Trabalho Prático II - Estrutura de Dados

Arthur Guilherme Rodrigues Luz

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

1 - Introdução

- O problema abordado trata-se da aplicação de alguns algoritmos de ordenação a fim de se fazer uma análise de tempo para diferentes valores de entrada. Os elementos a serem ordenados advém de um arquivo txt, onde em cada linha é proposta a seguinte configuração:
    - nome dados (CHAIMO 10101010, por exemplo)

- Dado o arquivo txt, devemos ordená-lo a partir de 4 possíveis configurações pré-estabelecidas e imprimir essa ordenação.

A seção 2 trata do método utilizado (implementação do código), já a seção 3 trata-se da análise de complexidade. As seções 4 e 5 tratam, respectivamente, das configurações experimentais e dos resultados obtidos. Por fim, temos a seção 6, que trata da conclusão, seguida das seções 7 e 8, que abordam instruções para compilação e a bibliografia utilizada.

2 - Método

2.1 - Configuração utilizada:

- Sistema operacional: Ubuntu 20.04.2.0 LTS
- Linguagem de programação implementada: C++
- Complicador utilizado: g++
- Processador utilizado: AMD Ryzen 2400g
- Quantidade de RAM: 8GB

2.2 - Algoritmos:

- Para armazenar de forma separada os dados lidos do arquivo txt, criei um “struct” com as propriedades nome e dado, para que assim eu pudesse aplicar as configurações de ordenação para priorizar de acordo com a propriedade da minha escolha.

- Os métodos de ordenação utilizados foram QuickSort, MergeSort, HeapSort e Radix Exchange Sort. As implementações foram retiradas dos slides

apresentados em aula e de sites na internet, já que o foco principal está mais na análise e não na implementação em si.

- **QuickSort:** Para implementar o QuickSort, utilizei duas funções. A função ”partition()” e a função “quickSort()”. A função “partition()” pega um “array” e define um elemento “x” da matriz como pivô. Coloca o “x” em sua posição correta na matriz (ou seja, a sua posição considerando a ordenação final correta) e coloca todos os elementos menores que “x” antes de “x”, assim como coloca todos os elementos maiores que “x” depois “x”. Já a função “quickSort()” tem um papel reduzido. Ela simplesmente faz uma chamada da função “partition()” e duas chamadas recursivas, gerando assim diversas partições que irão ordenar os elementos.

- **MergeSort:** Minha implementação divide o “array” de entrada em duas metades, faz uma chamada recursiva nessas duas metades e, em seguida, mescla as duas metades já ordenadas. A função “merge()” é usada para mesclar duas metades. A função “mergeSort()” faz duas chamadas recursivas, uma para cada metade, e chama a função “merge()” uma vez.

- **HeapSort:** Diferentemente dos slides, o HeapSort que implementei é zero-based, desse modo o filho esquerdo de um nó com ındice i é 2i + 1 e o filho direito 2i + 2. A partir desse princípio, na função “heapify()” construo o heap do “array” de entrada para que a partir disso, possa ordená-lo na função “heapSort()”, onde chamo o “heapify()”.

- **Radix Exchange Sort:** Utilizei a mesma implementação encontrada nos slides apresentados em aula, com a única diferença que passo o “struct” que criei como primeiro argumento para as funções “quicksortB()” e “radixSort()”. A função “quicksortB()” cria a partição baseada nos bits e faz duas chamadas recursivas. A função “radixSort()” simplesmente chama a função “quicksortB()” uma vez.

3 - Análise de Complexidade

- **QuickSort:** O tempo levado por este algoritmo depende da configuração do “array” de entrada, nos levando a três cenários possíveis:
  - Pior Caso: $O(n^2)$, que ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo.
  - Melhor Caso: $O(n \log(n))$, que ocorre quando cada partição divide o conjunto em duas partes iguais
  - Caso Médio: $O(n \log(n))$

- **MergeSort:** Temos que a função “merge()” têm um custo linear e chamamos ela 2 vezes para n/2 elementos. Logo, obtemos a função T(n) = 2T(n/2) + n, que equivale a um custo de O(n log n)

- **HeapSort:** O comportamento do Heapsort é sempre O(n log (n)), qualquer que seja a entrada. Sendo que a função “heapify()” tem custo O(log(n)) e a função “heapSort()”, por chamar a função “heapify()” para os nós internos, têm custo O(n log(n)).

- **Radix Exchange Sort:** O(n log(n))

4 - Configuração Experimental

- Foram propostas 4 configurações experimentais, sendo elas:

    - Configuração 1: ordenar os elementos do txt de entrada primeiro a partir do nome, utilizando o método QuickSort e, em seguida, ordenar pelos dados utilizando o HeapSort.

    - Configuração 2: ordenar a partir do nome, utilizando o método QuickSort e, em seguida, ordenar pelos dados utilizando o Radix Exchange Sort.

    - Configuração 3: ordenar a partir do nome, utilizando o método MergeSort e, em seguida, ordenar pelos dados utilizando o Heapsort.

    - Configuração 4: ordenar a partir do nome, utilizando o método MergeSort e, em seguida, ordenar pelos dados utilizando o Radix Exchange Sort.

- O objetivo desses experimentos foi fazer uma análise de tempo. Ou seja, observar quais métodos de ordenação que, quando aplicados em conjunto, são mais eficientes no arquivo de entrada em questão.

5 - Resultados

- O gráfico a seguir mostra o comportamento de cada uma das configurações propostas de acordo com o tamanho da entrada. É possível observar que, até 50000 entradas, o tempo de execução das configurações é bem similar, não havendo nenhum ganho de performance significativo. Mas após 100000 entradas, uma diferença mais evidente é percebida, pois a configuração 3 e 4 começam a ser executadas quase 1 segundo mais rápido do que as configurações 1 e 2.

- Isso ocorre pois nas configurações 1 e 2 o método QuickSort é utilizado e, como já foi visto, ele tem o pior caso em $O(n^2)$. Ou seja, apesar de ser o algoritmo mais eficiente que existe para uma grande variedade de situações, vimos que no caso em questão, para entradas grandes, o MergeSort acabou se saindo melhor. Portanto, a configuração mais eficiente foi a 4, que utiliza em conjunto os métodos MergeSort e Radix Exchange Sort.

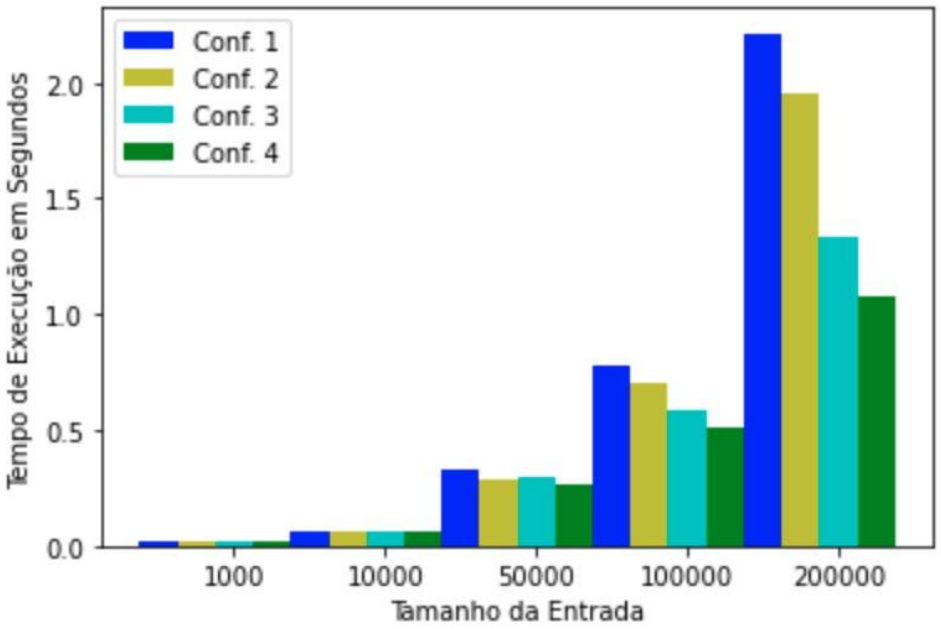
- Quanto a estabilidade das configurações, podemos destacar:

    - Configuração 1: Ao utilizar o método QuickSort e HeapSort nesta configuração, acabou por quebrar a estabilidade da ordenação, tendo em vista que ambos os métodos são instáveis.

    - Configuração 2: Pelo mesmo motivo da configuração 1, a estabilidade também é quebrada, já que o QuickSort é instável e a implementação do Radix Exchange Sort utilizada em aula também é instável.

    - Configuração 3: Ao utilizar o método MergeSort, primeiramente é garantida a estabilidade da ordenação, tendo em vista que este é um método estável. Contudo, em seguida, ordeno os dados utilizando o Heapsort, que é instável. Dessa forma, não há como garantir a estabilidade final dessa configuração.

    - Configuração 4: Como já foi dito, o método MergeSort e estável, então a estabilidade da primeira ordenação desta configuração é garantida. Logo em seguida, ao ordenar pelos dados utilizando o Radix Exchange Sort, que aqui foi implementado de forma instável, já não é mais garantida a estabilidade da configuração



6 - Conclusão

- Nesse trabalho, através da implementação de 4 algoritmos de ordenação distintos, ordenei um arquivo txt de acordo com 4 configurações de ordenação propostas.

- Foi possível observar que, com o arquivo de entrada especificado, quando o tamanho da entrada começa a ficar muito grande, algumas configurações se saíram melhores do que outras. Destaco aqui principalmente as configurações 1(QuickSort + HeapSort) e 4(MergeSort + Radix), pois estas foram as que apresentaram maior disparidade entre si.

- A configuração 4 deve ser considerada caso um alto custo de pior caso não possa ser tolerado, que é o que acontece com o QuickSort, que tem um custo de pior caso relativamente alto ($O(n^2)$).

7 - Instruções de Compilação

- Extraia a pasta zip para uma pasta de mesmo nome.
- Através do terminal, acesse a pasta extraída, em seguida a pasta tp e digite o comando make para executar o arquivo Makefile.
- A partir disso, acesse ./bin/run.out e insira o arquivo de entrada, seguido dos outros dois argumentos.
    - Ex: ./bin/run.out homologacao.txt 1 50000

8 - Bibliografia:

https://www.geeksforgeeks.org/merge-sort/
https://www.geeksforgeeks.org/quick-sort/
https://www.geeksforgeeks.org/heap-sort/
https://minerandodados.com.br/plotando-graficos-de-forma-facil-com-python/