Universidad Austral de Chile

Conocimiento y Naturaleza

22 JUNIO DE 2022

SEBASTIÁN MONTECINOS, MATIAS RIVERA

Universidad Austral De Chile

Profesor Héctor Ferrada

Introducción

Nuestro problema a resolver se llama string pattern matching, el cual consiste en encontrar la posición en donde comienza cada ocurrencia de una serie de caracteres llamado “patrón” en un texto dado.

El objetivo es implementar tres métodos de búsqueda de patrones en un texto, ver cuál de estos es el más eficiente y cuanta memoria extra ocupan. Para determinar esto, mediremos los tiempos que se demora cada algoritmo en encontrar todas las ocurrencias de distintos “patrones” en diferentes textos. Cabe destacar que nuestros algoritmos serán programados en lenguaje C++ y pseudocódigo.

La búsqueda exhaustiva, que es el primer método que probaremos, es de orden lineal, es decir, hay que recorrer todo el texto una vez para encontrar todas las ocurrencias del patrón, además, el tiempo de ejecución es proporcional al largo del texto. Y en términos de memoria utilizada, no ocupa memoria extra, ya que solo hace comparaciones, no crea arreglos o listas extra.

La búsqueda mediante la Lista Enlazada de Sufijos si ocupa memoria extra y a priori parece el que más necesitará, aunque en tiempo de ejecución se espera que sea al menos más rápido que la búsqueda exhaustiva.

Y por último la búsqueda mediante un arreglo de sufijos también requiere memoria extra pero menos que la lista enlazada, y al igual que el caso anterior se espera que sea más rápida que la búsqueda exhaustiva.

Hipótesis

Nuestra hipótesis consiste en que el algoritmo de fuerza bruta, será el que menos memoria extra usará, el más sencillo de programar, pero el que más tiempo necesitará al momento de ejecutar

El método de búsqueda mediante lista enlazada, será el que más memoria extra ocupará (debido a que se necesitaran muchos punteros) y a la vez tomará menos tiempo de ejecución respecto al método de fuerza bruta.

Y respecto al método de búsqueda mediante un arreglo, creemos que será bastante eficiente, ocupará menos memoria extra que una lista enlazada y también será más rápido que el método de búsqueda exhaustiva (tiempo de ejecución).

Resumen

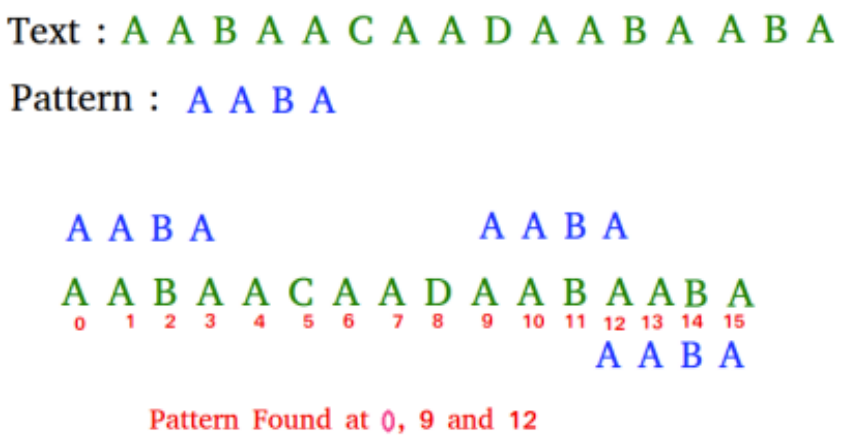
En una primera instancia se implementará el método de búsqueda exhaustiva, el cual es el menos eficiente y en teoría, el más fácil de programar. Este consiste en comparar carácter a carácter el patrón que se quiere buscar con el texto principal.

Luego se incluirá el método de búsqueda mediante lista enlazada de sufijos, la cual consiste en crear una lista doblemente enlazada que tendrá el orden lexicográfico de todos los sufijos (El orden lexicográfico depende de la tabla ASCII, es similar al orden alfabético).

Y el último método a implementar será un arreglo que contenga los índices de los n sufijos del texto ordenados lexicográficamente, y aprovecharnos de éste para así crear un algoritmo capaz de completar la tarea.

Metodología

Para implementar el algoritmo de búsqueda de fuerza bruta, compararemos el patrón con todos los índices del texto. Por ejemplo, sea K la cantidad de letras que posee el patrón, se comparan todos los substring de tamaño K que existen en el texto, si se encuentra una ocurrencia del patrón se almacena la posición donde este comienza en un arreglo y en caso de no encontrar ocurrencias, el arreglo quedaría vacío y en pantalla se mostrará un mensaje indicando que no se encontró ninguna coincidencia..
A continuación se presenta el pseudocódigo y tabla de ejecución respecto a la siguente entrada de texto y patrón:



Para conseguir escribir este algoritmo de manera correcta, primero se realizó un pseudocódigo que se usó de “borrador”, luego se fue ejecutando con las entradas ya mencionadas. A medida que se encontraba una discrepancia con el resultado final esperado se buscaba el error y se arreglaba. Reiteramos este proceso un par de veces y finalmente obtuvimos el algoritmo adecuado y correcto.
Cabe mencionar que las funciones strlen y strcmp serán sustituidas por funciones creadas por nosotros una vez escrito en código de C++, aquí fueron usadas con el fin de simplificar el pseudocódigo.

```
Inicio
        frase <- “AABAACAADAABAABA”
        patron <- “AABA”
        largoF <- strlen(frase)
        largoP <- strlen(patron)
        int arreglo[10]
        para i<-0 hasta i <- largoP-1 hacer
                j <- 0
                si (frase [i] == patron[j] ) entonces
                        pos <- i
                        mientras (frase[i] != ‘\0’ and patron[j] != ‘\0’ and frase[i] == patron[j])
                                i <- i+1
                                j <- j + 1
                        fin mientras
                        si (patron [j] == ‘\0’) entonces
                                arreglo[k] <- pos
                                k <- k+1
                                i <- i -largoP
                        si no
                                i <- pos
                                pos <- 0
                        fin si
                fin si
        fin para
        si (pos == 0) entonces
                si (k == 0) entonces
                        escribe “No se encontró ninguna coincidencia”
                si no
                        escribe “Búsqueda exitosa …”
                fin si
        fin si
        si (k != 0) entonces
                escribe “El patrón comienza en las posiciones: ”
                para n <- 0 hasta n <- k-1 hacer
                        escribe arreglo[i] , “ ”
                fin para
        fin si
fin
```

Link Tabla de ejecución: https://i.imgur.com/oCGcSM3.jpg

A continuación se presenta el pseudocódigo de un algoritmo que crea un arreglo con el índice de los sufijos de una cadena ordenados lexicográficamente, la cual usaremos más adelante para resolver el problema de una manera más rápida y eficiente.

```
Struct sufijo
inicio
        Int id
        char *suf
fin struct


Funcion compara(sufijo a , sufijo b)
Inicio
        Si (strcmp(a.suf , b.suf) < 0) entonces
                Retorna 1
        Si no
                Retorna 0
        Fin si
fin


Funcion *arrSufijos(char txt , int n)
Inicio
        Sufijo sufijos[n];
        para i<-0 hasta i <- n-1 hacer
                sufijos[i].id = i
                sufijos[i].suf = (txt+i)
        fin para
        sort(sufijos , sufijos + n , compara)
        *índices <- new int[n]
        Para i<-0 hasta i<-n-1 hacer
                indices[i] = sufijos[i].id
        fin para
        retorna índices
fin
```

Para aprovechar este arreglo de sufijos ordenados lexicográficamente, buscaremos la primera ocurrencia del patrón (según los índices del arreglo) y la última, entonces el patrón estará en todos los índices intermedios entre la primera y la última ocurrencia, un ejemplo a continuación.

	AABAACAADAABAABA
S_i	**Sufijo**
S_{15}	A
S_{12}	AABA
S_9	AABAABA
S_0	AABAACAADAABAABA
S_3	AACAADAABAABA
S_6	AADAABAABA
S_{13}	ABA
S_{10}	ABAABA
S_1	ABAACAADAABAABA
S_4	ACAADAABAABA
S_7	ADAABAABA
S_{14}	BA
S_{11}	BAABA
S_2	BAACAADAABAABA
S_5	CAADAABAABA
S_8	DAABAABA

Arreglo de índice de sufijos ordenados lexicográficamente

15	12	9	0	3	6	13	10	1	4	7	14	11	2	5	8

Si buscamos el patrón “BA” deberemos buscar la primera ocurrencia en este arreglo de índices de sufijos, la cual sería el sufijo 14, es decir la posición 11 en el arreglo. Luego buscar la última ocurrencia, que sería el sufijo 2, es decir la posición 13 de arreglo. Así se puede concluir que entre las posiciones 11 y 13 se encuentran los índices en donde existe dicho patrón.

El siguiente pseudocódigo muestra las funciones a usar para la solución de la lista doblemente enlazada, la idea es utilizar índices para los sufijos tal y como lo hacemos con el arreglo de sufijos de arriba, la diferencia, y en lo que se quiere aprovechar el hecho de tener una lista doblemente enlazada, es buscando desde el final hasta el inicio de la lista según la letra con la que comience el patrón a buscar, por ejemplo, si fuera la Z, se comenzará a buscar desde el final hasta el inicio, ya que es más probable que haya una Z al final de la lista ordenada que al principio, si por otro lado la primera letra fuera la B, se buscará desde el inicio hasta el final, se utilizara esta forma hasta la letra M, y desde la letra N en adelante, se buscará desde el final hasta el principio.

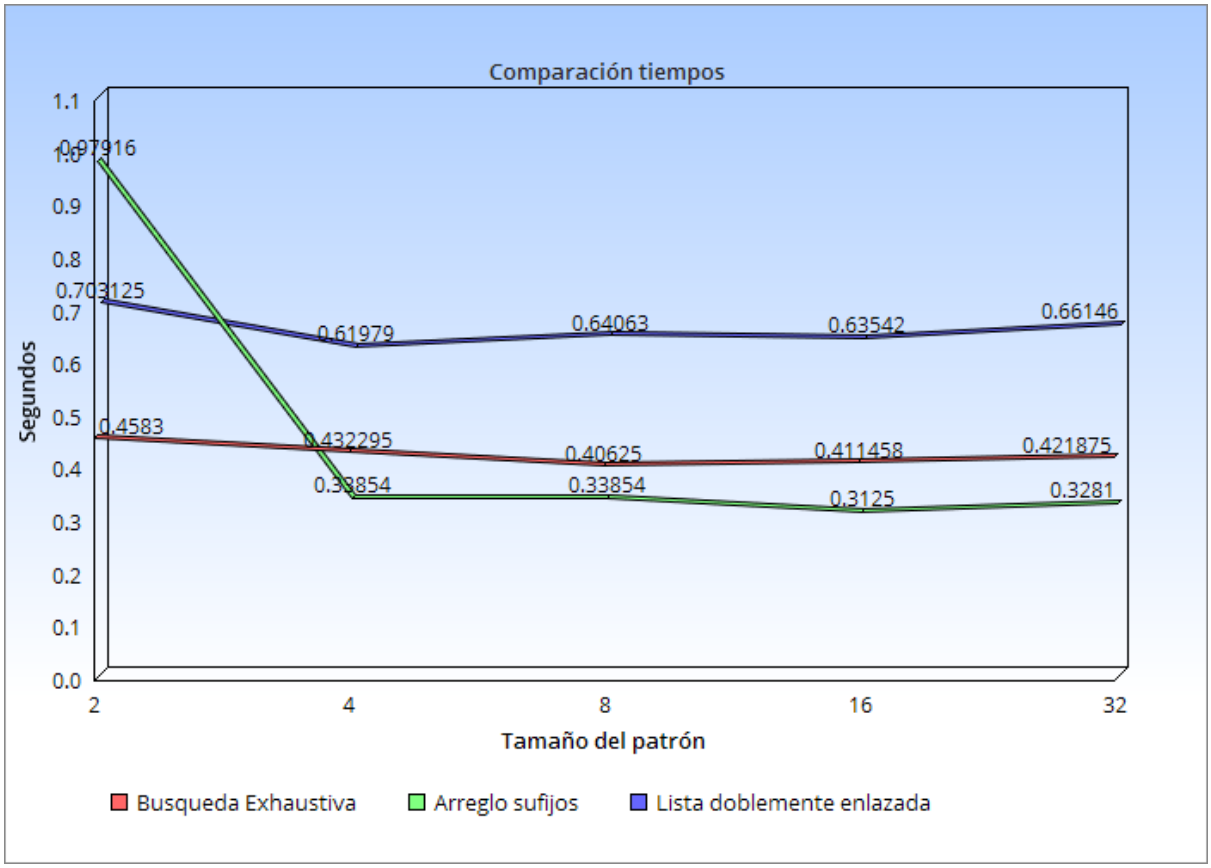
```
struct nodo
{
        char *suf
        int id
        nodo* siguiente
        nodo* atras
} *primero, *ultimo

funcion insertar Nodo(char sufi, int num)
inicio
        nodo* nuevo <- new nodo
        nuevo->*suf <- sufi
        nuevo->id <- num

        si(primero==NULL)
                primero <- nuevo
                primero->siguiente <- NULL
                primero->atrás <- NULL
                ultimo <- primero
        si no
                ultimo->siguiente <- nuevo
                nuevo->siguiente <- NULL
                nuevo->atras <- ultimo
                ultimo <- nuevo
        fin si

fin

funcion indicesLista(*indi)
inicio
        nodo* actual = new nodo
        actual = primero;
        si (primero!=NULL)
        int i <- 0

                mientras(actual!=NULL)
                inicio
                        indi[i] <- actual->dato
```

```
                    i++
                    actual = actual->siguiente

            fin mientras
        si no
            print(" La lista está vacía”)
        fin si
fin
funcion compara(sufijo a, sufijo b)
inicio
        si (strcmp(a.suf, b.suf)<0) entonces
            retorna 1
        si no
            retorna 0
        fin si
fin

funcion listaSufijos(char txt, int n)
inicio
        para i <-0 hasta i<-n-1 hacer
            insertarNodo(txt + i, i)
        fin para
        mergesort(nodo *primero) //aun no implementado por completo
        *indices <- new int[n]
        indicesLista(índices)
fin
```

Gráfico comparativo de resultados.

Para recrear este gráfico utilizamos patrones de distinto tamaño (2, 4, 8, 16 y 32), los cuales fueron GA, GATC, GATCAATG, GATCAATGAGGTGGAC y GATCAATGAGGTGGACACCAGAGGCGGGGACT.

El código fue ejecutado un total de 5 veces por patrón y guardamos los tiempos de búsqueda obtenidos en cada una de las ejecuciones, logrando así un promedio para cada caso. (Se usó el archivo de texto dna.50MB)



Con este gráfico podemos concluir que para patrones pequeños, el algoritmo de búsqueda exhaustiva fue el más veloz, pero a medida que crece el tamaño del patrón, el algoritmo de búsqueda mediante arreglo de sufijos ordenados, es el más eficaz.

Memoria extra

El método de búsqueda exhaustiva no utiliza memoria extra, ya que solo va comparando el patrón ingresado, letra por letra, en el texto de entrada.

El método de Arreglo de Sufijos, utiliza un arreglo de enteros (índices de sufijos ordenados lexicográficamente) del mismo tamaño que el texto de entrada.

Y el método que utiliza listas enlazadas dobles, utiliza una lista de tamaño n (n = tamaño del texto de entrada), la cual contiene un número entero y dos punteros, los cuales usan mucha memoria adicional.

Aclaración

El uso de strncmp se debe al hecho de que las funciones strncmp que intentamos implementar por nuestra cuenta tardaba demasiado en comparación a la de la librería string, aunque dejamos las funciones codificadas (comentadas), finalmente utilizamos la de la librería string ya que nos entregaba resultados más esperados.

Conclusión

En un principio, mencionamos en nuestra hipótesis los resultados que esperábamos encontrar. La cual dice, que el algoritmo más eficaz sería el algoritmo de búsqueda mediante arreglo de sufijos, luego el algoritmo de búsqueda mediante listas enlazadas y en última posición el algoritmo de búsqueda exhaustiva.

Nuestros resultados confirman nuestra primera teoría, el arreglo de sufijos acelera bastante la búsqueda del patrón. Al momento de comparar los otros dos algoritmos, nos llevamos una gran sorpresa, ya que no fue lo esperado, la búsqueda exhaustiva fue más veloz que la lista enlazada. Quizás, una mejor implementación del algoritmo de búsqueda que utiliza lista enlazada, daría vuelta el resultado, confirmando así nuestra hipótesis inicial.

.