UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas – ICEX
Departamento de Ciência da Computação – DCC
DCC205 – ESTRUTURAS DE DADOS

TRABALHO PRÁTICO II

Essa Coloração é Gulosa?

Luis Felipe Belasco Silva
2022035474

Professor: Wagner Meira Jr.

1. INTRODUÇÃO

Para o problema de avaliação da coloração do grafo, desejamos verificar se o grafo foi colorido por um algoritmo guloso. O programa recebe na linha de comando o método de ordenação que será utilizado para ordenar o grafo, o número de nós e, em seguida, a quantidade de vizinhos de cada nó e seus rótulos. A solução empregada utiliza o TAD Grafo, que implementa o TAD Nó, descritos nas classes “Graph” e “Node” respectivamente.

CLASSE NODE

A classe Node representa um nó em um grafo. Contém informações sobre rótulo, cor e vizinhos de um nó. Essa classe é principalmente utilizada em conjunto com a classe Graph para construir e manipular grafos.

Node()

- Descrição: Construtor padrão para a classe Node.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

int getLabel

- Descrição: Obtém o rótulo do nó.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

void setLabel(int newLabel)

- Descrição: Define o rótulo do nó.
- Parâmetros:
  - newLabel: novo rótulo do nó.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

void addNeighbour(int neighbourId)

- Descrição: Adiciona um vizinho ao nó.
- Parâmetros:
- neighbourId: O rótulo do novo vizinho.
- Complexidade:
  - Tempo: O(numNeighbours) (pior caso, se alocação dinâmica de memória for necessária)
  - Espaço: O(numNeighbours) (pior caso, se alocação dinâmica de memória for necessária)

int getColour()

- Descrição: Obtém a cor do nó.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

void setColour(int newColour)

- Descrição: Define a cor do nó.
- Parâmetros:
  - newColour: A nova cor para o nó.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

int getNeighbours()

- Descrição: Obtém um array de rótulos dos vizinhos.
- Complexidade:
  - Tempo: O(1)

int getNumNeighbours()

- Descrição: Obtém o número de vizinhos.

CLASSE GRAPH

A classe Graph representa um grafo e inclui os seguintes algoritmos de ordenação, como bubble sort, selection sort, insertion sort, quicksort, mergesort, heapsort e timsort. Após ordenar os membros do grafo com respeito a cor (utilizando rótulo para desempate), o programa utiliza um validador para verificar se as cores de cada nó foram definidas por meio de um algoritmo guloso.

Graph()

- Descrição: Construtor para a classe Graph.
- Parâmetros:
  - size: O tamanho do grafo.
- Complexidade:
  - Tempo: O(size)
  - Espaço: O(size)

~Graph()

- Descrição: Destrutor para a classe Graph.
- Complexidade:
  - Tempo: O(size)
  - Espaço: O(size)

Node* getMembers()

- Descrição: Obtém o array de nós que representa os membros do grafo.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

int getSize()

- Descrição: Obtém o tamanho do grafo.
- Complexidade:
  - Tempo: O(1)
  - Espaço: O(1)

Node getByLabel(int label)

- Descrição: Obtém um nó pelo seu rótulo.
- Parâmetros:
  - label: O rótulo do nó a ser recuperado.
- Complexidade:
  - Tempo: O(size)
  - Espaço: O(1)

void bubbleSort()

- Descrição: Ordena o grafo usando o algoritmo bubble sort.
- Complexidade:
  - Tempo: O(size^2)
  - Espaço: O(1)

void selectionSort()

- Descrição: Ordena o grafo usando o algoritmo selection sort.
- Complexidade:
  - Tempo: O(size^2)
  - Espaço: O(1)

void insertionSort()

- Descrição: Ordena o grafo usando o algoritmo insertion sort.
- Complexidade:
  - Tempo: O(size^2)
  - Espaço: O(1)

void insertionSort(int left, int right)

- Descrição: Ordena uma parte do grafo usando insertion sort.
- Parâmetros:
  - left: O índice inicial da parte a ser ordenada.
  - right: O índice final da parte a ser ordenada.
- Complexidade:
  - Tempo: O((right left + 1)^2)
  - Espaço: O(1)

void quickSort(int down, int up)

- Descrição: Ordena o grafo usando o algoritmo quicksort.
- Parâmetros:
  - down: O índice inicial da parte a ser ordenada.
  - up: O índice final da parte a ser ordenada.
- Complexidade:
  - Tempo: O(size * log(size))
  - Espaço: O(size * log(size))

int partition(int down, int up)

- Descrição: Particiona o grafo para quicksort.
- Parâmetros:
  - down: O índice inicial da parte a ser particionada.
  - up: O índice final da parte a ser particionada.
- Complexidade:
  - Tempo: O(size)
  - Espaço: O(1)

void merge(int left, int mid, int right)

- Descrição: Funde duas partes ordenadas do grafo.
- Parâmetros:
  - left: O índice inicial da primeira parte.
  - mid: O índice final da primeira parte e ínicio da segunda parte.
  - right: O índice final da segunda parte.
- Complexidade:
  - Tempo: O(size)
  - Espaço: O(size)

void mergeSort(int left, int right)

- Descrição: Ordena o grafo usando o algoritmo mergesort.
- Parâmetros:
  - left: O índice inicial da parte a ser ordenada.
  - right: O índice final da parte a ser ordenada.
- Complexidade:
  - Tempo: O(size * log(size))
  - Espaço: O(size)

void heapify(int n, int i)

- Descrição: Transforma uma subárvore em um heap.
- Parâmetros:
  - n: O tamanho do heap.
  - i: O índice da raiz da subárvore.
- Complexidade:
  - Tempo: O(size)
  - Espaço: O(1)

void heapSort()

- Descrição: Ordena o grafo usando o algoritmo heapsort.
- Complexidade:
    - Tempo: O(size * log(size))
    - Espaço: O(1)

void timSort()

- Descrição: Ordena o grafo usando o algoritmo timsort.
- Complexidade:
    - Tempo: O(size * log(size))
    - Espaço: O(size)

bool greedy()

- Descrição: verifica se o grafo foi colorido de maneira gulosa.
- Complexidade:
    - Tempo: O(size^2)
    - Espaço: O(1)

ESTRATÉGIAS DE ROBUSTEZ:

- Adição de tratamento de exceção para os dados da entrada
- Checagem de nullptr antes das deleções e dos acessos
- Caso default do switch trata para modo de ordenação inválido

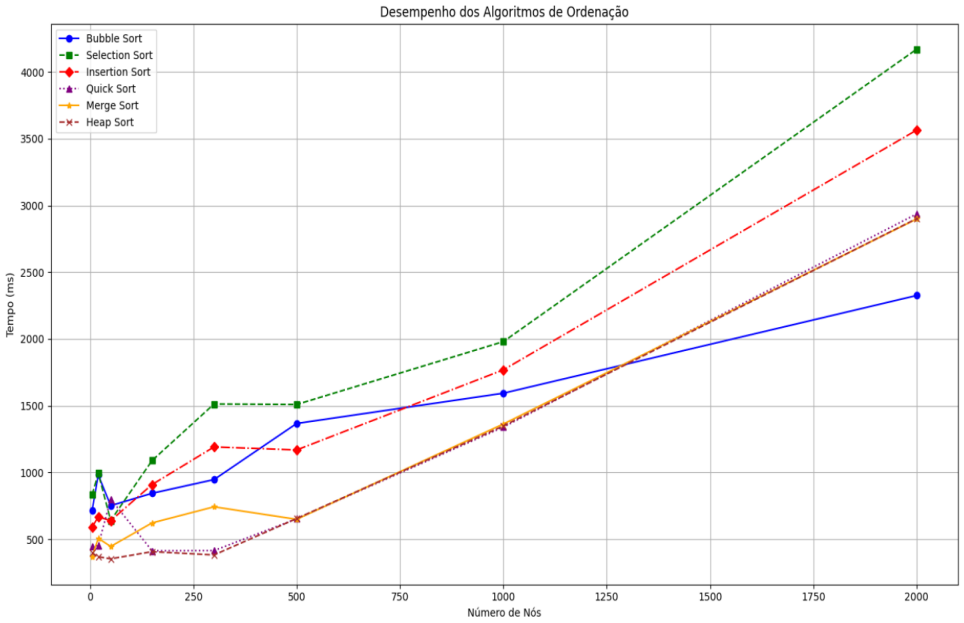
ANÁLISE EXPERIMENTAL:

Bubble Sort, Selection Sort e Insertion Sort: esses algoritmos de ordenação simples têm desempenho pior em comparação com os outros algoritmos apresentados. O Bubble Sort mostra um tempo de execução especialmente alto, tornando-se menos eficiente à medida que o número de elementos aumenta.

O Quick Sort demonstra um bom desempenho, especialmente em grandes conjuntos de dados. Sua complexidade média de tempo é geralmente melhor do que a de algoritmos de ordenação quadráticos, como Bubble, Selection e Insertion Sort.

O Merge Sort mantém um desempenho consistente e eficiente em uma ampla gama de tamanhos de conjunto de dados. Apresenta uma complexidade de tempo O(n log n), tornando-o eficaz para grandes conjuntos de dados.

O Heap Sort também demonstra um desempenho eficiente e é notavelmente constante em relação ao tamanho do conjunto de dados. Possui uma complexidade de tempo O(n log n) e é uma opção sólida para conjuntos de dados maiores.



Visualização em: grafico-ed2.png

DAS DECISÕES DE PROJETO

De maneira não usual, utilizou-se uma implementação de grafo com arrays. A diferença mais notável é na questão dos vizinhos de cada nó, que são definidos por um array de inteiros que são os rótulos recebidos na entrada, economizando espaço que seria gasto ao criar uma lista de Nós. É necessário pontuar que, apesar dessa implementação ser eficiente e simples de fazer, ela só é possível devido a natureza do problema e das entradas, já que as operações nos grafos só começam após sua definição, que também é feita de uma vez. Sendo assim, a estrutura de dados foi definida dessa forma, tendo em mente que os dados são estáticos.

CONCLUSÕES

O problema de coloração de grafos é utilizado para resolver problemas de rota em mapas, por exemplo, o que se prova bem útil no cotidiano. No contexto da atividade, utilizou-se dessa problemática para definir diferentes algoritmos de ordenação que seriam empregados na solução do problema e, por consequência, é possível observar o comportamento destes diante certos testes de estresse na entrada dos grafos. Ao final, conclui-se que é necessário avaliar o conjunto de dados com que se lida antes de escolher o algoritmo de ordenação a ser utilizado, para obter o maior proveito dessas ferramentas.

REFERÊNCIAS BIBLIOGRÁFICAS

1. Tim Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/timsort/
2. Quick Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/quick-sort/
3. Merge Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/merge-sort/
4. Insertion Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/insertion-sort/
5. Heap Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/heap-sort/
6. Selection Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/selection-sort/
7. Bubble Sort. Geeks For Geeks. Disponível em: https://www.geeksforgeeks.org/bubble-sort/
8. Luiz Chaimowicz e Raquel Prates, Slides virtuais da disciplina de Estruturas de Dados. Acesso em: Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.
9. Wagner Meira Jr., atividades práticas da disciplina Estrutura de Dados. Acesso em Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.