Trabalho Prático 2 - Biblioteca Digital de Arendelle

(Estudo Comparativo de Algoritmos de Ordenação)

Caio Guedes de Azevedo Mota

2018054990

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

caioguedes@ufmg.br

1. Introdução

O intuito deste trabalho prático é realizar uma comparação de eficiência entre versões de um mesmo algoritmo de ordenação, para mostrar o quanto diferentes variações de um mesmo algoritmo podem desempenhar melhor ou pior em certas situações do problema.

Especificamente, será examinado o desempenho de variações do *Quicksort*, um dos algoritmos mais populares de ordenação.

O *Quicksort* funciona da seguinte forma: dado um vetor a ser ordenado, o algoritmo escolhe um elemento desse vetor para ser um pivô. Após essa escolha, todos os elementos são comparados com o pivô no vetor. Caso o elemento seja menor, ele deve estar à direita do pivô, e caso maior, à esquerda. Então, o pivô fica no lugar e os dois subvetores à sua esquerda e direita são tornadas partições, e o algoritmo se repete em cada um dos subvetores, até o pivô ser o único elemento do subvetor. Ao final do processo recursivo de partições, os subvetores são juntados dois a dois, até formarem no final o vetor original, ordenado.

O *Quicksort* pode ser adaptado, para ter melhor desempenho frente a algum tipo de problema, por exemplo. As variações examinadas neste trabalho são:

1. ***Quicksort Clássico***: *quicksort* no qual o pivô é sempre o elemento central do vetor;
2. ***Quicksort Mediana de Três***: *quicksort* no qual o pivô será a mediana entre o elemento inicial, o final e o central do vetor;
3. ***Quicksort Primeiro Elemento***: *quicksort* no qual o pivô será sempre o primeiro elemento do vetor;
4. ***Quicksort Inserção 1%***: *quicksort* mediana de três, mas assim que a partição tiver menos de k = 1% elementos do tamanho original, é aplicado um *insertion sort* na partição;
5. ***Quicksort Inserção 5%***: mesmo que o anterior, porém k = 5%;
6. ***Quicksort Inserção 10%***: mesmo que o anterior, porém k = 10%;

7. ***Quicksort Não-Recursivo***: *quicksort* no qual não há chamadas recursivas, e os índices do vetor a particionar são guardados numa pilha implementada, simulando a pilha de recursão.

2. Implementação

A explicação da implementação será dividida em primeiro explicar os algoritmos de ordenação, e, depois, o funcionamento principal do programa.

Todas as implementações do *Quicksort* tem métodos de partição e métodos de ordenação. Os métodos de **partição** são da forma:

1. Seleção do pivô
2. Percorrer o vetor atual de baixo pra cima com um índice *i* e de cima para baixo com um índice *j*
3. Caso o elemento no dado índice *i* é menor do que o pivô, vai para o próximo *i*, e o mesmo vale para o elemento no índice *j* ser maior do que o pivô.
4. Caso não seja, troca o elemento do índice *i* maior que o pivô e o do índice *j* menor que o pivô de lugar.
5. Repetir até que os índices se cruzem
6. No final, retornar o índice *i*, que ao final será o índice do pivô escolhido.

A diferença entre os métodos de partição está na escolha do pivô. No *Quicksort Clássico*, é apenas o elemento obtido pelo índice no meio do vetor. No *Quicksort Mediana de Três*, é obtido esse mesmo elemento, mas é feita a mediana dele com o elemento inicial e o final. No *Quicksort Primeiro Elemento* é mais simples: é sempre o elemento inicial. Apenas esses três são os métodos de partição, enquanto os outros algoritmos utilizam desses em conjunto com outras estruturas.

Os métodos de **ordenação** para os *Quicksorts Clássico, Mediana de Três e Primeiro Elemento* são da forma:

1. Recebe o índice de início e fim do vetor a ordenar
2. Particiona e armazena o índice *q* do pivô após a partição
3. Caso o índice inicial *begin* seja menor que *q - 1*, ou seja, ainda há elementos entre o início e o pivô, chama o método de ordenação recursivamente com início = *begin* e fim = *q - 1*.
4. Caso o índice final *end* seja maior que *q*, ou seja, ainda há elementos entre o fim e o pivô, chama o método de ordenação recursivamente com início = *q* e fim = *end.*

A diferença do *Quicksort Inserção* está no método de ordenação dele, que contém um condicional antes dos passos acima. A ordenação dele recebe dois parâmetros a mais, o tamanho inicial do vetor e a porcentagem de corte desejada. O método sempre faz um cálculo para verificar a porcentagem da partição em relação ao tamanho inicial. Caso essa porcentagem seja igual à porcentagem de corte, é feito um *Insertion Sort* com a partição que sobrou. Dependendo, essa porcentagem de corte pode ser 1%, 5% ou 10% *(Observação: as partições feitas são todas com mediana de 3).*

O *Quicksort Iterativo* ou *Não-Recursivo* utiliza o método de partição do *Quicksort Clássico*, mas o seu método de ordenação é bem diferente do restante. É feita uma pilha (na forma de um vetor e um índice para o topo da pilha), em que são armazenados os índices das próximas partições a serem feitas. Há uma repetição central que repete até não houverem novos índices para partições na pilha, e dentro da repetição são feitas partições. O número de elementos entre o início da partição, o pivô e o final da partição é checado da mesma forma que os recursivos, mas os índices são armazenados na pilha ao invés de estarem nas novas chamadas.

Todos os métodos de partição contam com contadores globais que contam o número de comparações de elementos e o número de movimentações de elementos no vetor (considerando 2 movimentações por troca).

O **programa principal** realiza uma série de instruções, nessa ordem:

1. Lê os argumentos passados via linha de comando;
2. Estabelece o número de vetores para serem ordenados por execução (*n_testes*), e cria um vetor de tamanho *n_testes* para armazenar os tempos de execução;
3. Cria um número *n_testes* de vetores, que, dependendo do argumento passado, serão aleatórios, ordenados crescentemente ou ordenados decrescentemente;
4. Copia todos os vetores criados para uma matriz *arr_desord*, que armazena os vetores caso o parâmetro *-p* tenha sido passado;
5. Verifica o primeiro parâmetro passado para selecionar qual variação do *Quicksort* será utilizada;
6. Chama o método respectivo e ordena os vetores. Para cada ordenação, antes de ordenar um temporizador inicia e logo após ele para. Aí, o tempo de execução é armazenado no vetor criado anteriormente;
7. É rapidamente feito um *Quicksort Clássico* sobre os *n_testes* tempos de execução, para então escolher a mediana;
8. Após todos os testes, são impressos: o tipo de *Quicksort*, o tipo de vetor usado, o tamanho dos vetores, o número médio de comparações (total dividido por *n_testes*), o número médio de movimentações (total dividido por *n_testes*) e a mediana do tempo de execução;
9. Caso o parâmetro *-p* tenha sido passado, imprime os vetores desordenados armazenados anteriormente;
10. Deleta todos os vetores alocados.

O **formato de entrada e saída** dos dados se dá apenas pelo terminal, embora ao executar os testes modificações temporárias foram feitas para armazenar as saídas em um arquivo de texto. Tais detalhes serão mais explicados na análise experimental.

3. Instruções de Compilação e Execução

O programa é compilado usando o **g++** (compilador GNU para C++) e usando o padrão de linguagem **C++11**, devido à certas bibliotecas que utilizei para contar o tempo de ordenação dos algoritmos (*chrono*) serem incluídas apenas no C++11 ou superior.

Outro detalhe importante: devido à quantidade de chamadas recursivas para vetores muito grandes e casos ruins do problema, o tamanho da pilha de execução pode não ser o suficiente para executar certos testes. Para sanar isso, basta utilizar o comando `ulimit -s hard` antes de testar o programa para aumentar o tamanho da pilha.

Para a compilação do programa, deve-se utilizar o Makefile incluído com os arquivos do trabalho. Com o diretório principal aberto no terminal, digitando o comando `make` o programa será compilado e estará pronto para execução.

Para a execução do programa, o comando padrão é `./tp2 <Variação do Quicksort> <Variação do Vetor> <Tamanho do Vetor> [-p]`, sendo -p um parâmetro opcional.

4. Análise Experimental

Para cada uma das 7 variações do *Quicksort*, foram feitos testes com 3 tipos de vetores (vetor aleatório, vetor em ordem crescente e vetor em ordem decrescente), com 20 vetores de cada tipo, e com 10 tamanhos de vetor (de 50000 a 500000 elementos, com intervalos de +50000 entre tamanhos). Logo, foram feitos **4200 testes** no total. Os testes estão todos registrados no script **tester.bash**, que está junto dos arquivos do trabalho, caso necessária a referência.

Os testes foram feitos em uma máquina com as seguintes especificações técnicas:

- **Processador:** Intel Core i7-7500U @ 2.7 Ghz
- **Memória RAM:** 8GB
- **Memória Secundária:** SSD Kingston A400 120GB
- **Sistema Operacional:** Linux (distribuição: Ubuntu 19.04 Disco Dingo)

Todas as tabelas e gráficos analisados estarão na pasta **plots** incluída junto com o código e a documentação, com respectivos índices para consulta. Além disso, a pasta também contém uma tabela em *html* com todos os resultados de teste, e uma pasta *others* com outros gráficos gerados para comparação, mas não utilizados na análise. Os gráficos foram feitos com auxílio da linguagem Python e das ferramentas Pandas e Matplotlib, e o script gerador dos gráficos está no diretório raiz do trabalho, com o nome de *plot.py*.

4.1. Tempo de Ordenação

O primeiro gráfico (Gráfico 1) contém os dados de tempo de ordenação (mediana dos 20 testes) para vetores aleatórios. Pode-se perceber que, em geral, os

algoritmos sem a inserção parcial tiveram um desempenho superior aos com inserção parcial, com destaque ao com inserção de 5% e ao com inserção de 10%. Quanto maior é a porcentagem de uso do Insertion Sort, maior é o gargalo, já que o seu caso médio é pior do que o do Quicksort.

Entretanto, o cenário muda quando o problema é um vetor já ordenado (em ordem crescente ou decrescente, gráficos 2 e 3 respectivamente). Nesse caso os algoritmos com inserção passam a ser mais rápidos que os demais. Enquanto os Quicksort clássico, mediana de três e iterativo mostram desempenho semelhante ao anterior, os Quicksort com inserção são significativamente mais rápidos que esses, devido ao fato de ser uma operação linear (especificamente O(kn), sendo k o tamanho constante do subvetor da porcentagem desejada). Entretanto, tal comportamento é difícil observar pelos gráficos devido a outro cenário interessante: nesse caso, o tempo de ordenação do Quicksort de pivô sendo o primeiro elemento aumentou significativamente. Isso se deve ao fato de que o pivô sempre será o menor elemento, e isso sempre gerará o maior número possível de partições, já que, devido à ordenação, todos os elementos estarão sempre de um dos lados do primeiro (é o pior caso para essa versão do Quicksort).

4.2. Número de Comparações e Movimentações

Antes de dissertar sobre o número de comparações e movimentações, é importante deixar claro o que foi contado como comparação e movimentação relevante. Para as *comparações*, o número total de comparações foi incrementado em duas situações:

- **Comparação de elementos nos métodos de partição:** toda vez que um elemento do subvetor é comparado com o pivô, para decidir sua nova posição, o número total de comparações é incrementado em 1;
- **Comparação de elementos no método de inserção:** no *Insertion sort*, toda vez que um elemento é comparado a outro, o número total de comparações é incrementado em 1.

Já para as *movimentações*, o número total foi incrementado em outras duas situações:

- **Troca de elementos no vetor nos métodos de partição:** toda vez que dois elementos do vetor foram trocados, o número total de movimentações é incrementado em 2 (considerei apenas as movimentações que inserem elementos no vetor, logo elementos do vetor sendo escritos em variáveis auxiliares não foram consideradas);
- **Movimentação no método de inserção:** no *Insertion sort*, toda vez que um elemento é movido no vetor, o número total de movimentações é incrementado em 1.

Ao analisar o número médio de comparações e o número médio de movimentações com vetores aleatórios (Gráficos 4 e 7), percebe-se que todas as variações com inserção têm os números bem maiores que os números das variações sem

inserção (o que condiz com o caso do tempo de ordenação, já que quanto maior o número de operações, maior o tempo gasto). Isso se deve ao fato do processo de inserção ter o caso médio pior do que o processo de Quicksort. No caso de vetores aleatórios, os gráficos de tempo, comparações e movimentações são todos bem parecidos.

E, seguindo as tendências dos resultados de tempo, os números de *comparações* dos Quicksort com inserção diminuíram em relação aos outros ao falar de vetores ordenados (crescentes ou decrescentes, gráficos 5 e 6, respectivamente), assim como os números do Quicksort com pivô no primeiro elemento aumentam drasticamente (o que faz com que, no gráfico, seja difícil a visualização comparativa dos outros Quicksort).

Embora o número de comparações em vetores ordenados segue os outros gráficos, o número de *movimentações* muda um pouco (gráficos 8 e 9). Considerando duas movimentações por troca, o número de movimentações feitas pelos processos com inserção é menor. Isso é devido ao fato de que, ao inserir um elemento via inserção, o elemento não troca de lugar com os outros, mas é inserido apenas após todos os outros elementos já terem se movimentado, reduzindo o número total de movimentações.

5. Conclusão

Ao terminar a análise do experimento, é nítido o fato de que certas variações do Quicksort têm o desempenho melhor do que outras quando o problema é modelado de certa forma. Enquanto os Quicksort clássico, mediana de três e iterativo mantém um desempenho constante ao longo dos testes, o Quicksort com pivô no primeiro elemento é consideravelmente mais lento quando o vetor está ordenado (seja em ordem crescente ou decrescente). Já os Quicksort com inserção sofrem com vetores aleatórios, devido à lentidão do processo de inserção, mas têm vantagem com vetores ordenados.

A análise comparativa de algoritmos é uma parte importante ao decidir qual algoritmo utilizar, e entender qual é o problema que se deve resolver e qual o melhor algoritmo a aplicar nesse problema pode ser decisivo na hora de elaborar sistemas computacionais complexos.

6. Referências

Cormen, T., Leiserson, C., Rivest, R. e Stein, C. (2009) “Introduction to Algorithms - 3rd Edition”, The MIT Press, Cambridge, Massachusetts, Estados Unidos da América.

Chaimowicz, L. e Prates, R. “Ordenação: Quicksort”, pdf disponível no Moodle da turma de Estruturas de Dados 2019/1 da UFMG. acesso em: maio/2019.

Autor Desconhecido. “Quicksort”, https://www.geeksforgeeks.org/quick-sort/, acesso em maio/2019.

Mahapatra, S. “Measure execution time of a function in C++”, https://www.geeksforgeeks.org/measure-execution-time-function-cpp/, acesso em maio/2019.

Ahuja, K. “Command line arguments in C/C++”, https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/, acesso em maio/2019.

Barnwal, A. “Iterative Quicksort”, https://www.geeksforgeeks.org/iterative-quick-sort/, acesso em maio/19.