Universidad Veracruzana

Inteligencia Artificial

Análisis de Algoritmos

Actividad 6

Leonardo Flores Torres

17 de diciembre de 2022

Programar el algoritmo de Horspool para la búsqueda de palabras clave para los siguientes casos:

- Caso 1:
  - Entrada: Palabra de búsqueda, y archivo de texto `.txt`
  - Salida: No se encuentra la palabra, o bien sí se encuentra y la posición donde comienza la palabra (renglón y columna).
- Caso 2:
  - Entrada: Palabra de búsqueda y múltiples archivos de texto `.txt` (en una carpeta).
  - Salida: La lista de los archivos ordenados por relevancia (de mayor a menor). Esto es, cúantas veces se encuentra la palabra en cada archivo, y si dos archivos tienen el mismo conteo entonces será más relevante aquel en el la razón de las ocurrencias de la palabra de búsqueda respecto al total de palabras del documento sea mayor.

    Solución:

    El algoritmo de Horspool, también conocido como el algoritmo Boyer-Moore-Horspool, fue hecho para la búsqueda de palabras dentro de texto. Esto puede verse como la búsqueda de una subcadena de texto dentro de una cadena de entrada. Ejemplos del uso de este algoritmo se muestran en el curso *Data structures and Algorithms* [1] y en las notas de clase basadas en el libro *Análisis de algoritmos* [2].

    Este algoritmo busca en una cadena un patrón deseado, se va recorriendo la cadena hasta que se encuentra la primer letra del patrón y se guarda su posición en una tabla de ocurrencias, si el siguiente carácter de la cadena también coincide con el siguiente de la palabra buscada se vuelve a guardar su posición dentro de la tabla, y esto se repite hasta que todos los caracteres de la palabra clave son encontrados. Pero se deben realizar un par de consideraciones.

    Para realizar esta búsqueda se sobreponen los caracteres s_i de la cadena con aquellos del patrón c_j,

    $$\begin{array}{ccccccc} s_0 & s_1 & s_2 & \dots & s_n \\ c_0 & c_1 & c_2 & & \end{array},$$

    si hay un carácter de la cadena s_i que no esté dentro del patrón se mueve el patrón un número de espacios igual a su longitud y se realiza nuevamente la búsqueda, por ejemplo si $s_0 = c_0$, $s_2 = c_2$

pero $s_1 \neq c_1$,

$$\begin{array}{cccccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & \dots & & s_n \\ & & & c_0 & c_1 & c_2 & & & \end{array}.$$

El segundo caso es si un carácter s_i de la cadena se encuentra en el patrón pero no coincide con la posición actual en el patrón, entonces se mueve la cadena para hacerlos coincidir

$$\begin{array}{cccccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & \dots & & s_n \\ & & c_0 & c_1 & c_2 & & & & \end{array},$$

por ejemplo, si $s_4 \neq c_2$ pero $s_4 = c_0$,

$$\begin{array}{cccccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & \dots & & s_n \\ & & & & c_0 & c_1 & c_2 & \dots & & \end{array}.$$

El tercer caso es similar al primero, el último carácter del patrón coincide con su carácter en la cadena pero no hay coincidencias con el resto de caracteres del patrón,

$$\begin{array}{ccccccc} s_0 & s_1 & s_2 & \dots & & s_n \\ c_0 & c_1 & c_2 & & & \end{array},$$

esto es, si $s_2 = c_2$ pero $s_0 \neq c_0$ y $s_1 \neq c_1$, entonces la cadena se desplaza por la longitud total del patrón como en el primer caso,

$$\begin{array}{cccccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & \dots & & s_n \\ & & & c_0 & c_1 & c_2 & & & \end{array}.$$

El último caso es cuando el último carácter del patrón coincide con su respectivo carácter en la cadena pero hay otras coincidencias en el resto de caracteres

$$\begin{array}{cccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & \dots & & s_n \\ & c_0 & c_1 & c_2 & c_3 & & & \end{array},$$

por ejemplo, si $s_4 = c_3$ pero $s_3 \neq c_2$ y $s_4 = c_1$, $s_3 = c_0$,

$$\begin{array}{cccccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & \dots & & s_n \\ & & & c_0 & c_1 & c_2 & c_3 & & & \end{array},$$

entonces el desplazamiento de la cadena es similar al segundo caso.

Primero se carga el módulo `MiniFinder` en `julia` usando un alias para llamar más rápidamente a las funciones contenidas dentro de este. La búsqueda se hace mediante la función `horspool` que detecta solamente la primera ocurrencia del patrón en una cadena como se muestra a continuación:

```
julia> using Revise; using MiniFinder; mf = MiniFinder;

julia> text = "El curso de analisis de algoritmos es interesante. Leonardo estudia para su
↪  curso."
"El curso de analisis de algoritmos es interesante. Leonardo estudia para su curso."

julia> pattern = "leo"
"leo"
```

```
julia> mf.horspool(lowercase(text), pattern)
51

julia> pattern = "curso"
"curso"

julia> mf.horspool(lowercase(text), pattern)
3
```

En el caso en que el patrón es igual a `"curso"`, el algoritmo de Horspool no detecta la segunda ocurrencia. Finalmente lo que se desea para esta actividad es justamente esto, el poder contar cuantas ocurrencias hay del patrón en un texto, aunque para ello primero hay que diseñar una función que nos permita hacer esto en una línea de texto ya que el leer un archivo de texto `.txt` normalmente se hace por líneas. Esto se logró hacer mediante la función `occurrencesInLine` la cual recibe dos argumentos, el primero siendo una cadena de texto que representa una línea leída de alguna fuente y el segundo argumento el patrón de búsqueda.

Tomando el mismo texto guardado en la variable `text` definida anteriormente y realizando la búsqueda de los mismos patrones se obtiene lo siguiente:

```
julia> pattern = "leo"
"leo"

julia> occurrences = mf.occurrencesInLine(lowercase(text), pattern)
1-element Vector{Any}:
52

julia> text[occurrences[1]:end]
"Leonardo estudia para su curso."

julia> pattern = "curso"
"curso"

julia> occurrences = mf.occurrencesInLine(lowercase(text), pattern)
2-element Vector{Any}:
4
77

julia> text[occurrences[1]:end]
"curso de analisis de algoritmos es interesante. Leonardo estudia para su curso."

julia> text[occurrences[2]:end]
"curso."
```

El objetivo es poder observar como se realiza la detección de las ocurrencias. Por ejemplo, si se tiene una cadena de longitud n_c, siendo s_i los caracteres que conforman esa cadena, y se busca un patrón de longitud n_p, asumiendo que se encuentra la primera ocurrencia del patrón en la posición s_k donde $0 \leq k \leq n_c - n_p$, esto no asegura no haya otra ocurrencia del patrón en el resto de caracteres desde s_{k+1} hasta $s_{n_c - n_p}$. Se debe escribir una función que tome en cuenta lo ya mencionado y que aplique nuevamente la búsqueda del patrón en el resto de la cadena manteniendo al mismo tiempo

la posición relativa de la nueva cadena respecto a la cadena original.

cadena original	s_0	s_1	s_2	s_3	s_4	s_5	$\dots$	s_n
cadena nueva				s'_0	s'_1	s'_2	$\dots$	s'_n
índices	0	1	2	3	4	5	$\dots$	n

La función que se encarga de esto es `occurrencesInLines`, como ya se habia mostrado anteriormente. De manera similar, al ser ahora posible leer una línea se puede extender este concepto a leer un grupo de líneas, lo que es más, un grupo de líneas es un archivo de texto `.txt`. La función `occurrencesInFile` lee un archivo en este formato en un directorio indicado. Lo hace aplicando la función `occurrencesInLine` y guarda el índice de la línea en la que estaba y su ocurrencia en la línea actual en forma de tuplas $(\text{line}_i, \text{position}_j)$. Estas tuplas se van juntando en una lista donde se colectan todas las ocurrencias detectadas, y en el caso en que no se encuentre ninguna ocurrencia del patrón en el archivo indicado, la función imprime un mensaje de no haber encontrado ninguna y regresa una lista vacía. Ejemplo de esto se puede ver con un archivo de prueba, como se muestra a continuación:

```
julia> occurrences = mf.occurrencesInFile("../files/notabook_paragraphs_03.txt", "sed");

julia> occurrences
10-element Vector{Any}:
(1, 58)
(1, 168)
(1, 243)
(1, 383)
(3, 169)
(3, 243)
(3, 338)
(3, 952)
(3, 999)
(5, 307)

julia> occurrences = mf.occurrencesInFile("../files/notabook_paragraphs_03.txt", "desk");
No occurrences of "desk" where detected in "../files/notabook_paragraphs_03.txt".

julia> occurrences
Any[]
```

Los archivos de prueba para esta actividad se obtuvieron de forma aleatoria del sitio Lorem Ipsum[1] en el que se pueden generar textos con longitud arbitraria. Se decidió buscar los textos de esta manera para evitar complicar la limpieza de textos `.pdf` convertidos a `.txt`, además de que el propósito de esta actividad es solamente implementar la búsqueda.

Para completar el último punto de la actividad se tienen que contar de alguna manera las palabras que existen en un archivo y asi tener la densidad de palabras. Las funciones mencionadas hasta ahora están todas relacionadas a la detección de las ocurrencias de una sola palabra usada como patrón dentro de una cadena o conjunto de cadenas, no para contar cuáles son las palabras en un archivo ni su frecuencia. Se buscó un acercamiento mas *naive* para resolver el problema del conteo.

[1] `https://loremipsum.io/`

Para contar palabras se lee un archivo por líneas, cada línea es a su vez dividida por palabras a partir de la detección de ciertos símbolos de puntuación comunes en inglés como lo son `" .,:;-'!?"`, además de que todas las letras en mayúsculas son convertidas a minúsculas. Al momento de leer una palabra ésta se agrega a un diccionario que mantiene la cuenta de esa palabra mientras detecta una ocurrencia de ella en el texto, y este proceso se repite hasta haber pasado por todas las palabras dentro del archivo. La función dedicada para esta tarea es `wordCounterInFile` la cual recibe como argumento el `path` para un archivo. Tomando el mismo archivo anteriormente usado como ejemplo se muestra el siguiente conteo:

```
julia> wordcount, totalcount = mf.wordCounterInFile("../files/notabook_paragraphs_03.txt");

julia> wordcount
Dict{String, Int64} with 129 entries:
"maecenas"    => 2
"quam"        => 4
"odio"        => 3
"ullamcorper" => 4
"tempus"      => 3
"et"          => 3
"faucibus"    => 2
"dictumst"    => 1
"rutrum"      => 1
"vitae"       => 4
"tempor"      => 1
⋮             => ⋮

julia> totalcount
382
```

Lo que se muestra en `wordcount` son las palabras con sus respectivos conteos de ocurrencias, mientras que `totalcount` es el número total de palabras dentro del documento. Además, es posible saber cuál es el número de ocurrencias para una sola palabra, por ejemplo, para la misma anteriormente usada `"sed"`:

```
julia> get(words, "sed", -1)
10

julia> get(words, "desk", -1)
-1
```

La función `get` busca en el diccionario por llave (en este caso la palabra), y regresa el valor asociado a esa llave. En el caso en que no se encuentra esa llave entonces regresa un valor por defecto, en este caso es `-1`.

El procedimiento anteriormente descrito es fácilmente aplicable a un grupo de archivos en un directorio. Se escribió una función que identifique los archivos presentes en un directorio, aplique la función `mf.wordCounterInFile`, y que calcule el conteo del patrón y la densidad del patrón respecto al total de palabras para una futura comparación. Esta información se guarda en una lista de tuplas

ordenadas como `(patterncount, density, filename)` pero aquel documento en que el patrón no se haya detectado no se incluye en la lista. Ejemplo de esto se puede observar a continuación:

```
julia> collection = mf.wordCountInDirectory("../files/", "sed")
8-element Vector{Any}:
(10, 0.02617801047120419, "notabook_paragraphs_03")
(36, 0.029605263157894735, "notabook_paragraphs_10")
(96, 0.027126306866346424, "notabook_paragraphs_27")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(94, 0.02087960906263883, "notabook_paragraphs_34")
(101, 0.018703703703703705, "notabook_paragraphs_40")
(115, 0.01984126984126984, "notabook_paragraphs_45")

julia> collection = mf.wordCountInDirectory("../files/", "mollis")
7-element Vector{Any}:
(2, 0.001644736842105263, "notabook_paragraphs_10")
(3, 0.0008476970895733258, "notabook_paragraphs_27")
(10, 0.002701242571582928, "notabook_paragraphs_29")
(10, 0.0024055809477988932, "notabook_paragraphs_31")
(19, 0.004220346512661039, "notabook_paragraphs_34")
(9, 0.0016666666666666668, "notabook_paragraphs_40")
(6, 0.0010351966873706005, "notabook_paragraphs_45")

julia> collection = mf.wordCountInDirectory("../files/", "desk")
Any[]
```

Si la palabra buscada no se encuentra, el archivo no se lista, y si ningún archivo incluye el patrón en su texto, entonces la lista resultante es vacía. Ya que se tiene la información pertinente a cada documento es necesario ordenarlos de acuerdo al criterio mencionado en la actividad, este es, en orden descendente de acuerdo a su conteo de palabras. Para este propósito hay dos funciones que se pueden elegir de acuerdo a lo deseado, si solamente se quiere ordenar de acuerdo al número de ocurrencias o a la densidad del patrón respecto al total se puede utilizar la función `sortCollection`. Un ejemplo de esto se muestra a continuación al buscar el patrón `"sed"` en la colección de archivos usada hasta ahora:

```
julia> collection = mf.wordCountInDirectory("../files/", "sed")
8-element Vector{Any}:
(10, 0.02617801047120419, "notabook_paragraphs_03")
(36, 0.029605263157894735, "notabook_paragraphs_10")
(96, 0.027126306866346424, "notabook_paragraphs_27")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(94, 0.02087960906263883, "notabook_paragraphs_34")
(101, 0.018703703703703705, "notabook_paragraphs_40")
(115, 0.01984126984126984, "notabook_paragraphs_45")

julia> mf.sortCollection(collection, "wordcount")
8-element Vector{Any}:
(115, 0.01984126984126984, "notabook_paragraphs_45")
(101, 0.018703703703703705, "notabook_paragraphs_40")
(96, 0.027126306866346424, "notabook_paragraphs_27")
```

```
(94, 0.02087960906263883, "notabook_paragraphs_34")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(36, 0.029605263157894735, "notabook_paragraphs_10")
(10, 0.02617801047120419, "notabook_paragraphs_03")

julia> mf.sortCollection(collection, "density")
8-element Vector{Any}:
(36, 0.029605263157894735, "notabook_paragraphs_10")
(96, 0.027126306866346424, "notabook_paragraphs_27")
(10, 0.02617801047120419, "notabook_paragraphs_03")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(94, 0.02087960906263883, "notabook_paragraphs_34")
(115, 0.01984126984126984, "notabook_paragraphs_45")
(101, 0.018703703703703705, "notabook_paragraphs_40")
```

Aunque lo anterior es necesario para tener una idea de la importancia de nuestros archivos respecto a la ocurrencia del patrón no es suficiente para cumplir con el último inciso de la asignatura ¿Cómo diseñar un algoritmo de ordenamiento para ordenar de acuerdo a dos parámetros? Con esto en mente se buscó implementar un algoritmo sencillo [3], y además efectivo, que itera dos veces sobre la lista original intercambiando valores si detecta que uno es menor que otro, se adaptó para ajustarse a la lista de tuplas obtenida donde se guarda la información de los archivos y se agregó una condición extra para manejar el caso cuando el conteo de las palabras en un archivo es igual a otro:

```
function specialSort(collection)
    # Revome from collection elements without the pattern, i.e., word count and
    # density equal to -1
    collection = filter(x -> (x[1] != -1 && x[2] != -2), collection)
    n = length(collection)

    wordcount = map(x -> x[1], collection)
    density = map(x -> x[2], collection)

    for i in 1:n
        for j in 1:n
            if wordcount[i] < wordcount[j]
                wordcount[i], wordcount[j] = wordcount[j], wordcount[i]
                density[i], density[j] = density[j], density[i]
                collection[i], collection[j] = collection[j], collection[i]
            end
            if wordcount[i] == wordcount[j]
                newi, newj = density[i] < density[j] ? (j, i) : (i, j)   # index by density
                ↪  comparisson

                wordcount[i], wordcount[j] = wordcount[newj], wordcount[newi]
                density[i], density[j] = density[newj], density[newi]
                collection[i], collection[j] = collection[newj], collection[newi]
            end
        end
    end

```

Aunque si se aplica a la collección guardada en la variable `collection` del extracto de código inmediatamente arriba no habrá ningún cambio aparente

```
julia> mf.specialSort(collection)
8-element Vector{Any}:
(10, 0.02617801047120419, "notabook_paragraphs_03")
(36, 0.029605263157894735, "notabook_paragraphs_10")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(94, 0.02087960906263883, "notabook_paragraphs_34")
(96, 0.027126306866346424, "notabook_paragraphs_27")
(101, 0.018703703703703705, "notabook_paragraphs_40")
(115, 0.01984126984126984, "notabook_paragraphs_45")
```

Se considerará un caso en el que la primera entrada y la última de `collection` tienen el mismo conteo de palabras que la entrada para el archivo `"notabook_paragraphs_27"`,

```
julia> collection[1] = (96, 0.02617801047120419, "notabook_paragraphs_03")
(96, 0.02617801047120419, "notabook_paragraphs_03")

julia> collection[8] = (96, 0.01984126984126984, "notabook_paragraphs_45")
(96, 0.01984126984126984, "notabook_paragraphs_45")

julia> collection
8-element Vector{Any}:
(96, 0.02617801047120419, "notabook_paragraphs_03")
(36, 0.029605263157894735, "notabook_paragraphs_10")
(96, 0.027126306866346424, "notabook_paragraphs_27")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(94, 0.02087960906263883, "notabook_paragraphs_34")
(101, 0.018703703703703705, "notabook_paragraphs_40")
(96, 0.01984126984126984, "notabook_paragraphs_45")
```

Ahora, si se aplica la función de ordenamiento `specialSort` se esperaría que estas entradas con el mismo conteo de palabras sean ordenadas de acuerdo a su densidad justo como se muestra

```
julia> mf.specialSort(collection)
8-element Vector{Any}:
(36, 0.029605263157894735, "notabook_paragraphs_10")
(81, 0.02188006482982172, "notabook_paragraphs_29")
(87, 0.020928554245850373, "notabook_paragraphs_31")
(94, 0.02087960906263883, "notabook_paragraphs_34")
(96, 0.027126306866346424, "notabook_paragraphs_27")
(96, 0.02617801047120419, "notabook_paragraphs_03")
(96, 0.01984126984126984, "notabook_paragraphs_45")
(101, 0.018703703703703705, "notabook_paragraphs_40")
```

De esta manera termina la actividad de búsqueda y ordenamiento de textos de acuerdo al criterio de ocurrencias de un patrón en ellos, y al segundo criterio de ordenamiento sí es que algunos de ellos presentan el mismo conteo de ocurrencias.

Apéndice

```
module MiniFinder

using OffsetArrays    # To create arrays with first index equal to zero.

#=
Important notes

Link to Horspool algorithm explanation
- http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Text/Matching-Boyer-Moore2.html

Link of available ascii characters:
- https://www.rapidtables.com/code/text/ascii-table.html

How to 0-base index in Julia?
- https://medium.com/analytics-vidhya/0-based-indexing-a-julia-how-to-43578c780c37

Normalize entry lines
julia> Base.Unicode.normalize("día?= ~", stripmark=true, stripcc=true)

How to not use OffsetVectors?
How to use a dictionary for the ocurrence table?
=#

function occurrenceTable(textstring)
    # T -> textstring

    # textstring = isa(textstring, String) ? split(textstring, "") : textstring
    textstring = split(textstring, "")
    ascii = 256 # 128 # 256    # How many allowed ascii characters? There are 255 in total.

    T = OffsetVector(textstring, 0:(length(textstring)-1))
    lastocc = OffsetVector(ones(Int, ascii) .* -1, 0:(ascii-1))    # Initialize all to -1.

    for i in 0:(length(T)-1-1)    # Antes solo estana considerando un menos 1
        lastocc[Int(T[i][1])] = i
    end

    return lastocc
end

function horspool(textstring, pattern)
    # T -> textstring
    # P -> pattern

    # Dont even attempt it if the length of the textstring is smaller than
    # the length of the pattern.
    if length(textstring) < length(pattern)
        return -1
    end

    lastocc = occurrenceTable(pattern)
    textstring = split(textstring, "")
```

```
    pattern = split(pattern, "")

    T = OffsetVector(textstring, 0:length(textstring)-1)
    P = OffsetVector(pattern, 0:length(pattern)-1)

    n = length(T)
    m = length(P)

    i0 = 0

    while i0 <= n - m
        j = m - 1   # Start at the last char in the pattern

        # When the last char in the pattern is found, iterate over all chars and
        # compare to see if all match.
        while Int(P[j][1]) == Int(T[i0 + j][1])
            j = j - 1

            if j < 0
                return i0    # Starts indexing at 0
            end
        end

        i0 = i0 + (m - 1) - lastocc[Int(T[i0 + (m - 1)][1])]
    end

    return -1
end

function occurrencesInLine(line, pattern)
    patternlength = length(pattern)
    linelength = length(line)

    line_ = line

    occurrencelist = []
    occurrence = 0
    currentpos = 1

    while length(line_) >= patternlength
    # while occurrence != -1
        occurrence = horspool(line_, pattern)

        if occurrence == -1
            break
        end

        append!(occurrencelist, occurrence + currentpos)
        currentpos = currentpos + occurrence + patternlength

        if currentpos + patternlength > linelength
            break
        end

```

```
        line_ = line[currentpos:end]
        # println(line_)
    end

    return occurrencelist
end

function occurrencesInFile(file, pattern)
    occurlist = []

    open(file, "r") do io
        for (index, line) in enumerate(eachline(io))
            line_ = lowercase(line)
            occurrences = occurrencesInLine(lowercase(line_), pattern)

            if !isempty(occurrences)
                # push!(occurlist, (index, occurrences))
                occurrencesintuples = map(x -> (index, x), occurrences)
                append!(occurlist, occurrencesintuples)
            end
        end
    end

    if isempty(occurlist)
        println("No occurrences of \"$(pattern)\" where detected in \"$(file)\".")
    end
    return occurlist
end

function occurrencesInDirectory(directory, pattern)
    files = readdir(directory)
    occurdict = Dict()

    for file in files
        filename = findlast('.', file) ▷ x -> file[begin:x-1]    # remove extension from file name
        occurrences = occurrencesInFile(directory * file, pattern)

        occurdict[filename] = occurrences
    end
    return occurdict
end

function wordCountInFile(io)
    countbyword = Dict{String,Int}()
    totalwordcount = 0
    punctuationmarks = " .,:;-'!?"

    for line in eachline(io), word in split(line, in(punctuationmarks))    # Add punctuation symbols as
    ↪ needed
        lword = lowercase(word)
        countbyword[lword] = get(countbyword, lword, 0) + 1
        totalwordcount += 1
    end

```

```
    countbyword, totalwordcount
end

function wordCountInFile_(io, counter=Dict{String,Int}())
    for line in eachline(io), word in split(lowercase(line), !in('a':'z'))    # Add punctuation symbols as
    ↪ needed
        counter[word] = get(counter, word, 0) + 1
    end

    counter
end

function wordCountInDirectory(directory, pattern)
    files = readdir(directory)
    collection = []

    for file in files
        filename = findlast('.', file) ▷ x -> file[begin:x-1]    # remove extension from file name
        countbyword, totalwordcount = wordCountInFile(directory * file)
        patterncount = get(countbyword, pattern, -1)
        # density = !isequal(patterncount, -1) ? patterncount / totalwordcount : -1

        if patterncount != -1
            density = patterncount / totalwordcount
            push!(collection, (patterncount, density, filename))
        end
    end

    return collection
end

"""
    sortCollection(collection, sortby)

Sorts a vector of (wordcount, density, book) by wordcount or density in
descending order.
"""
function sortCollection(collection, sortby)
    options = Dict("wordcount" => 1, "density" => 2)
    option = get(options, sortby, -1)

    if !isequal(option, -1)
        return sort(collection, by = x -> x[option], rev=true)
    end
end

function specialSort(collection)
    # Revome from collection elements without the pattern, i.e., word count and
    # density equal to -1
    collection = filter(x -> (x[1] != -1 && x[2] != -2), collection)
    n = length(collection)

    wordcount = map(x -> x[1], collection)
    density = map(x -> x[2], collection)
```

```
213
214     for i in 1:n
215         for j in 1:n
216             if wordcount[i] < wordcount[j]
217                 wordcount[i], wordcount[j] = wordcount[j], wordcount[i]
218                 density[i], density[j] = density[j], density[i]
219                 collection[i], collection[j] = collection[j], collection[i]
220             end
221             if wordcount[i] == wordcount[j]
222                 newi, newj = density[i] < density[j] ? (j, i) : (i, j)   # index by density comparisson
223
224                 wordcount[i], wordcount[j] = wordcount[newj], wordcount[newi]
225                 density[i], density[j] = density[newj], density[newi]
226                 collection[i], collection[j] = collection[newj], collection[newi]
227             end
228         end
229     end
230
231     return collection
232 end
233
234 # function occurrenceDict(textstr)
235 #     lastocc = Dict()
236
237 #     for (i, char) in enumerate(textstr[begin:end-1])
238 #         lastocc[char] = i
239 #     end
240
241 #     return lastocc
242 # end
243
244 end # module MiniFinder
```

Referencias

[1] Shun Yan Cheung. The boyer-moore-horspool algorithm. http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Text/Matching-Boyer-Moore2.html, 2013. Visitado: 2022-11-25.

[2] Homero V. Ríos Figueroa, Fernando M. Montes Gonzáles, Víctor R. Cruz Álvarez. Análisis de algoritmos. Universidad Veracruzana, 2013.

[3] Stanley P. Y. Fung. Is this the simplest (and most surprising) sorting algorithm ever? *CoRR*, abs/2110.01111, 2021.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017.