Trabalho Prático 2

Essa coloração é gulosa?

Raquel Gonçalves Rosa - 2020105815

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

raquelgr@ufmg.br

1. Introdução

Neste trabalho, nos foi proposto responder a seguinte pergunta: *"É possível obter essa coloração através de um algoritmo guloso?"*. Para isso, era necessário primeiro verificar a "gulosidade" do grafo passado, e caso fosse guloso, ordenar seus vértices pela cor dado o algoritmo de ordenação escolhido nos argumentos de entrada. Tendo em vista isso, este documento possui o objetivo de explicitar a implementação da solução para o problema proposto.

2. Método

2.1 Implementação

O programa foi desenvolvido em C++ com compilador G++, em ambiente Linux.

2.2 Estrutura de dados

Para a implementação do programa foram criadas duas classes:

- Classe **Grafo** (arquivo Grafo.hpp), responsável por armazenar o grafo de fato, contém os seguintes métodos, todos públicos:
  - Grafo::DefinirQuantidadeDeVizinhos
  - Grafo::InserirVizinho
  - Grafo::DefinirCor
  - Grafo::VerificarColoracaoGulosa
- Classe **Vertice** (arquivo Vertice.hpp), responsável por salvar os dados do vértice, sendo eles: rótulo, cor, quantidade de vizinhos e seus vizinhos, sendo o último um

array de inteiros apenas com o rótulo de cada vizinho. Para essa classe, não foram criados métodos, mas o arquivo contém três funções auxiliares para os métodos de ordenação, sendo elas: Troca, EhMenor e EhMaior.

Além das classes, importante ressaltar que também foi necessário criar um método de ordenação próprio, denominado **PairSort**.

2.2.1 PairSort

Esse método teve como base o método de ordenação BubbleSort, a diferença principal é que ele ordena pares de elemento por vez, usando dois loops para o seguinte comportamento: o primeiro loop ordena pares de elemento, sendo que o primeiro elemento tem índice ímpar e o segundo faz o contrário, ou seja, ordena pares de elemento, sendo que o primeiro elemento tem índice par. Entretanto, sua complexidade não conseguiu ser melhor do que a do BubbleSort, ele também tem como pior caso $O(n^2)$ que acontece quando o vetor está ordenado de forma inversa.

2.3 Organização

O programa está dividido em:

- 9 cabeçalhos, contidos na pasta *include*, sendo eles:
    - BubbleSort.hpp
    - Grafo.hpp
    - HeapSort.hpp
    - InsertionSort.hpp
    - MergeSort.hpp
    - PairSort.hpp
    - QuickSort.hpp
    - SelectionSort.hpp
    - Vertice.hpp
- 10 arquivos de código, contidos na pasta *src*, sendo eles:
    - BubbleSort.hpp
    - Grafo.hpp
    - HeapSort.hpp
    - InsertionSort.hpp
    - MergeSort.hpp
    - PairSort.hpp
    - QuickSort.hpp

- SelectionSort.hpp
- Vertice.hpp
- **Main.cpp,** responsável pela leitura do comando passado via terminal e chamada das funções necessárias.

3. Análise de Complexidade

A seguir será realizada a análise de complexidade de tempo de todos os métodos implementados no projeto.

→ **Vertices.hpp:**
- Todos os métodos dessa classe são $O(1)$, pois os métodos não dependem da quantidade de vértices, eles são feitos apenas para comparação ou troca entre pares de vértices.

→ **Grafo.hpp:**
- O método **VerificarColoracaoGulosa** tem complexidade $O(n \cdot c \cdot m)$, sendo n a quantidade de vértices do grafo, c a quantidade de cores necessárias que cada vértice deve ter em seus vizinhos e m a quantidade de vizinhos em cada vértice.
- Todos os outros métodos são $O(1)$, pois são apenas métodos de definição das propriedades do grafo.

→ **BubbleSort.hpp:**
- Como explicado em aula, o pior caso acontece quando o vetor, nesse caso os vértices, estão ordenados de forma contrária, sendo então $O(n^2)$, pois é necessário fazer a troca em todos os elementos.

→ **InsertionSort.hpp:**
- De forma similar ao BubbleSort, o pior caso acontece quando o vetor, nesse caso os vértices, estão ordenados de forma contrária, sendo então $O(n^2)$, pois é necessário fazer a movimentação de todos os elementos.

→ **SelectionSort.hpp:**
- Nesse método todos os casos têm a mesma complexidade, $O(n^2)$, pois é feita a verificação de todos os elementos independente se o vetor está ou não ordenado.

→ **MergeSort.hpp:**
- Esse algoritmo utiliza o conceito de dividir para conquistar, ou seja, ele divide o vetor na metade e vai dividindo até que só tenha dois elementos, nesse ponto ele compara e ordena os elementos. Após isso, é feita a junção para obter o

vetor ordenado. Para métodos de divisão e conquista, a complexidade é $O(n \cdot \log \quad n)$.

→ **QuickSort.hpp:**

- Como visto em sala, o pior caso desse algoritmo é $O(n^2)$, esse caso acontece quando o pivô escolhido é o extremo do vetor. O melhor caso e o caso médio têm complexidade $O(n \cdot \log \quad n)$, que acontece quando o pivô é exatamente o meio do vetor, utilizando assim o conceito de divisão e conquista.

→ **HeapSort.hpp:**

- Como explicado em aula, o HeapSort tem um funcionamento que se baseia em uma árvore binária de busca, portanto em qualquer caso, a complexidade é $O(n \cdot \log \quad n)$.

→ **PairSort.hpp:**

- Como mencionado na explicação do método, sua complexidade no pior caso é $O(n^{\,2})$ que acontece quando o vetor está ordenado de forma inversa.

4. Estratégias de Robustez

Algumas verificações foram implementadas a fim de garantir a corretude dos algoritmos, foram feitas verificações de tamanho válido do vetor, ou seja, quantidade válida de vértices e índices de acesso válido para evitar vazamento de memória. Para verificar a ausência de vazamento de memória, foi utilizado a ferramenta V*algrind,* como mostra a figura abaixo.

```
==8473== Memcheck, a memory error detector
==8473== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8473== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==8473== Command: ./bin/tp2.out
==8473==
m 6
3 1 2 3
3 0 2 4
3 0 1 5
1 0
1 1
1 2
1 2 3 2 1 1
1 0 4 5 1 3 2
==8473==
==8473== HEAP SUMMARY:
==8473==     in use at exit: 0 bytes in 0 blocks
==8473==   total heap usage: 21 allocs, 21 frees, 75,344 bytes allocated
==8473==
==8473== All heap blocks were freed -- no leaks are possible
==8473==
==8473== For lists of detected and suppressed errors, rerun with: -s
==8473== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5. Análise Experimental

Apesar da tentativa de utilizar a biblioteca *Memlog,* não foi possível obter gráficos que demonstrassem grande significância, entretanto, utilizando a biblioteca *getrusage* foi possível obter alguns dados de desempenho em questão de tempo, descritos na tabela abaixo:

Método	Tempo de execução (segundos)
BubbleSort	0.121693
SelectionSort	0.133145
InsertionSort	0.114942
QuickSort	0.100770
MergeSort	0.105862
HeapSort	0.110187
PairSort	0.125033

Essas métricas foram obtidas a partir de um grafo de 2000 vértices, gerado pelo *GeradorDeTestes* disponibilizado pelos professores. Os dados não necessariamente refletem a realidade, eles podem sofrer pequenas variações dado que o tempo da máquina pode ser afetado por aplicativos internos do sistema, não havendo assim, uma forma de controlar o teste em sua totalidade.

Analisando os dados obtidos, confirma-se o que foi explicado na análise de complexidade, os métodos com menor complexidade são também os com menor tempo de execução, sendo eles o QuickSort, MergeSort e HeapSort. Dentre os métodos mais simples, todos possuem a mesma complexidade para o pior caso, e isso também se reflete no tempo de execução, sendo eles InsertionSort, BubbleSort e SelectionSort.

O método autoral PairSort teve um desempenho parecido com os métodos mais simples, apesar de ter sido planejado para ser uma melhoria do BubbleSort, seu tempo de execução não reflete uma melhor performance, entretanto, ele teve um tempo de execução acima do SelectionSort, o que não era esperado.

6. Conclusões

A implementação dos problemas põe em prática os conteúdos vistos em sala, e nos convida a explorar mais os métodos de ordenação existentes e suas variações, além do mais, o exercício de construir um método de ordenação próprio foi uma ótima forma de exercitar o conteúdo de ordenação num geral.

7. Bibliografia

CHAIMOWICZ, L; PRATES, R. Ordenação: Métodos Simples. 2020. Apresentação do Power Point.

CHAIMOWICZ, L; PRATES, R. Ordenação: MergeSort e QuickSort. 2020. Apresentação do Power Point.

CHAIMOWICZ, L; PRATES, R. Ordenação: Heapsort. 2020. Apresentação do Power Point.

Bubble sort - data structure and algorithm tutorials. Disponível em: <https://www.geeksforgeeks.org/bubble-sort/>. Acesso em: 12 nov. 2023.

Heap sort - data structures and algorithms tutorials. Disponível em: <https://www.geeksforgeeks.org/heap-sort/>. Acesso em: 12 nov. 2023.

Insertion sort - data structure and algorithm tutorials. Disponível em: <https://www.geeksforgeeks.org/insertion-sort/>. Acesso em: 12 nov. 2023.

Merge Sort - data structure and algorithms tutorials. Disponível em: <https://www.geeksforgeeks.org/merge-sort/>. Acesso em: 12 nov. 2023.

QuickSort - data structure and algorithm tutorials. Disponível em: <https://www.geeksforgeeks.org/quick-sort/>. Acesso em: 12 nov. 2023.

Selection sort – data structure and algorithm tutorials. Disponível em: <https://www.geeksforgeeks.org/selection-sort/>. Acesso em: 12 nov. 2023.