UNIVERSIDADE FEDERAL DE MINAS GERAIS

TRABALHO PRÁTICO 2
ESSA COLORAÇÃO É GULOSA?

VITOR HUGO BRAGA VIEIRA, MATRÍCULA: 2018076706
PROFESSORES : WAGNER MEIRA JÚNIOR / EDER FERREIRA DE FIGUEIREDO

Belo Horizonte
2023

1 INTRODUÇÃO

Este trabalho concentra-se na avaliação específica de grafos para determinar se a coloração aplicada é gulosa. Uma coloração é considerada gulosa quando cada vértice é colorido com a menor cor possível que ainda não foi usada em nenhum de seus vértices adjacentes. A importância deste estudo reside no fato de que a coloração gulosa é uma estratégia eficaz e comumente empregada devido à sua simplicidade e eficiência computacional, sendo particularmente útil em cenários onde é necessário encontrar uma solução de coloração rapidamente, mesmo que não seja a solução ótima. Após a verificação da coloração do grafo, procedemos com a ordenação dos vértices utilizando alguns dos algoritmos vistos na disciplina. Esta ordenação é realizada com base nas cores atribuídas, e em casos de cores iguais, os rótulos dos vértices são usados como critério de desempate. Esta etapa de ordenação tem implicações práticas significativas, como facilitar a visualização e análise do grafo colorido, além de ser útil em aplicações que exigem processamento posterior, como em algoritmos de otimização e em problemas de agendamento.

2 MÉTODO

2.1 ESPECIFICAÇÕES DA MÁQUINA :

- Processador: 11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz, 2918 Mhz, 4 Core(s), 8 Logical Processor(s)
- Memória RAM: Installed Physical Memory (RAM) 16.0 GB
- Sistema Operacional: WSL - UBUNTU 20.04 LTS
- Linguagem de Programação: C
- Compilador: gcc

2.2 ESTRUTURAS DE DADOS :

Na implementação deste trabalho, usamos listas encadeadas, arrays dinâmicos e uma estrutura auxiliar. Conforme o padrão das entradas, as listas encadeadas foram utilizadas na representação de cada vértice e suas conexões (arestas) com outros vértices, permitindo que cada lista de adjacência do grafo armazenasse um número variável de vizinhos (outros vértices aos quais está conectado). Os arrays dinâmicos foram utilizados na representação do próprio grafo e na armazenagem de listas de adjacência para cada vértice. Foi implementada uma estrutura auxiliar chamada 'VerticeCor' para armazenar informações sobre a cor e o índice de um vértice. Essa estrutura foi usada para facilitar a ordenação dos vértices pela cor e, em caso de empate, por índice.

2.3 TIPOS ABSTRATOS DE DADOS :

Grafo (Grafo):

- O TAD *Grafo* representa um grafo inteiro.
- Ele encapsula o número de vértices (`num_vertices`), o número de arestas (`num_arestas`), e a lista de adjacência (`lista_adjacencia`).
- As operações realizadas sobre este TAD incluem a criação de um novo grafo, inserção de vértices, inserção de arestas, verificação da coloração gulosa, ordenação dos vértices, e a liberação dos recursos associados.

Vértice (Vertice):

- O TAD *Vertice* representa um vértice no grafo.
- Ele contém informações sobre o valor ou identificador do vértice (`valor`), a cor atribuída ao vértice (`cor`), e um ponteiro para o próximo vértice na lista de adjacência (`prox`).
- Este TAD é utilizado na lista de adjacência para representar as conexões entre os vértices.

Vértice Cor (VerticeCor):

- O TAD *VerticeCor* é uma estrutura auxiliar usada para armazenar o índice e a cor de um vértice.
- Embora mais simples, este TAD é útil para a ordenação dos vértices do grafo com base em suas cores.

2.4 FUNÇÕES :

NovoGrafo(int numvertices):

- Cria um novo grafo com um número especificado de vértices.
- Aloca memória para o grafo e inicializa a lista de adjacência com valores nulos.
- Retorna um ponteiro para o novo grafo criado.

DeletaGrafo(Grafo* g):

- Libera a memória alocada para um grafo.
- Primeiro, percorre cada lista de adjacência e libera todos os vértices.
- Em seguida, libera a lista de adjacência e o próprio grafo.

InsereVertice(Grafo* g):

- Adiciona um novo vértice ao grafo.
- Aumenta o tamanho da lista de adjacência para acomodar o novo vértice.
- Incrementa o contador do número de vértices no grafo.

InsereAresta(Grafo* g, int v, int w):

- Adiciona uma aresta entre dois vértices (v e w) no grafo.
- Cria um novo vértice que é adicionado à lista de adjacência do vértice v.
- Incrementa o contador do número de arestas no grafo.

InsereCores(Grafo* g, int* cores):

- Atribui cores aos vértices do grafo.
- Percorre os vértices do grafo e atribui a cada um a cor correspondente do array de cores fornecido.

VerificaColoracaoGulosa(Grafo* g):

- Verifica se a coloração do grafo é gulosa.
- Para cada vértice, verifica se poderia ter sido colorido com uma cor menor sem conflitar com as cores dos vértices adjacentes.
- Retorna `true` se a coloração for gulosa, caso contrário, retorna `false`.

ImprimeVertices(VerticeCor* verticesOrdenados, int num_vertices):

- Imprime os índices dos vértices de um grafo na ordem em que estão armazenados no array `verticesOrdenados`.
- Utilizado para exibir os vértices após terem sido ordenados.

TrocaVerticeCor(VerticeCor* a, VerticeCor* b):

- Troca os valores de dois vértices. Usada nos algoritmos de ordenação.

Partition(VerticeCor* array, int low, int high):

- Parte fundamental do QuickSort. Organiza os elementos com base em um 'pivot'.
- Retorna o índice do 'pivot' após a organização.

QSort(VerticeCor* array, int low, int high):

- Implementação do algoritmo QuickSort. Ordena os elementos do array de forma recursiva usando a função Partition.
- Chama a si mesma para as partes do array antes e depois do 'pivot'.

QuickSort(Grafo* g):

- Prepara um array de vértices baseado no grafo e aplica o algoritmo QuickSort para ordená-los.
- Retorna o array de vértices ordenado.

Merge(VerticeCor* array, int left, int middle, int right):

- Combina dois subarrays ordenados (parte do MergeSort).
- Cria arrays temporários para facilitar a combinação.

MSort(VerticeCor* array, int left, int right):

- Implementação do algoritmo MergeSort. Divide o array em duas metades, chama a si mesma para cada metade e depois combina as duas metades ordenadas.

MergeSort(Grafo* g):

- Prepara um array de vértices baseado no grafo e aplica o algoritmo MergeSort para ordená-los.
- Retorna o array de vértices ordenado.

heapify(VerticeCor* array, int n, int i):

- Função usada no HeapSort para manter a propriedade de heap em uma árvore. Ajusta um nó no heap.

HSort(VerticeCor* array, int n):

- Implementação do algoritmo HeapSort. Organiza o array em um heap e então extrai os elementos um por um, mantendo a ordenação.

HeapSort(Grafo* g):

- Prepara um array de vértices baseado no grafo e aplica o algoritmo HeapSort para ordená-los.
- Retorna o array de vértices ordenado.

BubbleSort(Grafo* g):

- Implementação do algoritmo Bubble Sort. Compara pares adjacentes de vértices e os troca se estiverem na ordem errada.
- Retorna o array de vértices ordenado.

InsertionSort(Grafo* g):

- Implementação do algoritmo Insertion Sort. Reorganiza o array de vértices inserindo cada elemento na posição correta
- Retorna o array de vértices ordenado.

SelectionSort(Grafo* g):

- Implementação do algoritmo Selection Sort. Seleciona repetidamente o menor elemento do array não ordenado e o coloca na posição correta.
- Retorna o array de vértices ordenado.

BubbleSort_Variation(Grafo* g):

- Uma variação do Bubble Sort que ordena os elementos em passos pares e ímpares separadamente.
- Retorna o array de vértices ordenado.

OBS: Nos algoritmos instáveis, incluindo QuickSort, HeapSort e SelectionSort, foi implementada uma condição adicional para ordenar pelo índice do vértice quando as cores dos vértices são iguais.

3 ANÁLISE DE COMPLEXIDADE

Seja o número de vértices do grafo denotado por n e o número de arestas do grafo denotado por m.

Tabela 1 – Complexidade dos Algoritmos de Ordenação

Algoritmo	**Melhor Caso**	**Pior Caso**	**Caso Médio**	**Memória**
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
BubbleSort_Variation	O(n)	$O(n^2)$	$O(n^2)$	O(1)

Tabela 2 – Complexidade das Funções de Manipulação de Grafos

Função	**Complexidade**	**Motivo**
NovoGrafo	O(n)	Inicialização de um array de tamanho n
DeletaGrafo	O(n + m)	Liberação de todos os vértices e arestas
InsereVertice	O(n)	Uso de realloc, que pode ser O(n) no pior caso
InsereAresta	O(1)	Inserção direta na lista de adjacência
InsereCores	O(n)	Atribuição de cor a cada vértice
VerificaColoracaoGulosa	O(n + m)	Verificação em todos os vértices e suas arestas
ImprimeVertices	O(n)	Percorre e imprime cada vértice uma vez

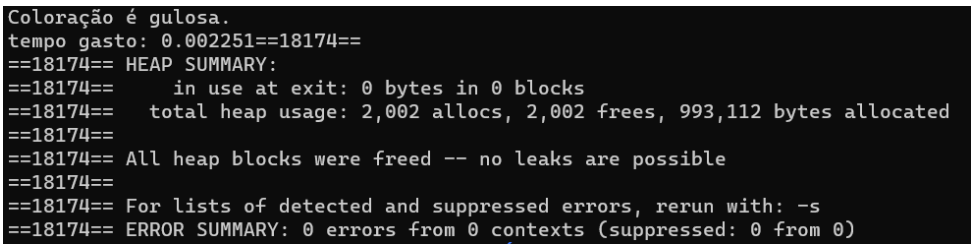
.

4 ESTRATÉGIAS DE ROBUSTEZ

Várias estratégias de robustez foram aplicadas tanto nos algoritmos de ordenação quanto nas funções de manipulação de grafos. Estas estratégias são importantes para garantir que o código funcione corretamente em uma variedade de cenários, lidando com erros potenciais e casos inesperados.

- **Verificações de Alocação de Memória:** Após alocações com `malloc` e `realloc`, verificações são realizadas para assegurar que a memória foi alocada com sucesso.
- **Tratamento de Casos de Falha:** Em situações de falha na alocação de memória, o código geralmente retorna `NULL` ou encerra a função, prevenindo a execução com dados inválidos.
- **Liberação de Recursos Alocados:** A função `DeletaGrafo` libera cuidadosamente a memória de cada vértice e da lista de adjacência, evitando vazamentos de memória.
- **Verificação de Entradas Válidas:** Funções como `InsereAresta` e `VerificaColoracaoGulosa` verificam a validade dos índices dos vértices.
- **Manuseio Cuidadoso de Ponteiros:** O uso correto de ponteiros é mantido, especialmente em funções críticas, para minimizar o risco de erros.
- **Validação de Parâmetros de Funções:** Parâmetros de entrada são validados para garantir que as funções lidem com entradas inválidas de maneira apropriada.
- **Uso de Lógica de Controle Condicional:** Estruturas condicionais são utilizadas para assegurar que certas ações ocorram apenas sob condições específicas.

Além disso, *não há memory leak no programa*, segue um exemplo testado usando o Valgrind.



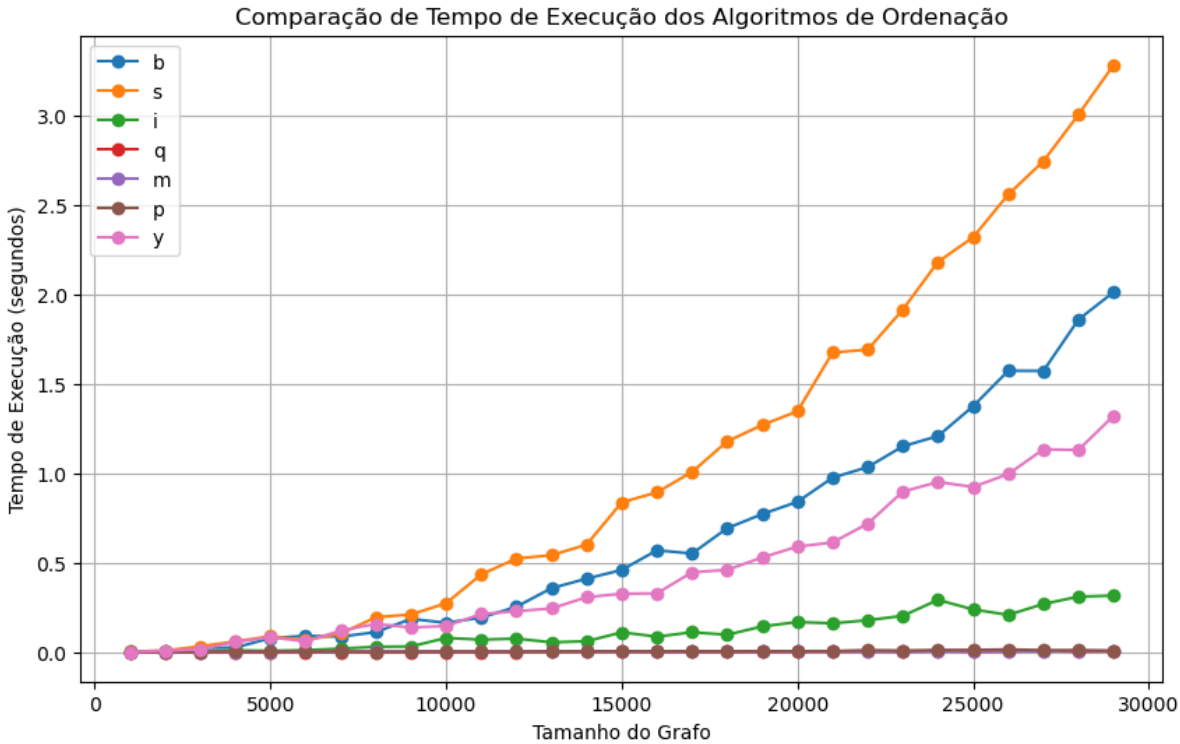
5 ANÁLISE EXPERIMENTAL

Para realizar uma análise experimental detalhada do código e avaliar sua eficiência, empregamos métodos de instrumentação de código utilizando a biblioteca time.h.

```
}   else if (opcao == 'y') {
    if (VerificaColoracaoGulosa(g)== true){
        clock_t begin = clock();
        VerticeCor* VerticesOrdenados =  BubbleSort_variation(g);
        //ImprimeVertices(VerticesOrdenados,numVertices);
        free(VerticesOrdenados);
        clock_t end = clock();
        printf("tempo gasto: %f", (double)(end - begin) / CLOCKS_PER_SEC);
    }
    else{
        printf("0");
    }
}
```

Figura 1 – Exemplo

Essa abordagem nos permitiu medir o tempo de execução de cada algoritmo de ordenação implementado no programa. O objetivo dessa análise era compreender como os diferentes algoritmos de ordenação se comportam em termos de eficiência de tempo, especialmente à medida que o tamanho do grafo aumenta. Para garantir uma avaliação abrangente e sistemática, desenvolvemos um script automatizado que executa testes em uma série de grafos com tamanhos variados. O intervalo de tamanhos de grafos testados variou de 1000 a 30000 vértices.



A análise detalhada do gráfico revela informações significativos sobre o desempenho dos algoritmos de ordenação nos cenários de teste. Entre os métodos de ordenação mais simples, observamos que o SelectionSort apresentou o menor nível de eficiência. Esse resultado está alinhado com as expectativas teóricas, dado que o SelectionSort tem uma complexidade de tempo $O(n^2)$ em todos os cenários - melhor, médio e pior caso. Por outro lado, o InsertionSort destacou-se como o mais eficiente entre os métodos simples. Essa superioridade pode ser atribuída à sua menor sobrecarga operacional em comparação com outros algoritmos de complexidade semelhante. Interessantemente, o algoritmo BubbleSort_variation, uma variação do BubbleSort que desenvolvi, mostrou-se mais eficiente do que o BubbleSort tradicional nos casos de teste, apesar de ambos compartilharem a mesma ordem de complexidade teórica. Esse resultado sugere que as otimizações implementadas na variação tiveram um impacto positivo no desempenho. Quanto aos métodos considerados mais eficientes, como QuickSort, HeapSort e MergeSort, os resultados dos testes mostraram tempos de execução muito próximos entre eles, com diferenças quase imperceptíveis. Esta observação é consistente com a teoria, uma vez que todos esses algoritmos têm uma complexidade de tempo médio de O(n log n). Tal proximidade nos tempos de execução destaca a eficácia desses algoritmos em lidar com grandes conjuntos de dados, reforçando sua adequação para aplicações em que o desempenho é um critério indispensável.

6 CONCLUSÃO

Neste trabalho, implementamos e analisamos vários algoritmos de ordenação, aplicando-os ao contexto específico de ordenação de vértices em grafos. Através desta investigação, conseguimos obter insights valiosos sobre a eficiência e aplicabilidade prática de cada método, desde algoritmos mais simples como BubbleSort, SelectionSort e InsertionSort, até métodos mais sofisticados como QuickSort, HeapSort e MergeSort. A implementação desses algoritmos nos proporcionou uma compreensão profunda não apenas das teorias subjacentes, mas também do impacto prático dessas teorias quando aplicadas a conjuntos de dados de diferentes tamanhos. Observamos, por exemplo, que enquanto algoritmos de complexidade $O(n^2)$ podem ser adequados para pequenos conjuntos de dados, sua eficiência diminui significativamente à medida que o tamanho do conjunto de dados aumenta. Por outro lado, algoritmos com complexidade O(n log n) demonstraram ser robustos e eficientes mesmo em grandes conjuntos de dados, sublinhando sua utilidade em aplicações onde o desempenho é uma consideração crítica. Além disso, a criação do BubbleSort_variation ilustrou como modificações e otimizações em algoritmos existentes podem levar a melhorias de desempenho. Este

estudo também enfatizou a importância de realizar análises experimentais para validar e compreender o comportamento dos algoritmos. A utilização da biblioteca time.h para medir o tempo de execução ofereceu uma perspectiva quantitativa que complementa o entendimento teórico, permitindo uma avaliação mais precisa e baseada em evidências da eficácia dos diferentes algoritmos.

7 REFERÊNCIAS BIBLIOGRÁFICAS

- `https://en.wikipedia.org/wiki/Greedy_coloring`
- *Eder Figueiredo e Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.*

8 INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Para compilar o programa, utilize o comando make all. Em seguida, para executar, navegue até o diretório onde se encontra o executável - neste caso, execute ./tp2.out. Ao iniciar o programa, você deve primeiro especificar o método de ordenação desejado da forma indicada nas instruções do TP2. Após isso, insira a quantidade total de vértices do grafo. As linhas que seguem devem detalhar as conexões dos vértices: cada linha deve começar com a quantidade de vértices conectados ao vértice i (referente à linha atual), seguida pelos índices dos vértices aos quais ele está conectado. Finalmente, forneça a lista de cores para a coloração do grafo, correspondendo a cada vértice.

Exemplo:

```
./tp2.out
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
```