UNIVERSIDAD AUTÓNOMA DE GUERRERO

UNIDAD ACADÉMICA DE CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN

MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN

TESIS:

ANÁLISIS DEL ALGORITMO DE LEVENSHTEIN
UNA APLICACIÓN SOBRE LA WEB

QUE PRESENTA:

IRVING JAIME SALGADO LINARES

PARA OBTENER EL GRADO EN:

MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

A S E S O R:

M.C.C. Roberto Rosario Cruz

C O A S E S O R:

M.I. Ricardo Peña Galeana

Acapulco Guerrero, Octubre de 2015

Agradecimientos

A mi Eterno Creador…

A mi amada Esposa, por su lucha constante, a mis amados hijos, mi pequeño gran Samurái por haber peleado grandes batallas a mi lado y por la dicha de estar conmigo una vez más, junto con su hermana, mi gran Alma Vieja única e incomparable como nadie, por la dicha de saber que algo grande tendrás que hacer...

A mis cabras locas por su gran apoyo incondicional, y a la varona mentora de mis días, por su gran amor y haberme enseñado y luchar a ritmos de tambor batiente día tras día, con gran entereza valentía…

A mis mentores Ricardo Peña Galeana, Roberto Rosario Cruz y por supuesto a Norma Ivone Peña Galeana, por haberme guiado durante la culminación de esta gran aventura y que gracias a ellos logré alcanzar uno de los anhelos de mi vida…

Infinitamente gracias...!

Vehemente la inspiración llegó…

En algún momento de mi vida, me propuse estudiar un grado más del que tuve hace algunos años atrás, y en primera instancia me dije que, “que osado sería si me aventurara con valentía”, pero yo sabía que eso no iba a pasar, de pronto la voz de mi conciencia me susurró, que era importante estudiar, porque para nosotros era la única forma de poder avanzar y progresar, ya que por alguna razón así nos envió el Eterno en esta vida terrenal, sin lugar a dudas a aprender a vivir y a evolucionar, charlando con un buen amigo, - que Phd es -, me comento que él tenía interés de que yo estudiara otra vez, fue cuando le dije que no, y él insistió tanto que ni más ni menos me convenció. Llego el momento de empezar, y con muchos trastabilleos empezamos a avanzar, y en el transeúnte del camino, aprendimos a analizar, a abstraer y hasta a racionalizar, - uff cosa difícil esa la de aprender a pensar - , pasado el tiempo y después de mucho batallar, a ritmos de tambor batiente, tal y como me lo enseño una mujer valiente desde que aprendí a caminar y de su mano claro está, llego el momento de poder materializar todas aquellas ideas, anhelos y deseos en algo que quedara para la posteridad, traspolar todos aquellos conocimientos adquiridos en un papel, en un pedestal, en una pared o en lo que fuera, qué más da, y ese momento ha llegado llenándome de felicidad, quizá nunca llegue a tener los millones, ni nadie me conozca más allá de lo que yo pueda imaginar, pero esto nació de un mero deseo por amor al arte y a la humanidad, y aquí se los dejo señores hecho toda una realidad…

Contenido

Agradecimientos 1
Lista de Figuras 4
Lista de Tablas 5
Lista de Algoritmos 6
Resumen 7
Capítulo I. Introducción 8
1.1 Estado del arte algoritmos sobre cadenas 10
1.2 Principales áreas de aplicación de los algoritmos de Coincidencia de Cadenas 13
1.3 Correspondencia de Cadenas 18
1.4 Conceptos Básicos 21
1.5 Distancias de Edición 24
Capítulo II. La Distancia Levenshtein 29
2.1 Coincidencia de Patrones 30
2.2 Coincidencia de Cadenas Aproximada 36
2.3 El algoritmo de Levenshtein 39
Capítulo III. Levenshtein sobre la Web 53
3.1 Planteamiento de la aplicación (análisis de la métrica) 55
3.2 El núcleo de la aplicación. 58
3.3 ¿Qué hace *jQuery*? 59
3.4 Desarrollo de la aplicación basada en la Web 61
3.5 Conclusiones y Trabajos futuros 83
Bibliografía 85

Lista de Figuras

1.3 Diferentes ejemplos de Dobletes..... 20
1.5 Proporciona la distancia de edición entre la cadena “string" y un número de otras cadenas..... 27
1.5.1 Ilustración de la Distancia de Edición para la secuencia “string”..... 28
2.1 Coincidencia de patrones Exacta (Pattern Matching)..... 31
2.1.1 Ejemplo del comando grep (implementación del algoritmo de pattern matching)..... 32
2.2 La Alineación de dos secuencias de ADN..... 36
2.2.1 Ejemplo del comando agrep, implementación del algoritmo de Approximate Pattern Matching..... 37
2.3 La Alineación de dos secuencias de ADN que muestran las operaciones de cambios, inserciones (“-“ en la línea superior) y eliminación (“-“ en la línea de abajo)..... 39
2.3.1 Cálculo Matricial Distancia de Levenshtein..... 40
2.3.2 Las tres operaciones básicas del algoritmo de Levenshtein..... 41
2.3.3 La ruta corresponde a la secuencia de operaciones de edición: inserción (a), eliminación (b), inserción (b), inserción (b), cambio (c,a)..... 44
2.3.3.1 Ejemplo de un rejilla grafo acíclico dirigido (DAG)..... 44
2.3.3.2 Resultado de la ejecución del algoritmo de Levenshtein..... 49
3.1 Ejemplo de la Desigualdad Triangular..... 57
3.4 Pantalla principal de la aplicación..... 74

Lista de Tablas

1.3 Ejemplos de doblete y su transformación 19
2.3 Las tres operaciones para encontrar el valor de la celda $min[i, j]$ 45
2.3.1 Implementación del algoritmo de Levenshtein 48
2.3.2 Los pasos y descripción del algoritmo de Levenshtein de la función arriba implementada 49
2.3.3 Paso 3 al 6; Cuando $\mathrm{x} = 1; \mathrm{y} = 1$. 50
2.3.3.1 Paso 3 al 6; Cuando $\mathrm{x} = 1; \mathrm{y} = 2$ 50
2.3.3.2 Paso 3 al 6; Cuando $\mathrm{x} = 1; \mathrm{y} = 3$.. 50
2.3.3.3 Paso 3 al 6; Cuando $\mathrm{x} = 1; \mathrm{y} = 4$. 51
2.3.3.4 Paso 3 al 6; Cuando $\mathrm{x} = 1; \mathrm{y} = 5$ 51
2.3.3.5 Paso 3 al 6; Cuando $\mathrm{x} = 1; \mathrm{y} = 6$ 51
2.3.4 Iteración de x = 2 52
2.3.4.1 Iteración de x = 3. 52
2.3.4.2 Iteración de x = 4. 52
2.3.4.3 Iteración de x = 5 52
2.3.4.4 Iteración de x = 6. 52
2.3.4.5 Resultado Final [x, j] = 2. 52

Lista de Algoritmos

2.1 La implementación del algoritmo KMP 35
2.3 Implementación de la función del algoritmo de Levenshtein 42
2.3.1 Implementación en C# del algoritmo de Levenshtein 45
3.4 Script funciones.js 61
3.4.1 Archivo HTML5 buscapalabra.html 74

Resumen

La búsqueda y coincidencia de nombres de personas o patrones es el núcleo de un creciente número de aplicaciones que van desde: el texto y minería de datos, la recuperación y extracción de la información en los motores de búsqueda, la duplicación y los sistemas de vinculación de datos.

Las variaciones y errores en los nombres generan la problemática de la coincidencia o correspondencia de cadenas y por ende las técnicas de aproximación de coincidencias de cadenas basadas sobre la codificación fonética o búsqueda de cadenas pueden ser aplicadas.

Por otro lado la variación de los nombres puede ser un problema importante para la identificación y búsqueda de personas, la coincidencia de nombres supone a priori que el nombre registrado escrito en un alfabeto refleja la identidad fonética de dos nombres o algún error de transcripción en la copia de algún nombre previamente grabado.

Las variaciones son particularmente marcadas en los datos que de forma automática son extraídos de las bases de datos, documentos semi - estructurados o no estructurados, haciendo que la tarea de emparejamiento sea esencial para la integración de la información.

El objetivo de este trabajo es presentar los algoritmos sobre coincidencia sobre cadenas, conjunto de cadenas, cadenas extendidas y expresiones regulares así como los métodos de búsqueda existentes aproximados y por ende presentar a detalle los algoritmos más prácticos y en especial el algoritmo de Levenshtein.

Por lo práctico nos referimos a que son eficaces en la práctica y que un programador normal puede desarrollarlos en pocas horas. Afortunadamente, estos criterios suelen coincidir en la correspondencia de cadenas. Por lo que nos centraremos en la búsqueda de forma lineal, lo que significa que no tenemos que construir estructuras de datos en las cadenas o textos.

Capítulo I. Introducción.

Un algoritmo es una descripción de cómo resolver un problema específico de tal manera, de que quienes analizan y desarrollan un determinado procedimiento para resolver alguna problemática, pueden seguir la descripción del modelo paso a paso, para llegar a la solución del problema.

La palabra algoritmo se deriva del nombre de un matemático persa, *Mohammed ibn Musa al-Khowarizmi*, quien vivió alrededor de los años 780 a 850, su trabajo en los algoritmos desarrollo el uso de los números arábigos y la notación decimal, también introdujo el uso de la palabra *al-jabr* o algebra propiamente dicho en las matemáticas modernas.

Nosotros utilizamos tanto las matemáticas como los algoritmos de forma inherente día con día, por ejemplo una receta para un pastel es un algoritmo, la mayoría de los programas o *software* desarrollado actualmente, se componen de algoritmos. Inventar algoritmos elegantes – *algoritmos que son muy simples y al mismo tiempo complejos, y requieren de los pasos menos posibles* – es uno de los principales retos dentro del ámbito de la programación.

Actualmente las cadenas de texto son uno de los objetos más básicos de la ciencia de la computación, una cadena es una secuencia finita de caracteres que están extraídos de un alfabeto. Por ejemplo una cadena binaria y una cadena de *ADN* contienen caracteres tomados del alfabeto *{0,1} y { 'a', 'c', 'g', 't' }.*

El campo de los algoritmos informáticos ha crecido desde principios de la década de los 60's, cuando los primeros programadores de computadoras comenzaron a prestar atención a la ejecución y estabilidad de los programas. Los limitados recursos de las computadoras en aquella época dieron lugar a un impulso adicional para la elaboración de algoritmos mucho más eficientes y con una mayor estabilidad.

La idea intuitiva más general de un [1]algoritmo se puede considerar como un procedimiento computacional bien definido, que toma algún valor o conjunto de valores, como entrada y produce cierto valor, o un conjunto de valores como salida, por lo tanto un algoritmo es una secuencia de instrucciones que transforman la entrada en una salida, todo esto a través de una ejecución sistemática de instrucciones.

Un ser humano con un lápiz y papel sería capaz de hacer esto, pero los seres humanos son generalmente lentos, comenten errores y prefieren no llevar a cabo un trabajo repetitivo. Una computadora es menos inteligente, pero puede llevar a cabo simples pasos de forma rápida y fiable.

Por lo que el diseño y análisis de algoritmos son esenciales y de una importancia fundamental en el campo de la ciencia de la computación, tal y como lo especificó el PhD. Donald E. Knuth con su frase célebre: *"Computer science is the study of algorithms".*

El trabajo presentado y desarrollado en esta tesis está enfocado al análisis y la construcción de algoritmos que abordan los problemas de la comparación de secuencias sobre cadenas.

.

[1] De acuerdo con el *American Heritage Dictionary,* la palabra Algoritmo es derivada del matemático persa Mohammed ibn Musa al-Khowarizmi (780 - 850), un musulmán matemático quien trabajo introduciendo los conceptos algebraicos y números arábigos.

1.1. Estado del arte algoritmos sobre cadenas

La ciencia es lo que nosotros entendemos lo suficientemente bien como para aplicarlo en una computadora o de alguna otra forma, el arte es todo lo que hacemos. Durante los últimos años una importante parte de las matemáticas ha sido transformada de un arte a una ciencia.

Por decirlo de alguna manera ya no tenemos que tener un brillante conocimiento con la finalidad de evaluar las sumas de coeficientes binomiales y de muchas fórmulas similares que surgen con frecuencia en la práctica, ahora podemos seguir un procedimiento mecánico y descubrir las respuestas totalmente sistemáticas.

Uno de los temas de mayor importancia del siglo pasado ha sido el reemplazo cada vez mayor del pensamiento humano por los programas de computadoras que sin lugar a dudas están desarrollados a través de algoritmos. Áreas enteras de negocios, científicas, médicas y actividades gubernamentales son ahora computarizadas, incluidos aquellos sectores en que los seres humanos teníamos la responsabilidad del razonamiento y era exclusivo de nosotros.

Los problemas con la coincidencia de cadenas van desde la simple tarea de buscar un texto único para una cadena de caracteres, a la búsqueda en una base de datos de un sinfín de ocurrencias aproximadas, el problema de la coincidencia de cadenas puede ser entendido como: la forma de encontrar una cadena con alguna propiedad dentro de una secuencia determinada de símbolos. Este es uno de los problemas más antiguos y generalizados de la ciencia de la computación, las aplicaciones que requieren algún tipo de correspondencia de cadenas se pueden encontrar prácticamente en todas partes.

Sin embargo, durante los últimos años hemos sido testigos de un gran aumento sumamente dramático en el interés por los problemas con cadenas, especialmente en las comunidades de rápido crecimiento de recuperación de la información y la biología computacional.

De manera particular, la correspondencia de cadenas es bien conocida por ser susceptible de enfoques que van desde lo teórico a lo extremadamente práctico, las soluciones teóricas han dado lugar a importantes logros algorítmicos, pero estos raramente son útiles en la práctica: un hecho bien conocido es que las ideas más simples funcionan mejor en la práctica.

Dos ejemplos típicos son el famoso algoritmo de Knuth-Morris-Pratt, algoritmo que en la práctica es dos veces más lento que el método de fuerza bruta, y el algoritmo de Boyer–Moore. En los libros más actuales sobre los algoritmos de texto, la parte de la correspondencia de cadenas cubre sólo los algoritmos teóricos clásicos, hay tres razones para ello.

En primer lugar los algoritmos prácticos son muy recientes, el más antiguo debe de tener al menos algunas décadas de antigüedad, algunos estudios recientes son muy nuevos para aparecer en la literatura o en los libros.

Estos algoritmos se basan generalmente en las nuevas técnicas, tales como el paralelismo de bits, las cuales han aparecido con la reciente generación de computadoras.

La segunda razón es que en esta área los logros teóricos se separan de las ventajas prácticas, la comunidad que trabaja en este tipo de algoritmos está interesada en algoritmos teóricamente atractivos, esto es, aquéllos que lograron las mejores complejidades y contienen complicados conceptos algorítmicos, el desarrollo de las comunidades se centra exclusivamente sobre algoritmos conocidos por ser rápidos en la práctica.

Y tercero, sólo en los últimos años han surgido nuevos algoritmos que combinan los aspectos de la teoría y la práctica como por ejemplo el algoritmo *BNDM, Backward-Dawg-Matching (BDM)* presentado por Crochemore y otros autores, utiliza un autómata de sufijo, o sea un autómata que describe la cadena de manera inversa, para reconocer cuando una secuencia de caracteres dada es factor de la cadena, por lo que el resultado ha sido una nueva tendencia a los algoritmos de coincidencia de cadenas rápidos y robustos.

Estas razones hacen que sea extremadamente difícil encontrar el algoritmo correcto, si uno no está en dicho campo, los algoritmos correctos existen, pero sólo un experto puede encontrarlos, reconocerlos y utilizarlos. Consideremos el caso de los profesionales del software, biólogos computacionales, investigadores o estudiantes que no participan directamente en el campo y se enfrentan a un problema de búsqueda de texto.

Por lo que se ven forzados a cavar en docenas de artículos, la mayoría de ellos de valor teórico, y muy complicados de implementar, por último se pierden en un mar de opciones, sin los antecedentes necesarios para decidir cuál es el mejor, por lo que los resultados típicos de esta situación son:

- Se decide poner en práctica el enfoque más simple, que, cuando está disponible se obtiene un rendimiento muy pobre y afecta la calidad global del desarrollo.
- Hacer una elección (normalmente desafortunada) e invertir mucho trabajo y tiempo en su implementación, sólo para obtener un resultado que en la práctica es tan malo como un enfoque ingenuo o peor aún.

1.2 Principales áreas de aplicación de los algoritmos de Coincidencia de Cadenas

La primera referencia de este problema, nos remonta al inicio de los años sesentas y setentas, donde la problemática aparece en un gran número de diferentes campos, en aquellos tiempos la principal motivación para este tipo de búsqueda llegó a ser la biología computacional, procesamiento de señales y recuperación de texto, las cuales siguen siendo la más grandes área de aplicación [NAV04].

El problema de la correspondencia de cadenas es que hay dos cadenas de texto; una es el texto *T [1…n]* la cadena principal que se da y la otra cadena es la cadena a buscar *P [1…m],* es decir es la cadena dada que va a ser comparada con la cadena principal dada *[m<=n].* La correspondencia de cadenas se encuentra de forma variable en un sinfín de aplicaciones, como esquemas de bases de datos, sistemas de redes [SINGAR12], procesadores de palabras y comparación de ADN.

Existen dos técnicas principales de correspondencia de cadenas, una es la correspondencia de cadenas exacta, como los algoritmos de *BruteForce*, *Boyer-Moore, Knuth-Morris-Pratt* [LEE2004], por mencionar algunos, y otra es la correspondencia aproximada como los algoritmos *Levenshtein o Edit Distance [LEV1966],* Hamming Distance y Longest Common Subsequence Distance [SOPIN 2007].

Algunas de las aplicaciones son los editores de texto, consultas en las bases de datos, en la bioinformática y la química-informática, sistemas de detección de instrucción de redes, correspondencia de patrones de ventana amplia (correspondencia de grandes cadenas), la recuperación de contenido musical, comprobador de sintaxis del lenguaje, secuencias de correspondencia de DNA, bibliotecas digitales, motores de búsqueda y muchas otras aplicaciones [SINGAR12].

Esta área se espera que crezca aún más debido a la creciente demanda de velocidad y eficiencia que proviene de la biología molecular, la recuperación de información, reconocimiento de patrones, la compilación, la compresión de datos, análisis de programas y la seguridad [SINGAR12]. Las aplicaciones prácticas de los algoritmos son ubicuas, e incluyen los siguientes ejemplos (por mencionar algunos):

Biología Computacional.

Las secuencias de ADN y de proteínas pueden ser vistas como textos largos sobre alfabetos específicos (como por ejemplo {*A, C, G,T*} en ADN), estas secuencias representan el código genético de los seres humanos. Buscar secuencias específicas sobre estos textos apareció como una operación fundamental para los problemas tales como la comparación del ADN a partir de piezas obtenidas por los experimentos, buscando características dadas en las propias cadenas de ADN, o determinar qué tan diferente son las dos secuencias genéticas, esto fue modelado como la búsqueda de coincidencias en un texto.

Sin embargo, la búsqueda exacta era de poca utilidad para esta aplicación, ya que los patrones rara vez coinciden exactamente con el texto: las medidas experimentales tienen errores de distintos tipos e incluso las cadenas correctas pueden tener pequeñas diferencias, algunas de ellas significativa debido a las mutaciones y alteraciones evolutivas, encontrar cadenas de ADN muy similares a los solicitados representan resultados significativos.

La biología computacional, desde entonces, evolucionó y se desarrolló a pasos agigantados, con un especial empuje en los últimos años debido al proyecto del "genoma" que tiene como objetivo la decodificación completa del ADN y sus aplicaciones potenciales.

Recuperación de Texto.

El problema de corregir errores ortográficos en el texto escrito es bastante antigua, tal vez la más antigua aplicación potencial para correspondencia de cadenas aproximadas. Podríamos encontrar referencias de los años veinte y tal vez hay otros más antiguos. Desde los años setenta la correspondencia de cadenas aproximadas es una de las más populares herramientas para hacer frente a este problema [DAM63].

Hay muchas áreas donde este problema aparece, y la recuperación de información (IR) es uno de los más demandantes, la recuperación de la información se basa en encontrar la información relevante en una colección de texto sumamente larga y la correspondencia de cadenas es una de las herramientas básicas para este tipo de problema.

Pero desafortunadamente la coincidencia clásica de cadenas no es suficiente, porque las colecciones de texto son cada vez más largas, por ejemplo la indexación pública (en el año de 90's) de la Word Wide Web contenía alrededor de 800 millones de páginas web y abarcaba alrededor de 6 terabytes de datos de texto en unos 3 millones de servidores [LawGil99].

Procesamiento de Señales.

Otra motivación temprana vino del procesamiento de señales y reconocimiento de voz, donde el problema general es determinar, dado una señal de audio, un mensaje de texto que está siendo transmitida [NAV04]. El surgimiento de la música digital a través de Internet requiere nuevos métodos de recuperación de información adaptados a las características y necesidades específicas.

Mientras que la recuperación de la música basada en la información de texto, como el título, los compositores, o la clasificación temática, se ha implementado en muchos sistemas existentes, la recuperación de una pieza de música basada en el contenido musical, en especial en un fragmento incompleto e imperfecto de la pieza musical deseada, aún no se ha explorado a fondo, por lo que otra de las áreas de reciente y de gran motivación proviene del procesamiento de señales.

Una de las mayores áreas trata con el reconocimiento de voz, en donde el problema general es determinar, dada una señal de audio, un mensaje de texto que se está transmitiendo, incluso los problemas tales como la simplificación de discernir una palabra de un pequeño conjunto de alternativas es compleja, ya que partes de la señal se pueden comprimir en tiempo, partes de un discurso que no pueden ser pronunciadas, etc., por lo que una correspondencia perfecta es prácticamente imposible.

Otro problema es la corrección de errores, la transmisión física de señales tiene tendencia a producir errores, para asegurar una transmisión correcta sobre cierto canal físico, es necesario ser capaz de recuperar el mensaje correcto después de una posible modificación (error) producida durante la transmisión. La probabilidad de dichos errores es obtenida del procesamiento teórico de la señal y utilizada para asignar un costo para ellos.

En este caso nosotros no podemos aún conocer que es lo que estamos buscando, sino que sólo queremos un texto que sea correcto (de acuerdo al código corrector de errores utilizado) y lo más cercano el mensaje recibido. A pesar que esta área no se ha desarrollado mucho con respecto a la búsqueda aproximada, tiene generada la medida más importante se similitud, conocida actualmente como *la distancia de Levenshtein* también llamada “distancia de edición”.

El procesamiento de señales es un área muy atractiva actualmente, el campo de rápida evolución de las bases de datos multimedia exige la capacidad para la búsqueda por contenido en imagen, audio y video, que son posibles aplicaciones potenciales para la coincidencia de cadenas aproximada.

La coincidencia de cadenas es uno de los mayores problemas en el área de la seguridad de la red y de muchas otras áreas, el aumento de la velocidad y del tráfico de la misma pueden causar que los algoritmos existentes lleguen a provocar un cuello de botella en el rendimiento.

Por lo que es de suma importancia desarrollar más y mejores algoritmos de coincidencia de cadenas, a fin de superar los problemas de rendimiento, actualmente existen varios algoritmos en uso, como por ejemplo y por mencionar algunos el algoritmo de *DP* (Devaki-Paul algorithm [PEPEBAB11]) que está dando buenos resultados en muchos casos, no obstante este algoritmo sólo se propuso para la coincidencia de patrones simple.

Los algoritmos de coincidencias de cadenas también son usados en los sistemas de detección de intrusión de redes (NIDS por sus siglas en ingles), los cuales son ampliamente reconocidos como una poderosa herramienta para identificar y desviar ataques maliciosos en la red.

En el campo de la seguridad de las redes, el patrón es una cadena que indica una intrusión en la red, ataques, virus, correo no deseado, o información no deseada, como por ejemplo el *snort y Bro [Intel 2010]* son sistemas de detección y prevención de intrusiones de red de código abierto, desarrollados por *Sourcefire.*

1.3. Correspondencia de Cadenas

Uno de los tipos más simples y naturales de representación de la información es por medio de los textos escritos, este tipo de datos se caracteriza por el hecho de que puede ser escrito como una larga secuencia de caracteres, parecido a una secuencia lineal que es llamada texto. Los textos son de vital importancia en los sistemas de "procesamiento de palabras", los cuales proporcionan facilidades para manipulación de los mismos, estos sistemas normalmente procesan objetos que son bastante grandes. La complejidad de los algoritmos de texto es también uno de los problemas centrales y más estudiados en la ciencia de la computación, por lo que podría decirse que este es el dominio en el que la práctica y la teoría se encuentran muy cerca uno del otro.

El problema básico de los textos o cadenas en la "*Stringology"* es llamado *coincidencia de cadenas,* es utilizado para acceder a la información para buscar un determinado patrón y sin lugar a dudas, en este momento muchos equipos de cómputo están resolviendo este problema como una operación frecuente usada en algunas aplicaciones del sistema, la coincidencia de patrones es comparable en este sentido al ordenamiento o clasificación.

Considere el problema de un lector del diccionario francés "Gran Larousse", que quiere que todas las entradas relacionadas con el nombre "Marie-Curie-Sklodowska", este es un ejemplo de un problema de coincidencia de patrones o correspondencia de cadenas.

En este caso el nombre "Marie-Curie-Sklodowska" es el patrón, en general nosotros podemos querer encontrar una cadena llamada *patrón de longitud* m *dentro de un texto de longitud* n*, en donde* n *es mayor que* m*, el patrón puede ser descrito en una forma más compleja para denotar un conjunto de cadenas y no sólo una sola palabra, en muchos casos* n *en muy grande.*

Breve historia de la comparación de secuencias

El parámetro más crítico en una comparación de secuencia es definitivamente establecer que tan similares son las secuencias dadas, una de las primeras referencias a un problema que involucra la comparación de secuencias se remonta a 1879. Esto consiste de un rompecabezas debido a *Lewis Carrol*[2] llamado dobletes, las reglas del rompecabezas son simples, dos palabras en inglés de igual longitud son proporcionadas, el objetivo es el de transformar la primera palabra en la segunda palabra, mediante la formación de palabras sucesivas de la misma longitud, cambiando sólo una letra a la vez.

Lingüísticamente hablando un doblete es una pareja de palabras que tienen su origen en un mismo étimo y es un fenómeno frecuente y propio de nuestro léxico latino. El mayor desafío es hacer la transformación en el menor número posible de palabras, este juego de palabras llego a ser un favorito cuando la revista "Vanity Fair" organizó un concurso del juego de los *dobletes.*

Un ejemplo dado por el propio Lewis Carroll es la transformación de la palabra "*HEAD" a "TAIL",* esto puede hacerse por la siguiente sucesión de transformaciones.

Ingles	Español
H E A D	Cabeza
h e a l	Curar
t e a l	Certeza
t e l l	Decir
t a l l	Alto
T A I L	Cola

Tabla. 1.3 Ejemplos de doblete y su transformación.

[2]Lewis Carroll también era conocido por sus colegas matemáticos como Charles L. Dodgson. La palabra doblete es un derivado de la palabra doble, esta proviene del latín duplus (doble).

Las dos palabras proporcionadas son llamadas "*dobletes",* las palabras interpuestas ["heal"], ["teal"], ["tell"], y ["tall"] son los enlaces y toda la serie de entradas "*una Cadena".* A menudo hay más de una solución a un doblete, por lo que puede encontrar más de una alternativa de solución con el mismo número de enlaces. Se ha de entender, además, que los enlaces deben ser palabras que se pueden encontrar en un diccionario de inglés o español estándar, y que lo nombres propios no son admisibles. La figura 1.3, nos muestras algunos ejemplos en el idioma inglés:

				FLUTE				
			BLACK	flite	BREAD			
		DEAD	blank	flits	break	TEARS		
	FIRE	lead	blink	flats	bleak	sears	HAND	
WELL	hire	lend	clink	feats	bleat	stars	band	MICE
dell	here	lent	chink	felts	blest	stare	bond	mite
doll	herd	lint	chine	fells	blast	stale	fond	mate
dole	head	line	whine	cells	boast	stile	food	mats
DONE	HEAT	LIVE	WHITE	CELLO	TOAST	SMILE	FOOT	RATS

Figura 1.3. Diferentes ejemplos de *Dobletes.*

Décadas más tarde en el desarrollo de la teoría de la codificación, esta simple noción de la distancia sería más formalmente introducida por *Wesley Hamming* [HAM1950] y es utilizada ampliamente para el diseño de códigos para la transmisión de datos.

Basta decir que por ahora que tenemos una función de la distancia, la cual será cero para las secuencias idénticas, que es la máxima para las secuencias idénticas. Hay muchas maneras de medir la similitud (distancia) de dos cadenas, por mencionar de manera rápida dos de los algoritmos que por excelencia son utilizados para esta operación son: *Edit distance y Longest Common Subsequence Distance.*

1.4 Conceptos Básicos

Alfabeto

Sea *A* una entrada de un conjunto no vacío finito de elementos llamados *Alfabeto,* los elementos de *A* son llamados *letras, caracteres o símbolos,* ejemplos típicos de un alfabeto son: *el conjunto ordinario de todas las letras, el conjunto de números binarios o el conjunto de símbolos del código ASCII.*

Los textos (también llamados cadenas o palabras) sobre un alfabeto *A son una* secuencia finita de elementos de *A,* la longitud (tamaño) de un texto es el número de estos elementos (con repeticiones), por lo tanto la longitud de la palabra "*alfabeto"* es 8.

La longitud de una palabra x esta denotada por $|x|$, la información de entrada para la mayoría de los problemas son palabras (cadenas) y el tamaño n de la entrada del problema será usualmente la longitud de la palabra de entrada. En algunas situaciones n denotará la longitud máxima o el total de la longitud de varias palabras (cadenas) si la entrada de un problema consiste en varias palabras.

La secuencia de letras con longitud cero *se llama cadena vacía* y se denota por **ε**, por las propias simplificaciones delimitadoras y separadoras usualmente empleados en una notación de secuencia se eliminan y una cadena se escribe como la simple yuxtaposición de las letras que lo componen.

Así ***ε****, a, b y baba* son cadenas en cualquier alfabeto que contiene las dos letras a y b, el conjunto de todas las cadenas sobre el alfabeto *A* es denotado por *A*,* y el conjunto de todas las cadenas en el alfabeto *A* excepto la cadena vacía ***ε*** de denota por A^+.

La longitud una cadena x está definida como la longitud de la secuencia asociada con la cadena x y esta denotada por $|x|$ (como mencionamos anteriormente). Nosotros la denotamos por $x[i]$, para i=0,1,…, $|x|$-1, la letra en el índice i de x tiene la limitación de que los índices inicien con 0, cuando $x \neq \varepsilon$, podemos decir específicamente que cada índice i=0,1,…, $|x|$-1 es una posición de x, esto deduce que la i-enésima letra de x es la letra en la posición i-1 y por lo tanto:

$$x = x[0]\, x[1] \ldots x[|x| - 1]$$

De este modo una definición elemental de la identidad entre cualesquiera dos cadenas x y y *es:*

$$x = y$$

Si y sólo si:

$$|x| = |y| \text{ y } x[i] = y[i] \text{ for } i = 0, 1, \ldots, |x| - 1$$

El conjunto de letras que se producen en la cadena x se denota por *alph(x),* por ejemplo:

Si x= abaaab, tenemos que $|x| = 6$ y alph(x)={a,b}

El producto – *que también se dice concatenación* – de dos cadenas x y y es la cadena compuesta de las letras de x seguidas por la letras de y, esto se denota por xy o también $x \cdot y$, y para mostrar la descomposición del resultado de la cadena resultante, el elemento neutro es ε, para cada cadena x y cada número natural n, definimos la n-enésima potencia de la cadena x, denotado por x^n, por x^0=ε y x^k=x^{k-1} x, para k =1, 2,., n, denotamos respectivamente por zy^{-1} y $x^{-1}z$ la cadena x y cuando z =xy, lo contrario – la imagen espejo- de la cadena x es la cadena $x^{\sim}$ definida por:

$$x^{\sim} = x[\,|x| - 1\,] \times [\,|x| - 2] \ldots x[0]$$

Definición de una Cadena.

Una cadena o secuencia es una sucesión de cero o más símbolos de un alfabeto ***Σ*** de cardinalidad s; la cadena con cero símbolos o cadena vacía es denotada por ***ε,*** el conjunto de todas las cadenas sobre el alfabeto Σ es denotado por Σ^+. Una cadena x de longitud m está representada por $x_1 \ldots x_{m,}$ donde x_i ***ϵ Σ*** para *1 ≤ i ≤* m.

Decimos que ***Σ*** *está* delimitada cuando s es una constante, y en caso contrario se encuentra sin límites, una cadena w es una subcadena de x *si* x= uwv; *para* u, v ***ϵ*** Σ^+; Equivalentemente decimos que la cadena w *se produce en la posición* | u | + *1 de la cadena* x.

La posición | u | + *1 se dice que es la posición inicial de* w *en* x *y la posición* | u | + | w | *la posición final de* w *en* x. *Una cadena* w *es un prefijo de* x *si* x= wu *para* u ***ϵ*** Σ^+. *Del mismo modo* w *es un sufijo de* x *si* x = uw *para* u *0* ***ϵ*** Σ^+.

1.5 Distancias de Edición

La noción de la distancia es un tanto dual a la similitud, esto se trata de secuencias como puntos en un espacio métrico, un conjunto $\boldsymbol{X}$ de elementos $x, y, \ldots \in \boldsymbol{X}$ es llamado un espacio métrico, siempre y cuando un número real $d(x,y)$ existe para cada par $(x,y) \in \boldsymbol{X}$.

Una medida de la distancia es una función que también se asocia a un valor numérico con un par de secuencias, pero con la idea de que cuanto mayor sea la distancia, más pequeña es la similitud y viceversa, la medida de la distancia normalmente satisfacen los axiomas matemáticos de una métrica y particularmente, los valores de la distancia nunca son negativos.

Sea $d(x,y)$ la distancia entre dos secuencias $x,y \in \Sigma^*$, si tal función resulta ser simétrica $d(x,y) = d(y,x)$, entonces $d()$ es también simétrica y satisface los siguientes axiomas:

- $\longrightarrow \forall x,\ d(x,x) = 0$
- $\longrightarrow$ [Positiva] $\forall x \neq y\ ,\ d(x,y) > 0$
- $\longrightarrow$ [Simétrica] $\forall x,y,\ d(x,y) = d(y,x)$
- $\longrightarrow$ [Desigualdad Triangular] $\forall x,y,z,\ d(x,z) \leq = d(x,y) + d(y,z)$

Ahora, tomando como referencia estos axiomas, definiremos algunas de variantes más tradicionales de las métricas de cadenas.

Distancia de Edición Simple

La distancia de edición simple $d_{D(x,y)}$, entre cadenas x y y es el número mínimo de caracteres insertados y/o borrados requeridos para hacer que x *sea* igual a y, la distancia es simétrica y está establecida por $0 \leq d_{D(x,y)} \leq$ max (| x |+ | y |), como por ejemplo sea $d_{D(\text{“mia”,”rima”})} = d_{D(\text{“mia”,”mira”})=1}$

Distancia de Hamming.

A finales de la década de 1940, Claude Shannon fue el que desarrollo la teoría de la información y comunicación como un modelo matemático, al mismo tiempo, Richard Hamming, un colega de Shannon de los laboratorios Bell, encontró una necesidad de la corrección de errores en su trabajo sobre computadoras. La comprobación de paridad ya se utilizaba para detectar errores en los cálculos de las computadoras basadas en relevadores o relés de ese entonces, Hamming se dio cuenta de que un patrón más sofisticado de comprobación de paridad, permitirá la corrección errores simples junto con la detección de errores dobles.

Los códigos que Hamming inventó, el código de corrección de errores simples junto con el código de detección de errores dobles y sus versiones extendidas, marcó el comienzo de la teoría de la codificación. En Teoría de la Información se denomina distancia de Hamming a la efectividad de los códigos de bloque y depende de la diferencia entre una palabra de código válida y otra.

Cuanto mayor sea esta diferencia, menor es la posibilidad de que un código válido se transforme en otro código válido por una serie de errores. A esta diferencia se le llama distancia de Hamming, y se define como el número de bits que tienen que cambiarse para transformar una palabra de código válida en otra palabra de código válida. Si dos palabras de código difieren en una distancia *d*, se necesitan *d* errores para convertir una en la otra.

Distancia de Edición de Levenshtein

En teoría de la información y ciencias de la computación se llama Distancia de Levenshtein, distancia de edición, o distancia entre palabras, al número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter.

Esta distancia recibe ese nombre en honor al científico ruso Vladimir Levenshtein, quien se ocupara de esta distancia en 1965. Es útil en algoritmos que determinan cuán similares son dos cadenas de caracteres, como es el caso de los correctores de ortografía. Es usual hablar de distancia o diferencias entre objetos, es por ello que no resulta extraño el término distancia o medida de similitud entre dos cadenas.

La distancia de Edición, $\boldsymbol{d_E}(\boldsymbol{x}, \boldsymbol{y})$, entre dos cadenas x y y, es el número mínimo de operaciones requeridas para transformar x *en* y, estas operaciones son:

- [Inserción] inserta una nueva letra a dentro de una x. Una operación de inserción sobre una $cadena \quad x = vw$ consiste en añadir una letra a, convirtiendo a $x\ en\ x' = vaw$.
- [Supresión] borrar una letra a de una cadena x. Una operación de eliminación dentro de una cadena $x' = vaw$, consiste en remover una letra, convirtiendo a $x\ en\ x' = vw$.
- [Sustitución] sustituir una letra a *en una cadena* x. Una operación de sustitución dentro de una cadena $x = vaw$ consiste en reemplazar una letra por otra, convirtiendo a $x\ en\ x' = vbw$.

La distancia es simétrica y está determinada por $0 \leq d_{E(x,y)} \leq \max(|x|, |y|)$, dentro de la literatura este indicador o métrica en muchos de los casos es llamada "Distancia de Edición".

La distancia de Hamming (definida en el punto anterior) puede ser vista como una distancia de edición, en donde un sólo reemplazo o sustitución está permitida.

El algoritmo de Levenshtein, permite inserciones, eliminaciones y sustituciones, en la definición simplificada, en todas las operaciones su costo es de 1, pero es una práctica común se asigna un costo de 2 a las sustituciones.

El resultado es entonces el número mínimo de inserciones, sustituciones y eliminaciones para hacer dos cadenas iguales, por lo que la distancia es simétrica.

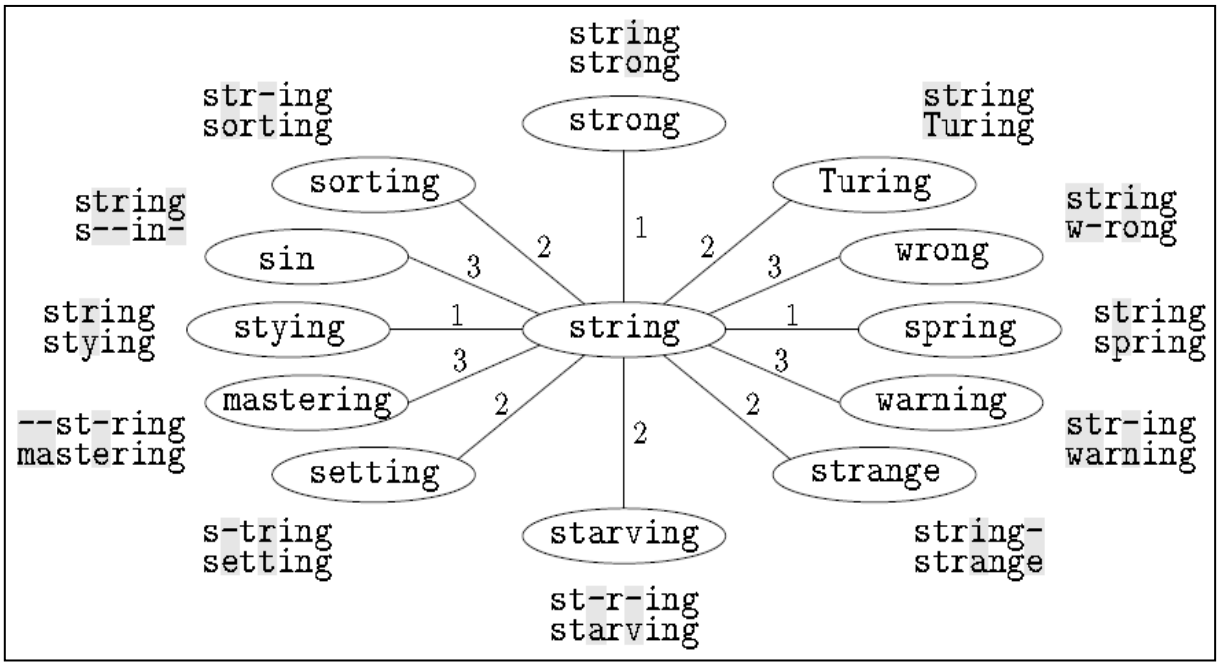


Figura 1.5. Proporciona la distancia de edición entre la cadena "string" y un número de otras cadenas. Nosotros usamos las palabras en inglés para este ejemplo, pero podría haber sido cualquier otro tipo de secuencias como secuencias de ADN, por ejemplo.

Distancia Levenshtein Ponderada

Esta es una generalización de uso común de la distancia de edición. Para cada par de símbolos a, b del alfabeto $\Sigma \cup \{\varepsilon\}$ permita que el *costo(a* $\rightarrow$ *b)* ≥ *0.* El costo (peso) de la sustitución de "a" a "b".

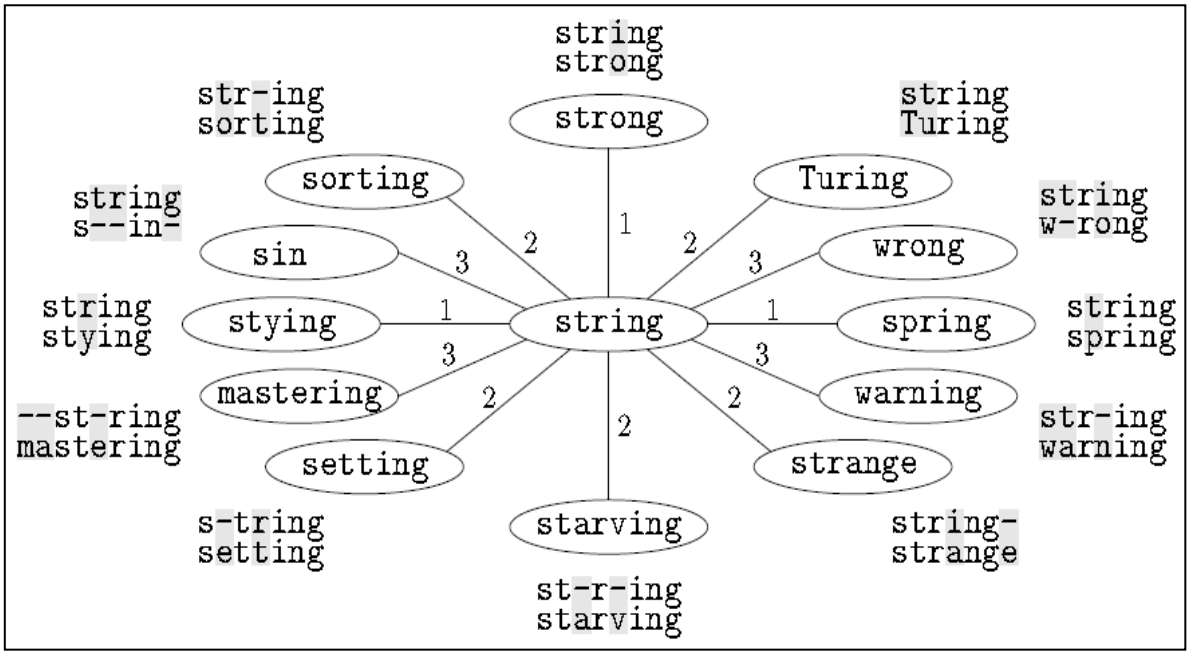


Figura 1.5.1. Una ilustración de la Distancia de Edición para la secuencia "string". Cada arista esta etiquetada con la distancia de edición entre las cadenas. Para cada par de cadenas damos una posible alineación óptima. Utilizamos el carácter "-" para representar el carácter 'ε'.

El símbolo vacío nos permite interpretar la inserción y la supresión como una operación de sustitución. Por ejemplo la sustitución de a $\rightarrow$ ε corresponde a la supresión de "a", y ε $\rightarrow$ a representa la inserción de "a".

La distancia de Levenshtein ponderada (también conocida como la distancia de Levenshtein generalizada) $\boldsymbol{d_w}$ (x, y), entre las cadenas "x"y "y", es el costo mínimo de la transformación de *"x"* en "*y"* cuando las operaciones permitidas son la inserción, la eliminación y la sustitución, con una función de costo, (ejemplo de lo anterior es la figura 1.5.1).

Capítulo II. La Distancia Levenshtein

La distancia de Levenshtein es una medida de la similitud entre dos cadenas, a las cuales nos referiremos como la cadena de origen (s) y la cadena de destino (t), la distancia es el número de eliminaciones, inserciones, o sustituciones necesarias para transformar s en t. Cuanto mayor sea la distancia Levenshtein, más diferentes son las cadenas, la métrica de las cadenas también a veces se llama distancia de edición.

La distancia de Levenshtein, distancia de edición o distancia entre palabras es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra, se usa ampliamente en teoría de la información y ciencias de la computación, se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter.

Esta distancia recibe ese nombre en honor al científico ruso Vladimir Levenshtein, es útil en programas que determinan cuán similares son dos cadenas de caracteres, como es el caso de los correctores de ortografía, reconocimiento de voz, análisis de ADN, detección de plagio, etc.

Por ejemplo, si la cadena (s) es "test" y la cadena (t) es "test", entonces la distancia de Levenshtein de la cadena (s, t) es 0, porque no se necesitan transformaciones, las cadenas son idénticas, por otro lado y por mencionar otro ejemplo si la cadena (s) es "$test$" y la cadena (t) es *"tent"*, entonces la distancia de Levenshtein de $la\ cadena\ (s, t)$ es 1, ya que una sustitución $(cambiar$ "s" a "n") es suficiente para transformar la $cadena\ (s)\ en\ la\ cadena\ (t)$.

2.1 Coincidencia de Patrones

La coincidencia de patrones exacta implica encontrar todas las ocurrencias de un patrón P dentro de una cadena S, en donde S es mayor que P, la forma más simple de la coincidencia de un patrón, el patrón de coincidencia exacta es todavía ampliamente utilizado en una variedad de búsquedas de cadenas dentro de los motores de búsqueda de internet para el procesamiento de textos.

Aunque el aumento de las velocidades de los procesadores y otros avances han reducido la respuesta de las búsquedas a tiempos insignificantes, la coincidencia de patrones exacta todavía sigue siendo un área útil de estudio y desarrollo por múltiples razones.

La idea principal de la coincidencia de patrones (*pattern matching)* aparece del lenguaje *SNOBOL*[3] (versión número 4) y representa la búsqueda dentro de una secuencia de símbolos, de la coincidencia de un patrón dado con ciertas propiedades, mediante secuencias o estructuras en árbol.

Las secuencias (cadenas de texto) representantes de un patrón se describen mediante expresiones regulares[4], las cuales usan su propio algoritmo para la búsqueda. Además, estás pueden verse como una estructura de ramificaciones de un árbol para cada elemento, el caso más sencillo es el de encontrar una cadena dentro de una secuencia.

[3] **SNOBOL** (StriNg Oriented symBOlic Language) es un lenguaje de programación de computadoras de muy alto nivel que surgió en la década de los 60 en los Laboratorios Bell merced al equipo formado por David J. Farber, Ralph E. Griswold y Ivan P. Polonsky.

[4] Expresión que describe un conjunto de cadenas representantes de un lenguaje. La *expresión regular* se construye mediante caracteres del alfabeto sobre el cual se define el lenguaje.

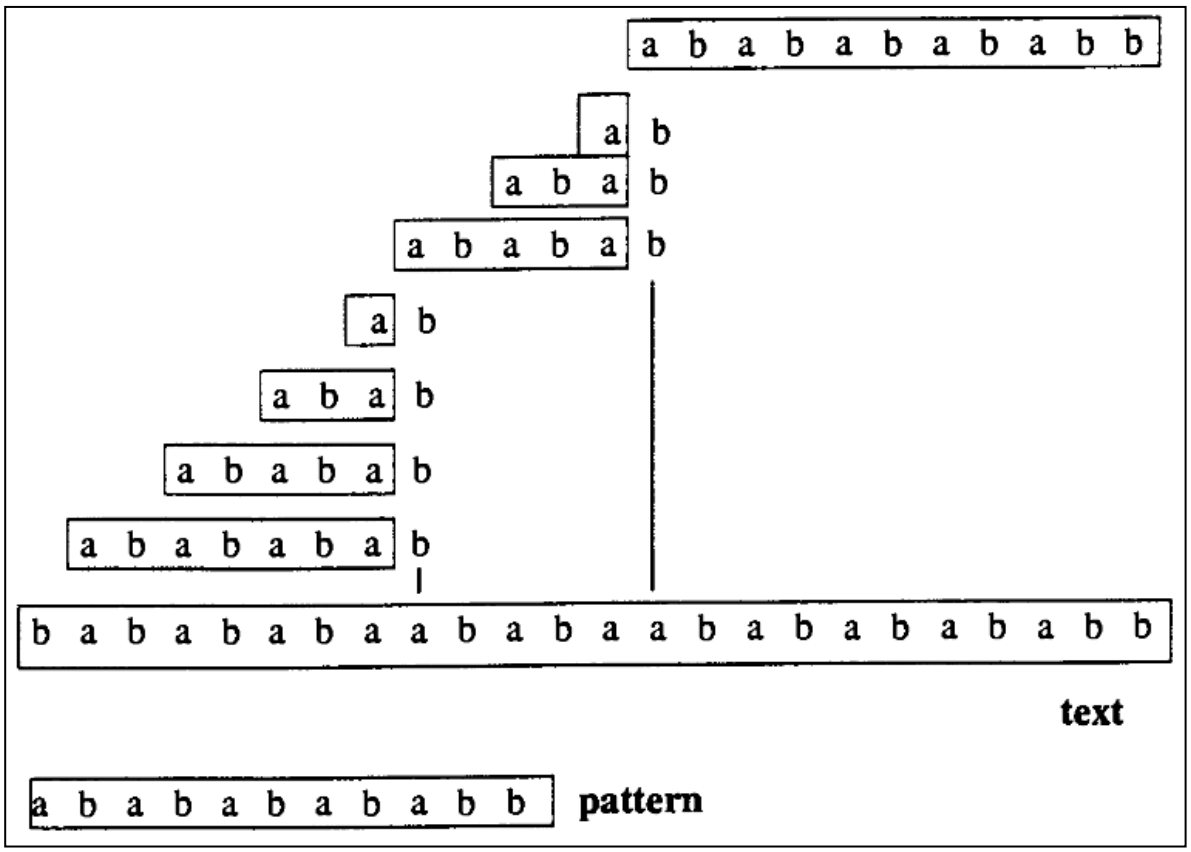


Figura 2.1 Coincidencia de patrones Exacta (Pattern Matching). La historia de algoritmo de MP en un ejemplo de texto y el patrón.

Dependiendo del lenguaje de programación, se usan patrones fundamentales para la evaluación del resultado del programa. La coincidencia de patrones ofrece más beneficios cuando la mayoría de las estructuras de datos son tan simples y flexibles como sea posible.

El *pattern matching* propiamente dicho, es fundamental en bases de datos y aplicaciones de procesamiento de texto, los editores de texto contienen un mecanismo para buscar en el texto las cadenas especificadas.

Algunos lenguajes de programación de *pattern matching* derivan parte de su habilidad en construir cadenas primitivas coincidentes, haciendo más fácil los programas de filtrado y modificación de texto.

Los programas verificadores de una entrada de texto buscan las palabras en el diccionario y rechazan cualquier cadena que no coincida, un ejemplo práctico seria el comando del sistema operativo *UNIX Grep*[5] como el ejemplo que se muestra en la figura 2.1.1

Ejemplo: (entrada)	**Ejemplo: (salida)**
Esta es una prueba para ver como funciona grep.	$> cat README.txt | grep --color **para**
Esta es una pueba para vir cmo funca grep.	Esta es una prueba **para** ver como funciona grep.
Esta es una linea.	Esta es una pueba **para** vir cmo funca grep.

Figura 2.1.1. Ejemplo del comando grep (implementación del algoritmo de *pattern matching*). El comando *cat* muestra el archivo README.txt pero al mismo tiempo hace una coincidencia de patrones exacta indicándole que cuando encuentre el patrón o cadena "*para"* la ponga de color rojo.

La coincidencia de patrones suele usarse para probar si las cosas tiene la estructura, para encontrar la estructura correspondiente, para recuperar la alineación de las partes y para substituir la parte coincidente por otra cosa, este es uno de los problemas más antiguos y generales dentro de la ciencia de computación y hay que destacar, que en los últimos años se ha visto aumentado el interés por los problemas de *coincidencias* sobre cadenas de texto, especialmente por la gestión de la rapidez y la eficiencia de la búsqueda.

En esencia la coincidencia de cadenas exacta (*exact pattern matching)* es la base para todas las aplicaciones de búsqueda de texto, el problema puede plantearse como:

- Dado un patrón *P* de longitud *m* y una cadena (o texto) *T* de longitud *n (m ≤ n),* encontrar todas las ocurrencias de *P* en *T* [CWN2007].

[5]Grep es una utilidad de la línea de comandos escrita originalmente para ser usada con el sistema operativo Unix, toma una expresión regular de la línea de comandos, lee la entrada estándar o una lista de archivos, e imprime las líneas que contengan coincidencias para la expresión regular.

- Matemáticamente podríamos decir: El problema de coincidencia de cadenas es el de encontrar todas las ocurrencias de un determinado patrón $P = p_1p_2...p_m$, en un texto largo $T=t_1t_2...t_n$, donde ambos T y P son secuencias de caracteres de un conjunto de caracteres finito ∑.

- Sea ∑ un alfabeto
    - Entrada: Una cadena $T = t_1 t_2... t_n$, y un patrón $P = p_1p_2...p_m$, por lo que $t_i, p_i \in \sum$
    - Salida: Todas las posiciones *i* en *T* donde hay una ocurrencia del patrón *P, es decir T[i+ k] = P [k+1] , 0 ≤ k ≤ m [NEU09] .*

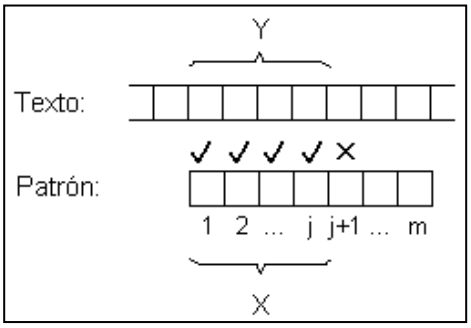
Por lo que la coincidencia es sólo una, lo que significa que la palabra o el patrón exacto es encontrado, como se mencionó anteriormente en el sistema operativo *UNIX*[6] existe un comando con una gran utilidad llamado "*grep",* que permite al usuario buscar de manera global las líneas que coincidan con la expresión regular, actualmente existen una gran variedad de algoritmos para resolver este problema, los algoritmos de coincidencia de patrones exacto más viejos y famosos son: el algoritmo de Knuth-Morris-Pratt (KMP) y el algoritmo de Boyer-Moore, estos algoritmos aparecieron por el año de 1977.

El algoritmo KMP es un algoritmo de búsqueda de subcadenas simple y por lo tanto su objetivo es buscar la existencia de una subcadena dentro de una cadena.

Para ello utiliza información basada en los fallos previos, aprovechando la información que la propia palabra a buscar contiene de sí (sobre ella se precalcula una tabla de valores), para determinar donde podría darse la siguiente existencia, sin necesidad de analizar más de 1 vez los caracteres de la cadena donde se busca.

[6]**Unix** (registrado oficialmente como **UNIX**®) es un sistema operativo portable, multitarea y multiusuario, desarrollado, en principio, en 1969, por un grupo de empleados de los laboratorios Bell de AT&T, entre los que figuran Ken Thompson, Dennis Ritchie y Douglas McIlroy.

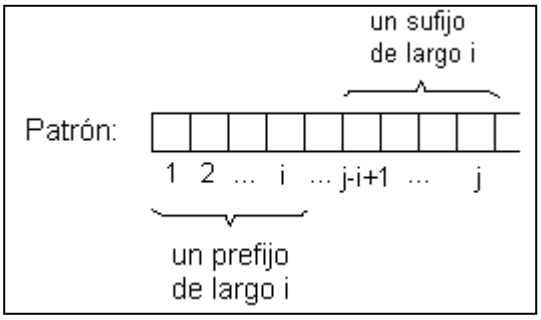
Suponga que se está comparando el patrón y el texto en una posición dada, cuando se encuentra una discrepancia.



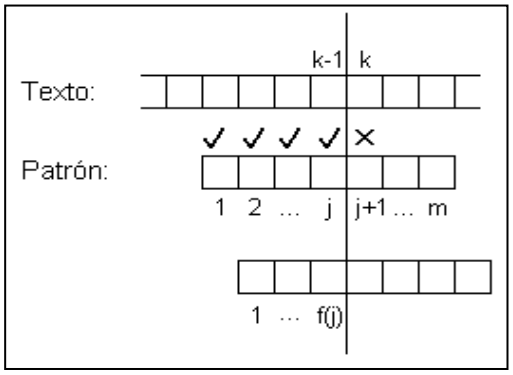
Sea X la parte del patrón que calza con el texto, e Y la correspondiente parte del texto, y suponga que el largo de X es j. El algoritmo de fuerza bruta mueve el patrón una posición hacia la derecha, sin embargo, esto puede o no puede ser lo correcto en el sentido que los primeros $j-1$ caracteres de X pueden o no pueden calzar los últimos $j-1$ caracteres de Y.

La observación clave que realiza el algoritmo Knuth-Morris-Pratt (en adelante KMP) es que X es igual a Y, por lo que la pregunta planteada en el párrafo anterior puede ser respondida mirando solamente el patrón de búsqueda, lo cual permite pre-calcular la respuesta y almacenarla en una tabla.

Por lo tanto, si deslizar el patrón en una posición no funciona, se puede intentar deslizarlo en 2, 3,..., hasta j posiciones. Se define la función de fracaso (failure_function) del patrón como:



Intuitivamente, $f(j)$ es el largo del mayor prefijo de X que además es sufijo de X. Note que $j = 1$ es un caso especial, puesto que si hay una discrepancia en $b1$ el patrón se desliza en una posición Si se detecta una discrepancia entre el patrón y el texto cuando se trata de calzar $bj + 1$, se desliza el patrón de manera que $bf(j)$ se encuentre donde bj se encontraba, y se intenta calzar nuevamente.



Suponiendo que se tiene f(j) pre-calculado, la implementación del algoritmo KMP es la siguiente:

```
// n = largo del texto
// m = largo del patrón
// Los índices comienzan desde 1
int k=0;
int j=0;
while (k<n && j<m)
{
  while (j>0 && texto[k+1]!=patron[j+1])
  {
    j=f[j];
  }
  if (texto[k+1])==patron[j+1]))
{
j++;
  }
  k++;
}
// j==m => coincidencia, j el patrón estaba en el
texto
```

Patron = "a a b a a a"
1 2 3 4 5 6

j	1	2	3	4	5	6
f(j)	0	1	0	1	2	2

Texto: "a a a a b a a b a a a b b"
j = 0 1 2
1 2
1 2 3 4 5
2 3 4 5 6 → **ok**

2.2 Coincidencia de Cadenas Aproximada

La definición de una coincidencia (también llamada como *match)* puede permitir ligeras diferencias entre el patrón y las coincidencias en el texto, a esto se le llama *Aproxximate Pattern Matching* o incluso *Aproxximate String Matching, el pattern matching aproximado* es fundamental para procesamiento de texto a causa de los múltiples errores que se pueden producir en el mismo, como se ha dicho anteriormente *la coincidencias de cadenas* es muy utilizada para las búsquedas de similitud de secuencias.

En la práctica las aplicaciones de coincidencia de patrones, la coincidencia exacta no es siempre pertinente, a menudo es más importante encontrar los objetos que corresponden a un patrón determinado en una forma razonablemente aproximada.

De manera más general, la coincidencia de patrones aproximada consiste en la localización de todos las ocurrencias de los caracteres dentro de un texto $\boldsymbol{y}$ *que son similares a una cadena* $\boldsymbol{x}$*,* esto consiste en producir la posición de los caracteres de $\boldsymbol{y}$ que están de distancia a lo sumo *k de x, para un entero natural k dado, por lo que asumimos que* $K < |X| \leq |Y|$.

Análogamente para las secuencias de ADN se llama problema de alineación (figura 2.2), y los algoritmos son basados principalmente en el llamado método algorítmico de programación dinámica.

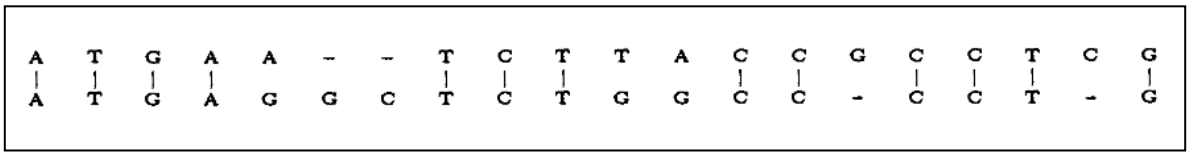


Figura 2.2. La Alineación de dos secuencias de ADN que muestran las operaciones de cambios, inserciones ("-" en la línea superior) y eliminación ("-" en la línea de abajo).

Considere el problema de búsqueda de cadenas, donde las diferencias entre los caracteres de un patrón y los caracteres de un texto son permitidos, cada diferencia se debe a un desajuste entre un carácter de un texto y un carácter de un patrón, o a un carácter superfluo dentro del patrón.

Siguiendo este mismo contexto decimos, *que dado un texto de longitud n, un patrón de longitud m y un entero k, aplicaremos los algoritmos seriales y paralelos para encontrar todas las ocurrencias de un patrón en el texto con la mayoría de las k diferencias presentadas.*

El más claro y antiguo ejemplo de un algoritmo de *coincidencia de cadenas aproximadas* empleado e implementado es el comando del sistema operativo *UNIX* llamado ***agrep***[7], tal y como lo muestra la figura 2.2.1.

```
agrep -2  -c funciona prueba
```

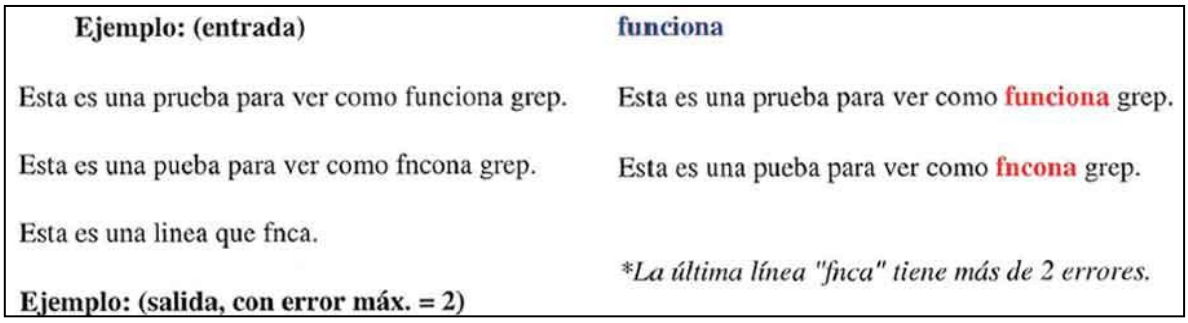


Figura 2.2.1. Ejemplo del comando agrep, implementación del algoritmo de Approximate Pattern Matching.

agrep (approximate grep) es un programa que realiza búsquedas aproximadas en cadenas de texto (*fuzzy string matching*), fue desarrollado por Udi Manber y Sun Wu entre 1988 y 1991, para ser usado en Unix. Más tarde fue adaptado para OS/2, DOS, y Windows.

[7]**Agrep (Approximate General Regular Expression Pattern Matcher)** es un programa de búsqueda de cadenas difusa.

Considere el problema de búsqueda de cadenas, donde las diferencias entre caracteres de un patrón y los caracteres del texto son permitidos, cada diferencia es debido a la falta de correspondencia entre un carácter del texto y un carácter del patrón, o un carácter superfluo en el texto, o un carácter superfluo en el patrón.

La correspondencia de cadenas o patrones aproximada también llamada “Coincidencia de cadenas con errores permitidos”, es el problema de encontrar un patrón *P* en un texto *T* cuando un número limitado de *K* diferencias es permitido entre el patrón y sus ocurrencias en el texto.

De los muchos modelos existentes que definen una “diferencia”, nos centramos en el más popular, llamada *La Distancia de Levenshtein o La Distancia de Edición [LEV1996],* existen otros modelos más complejos, especialmente en biología computacional, pero el modelo de distancia de edición ha recibido la mayor atención y los algoritmos más eficaces se han desarrollado para ello.

Como se mencionó anteriormente el problema de *coincidencia de cadenas (String Matching)* que permite errores, también llamado correspondencia de cadenas aproximada, tiene como objetivo general realizar una correspondencia de cadenas de un patrón en un texto en el que uno o ambos de ellos han sufrido algún tipo de corrupción.

El problema en su forma más general es encontrar un texto en un texto donde un patrón de texto dado ocurre, permitiendo un número limitado de “errores” en las coincidencias, por lo que cada aplicación utiliza un modelo de error diferente, el cual define que tan diferentes o como son diferentes esas dos cadenas, la idea de esta distancia entre las cadenas es hacer a ésta pequeña, que cuando una de las cadenas es probablemente una variante errónea de la otra en virtud del modelo de error en uso.

2.3. El algoritmo de Levenshtein.

Uno de los mejores casos estudiados de este modelo de error es la denominada distancia de edición lo que nos permite borrar, insertar y sustituir caracteres simples (con diferencia de 1) en ambas cadenas, si las diferentes operaciones tienen diferentes costos o los costos dependen de los caracteres involucrados.

Hablamos de la distancia de edición en general, en caso contrario si todas las operaciones cuestan 1, hablamos de la simple distancia de edición o simplemente distancia de edición (*ed),* en este último caso, simplemente buscamos el número mínimo de inserciones, supresiones y sustituciones (figura 2.3) para hacer que ambas cadenas sean iguales, como por ejemplo $ed(\text{“}survey,\text{”}\ \text{“}surery\text{”}) = 2$.

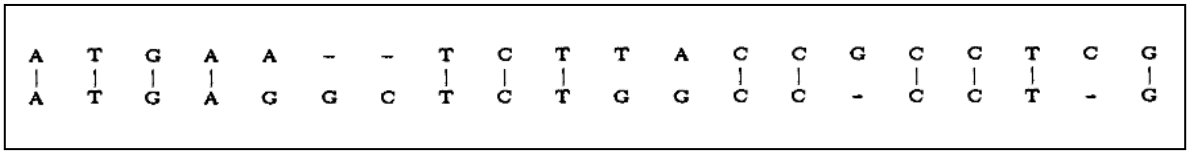


Figura 2.3. La Alineación de dos secuencias de ADN que muestran las operaciones de cambios, inserciones (“-“ en la línea superior) y eliminación (“-“ en la línea de abajo).

La búsqueda de patrones permite acceder rápidamente a zonas de interés dentro de gran cantidad de información, la búsqueda de patrones en un texto se define como: *Dado un patrón* $P = p_1 \ldots p_m$ *y un texto* $T = t_1 \ldots t_u$*, ambas secuencias de caracteres sobre el alfabeto finito ∑, encontrar todas las coincidencias de P en T, es decir, encontrar el conjunto { |x|, T = xPy }.*

El algoritmo de Levenshtein encuentra la manera más barata de transformar una cadena en otra, y este proceso es la base para muchos algoritmos de coincidencia de patrones.

Las transformaciones son realizadas haciendo alguna de estas operaciones: La inserción, la eliminación, y la sustitución (incluso algunas versiones extendidas también incluyen la transposición), a cada operación es asignado un costo y el resultado es el costo total de la transformación de una cadena en otra.

Este es un ejemplo de cómo funciona el algoritmo, e ilustra cómo se ve para todas las diferentes formas la operación para transformar una cadena en otra, como son las cadenas ejemplo; MATRIZ y NATRIZ que serán utilizadas para el análisis. El algoritmo inicia en la esquina superior izquierda de una matriz bidimensional indexada en las filas las letras de la palabra origen, y en columnas por las letras de la palabra objetivo.

Se rellena el resto de la matriz mientras encuentra todas las distancias entre cada prefijo inicial de la fuente sobre un lado y cada prefijo inicial del objetivo sobra la otra. Cada celda $[i, j]$ representa la distancia mínima entre las letras i primera de la palabra de origen, y las letras *j primera* de la palabra objetivo. Sólo es posible rellenar el valor de una celda en caso de que los valores de todos sus vecinos arriba y hacia la izquierda se han llenado (Figura 2.3.1).

		m	a	t	r	i	z
	0	1	2	3	4	5	6
n	1						
a	2						
t	3						
r	4						
i	5						
x	6						

Figura 2.3.1. Cálculo Matricial Distancia de Levenshtein.

Por lo tanto consideraremos tres principales tipos de diferencias entre dos cadenas x y y:

- $edit(x, y) \geq 0$
- $edit(x, y) = 0\ si\ x\ = y$
- $edit(x, y) = edit\ (y, x)(simetria)$
- $edit(x, y) \leq edit\ (x, z) + \ edit\ (z, y)\ (desigualdad\ triangular)$

La simetría de edición viene de la dualidad entre las supresiones y las inserciones: Una eliminación de la letra a de x con el fin de obtener y corresponde a una inserción de a en y para conseguir x.

Como por ejemplo el texto $x = wojtk$ puede ser transformado en $y = wjeek$ usando una eliminación, un cambio y una inserción, esto nos permite ver que $edit\ (wojtk, wjeek) \leq 3$, porque se usan las tres operaciones, de hecho este es el mínimo número de operaciones de edición para transformar $wojtk\ en\ wjeek$ (figura 2.3.2).

w	*o*	*j*	*t*		*k*	*Acción*
	↓					Eliminación
			↓			Cambio
				↓		Inserción
w		*j*	*e*	*e*	*k*	Resultado

Figura 2.3.2. Las tres operaciones básicas del algoritmo de Levenshtein

A partir de ahora, tenemos en cuenta que las palabras de x e y son fijas. La longitud de $\mathcal{X}$ es m, y la longitud de Y es n, y suponemos que $n > m$. Definimos la matriz $EDIT\ por$:

$$EDIT[i,j] = \ edit\ (x[1..i], y[1..j])$$

$$Para\ 0\ < i\ < m\ \text{y}\ 0\ <\ j\ < n\ Los\ valores\ límite\ se\ definen\ como\ sigue\ (por\ 0\ < i\ < m, 0\ <\ j\ < n):$$

$$EDIT[0,j] = j, \qquad EDIT[i,0] = \ i$$

(*) Esta es la fórmula sencilla para calcular los demás elementos.

$$EDIT[i,j] = \min(\ EDIT[i-1,j] + 1,$$
$$EDIT\ [i,j-1] + 1,\ \ EDIT[i-1,j-1] + \ \partial\ (x[i], y[j])$$

Donde $\delta(a,b) = \ 0\ si\ a = b, y\ \partial(a,b) = 1\ en\ otro\ caso,$ la formula refleja las tres operaciones, eliminación, inserción y cambio respectivamente, la implementación de la función queda establecida de la siguiente forma:

```
// Implementación de la función del algoritmo de Levenshtein
1 function r(x, y)
2 if x = y then return 0
3 otherwise return 1
4 function EDIT (A, B)
5    D[0 - |A|, 0 - |B|] Es una arreglo de números enteros
6   for i = 0 to |A|
7           D[i, 0] = i
8   for j = 0 to |B|
9           D[0, j] = j
10     for i = 1 to |A|
11      for j = 1 to |B|
12        D[i,j] = min (D[i-1,j]+ 1 //
                                          Eliminación De STRINGl[i]
                       D[i,j-1] + 1 //
                                          Inserción De STRING2[j]
D[i-1,j-1]+ r[A[i], B[j])//
Sustitución De STRINGl[i]con STRING2[j]
13 return D[|A|, |B|]
```

El algoritmo anterior calcula la distancia de edición de las cadenas $x\ e\ y$, esto almacena y calcula todos los valores de la matriz $D[i,j]$, aunque en $D[i,j]$ sólo una entrada, $D[A,B]$ se requiere.

Esto sirve para ahorrar tiempo, y a esta característica se denomina el método de programación dinámica. Otro posible algoritmo para calcular $d(x,y)$ podría ser utilizar el algoritmo clásico de *Dijkstra* para caminos más cortos.

Como podemos observar en la función $EDIT[A,B]$, en la línea 12, se reflejan las tres operaciones básicas: eliminación, inserción y cambio o sustitución en ese orden respectivamente.

Por lo que analizando la función se puede determinar que hay una formulación teórica de un grafo sobre el problema de edición de distancias, consideramos la rejilla anterior (figura 2.3.1) como un grafo denotado por G, compuesta por nodos $(i,j)(\ para\ 0\ \leq i\ \leq A, 0\ J\ \leq B)$.

El nodo $(i-1,j-1)$ está conectado a los tres nodos $(i-1,j),(i,j-1),(i,j)$, cuando se definen es decir, cuando $(\ i\ \leq A,\ J\ \leq B)$.

Cada arista de la rejilla del grafo tiene un peso de acuerdo con la recurrencia (*), las aristas a partir de $(i-1,j-1),(i-1,j)\ y\ \ (i,j-1)$, tienen un peso de 1, ya que corresponden a la inserción o eliminación de un carácter o símbolo, la arista de $(i-1,j-1)\ a\ (i,j)$ tiene el peso $\partial(x[i],\ y[j])$.

La figura 2.3.3. Nos muestra un ejemplo de una rejilla (grid) de un grafo, para las palabras *cabac* y *abcabbbaa,* la edición de distancia entre las palabras x y y es igual a la longitud de la ruta más corta dentro del grafo desde la $fuente\ (0,0)\ (esquina\ \ superior\ izquierda)\ hasta\ el\ decenso\ (m.n)$.

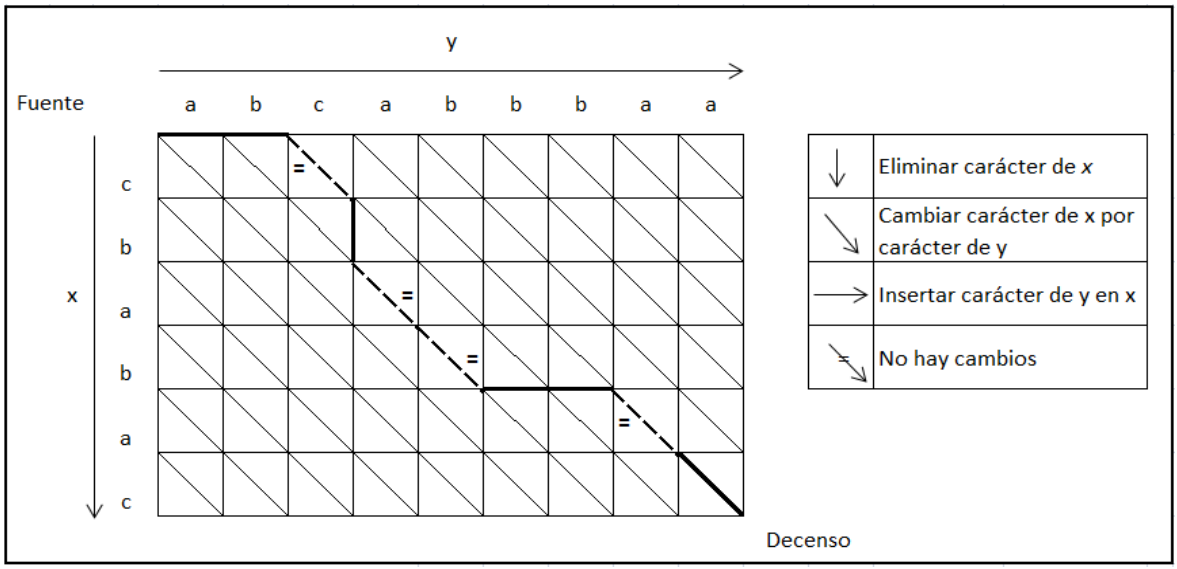


Figura 2.3.3 La ruta corresponde a la secuencia de operaciones de edición: inserción (a), eliminación (b), inserción (b), inserción (b), cambio (c,a).

Para dejar más en claro lo anterior veamos la siguiente gráfica.

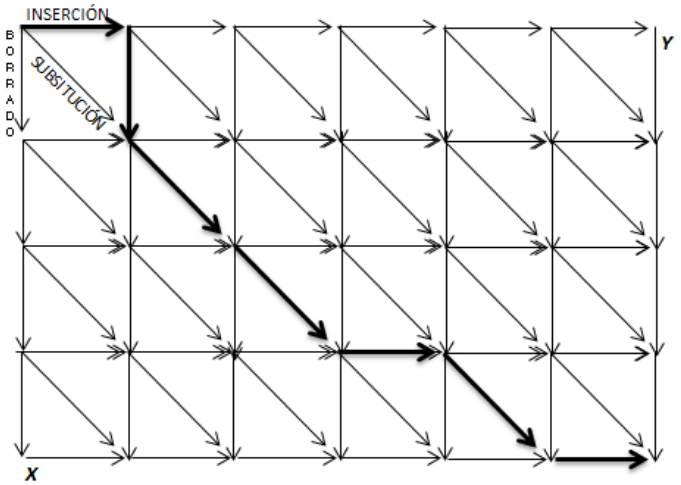


Figura 2.3.3.1. Ejemplo de un rejilla grafo acíclico dirigido (DAG)

Como se mencionó anteriormente la distancia Levenshtein se obtiene mediante la búsqueda de la forma más barata para transformar una cadena en otra. Las transformaciones son las operaciones de un solo paso de: inserción, eliminación y sustitución. En las versiones más simples las sustituciones cuestan dos unidades excepto cuando el origen y el destino son idénticos, en cuyo caso el costo es cero las inserciones y eliminaciones cuestan la mitad que las sustituciones.

El algoritmo comienza con la esquina superior izquierda de una matriz de dos dimensiones indexadas en filas por las letras de la palabra de origen, y en las columnas de las letras del destino. Se rellena el resto de la matriz, mientras que la búsqueda de todas las distancias entre cada prefijo inicial de la fuente por una parte y cada prefijo inicial del objetivo. Cada celda $d[i,j]$ representa la distancia (mínima) entre las primeras letras i de la palabra de origen y las primeras j letras del destino. Se pueden rellenar los valores de la celda sólo en el caso de que los valores de todos sus vecinos (hacia arriba y hacia la izquierda) se hayan llenado, para representar gráficamente lo anterior se muestra la siguiente tabla.

Diagonal	Arriba
Izquierda	min (arriba + eliminacion, diagonal + Sustitución, izquierda + inserción)

Tabla 2.3. Las tres operaciones para encontrar el valor de la celda $min[i,j]$.

Para dejar en claro cómo es que se implementa la función arriba mencionada, a continuación se implementó dicha función en C#:

```
1  using System;
2  using System.Collections.Generic;
```

```
3  using System.Linq;
4  using System.Text;
5
6  /// <summary>
7  /// Algoritmo de Levenshtein
8  /// Algorito de Coincidencia de patrones sobre cadenas
9  ///
10 /// </summary>
11 static class LevenshteinDistance
12 {
13     /// <summary>
14 /// Calcula la distancia entre dos cadenas.
15 /// </summary>
16     public static int Compara(string s, string t)
17     {
18
19      // Paso #1
20
21      int n = s.Length;
22      int m = t.Length;
23      int[,] d = new int[n + 1, m + 1];
24
25
26      if (n == 0)
27      {
28          return m;
29      }
30
31      if (m == 0)
32      {
33          return n;
```

```
34      }
35
36      // Paso #2
37     for (int x= 0; x <= n; d[x, 0] = x++) ;
38 for (int y = 0; y <= m; d[0, y] = y++) ;
39
40      // Paso #3
41      for (int x = 1; x <= n; x++)
42      {
43      // Paso #4
44          for (int y = 1; y <= m; y++)
45      {
46           // Paso #5
47           int cost = (t[y - 1] == s[x - 1]) ? 0 : 1;
48
49           // Paso #6
50           d[x, y] = Math.Min(
51
52                       Math.Min(d[x - 1, y] + 1, d[x, y - 1]
53                      + 1),
54 d[x - 1, y - 1] + cost);
55          }
56      }
57      // Paso 7
58      return d[n, m];
59     }
60 }
61
62 class Program
63 {
64     static void Main()
```

```
65      {
66          string cad1, cad2;
67  Console.WriteLine ("Instroduzca Cadena No.1:");
68           cad1 = Console.ReadLine();
69           Console.WriteLine ("Instroduzca Cadena No.2:");
70           cad2 = Console.ReadLine();
71           Console.WriteLine("La distancia mínima entre las
72  cadenas es:");
73
74  Console.WriteLine(LevenshteinDistance.Compara(cad1,
75  cad2));
76
      Console.ReadKey();
      }
  }
```

Tabla 2.3.1 Implementación del algoritmo Levenshtein en C#

La siguiente Tabla 2.3.2., nos muestra los pasos y descripción del algoritmo de Levenshtein de la función arriba implementada.

Pasos	Descripción	No. Línea
1	• Definimos n con la longitud de la cadena s. • Definimos m con la longitud de la cadena t. • Si n = 0, retornamos m y salimos • Si m = 0, retornamos n y salimos • Definimos y construimos una matriz conteniendo $0..m+1$ filas y de $0..n$ +1 columnas.	21..35
2	• Inicializamos la primera fila de $0..m$ • Inicializamos la primera Columna de $0..n$	37..38

3	• Examinamos o recorremos cada carácter de s ($i\ de\ 1\ a\ n$).	41
4	• Examinamos o recorremos cada carácter de t ($j\ de\ 1\ a\ m$)	44
5	• Si $s[i]$ es igual a $t[j]$, el costo es igual a 0 • Si $s[i]$ no es igual a $t[j]$, el costo es igual a 1	47
6	• Definimos la celda $d\,[i,j]$ de la matriz igual al mínimo de: A. La celda inmediatamente anterior más 1: $d\,[i-1,j]\ +\ 1$. B. La celda inmediatamente a la izquierda, más 1: $d[i,j-1]\ +\ 1$. C. La celda en diagonal arriba y a la izquierda, más el costo: $d\,[i-1,j-1]\ +\ costo$.	50..55
7	• Después de la etapas de iteración (3, 4 , 5, 6) están completas la distancia se encuentra en $d\,[i,j]$	74

Tabla 2.3.2. Número de pasos y descripción de la función del algoritmo de Levenshtein.

El resultado de la ejecución de la función implementada anteriormente es el siguiente (figura 2.3.3.2):

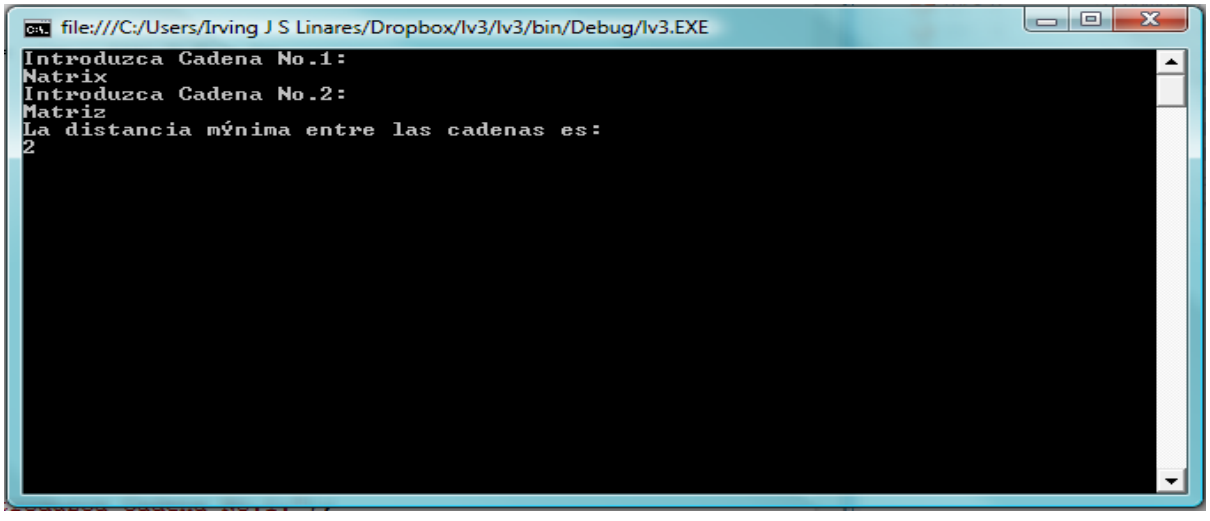


Figura 2.3.3.2 Resultado de la ejecución del algoritmo de Levenshtein.

Tabla 2.3.3. Paso 3 al 6; Cuando $x = 1; y = 1$.

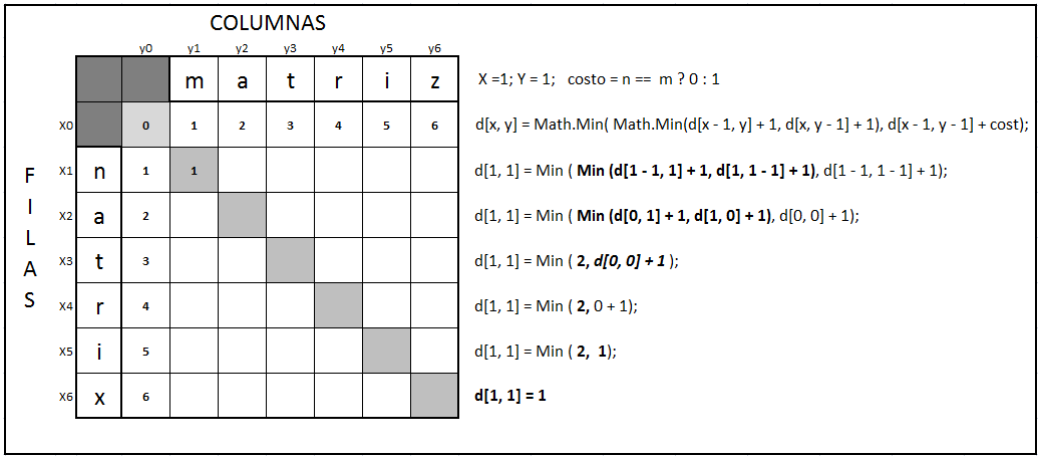


Tabla 2.3.3.1 Paso 3 al 6; Cuando $x = 1; y = 2$.

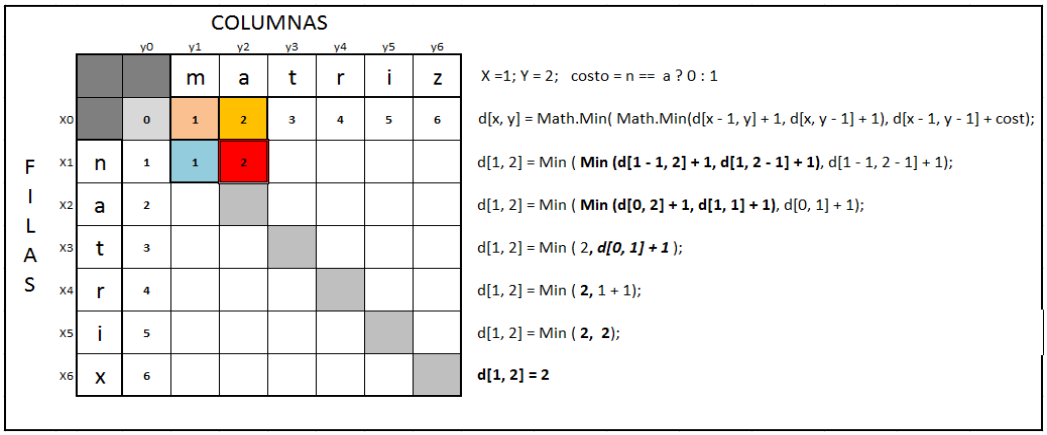


Tabla 2.3.3.2. Paso 3 al 6; Cuando $x = 1; y = 3$.

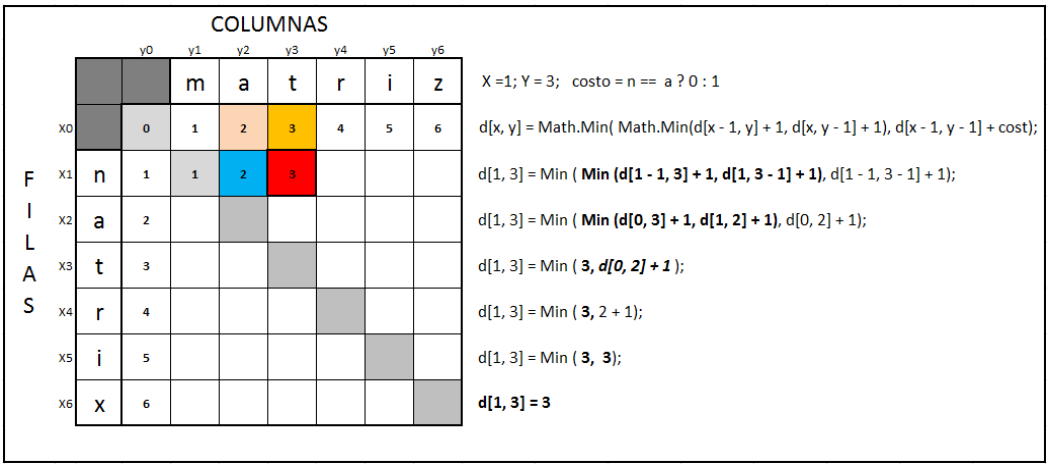


Tabla 2.3.3.3. Paso 3 al 6; Cuando $x = 1; y = 4$.

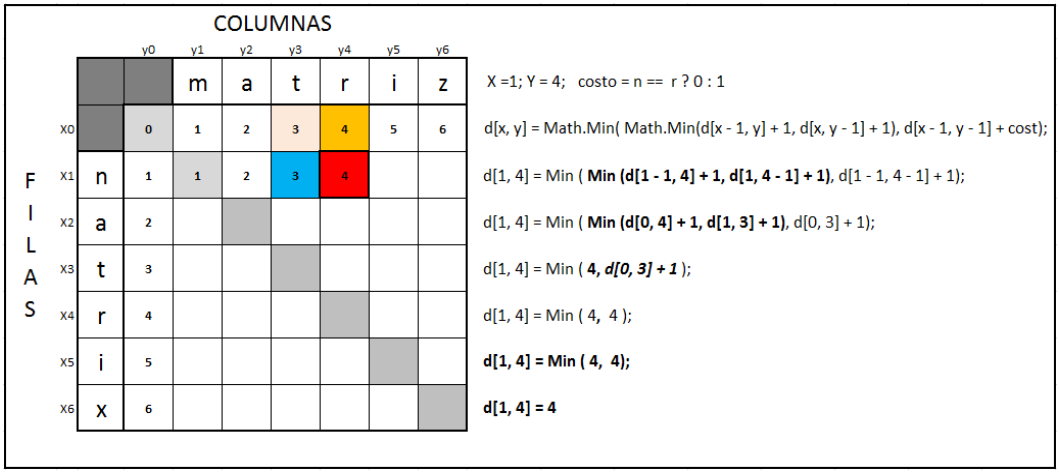


Tabla 2.3.3.4. Paso 3 al 6; Cuando $x = 1; y = 5$.

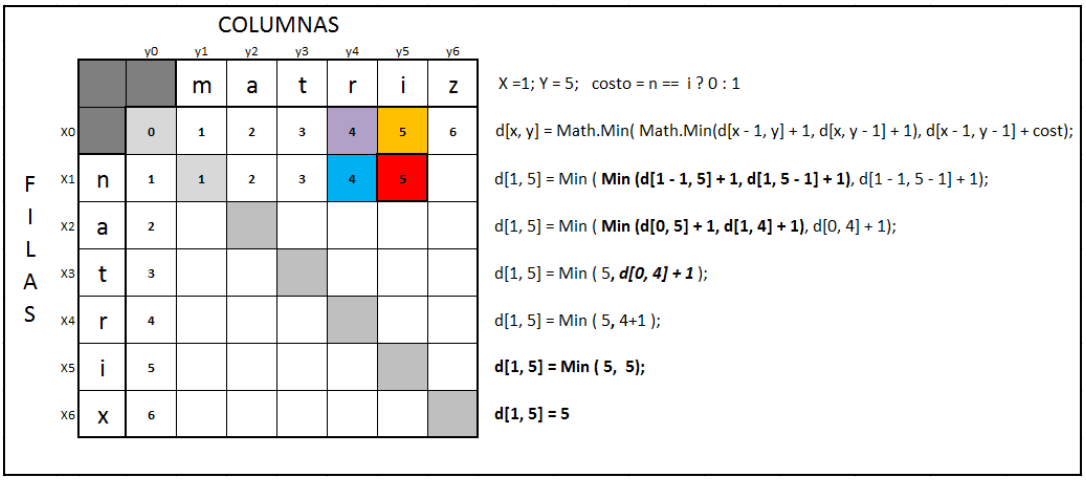
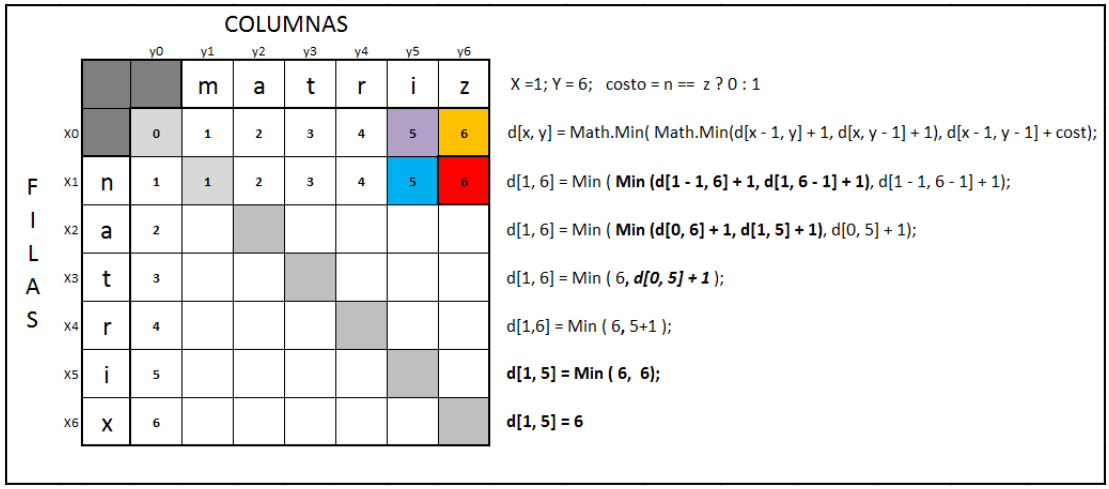


Tabla 2.3.3.5. Paso 3 al 6; Cuando $x = 1; y = 6$.



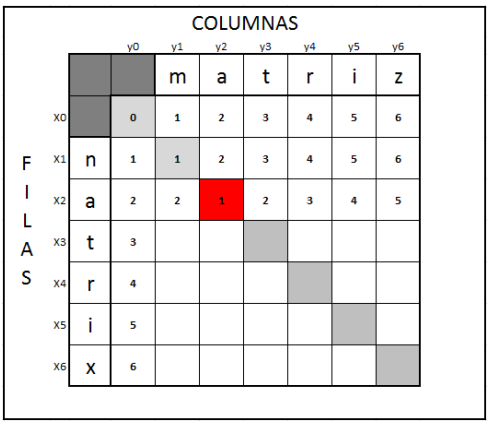


Tabla 2.3.4. Iteración de x = 2.

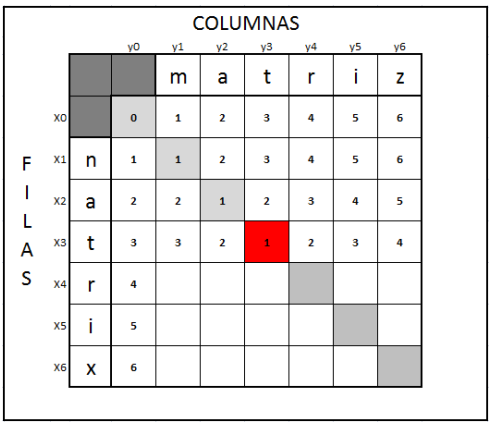


Tabla 2.3.4.1. Iteración de x = 3.

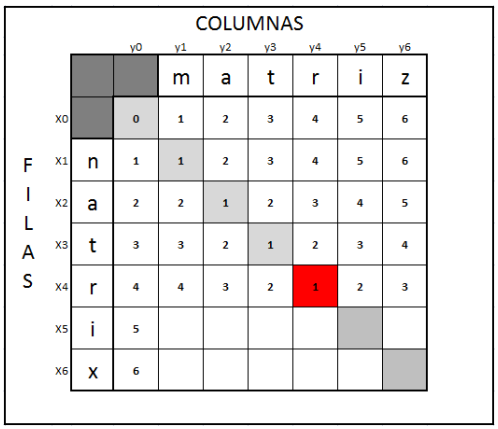


Tabla 2.3.4.2. Iteración de x = 4.

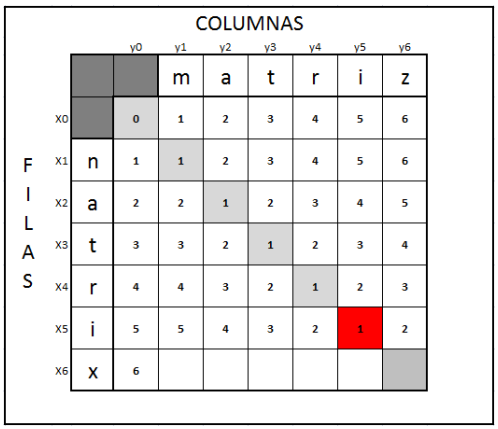


Tabla 2.3.4.3. Iteración de x = 5.

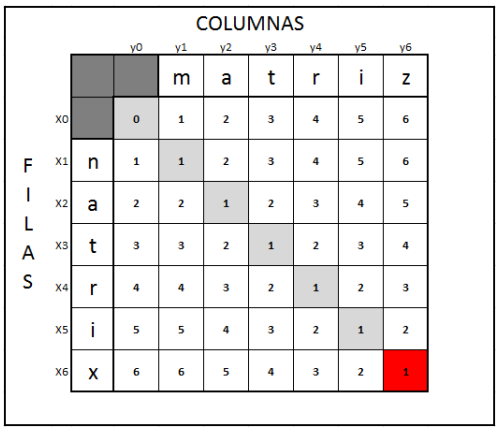


Tabla 2.3.4.4 Iteración de x = 6.

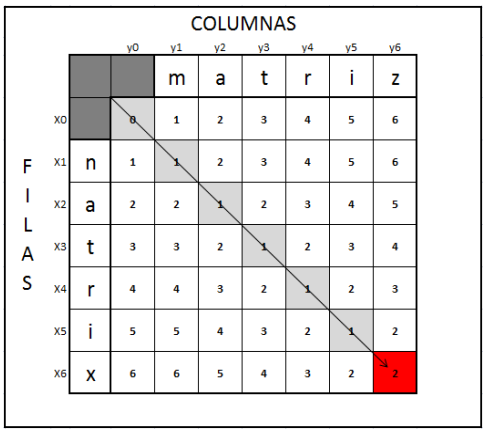


Tabla 2.3.4.5. Resultado Final [x,j] = 2.

- La distancia está en la esquina inferior derecha de la matriz, y es igual a 2.

Capítulo III. Levenshtein sobre la Web

Por alguna razón al iniciar la maestría en ciencias de la computación en mi unidad académica, sabía que me iba a encontrar con un sinfín de retos y problemas, uno de ellos y el principal durante toda mi estancia, fue la elección del tema de tesis, ya que como sabemos el principal objetivo de la maestría es la desarrollar habilidades que te permitan aplicar la ciencia de la computación para solucionar algún tipo de problema (cualquiera que fuese) utilizando el arma fundamental de todo científico de las ciencias de la computación: *"los algoritmos".*

Pero entonces: ¿Cómo y qué algoritmo aplicar para la solución de un problema *que involucraba búsqueda de patrones?,* por lo que me di a la tarea de realizar una investigación exhaustiva de que algoritmos habían sido utilizados como base para desarrollar todas las teorías existentes sobre algoritmos de búsqueda (ya que la gran mayoría de los algoritmos vistos en clase hablaban sobre este tema en particular), así fue como llegue al algoritmo de Levenshtein.

En la teoría de la información y ciencias de la computación, la distancia Levenshtein es una métrica de cadenas para medir la diferencia entre las mismas. Informalmente, la distancia Levenshtein entre dos palabras es el número mínimo de cambios de un sólo carácter (inserción, supresión, sustitución) necesaria para cambiar una palabra en el otro.

La frase distancia de edición se utiliza a menudo para referirse específicamente a la distancia de Levenshtein. Lleva el nombre de Vladimir Levenshtein, quien inventó esta distancia en 1965. Está estrechamente relacionado con las alineaciones de cadenas. Tomando en consideración lo citado anteriormente y habiendo estudiado toda la teoría acerca de la distancia de edición y al mismo tiempo desarrollado previamente el algoritmo de Levenshtein en el lenguaje C#, con la finalidad de encontrar alguna idea que me permitirá aplicar este conocimiento.

Y al mismo tiempo relacionarlo con la problemática que todos tenemos al realizar alguna búsqueda, buscar algún texto o patrón determinado, ya que la gran mayoría de los *"White papers"* hablan de ello, surgió una pequeña idea.

"*Si la Distancia de Edición nos permite obtener el número de cambios que sufrió una cadena para convertirse en otra, como utilizar esta métrica para realizar alguna búsqueda dentro de una navegador de internet".*

Por lo que ahora el problema ya no era tan sólo la aplicación del algoritmo en sí, sino al mismo tiempo que herramientas, tecnologías y lenguajes de programación basados en la Web debería de utilizar, de tal manera que me permitieran desarrollar mi idea.

En consecuencia era de suma importancia conocer acerca de las nuevas tendencias y lenguajes de programación para la Web, así que me puse manos a la obra y a investigar estos 4 lenguajes de programación: *Jquery, JavaScript, CSS y HTML5* que sin lugar a dudas me permitirían concluir mi proyecto.

Mi aplicación basada en la Web está constituida por 3 archivos de manera independiente, pero que todos en conjunto le dan la fuerza y la fortaleza a la aplicación:

Nombre del Archivo	Descripción.
1. Archivo buscapalabra.html	Archivo desarrollado en el estándar *HTML5*
2. Archivo funciones.js	Archivo desarrollado en el lenguaje *JavaScript y Jquery*
3. Archivo style.css	Archivo de Hoja de cascada de estilos.

Los cuales se detallaran en los siguientes puntos del este capítulo.

3.1. Planteamiento de la aplicación (análisis de la métrica)

Dado que el objetivo de desarrollar esta aplicación es realizar una búsqueda de un patrón específico en una cadena determinada y al mismo tiempo indicar la posible corrección ortográfica del patrón buscado en un entorno Web (navegador de Internet), era necesario profundizar en el análisis del algoritmo de Levenshtein para poder determinar la métrica.

Si pensamos por un minuto la infinidad de formas en que las personas comenten errores tipográficos, podemos llegar a una enorme lista, en ciertas ocasiones omitimos letras (como por ejemplo crecimiento por crecimieto), a veces agregamos demasiadas letras (Viirreinato por Virreinato) y algunas veces sustituimos una letra por otra, por todo esto tiene sentido tomar la distancia entre dos palabras para determinar el número más pequeño de tales transformaciones necesarias para convertir una palabra en otra y así conocer la métrica.

Para describir matemáticamente lo que significa una palabra mal escrita a ser “casi” la palabra deseada, forzosamente tenemos que conocer el concepto de métrica. En otras palabras queremos ver nuestro conjunto $\sum$ como un espacio métrico en el que podemos medir la distancia entre dos palabras (cuando nos referimos a un espacio métrico, los llamamos puntos).

Entonces el conjunto de todas las palabras en español validas es $E \subset \sum*$ es un subespacio, para corregir una palabra mal escrita $p \in \sum*$, nosotros podemos simplemente utilizar el punto más cercano en E con respecto a la métrica.

Por ende, la parte más difícil es describir la métrica correcta, y por supuesto antes de llegar allí, hay definir una métrica para que conozcamos qué propiedades necesitamos para la construcción de una métrica de palabras.

Una métrica sobre un conjunto X es una función (llamada la función de distancia o simplemente la distancia), la idea de asignar distancias a pares de puntos es precisamente lo que da origen a los espacios métricos.

Un espacio métrico es par (X, d) donde X es un conjunto no vacio y d es una función real llamada distancia o métrica la cual se define de la siguiente forma:

$$d: X\ x\ X \rightarrow \mathcal{R}$$

Donde $\mathcal{R}$ es el conjunto de los números reales, para todos $x, y, z\ en\ X$, se requiere esta función para satisfacer las siguientes condiciones:

1. $\boldsymbol{d(x,y) \geq 0}$	No negatividad, las distancias son siempre positivas y el único punto a distancia cero de un punto dado es él mismo.
2. $\boldsymbol{d(x,y) = 0}$	Si y sólo si $x = y$.
3. $\boldsymbol{d(x,y) = d(y,x)}$	La distancia es una función simétrica
4. $\boldsymbol{d(x,z) \leq d(x,y) + d(y,z)}$;	Una distancia satisface la desigualdad triangular, la longitud de uno de los lados de un triángulo es menor o igual a la suma de los otros dos lados.

Entonces una edición de $\mathcal{U}\ a\ \mathcal{V}$ es una secuencia de cambios elementales que comienza con $u = u_1 \dots u_k$ y termina en $v = v_1 \dots v_j$, la longitud de una edición es el número de ediciones elementales en la secuencia, finalmente definimos la edición de distancia entre u y v denotando a $d(u, v)$ como la longitud más corta de u a v.

Para verificar que esto sea una métrica, observamos que todos los cambios tienen una longitud no negativa y la única edición de longitud cero es la edición que no hace nada, así que si $d(x, y) = 0$ se infiere que $x = y$, en segundo lugar observamos que las ediciones son simétricas inherentemente, por ejemplo que si tenemos una edición de $x\ a\ y$ podemos simplemente invertir la secuencia y tenemos una edición valida de $y = x$.

Por último y no menos importante, tenemos que verificar la desigualdad triangular. Sean x, y, z palabras, por lo que queremos demostrar que $d\ (x, z) \leq d(x, y) + d(y, z)$, tomamos dos ediciones más cortas entre $\{x,y\}$ y $\{y, z\}$ y observamos que su composición es una edición valida de $x\ a\ z$. Siguiendo nuestra definición, por "componer", entendemos combinar las dos secuencias de operaciones en una secuencia de la manera más obvia, así que como se trata de una edición, esta longitud puede ser menor que la más corta de x a z (ver figura)

.

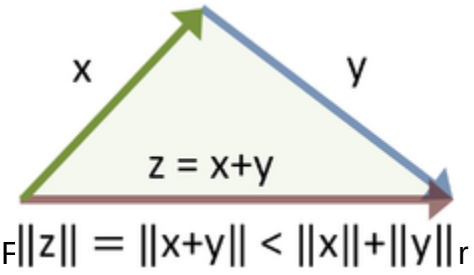


Figura 3.1. Ejemplo de la Desigualdad Triangular

Por lo tanto d es de hecho una métrica, e históricamente se llama métrica de Levenshtein.

3.2. El núcleo de la aplicación.

Habiendo recapitulado toda la teoría necesaria para el desarrollo de la aplicación llegamos al punto más importante: ¿Qué lenguaje de programación utilizar para hacer esto posible?,. Sin lugar a dudas esta fue la parte más difícil y complicada del desarrollo y para entender un poco más el código que se detallará en los siguientes puntos del capítulo, analizaremos el lenguaje esencial para dicha tarea: *el JQuery.*

Hoy en día el Internet es un entorno totalmente dinámico, y los desarrolladores se enfrentan a grandes obstáculos en la creación y desarrollo de sitios con grandes funcionalidades y estilos. Para hacer esto posible los desarrolladores están recurriendo a las bibliotecas de *JavaScript,* como *jQuery* para automatizar tareas comunes y simplificar las complicadas, una de sus principales bondades es su capacidad para ayudar en una amplia gama de tareas a las que antes no se tenía acceso.

Saber por dónde empezar puede parecer difícil, porque *jQuery* realiza tantas funciones diferentes que no imaginamos, sin embargo, existe una coherencia y simetría en el diseño de la biblioteca, y la mayoría de sus conceptos son tomados de la estructura de las hojas de estilo *(CSS) y HTML.*

3.3. ¿Qué hace *jQuery*?

La biblioteca *jQuery* proporciona una capa de abstracción de propósito general para scripting web común (common web scripting), y por tanto es útil en casi todas las situaciones de script.

Su naturaleza extensible significa que nunca podríamos cubrir todos los posibles usos y funciones en este trabajo, ya que constantemente se están desarrollando *plugins* para añadir nuevas capacidades y funcionalidades, sin embargo las características fundamentales son las siguientes:

- Elementos de acceso en un documento (*Access Elements in a document*). Sin una biblioteca JavaScript, son muchas líneas de código deben de ser escritas para recorrer el árbol *(Document Object Model – Modelo Objeto Documento)* y localizar partes específicas de la estructura de un documento *HTML*, un robusto y eficiente mecanismo selector ofrece *jQuery* para recuperar la pieza exacta del documento que se va a inspeccionar o a manipular.

- Para modificar la apariencia de una página web: *CSS* ofrece un potente método de influir en la forma en que un documento se renderiza, pero en ocasiones se queda corto cuando los navegadores web no soportan los mismos estándares. Con *jQuery*, los desarrolladores pueden llenar este vacío, confiando en el mismo soporte de estándares en todos los navegadores.

- Alterar el contenido de un documento. No se limita a meros cambios de apariencia, *jQuery* puede modificar el contenido de un documento en sí mismo con sólo pulsar unas teclas. El texto puede ser cambiado, las imágenes se pueden insertar o permutas, las listas pueden ser reordenadas, o toda la estructura del *HTML* puede ser reescrita y ampliada, todo con una única y fácil de usar interfaz de programación de aplicaciones *(API).*

- Responder a la interacción del usuario. Incluso los comportamientos más complejos y poderosos no son útiles si no podemos controlar el momento en que tienen lugar. La biblioteca *jQuery* ofrece una forma elegante para interceptar una amplia variedad de eventos, como por ejemplo un usuario haga clic en un enlace, y sin la necesidad de recargar el propio código HTML con controladores de eventos. Al mismo tiempo, su *API* de gestión de eventos elimina inconsistencias de navegación que a menudo plagan los desarrolladores web.

- Animar cambios hechos a un documento. Para aplicar efectivamente tales comportamientos interactivos, un diseñador también debe proporcionar información visual al usuario. La biblioteca *jQuery* facilita esto al ofrecer una serie de efectos como desvanecimientos y deslizamientos, así como un conjunto de herramientas para la elaboración de otros nuevos.

- Recuperar información de un servidor sin actualizar la página. Este patrón de código se ha conocido como *Asynchronous JavaScript And XML (AJAX),* y ayuda a los desarrolladores web en la elaboración de un sitio sensible, rico en funciones. La biblioteca *jQuery* elimina la complejidad específica del navegador de este proceso, lo que permite a los desarrolladores centrarse en la funcionalidad de servidor de servicios de fondo.

- Simplifica las tareas comunes de JavaScript. Además de todas los características específicas de la estructura del documento *jQuery*, la biblioteca proporciona mejoras básicas en la construcción de JavaScript como la iteración y la manipulación de matrices.

3.4 Desarrollo de la aplicación basada en la Web.

Una vez analizada de manera integral la librería *jQuery* y conociendo la potencia que implicada utilizarla en sistemas basados en la *Web*, me di cuenta que el único lenguaje que podía utilizar era *HTML 5* y la librería *jQuery*, por lo que para poder desarrollar la aplicación, tuve que implementar el siguiente archivo llamado *funciones.js*, utilizando *jQuery*, que prácticamente representa el núcleo de la aplicación.

- Script funciones.js

```
1  String.prototype.endsWith = function(suffix)  {  /*aquí estoy modificando el prototipo String, y le
 estoy agregando a todo objeto String la función endsWith*/


2       return this.indexOf(suffix, this.length - suffix.length) !== -1;
3  };
4
5   function Levenshtein(a, b) {
6          var n = a.length;
7         var m = b.length;
8
9          // Creación de matriz de cambios mínimos
10              var d = [];
11
12              // si una de las dos está vacía, la distancia es insertar todas las otras
13        if(n == 0)
14                return m;
15        if(m == 0)
16                return n;
17
18
19                for (var i = 0; i <= n; i++)
20                      (d[i] = [])[0] = i;
21                    for (var j = 0; j <= m; j++)
22                        d[0][j] = j;
```

```
23
24
25                  for (var i = 1, I = 0; i <= n; i++, I++)
26                    for (var j = 1, J = 0; j <= m; j++, J++)
27                          if (b[J] == a[I])
28                              d[i][j] = d[I][J];
29                          else
30                              d[i][j] = Math.min(d[I][j], d[i][J], d[I][J]) + 1;
31
32                    // el menor número de operaciones
33                      return d[n][m];
34 }
35
36     function cambiarProcesado() {

37             $("#cajaTexto textarea").hide();/*desaparezco el textarea*/
38             $("#cajaTexto #procesado").css("display","inline-block");/*le digo que el display de
procesando ya no sea none ahora sea inline-block*/
39            $("#cajaTexto #procesado").hide().fadeIn();/*aparezco el contenedor de texto procesado, el
article*/
40      }
41
42    function cambiarNuevo() {
43          $("#cajaTexto #procesado").hide();/*desaparezco el article con id procesado*/
44          $("#cajaTexto testarea").hade().faldean();/*aparezco el textoarea*/
45    }
46
47    function EncontrarOnKeyUp(evt){//cargar al dar enter en el campo de texto de la pista
48       var e = window.event || evt; // for trans-browser compatibility
49       var tecla = e.which || e.keyCode;

50      if (tecla == 13){Encontrar();}
51    }
52
53    function Encontrar() {
```

```
54
55          if ($('#findWord').val().length > 0) {  //si el campo de pista no está vacio
56
57            var shortest = -1;
58            var matchlev = null;
59            var sm = " ";
60            var i = 0;
61            var count = 0;
62
63           var pista = $('#findWord').val(); //Obtengo la pista
64           var texto = $('#my_textarea').val(); //Obtengo el texto en el textarea
65           var salida = ""; //Aquí voy a poner el texto que luego colocare en el article
66            var coin = 0; //coincidencias
67            var aprox = 0; //aproximaciones
68            var saprox = "";
69            var comit = "";  //carácter omitido
70            var palabra = ""; //palabra actual
71
72            var palabras = texto.split(" "); //convierto el texto del textarea en un arreglo de palabras que por
cierto son String, y por esa razón todos tienen ya la función endsWith


73              for (var i=0; i <= palabras.length-1; i++)  {//hago un recorrido del arreglo de palabras
74                comit = ""; //carácter omitido
75                palabra = ""; //palabra actual
76
77                 var temp = palabras[i];  //necesito quitarle antes puntos y comas
78
79        if (temp.endsWith(",") || temp.endsWith(".")) { //si la palabra temporal acaba en punto o coma,
hay que quitársela para procesar la palabra
80          palabra = temp.substring(0,temp.length-1); //obtengo la palabra
81          comit = temp.substring(temp.length-1); //extraigo y deposito aquí el carácter sobrante
82       }
83      else
84          palabra = temp; //si no hay nada que le sobre a la palabra simplemente la deposito en la variable
de palabra actual
```

```
85
86      var lev = Levenshtein(palabra, pista);//utilizo el algoritmo
87
88        if (lev ==0 ) {//si la distancia es 0
89              matchlev = pista; //martchlev es igual a pista
90              shortest = 0; //pongo esta bandera en 0
91              coin++; //le digo que hay una coincidencia mas
92             salida+="<div class='coincidencia'>" + palabra + "</div>" + comit + " "; //y pongo en rojo la
palabra antes de agregarla a la cadena de salida, y si existe algún carácter omitido que haya extraído lo
concateno
93       }
94    else
95        salida +=palabra + comit + " " ;//si la distancia es diferente de 0, simplemente concateno la palabra
sin estilo y concatenando el carácter omitido
96
97          if (lev <= shortest || shortest < 0) {//bueno esto lo deje igual
98              matchlev = pista;
99              shortest = lev;
100             sm = palabra;
101          }
102
103      if (shortest == 0)    //si encontré la palabra exacta le digo que encontré el patrón exacto
104             $("#aproximacion").text("patrón exacto encontrado");
105      else   //sino le digo cual fue la que más se parece
106            $("#aproximacion").text('Quiso decir "' + sm + '"?');
107      }
108
109    $("#cajaTexto #procesado").html(salida); //para casi finalizar pongo todo el texto de salida con todo
y los estilos, dentro de article procesado
110    $("#patron").text('"' + pista + '"'); //pongo en el área de información que el patrón es la pista
111    $("#cantidad").text(coin); //le digo cuantas veces coincidió la palabra,  en la primera línea uso html,
en las otras dos text
112 //la diferencia es que html respeta el significado de las tags de html y el text lo agarra como si fuera
texto simplemente
113 //en la primera ocupo las tags, así que conviene usar html
```

```
114
115   cambiarProcesado(); //desaparezco el textarea y aparezco el article con todo y el contenido que
acabo de hacer
116     }
117       else { // Si no había nada en el cambio de pista
118           $('#findWord').focus(); // Mando el foco a el campo de pista
119       }
120   }
121
122     function nuevo() {//este método es simplemente para borrar el contenido del campo findword (en
el que se pone la pista)
123     $("#findWord").val("");
124     $("#findWord").focus(); //Pongo el foco en el campo de la pista
125     cambiarNuevo(); //Y desaparezco el article con el texto procesado, para aparecer el textarea
126 }
```

Sin lugar a dudas este fue la parte más difícil y complicad del desarrollo y para entender un poco más el código anteriormente expuesto, analizaremos a fondo las líneas más importantes del mismo.

En las primeras líneas de código del 1 al 3, ocurre algo muy pero muy especial ya que *JavaScript* no contiene funciones para trabajar con cadenas, que me permitieran eliminar ciertos caracteres de las palabras, *jQuery* es un tipo de lenguaje diferente a OOP, es orientado a prototipos.

Esto significa que las variables que se crean se derivan de un modelo predefinido, llamado un prototipo, no es u*na definición de clase, aquí prácticamente estoy modificando el prototipo* String, y le estoy agregando a todo objeto *String* la función *endsWith.*

En la línea número 2, utilizo el método *indexOf ()* que devuelve la posición de la primera aparición de un valor especificado en una cadena, y este método devuelve

-1, si el valor de búsqueda nunca ocurre, y termino en la línea número 3 con la finalización de la modificación del prototipo *String.*

De los números de código o líneas 5 a la 34 se encuentra definida la función del algoritmo de Levenshtein, dicha función será utilizada en el número de línea 86.

En la línea 36, defino una función llamada *cambiarProcesado()*, en la cual como primer paso en la línea número 37, se define un selector *jQuery* que almacena el elemento *textarea* que se encuentra adentro del *section (línea 34 del archivo busca palabra.html, <section id="detalles">)* con *id=”cajaTexto”* y después a ese elemento llamado *textarea* lo esconde por medio del método *.hide*, $("#cajaTextotextarea").hide();., previamente establecido en el archivo *css (línea 50: #cajaTextotextarea{).* En otras palabras desaparezco el *textarea* y aparezco el *article* (línea 29 y 31 del archivo buscapalabra.html) con todo y el contenido que acabo de procesar.

En la línea 38, nuevamente ocupamos un selector pero esta vez para encontrar el elemento *con id=”procesado”,* que se encuentra dentro del *section* con *id=”cajaTexto”*, y en esta ocasión cambio la propiedad de estilo “display” de *none* (significa no mostrar el elemento) a *inline-block* (bloque en línea) para que se muestre en el lugar de *textarea*.

En la línea 39, se utiliza el mismo selector de la línea anterior para ubicar el objeto con *id = “procesado”*, para aparecerlo con un efecto de difuminación hacia dentro, y termínanos en la línea número 40, con la finalización de la función *cambiarProcesado()*, que tiene la finalidad de ocultar el área del texto a procesar para mostrar el texto procesado.

En la línea 42, definimos la función *cambiarNuevo(),* que es el proceso inverso de la función anterior en la cual ahora se oculta el contenedor de texto procesado para mostrar nuevamente el área de texto activo (texto o cadena inicial).

En la línea 47, se define la función *EncontrarOnKeyUp*, que recibe como parámetro el evento producido por el objeto que lo accionó *“evt”*, que contiene todas las propiedades del evento, accionada por el teclado (usuario) en el campo de texto don *id=”findWord”*, y escucha las teclas pulsadas, si la tecla corresponde a un “Enter”, entonces se realiza una llamada a la función *Encontrar (que es en esta función donde se realiza la parte esencial del archivo funciones.js)*.

Para finalizar las líneas 48 y 49 se utilizan para proporcionar compatibilidad entre navegadores, ya que en ciertas ocasiones se interpreta de diferente forma y con esto aseguramos que pueda ser utilizado desde cualquier navegador, la línea 50 tiene un sentencia *if,* que condiciona a que si la tecla que pulse el usuario fue *Enter* llama a la función *Encontrar.*

En la línea 53, se declara la función *Encontrar ,* la línea 55 contiene una condición *if*, en la cual se compara que la longitud el campo de texto *#findword* sea mayor que 0, esto implica que el usuario escribió un patrón a buscar, esta definición del selector *$('#findWord').val().length> 0)* es lo que le da la fuerza y fortaleza al *jQuery,* ya que en automático compara el valor introducido en el campo si es mayor que cero inicia la búsqueda del patrón introducido, sino de lo contrario nuevamente regresa el foco al campo de texto *#findWord (línea 120): $('#findWord').focus();.* De las líneas 57, 58, 59, 60, 61, se definen una serie de variables.

En la línea 65 se definió una variable de nombre *salida* que almacenara el conjunto de palabras evaluadas por el algoritmo de *LV*, en donde en la línea 92 se asume que el algoritmo encontró el patrón buscado y procederá a señalizar dicho patrón almacenado en la variable *palabra* encerrándola con una etiqueta semántica DIV con un estilo que se define en la clase coincidencia (línea 38 del archivo *style.css)* y posteriormente se le agrega los caracteres omitidos en caso de existir en caso contrario (línea 94) a la variable salida se le agregara únicamente la palabra actual sin ser remarcada con un estilo especial y el carácter omitido en caso de existir.

En las líneas siguientes 66, 69, 70, se definen una serie de variables, para saber las coincidencias, los caracteres omitidos, y la palabra actual.

En la línea 72, se encuentra la siguiente instrucción: *var palabras = texto.split(" ");, q*ue nos permite convertir el texto del *textarea*, definido en la línea 64 (*var texto = $('#my_textarea').val();*), en un arreglo de palabras que por cierto son *String*, recordemos que y por esa razón todos tienen ya la función *endsWith*, y en el momento que se vuelve un arreglo adquiere todas las propiedades del prototipo *String*. El método *split ()* se utiliza para dividir una cadena en un arreglo o matriz de subcadenas, y devuelve un nuevo arreglo.

En la línea 73, se encuentra un ciclo *for* que nos permite obtener la longitud de la del arreglo llamado *palabras: for (var i=0; i <= palabras.length-1; i++); en* las dos siguientes líneas 74 y 75 se definen dos variables *comit (carácter omitido) y palabra (palabra actual).*

En la línea 77, se define la variable llamada *temp: var temp = palabras[i];,* ya que es necesario tener dos variables de tipo *string* (en este arreglo), para poder quitarle los puntos y comas al texto original que se está analizando, recordemos que esta declaración se encuentra donde del ciclo *for* y la cantidad de interacciones será desde *i=0;* hasta la longitud del arreglo llamado *palabras* menos uno, *i <= palabras.length-1; i++.*

Ahora que cada palabra del texto original almacena en la variable *temp, (línea 72: var palabras = texto.split(" "); y línea 77: var temp = palabras[i];),* está siendo analizada por la propia interacción del ciclo *for,* podemos quitarle los puntos y comas que tienen cada una de las palabras analizadas, y esto sucede de las líneas 79, 80, 81 y 82.

En la línea 79, se realiza una sentencia de comparación *if: (if (temp.endsWith(",")* || *temp.endsWith(".")* || *temp.endsWith(";")) {),* y esto indica que si la condición es verdadera, o que si la palabra almacena en la variable *temp*, termina o acaba en una “coma”, “punto” o “punto y coma”, realice la instrucción que se encuentra en la

línea 80: *palabra = temp.substring(0,temp.length-1);.* El método substring () extrae los caracteres de una cadena, entre los dos índices especificados, y devuelve la nueva cadena, esta nueva cadena es almacenada en la variable llamada *palabra,* por lo que ahora en la línea 81 se extrae el ultimo carácter de la palabra que está siendo analizada: *comit = temp.substring(temp.length-1); y* se almacena en la variable *comit (carácter omitido),* todo esto ocurre siempre y cuando la condición sea verdadera.

En caso contrario, si no hay nada que le sobre a la palabra analizar (suponiendo que solamente se analizara un palabra sin puntos ni comas) la palabra (cadena) del arreglo que está siendo analizada *(temp)* se le asigna directamente a la variable llamada *palabra,* línea 84: *palabra = temp;*.

En la línea 86 se encuentra la instrucción de código que manda a llamar a la función *Levenshtein: var lev = Levenshtein(palabra, pista);,* la cual recibe como parámetros dos argumentos, la variable *palabra*, previamente depurada que proviene del arreglo *palabras* y el segundo parámetro que es la palabra en este caso patrón a buscar dentro del texto (*variable pista)*, y el valor que es devuelto por la función *Levenshtein* es asignado a una nueva variable llamada *lev,* previamente definida en la línea 84 (definición de la variable al vuelo), cabe recordar que el la función devuelve un valor numérico. En la línea 88, se encuentra la línea de código que evalúa si la el valor de la variable *lev (variable que trae el valor retornado de la función Levenshtein)* es igual a cero, entonces, el valor de la variable *pista* es asignado a la variable *matchlev* (línea 89) y a la variable *shortest* se le asigna un valor de cero, que implica que se ha encontrado la distancia más corta (línea 90), también se incrementa en uno la variable llamada *coin,* que contabiliza las coincidencias encontradas.

Por último en la línea 92 se concatena la palabra comparada o previamente analizada con los caracteres omitidos (variable *comit)*, pero esta línea tiene algo muy particular ya que la concatenación la realiza siempre y cuando el patrón a buscar sea exacto ya que se encuentra dentro de la condición y por ende lo pone en color rojo en el nuevo arreglo de palabras (variable *salida)* y esto es posible a la

clase llamada *coincidencias (" "),* que es invocada o llamada dentro de la misma línea de código.

Esta línea en particular es lo que le da la vistosidad al resultado mostrado en pantalla (patrón encontrado de color rojo), en caso contrario (línea 94) concatena la palabra comparada o previamente analizada con los caracteres omitidos sin ningún color, ya que si no se encuentra el patrón de cualquier forma se tienen que concatenar las *palabras* previamente comparadas con los caracteres omitidos, porque es esta variable llamada *salida,* la que se muestra en el resultado final en la pantalla.

En la línea 97, se realiza una condición, si esta distancia es menor que la siguiente distancia más corta encontrada *(lev* <= *shortest*) o si la siguiente palabra más corta aún no se ha encontrado (shortest < 0), se establecen la distancia más corta y la coincidencia más cercana (línea 99 y 100), y la línea 98 simplemente sirve para asignarle el valor de la variable *pista* a la variable *matchlev,* únicamente con fines ilustrativos antes de que conociera la potencialidad del *jquery* conjuntamente con el *css (hoja de cascada de estilos).*

Esto debido a que conforme iba puliendo el algoritmo y encontrado ideas o formas más poderosas de visualizar la información esa línea me servía para saber cuál era la palabra buscada y asignarla a una variable llamada *matchlev,* para posteriormente mostrarla.

En la línea 103, se evalúa una condición que compara que si el valor de la variable *shortest* es igual a cero, en otras palabras, si el patrón es encontrado entonces se define un selector con *id="aproximación"* y al método *text* se le asigna la cadena: "patrón exacto encontrado", el método tiene una característica esencial que no respeta las etiquetas HTML y inserta todo como texto plano (línea 104).

Y en caso contrario (línea 106) si la distancia no es igual a cero, implica que no hay coincidencia del patrón buscado pero si una aproximación, esta aproximación es almacenada en la variable *sm*, y el método *text* inyecta o establece la cadena *“Quiso decir” + sm + “?”,* al *id $("#aproximacion").*

Este *id=”aproximación”* se encuentra en el archivo *buscapalabra.html*, dentro de la etiqueta *section* que engloba desde la línea 34 hasta la línea 47, y esta llamada de *id* se encuentra en la línea 45, que es precisamente esta línea de código que muestra el mensaje *“patrón exacto encontrado”, o “Quiso decir”.*

En la línea 109 primeramente ubica el elemento con id=”procesado” que se encuentra dentro del *section* con *id=”cajaTexto”,* y se establece en él, el contenido de la variable *salida*, pero esta vez con el método *HTML*, que respeta la misma nomenclatura del propio lenguaje *HTML*, esto último me permite poner todo el texto que se encuentra almacenado en la variable de tipo arreglo llamada *salida* con todo y los estilos, dentro de *article procesado (líneas 29 hasta la 31 del archivo buscapalabra.html).*

En la línea número 110 se define un selector que busca al elemento con id =”patrón” *(línea 37 en el archivo llamado busca palabra.html, -)* y le asigna o concatena el contenido de la variable pista, con el método *text* que no respeta *HTML.*

En la línea 111, se define in selector que busca el elemento con el *id=”cantidad”* (línea 41 del archivo *buscapalabra.html*, * -)* y se le asigna el contenido de la variable *coin* que almacena las coincidencias encontradas por el algoritmo, por medio del método *text*, *coin* se incrementa en 1, en la línea 91 cada vez que encuentra una coincidencia.

En la línea 115 mandamos a llamar la función *cambiarProcesado();*, analizado anteriormente (línea 36).

En la línea 117 y 118 en caso de no haber patrón a buscar en la caja de texto *findWord,* se define el selector buscando el elemento con *id =”findWord”* (línea 22 del archivo *buscapalabra.html*; *<input id="findWord" type="text" placeholder="¿Qué palabra busca?" onkeyup="EncontrarOnKeyUp();" />*) y se enfoca a través del método *focus,* la línea 119 contiene la llave de cierre del caso contrario y la línea 120 contiene la llave de cierre de la función *Encontrar().*

Por último en la línea 121 de define la función *nuevo*, que tiene tres líneas de código, la primera (122) establece a un valor de espacio vacío el campo de texto con el *id=findword* a través del método *val*, esta función es invocada en la línea 24 del archivo *buscapalabra.html (<buttononclick="nuevo();">Nuevo</button>*).

La siguiente línea 123 define un selector que busca el elemento con *id=findWord* y lo enfoca, para finalizar la línea 125 manda a llamar la función *cambiarNuevo()* analizado anteriormente en la línea 42, que simplemente desaparece el *article* con el texto previamente procesado, para reaparecer el texto original contenido en el *textarea (línea 28 del archivo buscapalabra.html),* y por último la línea 126 se encuentra la llave *“}”* que termina la función principal.

- **Archivo Buscapalabra.html**

Este archivo fue codificado bajo un nuevo estándar de programación ***HTML5,*** que es la última evolución de la norma que define el estándar *HTML*. El término representa dos conceptos diferentes, se trata de una nueva versión del lenguaje *HTML*, con nuevos elementos, atributos y comportamientos, y un conjunto más amplio de tecnologías que permite el desarrollo de sitios Web y aplicaciones más diversas y de gran alcance.

Diseñado para ser utilizable por todos los desarrolladores de *Open Web*, por otro lado cuenta con numerosos recursos sobre las diferentes tecnologías que lo integran, que se clasifican en varios grupos según su función y por mencionar algunas de ellas:

- Semántica: lo que le permite describir con mayor precisión cuál es su contenido.
- Conectividad: lo que le permite comunicarse con el servidor de formas nuevas e innovadoras.
- Desconectado y almacenamiento: permite a páginas web almacenar datos, localmente, en el lado del cliente y operar fuera de línea de manera más eficiente.
- Multimedia: permite hacer vídeo y audio de ciudadanos de primera clase en la Web abierta.
- Gráficos y efectos 2D/3D: permite una gama mucho más amplia de opciones de presentación.
- Rendimiento e Integración: proporcionar una mayor optimización de la velocidad y un mejor uso del hardware del equipo.
- Dispositivo de Acceso: admite el uso de varios dispositivos de entrada y salida.
- *Styling*: deja a los autores escribir temas más sofisticados.

La especificación *HTML5* trae muchos nuevos elementos a los desarrolladores web, permitiéndoles describir la estructura de un documento web con semántica estandarizada, es así como se codifico el archivo antes mencionado el cual se describe a continuación (figura 3.4).

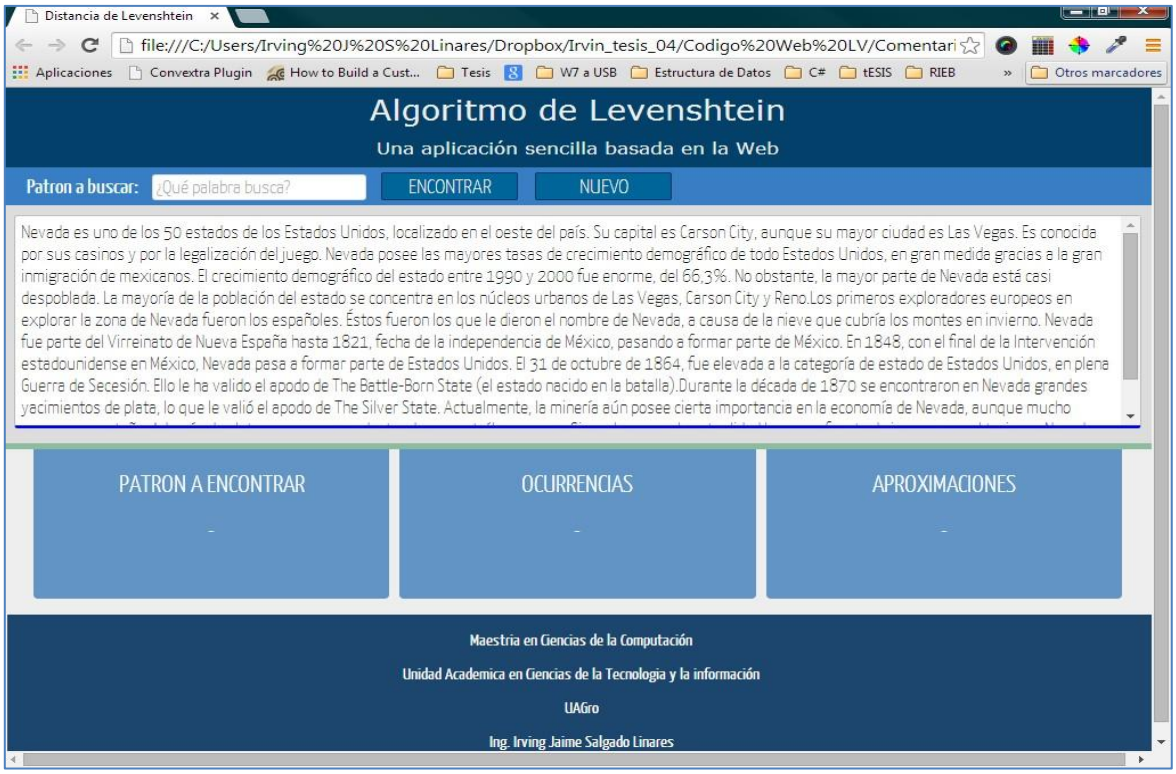


Figura 3.4. Pantalla principal de la aplicación.

- Archivo *HTML5: buscapalabra.html*

```
1 <!DOCTYPE html>
2 <html lang="es">
3<head>
4 <meta charset="UTF-8"/>
5 <title>Distancia de Levenshtein</title>
6
7 <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
8 <script src="funciones.js"></script>
9 <link rel="stylesheet" href="style.css"/>
10<link href='http://fonts.googleapis.com/css?family=Yanone+Kaffeesatz:400,300,200' rel='stylesheet'
type='text/css'>
11 </head>
```

```
12
13  <body>
14   <div id="cabeza">Algoritmo de Levenshtein</div>
15   <div id="cabeza2">Una aplicación sencilla basada en la Web </div>
16
17
18
19    <nav>
20
21    <div class="titulob">Patrón a buscar:</div>
22       <input id="findWord" type="text" placeholder="¿Qué palabra busca?"
onkeyup="EncontrarOnKeyUp();" />
23      <buttononclick="Encontrar();">Encontrar</button>
24      <buttononclick="nuevo();">Nuevo</button>
25   </nav>
26
27   <section id="cajaTexto">
28       <textarea id="my_textarea" placeholder="Aquí va el texto entre el que se va a hacer la
búsqueda">Nevada es uno de los 50 estados de los Est
ados Unidos, localizado en el oeste del país. Su capital es Carson City, aunque su mayor ciudad es Las
Vegas. Es conocida por sus casinos y por
la legalización del juego. Nevada posee las mayores tasas de crecimiento demográfico de todo Estados
Unidos, en gran medida gracias a la gran i
nmigración de mexicanos. El crecimiento demográfico del estado entre 1990 y 2000 fue enorme, del 66,3%.
No obstante, la mayor parte de Nevad
a está casi despoblada. La mayoría de la población del estado se concentra en los núcleos urbanos de Las
Vegas, Carson City y Reno. Los primer
os exploradores europeos en explorar la zona de Nevada fueron los españoles. Éstos fueron los que le
dieron el nombre de Nevada, a causa de la
nieve que cubría los montes en invierno. Nevada fue parte del Virreinato de Nueva España hasta 1821, fecha
de la independencia de México, pasa
ndo a formar parte de México. En 1848, con el final de la Intervención estadounidense en México, Nevada
pasa a formar parte de Estados Unidos.
El 31 de octubre de 1864, fue elevada a la categoría de estado de Estados Unidos, en plena Guerra de
Secesión. Ello le ha valido el apodo de The
Battle-BornState (el estado nacido en la batalla).Durante la década de 1870 se encontraron en Nevada
grandes yacimientos de plata, lo que le vali
ó el apodo de The Silver State. Actualmente, la minería aún posee cierta importancia en la economía de
Nevada, aunque mucho menos que antañ
o. Además de plata, es un gran productor de oro, petróleo y arena. Sin embargo, en la actualidad la mayor
fuente de ingresos es el turismo. Nevad
```

```
a es famosa por sus casinos, que se concentran en Las Vegas y en Reno.</textarea>
29 <article id="procesado">
30
31 </article>
32 </section>
33
34 <section id="detalles">
35        <article>
36          <h2>PATRÓN A ENCONTRAR</h2>
37 <div class="detalle" id="patron">-</div>
38        </article>
39     <article>
40          <h2>OCURRENCIAS</h2>
41          <div id="rasgo"><div class="detalle" id="cantidad">-</div></div>
42     </article>
43    <article>
44           <h2>APROXIMACIONES</h2>
45           <div id="rasgo"><div class="detalle" id="aproximacion">-</div></div>
46     </article>
47   </section>
48   <footer>
49      <p>Maestría en Ciencias de la Computación</p>
50      <p>Unidad Académica en Ciencias de la Tecnología y la información</p>
51 <center><p>UAGro</p></center>
52 Ing. Irving Jaime Salgado Linares
53 </footer>
54
55 </body>
56 </html>
```

Línea 1, *<!DOCTYPE.html>,* esta etiqueta le dice al motor de render que se está trabajando con el estándar *HTML5*, parte fundamental que el navegar identifica y reconoce que se está trabajando con el *HTML5*

Línea 2, *<html lang="es">*, cabecera del documento con un atributo que menciona que lenguaje se estará usando, de esta manera los traductores saben desde que lenguaje deben traducir, y el lenguaje utilizado es el español.

Línea 3*, <head>,* inicio de la cabecera, que es el área del documento donde se importan especificaciones y demás recursos que se utilizaran posteriormente.

Línea 4, *<meta charset="UTF-8"/>,* esta es una etiqueta de meta información, en este caso se utiliza para establecer que el juego de caracteres que se utilizara es *UTF-8*, que soporta caracteres especiales como "ñ" y tildes.

Línea 5*, <title>Distancia de Levenshtein</title>,* titulo de la página, es lo que aparecerá en el texto de la pestaña del navegador.

Línea 7, *<script type="text/javascript" src=http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js></script>,* importo el framework de *jquery.*

Línea 8, *<script src="funciones.js"></script>,* importo el archivo de funciones personalizadas que en conjunto forman la lógica de la aplicación (núcleo), archivo funciones.js. Línea 9, *<link rel="stylesheet" href="style.css"/>,* importo los estilos propios de la aplicación.

Línea 10, *<linkhref='http://fonts.googleapis.com/css? Family= Yanone+Kaffeesatz: 400,300,200' rel='stylesheet' type='text/css'>,* importo la fuente desde Google *fonts*, línea 11, *</head>,* cierro la etiqueta cabecera, esta sección me permite importar todos los archivos que requiero para hacer correr la aplicación.

Línea 13, *<body>,* con esta etiqueta le establezco el inicio del documento, en la línea 14 y 15, se establecen dos etiquetas *div, Algoritmo de Levenshtein y Una aplicación sencilla basada en la Web </div*>, la etiqueta * * define una división o una sección de un documento *HTML,* la etiqueta * * se utiliza para agrupar elementos de bloques y asignarles darles formato con *CSS.*

Por esta razón ambas capas tienen especificado un *id="cabeza"* y *id="cabeza2",* *este* id, establece ciertas propiedades desde archivo *style.css* (línea 117 y 130), como tipo de fuente, tamaño de la fuente, alineación del texto centrado, color de texto, color de fondo, ancho de la capa *div (width),* altura de la capa igual a *45px (height),* tipo de alineación vertical con valor de *top* y por último y no menos importante el *padding-top* igual a 5px, la propiedad padding-top establece el relleno superior (espacio) de un elemento.

Línea 19, *<nav>,* etiqueta que define semánticamente que el rol de esta sección es el de una barra de navegación. Como aclaración a lo anterior y a lo posterior, hay varias maneras de identificar o seleccionar un elemento dentro del documento *HTML*, por su:

- *TagName: (div,nav,input, etc.).*
- *Clase: es recomendable que las clases se utilicen cuando a varios elementos se le va a aplicar un estilo (css) o un comportamiento (js).*
- *Id: este, a diferencia de las clases, es recomendable que se use para identificar cuando es un único elemento.*

Línea 21, * Patrón a buscar: *, etiqueta informativa de la barra de navegación, que busca una clase definida en el archivo *style.css* (línea 110) dentro de la cual se establece un color, un tamaño para la fuente, el tipo de caja utilizada para un elemento *HTML* (propiedad display), un margen izquierdo (la propiedad *margin* establece todas las propiedades de los márgenes en una declaración), esta propiedad puede tener de uno a cuatro valores.

Línea 22, *<input id="findWord" type="text" placeholder="¿Qué palabra busca?" onkeyup="EncontrarOnKeyUp();" />,* este es el *textBox* a través del cual se obtendrá la pista o el patrón a buscar, tiene asignado un *ID*, ya que con *Javascript* es necesario asignarlo, para que por medio de este *ID* se puedan establecer

determinados comportamientos, el atributo *placeholder* define el texto auxiliar que aparecerá cuando no exista ningún valor escrito en él, este valor que se introduce (patrón o cadena) se envía por medio del *ID* a la archivo *JavaScript (funciones.js, línea 55).*

También tiene asociada una acción, en el evento *onKeyUp* (cuando se suelta la tecla), para realizar después con *JS* el algoritmo de búsqueda a través de la función *EncontrarOnKeyUp (archivo JS* línea 47), que en este caso está esperando que el usuario pulse la tecla *ENTER*, en vez de oprimir el botón *Encontrar*, que al final produce la misma acción del botón.

Línea 23, botón que tiene asociado en su evento *onClick* la función *Encontrar*, cuya tarea es accionar el mecanismo de búsqueda del patrón usando la pista contenida en el campo *findWord* (línea 53 del archivo *JS*), línea 24, botón que tiene asociado en el evento *onClick* la función *nuevo*, que limpia el campo *findWord,* y por último la línea 25 que cierra la etiqueta *nav (*barra de navegación).

En *HTML5*, la idea de las nuevas etiquetas semánticas como *nav, sectionarticle,* etc., es el de dejar el documento más legible y más fácil de entender, colocando explícitamente el rol que desempeñan ciertas áreas del documento, por lo demás cada una de las etiquetas que mencione antes son simples cajas como cualquier *div*, claro que estas ayudan a hacer mejor *SEO* y posicionarse mejor en los buscadores, los cuales buscan contenido organizado y de calidad.

Search Engine Optimization (SEO) es el proceso de modificar la visibilidad de un sitio web o una página web en los resultados de búsqueda "naturales" o sin pagar en un motor de búsqueda. *SEO* se pueden orientar a los diferentes tipos de búsqueda, incluyendo búsqueda de imágenes, búsqueda local, búsqueda de vídeo, búsqueda académica, la búsqueda de noticias y motores de búsqueda en general. Y por poner una pequeña analogía escribiré brevemente la función de las etiquetas, imaginémoslo como un periódico:

- *Header:* como el encabezado de un periódico, donde está su nombre, su precio y ciertas cosas que lo describen, en esa área se podría haber puesto: * Algoritmo de Levenshtein , Una aplicación sencilla basada en la Web *, porque estas cumplen dicha función de ser el descriptivo de la aplicación.

- *Section:* esta etiqueta define las secciones de un periódico, la sección de deportes, la sección policiaca etc.

- *Article:* y cada sección está formada por artículos, para lo cual está la etiqueta *article.*

- *Nav:* en un periódico no existe como tal, pero en un sitio es la barra donde están centralizados los controles o los enlaces hacia las diferentes partes del sitio.

- *Footer:* simplemente el pie de página donde van digamos que los créditos finales o información adicional.

Línea 27, *<section id="cajaTexto">*, esta caja es la más importante de la aplicación, pues es la encargada de proporcionar al usuario un lugar donde colocar cierta cadena de texto y en base a ella obtener los resultados de búsqueda, está formado por dos elementos.

El primero es el *textarea* (línea 28) con el identificador *my_textarea*, que es un campo de texto amplio en el cual se puede colocar cualquier cadena y jugar con ella editándola etc., pero este elemento *HTML* no permite señalizar de ninguna manera un texto, por lo cual solamente cumple el rol de recibir una cadena de entrada para posteriormente examinarla, pero no se pueden mostrar los resultados obtenidos del análisis y procesamiento del algoritmo.

El segundo, para erradicar la carencia del *textArea* se establece un contenedor *(article)* de *id procesado*, que inicialmente aparece vacío, puesto que no hay ningún texto de salida que colocar. Cuando el mecanismo de búsqueda termina su labor, toda la cadena evaluada se coloca aquí dentro (línea 29 y 31). La etiqueta <*article*> especifica el contenido independiente, autónomo, y por último la línea 32, cierra la sección de con la etiqueta *</section>.*

Para mostrar ciertos detalles sobre los resultados de búsqueda, se creó este panel, que está contenido en un *section* separado, porque este contenido ya es adicional al *section* anterior, línea 34 *<section id="detalles">,* a etiqueta <*section*> define las secciones de un documento, como capítulos, encabezados, pies de página o cualquier otra sección del documento.

En la línea 35 se define nuevamente una etiqueta *article*, en la línea 36 de define una etiqueta de encabezado *<h2>PATRÓN A ENCONTRAR</h2>,* en la línea 37 se definen una caja: * - ,* dentro de esta caja aparecerá posteriormente el patrón colocado en el campo de texto *findWord,* a partir de una función de *JS*, línea 109 *$("#patron").text("" + pista + "");*, y cerramos el *</article>* en la línea 38, para que esta caja tenga ciertas características específicas se le asigna una clase llamada *detalle (class="detalle")* definida en la línea 164 del archivo *css.* Analizando todo este conjunto de líneas y como recordatorio, en la línea 34 se define una sección llamada detalles, dentro de esa sección existen varios artículos y cada uno de esos artículos tiene una clase definida que les proporciona sus determinadas características.

Esto se define en el archivo *css,* línea 143 *#detalles{ , línea #detalles article{,* la línea 164 *#detalles article{,* y por último la línea 164 *#detalles article .detalle{.*

En la línea 39, ocurre exactamente lo mismo que en la línea 35, se define un *article* que se encuentra dentro de la sección *detalles,* se establece una etiqueta de encabezado *<h2>OCURRENCIAS</h2> (línea 40),* y en la línea 41 se establece una nueva caja llamada *“rasgo”,* y dentro de esa caja se define una caja con una

clase definida llamada *“detalle”* y un *id llamado “cantidad”,* la etiqueta <*div*> se utiliza para agrupar elementos de bloques y así darles formato con *CSS (razón por la cual se encuentra una caja dentro de la otra),* para que posteriormente dentro de esta caja aparezca la cantidad de coincidencias encontradas en el texto proporcionado, a partir de una función de *JS (línea 111 del JS),* y por ultimo cerramos en la línea 42 cerramos la etiqueta *article.*

De la línea 44 a la línea 47, sucede exactamente lo mismo que en la línea 39 a la 42, con la diferencia de que ahora en la caja con la clase detalle tiene un *id llamado aproximación,* dentro de esta caja aparecerá posteriormente si se encontró el patrón o la coincidencia más cercana, a partir de una función de *JS*, esto ocurre en la línea 101 o 103 dependiendo si se encuentra el patrón exacto o no, respectivamente.

En la línea 48, se establece la *etiqueta* <*footer*>, que define un pie de página de un documento o sección. Un elemento <*footer*> debe contener información acerca de su elemento contenedor, un pie de página suele contener el autor del documento, información de derechos de autor, enlaces a términos de uso, información de contacto, etc., en la línea 49, se define una etiqueta <*p*> define un párrafo, “ Maestría en Ciencias de la Computación”, en la línea 50 se vuelve a definir otro párrafo con la etiqueta <*p*>, “Unidad Académica de Ciencias de la Tecnología y la Información.

En la línea 51, se define un nuevo párrafo pero centrado, a través de la etiqueta <*center*> *y* <*p*>*UAGro*</*p*></*center*>, al mismo tiempo escribo una cadena: Irving J. S. Linares, cierro la etiqueta </*footer*> en la línea 53, la etiqueta </*body*> y para finalizar cierro la etiqueta <*html*>.

3.5 Conclusiones y Trabajos futuros

La distancia de Levenshtein es de fundamental importancia en diversos campos tales como la biología computacional y la búsqueda o procesamiento de texto, y en consecuencia, todos los problemas que de alguna manera involucran la distancia de edición están siendo ampliamente estudiados hoy en día.

El algoritmo de edición de distancia normalmente se ejecuta en secuencias de miles de millones de pares de bases. Unos pocos de estos pares (pero, aun así, millones) pueden clasificarse como polimorfismos de nucleótido simple (*SNP*), que son variaciones en las que las bases nucleicas situadas en determinada ubicación del genoma difieren en algunos de los individuos dentro de una especie.

La distancia de edición da una indicación de que tan cerca o parecidas dos cadenas pueden ser, medidas similares se utilizan para calcular una distancia entre secuencias de *ADN (cadenas sobre {A, C, G, T}*, o secuencias de proteínas (más de un alfabeto de 20 aminoácidos), para diversos fines, por ejemplo: para encontrar los genes o proteínas que pueden haber compartido funciones o propiedades o para inferir las relaciones familiares y los árboles evolutivos de diferentes organismos.

Por otro lado algoritmos de reconocimiento de voz similares a los del problema de edición distancia se utilizan en algunos sistemas de reconocimiento de voz, para encontrar una coincidencia cercana entre una nueva expresión y en una biblioteca de expresiones de anuncios, etc.

En este trabajo de investigación, se muestra una estandarización tanto en conceptos de métricas como en las nomenclaturas relacionadas, también muestra diferentes algoritmos pretendiendo dar una revisión de los problemas encontrados en la búsqueda de patrones utilizando diferentes distancias.

Siguiendo esta revisión y estado del arte se encuentra una necesidad de encontrar una generalización de distancias que permita ser más flexible en la búsqueda exacta, ya que la gran mayoría de los nuevos algoritmos diseñados utilizan como base extractos y técnicas del algoritmo analizado previamente.

Actualmente grandes volúmenes de información estructurada y no estructurada se derivan de la Web, toda esta información se encuentra totalmente imprecisa, por lo que la integración de los diferentes lenguajes, *frameworks* y tecnologías de programación, como son *HTML5, JQuery y CSS*, hacen que este algoritmo sea una poderosa herramienta para el análisis de patrones sobre la web, ya que a través del *JQuery* es posible controlar todos los eventos que ocurren en la ejecución del algoritmo.

La métrica de Levenshtein es una herramienta que permite conocer cuáles fueron los cambien que tuvo que sufrir una cadena para convertirse en otra. Comparar secuencias extremadamente grandes como las secuencias del genoma humano requiere tratamientos especiales a los algoritmos utilizados. Esto ocurre por el crecimiento cuadrático del tiempo de ejecución asintótico del algoritmo para obtener la distancia de Levenshtein.

El problema que representan las secuencias sumamente grandes, implica un problema de espacio (al guardarlas en la memoria de una computadora, o al escribirlas en un papel), de modo que se puede diseñar modificaciones a esta métrica, con el fin de para determinar la distancias parciales de una subsecuencia con otra, ambas de longitud manejable.

Como trabajo futuro, se plantea realizar una guía metodológica de naturaleza gráfica que apoye la selección de las técnicas más adecuadas para casos particulares de la vida real, y por supuesto por otro lado se concibe implementar en el algoritmo, algunas mejoras y técnicas de emparejamiento de patrones que permitan un mejor comportamiento en otras situaciones como: abreviaturas, palabras desordenadas, palabras en mayúsculas y minúsculas, etc.

Bibliografía

1. Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences, by Gonzalo Navarro and Mathieu Raffinot Cambridge University Press © 2002 (221 pages).

2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms Second Edition. The MIT Press Cambridge, Massachusetts London, England. 1990.

3. M. H. Alsuwaiyel. Algorithms Design Techniques and Analysis. Publishing House of Electronics Industry. Beijing. 2003.

4. Algorithms FOURTH EDITION, Robert Sedgewick and Kevin Wayne. Princeton University. First printing, March 2011.

5. Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. David Sankoff and Joseph Kruskal. 2000.

Artículos de publicación científica

[JONES04] Neil C. Jones, Pavel A. Pevzner, An Introduction to Bioinformatics Algorithms, A Bradford Book The MIT Press, Cambridge, Massachusetts London, England.2004

[NAV04] Gonzalo Navarro. A guide tour to approximate String Matching. Departamento de Ciencias de la Computación. Universidad de Chile. 1994

[SINGAR12] Nimisha Singla, Deepak Garg. International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-I, Issue-6, January 2012.

[BAE-NAV12] Baeza-Yates and Navarro approximate string matching for spam filtering. Innovative Computing Technology (INTECH), 2012 Second International Conference on, 18-20 Sept. 2012, Pages: 16-20.

[HEI04] Heikki Hyyrrö. Bit-Parallel Approximate String Matching Algorithms with Transposition. Journal of Discrete Algorithms 3 (2005) 215–229, Universidad de Tampere Finlandia.

[NEU09] Shoshana Neuburger. Pattern Matching Algorithms: An Overview. Department of Computer Science The Graduate Center, CUNY. September 15, 2009.

[NEWU70] NEEDLEMAN, S. AND WUNSCH, C. 1970. A general method applicable to the search for similarities in the amino acid sequences of two proteins. J. Mol. Biol. 48, 444–453

[HU00] Huron, David. “Perceptual and cognitive applications in music information retrieval.” International Symposium on Music Information Retrieval, October 23-25, 2000.

[HAM1950] R. W. Hamming. Error detecting and error correcting codes. The Bell System. Technical Journal, 29(2):147-160, 1950.

[HYFRENAV04] Heikki Hyyr, Kimmo Fredriksson3, and Gonzalo Navarro, String Matching Increased Bit-Parallelism for Approximate String Matching, 2004.

[Lev1996] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady, 10(8):707{710, 1966. Original in Russian in Doklady Akademii Nauk SSSR, 163(4):845{848, 1965.

[CWN2007] Inexact Pattern Matching Algorithms via Automata1, Chung W. NG, March 19, 2007.

Enlaces de Internet

[INTEL 2010] Whitepaper

- http://download.intel.com/embedded/networksecurity/324029.pdf

Chan S. Wikipedia, the free encyclopedia, June 2004

- http://en.wikipedia.org/wiki/Dynamic_programming

Programación Dinámica.

- http://es.wikipedia.org/wiki/Programaci%C3%B3n_din%C3%A1mica

Algoritmo Knuth-Morris-Pratt (KMP)

- http://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/BusqTexto/#2es/cc30a/BusqTexto/#2