Trabalho Prático 2
Essa coloração é gulosa?

Raissa Gonçalves Diniz
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG – Brasil
raissagdiniz@gmail.com

1. Introdução:

O objetivo deste trabalho é explorar e entender se toda coloração própria de um grafo k-colorível é, por natureza, uma coloração gulosa. Uma coloração própria de um grafo é uma atribuição de cores aos seus vértices de tal forma que nenhum par de vértices adjacentes — ou seja, vértices que são conectados por uma aresta — compartilhe a mesma cor. Já uma coloração gulosa é um método específico para colorir um grafo de maneira própria, em que cada vértice é colorido com a menor cor disponível que ainda não foi utilizada por nenhum de seus vizinhos.

Resumidamente, o programa recebe um grafo através do terminal, seguindo uma certa estrutura, com todos os valores, cores e ligações do grafo. Então, é feita uma análise se a coloração passada é gulosa. Caso não seja, o usuário verá um “0” na tela. Caso seja, o programa retorna “1” seguido dos vértices do grafo, ordenados, primeiramente, por cor (representadas por números inteiros), e, caso alguns deles tenham a mesma cor, por valor.

Logo, o cerne desta investigação gira em torno não só de determinar a relação entre estas duas abordagens de coloração, mas também, especialmente, de como lidar com a ordenação dos vértices do grafo de maneira mais rápida e eficiente. Para isso, foram analisados 6 algoritmos de ordenação mundialmente conhecidos (são eles: bubble sort, selection sort, insertion sort, quicksort, mergesort e heapsort) e, baseado no estudos destes, foi-se proposto um novo algoritmo customizado, levando em consideração as especificidades desse problema.

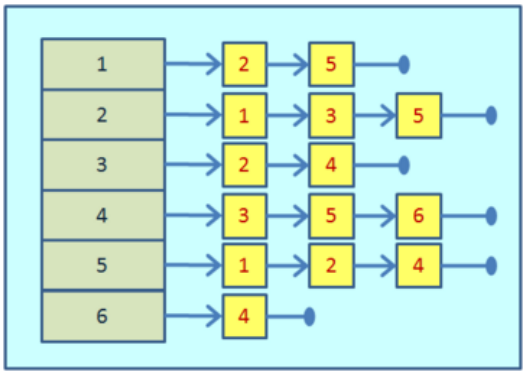
2. Método:

- Programa desenvolvido em C++
- Compilador g++ versão 9.4.0
- Windows 10 Home x64
- Linux Ubuntu 20.04

Especificações da máquina utilizada:
- Processador: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
- RAM: 8 GB

2.1. Estrutura de Dados

Para representar do grafo a ser analisado, foi utilizada uma lista de adjacências, em que cada vértice do grafo possui uma lista associada que armazena seus vizinhos. Exemplo:



O grafo da figura acima tem essa representação de lista de adjacência:		
1	adjacente a	2,5
2	adjacente a	1,3,5
3	adjacente a	2,4
4	adjacente a	3,5,6
5	adjacente a	1,2,4
6	adjacente a	4

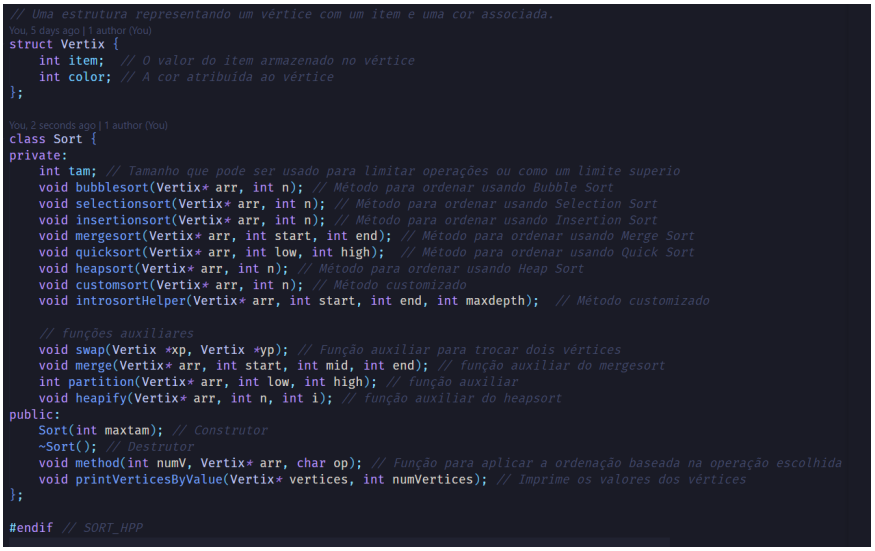
Essa estrutura de dados foi escolhida em detrimento à matriz de adjacências, outro método comum quando se trata de armazenamento de grafos, pois ela usa espaço proporcional ao número de arestas, o que as torna mais eficientes em termos de espaço para grafos esparsos. Além disso, como o grafo só é criado após o usuário passar os valores de seus vértices e cores no terminal, a implementação com ponteiros faz com que ele não só não precise ser inicialmente declarado com um tamanho fixo, como não são necessárias novas realocações durante a adição de novos vértices.

2.2. Funções e classes

Para a implementação do grafo, foram declarada duas classes: a classe Graph, que contém seu número de vértices, um array de cores (que são representadas por números inteiros), sua lista encadeada e seus principais métodos para inserção de vértices e cores, determinação se é guloso ou não, etc, e uma classe Node, usada para implementar os elementos da lista encadeada. A lista encadeada, em si, não está separadamente declarada, mas é utilizada ao longo dos métodos do grafo.

```
// Estrutura nó, usada na lista encadeada
You, 1 second ago | 1 author (You)
class Node {
public:
    int data; // Valor do nó (ou vértice)
    int color; // Cor do nó (ou vértice)
    Node* next; // Ponteiro para o próximo elemento
};
You, 1 second ago | 1 author (You)
class Graph {
public:
    int numVertices; // Número de vértices
    int* colors; // array o colors;
    Node** adjList; // Lista de adjacência

    Graph(); // Construtor
    ~Graph(); // Destrutor
    void insertVertex(); // Inserir um vértice no grafo
    void insertEdge(int v, int w); // Inserir uma aresta no grafo
    int getNumberOfVertices() const; // Retorna o número de vértices
    int getNumberOfEdges() const; // Retorna o número de arestas
    void printNeighbors(int v) const; // Imprime um vértice e seus "vizinhos"
    void addColor(int v, int c); // Atribui a cor 'c' ao vértice 'v'
    bool isGreedy(int v, int c); // Retorna se a coloração é gulosa ou não
};
```

A outra classe implementada foi denominada Sort, e ela é quem lida com os métodos de ordenação dos vértices do grafo na hora da impressão.



```
// Uma estrutura representando um vértice com um item e uma cor associada.
You, 5 days ago | 1 author (You)
struct Vertix {
    int item;   // O valor do item armazenado no vértice
    int color; // A cor atribuída ao vértice
};

You, 2 seconds ago | 1 author (You)
class Sort {
private:
    int tam; // Tamanho que pode ser usado para limitar operações ou como um limite superio
    void bubblesort(Vertix* arr, int n); // Método para ordenar usando Bubble Sort
    void selectionsort(Vertix* arr, int n); // Método para ordenar usando Selection Sort
    void insertionsort(Vertix* arr, int n); // Método para ordenar usando Insertion Sort
    void mergesort(Vertix* arr, int start, int end); // Método para ordenar usando Merge Sort
    void quicksort(Vertix* arr, int low, int high);  // Método para ordenar usando Quick Sort
    void heapsort(Vertix* arr, int n);  // Método para ordenar usando Heap Sort
    void customsort(Vertix* arr, int n); // Método customizado
    void introsortHelper(Vertix* arr, int start, int end, int maxdepth);  // Método customizado

    // funções auxiliares
    void swap(Vertix *xp, Vertix *yp); // Função auxiliar para trocar dois vértices
    void merge(Vertix* arr, int start, int mid, int end); // função auxiliar do mergesort
    int partition(Vertix* arr, int low, int high); // função auxiliar
    void heapify(Vertix* arr, int n, int i); // função auxiliar do heapsort
public:
    Sort(int maxtam); // Construtor
    ~Sort(); // Destrutor
    void method(int numV, Vertix* arr, char op); // Função para aplicar a ordenação baseada na operação escolhida
    void printVerticesByValue(Vertix* vertices, int numVertices); // Imprime os valores dos vértices
};

#endif // SORT_HPP
```

O struct de Vértices foi utilizado para criar um array com todos os vértices, facilitando a ordenação, que foi feita, primeiramente, por cor, e depois pelo valor. Sobre cada método de ordenação:

- **Bubble sort:**

Estabilidade: Estável, pois ele compara elementos adjacentes e apenas os troca se o elemento anterior for estritamente maior que o próximo. Não são necessárias modificações para a estabilidade.

Funcionamento: Compara pares adjacentes e troca suas posições se estiverem na ordem errada. Esse processo se repete até que a lista esteja ordenada.

- **Selection sort:**

Estabilidade: Instável.

Funcionamento: Encontra o menor (ou maior, dependendo da ordem desejada) elemento da lista e o troca com o primeiro elemento não ordenado, repetindo o processo até que toda a lista esteja ordenada.

O Selection Sort foi transformado em um algoritmo de ordenação estável ao introduzir uma inserção ordenada do menor elemento encontrado, em vez de trocá-lo com o primeiro elemento não ordenado. Agora, ele verifica se a cor do elemento é menor e, em caso de cores iguais, a ordem é decidida pelo valor do item, mantendo assim a ordem original dos elementos com cores iguais.

- **Insertion sort:**

Estabilidade: Estável, pois ele insere cada novo elemento na posição correta em relação aos elementos já ordenados, mantendo a ordem dos elementos iguais.

Funcionamento: Constrói uma lista ordenada um elemento de cada vez, pegando cada elemento da parte não ordenada e inserindo-o na posição correta na parte ordenada.

- **Quicksort:**

Estabilidade: Instável.

Funcionamento: Escolhe um elemento como pivô e particiona a lista em torno do pivô, colocando todos os elementos menores antes dele e todos os maiores depois. Repete o processo de forma recursiva para as sublistas.

O Quick Sort foi modificado no meu código para ser estável adicionando uma condição de desempate. Se dois elementos têm a mesma cor, a ordem é decidida pelo valor do item.

- **Mergesort:**

Estabilidade: Estável. O processo de merge preserva a ordem dos elementos iguais porque sempre pega o elemento da esquerda primeiro quando há um empate.

Funcionamento: Divide a lista em pares de elementos, ordena esses pares e então mescla os pares ordenados em listas maiores até que toda a lista esteja ordenada.

- **Heapsort:**

Estabilidade: Instável (modificações para torná-lo estável não são triviais e não foram implementadas).

Funcionamento: Transforma a lista em um heap (estrutura de dados tipo árvore), então remove repetidamente o maior elemento do heap e o coloca na parte ordenada da lista.

As comparações consideram a cor e, em caso de empate, o valor do item. No entanto, o Heapsort é por natureza instável e as alterações mencionadas para estabilidade não foram aplicadas ao código.

- **Método customizado:**

Após a análise de todos esses algoritmos, concluímos que, para esse problema, queremos um método de ordenação que tenha um bom desempenho médio (como o Quick Sort) e que também seja estável (como o Merge Sort) para manter a ordem dos vértices com a mesma cor pelo seu número. No entanto, cada algoritmo de ordenação fornecido tem seus pontos fortes e fracos dependendo da natureza dos dados:

- *Bubble Sort*: Simples, mas muito ineficiente para listas grandes (complexidade O(n^2)).
- *Selection Sort*: Também possui complexidade O(n^2), ineficiente para listas grandes.
- *Insertion Sort*: Bom para listas pequenas ou quase ordenadas, mas o caso médio e o pior caso são O(n^2).
- *Merge Sort*: Eficiente (O(n log n)) e estável, mas requer espaço extra para arrays.
- *Quick Sort*: Eficiente (O(n log n) em média), mas não estável e com pior caso O(n^2).
- *Heap Sort*: Eficiente (O(n log n)), mas não estável.

Logo, para esse caso, a abordagem utilizada foi um algoritmo que utiliza as vantagens dos métodos de ordenação acima com base na natureza e tamanho dos dados. Esse método de ordenação, que é “híbrido”, começa com o Quicksort e muda para Heapsort quando a profundidade da recursão excede um nível baseado no logaritmo do número de elementos a serem ordenados (evitando o pior caso do Quicksort, que é O(n^2)). Além disso, para partições muito pequenas, o algoritmo utiliza o Insertion Sort, que é mais eficiente para conjuntos de dados menores.

Esta abordagem tira as melhores partes de diferentes algoritmos

```
// Função personalizada para ordenação que usa Introsort
void Sort::customsort(Vertix* arr, int n) {
    customSortHelper(arr, 0, n - 1, 2 * log(n));
}

// Assistente recursivo para a execução do Introsort
void Sort::customSortHelper(Vertix* arr, int start, int end, int maxdepth) {
    if (end - start < 32) {
        insertionsort(arr + start, end - start + 1); // Insertion Sort para pequenas partições
    } else if (maxdepth == 0) {
        heapsort(arr + start, end - start + 1); // Heap Sort se a profundidade de recursão for muito alta
    } else {
        int pivot = partition(arr, start, end); // Calcula o pivô para a partição
        customSortHelper(arr, start, pivot - 1, maxdepth - 1); // Ordena os elementos antes do pivô
        customSortHelper(arr, pivot + 1, end, maxdepth - 1); // Ordena os elementos depois do pivô
    }
}
```

A profundidade máxima da recursão é determinada por 2 * log(n), onde n é o número de elementos a serem ordenados, garantindo uma complexidade de tempo geral esperada de O(n log n) com um uso de memória eficiente.

3. Análise de complexidade (de tempo e espaço)

Sobre o grafo:

1. *Graph() (Construtor):* Tempo O(1), Espaço O(1) - Inicializa variáveis.
2. *~Graph() (Destrutor):* Tempo O(n + m), Espaço O(1) - Percorre todas as listas de adjacências liberando os nós.
3. *insertVertex():* Tempo O(n), Espaço O(n) - Cria uma nova lista de adjacências e um novo array de cores, ambos de tamanho n+1, copiando os elementos antigos.
4. *insertEdge(int v, int w):* Tempo O(1), Espaço O(1) - Insere uma nova aresta no início da lista de adjacências, o que é uma operação constante.
5. *getNumberOfVertices():* Tempo O(1), Espaço O(1) - Retorna o valor de um membro da classe.
6. *getNumberOfEdges():* Tempo O(n + m), Espaço O(1) - Percorre todas as listas de adjacência para contar as arestas.
7. *printNeighbors(int v):* Tempo O(deg(v)), Espaço O(1) - Onde deg(v) é o grau do vértice v. Percorre os vizinhos de um único vértice.
8. *addColor(int v, int c):* Tempo O(1), Espaço O(1) - Atribui uma cor a um vértice, uma operação constante.
9. *isGreedy(int v, int c):* Tempo O(c * deg(v)), Espaço O(1) - Para um dado vértice v, verifica se a coloração até a cor c é gulosa, necessitando percorrer os vizinhos de v até c vezes.

Sobre cada método de ordenação:

1. Bubble Sort:

Complexidade de Tempo: Melhor caso: O(n) (quando o array já está ordenado e o algoritmo faz uma passagem sem trocas).

Caso médio e pior caso: $O(n^2)$ (onde n é o número de elementos a serem ordenados).

Complexidade de Espaço: O(1) (ordenamento no próprio array, sem uso de espaço extra significativo).

2. Selection Sort:

Complexidade de tempo: Melhor, médio e pior caso: O(n²) (independente da ordenação inicial do array).

Complexidade de Espaço: O(1) (ordenamento in-place).

3. Insertion Sort:

Complexidade de Tempo: Melhor caso: O(n) (quando o array está ordenado).

Caso médio e pior caso: O(n²) (quando o array está em ordem inversa ou aleatória).

Complexidade de Espaço: O(1) (não precisa de espaço adicional significativo).

4. Quicksort:

Complexidade de Tempo: Melhor caso e caso médio: O(n log n) (partições são balanceadas).

Pior caso: O(n²) (quando as partições são desbalanceadas, por exemplo, quando o pivot é sempre o menor ou maior elemento).

Complexidade de Espaço: Pior caso: O(n) (quando a recursão atinge a maior profundidade).

Melhor caso: O(log n) (com partições balanceadas).

5. Mergesort:

Complexidade de Tempo: Melhor, médio e pior caso: O(n log n) (particiona o array e depois faz as fusões de maneira estável).

Complexidade de Espaço: O(n) (precisa de espaço adicional para armazenar arrays temporários durante a ordenação).

6. Heapsort:

Complexidade de Tempo: Melhor, médio e pior caso: O(n log n) (cria uma estrutura de heap e então extrai os elementos).

Complexidade de Espaço: O(1) (in-place, mas a estrutura de heap é mantida no próprio array).

7. Customsort:

Complexidade de Tempo: Melhor, médio e pior caso: O(n log n) (combinação de Quicksort, Heapsort e Insertion Sort para evitar o pior caso do Quicksort).

Complexidade de Espaço: Pior caso: O(log n) (por causa do uso do Quicksort, que é a principal parte do Introsort).

4. Estratégias de robustez

O código do grafo implementa várias medidas para assegurar a sua robustez durante a manipulação de dados e gestão de memória. Quando novos vértices ou cores são adicionados, o código emprega o operador new com std::nothrow para evitar exceções em caso de falha na alocação de memória, permitindo a checagem segura de um ponteiro nulo antes de prosseguir. Além disso, a inserção de arestas e a atribuição de cores só acontecem após verificar se os índices dos vértices são válidos, prevenindo assim acessos fora dos limites do array. Todos os novos vértices são inicializados com valores inválidos para garantir um estado conhecido e controlado. O destruidor da classe cuidadosamente limpa a memória, evitando vazamentos ao deletar cada nó da lista de adjacências e os arrays associados. As operações que imprimem os vizinhos ou adicionam cores são protegidas por verificações de validade, e o método que avalia a coloração gulosa verifica cuidadosamente as cores dos vértices e de seus vizinhos para determinar a corretude da coloração. Essas precauções contribuem para a robustez do código, reduzindo o risco de erros comuns relacionados à memória e à lógica de programação.

5. Análise Experimental

Neste experimento, foi realizada uma análise experimental para comparar o desempenho dos diferentes algoritmos de ordenação. O objetivo era medir o tempo de execução médio de cada algoritmo para diferentes tamanhos de entrada, representados pelo número de vértices dos grafos. Os algoritmos avaliados foram: bubble sort, selection sort, insertion sort, quicksort, mergesort, heapsort e o método customizado.

Inicialmente, a análise experimental foi utilizada para avaliar o uso de memória durante a execução do programa, permitindo identificar possíveis vazamentos de memória e otimizar a alocação e desalocação de memória pelo programa. No caso do código apresentado, foi utilizado o Valgrind para realizar essa análise, uma ferramenta que permite identificar vazamentos de memória. Para utilizar o Valgrind, basta executar o programa com o comando "valgrind nome_do_programa". O Valgrind irá monitorar o uso de memória pelo programa e exibir possíveis erros de vazamento de memória. Ele foi executado com as opções *"--leak-check=full"* e *“--tool=cachegrind”* para realizar uma verificação completa de vazamentos de memória. A versão específica utilizada foi a 3.15.0.

Após a execução do programa, o Valgrind gerou um resumo do heap, mostrando que 12.002 bytes estão em uso em 1 bloco. Também forneceu um resumo de vazamento, indicando que não há vazamentos definitivamente perdidos ou indiretamente perdidos (ou seja, não há memória alocada que não possa ser acessada ou liberada). Além disso, não

```
==145== HEAP SUMMARY:
==145==     in use at exit: 12,002 bytes in 1 blocks
==145==   total heap usage: 3 allocs, 2 frees, 85,730 bytes allocated
==145==
==145== LEAK SUMMARY:
==145==    definitely lost: 0 bytes in 0 blocks
==145==    indirectly lost: 0 bytes in 0 blocks
==145==      possibly lost: 0 bytes in 0 blocks
==145==    still reachable: 12,002 bytes in 1 blocks
==145==         suppressed: 0 bytes in 0 blocks
==145== Reachable blocks (those to which a pointer was found) are not shown.
==145== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==145==
==145== For lists of detected and suppressed errors, rerun with: -s
==145== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
==146== I   refs:      2,402,378
==146== I1  misses:        1,675
==146== LLi misses:        1,624
==146== I1  miss rate:      0.07%
==146== LLi miss rate:      0.07%
==146==
==146== D   refs:        750,264  (545,400 rd   + 204,864 wr)
==146== D1  misses:       17,054  ( 14,362 rd   +   2,692 wr)
==146== LLd misses:       10,047  (  8,265 rd   +   1,782 wr)
==146== D1  miss rate:       2.3% (    2.6%     +     1.3%  )
==146== LLd miss rate:       1.3% (    1.5%     +     0.9%  )
==146==
==146== LL refs:          18,729  ( 16,037 rd   +   2,692 wr)
==146== LL misses:        11,671  (  9,889 rd   +   1,782 wr)
==146== LL miss rate:        0.4% (    0.3%     +     0.9%  )
```

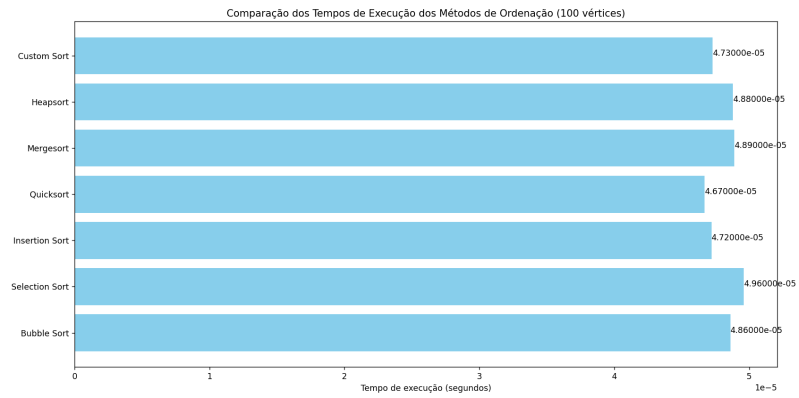
foram identificados possíveis vazamentos ou blocos suprimidos, como mostrado nas imagens:

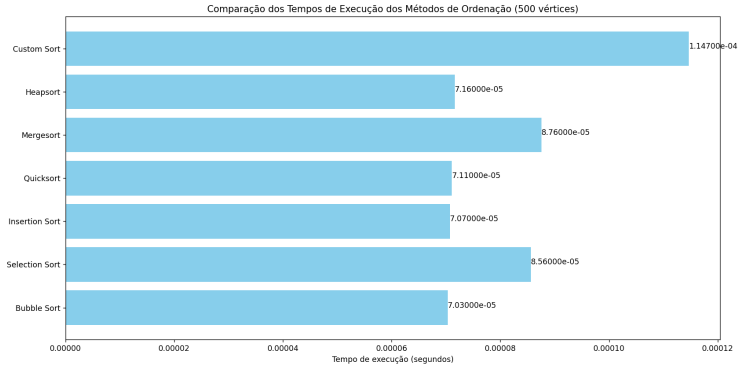
Para a análise dos algoritmos de ordenação, diferentes partes do programa foram perfiladas separadamente, de modo que os processos de criação e armazenamento dos grafos, além da verificação da coloração gulosa, que possuem diferentes tempos de execução, não “ofuscassem” a real análise do tempo de execução dos algoritmos. Outrossim, o código “GeradorCasosDeTeste”, fornecido no Moodle, foi utilizado para gerar grafos de tamanhos específicos para os testes.

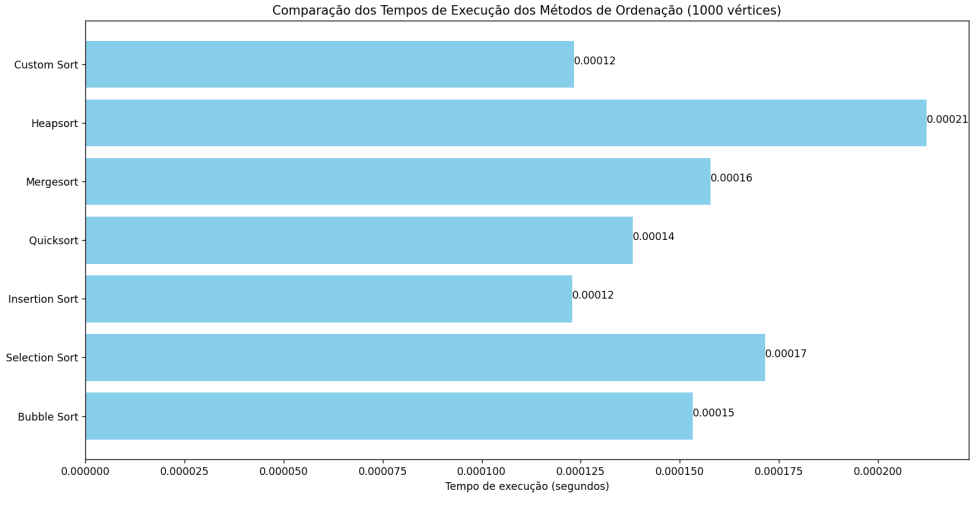
Para isso, o código de medição de tempo foi inserido manualmente em torno das seções do código desejadas, utilizando a biblioteca “*std::chrono”*, que é uma biblioteca de data e hora do C++ Standard Library. O código foi instrumentado em torno do momento em que as ordenações dos vértices do grafo são feitas, e entradas com diversos tamanhos foram passadas para cada um dos métodos, tal qual a seguinte imagem:

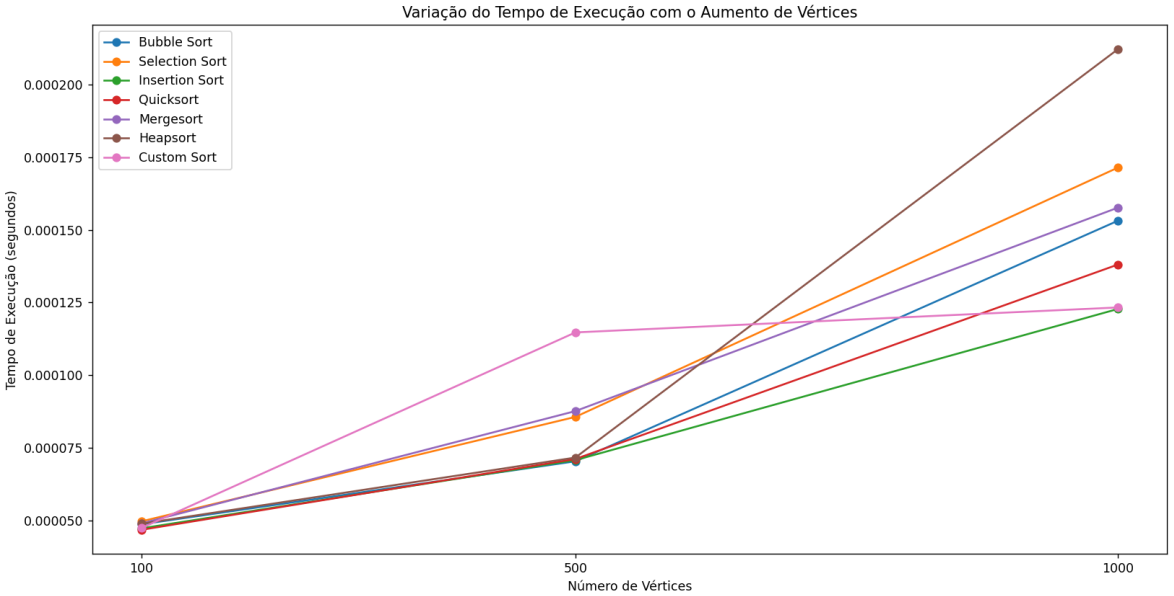
```cpp
auto inicio1 = std::chrono::high_resolution_clock::now();
sort.printVerticesByValue(vertices, numV);  // Imprime os vértices ordenados.
auto fim1 = std::chrono::high_resolution_clock::now();

std::chrono::duration<double> tempo_exec1 = fim1 - inicio1;
std::cout << "Tempo de execução Ordenação 1: " << tempo_exec1.count() << " segundos\n";
```

Com os dados já coletados, a biblioteca Matplotlib, da linguagem Python, foi utilizada para a geração dos gráficos. Caso deseje testá-la, você precisa ter tanto o Python instalado, como a biblioteca (abra um terminal ou prompt de comando e execute o seguinte comando: *“pip install matplotlib”*). Um código em python identificando como a geração dos gráficos deve ser feita foi criado, e executado com o comando “*python graph_plot.py”.* Os resultados são apresentados a seguir.









Com base no gráfico gerado, podemos realizar algumas análises sobre o comportamento dos algoritmos de ordenação conforme o tamanho da entrada aumenta:

1. **Bubble Sort, Selection Sort e Insertion Sort:**

Estes algoritmos apresentam um crescimento mais acentuado no tempo de execução conforme o número de vértices aumenta, o que é esperado, uma vez que são algoritmos de ordenação com complexidade quadrática no pior caso ($O(n^2)$).

O Bubble Sort parece ter o crescimento mais rápido no tempo de execução, o que sugere que ele pode ser o menos eficiente entre eles para entradas maiores.

2. **Quicksort e Mergesort:**

Estes métodos têm um crescimento mais moderado e são conhecidos por serem mais eficientes, com um tempo de execução médio de $O(n \log n)$. Isso é refletido no gráfico pela inclinação mais suave das linhas correspondentes a esses métodos.

Mergesort parece ter um leve aumento na taxa de crescimento para a entrada de 1000 vértices em comparação com Quicksort, mas ainda assim é significativamente mais eficiente que os métodos de ordenação quadrática.

3. **Heapsort:**

O Heapsort também tem complexidade $O(n \log n)$, mas o gráfico mostra um aumento significativo no tempo de execução para a entrada de 1000 vértices. Isso pode ser devido a constantes maiores no tempo de execução ou a peculiaridades específicas dos dados que podem não se alinhar bem com a estratégia de ordenação do Heapsort.

4. **Custom Sort:**

A linha que representa o Custom Sort (um algoritmo de ordenação personalizado) apresenta um comportamento interessante: ele começa sendo comparável aos algoritmos de ordenação eficientes para 100 vértices, mas seu tempo de execução aumenta dramaticamente para 500 vértices, e então tem um crescimento mais lento novamente para 1000 vértices.

Isso sugere que o Custom Sort pode ter uma estratégia que lida de maneira variável com diferentes tamanhos de entrada ou que há aspectos dos dados que afetam seu desempenho de maneiras não uniformes.

É evidente que os algoritmos com complexidade log-linear (Quicksort, Mergesort, Heapsort) geralmente se comportam melhor e são mais escaláveis com o aumento do tamanho da entrada do que os algoritmos com complexidade quadrática (Bubble Sort, Selection Sort, Insertion Sort).

A escolha do algoritmo de ordenação deve levar em conta não apenas o tamanho da entrada, mas também a natureza dos dados e o contexto de uso. Por exemplo, em ambientes onde as entradas são geralmente pequenas ou quase ordenadas, algoritmos como Insertion Sort podem se sair bem.

6. Conclusão

Através do trabalho prático realizado, pude explorar em profundidade o conceito de colorações próprias em grafos e sua relação com algoritmos gulosos. A investigação central questionou se toda coloração própria de um grafo k-colorível é necessariamente uma coloração gulosa, o que levou a uma compreensão mais refinada das propriedades de coloração em grafos e o papel de algoritmos gulosos em sua identificação, não só através da utilização de diferentes estruturas de dados para o armazenamento dos grafos, como também do raciocínio lógico necessário para realizar essa avaliação (e até mesmo para criar um novo método).

7. Referências

What is Sorting in C++: Bubble Sort, Insertion Sort, Merge Sort & More. Disponível em: <https://www.simplilearn.com/tutorials/cpp-tutorial/sorting-in-cpp>. Acesso em: 24 out. 2023.

Sorting Algorithms. Disponível em: <geeksforgeeks.org/sorting-algorithms/, Https://www;>. Acesso em: 24 out. 2023.

Exemplo de Lista de Adjacência [Imagem]. Wikipedia. https://pt.wikipedia.org/wiki/Lista_de_adjac%C3%AAncia. Acesso em: 03 nov. 2023.

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.