Trabalho Prático 2 - Coloração de Grafos

João Correia Costa (2019029027)

Novembro de 2023, Belo Horizonte

1 Introdução

Uma variedade de problemas na computação e na ciência podem ser modelados através de entidades e relações. Tal modelagem é comumente representada por meio de grafos coloridos que codificam as propriedades intrínsecas do sistema.

Nesse contexto, o presente trabalho busca validar a coloração de grafos, identificando se ela foi gerada por meio de um algoritmo guloso, ou seja, se um vértice v possui coloração i, então ele possui pelo menos um vizinho com cada uma das cores menores que i.

Tecnicamente, seja $G = (V, E)$ um grafo k-colorível e c uma k-coloração própria de G, a coloração c será gulosa se: $\forall v \in V(G)$, se $c(v) = i$ onde $0 < i \leq k$ com $0 < j < i$, existe um vértice u vizinho de v com $c(u) = j$.

Para solucionar o problema, os grafos foram modelados através de um array, como lista de adjacência, combinada com listas simplesmente encadeadas. A escolha de um array de tamanho fixo foi possível, pois sabemos de antemão o tamanho N do grafo a ser validado. As listas encadeadas armazenam o índice dos vizinhos associados ao vértice de índice correspondente a sua posição em array. Essa implementação será descrita em mais detalhes na seção seguinte. A priori, para validar o grafo, percorremos a estrutura array+lista-encadeada observando se a coloração gulosa é satisfeita.

A segunda etapa do problema consiste em ordenar os vértices por cor, em primeiro, e por índice em segundo. Aplicamos os algoritmos de ordenação clássicos: `bubble sort`, `selection sort`, `insertion sort`, `quick sort`, `merge sort`, `heap sort` e `count sort`, considerando o critério de ordenação citado. O objetivo consiste em analisar a performance desses algoritmos na ordenação dos vértices. Para isso, foram gerados grafos de tamanhos variados, medido o tempo de ordenação para cada algoritmo para construção de gráficos.

2 Método

2.1 Estrutura de Dados

A entrada de dados consiste em um inteiro N, que representa o tamanho do grafo. Subsequentemente, linha por linha, são fornecidos outros inteiros indicando o número de vizinhos para o vértice cujo índice corresponde ao número da linha. Por exemplo, a primeira linha descreve os vizinhos do vértice 0, a segunda linha descreve os vizinhos do vértice 1, e assim por diante. Após as N linhas que descrevem os vizinhos, há uma linha adicional que representa a coloração de cada vértice.

Para armazenar essas informações de maneira eficiente, é apropriado utilizar um array de tamanho N que contém ponteiros para listas encadeadas. Em outras palavras, na posição 0 em array, encontramos um ponteiro para uma instância da classe `Vertex` que descreve o vértice 0. Cada vértice é modelado por meio da classe `Vertex`, que é uma classe derivada da `LinkedList`. Além dos ponteiros comuns em listas encadeadas, que representam os índices dos vizinhos, a classe `Vertex` também armazena informações como a cor do vértice e o número de vizinhos. A representação gráfica dessa estrutura é exemplificada na figura abaixo.

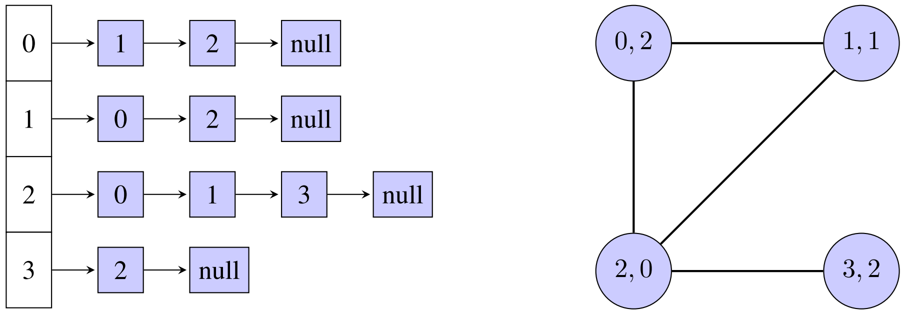

Figure 1: *Array de Lista Encadeadas e Grafo(índice, cor)*

Alternativamente, poderia ter sido empregada uma lista encadeada dupla; no entanto, a escolha da estrutura de dados composta por uma array e uma lista encadeada oferece um desempenho superior neste contexto. Em primeiro lugar, a array representa um bloco contíguo de memória, resultando em um acesso mais eficiente a um elemento em qualquer índice. Embora se possa argumentar que a array possui uma estrutura mais rígida em comparação com a lista encadeada, essa rigidez não se configura como uma desvantagem significativa no contexto específico dos grafos fornecidos, pois esses têm tamanho fixo.

Em segundo lugar, a combinação de uma array e uma lista encadeada proporciona flexibilidade em termos de tamanho, pois lista encadeada permite a alocação dinâmica de memória, adaptando-se de maneira eficaz ao número variável de vizinhos para cada vértice.

Assim, a escolha dessa estrutura de dados composta demonstra ser uma abordagem eficaz, otimizando o acesso e alocando dinamicamente recursos conforme necessário para representar o grafo.

2.2 Validação

A validação da coloração do grafo consiste em verificar se cada vértice v de cor i possui pelo menos um vizinho com cada uma das cores menores que i. Cada vértice, representado pela classe Vertex<int>, é dotado de um método denominado bool validate. Essencialmente, esse método percorre a lista encadeada contendo os índices dos vizinhos de v, busca a cor de cada vizinho na array de vértices Vertex<int>** graph e verifica se o conjunto de cores observado preenche todo o intervalo $[1, i)$.

A função auxiliar bool validate_graph, presente no arquivo utils.hpp, realiza uma iteração sobre todos os vértices na array, invocando o método validate para cada vértice individual. O resultado final é conjunção de todos os valores retornados pelos vértices, indicando uma coloração válida caso a conjução seja verdadeira (retorna true).

2.3 Ordenação

Após a validação do grafo, procede-se à ordenação dos vértices primeiro por cor e, em seguida, por índice. A implementação dessas operações é clássica, uma vez que envolve a ordenação de valores inteiros que representam cores e índices. Para fins práticos, a troca de posição dos vértices é realizada através da manipulação dos ponteiros para os vértices. Em outras palavras, se o vértice 5 deve ser colocado antes do vértice 0, os endereços de memória desses vértices são trocados entre si.

A função void swap_vertex é responsável por efetuar essa troca de endereços. Nesse contexto, a aplicação dos métodos de ordenação assemelha-se à ordenação de um vetor de inteiros. No entanto, em vez de acessar valores diretamente, utilizam-se getters para obter as cores e índices dos vértices, e a troca de posições é realizada manipulando os endereços de memória dos vértices.

Os métodos de ordenação aplicados são: bubble sort, selection sort, insertion sort, quick sort, merge sort, heap sort e count sort. A descrição breve desses algoritmos consta na seção de análise de complexidade.

3 Análise de Complexidade

Essa seção se dedica a analisar a complexidade assintótica, em termos de espaço e de tempo, das principais funções envolvidas em validar a coloroção e ordenar o grafo.

3.1 bool validate_graph

- O método itera sobre todos os n vértices do grafo (complexidade $O(n)$).
- Chama o método validate para cada vértice, o qual itera sobre os vizinhos (complexidade média $O(\text{grau do vértice})$).
- Portanto, a complexidade total é $O(n \cdot \text{grau médio})$.

3.2 bool Vertex::validate

- O método itera sobre as cores até target_color $= \textit{\_color} - 1$.
- Para cada cor, itera sobre todos os vizinhos (complexidade média $O(\text{grau do vértice})$).
- Assim, a complexidade total é $O(\textit{\_color} \cdot \text{grau médio})$.

Se considerarmos grafos de densidade constante, isto é, o grau médio é fixo, a complexidade final do algoritmo é dominada por validate_graph, resultando em uma complexidade média de $O(n)$. Observa-se que como o grau do vértice é fixo, o número de cores é limitada, logo Vertex::validate performa um número pequeno e fixo de iterações, complexidade $O(1)$.

3.3 `void bubble_sort(Vertex<int>**& graph, int n_vertex)`

- O método itera sobre todos os n_{vertex} vértices do grafo (complexidade $O(n_{\text{vertex}})$).
- Para cada vértice, o loop interno itera sobre $n_{\text{vertex}} - i - 1$ outros vértices.
- A complexidade total é $O(n_{\text{vertex}}^2)$.

O Bubble Sort tem custo de memória $O(1)$, pois opera in-place.

3.4 `void insertion_sort(Vertex<int>**& graph, int n_vertex)`

- O método itera sobre todos os n_{vertex} vértices do grafo (complexidade $O(n_{\text{vertex}})$).
- Para cada vértice, o loop interno executa um número variável de iterações, dependendo da posição correta do vértice na sequência ordenada até o momento.
- No pior caso, o número total de iterações do loop interno é $O(i)$, onde i é o índice do vértice no array.
- A complexidade total do algoritmo é, portanto, $O(n_{\text{vertex}}^2)$ no pior caso.

O Insertion Sort tem custo de memória $O(1)$, pois opera in-place.

3.5 `void selection_sort(Vertex<int>**& graph, int n_vertex)`

- O método itera sobre todos os $n_{\text{vertex}} - 1$ elementos do grafo (complexidade $O(n_{\text{vertex}})$).
- Para cada elemento na iteração externa, o loop interno encontra o índice do menor elemento na parte não ordenada do array (complexidade média $O(n_{\text{vertex}})$).
- A função `swap_vertex` é chamada uma vez por iteração externa.
- A complexidade total do algoritmo é $O(n_{\text{vertex}}^2)$.

O Selection Sort tem custo de memória $O(1)$, pois opera in-place.

3.6 `void quick_sort(Vertex<int>**& graph, int n_vertex)`

- A função `quick_sort` é uma implementação recursiva do algoritmo Quick Sort.
- A função principal `quick_sort(graph, p, r)` chama a função `partition` para encontrar a posição do pivô.
- Em seguida, chama recursivamente `quick_sort` para as partições à esquerda e à direita do pivô.
- A complexidade de tempo média do Quick Sort é $O(n \log n)$, onde n é o número de elementos a serem ordenados.
- No pior caso, a complexidade é $O(n^2)$, mas isso é raro na prática.
- A complexidade de espaço é $O(n)$ devido às chamadas recursivas, tornando-o eficiente em termos de memória.

3.7 `void merge(Vertex<int>** graph, int left, int mid, int right)`

- A função `merge` realiza a fusão de duas sub-arrays ordenadas em um único array ordenado.
- Calcula os tamanhos das sub-arrays ($n1$ e $n2$).
- Cria sub-arrays temporárias `left_arr` e `right_arr`.
- Preenche as sub-arrays temporárias com os elementos correspondentes da array original.
- Combina as sub-arrays de volta na array original em ordem ordenada.
- A complexidade temporal é $O(n \log n)$, onde n é o número total de elementos a serem ordenados.
- A complexidade espacial é $O(n)$ devido às sub-arrays temporárias.

3.8 `void heap_sort(Vertex<int>**& graph, int n_vertex)`

- A função `heap_sort` utiliza uma estrutura de heap para realizar a ordenação.
- Cria uma instância da classe `Heap` com capacidade inicial de `n_vertex`.
- Insere os vértices no heap com base em uma tupla contendo a cor e o ID do vértice.
- Cria uma array temporária `sortedGraph` para armazenar os vértices ordenados.
- Reordena a array `sortedGraph` com base na ordem de remoção do heap.
- Copia os vértices ordenados de volta para a array original `graph`.
- A complexidade temporal do Heap Sort é $O(n \log n)$, onde n é o número de elementos a serem ordenados.
- A complexidade espacial é $O(n)$ devido à array temporária `sortedGraph`.

3.9 `void count_sort(Vertex<int>**& graph, int n_vertex)`

- A função `count_sort` implementa o algoritmo de ordenação Counting Sort.
- Inicializa um array de contagem `count_arr` de tamanho n_vertex e o preenche com zeros.
- Conta a ocorrência de cada cor no grafo, incrementando os contadores em `count_arr`.
- Realiza a contagem cumulativa em `count_arr`.
- Cria um array temporário `output` para armazenar os elementos ordenados.
- Atribui os elementos de `graph` ao array `output` de acordo com a contagem cumulativa.
- Atualiza o array original `graph` com os elementos ordenados em `output`.
- Libera a memória alocada para o array temporário `output`.
- A complexidade temporal é $O(n)$, onde n é o número total de vértices no grafo.
- A complexidade espacial é $O(n)$ devido ao array de contagem `count_arr` e ao array temporário `output`.

4 Estratégias de Robustez

Com o objetivo de tornar o programa mais robusto e evitar problemas com entradas inválidas, foram criadas classes de exceção `ExceptionEmptyList`. Essa exceção é disparada, com uma mensagem de erro descritiva, caso a função de remoção de um elemento da lista seja invocada para uma estrutura vazia.

Para manter a integridade do programa e evitar vazamentos de memória, todos os Tipos Abstratos de Dados (TADs) implementam destrutores apropriados. Nos destrutores da lista encadeada, garantimos que o estado da instância retorne ao `default` e `asserts` checam se o tamanho da estrutura é zero.

Além disso, foram realizados testes com o Valgrind, e nenhum erro relacionado à alocação de memória foi observado.

Entretanto, é importante destacar que o programa ainda possui limitações, uma vez que não cobre um amplo espectro de possíveis entradas de dados, presumindo que o usuário fornecerá entradas corretas.

5 Análise Experimental

5.1 Algoritmos de Ordenação

O presente experimento foi conduzido com o propósito de avaliar o custo computacional de diferentes algoritmos de ordenação. Foram gerados 29 grafos, variando o número de vértices de 1000 a 30000. A quantidade de arestas em cada grafo foi estabelecida como 50% superior ao número de vértices. O tempo de execução de cada algoritmo de ordenação foi medido em relação a esses grafos, utilizando a biblioteca `chrono`. Os resultados temporais obtidos foram então representados graficamente, conforme ilustrado na Figura no Apêndice B.

Ao analisar o gráfico, destacam-se três grupos de desempenho. O grupo de pior desempenho inclui os algoritmos `bubble_sort` e `selection_sort`; em segundo lugar, temos o `insertion_sort`; e o terceiro grupo engloba os demais algoritmos, que apresentam um custo temporal abaixo de décimos de segundo para todos os grafos.

Presume-se que o desempenho superior do `insertion_sort` em relação ao primeiro grupo deve-se à sua adaptabilidade a dados parcialmente ordenados, ou seja, os grafos fornecidos já possuem uma ordenação razoável. Os algoritmos do primeiro grupo não se beneficiam da ordenação parcial.

Como esperado, os algoritmos `merge_sort`, `heap_sort`, `quick_sort` apresentaram desempenho muito superior aos outros grupos, especialmente para grafos grandes. Isso pode ser explicado, em primeiro lugar, pela função de complexidade desses três algoritmos, que no pior caso é $O(n \log n)$ para o `merge_sort` e `heap_sort`, e $O(n^2)$ para o `quick_sort`, embora geralmente o `quick_sort` apresente $O(n \log n)$, conforme observado no gráfico.

Além disso, o uso de estruturas de dados auxiliares, como Heaps e arrays extras, também contribuem para a eficiência desses três algoritmos.

No grupo de melhor desempenho, é destacado um terceiro algoritmo de ordenação: o Count Sort. O Count Sort é eficiente para conjuntos de dados com um intervalo pequeno de valores, como é o caso dos grafos do experimento, que têm uma faixa limitada de cores. Sua complexidade temporal é linear $O(n)$, onde n é o número de vértices, o que o torna uma escolha eficiente para o problema proposto.

5.2 Validação

Ao analisarmos o tempo de execução da função de validação da coloração dos grafos, observamos uma curva linear, o que está em conformidade com a análise de complexidade $O(n)$. A Figura correspondente pode ser encontrada no Apêndice B. Essa análise é fundamentada na compreensão da complexidade dos métodos envolvidos na validação da coloração: `bool validate_grapgh` e `bool validate`.

6 Conclusões

No projeto, abordamos o problema clássico de coloração em grafos, com ênfase na complexidade algorítmica e nas estruturas de dados implementadas em C++. Inicialmente, desenvolvemos um algoritmo para identificar se a coloração do grafo é gulosa, isto é, se um vértice v possui coloração i, então ele possui pelo menos um vizinho com cada uma das cores menores que i. Em seguida, aplicamos algoritmos clássicos de ordenação, seguindo o critério de cor e índice, conforme descrito na função `bool criterium`.

Ao longo do projeto, aprofundamos nosso entendimento das estruturas de dados eficientes para modelagem de grafos, como a combinação de array e lista encadeada. Também exploramos os algoritmos de ordenação clássicos, destacando as nuances entre suas complexidades assintóticas em termos de tempo e memória.

Este trabalho não apenas ampliou nossa habilidade de implementar soluções algorítmicas em C++, mas também proporcionou insights sobre a escolha e aplicação de estruturas de dados e algoritmos em cenários específicos, contribuindo para uma compreensão mais abrangente da ciência da computação.

7 Bibliografia

1. 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.

2. Introduction to Algorithms, Thomas H. Cormem, Charles E. Leiserson, Ronald L. Rivest.

A Instruções para Compilação e Execução

Observação: Certifique-se de que você tenha o compilador GCC (g++) instalado em seu sistema para a compilação.

A.1 Compilação do Projeto

Para compilar o projeto, siga as instruções abaixo:

1. Abra um terminal e navegue até o diretório raiz do projeto.
2. Certifique-se de que o projeto contenha a seguinte estrutura de diretórios:

   ```
   - src/
   - obj/
   - bin/
   - include/
   ```

3. Utilize o seguinte comando para compilar o projeto:

   ```
   make ou make all
   ```

   Isso irá compilar o projeto e gerar o executável `bin/tp2.out`.

A.2 Execução do Projeto

Para executar o projeto compilado, utilize o seguinte comando:

```
./bin/tp2.out
```

Este comando executará o programa principal.

A.3 Limpeza dos Arquivos Compilados

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

```
make clean
```

Isso removerá os arquivos objetos e executáveis.

B Figuras

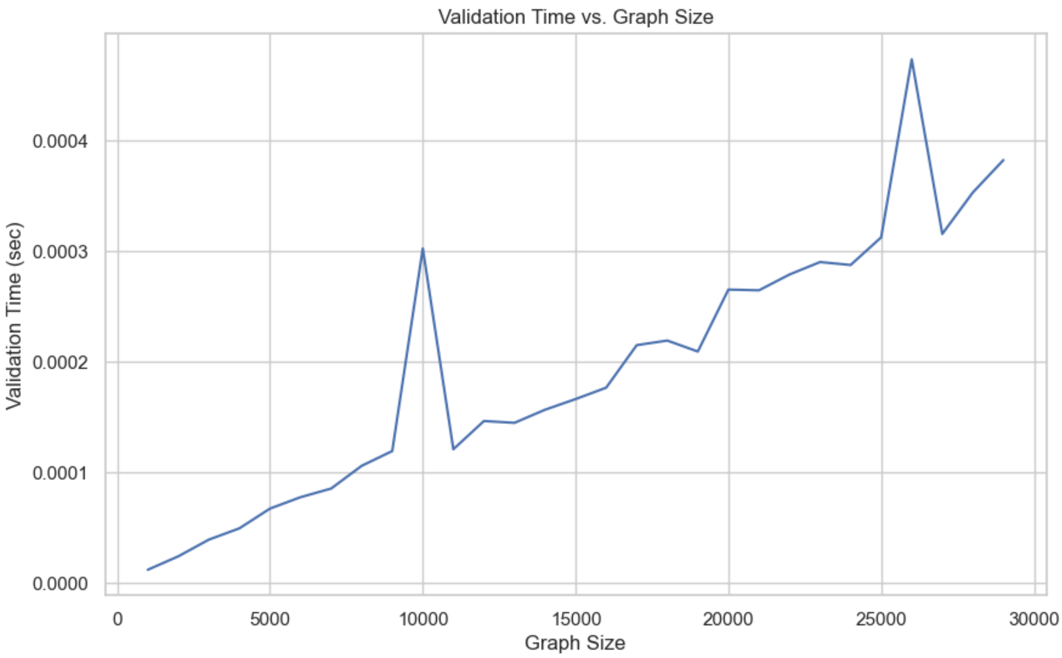

Figure 2: *Desempenho da função de validação dos grafos do experimento.*

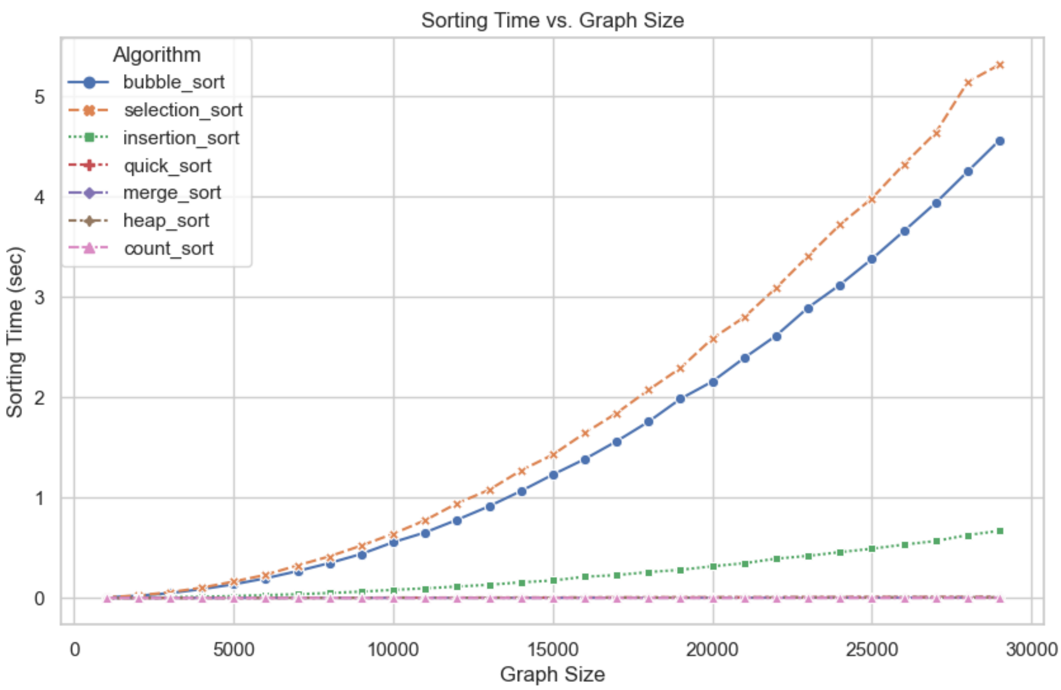

Figure 3: *Desempenho dos algoritmos de ordenação para os grafos do experimento.*