Relatório - REO 3

Arthur Henrique Sousa Cruz

Universidade Federal de Lavras

1 Execução

No arquivo **main.py** configure a variável **tamanho_testes** para os valores que deseja (exatos 3 valores), definindo assim o tamanho das entradas. Devido ao custo computacional de execução do Bubble Sort, ele é executado para entradas de até 10000 vértices.

Um arquivo pode ser passado por linha de comando, permitindo que os resultados obtidos sejam salvos no arquivo. Os resultados serão exibidos na tela do terminal caso seja passado o parâmetro ou não. Para executar o código utilize o seguinte comando:

python3 main.py (arquivo-de-saida)

2 Organização de arquivos

Os arquivos deste projeto foram organizados da seguinte forma:

- **algoritmos_ordenacao**: Arquivo que implementa as funções de ordenação;
- **main.cpp**: Arquivo que gera e executa testes;
- **teste_envio.txt**: Arquivo com os resultados dos experimentos realizados;

3 Bubble Sort

O Bubble Sort gastou 5049 passos para a ordenação de um vetor de tamanho 100 e 500499 para tamanho 1000. O tempo gasto foi de $0,69$ e 69 milissegundos para os respectivos casos. Para 1000000 de vértices o programa foi executado por 10 minutos, mas o resultado não foi encontrado.

A quantidade de passos do Bubble Sort pareceu crescer de forma semelhante à complexidade do pior caso. Quando o vetor está ordenado de forma invertida, o algoritmo deve percorrer n^2 elementos para realizar a ordenação, ou seja, O pior caso do Bubble Sort é de $O(n^2)$. A complexidade de espaço por sua vez é de $O(n)$, visto que não há geração de nenhum novo vetor.

4 Merge Sort

O Merge Sort gastou 772 passos para um vetor com 100 elementos, 10976 para um com 1000 e 20951424 para um com 1000000. Seu tempo foi de $0,2$ e 3 milissegundos, para os dois primeiros casos (respectivamente), e $6,04$ segundos para o terceiro caso.

Assim como o Bubble Sort, as quantidades de passos representaram bem a complexidade do Merge Sort. O algoritmo, que é baseado em divisão e conquista, divide o vetor e recursivamente ordena os componentes gerados. A complexidade das divisões é $\theta(\log n)$, visto que ele divide o vetor até que a componente tenha tamanho 1 não importando o caso de entrada. A junção dos componentes também não varia para o caso e tem custo $\theta(n)$, já que deve percorrer as componentes por completo. A fase de junção é realizada uma vez para cada uma das divisões, logo, temos uma complexidade total de $\theta(n \log n)$. A complexidade de espaço do Merge Sort também é de $\theta(n \log n)$, já que para cada nível na árvore de recursão são criados novos vetores com soma de tamanhos igual a n.

5 Quick Sort

O Quick Sort gastou 781 passos para um vetor com 100 elementos, 9884 para um com 1000 e 22250821 para um com 1000000. Seu tempo foi de $0,1$ e 2 milissegundos, para os dois primeiros casos (respectivamente), e $4,07$ segundos para o terceiro caso.

Na implementação feita, o Quick Sort sempre escolhe como pivô o elemento do meio do vetor de entrada e seu pior caso ocorre quando os pivôs selecionados são sempre os maiores (ou menores) elementos do vetor. Isso ocorre pois, ao escolher o maior (ou menor) elemento do vetor como pivô, a divisão do vetor será desequilibrada, fazendo com que no máximo um elemento seja ordenado a cada passo da recursão. Isso faz com que a complexidade no pior caso do Quick Sort seja $O(n^2)$. Sua complexidade de espaço, assim como no Bubble Sort, é de $O(n)$, já que não são gerados novos vetores.

6 Comparação dos algoritmos

Pelos testes realizados é possível concluir que o Bubble Sort não é o mais adequado para vetores de tamanho 100 ou mais. Ele teve um custo muito maior em passos e em tempo, se comparado ao Quick Sort e ao Merge Sort. Isso fica explícito para com a entrada de 1000000 de vértices, já que ele não foi capaz de achar resultados em 10 minutos.

O Merge Sort e o Quick Sort, por outro lado, apresentaram valores mais próximos tanto em relação ao número de passos quanto em tempo. Apesar disso, mesmo com uma quantidade maior de passos em dois dos três testes, o Quick Sort teve tempo menor para todas as três entradas. Isso demonstra que a complexidade nem sempre dita o resultado. Mesmo sendo $O(n^2)$ para o pior caso, o Quick Sort obteve resultados melhores do que o Merge Sort. Porém, é de se

acreditar que no pior caso (para os mesmos tamanhos de entrada) o tempo do Quick Sort seria maior do que o do Merge.

Assim sendo, para uma aplicação que se conhece os dados e sabe-se que o pior caso do Quick Sort seria algo raro, sua utilização é a melhor opção. Para uma aplicação em que, para entradas de mesmo tamanho, a variação de tempo de ordenação deve ser mínima, seria mais interessante um algoritmo estável como o Merge Sort.