



Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

ALGORITMOS E ESTRUTURAS DE DADOS

Relatório do Problema da 1ª Série de Problemas

Docente

Prof.^a Cátia Vaz

Alunos

36368	Mihail Cirja
43552	Samuel Costa

Abril 2018

Índice

Introdução	3
1 Descrição da Aplicação	3
2 <i>Abstract Data Types</i>	4
3 Operações.....	4
4 Avaliação Experimental	4

Introdução

Este relatório, referente ao problema do primeiro de três trabalhos práticos, foi elaborado no âmbito da unidade curricular de Algoritmos e Estruturas de Dados e acompanha a resolução de um problema. Este problema constituiu uma aplicação de técnicas de algoritmia, o uso amontoados binários e suas operações, que foram estudadas nas aulas, a medição do tempo de execução de um algoritmo através do custo assintótico, e a comparação de algoritmos com base no seu grau de crescimento. Um dos principais objetivos do presente trabalho é a implementação de algoritmos capazes de operar com grandes quantidades de dados, tendo em conta as limitações de memória e a otimização dos tempos de execução.

Ao longo do documento apresentam-se a descrição da aplicação desenvolvida, com base nos seus requisitos; a indicação do *Abstract Data Type* (ADT) usado; a explicação das operações suportadas e, por fim, notas sobre a avaliação experimental do trabalho realizado em que se justificam os tempos obtidos.

1 Descrição da Aplicação

A aplicação desenvolvida permite juntar ordenadamente dados provenientes de ficheiros ordenados lexicograficamente.

São analisados e guardados os argumentos de linha de comandos. São esperados os seguintes parâmetros de chamada:

- *i* – o índice da palavra na linha pelo qual as palavras do ficheiro estão ordenadas lexicograficamente;
- *output.txt* – o nome do ficheiro resultante pretendido;
- *f1.txt f2.txt ...* - os nomes dos ficheiros de entrada.

É alocado espaço para um array de *BufferedReader*. Cada um deles é respeitante a um dos ficheiros de entrada.

O algoritmo desenvolvido para dar resposta ao problema consiste em:

[1] Definir o objeto *HeapNode*, composto por três campos: uma dada linha de um dos ficheiros de entrada, a *i*-ésima palavra da linha e o número de ordem do ficheiro de origem;

[2] Alocar espaço para um Min-Heap de *HeapNodes* e preenchê-lo com elementos respeitantes à primeira linha de cada ficheiro, guardando *dim* a dimensão do Min-Heap;

[3] Organizar o array em Min-Heap;

[4] Remover a raiz do amontoado binário e combiná-la com o ficheiro resultado, e obter uma nova linha. Na solução apresentada, a leitura dos ficheiros é feita em paralelo.

[5] Se a linha obtida for NULL, i.e. chegou-se ao fim do ficheiro correspondente, significa que o *reader* relativo ao ficheiro está indisponível, tornando-se necessário substituir o *reader* indisponível por aquele na posição *dim-1* e recorrer ao algoritmo *minHeapify* para reorganizar o *Heap* a partir do nó raiz. Nesse caso, decrementar a dimensão do *Heap*, e obter novo elemento a partir de um *reader* disponível.

[6] Inserir novo elemento obtido do mesmo ficheiro do removido. Reorganizar o Min-Heap. Repetir [4] até todos os *Readers* estarem indisponíveis.

2 Abstract Data Types

Para dar resposta ao problema foi utilizado o ADT (*Abstract Data Type*) min-Heap, que foi implementado sobre um array. Foi selecionado este ADT, pois é possível a dado momento, pelas suas propriedades, obter o mínimo. Cada elemento do *heap* é um objecto da classe *HeapNode*, que é constituído por uma *String* que armazena uma linha de texto, por um inteiro, o número de ordem n do ficheiro fn de onde a palavra é proveniente, e uma *String* que armazena a palavra de índice i da linha de texto.

3 Operações

Oferece-se suporte às operações de inserção de elementos e remoção do máximo.

Dado que as operações **inserção e remoção** são executadas muitas vezes, tornou-se necessário reduzir ao mínimo a complexidade desta operação. Como a remoção do mínimo precede imediatamente a inserção de um novo elemento, e uma vez que o mínimo se encontra sempre na raiz do amontoado, retira-se a raiz $O(1)$, coloca-se o novo elemento no nó 0 do amontoado e aplica-se *minHeapify* à raiz do amontoado $O(\log_2 n)$. Desta forma, garante-se que a complexidade da operação não excede $O(1 + \log_2 n) = O(\log_2 n)$, que é a altura do amontoado binário.

A alternativa, com complexidade superior, envolveria realizar as duas operações em separado, ou seja, colocar a folha com a chave mais elevada na raiz, decrementar o número de nós e recorrer à operação *minHeapify* ($O(\log_2 n)$) para restaurar as propriedades do heap. Em seguida, coloca-se o novo elemento na folha de índice \dim do Min-Heap e recorre-se à operação *Decrease Key*, que tem custo $O(\log_2 n)$. Assim o custo desta operação seria $O(2\log_2 n)$.

É também suportada a operação de **junção**, com complexidade $O(\log_2 n)$. Por junção entende-se a remoção da raiz do *Heap*, a escrita dessa linha no ficheiro de resultados, a inserção de novo elemento no amontoado e a sua reestruturação.

4 Avaliação Experimental

A aplicação foi testada e os seus resultados apresentam conformidade com o esperado.

Seja m o número de ficheiros, e n o número médio de linhas em cada ficheiro. Por análise do código, constatou-se que o grau de crescimento do algoritmo é $O(m \times n \log_2 n)$, mas como, nos ensaios conduzidos, o número de ficheiros foi reduzido, ou seja, o tamanho da heap é reduzido, o algoritmo tende para $O(n \log_2 n)$.

Na tabela 1 apresentam-se os resultados dos testes efetuados. Os ensaios experimentais conduzidos não revelam que o grau de crescimento do algoritmo é $\log_2 n$, pois a cada duplicação do número de ficheiros, o tempo de execução também duplica, sugerindo que o grau de crescimento para o algoritmo é afinal $O(n \log_2 n)$.

nFiles	nLines	Time(s)	Uso Heap (mb)
3	1259526	0,326	1
12	1259526	0,305	3
24	1259526	0,295	7

Tabela 1 – resultados parciais dos testes efetuados

Se o grau de crescimento do algoritmo desenvolvido fosse $O(n)$, ter-se-ia que, se o tempo registado para o processamento de 3 ficheiros é 0,326s, então o tempo esperado de processamento para 12 ficheiros seria $0,326 \times 4 = 1,304s$. Registou-se um tempo de processamento efetivo de 83% face ao esperado.

Considere-se a tabela 2, que apresenta resultados experimentais para ensaios efetuados com numero diferente de ficheiros, e número invariante de linhas totais. Considera-se que o grau de crescimento é afetado por n , o número de ficheiros.

Amostras(#)	Tempo [ms]	Ficheiros
300	28	3
600	30	3
900	32	3
9000	48	3
30000	62	3
60000	95	3
90000	110	3
150000	128	3
300000	157	3
600000	223	3
1259538	360	3

Tabela 2 Resultados experimentais, dependentes do numero de linhas.

Observa-se que o tempo de execução do algoritmo praticamente não se altera, confirmando a o cálculo efetuado de $O(\log_2 n)$.

A Tabela 2 apresenta os resultados obtidos na sequencia do aumento das linhas totais. Neste caso o grau de crescimento é afetado pelo número total de linhas.

O gráfico seguinte apresenta a relação entre o numero de linhas totais e os tempos de execução:



Gráfico 1-Gráfico da evolução dos tempos de execução

Analisando o gráfico acima representado, podemos concluir que o custo assintótico do algoritmo implementado é aproximadamente linear, ou seja $O(n)$.

Dado o crescimento de utilização do Heap verificado, estima-se que, nas mesmas condições de utilização do Heap da JVM especificados (máx. 32mb), a aplicação desenvolvida suporte até aproximadamente 100 ficheiros. Adicionalmente nota-se que, pelas características da classe `BufferedWriter`, o tamanho máximo suportado para o ficheiro de resultados é 1Gb.