Sobre Matching en Hileras

Victor David Coto Solano

Diego Quirós Artiñano

Derek Rojas Mendoza

Universidad Nacional de Costa Rica

EIF-203: Estructuras Discretas Grupo 01-10am

Carlos Loria-Saenz

Ciclo I 2022

Índice

Índice de fíguras

Índice de tablas

Índice de algoritmos

Introducción General

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris...

String Matching

¿Qué es String Matching?

String Matching es cuando se agarra un patrón y se buscan todas las instancias de ese patrón dentro de un texto específico. String es el término que se usan para las cadenas de texto y matching la palabra en inglés que dice que dos cosas concuerdan, en este caso la hilera del patrón con la hilera del texto.

String Matching tiene varios usos en la vida real. Situaciones tan variadas como bases de datos musicales (Faro y Lecroq, 2013) y a la misma vez se puede usar para detección de plagio, forense digital, checkeo de palabras al escribir, detección de intrusión y muchísimos otros casos posibles (*Applications of String Matching Algorithms - GeeksforGeeks* (2020)). Para esta investigación nos vamos a basar en secuencias de ADN como uso del String Matching.

Tipos de String Matching

String Matching se subdivide en dos categorías: exactas y semejantes. Las dos de las categorías tienen sus propios algoritmos diferentes. En esta investigación no solo le daremos enfoque a la categoría de algoritmos de String Matching exactos, sino que nos vamos a analizar 4 algoritmos de este, los cuales serán: Fuerza Bruta (el más básico), Knuth-Morris-Pratt, Boyer-Moore y finalmente el Rabin-Karp. A cada uno se le va a dar un enfoque mayor en la siguiente parte de la investigación, pero en general lo que los destaca es que los primeros tres son con revisión de caracteres y el último utiliza hashing (que convierte uno o varios elementos de entrada a una función en otro elemento).

Algoritmos

Introducción

En esta parte de la investigación se van a analizar los cuatro algoritmos previamente mencionados. Para esto se va a explicar en términos generales como sirve el programa, se va a generar un algoritmo y finalmente se va a analizar este algoritmo con los seis pasos vistos en clase.

Fuerza Bruta (Naive Algorithm)

Introducción al Algoritmo de Fuerza Bruta

El algoritmo de fuerza bruta es un algoritmo que analiza de izquierda a derecha revisando por caracteres si un patrón existe dentro de un texto. Se puede considerar el método más básico de String Matching porque en cada carácter revisa si el patrón se cumple.

Implementación del Algoritmo de Fuerza Bruta

Algoritmo 1 Algoritmo de fuerza bruta

```
1:  procedure FUERZABRUTA((texto, patron))
2:      n ← len(texto)
3:      m ← len(patron)
4:      for i ≤ (n − m) do                ▷ Despues de n − m no puede ser el patron
5:          for j < m do                  ▷ Evalua el patron caracter por caracter
6:              if texto[i + j] ≠ patron[j] then   ▷ Evalua si los caracteres son los mismos
7:                  break
8:              end if
9:          end for
10:         if j = m then                 ▷ Si llega al final entonces existe y devuelve la posición
11:             return i
12:         end if
13:     end for
14:     return -1                         ▷ Si pasa por todo y no encuentra no existe devuelva -1
15: end procedure
```

Análisis del Algoritmo de Fuerza Bruta

Paso 1: Establecer el tamaño n de los datos

Hay dos variables que determinan el tamaño de los datos, estos serían n como la cantidad de caracteres del patrón y m la cantidad del texto. Entonces termina siendo $len(patrón) + len(texto) = n + m$

Paso 2: Determinar las operaciones de interés

Las operaciones de interés del algoritmo son las comparaciones de los caracteres. Estos tendrán valor de 1 cada uno.

Paso 3: Encontrar los casos base

Este algoritmo implica la combinación de las dos variables de tamaño un tiempo de recorrido del patrón y otro del texto.

$$T_{patrón}(0) = 0$$

$$T_{patrón}(1) = 1$$

$$T_{patrón}(n) = 1 + T_{patrón}(n-1)$$

$$T_{texto}(0) = 0$$

$$T_{texto}(1) = 1$$

$$T_{texto}(m) = 1 + T_{texto}(m-1)$$

y esto se junta para que de (porque para cada caracter del texto se evalúan todos los del patrón):

$$T_{FuerzaBruta}(n, m) = T_{texto}(m) * T_{patrón}(n)$$

Paso 4: Evaluando la ecuación recursiva

Como es una suma de unos

$$T_{patrón}(n) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{patrón}(n) = n$$

$$T_{texto}(m) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{texto}(m) = m$$

Con la ecuación que sacamos en el paso anterior y la evaluación de las sub-ecuaciones entonces llegamos a la conclusión de que:

$$T_{FuerzaBruta}(n, m) = m * n$$

Paso 5: O-grande

Con la ecuación de tiempo entonces concluimos que el O-grande del algoritmo es: $O(n * m)$

Paso 6

*instrumentación

Knuth-Morris-Pratt (KMP)

Introducción al Algoritmo de Knuth-Morris-Pratt

Es un algoritmo que hace las comparaciones de izquierda a derecha, este realiza la búsqueda usando información basada en fallos previos obtenidos del patrón, esto se hace en una fase de “procesamiento” que crea una tabla de valores sobre su propio contenido. Esta tabla se crea para ver donde podría darse la siguiente coincidencia sin buscar más de 1 vez los caracteres del texto o cadena de caracteres donde se realiza la búsqueda, el algoritmo es de tiempo O (n + m) siendo n = el tiempo de la fase de “searching” y m = el tiempo de la fase de procesamiento.

La tabla de fallos se encarga de evitar que cada carácter del texto sea analizado más de 1 vez, esto lo logra comparando el patrón consigo mismo para ver que partes se repiten. Este método guarda una lista con números le indican al algoritmo cuando debe devolverse desde la posición actual una vez que el patrón no coincida con el texto.

El texto y el patrón van avanzando simultáneamente mientras ambos coincidan, si una vez coinciden del todo pero la letra siguiente sigue cumpliendo con el patrón, entonces el algoritmo mueve el patrón 1 a la derecha, si no coinciden, entonces el patrón se empieza a devolver para intentar hacerlo coincidir con el texto.

Implementación del Algoritmo de Knuth-Morris-Pratt

Algoritmo 2 Algoritmo de Knuth-Morris-Pratt

```
1:  procedure KNUTH_MORRIS_PUTH((texto, patron))
2:      n ← len(texto)
3:      m ← len(patron)
4:      resultado = False
5:      listaDeIndices = []
6:      tablaDeFallo = [0] * m ▷ Tabla que se va a usar para devoluciones en los procesamientos
7:      i = 0
8:      j = 0
9:      procedure PROCESAMIENTO((patron, m, tablaDeFallo))
10:         longitud = 0                          ▷ longitud del sufijo del prefijo mas largo
11:         i = 1
12:         while i < m do
13:             if patron[i] == patron[longitud] then
14:                 longitud+ = 1
15:                 tablaDeFallo[i] = longitud
16:                 i+ = 1
17:             else
18:                 if longitud! = 0 then
19:                     longitud = tablaDeFallo[longitud − 1]
20:                 else
21:                     tablaDeFallo[i] = 0
22:                     i+ = 1
23:                 end if
24:             end if
25:         end while
26:     end procedure
27:     procedure BÚSQUEDA(())
28:         i = 0
29:         j = 0
30:         while i < n do
31:             if  thenpatron[j] == texto[i]
32:                 i+ = 1
33:                 j+ = 1
34:             end if
35:             if j == m then
36:                 listaDeIndices+ = [i − j]
37:                 resultado = True
38:                 j = tablaDeFallo[j − 1]
39:             else if i < n ∧ patron[j]! = text[i] then
```

```
            if j != 0 then
                j = tablaDeFallo[j - 1]
            else
                i += 1
            end if
        end if
    end while
  end procedure
  return listaDeIndices
end procedure
```

Análisis del Algoritmo de Knuth-Morris-Pratt

Paso 1: Establecer el tamaño n de los datos

El tamaño esta dado por la cantidad de elementos en el texto * la cantidad de elementos en el patron

$$n = len(txt) * len(pat)$$

Paso 2: Determinar las operaciones de interés

Comparaciones entre Pat y Txt y construcción de lps[]
Suponemos el peor caso: todas son iguales a la mas grande de todas
Asumimos que la mas grande vale 1

Paso 3: Establecer la relacion de recurrencia para $T_{KMP}(n, m)$ ***con*** $n = len(txt) \wedge m = len(Pat)$

Se obtuvo que:

$$T_{KMP}(n, m) = m \qquad sin = 0$$
$$= (2 + t_{kmpS}(n - 1)) + (1 + t_{kmpP}(m - 1)) \qquad sin > 0$$

con kmpS = KMP search y kmpP = MKP Processing

Paso 4: Resolver la ecuacion de $T_{KMP}(n, m)$ ***eliminando la recursion***

$$T_{kmp}(n, m) = 2n + m$$

Paso 5: Determinar el orden de crecimiento asintotico de $T_{KMP}(n, m)$

$$T_{kmp}(n, m) \sim O(n + m)$$

con n = el tiempo de la fase de “searching” y m = el tiempo de la fase de procesamiento.

Paso 6

*Instrumentación

Algoritmo de Boyer-Moore

Introducción del Algoritmo de Boyer-Moore

El algoritmo de Boyer-Moore realiza su búsqueda escaneando el patrón deseado desde la posición derecha hasta la posición izquierda. Este utiliza dos heurísticas para completar su

cometido, llamadas ‘Bad Character Rule’ y ‘Good Suffix Rule’. Para ambas heurísticas el algoritmo realiza un “preprocesamiento” del patrón, donde se elaboran dos tablas de interés:

- La tabla BC (Bad Character) basa su elaboración en el desplazamiento necesario para ir desde el carácter que se encuentra más a la derecha (del patrón) hasta la primera ocurrencia de los caracteres específicos que lo componen. Si
- La GS (Good Suffix) se elabora a partir de las coincidencias encontradas entre los sufijos del patrón y los caracteres restantes. Nos da el desplazamiento necesario, desde la izquierda, para encontrar dichas coincidencias. Estas pueden ser exactas (el prefijo aparece exactamente igual) o aproximadas (aparece una parte del sufijo).

Cuando el algoritmo va realizando las comparaciones entre el patrón y el texto y encuentre una discordancia, la decisión entre usar el desplazamiento recomendado por la tabla BC o la tabla GS dependerá netamente de cual de las dos le ofrezca un mayor “salto” o ventaja.

Implementación del Algoritmo de Boyer-Moore

Algoritmo 3 Algoritmo de Boyer_Moore

```
1: procedure BOYER-MOORE((patron, texto))
2:     sizeP = len(P)
3:     sizeT = len(T)
4:     boyerMooreBadChar = [0] * 256          ▷ 256 es el número generalmente aceptado como alfabéto
5:     for 0 ≤ i < sizeP − 1 do
6:         boyerMooreBadChar[ord(patron[i])] = sizeP − i − 1
7:     end for
8:     suff = [0] * sizeP
9:     f = 0
10:    g = sizeP − 1
11:    suff[sizeP − 1] = sizeP
12:    for sizeP − 2 ≥ i > −1 do
13:        if i > g ∧ suff[i + sizeP − 1 − f] < i − g then
14:            suff[i] = suff[i + sizeP − 1 − f]
15:        else
16:            if i < g then
17:                g = i
18:            end if
19:            f = i
20:            while g ≥ 0 ∧ P[g] == P[g + sizeP − 1 − f] do
21:                g− = 1
22:            end while
23:            suff[i] = f − g
24:        end if
25:    end for
26:    boyerMooreGoodSuffix = [sizeP] * sizeP
27:    for 0 ≤ i < sizeP do
28:        if suff[i] == i + 1 then
29:            for 0 ≤ j < sizeP − 1 − i do
30:                if boyerMooreGoodSuffix[j] == sizeP then
31:                    boyerMooreGoodSuffix[j] = sizeP − 1 − i
32:                end if
33:            end for
34:        end if
35:    end for
36:    for 0 ≤ i < sizeP − 1 do
37:        boyerMooreGoodSuffix[sizeP − 1 − suff[i]] = sizeP − 1 − i
38:    end for
39:    i = 0
```

```
    j = 0
    while j ≤ sizeT − sizeP do
        i = sizeP − 1
        while i! = −1 ∧ patron[i] == texto[i + j] do
            i− = 1
        end while
        if i < 0 then
            print(j)
            j+ = boyerMooreGoodSuffix[0]
        else
            j+ = max(boyerMooreGoodSuffix[i], boyerMooreBadChar[ord(T[i + j])] − sizeP + 1 + i)
        end if
    end while
end procedure
```

Análisis del Algoritmo de Boyer-Moore

Paso 1: Establecer el tamaño n de los datos

El tamaño de los datos esta dado por m + n (Siendo 'm' el tamaño del texto y 'n' el tamaño del patrón)

Paso 2: Determinar las operaciones de interés

Las operaciones van a variar dependiendo del módulo. Principalmente, utilizaremos de referencia aquellas relacionadas a comparaciones ('==', ¡'!=') y asignaciones ('=')

Paso 3: Encontrar los casos base

Dividiremos esta sección en Preprocesamiento y Búsqueda, de la forma:

- Preprocesamiento
  - Heurística de caracteres malos (w es el tamaño del alfabeto)

$$T_{HCM}(0) = w \qquad \texttt{si } n = 0$$

$$T_{HCM}(n) = 1 + T_{HCM}(n-1) \qquad \texttt{si } n > 0$$

  - Herística de sufijos buenos

$$T_{HSB}(0) = 0 \qquad \texttt{si } n \leq 1$$

$$T_{HSB}(n) = 1 + (n-1) + T_{HSB}(n-1) \qquad \texttt{si } n > 1$$

- Búsqueda

$$T_{patrón}(0) = 0$$

$$T_{patrón}(1) = 1$$

$$T_{patrón}(n) = 1 + T_{patrón}(n-1)$$

$$T_{texto}(0) = 0$$

$$T_{texto}(1) = 1$$

$$T_{texto}(m) = 1 + T_{texto}(m-1)$$

$$T_{Búsqueda}(n, m) = T_{patrón}(n) * T_{texto}(m)$$

Eso entonces se junta para que la ecuación de tiempo del Boyer-Moore de:

$$T_{Boyer-Moore}(m, n) = T_{HCM}(n) + T_{HSB}(n) + T_{Búsqueda}(n, m)$$

Paso 4: Evaluando la ecuación recursiva

- Preprocesamiento
  - Heurística de caracteres malos

$$T_{HCM}(n) = 1 + 1 + 1 + ... + 1 + w$$

$$T_{HCM}(n) = n + w$$

  - Heurística de sufijos buenos

$$T_{HSB}(n) = 1 + (n-1) + 1 + (n-2) + 1 + (n-3)... + 0$$

$$T_{HSB}(n) = n + (n-1) + (n-2) + (n-3)... + 0$$

$$T_{HSB}(n) = \frac{n * (n+1)}{2} = \frac{n^2 + n}{2}$$

- Búsqueda

$$T_{patrón}(n) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{patrón}(n) = n$$

$$T_{texto}(m) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{texto}(m) = m$$

$$T_{Búsqueda}(n, m) = n * m$$

$$T_{Boyer-Moore}(n, m) = (n + w) + (\frac{n^2 + n}{2}) + n * m$$

Paso 5: O-grande

Dado a los pasos anteriores podemos concluir que el O-grande de Boyer-Moore es $O(w) + O(n^2) + O(n * m) \sim O(n * m)$ Es importante destacar que aunque en el peor de los casos es $n * m$, pero que es considerado el más eficiente de los 4 porque el O-grande normal es $O(\frac{n}{m})$

Paso 6

Algoritmo de Karp-Rabin (o Rabin-Karp)

Introducción del Algoritmo de Karp-Rabin

El algoritmo de Karp-Rabin es el único de los que estamos analizando que usa el método de hashing. Para esto utiliza un número primo alto y una formula para sacar el hash value del patrón y de las secciones del texto que se están evaluando. Al encontrar un valor de hash que coincida entonces va a evaluar si la sección que está evaluando en el momento es igual que el patrón, carácter por carácter. Esta es la razón por la cual se utiliza un primo grande, porque al depender de residuos (primo todos los números van a salir modulo si mismo porque el primo solo es modulo 0 con si mismo) si el número evaluando es más grande que el primo entonces puede dar el mismo valor para lo que no debería.

Implementación del Algoritmo de Karp-Rabin

Algoritmo 4 Algoritmo de Karp-Rabin

procedure KARP_RABIN((patron, texto))
    $n = len(patron)$
    $m = len(texto)$
    $d = 256$ ▷ Este es el alfabeto defecto que sale en varios analisis (caracteres alfabeto inglés)
    $q = 33554393$ ▷ Cualquier número primo sirve, pero preferiblemente alto porque los pequeños solo hacen que el algoritmo corra como fuerza bruta porque más hashes concuerdan
    $h = d^{m-1} mod(q)$
    $ValorHashPatron = 0$
    $ValorHashVentanaTexto = 0$
    $listaIndices = []$
    for i = 0 <n **do**
        $ValorHashPatron = (d * ValorHashPatron + patron[i]) mod(q)$ ▷ Esta operación sirve con un abecedario normal como tabla ascii, para usarlo en Python el accesor se tiene que meter como parametro de ord()
        $ValorHashVentanaTexto = (d * ValorHashVentanaTexto + texto[i]) mod(q)$
    end for
    j = 0 ▷ definirla afuera para poder usarla dentro del for sin tener que redefinirla cada vez que empiece el for otra vez
    for $i = 0 \leq m - n$ **do**
        if ValorHashPatron == ValorHashVentanaTexto **then** ▷ Solo hacer fuerza Bruta cuando los valores hash concuerdan
            for $j = 0 < n$ **do**
                if $patron[j] != texto[i + j]$ **then** ▷ Si la fuerza bruta se incumple salga
                    break
                else
                    $j += 1$
                end if
            end for
            if $j == n$ **then**

```
            listaIndices+ = [i]          ▷ Al llegar al final de la fuerza bruta regista el indice
        end if
    end if
    if i < m − n then
        ValorHashVentanaTexto = (d∗(ValorHashVentanaTexto−texto[i]∗h)+texto[i+
n])mod(q)
        if ValorHashVentanaTexto < 0 then    ▷ GeeksForGeeks recomienda en caso de
que den hashes negativos
            ValorHashVentanaTexto+ = q
        end if
    end if
  end for
  return listaIndices
end procedure
```

Análisis del Algoritmo de Karp-Rabin

Paso 1: Establecer el tamaño n de los datos

Las dos variables que varían el tamaño de datos es la cantidad de caracteres del patrón n y la del texto m. Entonces termina siendo $len(patrón) + len(texto) = n + m$

Paso 2: Determinar las operaciones de interés

Las operaciones de interés son las comparaciones tanto del valor de hash como el carácter y van a contar como 1.

Paso 3: Encontrar los casos base

El algoritmo tiene tres sub-ecuaciones el de tiempo que tarda comparar los caracteres de patrón, los caracteres de texto y las comparaciones de los hashes.

$$T_{patrón}(0) = 0$$

$$T_{patrón}(1) = 1$$

$$T_{patrón}(n) = 1 + T_{patrón}(n-1)$$

$$T_{texto}(0) = 0$$

$$T_{texto}(1) = 1$$

$$T_{texto}(m) = 1 + T_{texto}(m-1)$$

$$T_{hashes}(0) = 0$$

$$T_{hashes}(1) = 1$$

$$T_{hashes}(m) = 1 + T_{hashes}(m-1)$$

$$T_{Karp-Rabin}(n,m) = T_{hashes}(m) + T_{texto}(m) * T_{patrón}(n)$$

Se hace de esta manera porque siempre se evalúan los hashes y después por aparte se hace un fuerza bruta de la sección.

Paso 4: Evaluando la ecuación recursiva

$$T_{patrón}(n) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{patrón}(n) = n$$

$$T_{texto}(m) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{texto}(m) = m$$

$$T_{hashes}(m) = 1 + 1 + 1 + 1... + 1 + 1$$

$$T_{hashes}(m) = m$$

$$T_{Karp-Rabin}(n,m) = m + m * n$$

Paso 5: O-grande

Dado a lo que evaluamos en los últimos pasos se sabe que O-grande es
$O(m) + O(m * n) \sim O(m * n)$
Es importante notar que ese tiempo ocurre cuando todos los caracteres son lo mismo entonces tiene que evaluar todos los caracteres. El tiempo normal es $O(m) + O(cn) \sim O(m + n), c$ constante.

Paso 6

*instrumentación

ADN

Importancia del String Matching con el ADN

El Ácido desoxirribonucleico o ADN es un ácido nucleico compuesto de 4 nucleótidos, estos son la adenina (A), timina (T), guanina (G) y citosina (C), además de estos 4 nucleótidos, existe un 5 que es el uracilo (U), pero este se encuentra en el ARN reemplazando a la timina (T). Este contiene las instrucciones y la información genética usadas en el desarrollo y funcionamiento de todo ser vivo, incluyendo algunos virus, además, es el responsable de la herencia genética,

como el color de ojos, el color de piel, el cabello y todo tipo de cosas que podemos heredar de nuestros padres o antepasados, sin embargo, también es responsable de heredar ciertas enfermedades genéticas, como por ejemplo, la fibrosis quística, la hemofilia, la enfermedad de Huntington, etc.

Con esto se buscar una base nitrogenada específica, hicimos pruebas con los cuatro algoritmos con una secuencia de ADN aleatoria generada en este sitio, y le corrimos con timeit.repeat()

Para efectos de esta investigación, hemos decidido utilizar la enfermedad de Huntington como ejemplo de string matching en una cadena de ADN.

La enfermedad de Huntington es una enfermedad hereditaria que da la instrucción al cuerpo de producir una proteína llamada Huntingtina(HTT). Aunque se desconoce la función de dicha proteína, se cree que juega un papel importante en las neuronas. Esta enfermedad es provocada por una mutación en un segmento del ADN conocido como una repetición del trinucleótido CAG (citosina, adenina y guanina). Este segmento CAG normalmente se repite de 10 a 35 veces en personas sanas, sin embargo, en personas con la enfermedad de Huntington, dicho segmento se repite de 36 a as de 120 veces.

Como se puede ver en la siguiente imagen:

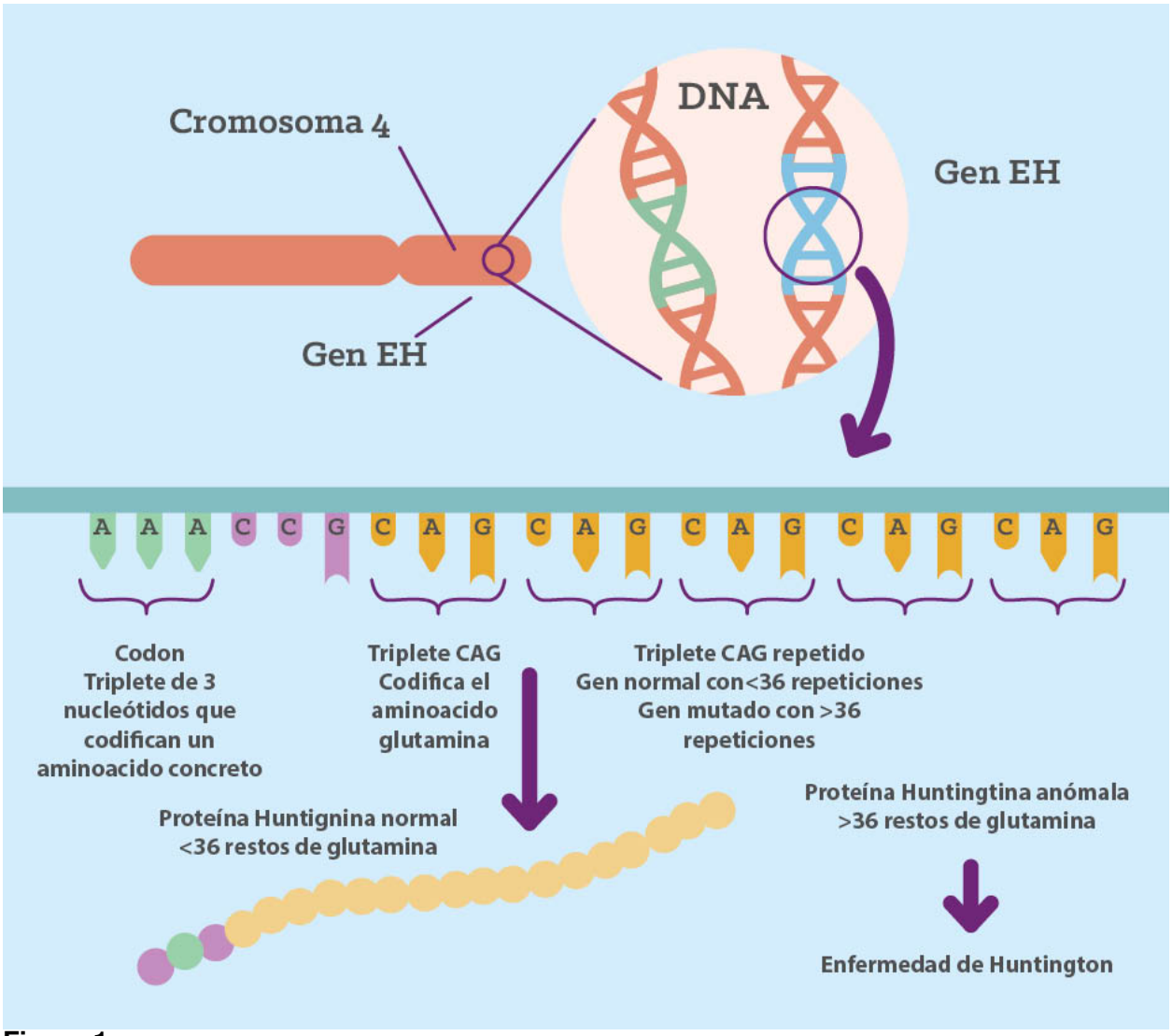

Figura 1
Causes de Huntington

Para efectos del string matching, vamos a comprobar cuantas veces se repite el segmento CAG en la cadena de ADN para comprobar si una persona padece de esta enfermedad.

Conclusión

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris...

Referencias

Applications of String Matching Algorithms - GeeksforGeeks. (2020, septiembre). Descargado de `https://www.geeksforgeeks.org/applications-of-string-matching-algorithms` ([Online; accessed 7. May 2022])

Faro, S., y Lecroq, T. (2013). The exact online string matching problem: A review of the most recent results. *ACM computing surveys*, *45*(2), 1–42.