Trabalho Prático 2

Essa coloração é gulosa?

Júlio Guerra Domingues (2022431280)

Estruturas de Dados - DCC/UFMG - 2023-2

Introdução

O presente trabalho aborda o problema de ordenação de grafos e a análise de coloração gulosa. Diferentes métodos de ordenação foram aplicados aos vértices dos grafos, com enfoque à avaliação de coloração gulosa, em que cada vértice recebe a menor cor possível que não é igual à cor de nenhum de seus vizinhos adjacentes.

O método adotado envolve a leitura de grafos e seus dados de coloração através da entrada padrão, seguidas pela aplicação de vários métodos de ordenação e a subsequente avaliação da natureza gulosa da coloração.

Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema Operacional: MacOS Sonoma 14.0
- Processador: ARM Apple Silicon M2
- Memória RAM: 8,00 GB

Desenvolvemos uma estrutura de dados para representar grafos, bem como classes e funções para manipulá-los. Para a implementação dos grafos, utilizou-se a estrutura de vetores, principalmente com base em sua simplicidade e facilidade no manuseio.

Foram implementados diversos algoritmos de ordenação (bolha, seleção, inserção, quicksort, mergesort, heapsort e um método próprio) adaptados à nossa estrutura. A função Greedy avalia se a coloração do grafo é gulosa.

Principais Estruturas de Dados e Classes

Classe Grafo:

- Objetivo: Representar um grafo com vértices e arestas.
- Métodos:
    - Grafo(int): Construtor que inicializa o grafo com o número de vértices especificado.
    - ~Grafo(): Destrutor que libera a memória alocada.
    - insereAresta(int, int): Adiciona uma aresta ao grafo.
    - setCor(int, int): Atribui uma cor a um vértice.
    - getGrafo(), getVertices(), getNumeroVertices(): Métodos de acesso para os atributos.
    - Greedy(Vetor[], Vetor&, int): Verifica se a coloração do grafo é gulosa.

Classe Ordenacao:

- Objetivo: Implementar diferentes métodos de ordenação.
- Métodos:
    - Diversos métodos para implementar algoritmos de ordenação (Bolha, Selecao, Insercao, QuickSort, MergeSort, Heapsort, BolhaImparPar).
    - Ordena(char): Escolhe o método de ordenação baseado no caractere de entrada.

A entrada do programa consiste na escolha do método de ordenação, seguida dos dados do grafo e coloração. O grafo é construído e processado usando as classes Grafo, Vetor e Item.

Ordenação: Após a leitura e inicialização do grafo, os vértices são ordenados usando a classe Ordenacao, que encapsula diversos métodos de ordenação. Além dos métodos de ordenação por comparação mais convencionais (Bolha, Selecao, Insercao, QuickSort, MergeSort e Heapsort), implementados conforme as discussões em sala de aula, implementou-se ainda uma variação do *bubblesort* conhecido como *Odd-Even Transposition Sort* (BolhaImparPar).

O algoritmo de ordenação paralelo BolhaImparPar, proposto por Habermann em 1972, divide todas as operações de comparação e troca em duas fases: ímpar e par.

Essas duas fases são repetidas até que o arranjo esteja completamente ordenado. Sua principal vantagem consiste na capacidade de realizar várias comparações e trocas em paralelo, o que o torna adequado para ambientes de processamento paralelo. É um modelo estável simples e de fácil implementação, com boa performance em sistemas paralelos. Em sistemas de processamento único ou sequencial, todavia, seu desempenho é limitado.

Verificação de Coloração Gulosa: A função Greedy na classe Grafo avalia se a coloração do grafo é gulosa.

Saída: O programa imprime 0 se a coloração não for gulosa; caso contrário, imprime 1 seguido da permutação dos vértices ordenados.

Análise de Complexidade

A análise de complexidade de tempo e espaço do programa será focada nos algoritmos de ordenação implementados e nas operações dos grafos.

Funções de Ordenação:

- Bolha
  - Complexidade de Tempo: O pior caso é $O(n^2)$, onde n é o número de elementos. Isso acontece quando o vetor está em ordem inversa.
  - Complexidade de Espaço: O(1) - A ordenação é feita no local, e não são necessárias estruturas de dados adicionais significativas.
- Seleção
  - Tempo: $O(n^2)$ tanto no melhor quanto no pior caso.
  - Espaço: O(1) - Não requer espaço adicional significativo.
- Inserção
  - Tempo: $O(n^2)$ no pior caso e O(n) no melhor caso.
  - Espaço: O(1) - Ordenação no local.
- QuickSort:
  - Tempo: O(n log n) no caso médio e $O(n^2)$ no pior caso.
  - Espaço: O(log n) devido à pilha de recursão no caso médio.
- MergeSort:
  - Tempo: O(n log n) em todos os casos.
  - Espaço: O(n) - requer espaço adicional para a fusão.
- HeapSort:
  - Tempo: O(n log n) em todos os casos.
  - Espaço: O(1) - Ordenação no local.
- BolhaImparPar (*Odd-Even Transposition Sort*)
  - Tempo: $O(n^2)$ em sistemas sequenciais; em paralelo, pode ser mais eficiente.
  - Espaço: O(1) - Não requer espaço adicional.

Ressalta-se ainda a estabilidade dos métodos Bolha, Inserção, MergeSort e BolhaImparPar, que podem ser necessárias em alguns contextos de aplicação (o que não é o caso do presente trabalho).

Operações do Grafo:

- Inserção de Aresta:
  - Complexidade de Tempo: O(1) - A inserção de uma aresta é uma operação direta.
  - Complexidade de Espaço: Dependente da estrutura de dados usada para armazenar as arestas.
- Definição de Cor:
  - Tempo: O(1) - Atribuir uma cor a um vértice é uma operação direta.

    - Espaço: O(1) - Não requer espaço adicional.
- Verificação de Coloração Gulosa (Greedy):
    - Tempo: O(V + E), onde V é o número de vértices e E o número de arestas. Isso porque precisa verificar a cor de cada vértice e suas arestas adjacentes.
    - Espaço: O(1) - Não requer espaço adicional significativo.

Estratégias de Robustez

Algumas estratégias de robustez foram implementadas, de modo a garantir que o programa lide com exceções e erros de forma apropriada, evitando falhas inesperadas e tornando-o mais resistente a entradas incorretas ou inesperadas. Destacam-se as seguintes:

- Tratamento de erros e exceções: Foram adicionadas condições de erro em partes mais susceptíveis do código (notadamente aquelas em que há leitura de entrada), para fornecer mensagens de erro adequadas e informativas e garantir que o programa não falhe inesperadamente.

- Validação de entrada: Garantindo que os dados lidos (como número de vértices, arestas e cores) estejam dentro de um intervalo aceitável e façam sentido no contexto do problema. Isso evita problemas como índices fora de alcance.

- Lançamento de Exceções para Índices Fora do Intervalo: Nas funções insereAresta e setCor da classe Grafo, há verificações para garantir que os índices de vértices estejam dentro do intervalo válido. Se um índice estiver fora do intervalo, uma exceção do tipo std::out_of_range é lançada.

- Testes abrangentes: Alguns testes foram implementados no Makefile (podem ser verificados ao rodar com *make tests*) e o programa foi testado no VPL no Moodle, o que contribui para sua robustez.

Análise Experimental

O programa foi avaliado por meio das ferramentas Valgrind, biblioteca Memlog (fornecida pelo professor) e chrono (biblioteca do c++).

Por meio do gerador de casos de teste fornecido, diversos grafos foram simulados. Seguem os resultados e conclusões para cada método de ordenação:

- Bolha: Desempenho em geral mais lento, especialmente nos conjuntos de dados grandes. Em teoria, seria o método mais rápido em dados quase ordenados, por realizar menos trocas nesse contexto, mas isso não ocorreu nos casos de teste.
- Seleção: Desempenho discretamente superior ao da bolha, por realizar menos trocas em média. Menor variabilidade de acordo com a ordenação (ou não) da entrada.
- Inserção: Em geral, bastante eficiente em conjuntos de dados pequenos. Para conjuntos grandes apresentou tempo de execução semelhante ao da bolha.
- QuickSort: Método mais rápido em quase todos os contextos. Sob o ponto de vista de avaliação da localidade de referência, notou-se boa localidade de cache em tamanhos menores.
- MergeSort: Método eficiente de forma geral, independente da natureza dos dados. Consumo de memória superior na comparação com o QuickSort.
- HeapSort: Eficiente, de forma geral, porém inferior ao QuickSort em tempo e memória (provavelmente por operações complexas de heap).
- BolhaImparPar: Desempenho levemente superior ao da Bolha, significativamente pior que o QuickSort para todos os casos testados.

Conclusões

Ao longo desse trabalho, pudemos perceber a importância da escolha de métodos de ordenação de acordo com o perfil e volume dos dados avaliados. Métodos como QuickSort e MergeSort se destacaram pela eficiência em grandes conjuntos de dados, em consonância com suas complexidades teóricas favoráveis e discussões em sala de aula, O(n log n). Em contraste, métodos mais simples como Bolha e Seleção apresentaram limitações, especialmente em grandes volumes de dados, refletindo a importância de escolher estratégias de ordenação alinhadas com as características dos dados e as necessidades específicas do problema. A eficácia do método BolhaImparPar implementado, embora discretamente superior ao da Bolha, não alcançou a dos métodos mais avançados, ressaltando a relevância de selecionar algoritmos adequados para cada situação.

A avaliação da coloração gulosa, ainda que relevante para entender as propriedades do grafo e validar a solução proposta, não teve impacto significativo para a performance dos algoritmos de ordenação. A partir dos experimentos e análises realizadas, identificou-se o QuickSort como a escolha mais robusta e eficiente para a ordenação dos vértices do grafo na maioria dos casos, devido à sua eficiência temporal e boa localidade de referência.

Bibliografia

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press. ISBN: 9780262046305.
2. Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-Wesley Professional.
3. Habermann, N. (1972). Parallel Neighbor Sort (or the Glory of the Induction Principle). CMU Computer Science Report. Technical report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Rd Springfield, VA 22151.