Essa coloração é gulosa?

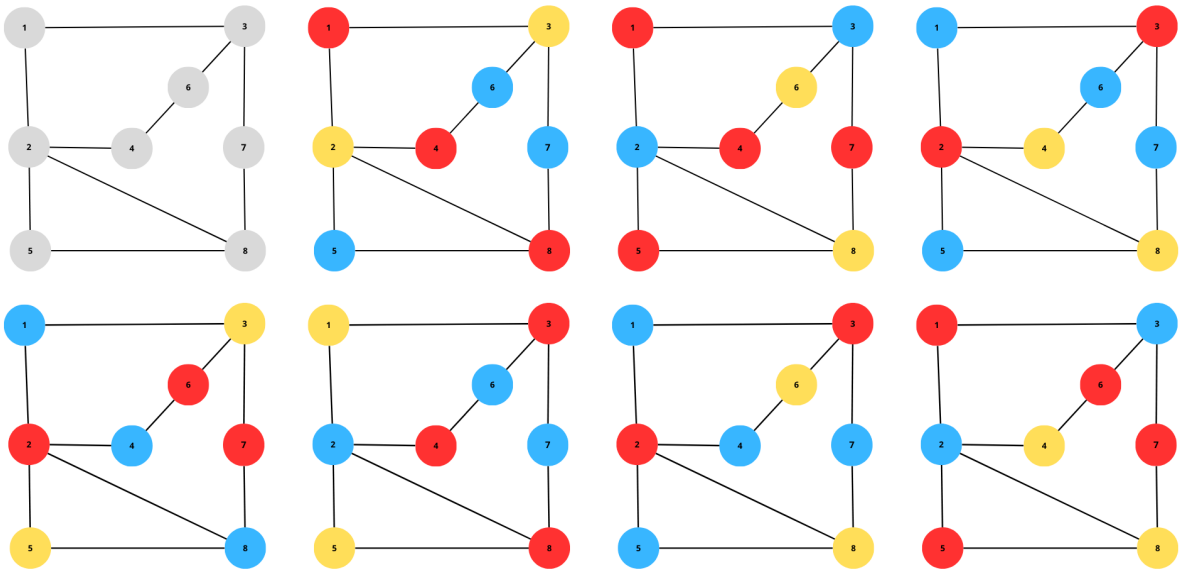
Trabalho Prático 2

1. Introdução

Nesse Trabalho Prático, exploramos a verificação da gulosidade de colorações de grafos, um problema fundamental na teoria dos grafos. Em vez de atribuir cores aos vértices, nosso foco foi determinar se uma dada coloração é gulosa, ou seja, se pode ser obtida por um algoritmo guloso. Um algoritmo guloso é uma abordagem heurística que faz escolhas locais ótimas em cada etapa, com o objetivo de alcançar uma solução globalmente ótima. No contexto deste trabalho, a gulosidade refere-se à capacidade de um conjunto de vértices em um grafo ser colorido de forma que cada vértice esteja conectado a pelo menos um vértice de cada cor menor que a dele.

Caso o grafo analisado seja guloso, implementamos métodos clássicos de ordenação, como bolha, seleção, inserção, quicksort, mergesort e heapsort, além de desenvolver um método de ordenação próprio adaptado ao problema, para ordenar os vértices do grafo de acordo com a ordem de preenchimento de suas cores. Investigamos diferentes conjuntos de entradas para avaliar o desempenho dos algoritmos, considerando propriedades como adaptabilidade e estabilidade. A análise comparativa dos métodos permitiu identificar suas vantagens e limitações, destacando a importância da escolha adequada do método de ordenação para resolver o problema da gulosidade de colorações de grafos.

Adicionalmente, para ilustrar a relevância da escolha do vértice inicial na gulosidade, foi feito um exemplo prático utilizando um grafo específico. Ao explorar diferentes estratégias de escolha dos vértices iniciais para a aplicação do algoritmo guloso, podemos observar variações significativas nas colorações obtidas. A capacidade de representar essas variações visualmente por meio de gráficos coloridos fornece uma compreensão mais profunda de como essa seleção impacta diretamente no resultado final.



2. Método

2.1. Descrição da Implementação

O programa foi desenvolvido em C++ e implementa um algoritmo de ordenação relacionado ao problema de coloração de grafos. O código se baseia na estrutura de dados de um grafo representado por vértices e suas arestas (implícitas pela lista de adjacência). Cada vértice contém informações sobre sua cor, grau e lista de adjacência.

Para resolver o problema de coloração de grafos, o programa implementa a função Guloso, que verifica se um vértice está conectado a pelo menos um outro vértice de cada cor abaixo da sua (as cores são ordenadas pelo seu valor). Além disso, o programa oferece opções para escolha de diferentes algoritmos de ordenação, como Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort e uma implementação personalizada de ordenação. Esses algoritmos são escolhidos para ordenar a ordem em que os vértices do grafo foram coloridos.

2.2. Estruturas de Dados

Vertice:

A estrutura de dados Vertice representa um vértice em um grafo, contendo informações essenciais sobre o vértice, como seu identificador único, a cor atribuída, o grau e uma lista de adjacência.

2.3. Funções e Métodos Implementados do main.cpp

As principais funções são:

Guloso(Vertice* vertices[], int n)

O algoritmo Guloso verifica se a coloração atual dos vértices é válida de acordo com a condição gulosa. Para cada vértice, o algoritmo verifica se o vértice está conectado a pelo menos um vértice de cada cor menor que a sua própria cor.

Laço Externo:

- O algoritmo percorre todos os vértices no grafo.

Laço Interno:

- Para cada vértice, o algoritmo verifica se ele está conectado a pelo menos um vértice de cada cor menor que a sua própria cor.
- Para isso, ele itera sobre a lista de adjacência do vértice atual e verifica se existem vizinhos de cada cor menor. Isso é feito através da comparação das cores dos vizinhos com cores de 1 até "vertices[k]->cor - 1", onde "k" é o índice do vértice atual.

Verificação:

- Se, para algum vértice, não existir pelo menos um vizinho de cada cor menor, o algoritmo retorna "false", indicando que a coloração não é válida segundo a condição gulosa.

Retorno:

- Se o algoritmo percorrer todos os vértices sem encontrar violações na condição gulosa, retorna "true", indicando que a coloração é válida.

Estabilizar(Vertice* vertices[], int n)

A função “Estabilizar” é utilizada nos algoritmos de ordenação para garantir a estabilidade do processo, ou seja, preservar a ordem relativa dos vértices com chaves iguais durante a ordenação. Essa função compara os vértices adjacentes e, se eles tiverem a mesma cor, verifica se a ordem relativa entre eles é a mesma da posição inicial no array. Se não for, realiza a troca para manter a ordem estável.

BubbleSort(Vertice* vertices[], int n)

O algoritmo de ordenação Bubble Sort percorre a lista de vértices várias vezes, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Esse processo é repetido até que a lista esteja totalmente ordenada. A função "Troca" é usada para trocar dois vértices.

SelectionSort(Vertice* vertices[], int n)

O algoritmo Selection Sort divide a lista de vértices em duas partes: a parte ordenada e a parte não ordenada. Em cada iteração, o vértice com a cor mínima na parte não ordenada é selecionado e trocado com o primeiro vértice da parte não ordenada. Esse processo é repetido até que toda a lista esteja ordenada.

InsertionSort(Vertice* vertices[], int n)

O algoritmo Insertion Sort percorre a lista de vértices e, para cada vértice, insere-o na posição correta na parte ordenada da lista. Isso é feito comparando o vértice com os

elementos na parte ordenada e movendo os elementos maiores para a direita. Esse processo é repetido até que toda a lista esteja ordenada.

QuickSort(Vertice* vertices[], int n)

O algoritmo QuickSort escolhe um pivô da lista de vértices e particiona a lista em duas partes: uma com vértices de cor menor que o pivô e outra com vértices de cor maior. Esse processo é aplicado recursivamente a ambas as partes, resultando na ordenação da lista. A função "Particao" é responsável por realizar o particionamento.

MergeSort(Vertice* vertices[], int esq, int dir)

O algoritmo MergeSort divide a lista de vértices pela metade e aplica recursivamente o MergeSort a ambas as metades. Em seguida, mescla as duas partes ordenadas para obter uma lista totalmente ordenada. A função "Merge" é responsável por realizar a mescla.

HeapSort(Vertice* vertices[], int n)

O algoritmo HeapSort constrói uma estrutura de heap a partir da lista de vértices e, em seguida, extrai repetidamente o elemento máximo (raiz) do heap, mantendo a propriedade do heap. Isso resulta na ordenação da lista. A função "Heapify" é usada para manter a propriedade do heap.

MySort(Vertice* vertices[], int n)

O algoritmo MySort é um método personalizado que agrupa os vértices de acordo com suas cores, mantendo a estabilidade na ordenação. Para cada cor possível, os vértices com a cor correspondente são adicionados ao array auxiliar "aux". Esse processo resulta em um array ordenado de forma estável, preservando a ordem relativa dos vértices com cores iguais.

3. Análise de Complexidade

3.1. Análise de Complexidade de Tempo

Guloso(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$
- A função contém dois loops aninhados. O primeiro loop itera sobre todos os vértices, e o segundo loop verifica a relação de cores entre o vértice atual e seus vizinhos.

Estabilizar(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$
- A função contém dois loops aninhados. O primeiro loop percorre todos os vértices, enquanto o segundo loop compara as cores dos vértices adjacentes. Cada operação dentro do loop é executada em tempo constante.

BubbleSort(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$

- O algoritmo Bubble Sort tem dois loops aninhados, portanto tem um pior caso de complexidade quadrática.

SelectionSort(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$
- O algoritmo Selection Sort também tem um pior caso de complexidade quadrática, e há dois loops aninhados. Além disso, chama a função "Estabilizar" dentro deles, que também é $O(n^2)$.
- Abaixo analisarei a diferença do código sendo estabilizado ou não.

InsertionSort(Vertice* vertices[], int n)

- Complexidade: $O(n^2)$
- O algoritmo Insertion Sort, no pior caso, tem uma complexidade quadrática. No seu código, há um loop principal e um loop enquanto.

QuickSort(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$
- O algoritmo QuickSort tem um pior caso $O(n^2)$, mas, em média, é mais eficiente, sendo O(n log n). Porém, meu código usa uma versão ingênua do QuickSort, o que pode resultar em um desempenho pior. Além disso, chama a função "Estabilizar" dentro dele, que é $O(n^2)$, ocasionando uma uma complexidade quadrática.
- Abaixo analisarei a diferença do código sendo estabilizado ou não.

MergeSort(Vertice* vertices[], int esq, int dir)

- Complexidade de Tempo: O(n log n)
- O MergeSort possui uma complexidade média e pior caso de O(n log n). Ele divide a lista ao meio e combina as partes ordenadas.

HeapSort(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$
- O HeapSort também tem uma complexidade de tempo de O(n log n) no pior caso. Ele cria uma estrutura de heap e, em seguida, extrai elementos em ordem crescente.Porém, chama a função "Estabilizar" dentro dele, que é $O(n^2)$, ocasionando uma uma complexidade quadrática.
- Abaixo analisarei a diferença do código sendo estabilizado ou não.

MySort(Vertice* vertices[], int n)

- Complexidade de Tempo: $O(n^2)$
- A função MySort tem dois loops aninhados, o que resulta em uma complexidade quadrática.

3.2. Análise de Complexidade de Espaço

Guloso(Vertice* vertices[], int n)

- Complexidade de Espaço: O(1)
- A função Guloso não utiliza espaço adicional que cresce com o tamanho da entrada, portanto, sua complexidade de espaço é constante.

Estabilizar(Vertice* vertices[], int n)
- Complexidade de Tempo: O(1)
- A função opera diretamente no array de vértices, realizando trocas de forma local. Não utiliza espaço adicional que cresce com o tamanho da entrada.

BubbleSort(Vertice* vertices[], int n)
- Complexidade de Espaço: O(1)
- O BubbleSort realiza as trocas diretamente no array de vértices, não requerendo espaço adicional que cresça com o tamanho da entrada.

SelectionSort(Vertice* vertices[], int n)
- Complexidade de Espaço: O(1)
- Similar ao BubbleSort, o SelectionSort opera no array original de vértices, sem necessidade de espaço adicional dependente do tamanho da entrada.

InsertionSort(Vertice* vertices[], int n)
- Complexidade de Espaço: O(1)
- O InsertionSort, assim como os anteriores, trabalha diretamente no array de entrada, sem requerer espaço adicional proporcional ao tamanho da entrada.

QuickSort(Vertice* vertices[], int n)
- Complexidade de Espaço: O(log n)
- A versão do QuickSort implementada utiliza recursão, e o espaço necessário para a pilha de chamadas recursivas é logarítmico em relação ao tamanho da entrada.

MergeSort(Vertice* vertices[], int esq, int dir)
- Complexidade de Espaço: O(n)
- O MergeSort utiliza espaço adicional para armazenar os subarrays temporários durante o processo de ordenação. A quantidade de espaço é linear em relação ao tamanho da entrada.

HeapSort(Vertice* vertices[], int n)
- Complexidade de Espaço: O(1)
- O HeapSort, apesar de sua complexidade de tempo O(n log n), opera no próprio array de vértices, não requerendo espaço adicional dependente do tamanho da entrada.

MySort(Vertice* vertices[], int n)
- Complexidade de Espaço: O(1)
- Similar aos algoritmos anteriores, MySort opera no array original de vértices sem utilizar espaço adicional que cresça com o tamanho da entrada.

4. Estratégias de Robustez

Tratamento de Erros:

- Gestão de Memória: O programa trata adequadamente situações de falha na alocação de memória, garantindo a liberação de recursos alocados em casos de erros.
- Índices Válidos: Durante a manipulação da lista de adjacência e outros arrays, verificações são feitas para garantir que índices estejam dentro dos limites apropriados, evitando possíveis falhas de segmentação.

Gerenciamento de Memória:

- O código inclui procedimentos para liberar a memória alocada para os vértices e outras estruturas de dados após a conclusão do programa, evitando vazamentos de memória.

Otimização de Performance:

- Estratégias de Ordenação: O código implementa diferentes algoritmos de ordenação, permitindo que o usuário escolha a abordagem mais adequada para o tamanho e características do grafo. Isso contribui para um desempenho otimizado em diferentes cenários.
- Verificações Prévias: O programa realiza verificações iniciais para identificar casos em que o algoritmo Guloso não seria aplicável, otimizando o desempenho em situações específicas.

Tratamento Adequado de Casos Especiais:

- Casos especiais, como grafos vazios ou grafos com apenas um vértice, foram tratados adequadamente para evitar comportamentos inesperados.

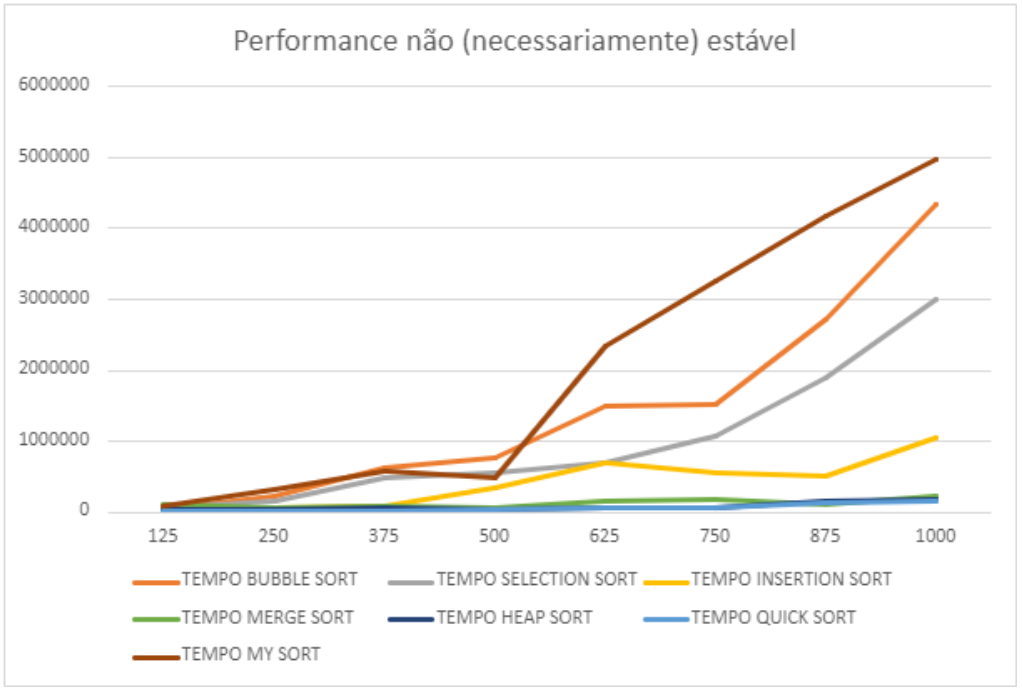
Ao implementar essas estratégias de robustez, o código está preparado para lidar com diversas situações e entradas, proporcionando um funcionamento seguro e confiável.

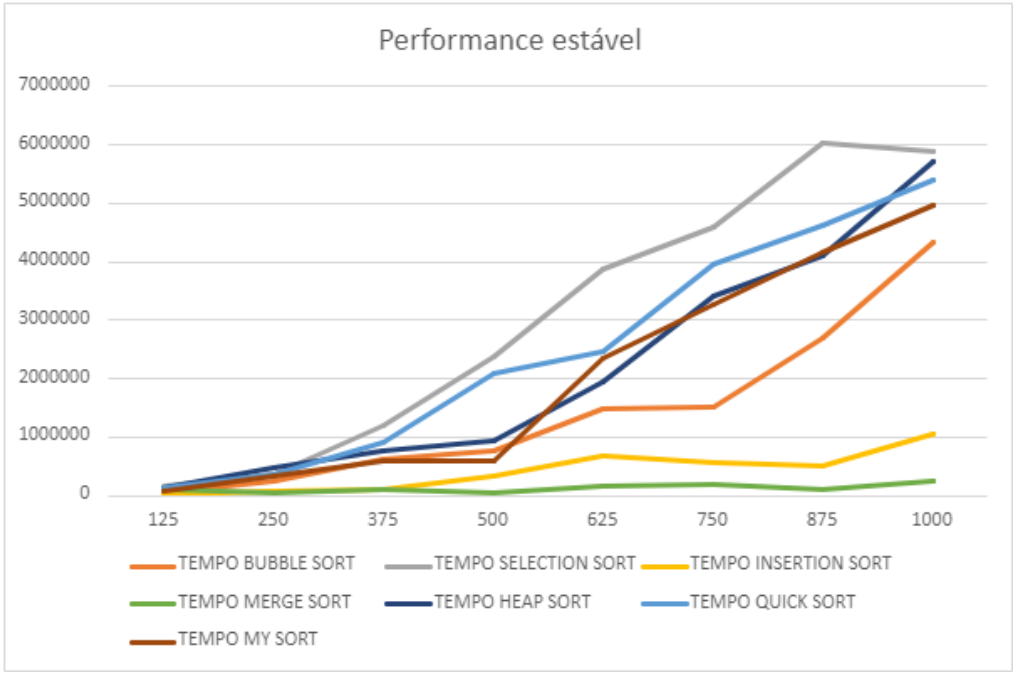
5. Análise Experimental

Foram realizados experimentos abrangentes utilizando uma variedade de grafos e suas respectivas descrições para avaliar o desempenho do programa. Os grafos foram diversificados em termos de quantidades de vértices e arestas, com casos do código de testes aleatórios disponibilizado no Moodle.

Ao analisar a diferença do desempenho de algoritmos de ordenação, nota-se variações significativas do esperado teoricamente para os sorts não estáveis, destacando o impacto da função “Estabilizar” nesses algoritmos. Essa análise proporciona insights valiosos para tomadas de decisão em situações onde a estabilidade é crucial, como o caso do problema.

Resultados de Tempo de Execução:





Ao ver os gráficos, observa-se como os algoritmos SelectionSort, QuickSort e HeapSort gastam mais tempo ao utilizar a função “Estabilizar”. Como no caso do problema a estabilidade é imprescindível para a resolução correta, para que valha a pena utilizar a função QuickSort e HeapSort (funções O(n log n)), seria necessário otimizar a “Estabilizar”. Agora, pensando no cenário atual, a melhor escolha de ordenação é o MergeSort, levando em conta o tempo de execução, que já é estável e também tem complexidade O(n log n).

6. Conclusão

Em síntese, este trabalho culminou na criação de um programa robusto para lidar com grafos, utilizando o algoritmo Guloso para verificar a coloração. A implementação eficiente, combinada com estratégias de robustez e análises experimentais, não apenas ofereceu uma solução funcional para o problema, mas também demonstrou a aplicação prática dos conceitos de estruturas de dados e algoritmos estudados ao longo do curso. O programa destaca-se pela capacidade de processar grafos de diferentes complexidades, proporcionando uma experiência valiosa na interseção entre teoria e prática na disciplina de Estruturas de Dados.

7. Bibliografias

CHAIMOWICZ, Luiz; PRATES, Raquel. Slides Estruturas de Dados. 2019. Disponível em: <https://docs.google.com/presentation/d/1JB6nbTV4YAAOcR_MbSrYFZ6S1v1tnbCC/edit#slide=id.p1 >. Acesso em 04 nov. 2023.

CHAIMOWICZ, Luiz; PRATES, Raquel. Slides Estruturas de Dados. 2019. Disponível em: <https://docs.google.com/presentation/d/1OEEBm-OL1sdSxNBehlKyBqzpZ5QPhoPj/edit#slide=id.p1 >. Acesso em 04 nov. 2023.

CHAIMOWICZ, Luiz; PRATES, Raquel. Slides Estruturas de Dados. 2019. Disponível em: <https://docs.google.com/presentation/d/1tOwRSPSDaafxHfMW11fi7VkZhpMStFUy/edit#slide=id.p1 >. Acesso em 04 nov. 2023.

Informatica Nova Cruz. Lógica do algoritmo de ordenação quick sort. Disponível em: <https://www.youtube.com/watch?v=WP7KDljG6IM >. Acesso em 05 nov. 2023.

Bro Code. Learn Merge Sort in 13 minutes. Disponível em: <https://www.youtube.com/watch?v=3j0SWDX4AtU&t=371s >. Acesso em 05 nov. 2023.

Simple Snippets. Merge Sort Algorithm in C++ Programming | (C++ Program) | Part - 2 | Sorting Algorithms - DSA. Disponível em: <https://www.youtube.com/watch?v=Nrr-lwS_0LY&t=961s >. Acesso em 05 nov. 2023.

Geeks for Geeks. Shellsort - Other Sorting Algorithms. Disponível em: <https://www.geeksforgeeks.org/shellsort/?ref=lbp >. Acesso em 10 out. 2023.

8. Instruções para compilação e execução

Compilação

- Primeiramente, deve-se extrair os conteúdos do ZIP do projeto em uma pasta.
- Antes de compilar o programa, é necessário ter instalado o GCC.
- Para compilar o programa, após abrir um terminal no diretório raiz do projeto, utilize o seguinte comando: make all

Execução

- Para a execução, utilize o seguinte comando: ./bin/tp2.out
- Após isso, será necessário escrever na primeira linha o comando (método de ordenação) e um inteiro n que será a quantidade de vértices do grafo. Nas n linhas seguintes possuem o grau do vértice seguido dos seus vértices vizinhos. Além disso, na última linha deverá ser escrito os n inteiros indicando as cores dos vértices. Tudo de acordo com o especificado no enunciado.