

1) REPORTE CUMPLE
2) TIENE BUENA ELABORACION PROPIA

RE : 90%

3) PUEDE SER CONVERTIDO EN UN TUTORIAL

EXTRA : 10
(INDIVIDUAL)

SI SE MEJORAN
DETAILS

Sobre *matching* en hileras

100%

4) FUE, INDIVIDUAL

Isaac Palma Medina

5) NO SE USO
ADN EN
EJEMPLOS
INICIALES

Escuela de Informática, Universidad Nacional de Costa Rica

EIF203 Estructuras discretas

NRC 41713

Dr. Carlos Loría Sáenz

Julio de 2022

Informative only

Índice general

Capítulo I Introducción.....	1
Resumen documental	1
Palabras clave.....	1
Keywords.....	1
Justificación general.....	2
Capítulo II Desarrollo	3
Acercamiento al <i>string matching</i>	3
Ejemplo básico	3
Algoritmos de resolución	3
Solución ingenua o fuerza bruta	3
Solución por el método Knuth-Morris-Pratt (KMP)	6
Solución por el método Boyer-Moore	9
Caso de uso en genética molecular	10
Nociones del ADN	10
Aplicación de los patrones	11
Implementación y análisis de los algoritmos	11
Análisis algoritmo naive	11
Análisis algoritmo KMP	13
Análisis algoritmo Boyer-Moore	16
Capítulo III Conclusión	17
Referencias.....	18
Anexos.....	20
Anexo 1	20
Anexo 2	20
Anexo 3	20

Índice de ilustraciones

Ilustración 1 Obtención de la <i>prefix table</i>	6
Ilustración 2 Búsqueda de un patrón con KMP	7
Ilustración 3 Procedimiento del algoritmo Boyer-Moore.....	9
Ilustración 4 Gráfico descriptivo del tiempo de corrida SM naive	12
Ilustración 5 Gráfico descriptivo del tiempo de corrida SM KMP.....	13
Ilustración 6 Autómata inicial	14
Ilustración 7 Autómata con nuevos arcos	15
Ilustración 8 Autómata final	15
Ilustración 9 Gráfico descriptivo del tiempo de corrida SM Boyer-Moore	16

Índice de algoritmos

Algoritmo 1 SM naive.....	5
Algoritmo 2 SM KMP	8
Algoritmo 3 SM Boyer-Moore.....	10

Informative only

Capítulo I Introducción

Resumen documental

con el objetivo de poder
~~Introducción:~~ Bajo la premisa de desear encontrar las ocurrencias de un patrón en un texto dado, se han desarrollado diversas técnicas para esta labor. Entre ellas destacan Knuth-Morris-Pratt, publicado en 1976, y Boyer-Moore, el cual fue publicado en 1977. Ambos algoritmos, y sus respectivos iguales, brindan una solución efectiva, la cual puede ser estudiada y entendida, para comprender mejor sus casos de uso.

Materiales y métodos: En esta investigación fueron utilizados diversos recursos para su elaboración, optando principalmente por la revisión documental.

Palabras clave

String matching, patrón, texto, ocurrencia, índice y match.

Introduction: Under the premise of wanting to find the occurrences of a pattern in a given text, several techniques have been developed for this task. Among them are Knuth-Morris-Pratt, published in 1976, and Boyer-Moore, which was published in 1977. Both algorithms, and their respective equals, provide an effective solution, which can be studied and understood, to better understand their use cases.

Materials and methods: In this research, several resources were used for its elaboration, opting mainly for documentary review.

Keywords

String matching, pattern, text, occurrences, index, and match.

Justificación general

El *string matching*, es considerado un problema fundamental en informática, este posee muchas aplicaciones en distintos campos, como por ejemplo los editores de texto, sin embargo, su alcance puede ser extendido hasta usos en el análisis genético molecular. El problema consiste en encontrar las apariciones (ocurrencias) de una cadena de caracteres en un determinado texto. ✓ REF

Existen varios algoritmos que buscan brindar una solución a dicha premisa, el alcance del presente proyecto toma en cuenta algunos de estos, entre los que se encuentran: solución naive (ingenua), solución por medio del método Knuth-Morris-Pratt y la resolución por medio del algoritmo Boyer-Moore. ✓

El entendimiento de dichos algoritmos resulta crucial para aplicaciones en el mundo informático, sobre todo en el sector del desarrollo, en ese caso, nulo conocimiento sobre el tema podría resultar en determinadas consecuencias, por ejemplo, referente a la eficiencia.

desarrollo
en que sentido?

Informative only

Capítulo II Desarrollo

Acercamiento al *string matching*

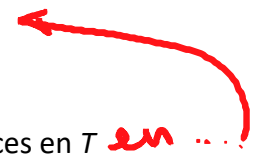
El *string matching* o búsqueda de subcadenas, intenta “(...) encontrar posiciones (índices) donde una o varias subcadenas (también llamadas patrones) son encontradas en una cadena más grande (texto buscado)” (Gimel’farb, 2011, diap. 3). Por lo que para un texto dado se deben encontrar aquellas ocurrencias del texto (subcadena) deseado.

Ambos componentes de dicho problema (patrón y texto), son la concatenación de elementos de un “alfabeto” (un conjunto de caracteres de tamaño constante) denotado generalmente por el símbolo Σ . (Park, 2015, diap. 3). Otras notaciones populares en el estudio del *string matching*, son:

- El texto en el que se busca se denota como T
- El patrón buscado es llamado P , y es menor en cantidad de elementos que T
- n y m son el largo de P y T , respectivamente

(Park, 2015, diap. 3)

Ejemplo básico

- $T = \text{AGCATGCTSCAGTCATGCTTAGGCTA}$ *destaca* 
- $P = \text{GCT}$
- Conclusión: T tiene tres ocurrencias de P o P aparece tres veces en T *en ...!*

En el ejemplo se limita a coincidencias exactas de P en T , y no necesariamente de ocurrencias parciales de P en T (Park, 2015, diap. 4).

Algoritmos de resolución

Mediante la implementación de algoritmos, existen distintas maneras de resolver el problema, por lo que hay formas más eficientes que otras de lograr el *string matching*; no se cubren *matches* parciales. A continuación, se cubren distintos algoritmos famosos para dicho propósito.

Solución ingenua o fuerza bruta

La solución *naive*, ingenua o fuerza bruta hace uso de la implementación de *windows* o ventanas, se denomina el *window size* (ws) a la cantidad de caracteres que posee la cadena patrón.

Se comienza por identificar los caracteres que van desde el índice 0 hasta el índice que coincida con el $ws - 1$, verificando si dicha hilera de T coincide con la cadena P . (Agrawal, 2020).

Ejemplo SM naive

Cadena P

A	A	B	A
0	1	2	3

Se obtiene una ws de 4.

Cadena T

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

Se verifican los caracteres según el ws . Hecha la verificación se puede devolver el índice en el que la ocurrencia se da (en este caso 0), y continuar con la búsqueda. (Agrawal, 2020).

Continuación de la búsqueda

Se hace un corrimiento de la ventana hacia la derecha, esta se va moviendo en busca de P . (Agrawal, 2020).

Segunda iteración

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

⋮

Última iteración

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

De lo anterior se puede notar que la ventana se mueve desde el índice 0 hasta $n - m$, además m es usado para acceder a los componentes de una subhilera de T de m elementos.

Implementación en código

Se puede inferir que se necesitarán dos ciclos, uno que recorra desde 0 hasta $n - m$ y otro que se encargue de recorrer los elementos de la ventana. Además, se deberá devolver el índice en que el patrón ocurre. Implementado en Python 3.9 de la siguiente manera:

Algoritmo 1 SM naive

→ last [index]

```

1. def naive(t:str, p:str):
2.     m:int = len(p)
3.     n:int = len(t)
4.     a:list = []
5.     for i in range(n - m + 1):
6.         flag:bool = True
7.         for j in range(m):
8.             if t[i + j] != p[j]:
9.                 flag = False
10.                break
11.            if j == m - 1 and flag:
12.                a.append((i, i + m - 1))
13.        return a
14.
15. print(naive("AGCATGCTGCAGTCATGCTTAGGCTA", "GCT"))
16.
17. """
18. Output
19. [(5, 7), (16, 18), (22, 24)]
20. """
21.

```

Interpretación propia de otro código fuente: Agrawal (2020a) y garg10may (2018)

El algoritmo anterior comienza por recibir dos hileras, de las cuales se obtiene el largo del texto T , y el tamaño de la ventana o el largo de la hilera P . Posterior a eso se entra a un ciclo *for* el cual procede a “mover” la ventana por T , y se crea una bandera para determinar si en el próximo ciclo se encuentra una discrepancia, en caso de localizar la bandera toma el valor de falso y el ciclo secundario se detiene, pasando a la próxima interacción del primer *for*. Si el *for* secundario se lograra completar significa que en ese rango se encuentra P , y se imprime el índice.

Problema de la solución naive

El problema de esta solución está relacionado con que recorre demasiadas veces la hilera. En caso de fallas con una comparación el algoritmo se devolverá hasta haber comprobado la totalidad de T , implicando que en hileras de una gran extensión el algoritmo es sumamente ineficiente. Este algoritmo posee una complejidad de $O(mn)$, representado por $O(m)$, debido a los múltiples recorridos que este debe hacer según la cantidad de caracteres de m y n . (Bari, 2018).

✓
(donde m y
n son ...
(recorridos))

Solución por el método Knuth-Morris-Pratt (KMP)

El algoritmo KMP, tiene el mismo objetivo que el anterior, buscar un patrón P en un texto T , esto lo logra por medio la no regresión del acceso a la hilera n . Se hace uso de una *longest prefix suffix* (LPS) o *prefix table*, para definir la lógica en el que P va a ser comparado con T . KMP posee un tiempo de $O(n + m)$, también representado por $O(m)$ (la que por principio es la hilera más larga). (Bari, 2018).

Obtención del LPS

Se trata de observar el prefijo más largo que a la vez es sufijo de P , pero no igual a P , dicha tabla será representada en un arreglo del mismo tamaño de la hilera. (Bari, 2018). Hilera de ejemplo:

a	b	c	d	a	b	c
---	---	---	---	---	---	---

Construcción de la tabla

Bari (2018) señala que, la tabla se construye de manera que se van tomando elementos de la hilera, la posición 0 se toma como el inicio del prefijo, posteriormente se recorre la hilera P , obteniendo distintos sufijos hasta que se logre un *match*, en ese caso se le asignará un índice de ocurrencia a esa posición en caso contrario, se le dará el valor de 0:

Ilustración 1 Obtención de la *prefix table*

Obteniendo el LPS							eq(x, y)		lado_iz(x) o lado_de	
Elemento	a	b	c	d	a	b	c	¿Es x igual a y? Output: Sí o No	¿Hay elementos a la izquierda (iz) / derecha (de) de x que no se hayan comparado? Output: Sí o No	
Índice	0	1	2	3	4	5	6			

<div><div><div>a</div><div>b</div></div><div>0</div></div> <p>eq(a, b) No lado_de(a) No lado_iz(b) No</p> <p>Al no ser iguales y al no haber más elementos se asigna un 0</p> <div><div>a</div><div>b</div></div> <div>0</div>	<div><div><div>a</div><div>b</div><div>c</div></div><div>0</div><div>0</div></div> <p>eq(a, c) No lado_de(a) Sí lado_iz(c) Sí</p> <div><div><div>a</div><div>b</div><div>c</div></div><div>0</div><div>0</div></div> <p>eq(ab, bc) No lado_de(ab) No lado_iz(bc) No Se asigna un 0</p> <div><div><div>a</div><div>b</div><div>c</div></div><div>0</div><div>0</div><div>0</div></div>	<div><div><div>a</div><div>b</div><div>c</div><div>d</div></div><div>0</div><div>0</div><div>0</div><div></div></div> <p>eq(a, d) No... eq(ab, cd) No... eq(abc, bcd) No... Se asigna un 0</p> <div><div><div>a</div><div>b</div><div>c</div><div>d</div><div>a</div></div><div>0</div><div>0</div><div>0</div><div>0</div><div></div></div> <p>eq(a, a) Sí... eq(ab, da) No... eq(abc, cda) No... eq(abcd, bcda) No... Debido al sí, se asigna el índice de esa ocurrencia 1</p>	<div><div><div>a</div><div>b</div><div>c</div><div>d</div><div>a</div><div>b</div></div><div>0</div><div>0</div><div>0</div><div>0</div><div>1</div><div></div></div> <p>eq(a, b) No... eq(ab, ab) Sí... eq(abc, dab) No... eq(abcd, cdab) No... eq(abcda, bcdab) No... Se asigna un 2</p> <div><div><div>a</div><div>b</div><div>c</div><div>d</div><div>a</div><div>b</div><div>c</div></div><div>0</div><div>0</div><div>0</div><div>0</div><div>1</div><div>2</div><div></div></div> <p>eq(a, c) No... eq(ab, bc) No... eq(abc, abc) Sí... eq(abcd, dabc) No... eq(abcda, cdabc) No... eq(abcdab, bcdabc) No... Se asigna un 3</p>	<div>Resultado</div> <div><div><div>a</div><div>b</div><div>c</div><div>d</div><div>a</div><div>b</div><div>c</div></div><div>0</div><div>0</div><div>0</div><div>0</div><div>1</div><div>2</div><div>3</div></div>
--	---	---	--	---

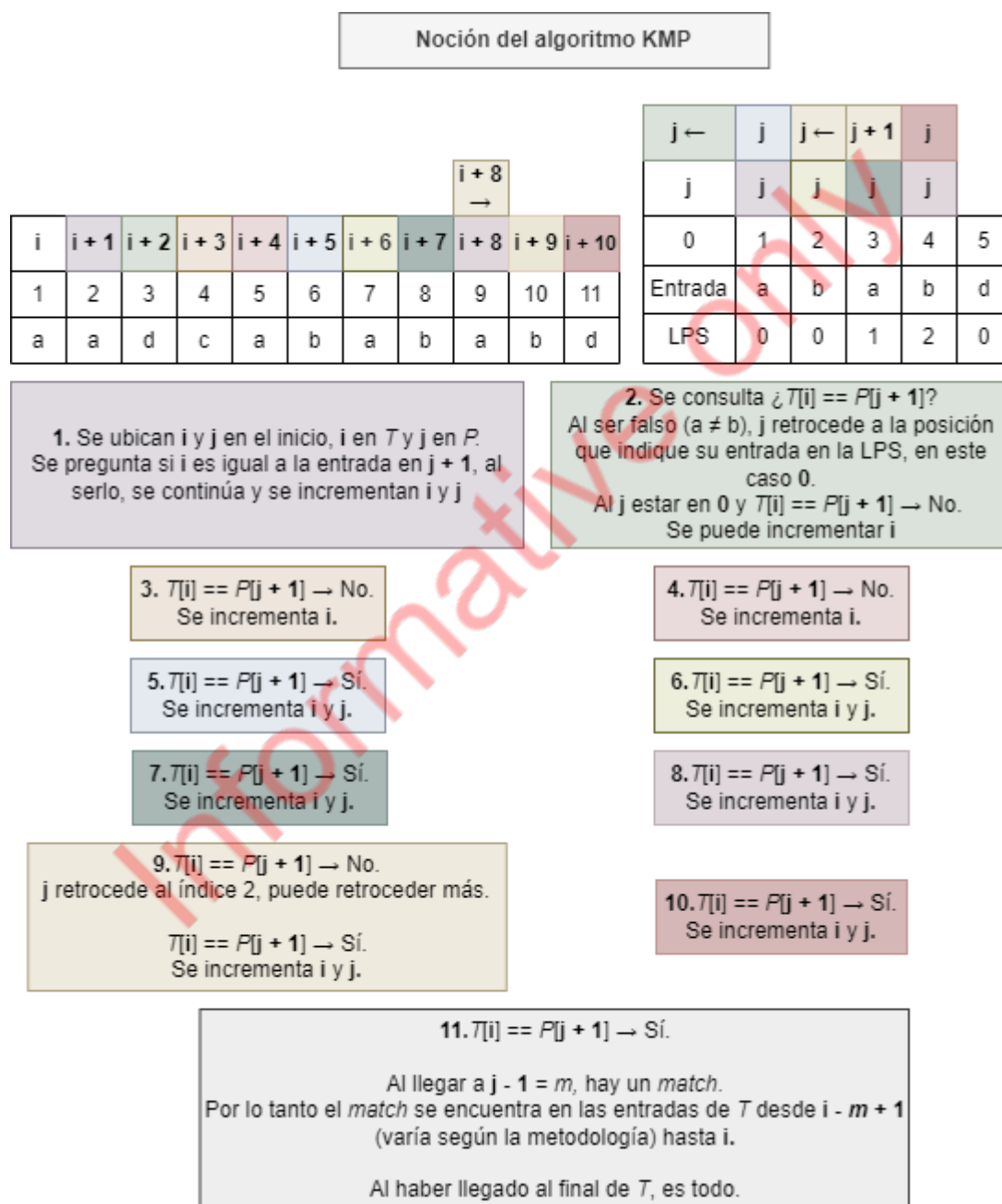
Elaboración propia ✓

Una vez obtenida la tabla, se hará uso del grueso del algoritmo KMP, el cual constará de recorrer T , sin la necesidad de nunca retroceder, aquella hilera en la que se harán constantes retrocesos será la *prefix table*. (Bari, 2018).

Noción visual del algoritmo KMP

Dados, $T = \text{aadcababababd}$ y $P = \text{ababd}$.

Ilustración 2 Búsqueda de un patrón con KMP



Elaboración propia

Implementación en código

Algoritmo 2 SM KMP

```

1. def get_lps(p:str): # Construcción de la LPS.
2.     m:int = len(p)
3.     lps = [0] * m
4.     index_behind, i = 0, 1
5.     while i < m:
6.         if p[i] == p[index_behind]:
7.             lps[i] = index_behind + 1
8.             index_behind += 1
9.             i += 1
10.        elif index_behind == 0:
11.            lps[i] = 0
12.            i += 1
13.        else:
14.            index_behind = lps[index_behind - 1]
15.    return lps
16.
17. def knuth_morris_pratt(t:str, p:str):
18.     m:int = len(p)
19.     n:int = len(t)
20.     lps:list = get_lps(p) # Obtiene la LPS correspondiente al patrón.
21.     i:int = 0
22.     j:int = 0
23.     a:list = []
24.     while i < n: # Saltos entre los índices según la LPS.
25.         if t[i] == p[j]:
26.             i, j = i + 1, j + 1
27.         if j == m :
28.             a.append((i - m, i - 1))
29.             j = lps[j - 1]
30.         elif i < n and p[j] != t[i]:
31.             if j != 0 :
32.                 j = lps[j - 1]
33.             else :
34.                 i += 1
35.     return a
36.
37. print(knuth_morris_pratt("aadcabababd", "ababd"))
38.
39. """
40. Output
41. [(6, 10)]
42. """
43.

```

Interpretación propia de otro código fuente: Obtención del LPS: NeedCode (2021) y Knuth et al. (1977), Búsqueda en T: Jain (2022), m00nlight (2016) y Knuth et al. (1977)

El algoritmo ataca directamente una debilidad de la solución *naïve*, pues omite la regresión sobre T , lo anterior ocurre ya que la búsqueda se logra sobre la base de los datos obtenidos de fallos previos. La tabla (LPS) es creada con la finalidad de determinar la próxima ocurrencia de un elemento del patrón en P , iterándose únicamente este. (Knuth et al., 1977, p. 329).

Solución por el método Boyer-Moore

Bajo la misma meta nace dicho algoritmo, inicialmente desarrollado en 1997, posee como principal diferenciador el preprocesamiento de la cadena (patrón P), lo cual permite el salto de comparaciones en el texto T. (Hutch, 2004).

LPS es pre-procesamiento

Dicho algoritmo hace uso de comparaciones de derecha a izquierda con la finalidad de aprovechar los *mismatches* que se puedan presentar durante el análisis. Para el procedimiento se hace uso de la heurística denominada *bad character rule* (BCR), el cual indica que al haber una discordancia se saltarán comparaciones hasta que el *mismatch* se convierta en una coincidencia, o el patrón P sea movido hasta haber pasado en su totalidad aquella discordancia.

Ilustración 3 Procedimiento del algoritmo Boyer-Moore

Texto	D	E	F	F	E	F	D	E	F	E	F	D
Patrón	E	E	F	F	F	F	D	E				

Elaboración propia

En dicha imagen se debe mover el patrón hasta que aquel carácter incorrecto coincida, en este caso que la letra E del *mismatch* coincida con una E correspondiente en el patrón. Se analiza la hilera P (más eficiente que utilizar a T) hasta encontrar su igual en esta, con la finalidad de evitar dos comparaciones innecesarias.

Texto	D	E	F	F	E	F	D	E	F	E	F	D
Patrón				E	E	F	F	F	D	E		

Elaboración propia

De esta manera se continuará aplicando el algoritmo hasta encontrar una coincidencia, en síntesis, se analiza la hilera de derecha a izquierda, en caso de una discordancia, se pregunta si ese carácter problemático existe en el patrón P, en caso de existir se encaja con la primera coincidencia en T, si no existiese se hace el salto de todas las comparaciones según el largo de P. Hasta encontrar y contabilizar alguna coincidencia.



Implementación en código

Algoritmo 3 SM Boyer-Moore

```

1. def boyer_moore(t:str, p:str):
2.     m:int = len(p)
3.     n:int = len(t)
4.     a:list = []
5.     bad_char:str = [-1] * (256)
6.     # Se inicializa según la cantidad posible de caracteres, en este caso 256 según
    ASCII.
7.     for i in range(m):
8.         bad_char[ord(p[i])] = i
9.         # En la lista de caracteres, para cada elemento del patrón, guarda su valor
    posicional en este (primera ocurrencia).
10.    j:int = m - 1
11.    s = 0
12.    while s <= (n - m): # Tomando en cuenta los largos de t y p.
13.        while j >= 0 and p[j] == t[s + j]:
14.            j -= 1
15.        if j < 0:
16.            a.append((s, s + m - 1)) # En caso de coincidencia lo guarda.
17.            if s + m < n:
18.                s += m - bad_char[ord(t[s + m])]
19.            else:
20.                s += 1
21.        else:
22.            s += max(1, j - bad_char[ord(t[s + j])])
23.        j = m - 1
24.    return a
25.
26. print(boyer_moore("trusthardtoothbrushes", "tooth"))
27. print(boyer_moore("iscreamyouscreamweallscreamforicecream", "scream"))
28.
29. """
30. Output
31. [(9, 13), (21, 25)]
32. [(1, 6), (10, 15), (21, 26)]
33. """
34.

```

ADN HUBIERA SIDO BUENO TAMBIÉN

Interpretación propia de otro código fuente: Deshmukh (2021) y Langmead (2015).

Caso de uso en genética molecular

Una aplicación directa de los algoritmos de *string matching* es en el caso de la genética molecular, más específicamente en la secuenciación del ácido desoxirribonucleico (ADN).

Nociones del ADN

Primeramente, el ADN debe ser definido como el material que contiene la información hereditaria en los humanos y casi todos los demás organismos (MedlinePlus, 2022) dicha estructura posee una unidad mínima denominada “nucleótido” visualizada generalmente como un eslabón que es parte de una cadena de ADN o de ARN. Estos nucleótidos deben estar unidos a un nucleótido “correspondiente” dando lugar a la

secuenciación del ADN, los puentes de enlace entre estos son denominados pares de bases. (MedlinePlus, 2022).

Los pares de bases nitrogenadas se conforman por cuatro candidatos, los cuales son adenina (A), citosina (C), guanina (G) y timina (T). Dichos pares se ordenan de la siguiente forma: adenina con timina y guanina con citosina. (LabTest, 2020).

La información de la secuencia del ADN se trata de la clave para el entendimiento del genoma humano y para diversos estudios referentes al campo de la genética. (LabTest, 2020).

Aplicación de los patrones

La aplicación de los algoritmos desarrollados se da al momento de buscar patrones específicos en una cadena de estas bases, dichos patrones podrían considerarse indicadores de algún tipo de enfermedad, o relaciones de herencia.

Implementación y análisis de los algoritmos

Continuando con la implementación y entendimiento de los algoritmos estudiados se denota la importancia de conocer el comportamiento de estos en el tiempo, dicha medida es útil para el estudio de estos, principalmente en el tema de eficiencia y uso de recursos. Para dicho objetivo es usada la estrategia de “análisis de seis pasos”, además se incluyen algunos autómatas.

Análisis algoritmo naive

Tamaño de los datos

Se asume al tamaño de los datos como la cantidad total de caracteres en una hilera T , es decir el largo de T , denominado n . Para dicho propósito también se considera a m como el tamaño del patrón P a buscar.

Operaciones relevantes

En cuanto a las operaciones de interés se asume el acceso a la hilera “[]” como una operación importante para el análisis.

Ecuación

El algoritmo puede ser separado en los dos ciclos que le componen, en el caso de que el largo del texto fuese $n = 4$ y el del patrón $m = 2$, el primer ciclo iteraría hasta 3 ($n - m + 1$), el cual contiene al segundo que se iterará hasta 2 (m). Es decir, por cada repetición del primer ciclo se ejecutará dos veces el segundo, en este caso serían $3 * 2 = 6$ veces.

Para dicho ejemplo se cuentan 2 ("[]") operaciones por cada iteración del segundo ciclo, el contador de operaciones tendría en valor de $2 * 6 = 12$ veces.

De manera que se deduce que el contador de accesos a lista tomará el valor de la cantidad de veces que itera el primer ciclo por las veces que se ejecutará el segundo ciclo, dando como resultado la totalidad de accesos a la hilera. Lo anterior puede ser descrito como $2((n - m + 1) * m) = 2nm - 2m^2 + 2m$.

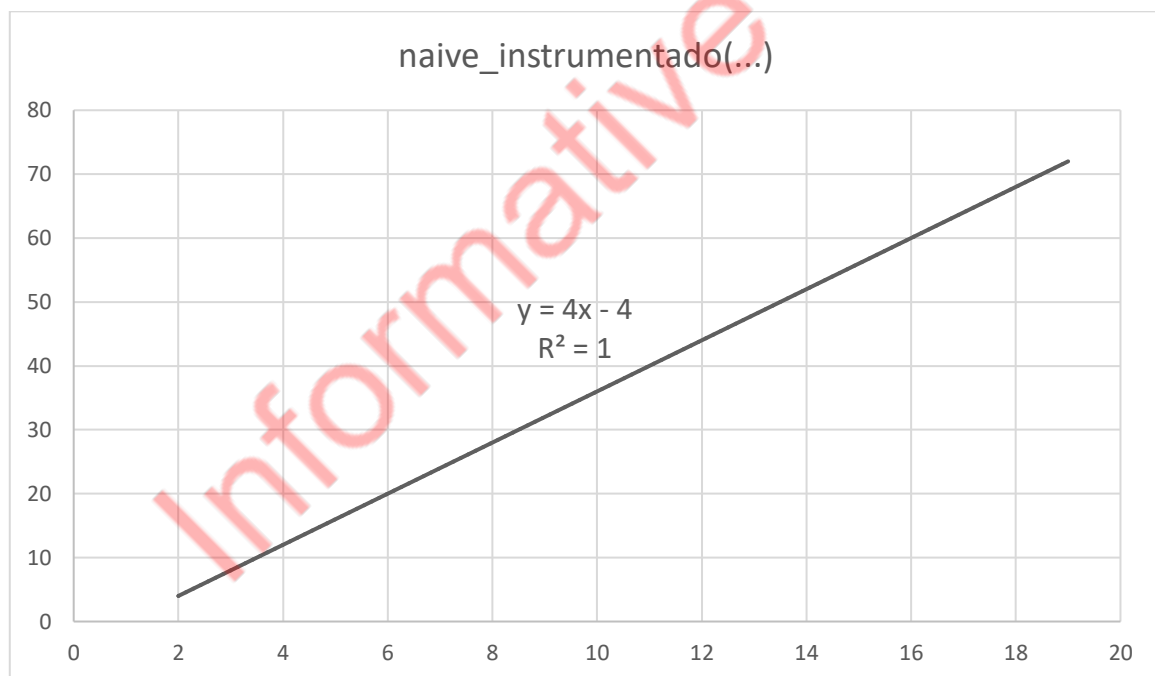
Orden de crecimiento

El orden de crecimiento del algoritmo es $O(nm)$, esto se justifica debido a la manera en que los ciclos *for* están distribuidos.

Instrumentación del código y resultados

El [código instrumentado](#) se anexa, este fue probado de manera que el largo de la hilera T se encontrara en constante crecimiento, mientras que P mantenía su valor de 2. La instrumentación dio como resultado el siguiente gráfico.

Ilustración 4 Gráfico descriptivo del tiempo de corrida SM naive



Elaboración propia

Análisis algoritmo KMP

Tamaño de los datos

Al igual que el algoritmo naive, se asume al tamaño de los datos como el largo de T , denominado n , además de a m como el tamaño del patrón P a buscar.

Operaciones relevantes

Se mantiene a acceso a la hilera “[]” como operación de interés.

Ecuación

En cuanto a la ecuación del algoritmo se toma en cuenta el ciclo que este contiene, dependiente de una variable i la cual será incrementada según n . A partir de ahí se realizarán varios accesos a la hilera, los cuales en caso de no encontrarse en el texto sumarán un total de 4, por cada iteración, es decir que la ecuación es $4n$.

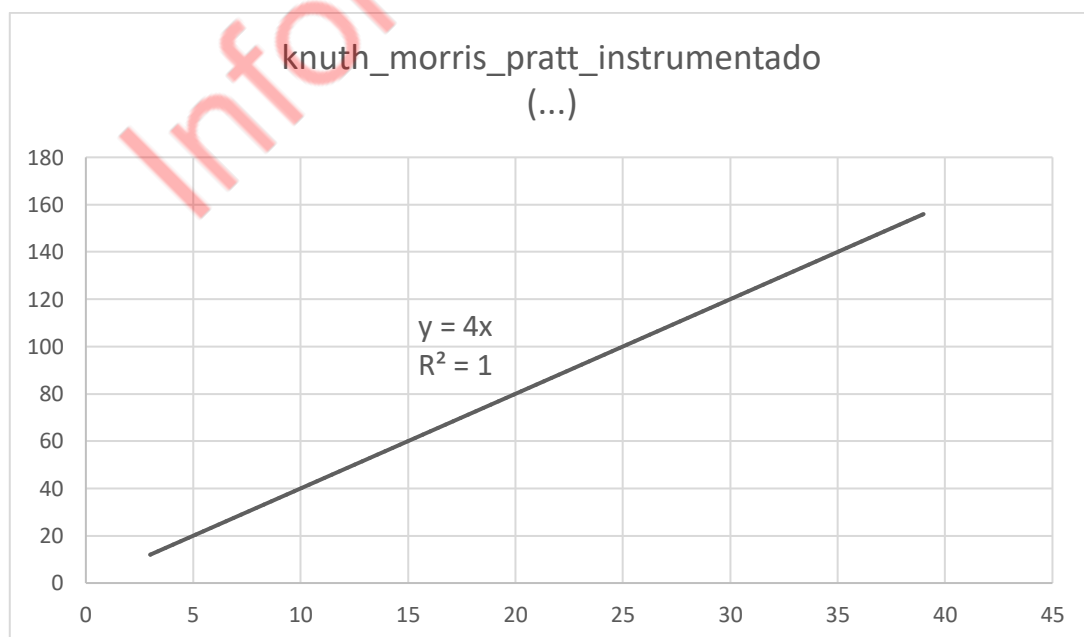
Orden de crecimiento

El orden de crecimiento es $O(n)$, esto debido a que la cantidad de accesos e iteraciones es directamente dependiente al largo de la hilera T .

Instrumentación del código y resultados

Se anexa dicho [código](#), en este se cuenta la cantidad de accesos a la hilera, de esta manera la sumatoria de todos los accesos es finalmente devuelta. El análisis de los datos obtenidos genera el siguiente gráfico.

Ilustración 5 Gráfico descriptivo del tiempo de corrida SM KMP



Elaboración propia

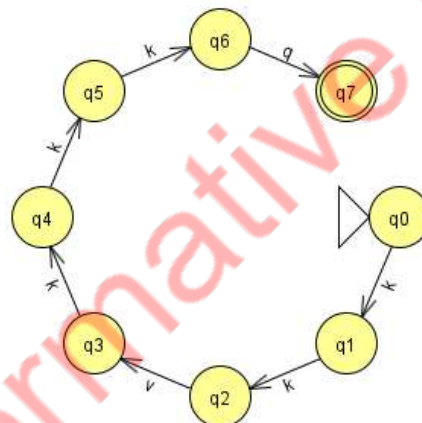
Representación en autómeta

Una representación útil de dicho algoritmo es por medio de un autómeta finito (FA), en este caso se puede realizar un análisis de la hilera que desea ser encontrada. Dicho proceso se realiza colocando los estados correspondientes a todos los elementos del patrón, posteriormente se hace la conexión para un caso simple de aceptación, haciendo uso del vocabulario se puede construir fácilmente una tabla que permite visualizar aquellos estados a los que se debe recurrir en caso de un *mismatch*. En caso de que se presente esa situación el autómeta recurrirá al último estado que presentase el carácter. (Guna, 2014).

Para el ejemplo a desarrollar se recurrirá a:

- Patrón = kkvkkkq
- Vocabulario = q, v, k

Ilustración 6 Autómeta inicial



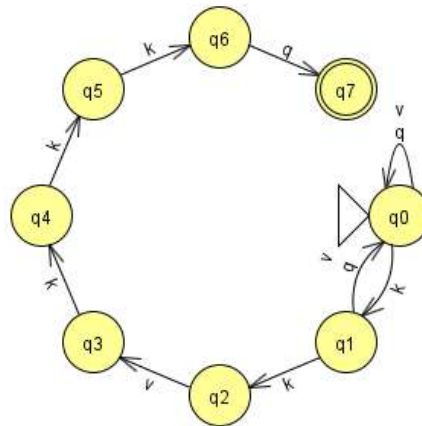
Elaboración propia

Para el FA anterior son analizadas las posibilidades según el número de estados que este posee, de la siguiente manera (Guna, 2014):

Estado	0	1	2	3	4	5	6	7
q	0	0	-
v	0	0	-
k	1	2	-

En las celdas interiores de la tabla se coloca aquel estado al que el autómeta debería posicionarse en caso de que “leyera” un carácter distinto al esperado, en ese caso se realizaría un nuevo arco, como se muestra a continuación:

Ilustración 7 Autómata con nuevos arcos

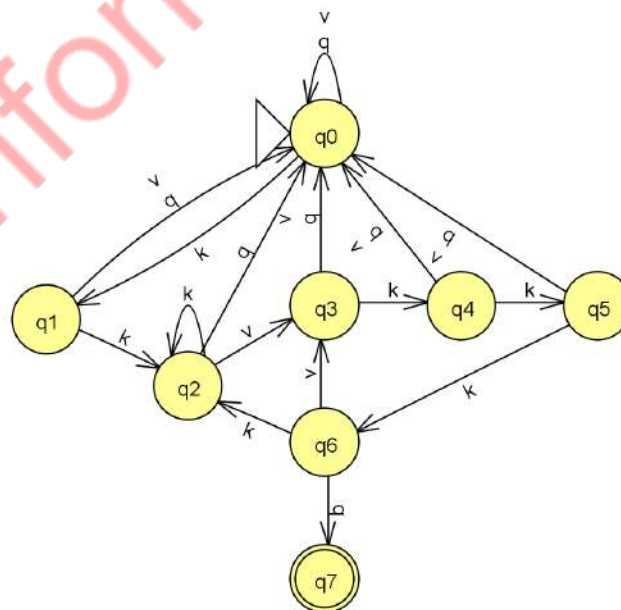


Elaboración propia

En caso de un *mismatch* se partirá desde la posición del estado donde se encuentre leyendo en ese momento.

Estado	0	1	2	3	4	5	6	7
q	0	0	0	0	0	0	7	-
v	0	0	3	0	0	3	3	-
k	1	2	2	4	5	6	2	-

Ilustración 8 Autómata final



Elaboración propia

El autómata anterior reconoce únicamente la hilera deseada, hace uso de la lógica detrás del algoritmo KMP para el reconocimiento de esta, específicamente los saltos que se hacen según los *mismatch*.

Análisis algoritmo Boyer-Moore

Tamaño de los datos

Se denomina al largo de Tn , además de a m como el tamaño del patrón P a buscar. El tamaño de los datos reside principalmente en n .

Operaciones relevantes

Al igual que en casos anteriores el acceso a la hilera " $[]$ " es la operación de interés.

Ecuación

El algoritmo consta de dos ciclos *while*, los cuales siguen el procedimiento descrito, el segundo *while* será ejecutado según el largo de las hileras y las coincidencias que se vayan presentando durante la ejecución, este se ve englobado por otro el cual depende de que una variable c sea menor a la resta de $n - m$. Si se contabilizan los accesos a lista, se obtiene un total de dos accesos para cada iteración del segundo *while*, multiplicado por $n - m$, a lo cual se le suma un acceso dependiente de una comparación entre el valor de j y 0, cuya cantidad de veces que es tomado en cuenta depende del primer ciclo.

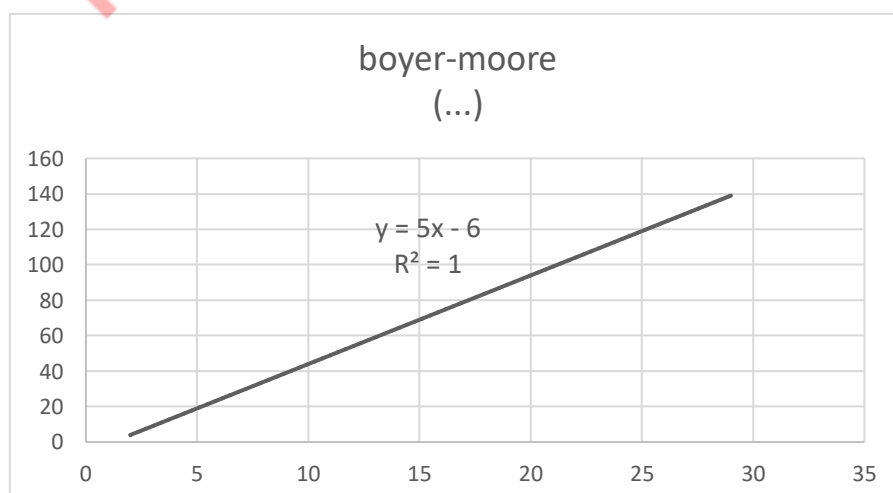
Orden de crecimiento

En este caso se puede indicar que el orden de crecimiento es $O(n)$, debido a su comportamiento lineal en el tiempo.

Instrumentación del código y resultados

El análisis de los datos generados por el [código](#) resulta en el siguiente gráfico.

Ilustración 9 Gráfico descriptivo del tiempo de corrida SM Boyer-Moore



Capítulo III Conclusión

1. Los algoritmos de *string matching* son una potente herramienta para buscar patrones en un texto. Mediante el uso de estos algoritmos, es posible encontrar eficazmente los patrones deseados en un documento de texto, por ejemplo.
2. Existe una gran variedad de algoritmos de comparación de cadenas, cada uno con sus propias ventajas y desventajas. El mejor algoritmo para utilizar depende de las necesidades específicas de la aplicación. Sin embargo, con el algoritmo adecuado, es posible acelerar el proceso de encontrar patrones.
3. Además de la búsqueda de patrones en el texto, los algoritmos de comparación de cadenas también pueden utilizarse en el estudio del ADN. Utilizando estos algoritmos, se pueden encontrar patrones deseados en una secuencia de ADN. Esto puede ser útil en una variedad de aplicaciones, como determinar la función de un gen en particular.

4. APRENDIZAJE PERSONAL :

Referencias

- Agrawal, M. (2020a). *Search Algorithm* [Software]. Recuperado de: <https://www.youtube.com/watch?v=nK7SLhXcqRo>
- Agrawal, M. (2020b, febrero 17). *Naive Algorithm for Pattern Searching | GeeksforGeeks* [Vídeo]. YouTube. Recuperado de: <https://www.youtube.com/watch?v=nK7SLhXcqRo>
- Bari, A. (2018, 25 marzo). *9.1 Knuth-Morris-Pratt KMP "string" Matching Algorithm*. Recuperado de: <https://www.youtube.com/watch?v=V5-7GzOfADQ>
- Deshmukh, N. (2021, March 29). *Boyer moore algorithm for pattern searching - Kalkicode*. Recuperado de: <https://kalkicode.com/boyer-moore-algorithm-pattern-searching>
- garg10may. (2018). *Find Algorithm* [Software]. Recuperado de: <https://stackoverflow.com/questions/41199057/naive-string-search-algorithm-python>
- Gimel'farb, G. (2011). *String Matching Algorithms* [Diapositivas]. University of Auckland. Recuperado de: <https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf>
- Guna, A. (2014, 5 diciembre). *KMP Explained (building the DFA)*. YouTube. Recuperado de: https://www.youtube.com/watch?v=hBXzOq_1yRk
- Hutch. (2004). *Boyer Moore Exact Pattern Matching Algorithms*. Recuperado de: <http://www.movsd.com/bm.htm>
- Jain, B. (2022, 17 enero). *Python Program for KMP Algorithm for Pattern Searching*. GeeksforGeeks. Recuperado de: <https://www.geeksforgeeks.org/python-program-for-kmp-algorithm-for-pattern-searching-2/>
- Knuth, D. E., Morris, Jr., J. H., & Pratt, V. R. (1977). *Fast Pattern Matching in Strings*. SIAM Journal on Computing, 6(2), 323–350. Recuperado de: <https://sci-hub.se/https://doi.org/10.1137/0206024>
- LabTest. (2020, 2 abril). *Secuenciación del ADN | Lab Tests Online-ES*. Recuperado de: <https://labtestsonline.es/articles/secuenciacion-del-adn>
- Langmead, B. (2015, May 19). *ADS1: Practical: Implementing Boyer-Moore*. Youtube. Recuperado de <https://www.youtube.com/watch?v=CT1IQN73UMs>
- m00nlight. (2016). *Python KMP algorithm*. GitHub Gist. Recuperado de: <https://gist.github.com/m00nlight/daa6786cc503fde12a77>
- MedlinePlus. (2022). *¿Qué es el ADN?: MedlinePlus Genetics*. Recuperado de: <https://medlineplus.gov/spanish/genetica/entender/basica/adn/>

NeetCode. (2021, 14 noviembre). *Knuth–Morris–Pratt KMP - Implement strStr() - Leetcode 28 - Python*. YouTube. Recuperado de:

<https://www.youtube.com/watch?v=JoF0Z7nVSrA>

Normas APA. (2019). *Guía Normas APA*. Recuperado de: <https://normas-apa.org/wp-content/uploads/Guia-Normas-APA-7ma-edicion.pdf>

Park, J. (2015). *String Algorithms* [Diapositivas]. Stanford University. Recuperado de: <https://web.stanford.edu/class/cs97si/10-string-algorithms.pdf>

Slade, M. (2014). *Boyer Moore Horspool Algorithm*. Youtube. Recuperado de: <https://www.youtube.com/watch?v=PHXAOKQk2dw>

Informative only

Anexos

Los enlaces apuntan a GitFront, de manera que la visualización de los datos es privada y no listada. Únicamente individuos con dichos enlaces pueden acceder a la información.

Anexo 1

Título: SM naive e instrumentado

Tipo: Interpretación de otro código fuente.

Autores fuente: Meenal Agrawal y garg10may.

Enlace a GitHub:

<https://gitfront.io/r/Isaac-PM/ddihszkNkCt4/matching-en-hileras/tree/Naive/>

Anexo 2

Título: SM KMP e instrumentado

Tipo: Interpretación de otro código fuente.

Autores fuente: Obtención del LPS: NeedCode (2021) y Knuth et al. (1977), Búsqueda en T: Jain (2022), m00nlight (2016) y Knuth et al. (1977).

Enlace a GitHub:

<https://gitfront.io/r/Isaac-PM/ddihszkNkCt4/matching-en-hileras/tree/Knuth-Morris-Pratt/>

Anexo 3

Título: SM Boyer-Moore e instrumentado

Tipo: Interpretación de otro código fuente.

Autores fuente: Deshmukh (2021) y Langmead (2015).

Enlace a GitHub:

<https://gitfront.io/r/Isaac-PM/ddihszkNkCt4/matching-en-hileras/tree/Boyer-Moore/>