Trabalho Prático 1

Malditos Macacos!!

1. Introdução

O objetivo deste trabalho foi solucionar o problema de ordenação em uma quantidade de memória limitada dos arquivos de texto gerados pelos macacos de Mathias. As informações que foram fornecidas para a resolução do problema são a quantidade de memória disponível, o tamanho das linhas digitadas pelos macacos e os arquivos de entrada e saída. Uma solução válida se dá pelo resultado correto da ordenação do arquivo de entrada no arquivo de saída e pela alocação de uma quantidade de memória igual ou abaixo da requisitada.

2. Modelagem do problema

A solução dada utiliza a estratégia de intercalação balanceada de F caminhos para ordenar o arquivo de entrada no arquivo de saída. Através da execução do algoritmo, alcançamos a ordenação. Através da limitação dos tamanhos das estruturas de dados utilizadas no algoritmo, alcançamos a utilização igual ou abaixo da requisitada.

Para respeitar o limite de memória ao carregar linhas da memória externa, um cálculo é feito para estabelecer a quantidade M de linhas que podem estar simultaneamente na memória interna. M é calculado através da divisão do número de KB de memória disponíveis *memory* e do tamanho das linhas C que o arquivo de entrada contém, ambas informações disponíveis nas entradas do programa.

Para armazenar as linhas lidas da memória externa foi criado um vetor de strings, chamado de Buffer. O vetor tem M posições que guardam strings de C caracteres. Assim que o programa inicia a memória é alocada dinamicamente através da função *mathias_malloc*.

A principal estrutura utilizada é o Heap, que têm função de ordenar em memória interna linhas lidas dos arquivos utilizados no algoritmo. Esta estrutura é organizada como uma árvore binária balanceada, com a regra de que nós “pais” sempre serão menores do que nós “filhos”. Heaps que seguem esta regra são conhecidos como Min-Heaps.

O Heap foi implementado através de um vetor de números inteiros de tamanho M, e em toda a execução do algoritmo guarda índices das linhas no Buffer. O critério de ordenação se dá pela resposta da função *a_menor_que_b*.

A primeira etapa de execução do algoritmo desenvolvido realiza uma passada pelo arquivo de entrada para dividi-lo em blocos ordenados iniciais. O Heap é preenchido e balanceado inicialmente com as primeiras M linhas do arquivo de entrada. Depois é feita a retirada da linha L1 que “está” no topo do Heap, e a linha L1 é escrita no arquivo de saída. Uma nova linha L2 é lida e colocada no topo do Heap, que é balanceado através do rebaixamento

desta até seu local adequado. Caso a linha L2 seja menor que a linha L1, o índice da linha no buffer é guardado com seu valor negativo no Heap para indicar que esta linha pertence à outro bloco que não este que está sendo gerado no arquivo de saída. Quando um índice retirado do topo do heap é de sinal diferente, o bloco que estava sendo gerado é terminado e a produção de um novo bloco se inicia. O algoritmo segue com esta estratégia até que o conteúdo do arquivo de entrada termine, gerando B blocos no arquivo de saída.

A segunda etapa de execução do algoritmo consiste na intercalação dos B blocos gerados na primeira etapa, até que reste somente 1 bloco no arquivo de saída correspondendo ao conteúdo do arquivo de entrada ordenado. Primeiro são criados F arquivos auxiliares, nos quais serão copiados os B blocos gerados no arquivo de saída, de tal forma que no máximo haverá M arquivos auxiliares. Através da mesma estratégia de ordenação com Heap na primeira etapa, as primeiras M linhas são carregadas no Heap, a linha L1 em seu topo é retirada, sendo escrita no arquivo de saída, e uma nova linha L2 é lida do arquivo da qual a linha L1 foi retirada. Quando um bloco acaba, o arquivo fica “travado” esperando todos os outros blocos que estavam sendo intercalados serem terminados. A execução segue intercalando os blocos até que o conteúdo de todos os arquivos auxiliares seja lido, gerando uma quantidade menor que B de blocos é gerada no arquivo de saída.

3. Análise Teórica do Custo Assintótico

3.1. Análise Teórica do Custo Assintótico de Tempo

A análise de custo assintótico de tempo descreve a complexidade do algoritmo frente à entrada e a quantidade de operações de entrada/saída. O resultado é concluído a partir da análise separada das duas etapas seguintes.

3.1.1. Criação de blocos iniciais

Nesta etapa algoritmo realiza uma primeira passada para criar blocos ordenados iniciais. Utilizando a estrutura Heap a solução consegue gerar blocos de $2 \times M$ linhas, sendo M o tamanho máximo de linhas que podem ser armazenadas na memória interna, e analogamente o tamanho do Heap. Sendo B a quantidade de blocos gerados inicialmente por esta etapa:

$$B = ceil(N \div (2 \times M))$$

São feitas operações de leitura/escrita e inserção no Heap para cada uma das N linhas. Aqui consideramos o custo das operações de leitura/escrita como constantes. O custo para se inserir um nó no Heap é de $log(M)$, sendo M o tamanho máximo do Heap durante a execução. Assim, o custo assintótico para se fazer uma intercalação é da ordem de $O(N \times log(M))$.

3.1.2. Intercalação dos blocos

Nesta etapa o algoritmo intercala os blocos gerados inicialmente. A solução utilizou M como quantidade de blocos intercalados ao mesmo tempo, a fim de utilizar toda a memória disponível neste passo. Assim, à cada intercalação a quantidade de blocos B se torna B/M. Ao final da intercalação de todos os blocos, foram executadas $log_M(B)$ intercalações.

Em cada intercalação são feitas operações de leitura/escrita e inserção no Heap para cada uma das N linhas. Como visto na análise da etapa anterior, o custo assintótico destas operações é de $O(N \times log(M))$. Assim, a complexidade deste passo de intercalação dos blocos é da ordem de $O(log_M(B) \times N \times log(M))$.

3.1.3. Complexidade assintótica de tempo final

A complexidade prevalecente das duas etapas corresponde à da segunda etapa, tornando a complexidade final de tempo do programa é da ordem de $O(log_M(B) \times N \times log(M))$. Em relação à quantidade de passadas $P(N)$ no arquivo de entrada de tamanho N que o algoritmo realiza, podemos concluir a partir da análise da segunda etapa que a pode ser calculada por $P(N) = log_M(B)$.

3.2. Análise Teórica do Custo Assintótico de Espaço

Como foi dito na introdução, a quantidade de linhas M que podem ser guardadas na memória interna pode ser calculada através da divisão da quantidade de memória disponível *memory* pela da quantidade de caracteres C de cada linha no arquivo de entrada. Todas as estruturas alocadas dinamicamente têm como limite de tamanho M, logo a complexidade assintótica de espaço é da ordem de $O(M)$.

4. Análise do Experimento

O experimento feito utiliza os casos de teste fornecidos pelo monitor. Os dados de memória utilizada e tempo gasto para a execução foram coletados da saída do *script* "*run_tests.sh*". O experimento foi feito usando uma máquina com processador Intel Core™ i7-3630QM CPU 2.40GHz × 8 e memória RAM de 7,5 GiB.

As execuções foram feitas de tal forma que conseguiremos ver variações de N (número de linhas do arquivo de entrada) de 10 linhas até 640 linhas, C (número de caracteres de cada linha) de 10 a 80 caracteres e *memory* (quantidade de memória disponível) de 1KB até 1000KB. Tais variações influenciam o tempo de execução do programa, diminuindo ou aumentando de acordo com seus valores.

Para evitar valores que não representem a fielmente o tempo gasto foram feitas 5 execuções para cada caso de teste, sendo tirada a média dos 5 tempos para se chegar ao tempo gasto em cada ponto. Abaixo encontram-se os resultados das execuções:

Tempos de execução de acordo com a memória disponível

Legenda: C=10 C=20 C=40 C=80

Analisando os 4 gráficos sem entrar em detalhes das linhas podemos ver que quanto mais aumentamos a memória disponível, maior o tempo gasto. Quanto mais memória, mais linhas podemos colocar na memória interna, maior o valor de M. Isso confirma a influência do valor de M na complexidade tempo.

Para confirmar a influência do número de linhas N na execução do algoritmo, podemos enxergar no primeiro gráfico (1KB) que quanto maior o número de linhas (eixo x), maior o tempo de execução observado. Para 10KB e 100KB o mesmo comportamento pode ser observado, mas para 1000KB todas as linhas parece estar praticamente horizontais. Isso se deve ao fato de que M se torna tão grande neste caso que o tempo de construção das

estruturas é muito maior em relação ao tempo de execução do algoritmo. Assim, a influência de N no último caso não pode ser percebida.

Sobre a influência do número de caracteres C, para enxergarmos isso é preciso ver como as diferentes linhas do mesmo gráfico estão dispostas. Tome o primeiro gráfico que define 1KB como a quantidade de memória disponível. Para todas as linhas do gráfico temos a mesma quantidade de memória, mas o que está sendo variado de linha para linha é a quantidade C de caracteres. C entra como divisor da quantidade de memória disponível, e quanto maior seu valor menor será a memória interna disponível . Isso confirma a nossa análise teórica, uma vez que quanto menor o número de linhas que podem ser carregadas na memória, maior o tempo gasto.

5. Conclusão

Através dos casos de teste pode-se ver que há um problema diante da escolha da quantidade de memória disponível para o algoritmo. Quanto mais memória disponível, mais tempo será gasto para criar as estruturas necessárias. Assim a escolha deve estar de acordo com o tamanho do arquivo de entrada. A melhor escolha de memória do experimento foi de 1KB, pois gera cenário em que realmente não é possível colocar todas as linhas do arquivo de entrada na memória disponível e que o tempo de execução é em maior parte influenciado pela própria execução do algoritmo. Sobre o problema a ser resolvido, o algoritmo ordena as linhas com sucesso. Logo, Mathias poderá continuar em busca de realizar seus desejos com o “Infinite Monkey Theorem”.