Algoritmos e Estruturas de Dados

1ª Série

Junção ordenada de ficheiros, sem repetições

Nº 50493 Bernardo Pereira

Nº 50512 António Paulino

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2022/2023

16/4/2023

Índice

1. Introdução

Pretende-se desenvolver uma aplicação que permita juntar de forma ordenada os dados provenientes de vários ficheiros, produzindo um novo ficheiro de texto ordenado de modo crescente e sem repetições. O ficheiro produzido contém uma palavra por linha como indicado no enunciado.

2. Junção ordenada de ficheiros

2.1 Análise do problema

O problema é constituído por um número finito de ficheiros, em que cada linha de um ficheiro tem uma palavra. Os ficheiros estão ordenados lexicograficamente por ordem crescente e o número total de palavras é m.

As operações necessárias são as seguintes:

1. Leitura dos ficheiros de input.
2. Criação de um ficheiro output onde se escreverão as palavras por ordem crescente e sem repetição
3. Filtragem das palavras repetidas (mantendo a ordem crescente das palavras)
4. Escrita das palavras

A abordagem implementada para as operações foi a seguinte:

1. É criado um array com tamanho n - 1 em que n é a quantidade de ficheiros (o primeiro ficheiro passado como input é o ficheiro onde é escrito o resultado), e com os elementos do array sendo os leitores dos ficheiros.

2. É criado um "escritor" para escrever o output do programa a desenvolver.

3. Para cada leitura de palavra, procura-se a menor palavra de todos os ficheiros. Assim, mantém-se a ordem crescente de palavras. Verifica-se se a palavra anteriormente escrita é igual à atual ou não e, se o for, não é escrita. Como as palavras de cada ficheiro estão ordenadas de ordem crescente lexicograficamente, se existirem palavras repetidas nos ficheiros, serão adjacentes. Finalmente, lê-se a próxima linha do ficheiro que tinha a menor palavra. Este processo repete-se até não serem encontradas mais linhas.
4. É escrita cada palavra lida depois de se verificar que não é repetida.

2.2 Estruturas de Dados

Neste trabalho utilizamos as seguintes estruturas de dados:

Array

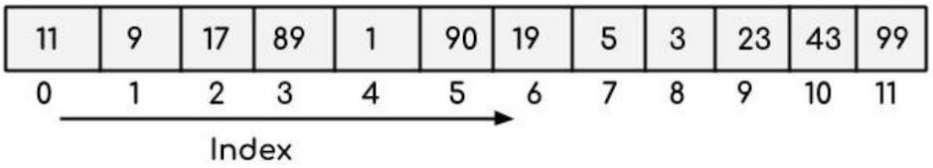

Figura 1: Exemplo da estrutura de dados *array*.

Lista

Figura 2: Exemplo da estrutura de dados Lista

Heap

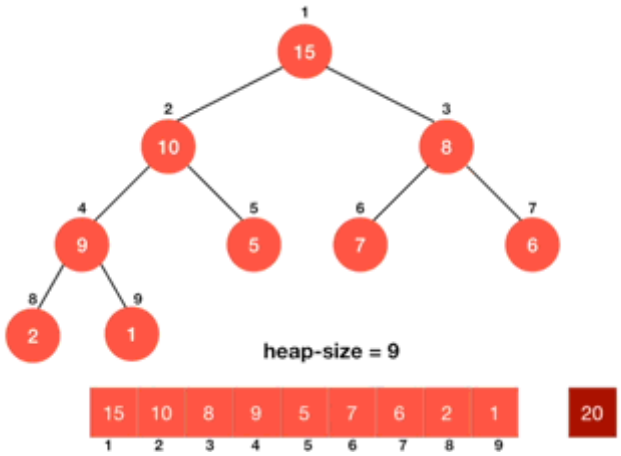

Figura 3: Exemplo da estrutura de dados Heap

2.3 Algoritmos e análise da complexidade

Principais algoritmos utilizados nesta implementação:

Priority Queue (Min-Heap), Filtragem de palavras com ordenação

A Priority Queue implementa os seguintes algoritmos/funções

cmpResult(Pair1 , Pair2)

retorna o valor da subtração entre Pair1.first e Pair2.first,

se Pair1.first é nulo, retorna -1

se Pair2.first é nulo, retorna 1

isEmpty(), $O(1)$:

Retorna true se a Priority Queue está vazia, se não, retorna false

Peek(), $O(1)$:

Retorna o elemento com menor prioridade na Priority Queue

Se a fila de prioridade estiver vazia, retorna nulo.

poll(), $O(\log(n))$:

Remove e retorna o elemento com a menor prioridade na fila de prioridade

Chama minHeapify (organiza o Min-Heap)

Se a fila de prioridade estiver vazia retorna nulo

minHeapify(i : Int), $O(\log(n))$:

Enquanto não for atingido o fim do heap, e um dos filhos do elemento do índice i for menor, troca-se o elemento do índice i

Com o menor dos dois filhos

Parent(i) , $O(1)$:

Retorna o indice do pai de i

Left(i), $O(1)$:

Retorna o índice do filho esquerdo de i

Right(i), $O(1)$:

Retorna o índice do filho direito de i

exchange(i, j), $O(1)$:

Troca os elementos nas posições i e j do heap

offer (element), $O(\log(n))$:

Insere um elemento na Priority Queue,

Chama decreaseKey (organiza o Min-Heap)

Retorna true se a inserção for bem sucedida, se não retorna false

decreaseKey(i), $O(\log(n))$:

Enquanto i não é o topo da priority queue e o pai do elemento no índice i é menor que o seu pai, trocam-se

Todas as funções realizam operações constantes, exceto as funções minHeapify e decreaseKey. As funções poll e offer chamam as funções minHeapify e decreaseKey respectivamente, e têm as suas complexidades temporais.

Análise da função decreaseKey:

Substituições sucessivas

$$T(n) = T\left(\frac{n}{2}\right) + O(1) =$$

$$= T\left(\frac{n}{4}\right) + 2\,O(1)$$

$$= T\left(\frac{n}{8}\right) + 3\,O(1)$$

$$= T\left(\frac{n}{2^k}\right) + K\,O(1), \qquad k = log2(n)$$

$$= T(1) + log2(n)\,O(1) =$$

$$= log(n)$$

Análise da função minHeapify:

Teorema mestre

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

$$a = 1, \qquad b = \frac{3}{2}, \qquad f(n) = O(1)$$

$$f(n) = O\left(n^{log3/2(1)}\right) = O(1) \rightarrow Caso\ 2$$

$$T(n) = O\left(n^{log3/2(1)} \log(n)\right) = \log(n)$$

Conclui-se que as funções minHeapify e decreaseKey ambas têm complexidade temporal O(log(n))
Como não são utilizadas estruturas de dados adicionais nestas funções, as suas complexidades espaciais são O (1)

Algoritmo de filtragem de palavras com ordenação (pseudo-código):

```
M = quantidade de ficheiros input – 1
Readers = Array com tamanho m, os elementos são os leitores de cada ficheiro
PriorityQueue = PriorityQueue(Array de pares com tamanho m, nulos inicialmente)
for (índice no Array de readers)
        PriorityQueue.offer(Pair(line, index))
 outputFile = Escritor do primeiro elemento de args
 lastline = “”
while(true)
        minLine, minIndex = PriorityQueue.poll()
        se (minLine é igual null) break
        se (minLine é differente de lastline)
                Escrever palavra no ficheiro de output
                lastline = minline
        PriorityQueue.offer(Pair(Próxima linha de readers[minIndex], minIndex))
Fechar ficheiro de escrita
Fechar ficheiros de leitura
```

Análise do algoritmo:

$$T(n) = O(1) + O(n) + O(n) + O(n) + n\,O(1) + O(1) + O(1) + O(m)\,(\,log(n)\;O(1) + O(1) + O(1) + O(1) + log(n)) + O(1) + O(1) =$$

$$= 4\,O(n) + 5\,O(1) + O(m\,log\,(n)) =$$

$$= O(m\,log(n))$$

Este algoritmo tem complexidade O(m log(n)), em que m é o número total de palavras e n é o número total de ficheiros. Isto porque o algoritmo percorre todas as m palavras passadas pelos ficheiros, e procura o ficheiro que contêm a menor palavra em cada iteração. Como a procura do ficheiro que contêm a menor palavra é feita através de uma priority queue, a complexidade dessa pesquisa é de log(n), se não fosse utilizada uma priority queue, a complexidade seria O(mn), já que teriam de ser percorridos todos os ficheiros para encontrar a menor palavra.

Em termos de complexidade espacial, este algoritmo é O(n), em que n é o número de ficheiros, isto porque é criado um array com n elementos, e uma priority queue com também n elementos.
Dado isto, a complexidade espacial deste algoritmo é de O(2n) = 2(O(n)) = O(n)

3 Avaliação Experimental

Para testar os algoritmos, serão utilizadas amostras de 1, 10, 20, 30, 50, 100, 200, 300, 500 e 1000 ficheiros, sendo que esses ficheiros serão os ficheiros f1, f2 e f3 colocados num array de forma aleatória.

Os testes serão realizados numa máquina com o processador Intel I7 12650h, 16 Gb ram 4800mhz

	Nº de ficheiros a ordenar									
	1	10	20	30	50	100	200	300	500	1000
JoinFiles1.kt	78.36	502.81	1044.53	1533.39	2712.85	6141.34	14114.12	22214.55	37684.27	89502.35
JoinFiles2.kt	80.10	450.90	1124.929	1603.18	2977.54	6550.27	14205.98	23506.47	41535.69	103347.4

Tabela 1: Resultados do tempo em milissegundos de execução de algoritmos de ordenação considerando várias amostras. Média de 5 valores por amostra

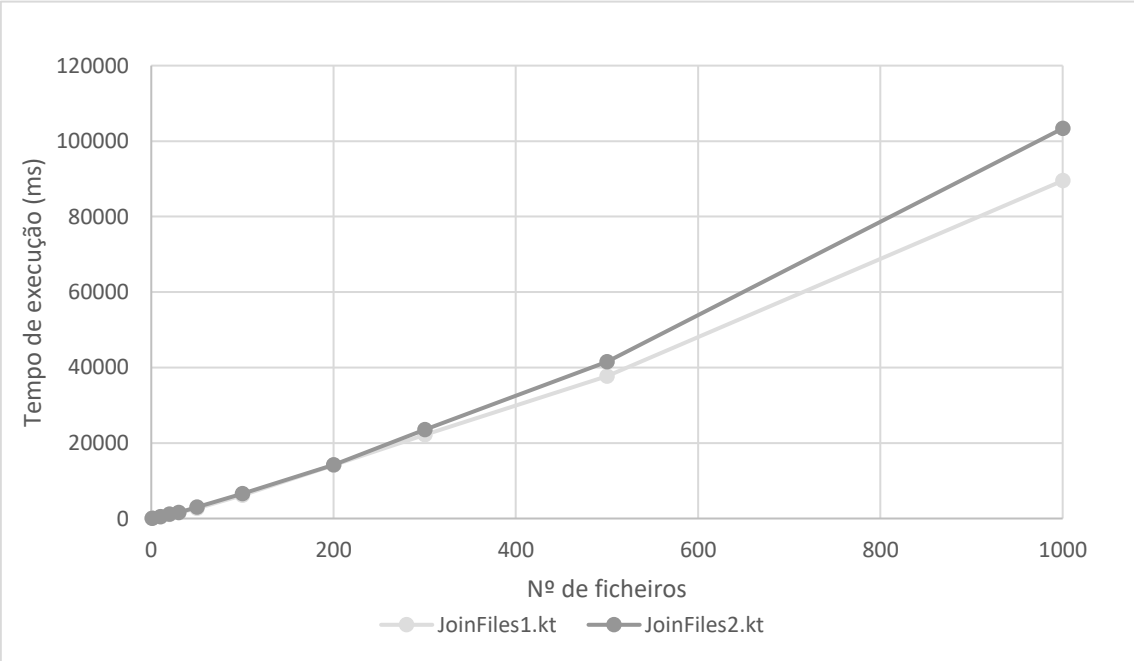

Figura 2: Comparação dos tempos de execução das duas implementações.

Como é possível observar no gráfico e na tabela 1, as duas implementações aparentam ter uma complexidade temporal linear / logarítmica linear, quando comparados com o gráfico da figura 4. Então verifica-se experimentalmente a análise dos algoritmos na secção 2.3.

Também é possível observar que a segunda implementação é ligeiramente mais lenta do que a primeira. Isto deve-se ao facto de que a primeira implementação utiliza a estrutura de dados array para armazenar os leitores, enquanto a segunda implementação utiliza a estrutura de dados list.

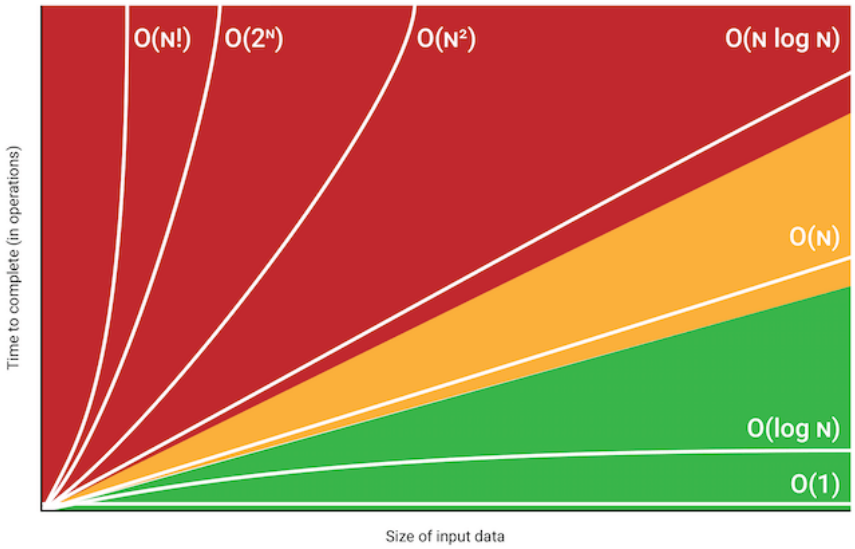

Figura 4: Gráfico de notação O-Grande

4 Conclusões

Neste trabalho foi implementado um algoritmo que dá "merge" a múltiplos ficheiros, criando um novo ficheiro com palavras por ordem crescente e não repetidas. Foi possível implementar um algoritmo com complexidade temporal O(m log(n)) e complexidade espacial O(n) através da utilização de priority queues. Uma das limitações deste algoritmo é que todos os ficheiros input têm que ter as palavras todas ordenadas lexicograficamente, se os ficheiros não estivessem ordenados, não seria possível através deste algoritmo criar um novo ficheiro ordenado e sem repetições. Finalmente, também se verificou através da comparação entre as duas implementações que a utilização da estrutura de dados Array torna o algoritmo mais rápido em relação a outras.