Trabalho Prático 2

Essa coloração é gulosa

Davi Porto Araújo 2022036004

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

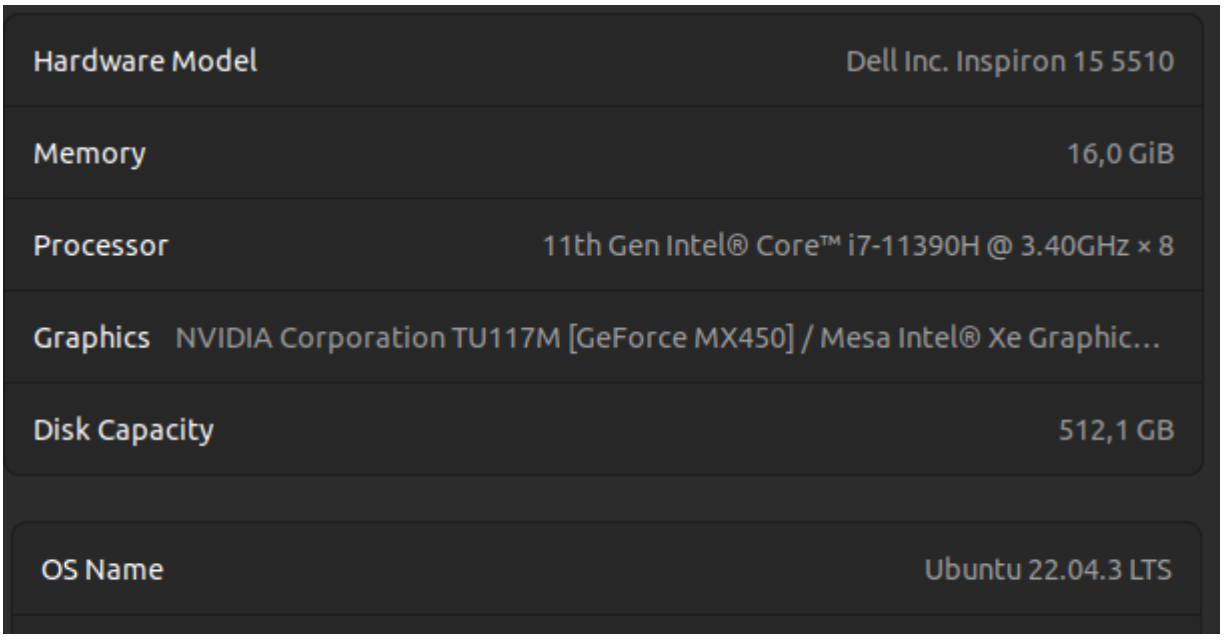
Belo Horizonte – MG – Brazil
portodaviporto@gmail.com

Introdução

Este relatório descreve um programa que visa decifrar se a coloração usando um determinado grafo foi feita por um algoritmo guloso. Conjuntamente diversos algoritmos de ordenação podem ser utilizados previamente, dado que o algoritmo que verifica se é guloso ou não necessita de receber os dados ordenados.

Descrição do ambiente

O ambiente em que o programa foi desenvolvido é testado é o seguinte:



• Linguagem c++

• compilador g++

Método

3.1 Tipos abstratos de dados:

O programa apresenta duas estruturas de dados principais, uma para representar o grafo que será analisado, está dividida em 2 partes: o grafo e os vértices. A segunda é um heap, este foi necessário para para a implementação do método de ordenação heap sorte.

O grafo foi implementado usando um ponteiro de ponteiro para representar um array de vértices. Escolhi usar um array de apontadores em vez dos vértices em si porque a mudança de 2 valores, algo que acontece com frequência alta nos métodos de ordenação, fica mais barata, dado que com os apontadores apenas é necessário inverter os endereços de memória que os mesmo guardam, isto pode ser feito com o método std::swap() fornecido pela std(standard library) do c++.

Caso contrário, isto é teria armazenado os vértices em si no array a mudança envolveria uma cópia inteira de cada vértice a cada mudança. Além da necessidade de variáveis auxiliares o que além de aumentar o tempo gasto aumenta o gasto de memória. O que impactaria severamente a performance do programa dado a grande quantidade de trocas feitas por alguns métodos de ordenação.

Por sua vez, o heap foi implementado usando um array que representa uma árvore em vez de apontadores, isto tanto porque o tamanho é fixo quanto pela facilidade de achar os filhos e pais de cada nó usando multiplicação e adição. Desta forma o espaço consumido pelos heap é menor, sem os apontadores adicionais desnecessários é tem-se uma maior celeridade para achar nós filhos ou pais dado que os itens estão sequencialmente na memória é não tem que ficar percorrendo apontadores causando indireções.

.2 Métodos

O programa possui 2 métodos principais, além de um para cada método de ordenação implementado, totalizando 9 métodos. Também foram implementados

overloading operators, de comparação < e <=, também o << para imprimir o grafo quando o algoritmo usado para a coloração for guloso.

Os métodos de ordenação como bubble, insert e selection sort. Estes foram inspirados pelos slides e aulas disponibilizadas pelo professor no ambiente virtual. o algoritmo de ordenação implementado por mim visa evitar o pior caso do quicksort. Isto foi feito levando o menor elemento para a primeira posição do vetor, fazendo assim com que a partição seja melhor feita.

Os métodos que que foram *overloaded*, os de comparação < e <= visam considerar o critério de desempate quando 2 vértices possuem a mesma cor na ordenação, isto foi feito considerando também o *label* de cada vértice. Já o << foi feito para, quando o algoritmo for guloso, imprimir a permutação utilizada.

Finalmente o método que verifica se a coloração é resultante de um algoritmo guloso. Este percorre os vértices do grafo e, para cada vértice com uma cor diferente da menor cor encontrada até o momento, verifica se todos os vértices vizinhos com cores menores já foram coloridos com sucesso. Se algum vizinho não foi colorido adequadamente, o método retorna falso, indicando que a coloração não é gulosa. Caso contrário, retorna verdadeiro, indicando que a coloração é resultado de um algoritmo guloso.

Análise de Complexidade

A complexidade dos algoritmos de ordenação segue a mesma dos slides, dado que os algoritmos não possuem modificações bruscas. O meu próprio também é nlog(n) porque o que é feito para melhorar o quicksort custa O(n) como nlog(n) é assintoticamente dominante o algoritmo continua nlog(n).

O algoritmo que verifica se a coloração do grafo é resultado de um algoritmo guloso. Por sua vez tem complexidade, no pior caso, $O(n^3)$ que é polinomial. No entanto o caso médio tende a ser melhor, isto porque assim que o algoritmo detecta que o grafo não possui coloração gulosa ele retorna falso.

Estratégias de Robustez

As principais estratégias usadas foram o auxílio do valgrind durante o desenvolvimento e o uso de exceções. Em pontos críticos do programa, a título de exemplo, a entrada é verificada, caso o método de ordenação escolhido seja diferente de um dos suportados uma exceção é lançada e interrompe a execução do programa.

Em outras partes também o mesmo cuidado foi tomado, como o uso de unsigned integer onde foi possível para evitar entradas negativas onde não faria sentido, como por exemplo na quantidade de alguma entidade, exemplificando o número de vértices.

Além disso o código se encontra padronizado integralmente em c++, usando o idioma inglês, seguindo os padrões snake case para a nomenclatura de variáveis e métodos. o camelCase para classes e afins. Concomitantemente a implementação foi segregada da especificação. Respectivamente em translation units(.cpp) e header files (.h).

Todos os header files do código foram também documentados usando o padrão javadoc. Conjuntamente os arquivos do tipo translation units(.cpp) foram comentados em pontos específicos norteando facilitar o entendimento.

Por último, alguns testes foram executados utilizando a ferramenta valgrind, observando diversos aspectos do programa com os módulos disponíveis como o que verifica possíveis vazamentos de memória.

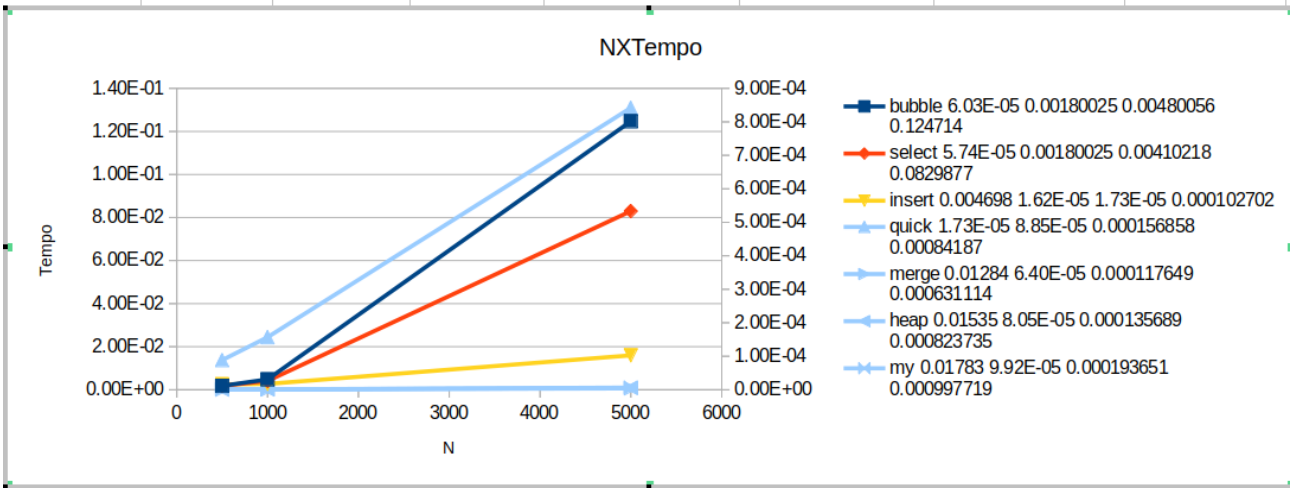
Para tentar identificar algum problema relacionado a este quesito. Vários testes de estresse foram executados. Estes foram projetados com o auxílio do gerador de casos de teste disponibilizado no ambiente virtual. Todos os métodos de ordenação foram testados com um grafo de dez mil vértices. Isto foi executado por um script feito em python. O teste reafirmou a corretude do programa como pode ser observado abaixo em um dos casos .

```
==199947== Memcheck, a memory error detector
==199947== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==199947== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==199947== Command: ./tp2.out
==199947== Parent PID: 43787
==199947==
==199947==
==199947== HEAP SUMMARY:
==199947==     in use at exit: 0 bytes in 0 blocks
==199947==   total heap usage: 23,178 allocs, 23,178 frees, 665,866 bytes allocated
==199947==
==199947== All heap blocks were freed -- no leaks are possible
==199947==
==199947== For lists of detected and suppressed errors, rerun with: -s
==199947== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Análise Experimental

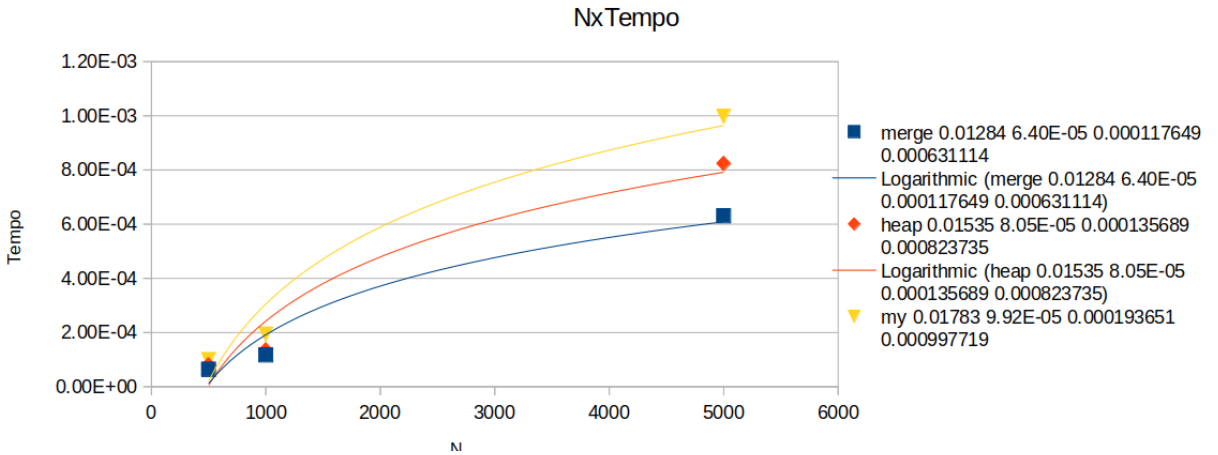
Como supracitado, utilizei um script em python para rodar o programa com cada método de ordenação, as entradas foram geradas pelo pelo script disponibilizado no moodle. O gráfico a seguir mostra os resultados de cada algoritmo de ordenação.

n	bubble	select	insert	quick	merge	heap	my
100	6.03E-05	5.74E-05	0.004698	1.73E-05	0.01284	0.01535	0.01783
500	0.00180025	0.00180025	1.62E-05	8.85E-05	6.40E-05	8.05E-05	9.92E-05
1000	0.00480056	0.00410218	1.73E-05	0.000156858	0.000117649	0.000135689	0.000193651
5000	0.124714	0.0829877	0.000102702	0.00084187	0.000631114	0.000823735	0.000997719



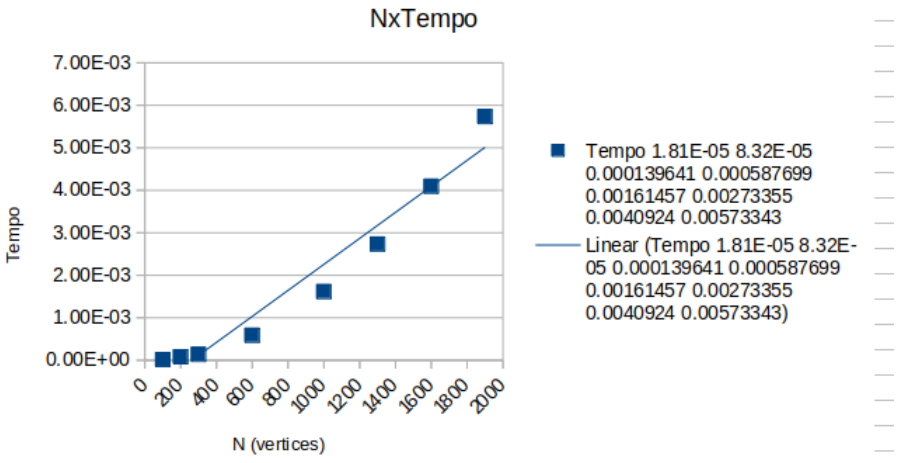
Como pode-se observar acima os algoritmos mais inocentes apresentam uma complexidade linear, já os mais elaborados apresentam uma complexidade logarítmica, que por questões de perspectiva não foi possível analisar neste gráfico mas será possível no seguinte. Devido a peculiaridades da entrega o quicksort não performou tão bem quanto o esperado, entretanto com alguns ajustes como feito no meu próprio método ele passa a performar otimamente.

n	merge	heap	my
100	0.01284	0.01535	0.01783
500	6.40E-05	8.05E-05	9.92E-05
1000	0.00011765	0.00013569	0.00019365
5000	0.00063111	0.00082374	0.00099772



Neste gráfico é possível observar melhor o caráter logarítmico dos outros algoritmos como esperado, especialmente a melhora que o meu apresentou em relação ao quicksort tradicional com as entradas fornecidas pelo gerador de testes.

n	Tempo
100	1.81E-05
200	8.32E-05
300	0.000139641
600	0.000587699
1000	0.00161457
1300	0.00273355
1600	0.0040924
1900	0.00573343



Por último, acima temos a análise experimental do tempo gasto pelo método que verifica se a coloração do grafo foi gerado por um algoritmo guloso. Como supracitado na parte teórica, o algoritmo apresenta uma complexidade linear, o que foi comprovado pelo gráfico acima.

Conclusões

Como explicitado anteriormente, o programa apresentado verifica se a coloração do grafo foi gerada por um algoritmo guloso, para isto ele preordena os vértices do grafo, através de um dos 7 métodos de ordenação que pode ser escolhido.

Durante a arquitetura e confecção do código, vários conceitos foram revisitados, desde POO(programação orientada a objetos) como estruturas de dados, como heap, maneiras de implementar grafos, adaptadas a eficiente solução do problema proposto. Além disso foi possível entender como cada método de ordenação funciona e suas diferenças tanto do ponto de vista de implementação como análise da performance dos mesmos.

Bibliografia

1. Slides da disciplina fornecidos pelos professores no moodle, juntamente com as aulas que ocorreram de forma remota durante a pandemia.
2. Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática3a. Edição, Elsevier, 2012.
3. Discrete Mathematics and Its Applications Eighth Edition Kenneth H. Rosen
4. https://www.geeksforgeeks.org/greedy-algorithms acessado em 5 de novembro de 2023.
5.