Algoritmos e Estruturas de Dados

1ª Série

(Problema)

Análise de ocorrência de menções em tweets

N. 50548 Nome: Gonçalo Amigo
N. 50518 Nome: Rita Monteiro
N. 50500 Nome: Humberto Carvalho

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2022/2023

09/04/2024

Índice

1. Introdução

O projeto a implementar baseia-se no desenvolvimento duma aplicação na linguagem *Kotlin* que analise os *Tweets* do mundial de futebol de Brasil de 2014, provenientes da aplicação X (antigamente denominada por *Twitter*), de modo a se obter os tweets mais mencionados e os k-tweets mais próximos duma certa instância.
O relatório divide-se em diversas seções, todas com o propósito de dar um maior entendimento ao utilizador da aplicação desenvolvida. Sendo essas seções:

- Descrição do problema em mãos
- Análise de cada método
- Quais as estruturas de dados mais adequadas para a implementação dos métodos
- A complexidade em termos de tempo de execução e em espaço
- Análise experimental dos algoritmos implementados com testes
- Conclusão

2. Análise de ocorrência de menções em tweets

Foi proposto o desenvolvimento de uma aplicação que permita analisar as menções em *tweets* relacionadas com o mundial de futebol do Brasil em 2014. Cada *tweet* é composto por certas propriedades tais como os *hashtags,* a data de criação do *tweet*, a identificação do utilizador e do *tweet* em si.

De forma a analisar os tweets, foram implementadas duas abordagens: i) uma pesquisa sobre os *hashtags* mais mencionados; ii) os *tweets* com uma data mais próxima duma data específica.

Na secção 2.1, será explicada a abordagem encontrada para resolver o problema. Na secção 2.2, após explicação da abordagem tomada, quais as estruturas de dados utilizadas e para finalizar, na secção 2.3, serão apresentados os algoritmos utilizados, de forma também a analisar a complexidade do algoritmo.

2.1 Análise do problema

Como já foi referido no início da secção 2, foi implementado uma aplicação que permite analisar um conjunto de *tweets*.

A primeira abordagem a ser implementada é a verificação de ocorrências de *hashtags* num ficheiro específico. Para implementação deste método é necessário ter um conta alguns aspetos: i) leitura do ficheiro; ii) tratamento do array de strings proveniente dessa leitura; iii) a partir desse tratamento, verificar o número de ocorrências; iv) apresentar qual o *hashtag* mais mencionado e o número de vezes que foi mencionado.

Para a leitura do ficheiro que contém as hashtags que o utilizador quer procurar, foi utilizado um *import* da linguagem *Java* (*import java.io.File*) que permite ler o ficheiro pretendido, e colocá-lo numa *string.* Para o tratamento da leitura do ficheiro que tem todos os Tweets, foi necessário iterar sobre o ficheiro, e após obter um array de Tweets, o mesmo foi filtrado de modo a apenas apresentar os hashtags de cada Tweet, e através

de certas funções, foi possível retirar o *array* desejado. Após o tratamento necessário desse array, o mesmo é passado ao método implementado. Na função, para o tratamento inicialmente, é feita a comparação entre os dois arrays (o *array* passado em parâmetro e o *array* proveniente do ficheiro *hashtag.tag*) de modo a filtrar apenas os hashtags pretendidos pelo utilizador, logo a informação de todos os *hashtags* que estavam no ficheiro estão neste momento numa estrutura de dados que facilita a iteração sobre a informação pretendida. Através desta estrutura, é efetuado um ciclo de modo a contar as ocorrências de cada *hashtag*. Tendo o valor de ocorrências, é preciso uma estrutura de dados que permita identificar qual o mais mencionado (isto é, aquele ao qual a variável das ocorrências teve o maior valor inteiro), logo é colocada a informação das ocorrências num *Max Heap* (isto é, uma árvore binária quase completa, que coloca de modo decrescente da raiz ao topo, os valores das ocorrências). Desta forma, o valor pretendido será o valor da raíz do Max Heap, pois é o valor mais elevado.

A segunda abordagem a ser desenvolvida permite analisar as datas mais próximas de uma dataString que é passada no parâmetro da função, o qual chamamos de nearestTweetR(). Para implementação deste método é necessário ter um conta alguns aspetos: i) leitura das datas do ficheiro; ii) Cálculo das diferenças de tempo; iii) Ordenação com um min-heap; iv) Retorno dos k Tweets mais próximos da dataString.
Para a leitura das datas do ficheiro foi utilizado a função dateStringToSeconds(dateString: String): Long, que permite, tal como o nome indica, converter uma data em formato String para um valor do tipo Long. De seguida é calculado a diferença entre o valor de tempo de cada tweet com a dataString, com isto obtemos o resultado das diferenças entre a dataString com todas as datas dos Tweets num array. De seguida é ordenado o array utilizando um min heap, para que quando maior os índices menor será a diferença entre cada tweet com a dataString. Consequentemente é retornado os valores dos índices do array de 0 a k-1.

2.2 Estruturas de Dados

As estruturas de dados utilizadas na implementação do primeiro método foram o *Array* e uma Árvore Binária, mais precisamente, um *Max Heap.*

Arrays, são estruturas de dados muito utilizadas pela sua versatilidade em relação às listas na linguagem *Kotlin.* Um array permite alterar certos valores do mesmo, mantendo um tamanho fixo. Já as listas imutáveis não permitem alterar diretamente o seu valor. Nas listas, caso se queira alterar algum valor, terá que ser criada uma nova instância dessa mesma lista.

Logo a escolha desta estrutura de dados permite um mais fácil acesso e alteração de dados do que uma lista imutável. Na figura 1, está ilustrado um exemplo de um *array* de *strings* após ler o ficheiro *hashtag.tag*.

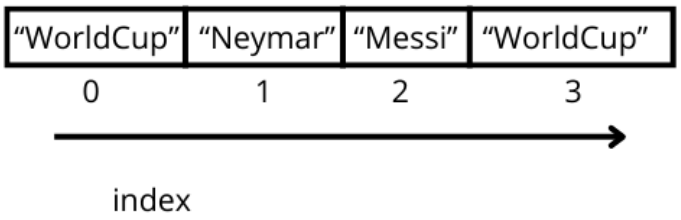

Figura 1: Exemplo da estrutura de dados *array* de acordo com o primeiro método.

Heaps são estruturas de dados hierárquicas, uma vez que os mesmos têm vários níveis e nós, ao qual os nós inferiores têm valores menores que o nós superiores.

A escolha para a utilização desta estrutura baseia-se na facilidade de organização dos dados uma vez que, a mesma, através dum *array* cria uma árvore de acordo com uma chave. Cada nó tem uma identificação, que é a sua chave.

No primeiro método é usado um *Max Heap* para este mesmo propósito, pois através de um valor chave, que neste caso será a contagem de ocorrências.

Na Figura 2, encontra-se um exemplo do Max Heap que tem as ocorrências de acordo com o exemplo dado na Figura 1.

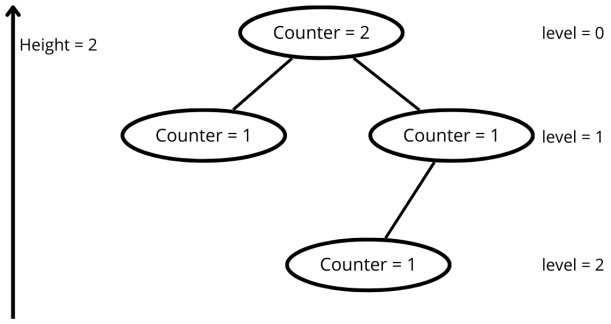

Figura 2 - exemplo do Max Heap de acordo o array da Figura 1

2.3 Algoritmos e análise da complexidade

2.3.1 *moreMentioned*

Na implementação do método *moreMentioned* (analisa as menções dos hashtags), a parte mais complexa poderá estar em torno da contagem das ocorrências e da construção do MaxHeap.

```
                    while (j <= size - 1) {
                      if (j == size - 1) {
    if (hashToPickedHash[i] == hashToPickedHash[j]) counter++
            counterArr += counter
              mostMentionedHash = hashToPickedHash[i]
            counter = 1
             i++
             j = i + 1
}
    if (hashToPickedHash[i] == hashToPickedHash[j]) counter++
             j++
  if (i == size - 1) break
}
```

Figura 3 - Código para a implementação da contagem das ocorrências

Como pode ser evidenciado na Figura 3, de modo a obter as ocorrências de um certo *hashtag,* foi necessário iterar sobre o array hashToPickedHash (array com os hashtags).

Caso a variável j, tenha o mesmo valor que size - 1 (*hashToPickedHash.size* - 1) será verificado se esse valor final do array é um dos hashtags pretendidos. Caso o seja, irá incrementar o counter (variável que atualiza o valor das ocorrências a cada passagem pelo array), esse mesmo valor adiciona ao array counterArr() (array que irá conter todas as ocorrências), reinicia o valor de counter para 1, incrementa a variável i e atualiza a variável j.
Caso a variável j, não seja o índice final, será feito um incremento do counter apenas, pois poderá haver mais ocorrências desse hashtag no array. Posteriormente incrementa a variável j.
Caso a variável i fique com o valor do índice final do array, significa que já foi feita a iteração sobre todos os hashtags pretendidos pelo utilizador, e que não necessita de iterar mais sobre o array, logo saí do ciclo.

```
buildMaxHeap(counterArr, counterArr.size, compare)

fun buildMaxHeap(h: IntArray, n: Int, cmp:(Int, Int) -> Int) {

for (i in n / 2 - 1 downTo 0)

maxHeapify(h, i, n, cmp)

}

val compare = {a: Int, b: Int -> a - b}
```

Figura 4 - Código para a implementação do *MaxHeap*

Na Figura 4, encontra-se o código para a implementação do *MaxHeap* utilizado para transformar o array das ocorrências numa árvore binária de ordem decrescente da raiz às folhas.
Na função *moreMentioned*() é feita a chamada à função buildMaxHeap que recebe um *IntArray*, que caso presente é o *counterArr*(), a dimensão do array e uma função de comparação de modo a comparar as repetitivas ocorrências.
Na implementação da função buildMaxHeap é feita a chamada à função maxHeapify que organiza o heap de modo obedecer à definição do MaxHeap, pois da forma como o array está estruturado, poderá não estar de ordem decrescente, logo o maxHeapify trata disso comparado cada nó da árvore.

Num todo, estes são os algoritmos mais complexos dentro da primeira pergunta do problema em mãos.

No cálculo abaixo encontra-se a complexidade desta primeira função:

$$C(n) = O(1) + O(n) + O(n) + O(n) = O(n)$$

A função tem uma complexidade linear, o que classifica a função como eficiente, logo para elevados valores, a função *moreMentioned*() tem um comportamento bom, a nível de execução. Em questão de complexidade de espaço, é também linear ($O(n)$) visto que a dimensão do array criado quando é efetuado o tratamento dos hashtags é de N elementos.

2.3.2 *nearestTweetR*

Na implementação da função *nearestTweetR*(), recebe uma lista de tweets, um número inteiro k e uma data em formato de string. Ela calcula as diferenças de tempo entre cada tweet e a data fornecida, cria um array de pares (ID, Tempo), constrói um heap mínimo, ordena o array utilizando o *heap sort* e, finalmente, retorna os k tweets mais próximos da data fornecida.

- ➔ Descrição dos Algoritmos Utilizados:
  - ◆ Cálculo das Diferenças de Tempo:
    - Algoritmo: Iteração sobre a lista de tweets para calcular a diferença de tempo entre cada tweet e a dateString
    - Complexidade Temporal: O(n), pois armazena-se a diferença de tempo para cada tweet em um array.
    - Complexidade Espacial: O(n), pois armazena-se a diferença de tempo para cada tweet em um array.
  - ◆ Construção do Heap Mínimo:
    - Algoritmo: Utilização do algoritmo *buildMinHeapL* para construir um heap mínimo.
    - Complexidade Temporal: O(n), onde n é o número de elementos no array.
    - Complexidade Espacial: O(1), pois a construção é realizada in-place, sem consumo adicional de memória.
  - ◆ Ordenação do Array usando *Heap Sort*:
    - Algoritmo: Utilização de um loop para repetidamente trocar o primeiro elemento do heap com o último elemento não classificado do array e, em seguida, restaurar a propriedade de heap usando *minHeapifyL*.
    - Complexidade Temporal: O(n log n), onde n é o número de elementos no array.
    - Complexidade Espacial: O(1), pois a ordenação é realizada in-place.

- Seleção dos k Elementos Mais Próximos:
  - Algoritmo: Utilização do método take para selecionar os k primeiros elementos do array.
  - Complexidade Temporal: O(k log n), onde k é o número de elementos a serem selecionados e n é o número total de elementos no array.
  - Complexidade Espacial: O(k), pois armazena-se apenas os k elementos selecionados.

➔ Consulta/Atualização das Estruturas de Dados:

- A lista de *tweets* é consultada para calcular as diferenças de tempo entre cada *tweet* e a *dateString*.
- As estruturas de pares (ID, Tempo) são atualizadas durante o cálculo das diferenças de tempo e ao ajustar as diferenças para garantir que sejam todas positivas.
- O heap mínimo é construído com base no *array* de pares (ID, Tempo) e é utilizado para ordenar o array.
- O *array* ordenado é consultado para selecionar os k tweets mais próximos da dateString.

3. Avaliação Experimental

De acordo com os cálculos efetuados, a complexidade da função *moreMentioned()* é linear, que é apresentada pelo gráfico da Figura 5. Em termos de avaliação de resultados, como o algoritmo é linear, os tempos de execução são consideravelmente bons, estando a maioria dos tempos na gama dos milissegundos.

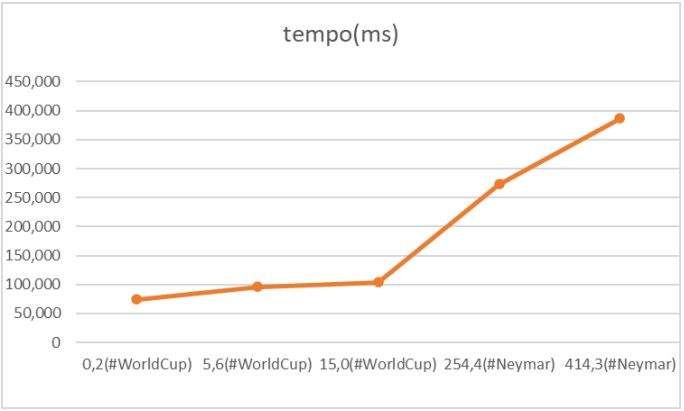

Figura 5 - Complexidade calculada da função *moreMentioned*()

Para a avaliação experimental da função *moreMentioned(),* para um ficheiro com três tweets, o algoritmo é significativamente rápido, mas para o ficheiro de um milhão de tweets, o algoritmo demora o seu tempo, isto porque a implementação feita para o tratamento de cada Tweet, acaba por ter uma complexidade de $O(n^2)$, logo tem um comportamento quadrático.

Com base nos cálculos realizados, a função nearestTweetR apresenta uma complexidade O(k log n), conforme indicado pelo gráfico. Isso significa que, em geral, os tempos de execução são consideravelmente bons, permanecendo na faixa dos segundos na maioria dos casos.

A avaliação dos resultados mostra que a eficiência do algoritmo se traduz em tempos de execução consistentemente rápidos, mesmo para conjuntos de dados maiores. Isso garante uma resposta rápida da função, especialmente em cenários onde a análise em tempo real de tweets é essencial.

4. Conclusões

Em suma, com o desenvolvimento deste trabalho foi possível adquirir competências no âmbito de diferentes estruturas de dados, principalmente nos Heaps e uma vertente dos mesmos, Priority Queues, de modo a obter os resultados esperados, tanto dos hashtags mais mencionados e dos k-elementos mais próximos duma certa data, como também permitir ao programador uma ágil implementação dos algoritmos de forma a que os mesmos tenham um tempo de execução linear ou quase linear.

Em termos gerais, como grupo, o trabalho desenvolvido foi desafiador pois foi colocado em prática novas estruturas de dados que previamente não teriam sido utilizadas no âmbito de outras disciplinas e diferentes maneiras de pensar uma vez que neste momento, como programadores, temos de pensar numa implementação que faça o pretendido mas não só, que também seja eficiente e que utilize menos recursos.