Compilação do programa:
    - $ make all
2. Execução do programa:
    - $ ./bin/tp2.out
3. Limpeza dos arquivos:
    - $ make clean

APÊNDICE – FIGURAS

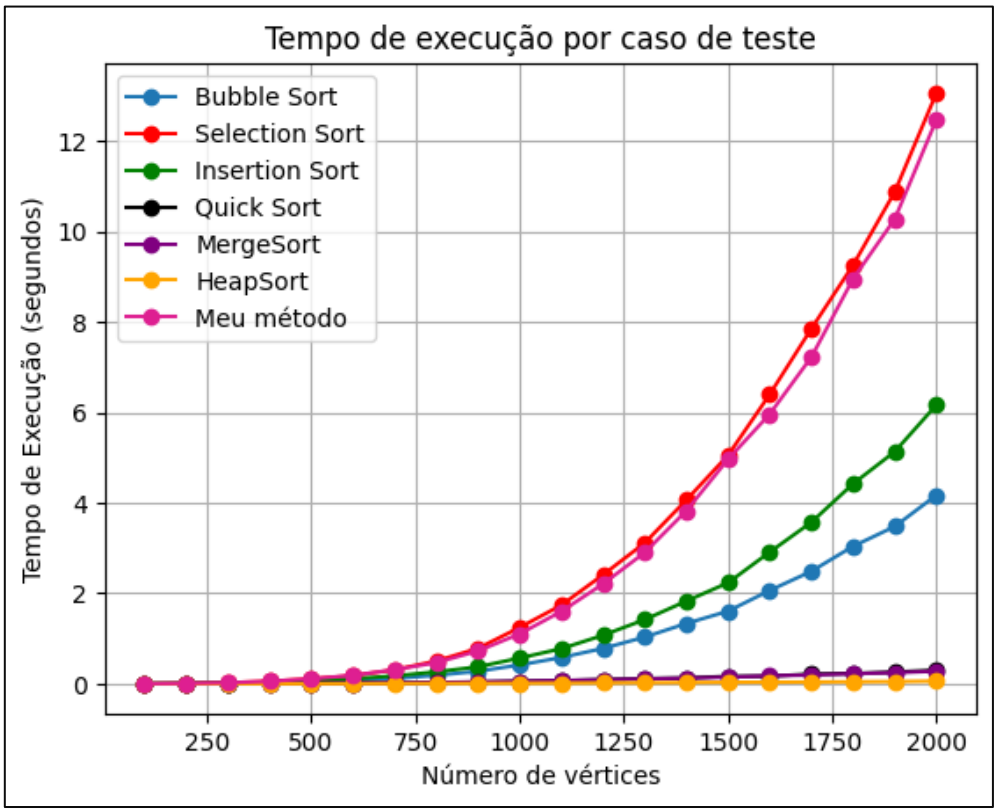

Figura 1

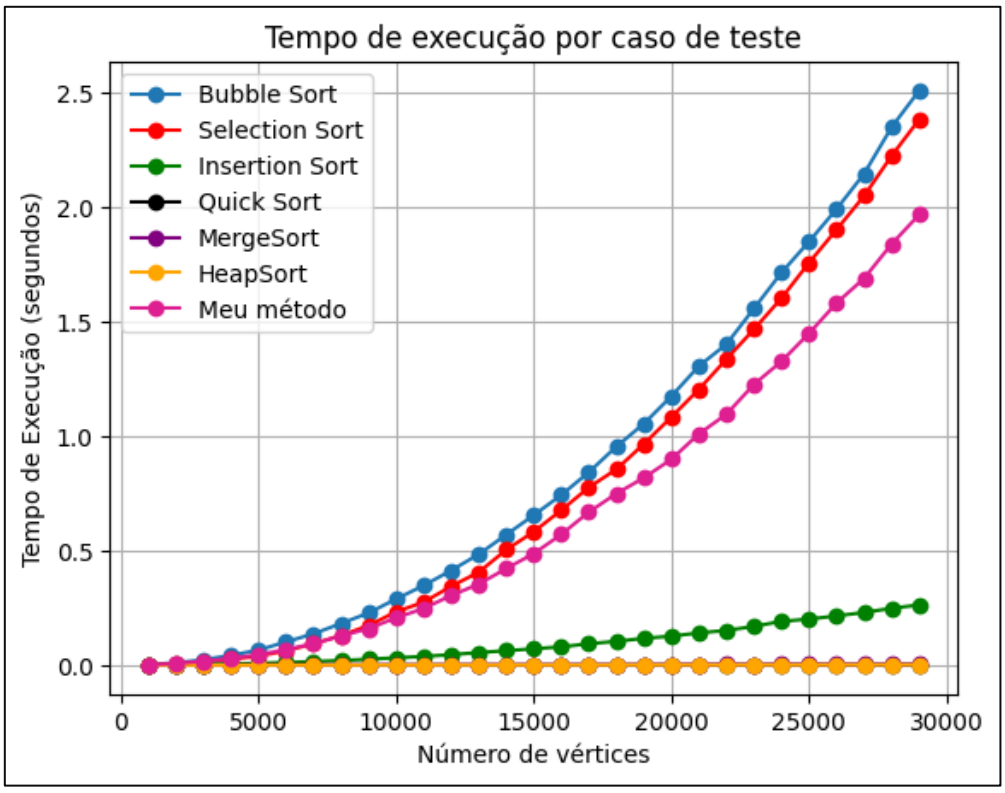

Figura 2