UNIVERSIDADE FEDERAL DE MINAS GERAIS

Lucas Rafael Costa Santos

2021017723

TRABALHO PRÁTICO 2

Essa Coloração é Gulosa?

Belo Horizonte - MG, 12 de novembro de 2023

1. Introdução

Este trabalho tem como objetivo testar os conhecimentos do aluno quanto ao desenvolvimento e aplicação de estruturas de dados. Para isso foi proposto um problema que lida com a ordenação e coloração de grafos. A questão em foco envolve a análise de um grafo dado com 'n' vértices, seus vizinhos correspondentes e a atribuição de cores. A tarefa consiste em organizar os vértices de acordo com suas cores e, posteriormente, verificar se a coloração é gulosa.

2. Método

2.1 Estrutura de dados

A principal Estrutura de Dados presente neste trabalho foi o grafo. Um grafo G é um par (V,A), em que V é um conjunto de objetos distintos e A é um conjunto de pares de elementos de V. Os elementos de V são chamados de vértices, enquanto os elementos de A são chamados de arestas.

Visualmente, podemos imaginar um grafo simplesmente como um conjunto de pontos e linhas conectando pares desses pontos. Neste caso, os pontos representam os vértices e as linhas representam as arestas conectando pares de vértices.

Além do grafo, também foi utilizada a Lista de Adjacência, que serviu como base para a construção do nosso grafo, uma Lista Encadeada, para guardar as cores na chamada da função “EhGuloso”, e os algoritmos de ordenação, sendo eles: *Bubble Sort, Selection Sort, Insertion Sort, Quicksort, Mergesort, Heapsort* e um método de ordenação próprio.

2.2 Classes

Foi elaborada a classe "Grafo" para a implementação de uma estrutura de grafo utilizando lista de adjacência. Essa classe fornece possui um construtor, e métodos para inserir vértices e arestas, inserir cores nos vértices, obter a quantidade de vértices e arestas e verificar se é possível colorir o grafo de maneira gulosa.

A classe "Grafo" é composta por uma instância da classe "ListaAdjacencia", que gerencia a representação do grafo por meio de uma lista de adjacência. A classe "ListaAdjacencia" oferece métodos para inserir vértices e arestas, obter a quantidade de vértices e arestas, bem como acesso aos vértices do grafo.

Cada vértice na lista de adjacência é representado por uma estrutura que contém um valor, uma cor (inicializada como -1), um ponteiro para o próximo vértice e métodos específicos para sua manipulação.

Além disso, a classe "Grafo" possui um array de cores para auxiliar no processo de coloração. Para verificar se é possível colorir o grafo de maneira gulosa, foi implementado o

método "EhGuloso", que recebe um array de itens do tipo "TipoItem" e o número de cores desejado.

Também foram implementadas as classes ListaCores", para gerenciar a lista de cores utilizada na checagem da coloração dos vértices e "TipoItem", para representar um item do grafo.

2.2 Funções / Métodos

A principal função do nosso código é a função “EhGuloso”, que serve para determinar se nosso grafo possui ou não uma coloração gulosa, levando em consideração as condições relacionadas às cores dos vizinhos de cada vértice. Se as condições para todos os vértices forem atendidas, a coloração é considerada gulosa.

Sabe-se que, a coloração gulosa é uma forma de colorir os vértices de um grafo de modo que nenhum par de vértices adjacentes tenha a mesma cor, utilizando o menor número possível de cores. Para analisar isso, a função recebe dois parâmetros: um array de objetos do tipo TipoItem representando os vértices do grafo e o número total de cores disponíveis (numCores).

Entre as principais etapas e lógica presentes, temos:

- Obtenção de Informações do Grafo:

  Obtém o número total de vértices no grafo usando QuantidadeVertices().

  Obtém a lista de adjacência dos vértices usando vertices.ObterVertices().

- Mapeamento dos Índices Originais:

  Cria um array de mapeamento para associar os índices originais dos vértices com os índices após a ordenação. Isso é necessário para trabalhar com os vértices na ordem correta após a ordenação.

- Iteração pelos Vértices:

  Itera sobre cada vértice no grafo.

  Para cada vértice, inicializa variáveis para rastrear informações sobre os vizinhos desse vértice.

- Verificação das Cores dos Vizinhos:

  Percorre a lista de adjacência do vértice atual para examinar seus vizinhos.

  Conta o número de vizinhos, a maior cor entre os vizinhos, o número de vizinhos com cor menor que a cor atual e a maior cor entre esses vizinhos menores.

- Verificação das Condições de Coloração Gulosa:

  Com base nas informações dos vizinhos, verifica se a coloração atual é uma coloração gulosa.

As condições verificam se a cor atual não ultrapassa o número total de vértices, se é maior que o número de vizinhos, se é maior que a maior cor entre os vizinhos, se é maior que o número de vizinhos com cor menor que a cor atual e se é maior que a maior cor entre esses vizinhos menores.

- Liberação de Memória:

Libera a memória alocada para o array de mapeamento.

- Resultado:

Se todas as verificações passarem para todos os vértices, a função retorna true, indicando que a coloração é uma coloração gulosa. Caso contrário, retorna false.

2.3 Algoritmos de Ordenação

Bubble Sort:

O algoritmo de ordenação *Bubble Sort* percorre a lista múltiplas vezes, compara elementos adjacentes e troca-os se estiverem na ordem errada. Esse processo é repetido até que a lista esteja ordenada. O pior caso ocorre quando a lista está inversamente ordenada, sendo assim, resulta em uma complexidade de tempo quadrática O(n^2). Apesar de ser simples, o *Bubble Sort* não é tão eficiente para grandes conjuntos de dados.

Selection Sort:

O *Selection Sort* divide a lista em duas partes: a parte ordenada e a parte não ordenada. A cada iteração, o algoritmo encontra o menor elemento da parte não ordenada e o troca com o primeiro elemento não ordenado. Esse processo é repetido até que toda a lista esteja ordenada. O *Selection Sort* também tem uma complexidade de tempo quadrática O(n^2) no pior caso.

Insertion Sort:

O *Insertion Sort*, por sua vez, percorre a lista, comparando cada elemento com os elementos anteriores e os move para a posição correta na parte ordenada da lista. É eficiente para listas pequenas, com uma complexidade de tempo O(n^2) no pior caso. No entanto, é menos eficiente para conjuntos de dados maiores em comparação com algoritmos mais avançados.

Quick Sort:

Já *Quick Sort* é famoso por utilizar a estratégia de divisão e conquista. Assim, ele escolhe um elemento como pivô e particiona a lista em duas partes, onde os elementos menores que o pivô ficam à esquerda e os maiores à direita. Esse processo é aplicado recursivamente nas sub-

listas. O *Quick Sort* tem uma boa performance na prática e uma complexidade média de tempo O(n log n), mas pode acabar sendo O(n^2) no pior caso.

Merge Sort:

O *Merge Sort* divide a lista em duas metades, ordena cada metade e, em seguida, combina as duas metades ordenadas. Ele utiliza a estratégia de divisão e conquista, garantindo uma complexidade de tempo O(n log n) em todos os casos. O *Merge Sort* é eficiente para grandes conjuntos de dados, mas consome mais memória devido à necessidade de armazenar temporariamente a lista dividida.

Heap Sort:

O *Heap Sort* transforma a lista em uma árvore binária de heap, em que o elemento máximo (ou mínimo, dependendo da ordem desejada) é removido e inserido na parte ordenada da lista. Esse processo é repetido até que a lista esteja ordenada. O *Heap Sort* tem uma complexidade de tempo O(n log n) e é eficiente em termos de espaço.

My Sort:

Por fim, temos o algoritmo *My Sort*. Aqui, implementamos uma variação do algoritmo *Shell Sort*, que, por sua vez, é uma melhoria do *Insertion Sort*. Ele basicamente utiliza um intervalo variável entre os elementos a serem comparados e trocados, o que o torna mais eficiente que *o Insertion Sort* padrão. A ideia-chave por trás deste algoritmo é utilizar uma sequência específica de intervalos para a comparação e troca de elementos. Neste caso, o algoritmo utiliza a sequência de intervalos proposta por Donald Shell, que é dada pela fórmula h = h * 3 + 1. O algoritmo começa com um grande intervalo h e o reduz progressivamente até que h seja igual a 1.

O *My Sort* tem uma complexidade de tempo que pode variar dependendo da sequência utilizada para determinar os intervalos, mas geralmente é mais eficiente que o *Insertion Sort* .

3. Análise de Complexidade

A análise de complexidade do código pode ser dividida em várias partes, considerando as operações básicas dentro de cada bloco de código. Podemos analisar em partições da seguinte forma:

- Leitura do Método de Ordenação e Número de Vértices:

Complexidade de Tempo: O(1) - Operações de leitura constantes.

Complexidade de Espaço: O(1) - Armazenamento de duas variáveis.

- Criação do Grafo:

    Complexidade de Tempo: O(n) - Loop que insere n vértices.
    Complexidade de Espaço: O(1) - Armazenamento de uma estrutura de grafo.

- Leitura dos Vizinhos dos Vértices e Inserção de Arestas:

    Complexidade de Tempo: O(n + m), onde m é o número total de arestas.
    Complexidade de Espaço: O(m) - Armazenamento dos vizinhos.

- Leitura das Cores dos Vértices:

    Complexidade de Tempo: O(n) - Loop que lê as cores.
    Complexidade de Espaço: O(n) - Armazenamento do vetor de cores.

- Escolha do Método de Ordenação e Aplicação no Vetor de Cores:

    A complexidade dos métodos de ordenação já foi discutida no tópico anterior, mas é importante destacar que ela varia devido ao método escolhido e ao tamanho da entrada. Havendo situações em que no pior caso a complexidade é O(n^2). Ou casos que têm complexidade O(n log n).

- Inserção de Cores no Grafo:

    Complexidade de Tempo: O(n) - Loop que insere cores no grafo.
    Complexidade de Espaço: O(1) - Operações constantes de armazenamento.

- Verificação se a Coloração é uma Coloração Gulosa:

    Complexidade de Tempo: O(n^2) - Dois loops aninhados.
    Complexidade de Espaço: O(n) - Armazenamento temporário para o mapeamento.

- Impressão do Resultado:

    Complexidade de Tempo: O(n) - Loop de impressão.
    Complexidade de Espaço: O(1) - Operações constantes de armazenamento.

- Desalocação de Memória:

    Complexidade de Tempo: O(1) - Operação de desalocação.
    Complexidade de Espaço: O(1) - Operações constantes de desalocação.

A complexidade geral do programa será dominada pela verificação se a coloração é gulosa, pois ela terá complexidade O(n^2), enquanto os algoritmos de ordenação terão essa complexidade somente no pior caso. Sendo assim, se o vetor de cores for grande, a escolha do algoritmo de ordenação também pode ter um impacto significativo na eficiência do programa.

4. Estratégias de Robustez

A fim de assegurar a robustez do programa, foram implementadas algumas estratégias no código. Primeiramente, há a verificação da entrada, na qual o código verifica se a leitura do método de ordenação e do número de vértices foi bem-sucedida. Se a leitura falhar ou se o número de vértices for não positivo, uma mensagem de erro é exibida e o programa encerra com um código de retorno diferente de zero. Essa abordagem também é aplicada ao número de vizinhos e às cores dos vértices.

Outro ponto de destaque é o manuseio de memória dinâmica. Como nosso código utiliza alocação dinâmica de memória para o vetor de cores e os vizinhos dos vértices. É garantido que a memória alocada é liberada corretamente usando *delete[]* quando não é mais necessária.

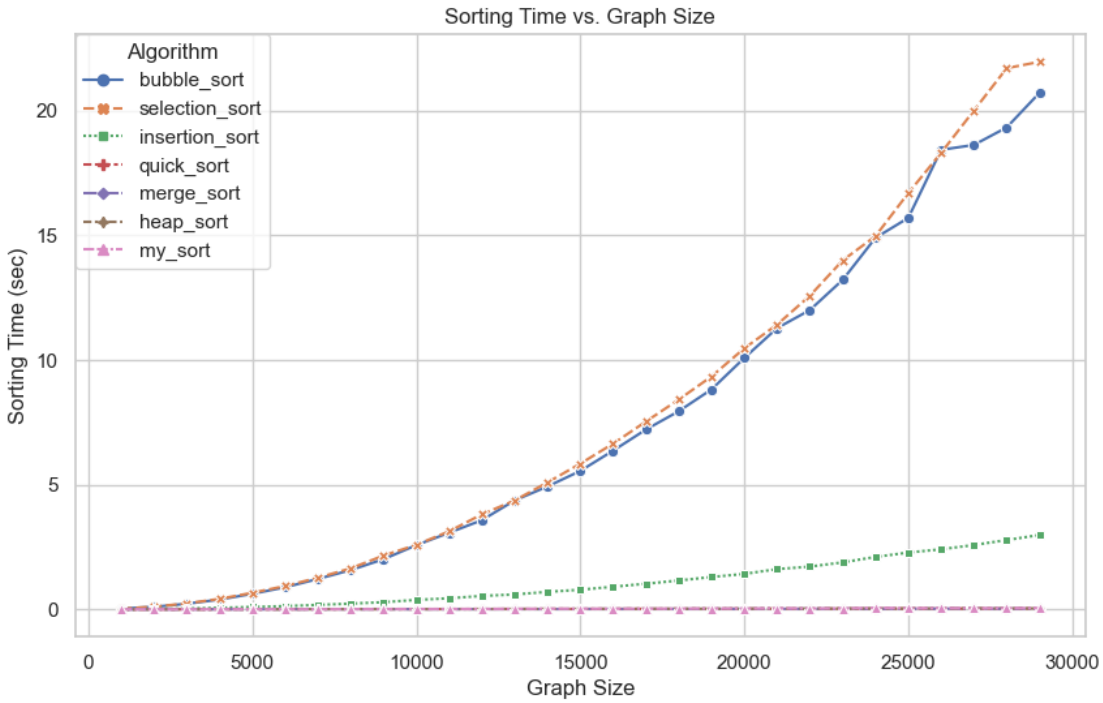
A escolha do método de ordenação é realizada de maneira eficiente por meio de uma estrutura switch-case, baseada no caractere lido. Caso o caractere não corresponda a nenhum caso, o programa exibe uma mensagem de erro e encerra com um código de retorno diferente de zero.

Durante a leitura dos vizinhos dos vértices, o código verifica a validade do vértice (dentro do intervalo [0, n)), exibindo uma mensagem de erro e encerrando o programa se um vértice inválido for encontrado. De maneira análoga, ao ler as cores dos vértices, o código verifica se a cor é válida (não negativa) e, em caso contrário, exibe uma mensagem de erro e encerra o programa.

Por fim, o código assegura uma desalocação adequada da memória alocada dinamicamente antes de encerrar o programa, utilizando *delete[]* para liberar a memória associada ao vetor de cores. Além disso, a utilização de *std::cerr* para mensagens de erro é uma prática sólida, pois garante que tais mensagens sejam direcionadas ao fluxo de erro padrão, evitando interferências na saída padrão do programa.

5. Análise Experimental

Para analisar o custo computacional dos diferentes algoritmos de ordenação foram criados 29 grafos, os quais variam seu número de vértices entre 1000 e 30000, com a quantidade de arestas estabelecida como 50% superior ao número de vértices. A execução de cada algoritmo de ordenação foi medida em relação a esses grafos, utilizando a biblioteca *chrono*. Os resultados temporais foram então representados graficamente, como mostrado na Figura a seguir.



Ao observar a imagem, é possível perceber que os algoritmos *bubble_sort* e *selection_sort* possuem um crescimento relativamente maior que os demais. Essa tendência pode ser explicada pela complexidade de ambos, que é O(n^2), o que, de fato é bem representado no gráfico.

Na sequência, temos o *insertion_sort*, que se destaca por ser mais eficiente do que os algoritmos mencionados anteriormente, sendo então, uma opção razoável para conjuntos de dados de tamanho moderado graças à sua complexidade de O(n^2) no pior cenário.

Por fim, os algoritmos *merge_sort*, *heap_sort* e *quick_sort* demonstram um desempenho superior aos demais. Essa superioridade decorre do fato de esses algoritmos possuírem complexidade O(n log n), ou, no caso do *quick_sort*, O(n^2) no pior cenário. Vale ressaltar que o algoritmo *my_sort*, inspirado numa melhoria do *insertion_sort*, exibe um desempenho notável, chegando a resultados comparáveis aos algoritmos mais eficientes mencionados.

Após essa análise, concluímos que, para valores inferiores a 5000, o método de ordenação não impacta significativamente no desempenho. No entanto, a partir desse ponto, deve-se evitar o uso do *bubble_sort* e *selection_sort*. Já o *insertion_sort* pode ser empregado até aproximadamente 12000, sendo então recomendável a transição para os demais algoritmos, que oferecem resultados mais satisfatórios.

6. Conclusão

Neste trabalho, exploramos a problemática da coloração gulosa em grafos, e a aplicação de diversos algoritmos de ordenação. Ao enfrentar desafios relacionados à coloração de grafos, aprofundamos nosso entendimento sobre a eficácia da abordagem gulosa e a influência dos diferentes algoritmos de ordenação na resolução desses problemas específicos.

Este projeto proporcionou-nos uma compreensão mais ampla das nuances práticas envolvidas na coloração de grafos e na seleção criteriosa de algoritmos de ordenação. Além disso, demonstrou a importância da análise de complexidade do código, destacando como entradas mais extensas podem impactar o desempenho de um programa. Também aprendemos como a análise experimental pode ser uma ferramenta diferencial para verificar a consistência do código e identificar possíveis problemas de alocação de memória.

7. Compilação e execução

Para compilar o programa, basta navegar até o diretório raiz do projeto por meio de um terminal e utilizar o comando:

Make ou make all

Para executá-lo acesse o executável gerado com o comando

./bin/tp2.out

Para remover os arquivos compilados e executáveis, utilize o seguinte comando:

make clean

8. Bibliografia

NOIC - Material de Matemática. Disponível em: https://noic.com.br/materiais-matematica/cursos-de-matematica-noic/curso-de-combinatoria/grafos-definicoes/. Acesso em: 11 de novembro de 2023.

Aulas - Estruturas de Dados para Grafos. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphdatastructs.html. Acesso em: 11 de novembro de 2023.

FEOFILOFF, Paulo. Algoritmos de Ordenação. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html. Acesso em: 11 de novembro de 2023.

Slides da Disciplina.