UNIVERSIDAD MAYOR DE SAN ANDRÉS

FACULTAD DE CIENCIAS PURAS Y NATURALES

CARRERA DE INFORMATICA

TESIS DE GRADO

"COMPARACIÓN DE ALGORITMOS DE EMPAREJAMIENTO APROXIMADO DE CADENAS SOBRE AUTÓMATAS DIFUSOS"

PARA OPTAR AL TÍTULO DE LICENCIATURA EN INFORMATICA
MENCIÓN: CIENCIAS DE LA COMPUTACIÓN

POSTULANTE: PABLO VICENTE HELGUERO VARGAS
TUTOR METODOLOGICO: Mg. Sc. ROSA FLORES MORALES
ASESOR: Mg. Sc. MOISES MARTIN SILVA CHOQUE

LA PAZ – BOLIVIA

2018

UNIVERSIDAD MAYOR DE SAN ANDRÉS
FACULTAD DE CIENCIAS PURAS Y NATURALES
CARRERA DE INFORMÁTICA

LA CARRERA DE INFORMÁTICA DE LA FACULTAD DE CIENCIAS PURAS Y NATURALES PERTENECIENTE A LA UNIVERSIDAD MAYOR DE SAN ANDRÉS AUTORIZA EL USO DE LA INFORMACIÓN CONTENIDA EN ESTE DOCUMENTO SI LOS PROPÓSITOS SON ESTRICTAMENTE ACADÉMICOS.

LICENCIA DE USO

El usuario está autorizado a:

a) visualizar el documento mediante el uso de un ordenador o dispositivo móvil.
b) copiar, almacenar o imprimir si ha de ser de uso exclusivamente personal y privado.
c) copiar textualmente parte(s) de su contenido mencionando la fuente y/o haciendo la referencia correspondiente respetando normas de redacción e investigación.

El usuario no puede publicar, distribuir o realizar emisión o exhibición alguna de este material, sin la autorización correspondiente.

AGRADECIMIENTO

A mi tutora Mg. Sc. Rosa Flores Morales por la paciencia, conocimientos y consejos en el desarrollo de la presente tesis. De igual forma agradezco a mi asesor Mg. Sc. Moises Martin Silva Choque por los consejos, sugerencias y ser un gran asesor a la hora de guiarme.

RESUMEN

El creciente interés en la lógica difusa y su incorporación en diversas áreas de la investigación científica han hecho que esta se incorpore en áreas como la inteligencia artificial, bases de datos, control de sistemas. Uno de los no tan nuevos campos de aplicación de la lógica difusa es en la teoría de lenguajes y autómatas donde se incorpora la imprecisión de la lógica difusa en las transiciones de estados, en los símbolos que maneja el autómata o en los estados finales por nombrar algunas incorporaciones de la lógica difusa en la teoría de autómatas.

Un campo donde la teoría de autómatas difusos está recientemente siendo incorporado es en el problema del emparejamiento aproximado de cadenas, es aquí donde se propone que mediante el uso de autómatas difusos específicos como el parcial o intuicionista se maneje la imprecisión necesaria en este emparejamiento de caracteres y ya varios métodos han sido propuestos.

Se propone un estudio de estos autómatas difusos adaptados para el problema del emparejamiento aproximado de cadenas sobre retículas residuales completas que son el sustituto formal del intervalo $[0,1]$ antes manejado en lógica difusa, junto a una comparación de estos autómatas difusos con los algoritmos tradicionales o clásicos diseñados para dicho problema.

Palabras clave: Autómatas difusos, Emparejamiento aproximado de cadenas, retículas residuales completas.

ABSTRACT

The fuzzy logic is increasing in importance in many areas of the computer science like the artificial intelligence, data bases and control systems, and one of the newest fields where the fuzzy logic is being applied is in the automata and language theory where is incorporated in the transition functions, in the symbols of the automata with the fuzzy symbols and in functions of the final states of the automata.

Nowadays, the fuzzy automata are being applied a new area of the computer science, the approximate string matching, where the authors proposed use the fuzziness of the fuzzy logic to manage the imprecision needed while comparing strings and many methods has been proposed to manage not just the edit distance of the approximate string matching is also proposed to use the operations of the fuzzy logic to evaluate the similarity between strings.

A study and analysis of the fuzzy automata for approximate string matching over complete residuated lattices is proposed, the complete residuated lattices are the algebraic structure where the fuzzy logics is built, and also a comparison between fuzzy automata with the classical algorithms for approximate string matching.

Keywords: Fuzzy automata, approximate string matching, complete residuated lattices.

. Índice de Tesis

Índice de Tablas

Índice de Figuras

1. PROBLEMA DE INVESTIGACIÓN

1.1. INTRODUCCIÓN

Este trabajo se centra en el problema de encontrar cadenas que aproximadamente coincidan con un patrón, lo que es conocido en ciencias de la computación como emparejamiento de patrones aproximados[1]. El problema, en su forma más general, es encontrar un segmento en un texto donde un patrón dado ocurra, permitiendo un cierto número de errores en el emparejamiento. Cada aplicación usa un modelo diferente de manejo de error, el cual define cuán diferente son dos cadenas. La idea para esta 'distancia' entre cadenas es que sea más pequeña cuando una de las cadenas probablemente sea una variante errónea de la otra bajo el modelo de error que se está usando.

El emparejamiento aproximado de cadenas es aplicado en el mundo real en muchas áreas de las ciencias de la computación como en búsqueda en textos, bioinformática, reconocimiento de patrones, procesamiento de señales entre otras. Uno de los casos más estudiados en búsqueda de patrones con errores, llamado modelo de emparejamiento aproximado de cadenas, es el que emplea la distancia de edición, la cual permite borrar, insertar y reemplazar caracteres en la cadena observada en relación con la cadena patrón.

Muchas de las publicaciones acerca de métodos de clasificación de cadenas con errores están disponibles en la literatura, muchas de ellas están basadas en métodos estadísticos, aunque también existen otros con métodos difusos. Un sistema difuso puede

[1] Del original en inglés "*approximate string matching*".

ser más intuitivo para la evaluación de semejanza entre dos cadenas, ya que requiere menos normalizaciones y el manejo de la incertidumbre que poseen dichos sistemas es una cualidad importante.

Por su parte, la teoría de conjuntos difusos tiene su origen en la necesidad de formalizar la imprecisión y presentarla usando modelos matemáticos. Lofti Zadeh comienza su estudio en el año 1965, quien provee una manera de distinguir la pertenencia de un elemento a una clase particular, también introduce la noción de relaciones difusas, en particular las de equivalencia difusa y cuasi ordenes difusos. El estudio parte de la teoría clásica de conjuntos, añadiendo la función de pertenencia al conjunto, la misma que es definida como un número real entre 0 y 1, introduciendo así el concepto de conjunto difuso asociado a un determinado valor lingüístico.

Debido al hecho que las relaciones difusas permiten expresar rastros de conectividad apenas perceptibles entre elementos, son extensamente aplicados en el modelado de varios conceptos en las llamadas ciencias suaves como la psicología, lingüística y muchas otras áreas de la ciencia. Dada estas características y especialmente el valor que tiene el tratamiento de imprecisión en el emparejamiento de las cadenas, uno de los nombres con el cual también es conocido el emparejamiento aproximado de cadenas, el estudio de la lógica difusa dentro del tema tratado por la presente tesis, podrían llevar a mejores resultados en la búsqueda de patrones aproximados frente a los algoritmos clásicos de búsqueda.

En recientes trabajos, varias estructuras generales de valores de verdad son usadas en lugar del intervalo [0,1]. Dichas estructuras algebraicas son construidas con base en

ciertas propiedades de distintos tipos de lógicas, por ejemplo, se tiene la *Gödel* álgebra, *Wajsberg* álgebra, *Product* álgebra, *BL* álgebra, *Heyting* álgebra, *Complete Orthomodular Lattices* por mencionar algunas de ellas. En la presente tesis se contempla el trabajo con base en retículas residuales completas, las cuales incluyen a las cinco primeras algebras mencionadas como casos especiales.

Desde el principio de la teoría de conjuntos difusos, autómatas y lenguajes difusos son estudiados como un medio para cerrar la brecha entre la precisión de los lenguajes de computadora y la incertidumbre e imprecisión, que son frecuentemente encontrados en el estudio de los lenguajes naturales. El estudio de autómatas y lenguajes difusos fue comenzado por Santos y otros investigadores en el año1960, y durante las posteriores décadas, los autómatas y lenguajes difusos han ganado un amplio campo de aplicación incluyendo análisis léxico, descripción de lenguajes de programación y naturales, sistemas de aprendizaje, sistemas de control, monitoreo clínico, reconocimiento de patrones, corrección de errores, redes neuronales, representación del conocimiento, bases de datos, sistemas con eventos discretos entre otros.

En el presente trabajo de tesis se propone realizar el modelado e implementación de distintos tipos de autómatas difusos sobre $\mathcal{L}$ donde $\mathcal{L}$ es un álgebra, precisamente una retícula residual completa $\mathcal{L} = (L, \wedge, \vee, \otimes, \rightarrow, 0, 1)$. Sobre estos autómatas se modelarán distintos tipos de algoritmos de emparejamiento aproximado de cadenas. Con la finalidad de evaluar su desempeño frente a los algoritmos aproximados de cadenas, en términos de tiempo de ejecución, espacio de memoria ocupado y ocurrencias encontradas.

1.2. ANTECEDENTES

El problema del emparejamiento de cadenas con k errores consiste en encontrar todas las ocurrencias de un patrón de longitud m en un texto de longitud n de tal forma que a lo más en k posiciones el texto y el patrón tengan diferentes símbolos. Se asumirá que $0 < k < m$ y $m \leq n$. El caso donde $k = 0$ es bien conocido como emparejamiento exacto de cadenas, y si $k = m$ la solución es trivial.

(Landau & Vishkin, 1986) proporcionan el primer algoritmo eficiente para resolver este problema particular. Su algoritmo tiene una complejidad $O(kn + km \log m)$ y $O\big(k(n+m)\big)$ en espacio. A pesar de ser rápido, el espacio requerido es inaceptable para la mayoría de los casos prácticos. (Galil & Giancarlo, 1986) logran una mejora de este algoritmo usando un algoritmo de antecesor común estático sobre un árbol de sufijos. Así, la constante del termino $O(kn)$ ya no es tan grande, y el algoritmo final es algo más lento en la práctica que el algoritmo de Landau y Vishkin.

Como fue mencionado antes, desde el principio de la teoría de los conjuntos difusos, autómatas y lenguajes difusos fueron estudiados como un medio para cerrar la brecha entre la precisión de los lenguajes de computadora y la vaguedad e imprecisión, que son frecuentemente encontrados en el estudio de los lenguajes naturales. El estudio de los autómatas difusos fue iniciado en 1960 por (Santos, 1968), (Wee, 1969) y (Lee & Zadeh, 1969). La idea de estudiar autómatas difusos con valores de pertenencia en un conjunto estructurado abstracto inicia con (Wechler, 1978), y en los últimos años el estudio se ha centrado en los autómatas difusos con valores de pertenencia en retícula residual completa, monoides reticulares ordenados, y otros tipos de retículas, desde finales de los

años 1960 hasta principios de los años 2000 también son considerados los lenguajes y autómatas difusos con valores de pertenencia en estructuras de *Gödel* (Dubonis & Prade, 1980), (Gupta, Sardis, & Gaines, 1977).

Los autómatas difusos tomando valores de pertenencia en retículas residuales completas fueron estudiados por primera vez por (Qiu D. , 2001), donde algunos conceptos básicos fueron discutidos, y después Qiu y su equipo llevaron a cabo extensa investigación en estos autómatas difusos. Desde un punto de vista diferente los autómatas difusos fueron estudiados por (Ciric & Ignjatovic, 2012). Autómatas difusos con valores de pertenencia en *lattice-ordered monoids* fueron investigadas por (Li & Li, 2007), y autómatas que generalizan los autómatas difusos sobre cualquier retícula, además de autómatas con pesos sobre semi anillos, fueron estudiados recientemente por (Ciric, Droste, Ignjatovic, & Vogler, 2010), (Droste, Stüber, & Vogler, 2010).

Los autómatas difusos se tomaron como la generalización natural de los autómatas finitos no deterministicos y los autómatas finitos deterministas, estos autómatas difusos tienen un solo estado inicial y una función de transición determinística, con un conjunto difuso de estados terminales. Estos autómatas fueron introducidos por (Belohlavek, 2002) y en el trabajo de (Ciric, Droste, Ignjatovic, & Vogler, 2010) estos autómatas pasaron a ser llamados autómatas difusos deterministicos *crisp*.

El problema de la conversión de los autómatas difusos a su lenguaje equivalente es estudiado por (Belohlavek, 2002), en el contexto de un autómata finito difuso sobre una retícula distribuida completa, y (Li & Pedrycz, 2005) en el contexto de un autómata finito difuso sobre un monoide reticular ordenado, lo cual fue llamado determinización de

autómatas difusos. Algoritmos de determinización que fueron introducidos por los anteriores autores mencionados generalizaban la construcción del conjunto. Otro algoritmo, propuesto por (Ignjatovic, Ciric, & Bogdanovic, Determinization of fuzzy autmomata with memebership values in complete residuated lattices., 2008), también generaliza la construcción del conjunto y produce un pequeño autómata determinístico difuso conciso al igual que (Belohlavek, 2002), (Li & Pedrycz, 2005). Estos autómatas determinísticos difusos concisos también podían ser construidos por la congruencia derecha de Nerode partiendo de un autómata difuso y fueron llamados en (Ignjatovic J. , Ciric, Bogdanovic, & Petkovic, 2010) autómatas de Nerode.

El autómata de Nerode que fue construido en (Ignjatovic, Ciric, & Bogdanovic, 2008) a partir de un autómata finito difuso sobre una retícula residual completa, y se notó que una construcción idéntica puede ser hecha en un contexto más general, para autómatas finitos difusos sobre monoides reticulares ordenadas, e incluso para autómatas con pesos sobre semi anillos. El algoritmo propuesto por (Jancic, Ignjatovic, & Ciric, 2011) produce un autómata determinístico concisos o un autómata con pesos que son aún más pequeños que los autómatas de Nerode.

Otro modelo de autómata difuso son los llamados autómatas con estados difusos, vienen siendo discutidos en una serie de artículos lidiando con sistemas difusos con eventos discretos (Liu & Qiu, 2008), (Qiu & Liu, 2009), (Xing, Zhang, & Huang, 2012). El conjunto de estados de estos autómatas es la colección de todos los subconjuntos difusos de un conjunto dado A, el estado inicial es un subconjunto de A y el conjunto de estados finales es una colección de subconjuntos difusos de A. Las transiciones son

definidas como composiciones de estados difusos con relaciones difusas pre especificadas que son llamados eventos difusos. Alternativamente, los conjuntos de eventos difusos pueden entenderse como una familia de relaciones de transiciones difusas indexadas por un conjunto ordinario de eventos o entradas.

En la presente tesis se usará una definición algo diferente de un autómata con estados difuso. Para dar la definición de estos autómatas que incluyen algunos importantes autómatas, se asume que los conjuntos de estados difusos son cualquier colección - posiblemente finita - de subconjuntos difusos de A. El estado inicial es un subconjunto de A, y los estados terminales son modelados por un subconjunto difuso del conjunto de estados difusos. Finalmente, las transiciones son definidas como una función que mapea el producto cartesiano del conjunto de estados difusos y el alfabeto de entrada hacia el conjunto de estados difusos.

1.3. PROBLEMÁTICA

La incorporación de algoritmos de emparejamiento aproximado de cadenas sobre autómatas difusos se ha realizado en algunos trabajos (Ravi, Choubey, & Tripati, 2013), (Chatterjee, Henzinger, Ibsen-Jensen, & Otop, 2017), (Echabone, Garitagoitia, & González de Mendívil), (Andrejková, Almarimi, & Mahmoud, 2013) los cuales modelan la distancia de edición sobre autómatas difusos intuicionistas y autómatas difusos parciales respectivamente además de ciertas propiedades de los lenguajes difusos para el tratamiento de la proximidad entre cadenas.

Dichos trabajos se centran en autómatas y algoritmos particulares, por lo cual no existe una aplicación extensa de los autómatas difusos en el área del emparejamiento

aproximado de cadenas ni mucho menos una comparación del desempeño de dichos algoritmos sobre autómatas frente a los propios algoritmos de emparejamiento aproximado de cadenas.

a) Definición del problema

¿Los algoritmos de emparejamiento aproximado modelados en distintos tipos de autómatas difusos, tienen el mismo desempeño en distintos casos de prueba con diferentes patrones y textos frente a los algoritmos de emparejamiento aproximado de cadenas?

1.4. OBJETIVOS

a) Objetivo general

Estudiar y realizar una evaluación de distintos tipos de autómatas difusos modelando sobre ellos algoritmos de emparejamiento aproximado de cadenas, comparando el tiempo de ejecución, espacio en memoria y ocurrencias encontradas con los algoritmos de emparejamiento aproximado de cadenas.

b) Objetivos específicos

- Revisar el estado del arte referido a Autómatas difusos y algoritmos aproximados de cadenas.
- Programar algoritmos de emparejamiento aproximado de cadenas sobre autómatas difusos.
- Generar un dataset con diferentes patrones y textos, considerando distintos alfabetos para su construcción.
- Evaluar el desempeño de los programas con base en tiempos de ejecución, espacio en memoria y ocurrencias encontradas.

1.5. HIPÓTESIS

Los algoritmos de emparejamiento aproximado de cadenas modelados sobre autómatas difusos tienen un mejor desempeño, en términos de tiempo de ejecución, espacio en memoria y ocurrencias encontradas a comparación de los algoritmos de emparejamiento aproximado de cadenas.

1.6. JUSTIFICACIÓN

El creciente interés en la lógica difusa y su incorporación en muchas áreas de investigación y estudio hacen necesario la incorporación de un medio que sea capaz de adaptarse a las propiedades de la lógica difusa y que incorpore una aproximación sistemática al razonamiento. Los Autómatas Difusos combinan dichas características ya que estos son capaces de manejar espacios continuos y modelar la imprecisión que está presente en muchos sistemas.

Dicha característica es la causa de muchas investigaciones en el área, pero a pesar de ello existen ciertos temas que aún están siendo establecidos y requieren cierta revisión. En el caso del emparejamiento aproximado de cadenas existen ciertos trabajos relacionados al tema tratado (Ravi, Choubey, & Tripati, 2013), (Chatterjee, Henzinger, Ibsen-Jensen, & Otop, 2017), (Andrejková, Almarimi, & Mahmoud, 2013), (Echabone, Garitagoitia, & González de Mendívil) pero dichos trabajos están enfocados en autómatas o algoritmos particulares y no son comparados ni evaluados entre ellos.

Al emprender la investigación de los algoritmos de emparejamiento aproximado de cadenas sobre autómatas difusos y su evaluación, se profundiza el conocimiento aportando así a futuras investigaciones, ya que se está trabajando en un área en desarrollo.

Así también como bases prácticas y teóricas para la aplicación de dichos algoritmos en áreas como el análisis léxico, representación del conocimiento, reconocimiento de patrones, corrección de errores, bases de datos, entre otros.

Por lo tanto, el desarrollo del trabajo se justifica, no sólo por los posibles resultados en el manejo o reconocimiento de cadenas y su aplicación en distintas áreas mencionadas, sino también desde el punto de vista del tratamiento de temas nuevos en la investigación científica de la región con la finalidad de aportar al área de la informática teórica.

1.7. ALCANCES Y LÍMITES

Los límites del trabajo propuesto se describen en las siguientes líneas:

- No se contemplan otras funciones de distancia que no encajen con el modelo de sustitución de subcadena. Esto ya que dichos modelos son muy diferentes a nuestro enfoque. Algunos de estos son aquellos que contemplan inversiones de cadenas (Kececioglu & Sankoff, 1995), permutaciones de cadenas (Tichy, 1984), intercambios (Amir, Aumann, Landau, Lewenstein, & Lewenstein, 1997) y aquellos basados en subcadenas comunes (Ukkonen, 1992).
- Se considera emparejamiento de patrones sobre secuencias de símbolos, y como máximo, generalizar el patrón a una expresión regular. Extensiones como búsqueda aproximada en textos multidimensionales (Navarro & Baeza-Yates, 1999), en grafos (Amir & Lewenstein, Pattern matching in hypertext, 1997), búsqueda aproximada de multi-patrones (Muth & Manber, 1996) no son consideradas.

- Se deja de lado algoritmos no estándares, como aquellos de aproximación, probabilísticos o algoritmos paralelos (Tarhio & Ukkonen, 1988), (Karloff, 1993)
- Así como también algoritmos de paralelismo de bit ya que estos siguen un enfoque dependiente de la arquitectura del sistema y manejo de palabras y el trabajo esta enfoca más en la imprecisión de las cadenas tratadas.

2. EMPAREJAMIENTO APROXIMADO DE CADENAS

En este capítulo se plantea la teoría relacionada con el emparejamiento aproximado de cadenas, la distancia de edición, el primer algoritmo formalizado en programación dinámica y los teoremas principales para el desarrollo de los posteriores algoritmos. Definiciones e información son recopiladas principalmente de (Navarro, 2001), (Jokinen, Tarhio, & Ukkonen, 1988), (Navarro & Raffinot, Flexible Pattern Matching in Strings, 2002), (Wu, Manber, & Myers, 1992), (Navarro G. , A Partial Deterministic Automaton for Approximate String Matching, 1997).

2.1. DEFINICIÓN

Sea Σ un alfabeto no vacío y finito, para cualquier cadena $s \in \Sigma^*$, se denota la longitud de cadena como $|s|$, s_i como el i – ésimo carácter de s, así mismo, $s_{i...j}$ denota la subcadena de s comenzando en el i – ésimo carácter hasta el j – ésimo, por último se tiene a λ como la cadena vacía para el resto del texto.

Sea Σ un alfabeto finito, $T \in \Sigma^*$ un texto de longitud $|T| = n$, y $P \in \Sigma^*$ un patrón de longitud $|P| = m$, $k \in \mathbb{R}$ representa el máximo número de errores permitidos. Se define $ed: \Sigma^* \times \Sigma^* \to \mathbb{R}$ como una función de distancia.

Sea $L(P)$ el lenguaje generado por el patrón P, es decir, $L(P)$ consiste en todas las cadenas que coinciden con P. Se define la $vecindad - k$ de P, $L_k(P)$, como:

$$L_k(P) = \{x | y, y \in L(P), ed(x, y) \leq k\}$$

El problema es presentado como: dados T, P, k y ed, encontrar las subcadenas en el texto que pertenezcan a $L_k(P)$, o también más comúnmente como retornar el conjunto de todas las posiciones j del texto tal que exista un i que cumpla $ed(P, T_{i...j}) \leq k$.

La distancia de edición, $ed(x, y)$, entre dos cadenas x e y es el mínimo costo de una secuencia de operaciones que transforman x en y, el costo de una secuencia de operaciones es la suma de los costos de cada una de las operaciones individuales. Las operaciones son un conjunto finito de reglas de la forma $I_{z_i} = t_k, D_{w_j} = t_r, R_{z_i,w_j} = t_s$, donde z_i y w_j son caracteres de las cadenas z, w respectivamente y t_k, t_r, t_s son números reales no negativos, una vez que alguna secuencia de operaciones convierte z en w, no es posible realizar más operaciones sobre w.

Si para cada operación de la forma $I_{z_i}, D_{w_j}, R_{z_i,w_j}$ existe la operación respectiva $D_{z_i}, I_{w_j}, R_{w_j,z_i}$ con el mismo costo, entonces la distancia de edición es simétrica, $ed(z, w) = ed(w, z)$. Para cualquier cadena x e y, $ed(x, x) = 0$, y se tiene que $ed(x, z) \leq ed(x, y) + ed(y, z)$.

El conjunto de posibles operaciones está restringido a:

- Inserción: I_α, insertar el carácter α.
- Eliminación: D_α, eliminar el carácter a.
- Sustitución: $R_{\alpha,\beta}$, reemplazar el carácter a por b.

Se define las funciones de distancia más conocidas y usadas basadas en las operaciones de edición descritas:

Distancia de Levenshtein (Levenshtein, 1965). – El cual permite inserciones, eliminaciones y sustituciones. En la definición simplificada todas las operaciones tienen un costo de 1. Esta definida como el menor número de inserciones, eliminaciones y sustituciones para hacer dos cadenas iguales.

Distancia de Hamming (Sankoff & Kruskal, 1983). – Permite solamente sustituciones de costo uno en la versión simplificada.

Distancia de subsecuencia común máxima (Needleman & Wunsch, 1970) (Apostolico & Guerra, 1987). – Permite solamente inserciones y eliminaciones, todas con coste 1. El nombre hace referencia al hecho de que mide la longitud de la mayor secuencia de caracteres coincidentes entre las dos cadenas, respetando el orden de los caracteres.

Bajo estas distancias de edición, el valor $\eta = {}^{k}/_{m}$ es conocido como el nivel de error, este valor da la idea de la ratio de error permitido en el emparejamiento, es decir, la fracción del patrón que puede presentar errores.

La longitud del patrón puede ser tan corta como 5 letras o cientos de letras, el número de errores permitidos k debe satisfacer que $\eta = {}^{k}/_{m}$ es un valor bajo, en el rango $1/m$ y $1/3$. La longitud del texto puede ser tan corta como unos cientos de letras a gigabytes de archivos, y por último el tamaño del alfabeto no está restringido, pero podría ir desde 4 a alfabetos de 256 elementos.

2.2. ESTADÍSTICAS DEL PROBLEMA

Los resultados obtenidos en (Réigner & Szpankowski, 1997) se mantienen asumiendo que los caracteres del texto son generados independientemente con probabilidades fijas, es decir, un modelo Bernoulli, donde todos los caracteres ocurren con la misma probabilidad $1/|\Sigma|$. El problema de la distancia de edición media entre dos cadenas esta estrechamente relacionado con la subsecuencia común máxima, los resultados de (Chavátal & Sankoff, 1975) y (Deken, 1979) son aplicados a este caso. Se tiene que para dos cadenas cualquiera de longitud m, $m - lcs \leq ed \leq 2(m - lcs)$, donde lcs es la longitud de la subsecuencia común máxima.

Como fue probado en (Chavátal & Sankoff, 1975), la lcs promedio está entre $m/\sqrt{|\Sigma|}$ y $me/\sqrt{|\Sigma|}$ para un alfabeto grande, y así la distancia de edición media está entre $m\left(1 - e/\sqrt{|\Sigma|}\right)$ y $2m\left(1 - 1/\sqrt{|\Sigma|}\right)$ (Sankoff & Mainville, Common Subsequiences and Monotone Subsequences, 1983).

Delimitar la probabilidad de un emparejamiento con errores tiene más importancia que la distancia de edición media en este caso. Sea $f(m, k)$ la probabilidad de que un patrón aleatorio de longitud m coincida en una posición de un texto con hasta k errores bajo la distancia de edición. En (Baeza-Yates & Navarro, 1999), (Navarro & Baeza-Yates, 1999) los límites superiores e inferiores del máximo nivel de error η^* para el cual $f(m, k)$ es decreciente exponencial sobre m son encontrados.

a. Límite superior

El límite superior para η^* proviene de la demostración de que la probabilidad de coincidencia es $f(m,k) = O(\mu^m)$ para:

$$\mu = \left(\frac{1}{|\Sigma| \eta^{\frac{2\eta}{1-\eta}} (1-\eta)^2} \right)^{1-\eta} \leq \left(\frac{e^2}{|\Sigma|(1-\eta)^2} \right)^{1-\eta}$$

Donde se puede notar que $\mu = 1/|\Sigma|$ para $\eta = 0$ y μ tiende a 1 a medida que η crece.

Esta probabilidad de emparejamiento es exponencialmente decreciente sobre m tanto como $\mu < 1$, lo que es equivalente a:

$$\eta < 1 - \frac{e}{\sqrt{|\Sigma|}} - O(1/|\Sigma|) \leq 1 - \frac{e}{\sqrt{|\Sigma|}}$$

Por lo tanto, $\eta < 1 - e/\sqrt{|\Sigma|}$ es una condición en el nivel de error que asegura pocos errores, así el máximo nivel η^* satisface $\eta^* > 1 - e/\sqrt{|\Sigma|}$.

Dicha demostración es obtenida usando un modelo combinatorio, basado en la observación de que $m - k$ caracteres deber coincidir en el mismo orden en dos cadenas que coinciden con k errores, y todas las posibles alternativas de seleccionar los caracteres coincidentes de ambas cadenas son contabilizadas.

b. Límite inferior

Tomando un punto de vista optimista, considerando que solo se realizan sustituciones, la distancia de edición es simple de analizar, usando un modelo combinatorio se demuestra que la probabilidad de coincidencia es $f(m,k) = v^m m^{-1/2}$, donde:

$$\nu = \left(\frac{1}{(1-\eta)|\Sigma|}\right)^{1-\eta}$$

Por lo tanto, un límite superior para el máximo η^* bajo esta condición es $\eta^* \leq 1 - 1/|\Sigma|$, ya que caso contrario $f(m,k)$ no sería exponencialmente decreciente en m.

c. Distancia de edición media

En (Navarro G. , A Guided Tour to Approximate String Matching, 2001) se presenta un resultado en el cual se prueba que la distancia de edición es mayor a $m\left(1 - e/\sqrt{|\Sigma|}\right)$ que había sido obtenida en (Chavátal & Sankoff, 1975). Sea $p(m,k)$ la probabilidad de que la distancia de edición entre dos cadenas de longitud m sea a lo más k. $p(m,k) \leq f(m,k)$ ya que pueden coincidir cualquier sufijo de longitudes de $m-k$ a $m+k$, así, la distancia de edición media es:

$$\sum_{k=0}^{m} k * P(ed = k) = \sum_{k=0}^{m} P(ed > k) = \sum_{k=0}^{m} 1 - p(m,k) = m - \sum_{k=0}^{m} p(m,k)$$

Como $p(m,k)$ se incrementa con k, es mayor que $m - \big(Kp(m,K) + (m-K)\big) = K\big(1 - p(m,K)\big)$ para cualquier K. En particular, para $K/m < 1 - e/\sqrt{|\Sigma|}$ se tiene que $p(m,K) \leq f(m,K) = O(\mu^m)$ para $\mu < 1$. Por lo tanto, si se toma K tal que $K = m\left(1 - e/\sqrt{|\Sigma|}\right) - 1$ se tiene que la distancia de edición es por lo menos $m\left(1 - e/\sqrt{|\Sigma|}\right) + O(1)$ para cualquier alfabeto.

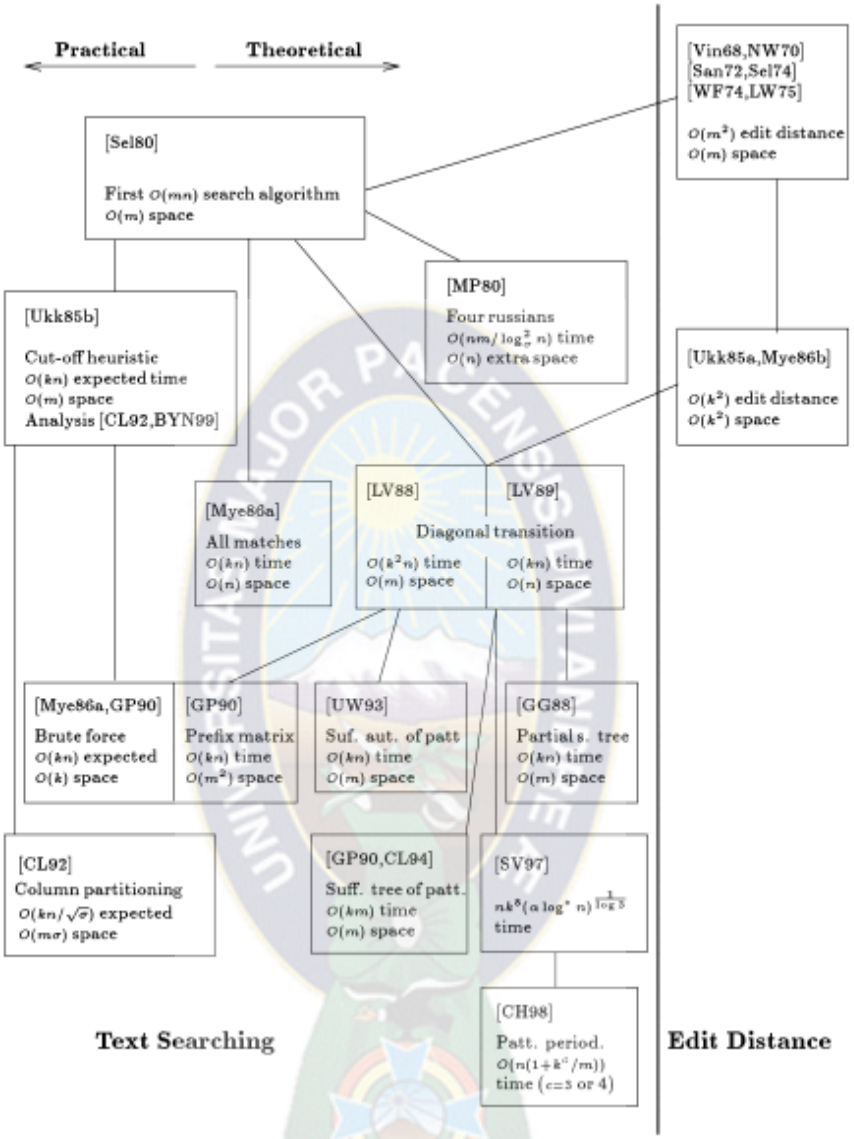

Figura 1 Taxonomía de los algoritmos basados en programación dinámica. Los algoritmos son representados por la inicial del autor y el año. Fuente: *(Navarro G. , 2001).*

2.3. ALGORITMOS BASADOS EN PROGRAMACIÓN DINÁMICA

La más antigua de las soluciones al problema del emparejamiento aproximado, muchas de las mejoras teóricas pertenecen a esta categoría, pero no son competitivas en la práctica. Los mejores resultados obtenidos son algoritmos con peor caso $O(kn)$ y $O\left(kn/\sqrt{|\Sigma|}\right)$ como caso medio. El primer algoritmo, el cual fue descubierto en distintas

áreas por distintos autores pero que al final es atribuido a (Sellers, 1980) por ser quien lo propuso como un algoritmo de búsqueda pertenece a esta categoría, dicho algoritmo es descrito y analizado más adelante. En la figura 1 se muestra la taxonomía de los algoritmos basados en programación dinámica.

2.4. ALGORITMOS BASADOS EN AUTÓMATAS

Un área altamente estudiada ya que proporciona el mejor peor caso en tiempo de ejecución, sin embargo, existe una dependencia exponencial en tiempo y espacio en m y k que limita su aplicación práctica en determinados casos. Una alternativa al considerar el problema es modelar la búsqueda mediante un autómata finito no determinista, dicho autómata fue propuesto por (Ukkonen, 1985) y utilizado en varios trabajos en su forma no determinística (Wu & Manber, 1992), (Baeza-Yates, Some new results on approximate string matching, 1991). En la figura 2 se muestra la taxonomía de los algoritmos basados en autómatas determinísticos.

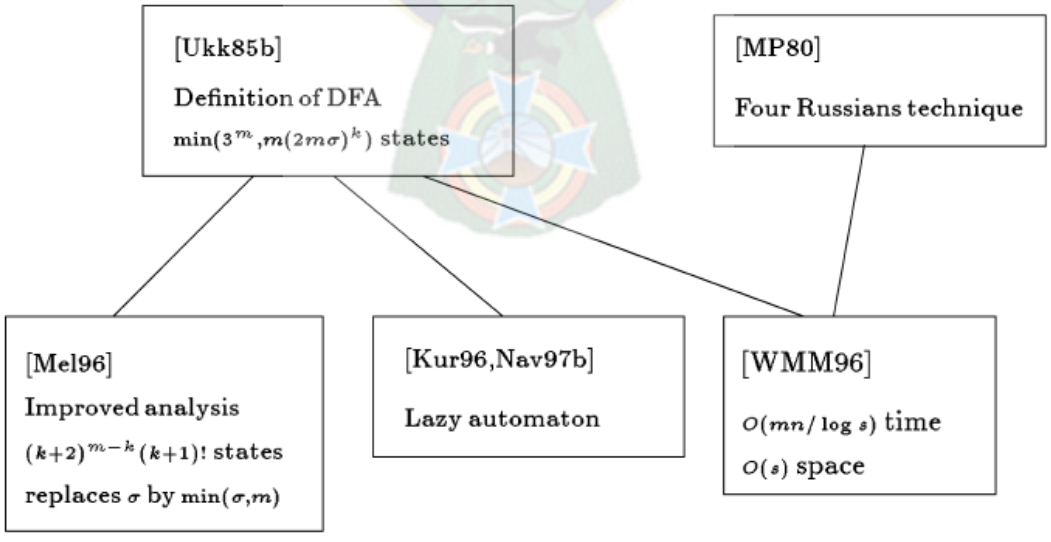

Figura 2 Taxonomía de los algoritmos basados en autómatas determinísticos. Los algoritmos son representados por la inicial del autor y el año. Fuente: *(Navarro G. , 2001)*.

Considere el AFN de la figura 3 construido para el patrón “autómata” con $k = 2$ como máximo valor de distancia de edición. Cada fila representa el número de errores encontrados, las transiciones horizontales representan el emparejamiento exitoso de algún carácter, las transiciones verticales la inserción de un carácter en el patrón, las transiciones diagonales la sustitución de un carácter y finalmente las transiciones diagonales por λ la eliminación de un carácter.

Simular el funcionamiento de dicho autómata en su forma no determinística representa un gran gasto de memoria, ya que cuando un estado está activo, todos los estados de la misma columna también lo estarán. Dicho autómata puede ser llevado a su equivalente determinístico en un tiempo $O(n)$ en el peor caso, pero el principal problema tratado es la construcción de un autómata finito determinístico.

2.5. ALGORITMOS BASADOS EN PARALELISMO DE BIT

Estos algoritmos están basados en explotar el paralelismo de las computadoras cuando trabajan con bits. La idea es paralelizar otro algoritmo usando bits, los resultados obtenidos son interesantes desde un punto de vista práctico, y son muy significativos

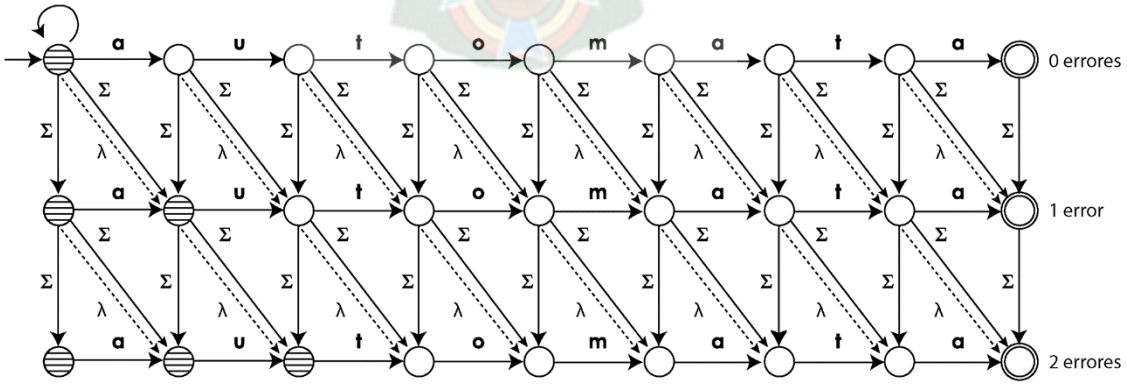

Figura 3 Autómata Finito no Determinístico para el patrón “autómata” con hasta dos errores.

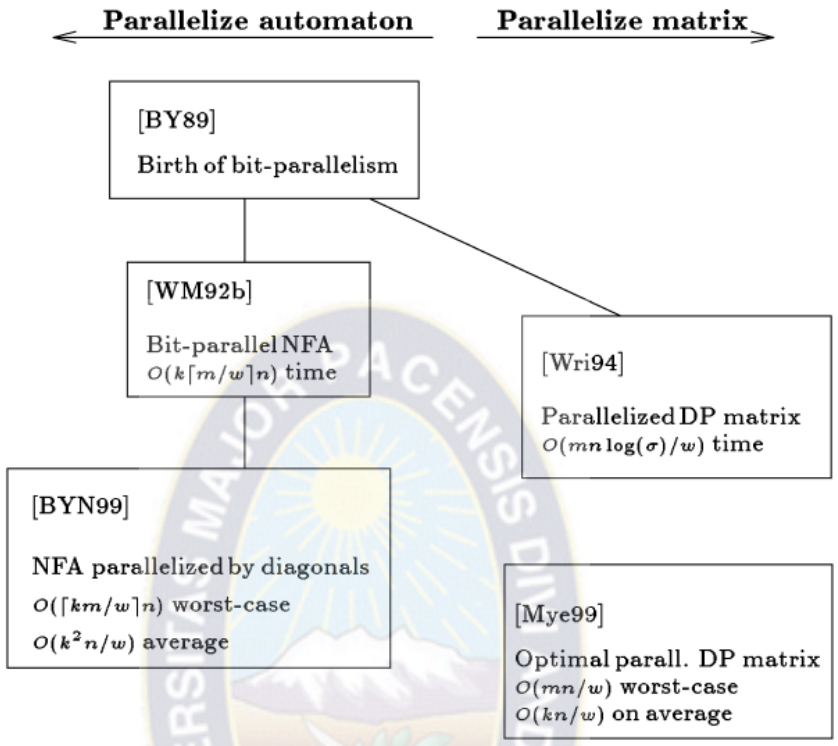

Figura 4 Taxonomía de los algoritmos basados en paralelismo de bit. Los algoritmos son representados por la inicial del autor y el año. Fuente: *(Navarro G. , 2001).*

cuando se trabaja con patrones cortos. Existen dos principales tendencias el paralelizar el autómata o paralelizar la matriz de programación dinámica. En la figura 4 se muestra la taxonomía de los algoritmos basados en paralelismo de bit.

2.6. ALGORITMO DE SELLERS

Sean x, y dos cadenas y se requiere calcular la distancia de edición $ed(x, y)$. Se construye la matriz $C_{|x+1|\times|y+1|}$, donde $C_{i,j}$ representa el número mínimo de operaciones necesarias para convertir $x_{1\ldots i}$ en $y_{1\ldots j}$.

Algorithm Seller's Algorithm

1: **procedure** SELLERS

2: $C_{i,0} = \sum_{1 \leqslant r \leqslant i} D_{x_r}$

3: $C_{0,j} = \sum_{1 \leqslant r \leqslant j} I_{y_r}$

4: $C_{i,j} = \min\left(C_{i-1,j-1} + R_{x_i,y_j}, C_{i-1,j} + D_{x_i}, C_{i,j-1} + I_{y_j}\right)$

Figura 5 Algoritmo de Sellers.

Así mismo, la posición $C_{|x|,|y|} = ed(x,y)$ y las variables $, I_{y_j}$ representan la sustitución del carácter x_i por y_j, la eliminación del carácter x_i y la inserción del carácter y_j respectivamente. $C_{i,0}$ y $C_{0,j}$ representan la distancia de edición entre una cadena de longitud i o j y la cadena vacía.

Considerando los últimos caracteres x_i y y_j, si los caracteres son iguales entonces no es necesario considerarlos y se procede con el valor de conversión de $x_{1\ldots i-1}$ en $y_{1\ldots j-1}$. Por otro lado, si no coincidieran, se debe evaluar el eliminar x_i, insertar y_j o sustituir x_i por y_j. La tabla 1 ilustra el algoritmo calculando ed(información,informática), donde se asume que el costo de todas las operaciones es 1 y 0 en el caso del reemplazo de un carácter por sí mismo.

El algoritmo tiene una complejidad de $O(|x||y|)$ en el peor y caso medio. Por otra parte, el espacio requerido es solo de $O(\min(|x|,|y|))$, esto ya que sólo es necesaria la columna anterior para calcular la siguiente, y por lo cual solo se conserva una columna y se realiza la actualización de la misma.

La secuencia de operaciones realizadas para transformar x en y pueden ser fácilmente deducidas de la matriz, trazando el camino desde $C_{|x|,|y|}$ hacia $C_{0,0}$ siguiendo las

	λ	i	n	f	o	r	m	a	t	i	c	a
λ	0	1	2	3	4	5	6	7	8	9	10	11
i	1	0	1	2	3	4	5	6	7	8	9	10
n	2	1	0	2	3	4	5	6	7	8	9	10
f	3	2	1	0	1	2	3	4	5	6	7	8
o	4	3	2	1	0	1	2	3	4	5	6	7
r	5	4	3	2	1	0	1	2	3	4	5	6
m	6	5	4	3	2	1	0	1	2	3	4	5
a	7	6	5	4	3	2	1	0	1	2	3	4
c	8	7	6	5	4	3	2	1	1	2	2	3
i	9	8	7	6	5	4	3	2	2	1	2	3
o	10	9	8	7	6	5	4	3	3	2	2	3
n	11	10	9	8	7	6	5	4	4	3	3	3

Tabla 1 Algoritmo de Sellers para el cálculo de la distancia de edición entre “información” e “informática” la cual tiene como resultado una distancia de edición igual a 3.

posiciones que coincidan con la formula presentada. En este caso es necesario el almacenamiento de toda matriz o por lo menos de un área alrededor de la diagonal principal.

Los siguientes teoremas son esenciales para la mejora teórica de los algoritmos y son expuestos y demostrados en (Masek & Paterson, 1978).

Teorema 2.1 Sea $d_j[i] = C_{i,j} - C_{i-1,j}$ y $d'_j[i] = C_{i,j} - C_{i,j-1}$ las diferencias horizontal y vertical en la matriz C, si se tiene un d_j o d'_j es posible expresar estos vectores en términos de los vectores d_{j-1}, d'_{j-1} respectivamente en una formula recursiva.

$$d_j[i] = C_{i,j} - C_{i-1,j}$$

$$= \min\left(C_{i-1,j-1} + R_{a,b}, C_{i-1,j} + D_a, C_{i,j-1} + I_b\right) - C_{i-1,j}$$

$$= \min\left(C_{i-1,j-1} + R_{a,b} - C_{i-1,j}, C_{i-1,j} + D_a - C_{i-1,j}, C_{i,j-1} + I_b - C_{i-1,j}\right)$$

$$= \min\left(R_{a,b} - \left(C_{i-1,j} - C_{i-1,j-1}\right), D_a, I_b + C_{i,j-1} - C_{i-1,j} + C_{i-1,j-1} - C_{i-1,j-1}\right)$$

$$= \min\left(R_{a,b} - \left(C_{i-1,j} - C_{i-1,j-1}\right), D_a, I_b + \left(C_{i,j-1} - C_{i-1,j-1}\right) - \left(C_{i-1,j} - C_{i-1,j-1}\right)\right)$$

$$= \min\left(R_{a,b} - \left(C_{i-1,j} - C_{i-1,j-1}\right), D_a, I_b + \left(C_{i,j-1} - C_{i-1,j-1}\right) - \left(C_{i-1,j} - C_{i-1,j-1}\right)\right)$$

$$d_j[i] = \min\left(R_{a,b} - \left(C_{i-1,j} - C_{i-1,j-1}\right), D_a, I_b + d_{j-1}[i] - \left(C_{i-1,j} - C_{i-1,j-1}\right)\right)$$

La incorporación de $C_{i-1,j-1} - C_{i-1,j-1}$ se justifica si se define $c_{i,j} = C_{i,j} - C_{i,j-1}$ con lo cual la formula recursiva d_j queda:

$$d_j[i] = \min\left(R_{a,b} - c_{i-1,j}, D_a, I_b + d_{j-1}[i] - c_{i-1,j}\right)$$

Es importante notar que $d_j'[i]$ y $c_{i,j}$ están definidos de la misma forma, es decir, tienen valores iguales. Se opta por renombrar la diferencia $C_{i,j} - C_{i,j-1}$ dentro de la formula de d_j por fines prácticos, ya que se desea tener independencia en los cálculos de diferencias horizontales y verticales. Por lo cual es importante expresar $c_{i,j}$ en forma de una ecuación recursiva en términos de $c_{i-1,j}$.

$$c_{i,j} = C_{i,j} - C_{i,j-1}$$

$$= \left(d_j[i] + C_{i-1,j}\right) - \left(d_{j-1}[i] - C_{i-1,j-1}\right)$$

$$= d_j[i] - d_{j-1}[i] + \left(C_{i-1,j} - C_{i-1,j-1}\right)$$

$$c_{i,j} = d_j[i] - d_{j-1}[i] + c_{i-1,j}$$

Para d'_j:

$$d'_j[i] = C_{i,j} - C_{i,j-1}$$

$$= \min\left(C_{i-1,j-1} + R_{a,b}, C_{i-1,j} + D_a, C_{i,j-1} + I_b\right) - C_{i,j-1}$$

$$= \min\left(C_{i-1,j-1} + R_{a,b} - C_{i,j-1}, C_{i-1,j} + D_a - C_{i,j-1}, C_{i,j-1} + I_b - C_{i,j-1}\right)$$

$$= \min\left(C_{i-1,j-1} + R_{a,b} - C_{i,j-1}, C_{i-1,j} + D_a - C_{i,j-1} + C_{i-1,j-1} - C_{i-1,j-1}, I_b\right)$$

$$= \min\left(R_{a,b} - \left(C_{i,j-1} - C_{i-1,j-1}\right), D_a + \left(C_{i-1,j} - C_{i-1,j-1}\right) - \left(C_{i,j-1} - C_{i-1,j-1}\right), I_b\right)$$

$$d'_j[i] = \min\left(R_{a,b} - \left(C_{i,j-1} - C_{i-1,j-1}\right), D_a + d'_{j-1}[i-1] - \left(C_{i,j-1} - C_{i-1,j-1}\right), I_b\right)$$

Se define $c'_{i,j} = C_{i,j} - C_{i-1,j}$ con lo cual d'_j queda:

$$d'_j[i] = \min\left(R_{a,b} - c'_{i,j-1}, D_a + d'_{j-1}[i-1] - c'_{i,j-1}, I_b\right)$$

Y de igual forma que antes se expresa $c'_{i,j}$ en forma de una ecuación recursiva en términos de $c'_{i,j-1}$:

$$c'_{i,j} = C_{i,j} - C_{i-1,j}$$

$$= \left(d'_j[i] + C_{i,j-1}\right) - \left(d'_j[i-1] + C_{i-1,j-1}\right)$$

$$= d'_j[i] - d'_j[i-1] + \left(C_{i,j-1} - C_{i-1,j-1}\right)$$

$$c'_{i,j} = d'_j[i] - d'_j[i-1] + c'_{i,j-1}$$

Teorema 2.2 Sea $I = \max\{I_a | a \in \Sigma\}$, $D = \max\{D_a | a \in \Sigma\}$. Para todas las cadenas x, y tal que $1 \leq i \leq |x|, 1 \leq j \leq |y|$ se cumple que:

$$-I \leq d_j[i] \leq D$$

$$-D \leq d'_j[i] \leq I$$

Si para que las cadenas $x_{1\ldots i}, y_{1\ldots j}$ coincidan es necesario primero eliminar x_i, entonces:

$$C_{i,j} \leq C_{i-1,j} + D_{x_i} \text{ además } D_{x_i} \leq D$$

$$C_{i-j} - C_{i-1,j} \leq D$$

$$d_j[i] \leq D$$

Como $x_{1\ldots i-1}$ empareja con $y_{1\ldots j}$, así:

$$C_{i-1,j} \leq I_{x_i} + C_{i,j}$$

$$C_{i,j} - C_{i-1,j} \geq -I_{x_i} \text{ además } -I_{x_i} \geq -I$$

$$C_{i,j} - C_{i-1,j} \geq -I$$

$$d_j[i] \geq -I$$

Por lo cual se tiene que $-I \leq d_j[i] \leq D$.

Por otro lado, si para que las cadenas $x_{1\ldots i}, y_{1\ldots j}$ coincidan es necesario primero la inserción de y_j, entonces:

$$C_{i,j} \leq C_{i,j-1} + I_{y_j} \text{ además } I_{y_j} \leq I$$

$$C_{i-j} - C_{i,j-1} \leq I$$

$$d'_j[i] \leq I$$

Como $x_{1\ldots i-1}$ empareja con $y_{1\ldots j}$, así:

$$C_{i,j-1} \leq D_{y_j} + C_{i,j}$$

$$C_{i,j} - C_{i,j-1} \geq -D_{y_j} \text{ además } -D_{y_j} \geq -D$$

$$C_{i,j} - C_{i,j-1} \geq -D$$

$$d'_j[i] \geq -D$$

Por lo cual se tiene que $-D \leq d'_j[i] \leq I$.

Sea $\varpi = \{D_a | a \in \Sigma\} \cup \{I_a | a \in \Sigma\} \cup \{R_{a,b} | a, b \in \Sigma\}$. El conjunto ϖ es discreto si y solo si existe una constante r tal que cada elemento de ϖ es un múltiplo de r. Para alfabetos finitos, la función de costo que tienen como imagen enteros o números racionales son siempre discretos.

Para el presente trabajo, es importante señalar que las variables que representan la inserción, eliminación caracteres toman los valores $D_{x_i} = 1, I_{y_j} = 1, \forall x_i \in x, y_j \in y$, es decir, el costo de inserción o eliminación de cualquier carácter es 1, por otro lado, el reemplazo de caracteres $R_{x_i,y_j} = 1, \forall x_i \neq y_j$ y $R_{x_i,y_j} = 0, \forall x_i = y_j$ lo cual es interpretado como, el reemplazo entre caracteres distintos tiene un costo de 1 y en el caso del reemplazo de caracteres iguales un costo de 0.

Algorithm Seller's Algorithm

```
1:  procedure SELLERS
2:      for i ← 0, m do
3:          C_{i,0} = i
4:      for j ← 1, n do
5:          C_{0,j} = j
6:          for i ← 1, m do
7:              if x_i = y_j then
8:                  C_{i,j} = C_{i-1,j-1}
9:              else
10:                 C_{i,j} = 1 + min(C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1})
```

Figura 6 Algoritmo de Sellers para el cálculo de la distancia de edición entre dos cadenas con 1 como costo de las operaciones de inserción, eliminación y reemplazo de caracteres distintos.

Tras dicha asignación de valores el algoritmo de Sellers antes presentado se calcula como se muestra en la figura 6. De la misma forma modificamos los teoremas con los valores asignados a las operaciones, para el teorema 2.1 las diferencias verticales y horizontales $d_j[i]$ y $d'_j[i]$ de la matriz C son calculadas por la fórmula:

$$d_j[i] = \min\left(R_{a,b} - c_{i-1,j}, 1{,}1 + d_{j-1}[i] - c_{i-1,j}\right)$$

$$d'_j[i] = \min\left(R_{a,b} - c'_{i,j-1}, 1 + d'_{j-1}[i-1] - c'_{i,j-1}, 1\right)$$

Donde el valor de $R_{a,b} = 1$ si $a \neq b$ o $R_{a,b} = 0$ si $a = b$. De manera similar, el teorema 2.2 tiene los siguientes cambios, los valores $I = \max\{I_a | a \in \Sigma\} = 1$ ya que la eliminación de cualquier carácter es 1, siguiendo un razonamiento similar se tiene que, $D = \max\{D_a | a \in \Sigma\} = 1$. Por lo cual, los límites de las diferencias son:

$$-1 \leq d_j[i] \leq 1$$

$$-1 \leq d'_j[i] \leq 1$$

Algorithm Seller's Algorithm

```
1:  procedure SELLERS
2:      for i ← 0, m do
3:          C_i = i
4:      for j ← 1, n do
5:          antVal = 0
6:          for i ← 1, m do
7:              aux = C_i
8:              if x_i = y_j then
9:                  C_i = antVal
10:             else
11:                 C_i = 1 + min(antVal, C_{i-1}, C_i)
12:             antVal = aux
13:         if C_m ⩽ k then
14:             Ocurrencia en j
```

Figura 7 Algoritmo de Sellers adaptado a búsqueda de patrones en textos donde solo se conserva la columna actual de avaluación de la matriz C.

La primera modificación del algoritmo de Sellers se realiza ya que en la práctica en una búsqueda de algún patrón en un texto que contenga millones de caracteres el almacenamiento necesario para la matriz seria excesivo, por lo cual el algoritmo original de Sellers tiene buen desempeño cuando se trata de evaluar dos cadenas, pero en el caso de encontrarse con texto extensos es necesario una modificación de dicho algoritmo.

Para adaptar el algoritmo para la búsqueda de patrones en textos, el algoritmo es básicamente igual, con la diferencia de que se debe tomar en cuenta que cualquier posición del texto es un potencial inicio de alguna cadena que empareja con el patrón. Se modifica el algoritmo haciendo $C_{0,j} = 0, \forall j \in 0 \ldots n$, lo cual se interpreta como que el patrón vacío coincide con cero errores en cualquier posición del texto.

El algoritmo inicializa la columna vector C con los valores $C_i = i$ y realiza el procesamiento del texto carácter a carácter. En cada nuevo carácter evaluado T_j, la

	λ	L	I	M	P	I	E	Z	A		L	I	M	P	I	A	D	O	R		...
λ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...
L	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	...
I	2	1	0	1	2	1	2	2	2	2	1	0	1	2	1	2	2	2	2	2	...
M	3	2	1	0	1	2	2	3	3	3	2	1	0	1	2	2	3	3	3	3	...
P	4	3	2	1	0	1	2	3	4	4	3	2	1	0	1	2	3	4	4	4	...
I	5	4	3	2	1	0	1	2	3	4	4	3	2	1	0	1	2	3	4	5	...
A	6	5	4	3	**2**	**1**	**1**	**2**	3	4	5	4	3	**2**	**1**	**0**	**1**	**2**	3	4	...

Tabla 2 Algoritmo de Sellers adaptado a búsqueda de patrones en textos, tomar en cuenta que sólo se mantiene una columna en memoria la cual se actualiza al leer el siguiente carácter del texto. Se marca en rojo las posiciones que son reportadas como ocurrencias.

columna vector C se actualiza con la fórmula conocida y cada posición del texto donde $C_m \leq k$ debe ser reportada como ocurrencia del patrón. En la Figura 7 se muestra el algoritmo de Sellers adaptado a búsqueda en textos y la Tabla 2 ilustra el algoritmo en la búsqueda de un patrón en un texto.

El tiempo de tiempo de ejecución de este algoritmo $O(mn)$ y el espacio requerido $O(m)$. Dada la característica de solo conservar la columna vector C hace imposible el rastreo de las operaciones para transformar x en y. Muchos de los algoritmos al igual que este no presentan un método para obtener dicha secuencia de operaciones esto debido al alto costo en almacenamiento que eso supondría.

La siguiente modificación en el algoritmo de programación dinámica fue propuesta por (Ukkonen, 1985), la cual tiene un tiempo de ejecución de $O(kn)$ medio (Chang & Lampe, 1992). Ukkonen observó que las posiciones de las columnas que tenían valores mayores a $k + 1$, de arriba hacia abajo, no eran relevantes, es decir, no formaban parte de ninguna secuencia de operaciones que llevaban a una ocurrencia.

Algorithm Ukkonen's Algorithm

```
1:  procedure UKKONEN
2:      for i ← 0, k do
3:          C_i = i
4:      lact = k
5:      for j ← 1, n do
6:          antVal = 0
7:          for i ← 1, lact do
8:              aux = C_i
9:              if x_i = y_j then
10:                 C_i = antVal
11:             else
12:                 C_i = 1 + min(antVal, C_{i-1}, C_i)
13:             antVal = aux
14:         while C_lact > k do
15:             lact = lact − 1
16:         if lact = m then
17:             Ocurrencia en j
18:         else
19:             lact = lact + 1
```

Figura 8 Algoritmo de Ukkonen, nótese que solo se conserva la columna actual de avaluación de la matriz C, el algoritmo tiene un tiempo esperado de $O(kn)$.

Por lo cual define una celda o posición activa a aquella que tiene un valor de a lo más k, su algoritmo conserva la última posición activa por columna evitando trabajar en las posiciones siguientes. La última posición activa debe ser re calcula para cada columna, cuando se lee el siguiente carácter del texto, la última posición activa puede incrementarse en una unidad. De igual forma, también es posible que esta última posición sea inactiva para el siguiente carácter leído por lo cual es necesario también tomar en cuenta este caso.

La figura 8 ilustra el pseudo código de dicho algoritmo y la tabla 3 ejemplifica el algoritmo en la búsqueda del patrón “LIMPIA” en un texto.

	λ	L	I	M	P	I	E	Z	A		L	I	M	P	I	A	D	O	R		…
λ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…
L	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	…
I	2	1	0	1	2	1	2	2	2	2	1	0	1	2	1	2	2	2	2	2	…
M		2	1	0	1	2	2	3			2	1	0	1	2	2	3	3			…
P			2	1	0	1	2	3			2	1	0	1	2	3	4				…
I				2	1	0	1	2				2	1	0	1	2	3				…
A					2	1	1	2					2	1	0	1	2				…

Tabla 3 Algoritmo de Ukkonen, nótese que los espacios en blanco corresponden a posiciones de la matriz no calculadas.

3. RETÍCULAS RESIDUALES COMPLETAS Y RELACIONES DIFUSAS

Los términos y la notación usada en este capítulo siguen las nociones y notaciones de los siguientes textos: (Blyth, 2006), (Birkhoff, 1964) para la sección de conjuntos ordenados y retículas, (Belohlávek, 2002) para la sección de retículas residuales completas y para la de conjuntos y relaciones difusas. Toda la información, teoremas y definiciones son extraídas de los textos: (Ciric, Ignjatovic, & Bogdanovic, 2009), (Micic, 2014) principalmente.

3.1. CONJUNTOS ORDENADOS Y RETÍCULAS

Sea A un conjunto no vacío, para cualquier $R \subset A \times A$ se dice que R es una relación binaria o una relación sobre A. Las relaciones más conocidas y usadas son:

- La relación vacía denotada por $\emptyset$.
- La relación de identidad $\Delta_A = \{(x, x) | x \in A\}$.
- La relación universal $\nabla_A = \{(x, y) | x, y \in A\}$.

Si $a, b \in A$ y pertenecen a una relación R entonces puede denotarse por $(a, b) \in R$ o aRb. Así dadas las relaciones R y S sobre el conjunto A definimos la composición de las relaciones $R \circ \mathrm{S}$ y la relación inversa R^{-1} como:

$$R \circ \mathrm{S} = \{(a, c) \in A \times A | (\exists b \in A)(a, b) \in R \wedge (b, c) \in S\}$$

$$R^{-1} = \{(a, b) | (b, a) \in R\}$$

Dada una relación R sobre un conjunto no vacío A se dice:

- Reflexiva sí $(a,a) \in R, \forall a \in A$.
- Simétrica sí $(a,b) \in R \rightarrow (b,a) \in R, \forall a,b \in A$.
- Antisimétrica sí $(a,b) \in R \wedge (b,a) \in R \rightarrow a = b, \forall a,b \in A$.
- Transitiva sí $(a,b) \in R \wedge (b,c) \in R \rightarrow (a,c) \in R, \forall a,b,c \in A$.

Se llama relación de equivalencia sobre A a aquellas relaciones que sean reflexivas, simétricas y transitivas. Una relación reflexiva, antisimétrica y transitiva sobre un conjunto no vacío A es llamada orden parcial sobre A.

Un orden parcial u orden es denotado por $\leqslant$, así el par $(A, \leqslant)$, donde A es un conjunto no vacío y $\leqslant$ es un orden sobre A, es llamado un conjunto ordenado parcial o simplemente un conjunto ordenado. Por lo que se dirá simplemente que A es un conjunto ordenado.

Un ejemplo importante sería el conjunto ordenado $(\mathbb{N}, |)$, donde $\mathbb{N}$ es el conjunto de números naturales y $|$ la relación de divisibilidad, y cumple que:

- $a|a, \forall a \in \mathbb{N}$.
- $a|b \wedge b|a \rightarrow a = b, \forall a,b \in \mathbb{N}$.
- $a|b \wedge b|c \rightarrow a|c, \forall a,b,c \in A$.

Si $\leqslant$ es un orden sobre A, entonces $<$ denota la relación sobre A dada por:

$$a < b \leftrightarrow a \leqslant b \wedge \mathrm{a} \neq b$$

Un orden $\leqslant$ sobre A es ordenado lineal si $\forall a, b \in A$, se tiene que $a \leqslant b$ o $b \leqslant a$ en cuyo caso A es un conjunto linealmente ordenado.

Una función ϕ del conjunto ordenado A al conjunto ordenado B es llamado isotónica o preservadora de orden si $a \leqslant b \rightarrow \phi(a) \leqslant \phi(b), \forall a, b \in A$, y es llamada antitónica si $a \leqslant b \rightarrow \phi(a) \geqslant \phi(b), \forall a, b \in A$. Por otro lado, la función ϕ es un isomorfismo de los conjuntos ordenados A y B o isomorfismo ordenado de A a B, sí ϕ es biyectiva de A a B así ϕ y ϕ^{-1} son funciones isotónicas.

Sea A un conjunto ordenado y el elemento $a \in A$ se dice:

- Elemento mínimo si $x \leqslant a \rightarrow x = a, \forall x \in A$.
- Elemento máximo si $a \leqslant x \rightarrow x = a, \forall x \in A$.
- Menor elemento si $\forall x \in A \rightarrow a \leqslant x$.
- Mayor elemento si $\forall x \in A \rightarrow x \leqslant a$.

Sea H un subconjunto no vacío del conjunto ordenado A, el elemento $a \in A$ se dice:

- Límite superior si $x \leqslant a, \forall x \in H$.
- Límite inferior si $a \leqslant x, \forall x \in H$.
    - Supremo del conjunto H, sí es el límite superior de H y $\forall b$ límite superior de H,se mantiene $a \leqslant b$.
    - Ínfimo del conjunto H, sí es el límite inferior de H y $\forall b$ límite inferior de H,se mantiene $b \leqslant a$.

El supremo de H es denotado $\bigvee H$ y el ínfimo $\bigwedge H$. Si $H = \{a_i | i \in I\}$ se puede representar el supremo e ínfimo por:

$$\bigvee_{i \in I} a_i \text{ y } \bigwedge_{i \in I} a_i$$

Un conjunto ordenado, donde cada subconjunto de dos elementos tiene supremo e ínfimo es llamado retícula. Sea L una retícula. Definimos las operaciones binarias:

$$\bigvee(a, b) = a \vee b \text{ y } \bigwedge(a, b) = a \wedge b$$

Las operaciones $\vee$ y $\wedge$ son llamadas unión e intersección. Así, $\bigvee H$ es la unión del conjunto H y $a \vee b$ es la unión de los elementos a y b, de igual forma $\bigwedge H$ es la intersección del conjunto H y $a \wedge b$ es la intersección de los elementos a y b.

Teorema 3.1 Sea L una retícula, $\forall a, b, c \in L$ se cumplen las siguientes condiciones llamadas axiomas de retículas:

- $a \wedge a = a, a \vee a = a$ (idempotencia).
- $a \wedge b = b \wedge a, a \vee b = b \vee a$ (conmutatividad).
- $(a \wedge b) \wedge c = a \wedge (b \wedge c), (a \vee b) \vee c = a \vee (b \vee c)$ (asociatividad).
- $a \wedge (b \vee a) = a, a \vee (b \wedge a) = a$ (absorción).

Un subconjunto X de una retícula L es llamada subretícula sí $a \wedge b \in X$ y $a \vee b \in X$ para todo $a, b \in X$.

Para una retícula L y un elemento $a \in L$, las subretículas son de intervalo semiabierto de la retícula L,

$$[a) = \{x \in L | a \leqslant x\} \text{ y } (a] = \{x \in L | x \leqslant a\}$$

Y $\forall a, b \in L$ las subretículas son de intervalo abierto y cerrado respectivamente

$$(a, b) = \{x \in L | a < x < b\} \text{ y } [a, b] = \{x \in L | a \leqslant x \leqslant b\}$$

Un subconjunto no vacío I de una retícula L es llamada ideal si:

1) $x \leqslant a$ implica que $x \in I$, para todo elemento $a \in I$ y $x \in L$.
2) $a \vee b \in I, \forall a, b \in I$.

es decir que el subconjunto I es el ideal de la retícula L sí $a \vee b \in I \; sii \; a, b \in I$.

La noción dual de este concepto es el ideal dual, definido como un subconjunto no vacío D de la retícula L si:

1) $a \leqslant x$ implica que $x \in I$, para todo elemento $a \in D$ y $x \in L$.
2) $a \wedge b \in I, \forall a, b \in I$.

El menor elemento de una retícula L, si existiese, es expresado por 0, y el mayor elemento, si existiese, es expresado por 1, las retículas que cumplen estas condiciones son llamadas retículas limitadas.

Las retículas distributivas llamadas así ya que cumplen las siguientes condiciones:

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c), \forall a, b, c \in L$$

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c), \forall a, b, c \in L$$

Sea L una retícula limitada con 0 y 1 como menor y mayor elemento respectivamente. Un elemento $b \in L$ es llamado complemento del elemento $a \in L$ si:

$$a \wedge b = 0, a \vee b = 1$$

En tal caso, $a \in L$ es también complemento del elemento $b \in L$, por lo cual la relación de ser complemento de es una relación simétrica. En cada retícula distributiva cada elemento $a \in L$ tiene a lo más un complemento, el cual es denotado por $a' \in L$.

Una retícula distributiva limitada en donde cada elemento tiene un complemento es llamada retícula booleana. La función $a \to a'$ es una operación unaria sobre L llamada operación de complemento. Una retícula donde cada subconjunto tiene un supremo e ínfimo es llamada retícula completa, donde evidente que toda retícula completa es limitada. Un subconjunto K de una retícula completa L es una subretícula completa de L si el supremo e ínfimo de cada subconjunto no vacío de K pertenece a K.

3.2. RETÍCULAS RESIDUALES COMPLETAS

En lógica difusa, cada proposición tiene asignado un grado de verdad de una escala L de valores de verdad. Si las proposiciones φ y ψ tienen a y b de grado de verdad respectivamente denotado por $\|\varphi\| = a$ y $\|\psi\| = b$, entonces $a \leqslant b$ denota que φ tiene menor grado de verdad que ψ, por lo que L es un orden parcial. Si se asigna a 0 el valor de verdad de "completamente falso" y a 1 se asigna el valor de verdad de "completamente

verdadero", entonces L es provisto con un orden parcial $\leqslant$ además de 0 y 1 como su menor y mayor elemento en L.

Sea $\{\varphi_i | i \in I\}$ un conjunto de preposiciones. Podemos asumir que el valor de verdad de la expresión "existe un i tal que φ_i" es un supremo de todos los valores de verdad φ_i, es decir, $\|\text{existe un } i \text{ tal que } \varphi_i\| = \bigvee_{i \in I} \|\varphi_i\|$. Por lo tanto, L debe tener un supremo e ínfimo arbitrarios, es decir, $(L, \leqslant, 0, 1)$ debe ser una retícula completa.

Dicha escala L requiere tener funciones de verdad de conectivos lógicos, se añade las funciones binarias $\otimes: L \times L \to L$ como la conjunción, $\to: L \times L \to L$ como implicación.

Como es sabe, la conjunción es asociativa y conmutativa, además el hecho de que el grado de verdad de la conjunción de una proposición φ con una completamente verdadera ψ sea igual a φ, $\varphi \otimes 1 = \varphi$, implica que 1 sea un elemento natural para $\otimes$, por lo cual, $(L, \otimes, 1)$ debe ser un monoide conmutativo, es decir cumple que:

- $\forall a, b \in L, a \otimes b \in L$
- $\forall a, b, c \in L, a \otimes (b \otimes c) = (a \otimes b) \otimes c$
- $\exists! e \in L, \forall a \in L | e \otimes a = a \otimes e = a$
- $\forall a, b \in L, a \otimes b = b \otimes a$

En la lógica clásica uno de los conceptos más comúnmente usados es el modus ponens. Una apropiada formulación del modus ponens en lógica difuso sería: si φ tiene un grado de verdad de por lo menos a y $\varphi \to \psi$ tiene un grado de verdad de por lo menos b entonces se infiere que ψ es valido en un grado de por lo menos $a \otimes b$.

Dicha formulación debe satisfacer dos puntos: debe ser correcto o solvente y dar la estimación de validez más alta posible de ψ. Solvencia, si el grado de verdad de φ es por lo menos a $(a \leqslant \varphi)$[2] y el grado de verdad de $\varphi \to \psi$ es por lo menos b $(b \leqslant \|\varphi \to \psi\|)$ entonces el valor de verdad de ψ es por lo menos tan alto como el grado obtenido por modus ponens $(a \otimes b \leqslant \|\psi\|)$.

Mayor estimación de validez posible de ψ, Sea $\|\varphi\| = a$ y $\|\psi\| = c$ entonces $\|\varphi \to \psi\| = a \to c$ y por la condición de solvencia, $a \otimes (a \to c)$ debe ser la menor estimación de c, es decir, $a \otimes (a \to c) \leqslant c$. Como $\otimes$ es no decreciente, cuanto mayor es la menor estimación de $a \otimes (a \to c)$, mayor es $a \to c$. Y como se quiere $a \otimes (a \to c)$ sea la más alta posible estimación de c, $a \to c$ debe ser del mayor grado posible para que $a \otimes (a \to c) \leqslant c$ se cumpla. En efecto, si b es cualquier grado de verdad para el cual se cumple $a \otimes b \leqslant c$ entonces $b \leqslant a \to c$.

De acuerdo a esto, $\to$ y $\otimes$ deben forman un par adjunto, por lo cual cumplen la propiedad de adjunción:

$$\forall a, b, c \in L, a \otimes b \leqslant c \leftrightarrow a \leqslant b \to c$$

La estructura algebraica que cumple todas las condiciones anteriores es una retícula residual completa.

Una retícula residual es un algebra $\mathcal{L} = (L, \wedge, \vee, \otimes, \to, 0, 1)$ tal que:

[2] La notación $a \leqslant b$ denota que la proposición con valor de verdad a tiene menor grado de verdad que la proposición con grado de verdad b.

(L1). $(L,\wedge,\vee,0,1)$ es una retícula con 0 como menor elemento y 1 como mayor elemento.

(L2). $(L,\otimes,1)$ es un monoide conmutativo con 1.

(L3). $\otimes$ y $\rightarrow$ forman un par adjunto.

Si, además, $(L,,\wedge,\vee,0,1)$ es una retícula completa, entonces $\mathcal{L}$ es llamada retícula residual completa.

Las operaciones $\otimes$ llamada multiplicación y $\rightarrow$ llamado residuo, modelan la conjunción y la implicación, el supremo $\vee$ e ínfimo $\wedge$ modelan los cuantificadores existencial y general respectivamente. Sobre una retícula residual completa se definen las siguientes operaciones:

- Bi residuo o bi implicación: $a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$
- Negación: $\neg a = a \rightarrow 0$
- Grado n – ésimo: $a^0 = 1$ y $a^{n+1} = a^n \otimes 1$

La bi implicación es la operación usada para modelar la equivalencia de valores de verdad, mientras que la negación es usada para modelar el complemento de un valor de verdad.

Las estructuras de valores de verdad más estudiadas y aplicadas, definidas sobre el intervalo $[0,1]$ con:

$$a \wedge b = \min(a,b) \quad \text{y} \quad a \vee b = \max(a,b)$$

Estructura de *Lukasiewicz*:

$$a \otimes b = \max(a + b - 1,0),\ a \to b = \min(1 - a + b, 1)$$

Estructura de *Goguen*:

$$a \otimes b = a * b,\ a \to b = \begin{cases} 1, & si\ a \leqslant b \\ b/a, & \text{e.o.c.} \end{cases}$$

Estructura de *Gödel*:

$$a \otimes b = \min(a, b)\ ,\ a \to b = \begin{cases} 1, & si\ a \leqslant b \\ b, & e.o.c \end{cases}$$

Otro importante conjunto de valores de verdad es el conjunto $\{a_0, a_1, \dots, a_n\}$, donde $0 = a_0 < \cdots < a_n = 1$ con:

$$a_k \otimes a_l = a_{\max(k+l-n,0)} \text{ y } a_k \to a_l = a_{\min(n-k+l,n)}$$

Las estructuras de Lukasiewicz, Goguen y Gödel son retículas residuales inducidas por las normas – t. La norma triangular o norma – t es una operación binaria sobre $[0,1]$ la cual es asociativa, conmutativa, monótona. Así la función $\otimes: [0,1] \times [0,1] \to [0,1]$ satisface asociatividad, conmutatividad, monotonicidad no decreciente y elemento neutro:

$$(a \otimes b) \otimes c = a \otimes (b \otimes c)$$

$$a \otimes b = b \otimes a$$

$$b_1 \leqslant b_2 \Rightarrow a \otimes b_1 \leqslant a \otimes b_2$$

$$a \otimes 1 = a$$

Generalmente, un algebra $([0,1],\wedge,\vee,\otimes,\rightarrow,0,1)$ es una retícula residual completa si y solo si $\otimes$ es una norma – t continua por izquierda, es decir, $\lim_{n\to\infty}(a_n \otimes b) = \left(\lim_{n\to\infty} a_n\right) \otimes b$ y el residuo está definido por $x \rightarrow y = \bigvee\{u \in [0,1] | u \otimes x \leqslant y\}$.

Los siguientes teoremas muestran las propiedades básicas de las retículas residuales:

Teorema 3.2 Para toda retícula residual se cumple:

$$b \leqslant a \rightarrow (a \otimes b),\ a \leqslant (a \rightarrow b) \rightarrow b$$

$$a \otimes (a \rightarrow b) \leqslant b$$

$$a \leqslant b \Leftrightarrow a \rightarrow b = 1$$

$$a \rightarrow a = 1, \qquad a \rightarrow 1 = 1, \qquad 1 \rightarrow a = a$$

$$0 \rightarrow a = 1$$

$$a \otimes 0 = 0 \otimes a = 0$$

$$a \otimes b \leqslant a, \qquad a \leqslant b \rightarrow a$$

$$a \otimes b \leqslant a \wedge b$$

$$a \otimes b \rightarrow c = a \rightarrow (b \rightarrow c)$$

$$(a \rightarrow b) \otimes (b \rightarrow c) \leqslant (a \rightarrow c)$$

$a \rightarrow b$ es el mayor elemento de $\{c | a \otimes c \leqslant b\}$

$a \otimes b$ es el menor elemento de $\{c | a \leqslant b \rightarrow c\}$

El siguiente teorema muestra que respecto de $\leqslant$, la operación $\otimes$ es isotónica en ambos argumentos, la operación $\rightarrow$ es isotónica en el segundo y antitónica en el primer argumento.

Teorema 3.3 En cualquier retícula residual se cumple que:

$$b_1 \leqslant b_2 \Longrightarrow a \otimes b_1 \leqslant a \otimes b_2$$

$$b_1 \leqslant b_2 \Longrightarrow a \to b_1 \leqslant a \to b_2$$

$$a_1 \leqslant a_2 \Longrightarrow a_2 \to b \leqslant a_1 \to b$$

Teorema 3.4 En cualquier retícula residual se cumplen las siguientes desigualdades:

$$a \to b \leqslant (a \wedge c) \to (b \wedge c)$$

$$a \to b \leqslant (a \vee c) \to (b \vee c)$$

$$a \to b \leqslant (a \otimes c) \to (b \otimes c)$$

$$a \to b \leqslant (b \to c) \to (a \to c)$$

$$a \to b \leqslant (c \to a) \to (c \to b)$$

Los siguientes teoremas describen la relación entre los operadores $\wedge$ y $\vee$, para cualquier familia de elementos de una retícula residual y las operaciones $\otimes$ y $\to$.

Teorema 3.5 Las siguientes afirmaciones se cumplen para todo conjunto I:

$$a \otimes \left(\bigvee_{i \in I} b_i \right) = \bigvee_{i \in I} (a \otimes b_i)$$

$$a \to \left(\bigwedge_{i \in I} b_i \right) = \bigwedge_{i \in I} (a \to b_i)$$

$$\left(\bigvee_{i \in I} a_i \right) \to b = \bigwedge_{i \in I} (a_i \to b)$$

$$\bigvee_{i\in I}(a_i \to b) = \left(\bigwedge_{i\in I} a_i\right) \to b$$

$$a \otimes \left(\bigwedge_{i\in I} b_i\right) \leqslant \bigwedge_{i\in I}(a \otimes b_i)$$

$$\bigwedge_{i\in I}(a_i \to b_i) \leqslant \left(\bigwedge_{i\in I} a_i\right) \to \left(\bigwedge_{i\in I} b_i\right)$$

$$\bigwedge_{i\in I}(a_i \to b_i) \leqslant \left(\bigvee_{i\in I} a_i\right) \to \left(\bigvee_{i\in I} b_i\right)$$

$$\bigvee_{\in}(a \to b_i) \leqslant a \to \left(\bigwedge_{i\in I} b_i\right)$$

Teorema 3.6 En cualquier retícula residual se cumplen las siguientes afirmaciones:

$$a \to b = \big((a \to b) \to b\big) \to b$$

$$a \otimes (a \to b) = b \Leftrightarrow (\exists c)(a \otimes c = b)$$

$$a \to (a \otimes b) = b \Leftrightarrow (\exists c)(a \to c = b)$$

$$a \to (a \otimes b) = b \Leftrightarrow (\exists c)(a \to c = b)$$

$$(b \to a) \to a = b \Leftrightarrow (\exists c)(c \to a = b)$$

$$(a \wedge b) \otimes (a \vee b) \leqslant a \otimes b$$

$$a \vee b \leqslant \big((a \to b) \to b\big) \wedge \big((b \to a) \to a\big)$$

$$a \wedge b \geqslant a \otimes (a \to b)^3$$

$$a \otimes (b \to c) \leqslant b \to (a \otimes c)$$

3.3. CONJUNTOS Y RELACIONES DIFUSAS

Se toma a la retícula residual completa $\mathcal{L}$ como la estructura de valores de verdad en el resto del texto.

Un subconjunto difuso de un conjunto A sobre $\mathcal{L}$, es cualquier función de A en $\mathcal{L}$. Sean f y g dos subconjuntos difusos de un conjunto A, la igualdad de f y g se define como la función de igualdad usual:

$$f = g \text{ si y solo si } f(x) = g(x), \forall x \in A$$

La inclusión $f \leqslant g$ es definida:

$$f \leqslant g \text{ si y solo si } f(x) \leqslant g(x), \forall x \in A$$

Junto a este orden parcial el conjunto $\mathcal{F}(A)$ de todos los subconjuntos difusos de A forman una retícula residual completa, donde la intersección $\bigwedge_{i \in I} f_i$ y la unión $\bigvee_{i \in I} f_i$ de una familia arbitraria $\{f_i\}_{i \in I}$ de subconjuntos difusos de A son funciones de A en $\mathcal{L}$ definidos como:

$$\left(\bigwedge_{i \in I} f_i\right)(x) = \bigwedge_{i \in I} f_i(x), \qquad \left(\bigvee_{i \in I} f_i\right)(x) = \bigvee_{i \in I} f_i(x)$$

[3] La relación $a \geqslant b$ representa la relación inversa de $a \leqslant b$.

y el producto $f \otimes g$ es un subconjunto difuso definido como: $(f \otimes g)(x) = f(x) \otimes g(x), \forall x \in A$.

La parte concisa de un subconjunto difuso $f \in \mathcal{F}(A)$ es un subconjunto conciso de A definido como $\hat{f} = \{a \in A | f(a) = 1\}$. El kernel de un subconjunto difuso f de un conjunto A, denotado por ker_f, es una relación de equivalencia sobre A definida como:

$$ker_f = \{(x, y) \in A \times A | f(x) = f(y)\}$$

La altura de un subconjunto difuso f, denotado por $\|f\|$, es definido por:

$$\|f\| = \bigvee_{x \in A} f(x)$$

Una relación difusa entre los conjuntos A y B es cualquier función de $A \times B \to \mathcal{L}$. El conjunto de todas las relaciones entre A y B se denota con $\mathcal{R}(A, B)$. En particular, una relación difusa sobre el conjunto A es cualquier función de $A \times A \to \mathcal{L}$, el conjunto de todas las relaciones sobre A se denota por $\mathcal{R}(A)$.

La inversa de una relación difusa $\varphi \in \mathcal{R}(A, B)$ es una relación difusa $\varphi^{-1} \in \mathcal{R}(B, A)$ definida por $\varphi^1(b, a) = \varphi(a, b), \forall a \in A, \forall b \in B$.

Una relación concisa es una relación difusa que toma valores del conjunto $\{0,1\}$, y si φ es una relación concisa de A hacia B, las expresiones $\varphi(a, b) = 1$ y $(a, b) \in \varphi$ tienen el mismo significado. La parte concisa de una relación difusa φ, denotada por $\hat{\varphi}$ es una relación concisa donde se satisface:

$$(a, b) \in \hat{\varphi} \Leftrightarrow \varphi(a, b) = 1, \qquad \forall a, b \in A$$

Para los conjuntos no vacíos A, B, C y las relaciones difusas $\varphi \in \mathcal{R}(A,B), \psi \in \mathcal{R}(B,C)$ la composición $\varphi \circ \psi$ es una relación difusa de $\mathcal{R}(A,C)$ definida como:

$$(\varphi \circ \psi)(a,c) = \bigvee_{b \in B} \varphi(a,b) \otimes \psi(b,c), \forall a \in A \wedge c \in C$$

Si $f \in \mathcal{F}(A), \varphi \in \mathcal{R}(A,B)$ y $g \in \mathcal{F}(B)$, las siguientes composiciones son subconjuntos difusos de B y A respectivamente.

$$(f \circ g)(b) = \bigvee_{a \in A} f(a) \otimes \varphi(a,b)\,, \forall b \in B$$

$$(\varphi \circ g)(a) = \bigvee_{b \in B} \varphi(a,b) \otimes g(b), \forall a \in A$$

En particular, para los subconjuntos difusos f y g de A:

$$f \circ g = \bigvee_{a \in A} f(a) \otimes g(a)$$

Sean A, B, C y D conjuntos no vacíos, entonces para cualquier $\varphi_1 \in \mathcal{R}(A,B), \varphi_2 \in \mathcal{R}(B,C)$ y $\varphi_3 \in \mathcal{R}(C,D)$ se cumple:

$$(\varphi_1 \circ \varphi_2) \circ \varphi_3 = \varphi_1 \circ (\varphi_2 \circ \varphi_3)$$

Para $\varphi_0 \in \mathcal{R}(A,B), \varphi_1, \varphi_2 \in \mathcal{R}(B,C)$ y $\varphi_3 \in \mathcal{R}(C,D)$ se cumple:

$$\varphi_1 \leqslant \varphi_2 \text{ implica que } \varphi_0 \circ \varphi_1 \leqslant \varphi_0 \circ \varphi_2 \text{ y } \varphi_1 \circ \varphi_3 \leqslant \varphi_2 \circ \varphi_3$$

Para cualquier $\varphi \in \mathcal{R}(A,B), \psi \in \mathcal{R}(B,C), f \in \mathcal{F}(A), g \in \mathcal{F}(B)$ y $h \in \mathcal{F}(C)$ se verifica que:

$$(f \circ \varphi) \circ \psi = f \circ (\varphi \circ \psi), (f \circ \varphi) \circ g = f \circ (\varphi \circ g), (\varphi \circ \psi) \circ h = \varphi \circ (\psi \circ h)$$

Finalmente, para todo $\varphi, \varphi_i \in \mathcal{R}(A, B), (i \in I)$ y $\psi, \psi_i \in \mathcal{R}(B, C), (i \in I)$ se tiene:

$$(\varphi \circ \psi)^{-1} = \varphi^{-1} \circ \psi^{-1}$$

$$\varphi \circ \left(\bigvee_{i \in I} \psi_i \right) = \bigvee_{i \in I} (\varphi \circ \psi_i), \qquad \left(\bigvee_{i \in I} \varphi_i \right) \circ \psi = \bigvee_{i \in I} (\varphi \circ \psi_i)$$

$$\left(\bigvee_{i \in I} \varphi_i \right)^{-1} = \bigvee_{i \in I} \varphi_i^{-1}$$

Una relación difusa $\mathcal{R}$ sobre A se dice que es:

- Reflexiva difusa si $\mathcal{R}(a, a) = 1, \forall a \in A$.
- Simétrica difusa si $\mathcal{R}(a, b) = \mathcal{R}(b, a), \forall a, b \in A$.
- Transitiva difusa si $\mathcal{R}(a, b) \otimes \mathcal{R}(b, c) \leqslant \mathcal{R}(a, c), \forall a, b, c \in A$.

Para una relación difusa R sobre el conjunto A, la relación R^{∞} sobre A es la menor relación difusa transitiva sobre A que contiene R, también llamada clausura transitiva de R y se define como:

$$R^{\infty} = \bigvee_{n \in N} R^n$$

Una relación difusa reflexiva, simétrica y transitiva sobre A se llama equivalencia difusa. Con respecto a la inclusión de relaciones difusas, el conjunto $\mathcal{E}(A)$ de todas las equivalencias difusas sobre A es una retícula completa. Una equivalencia difusa E sobre

un conjunto A es llamada una igualdad difusa si $E(a,b) = 1$ implica $a = b, \forall a, b \in A$, es decir, E es una igualdad difusa si y solo si su parte concisa $\hat{E}$ es una igualdad concisa.

La clase de equivalencia de una relación difusa E sobre A determinada por $a \in A$ es el subconjunto difuso E_a de A definido por:

$$E_a(b) = (a,b), \text{para cada } b \in A$$

El conjunto $A/E = \{E_a | a \in A\}$ es llamado el conjunto factor de A respecto a E. La función natural de A hacia A/E es la relación difusa $\varphi_E \in \mathcal{R}(A, A/E)$ definida por:

$$\varphi_E(a, E_b) = E(a,b), \forall a, b \in A$$

Una relación difusa sobre un conjunto A que sea reflexiva y transitiva es llamada un cuasi – orden difuso, al igual que el conjunto $\mathcal{E}(A)$, el conjunto $Q(A)$ de todos los cuasi – ordenes difusos sobre $\mathcal{A}$ es una retícula completa. Si R es la unión en $Q(A)$ de una familia de cuasi – ordenes difusos $\{R_i\}_{i \in I}$ sobre A, entonces R se representa:

$$R = \left(\bigvee_{i \in I} R \right)^{\infty} = \bigvee_{n \in N} \left(\bigvee_{i \in I} R \right)^{n}$$

El $R - afterset$ de a es el conjunto difuso $R_a \in L^A$ definido como:

$$R_a(b) = R(a,b), \text{para cualquier } b \in A$$

De igual forma, el $R - foreset$ de a, es el conjunto difuso $R^a \in L^A$ definido como:

$$R^a(b) = R(b,a), \text{para cualquier } b \in A$$

El conjunto de todos los $R - aftersets$ se denota por A/R, y el conjunto de todos los $R - foresets$ se denota por $A\backslash R$. Claramente, si R es una equivalencia difusa, entonces $A/R = A\backslash R$ es el conjunto de todas las clases de equivalencia de R.

Para un cuasi – orden difuso R sobre un conjunto A, la relación difusa E_R definida como $E_R = R \wedge R^{-1}$ es una equivalencia difusa sobre A la cual es llamada una equivalencia natural difusa de R.

Un cuasi – orden difuso R sobre un conjunto A es un orden difuso si $R(a,b) = R(b,a) = 1$ implica que $a = b, \forall a, b \in A$, esto si la equivalencia natural difusa E_R de R es una igualdad difusa. Claramente, un cuasi – orden difuso R es un orden difuso si y solo si su parte concisa $\hat{R}$ es un orden conciso.

Si f es un subconjunto difuso arbitrario de A, las relaciones difusas R_f y R^f sobre A son cuasi – ordenes difusos sobre A, además la relación difusa E_f definida a continuación es una equivalencia difusa sobre A.

$$R_f(a,b) = f(a) \to f(b), \qquad R^f(a,b) = f(a) \to f(b)$$

$$E_f(a,b) = f(a) \leftrightarrow f(b), \qquad a, b \in A$$

Teorema 3.7 Sea R un cuasi – orden difuso sobre el conjunto A y E la equivalencia natural difusa de R, entonces:

- Para $a, b \in A$ arbitrarios, las siguientes condiciones son equivalentes:

  A. $E(a,b) = 1$

B. $E_a = E_b$

C. $R^a = R^b$

D. $R_a = R_b$

- Las funciones $R_a \to E_a$ de A/R a A/E, y $R_a \to R^a$ de A/R a $A\backslash R$ son funciones biyectivas.

Lema 3.1 Sean $P, R \in Q(A)$ cuasi – ordenes difusos sobre A, entonces la relación $P \wedge R$ también lo es.

Lema 3.2 Sean $E, F \in \mathcal{E}(A)$ equivalencias difusas sobre A, entonces la relación $E \wedge F$ también lo es.

Lema 3.3 Sean $P, R \in Q(A)$ y $P \leqslant R$, entonces $P \circ R = R$.

Lema 3.4 Sean $P, R \in Q(A)$ y $P \leqslant R$, entonces se cumple que para cualquier $a \in A$:

$$P(b, c) \leqslant R^a(c) \to R^a(b), \qquad b, c \in A$$

Lema 3.5 Sean $E, F \in \mathcal{E}(A)$ y $E \leqslant F$, entonces $\forall a \in A$ se cumple que:

$$E(b, c) \leqslant F^a(b) \leftrightarrow F^a(c), \qquad b, c \in A$$

4. AUTÓMATAS DIFUSOS

Los conceptos y definiciones de este capítulo son presentados acorde a los resultados obtenidos en (Ignjatovic, Ciric, Bogdanovic, & Petkovic, 2010), (Stamenkovic, Ciric, & Ignjatovic, 2014), (Micic, 2014), (Ravi, Choubey, & Tripati, 2014), (Echabone, Garitagoitia, & González de Mendívil) y (Stamenkovic, Ciric, & Ignjatovic, Different Models of Automata with Fuzzy States, 2015).

4.1. DEFINICIÓN Y CONCEPTOS FUNDAMENTALES

Se toma a la retícula residual completa $\mathcal{L}$ como la estructura de valores de verdad y Σ como un alfabeto finito en el resto del texto.

Dado un alfabeto finito Σ y una función $f_{\tilde{L}}(w): \Sigma^* \to L$, entonces el conjunto $\tilde{L} = \{(w, f_{\tilde{L}}(w)) | w \in \Sigma^*\}$ es llamado lenguaje difuso sobre Σ y $f_{\tilde{L}}(w)$ la función de pertenencia de $\tilde{L}$. Así, $\tilde{L}$ es llamado un lenguaje difuso regular si:

- El conjunto $\{m | m \in L \text{ y } S_{\tilde{L}}(m) \neq \emptyset\}$ es finito.
- Para cada $m \in L$, el conjunto $S_{\tilde{L}}(m) = \{w | w \in \Sigma^* \text{ y } f_{\tilde{L}}(w) = m\}$ es regular.

Dado un alfabeto finito Σ y las funciones $f_{\tilde{L}}(w): \Sigma^* \to L$ y $g_{\tilde{L}}(w): \Sigma^* \to L$, entonces el conjunto $\{(w, f_{\tilde{L}}(w), g_{\tilde{L}}(w)) | w \in \Sigma^*\}$ es llamado lenguaje intuicionista difuso sobre Σ. Aquí, $f_{\tilde{L}}(w), g_{\tilde{L}}(w)$ representan las funciones de pertenencia y no pertenencia de $\tilde{L}$ respectivamente y para cualquier $w \in \Sigma^*, 0 \leq f_{\tilde{L}}(w) + g_{\tilde{L}}(w) \leq 1$.

Sea $\tilde{L}$ un lenguaje intuicionista difuso sobre Σ con $f_{\tilde{L}}(w)$ y $g_{\tilde{L}}(w)$ como funciones de pertenencia y no pertenencia, $\tilde{L}$ se dice lenguaje intuicionista difuso regular si:

- Los conjuntos $\{m|m \in L \text{ y } S_{\tilde{L}}(m) \neq \emptyset\}$ y $\{l|l \in L \text{ y } S_{\tilde{L}}(l) \neq \emptyset\}$ son finitos.
- Para cada $m \in L$, el conjunto $S_{\tilde{L}}(m)$ y para cada $l \in L$, el conjunto $S_{\tilde{L}}(l)$ son regulares.

Donde $S_{\tilde{L}}(m) = \{w|w \in \Sigma^* \text{ y } f_{\tilde{L}}(w) = m\}$, y $S_{\tilde{L}}(l) = \{w|w \in \Sigma^* \text{ y } g_{\tilde{L}}(w) = l\}$, además $S_{\tilde{L}}(m) = S_{\tilde{L}}(l)$, es decir, m y l representan el valor de pertenencia y no pertenencia de una misma cadena.

Un sistema de transición difusa sobre $\mathcal{L}$ y Σ es un par $\mathcal{A} = (Q, \delta)$ donde:

- Q es un conjunto no vacío de estados.
- $\delta: A \times \Sigma \times A \to L$ es un subconjunto difuso de $A \times \Sigma \times A$, llamado función de transición difusa.

Un autómata difuso sobre $\mathcal{L}$ y Σ, o simplemente un autómata difuso es una cuádrupla $\mathcal{A} = (Q, \delta, \sigma, \tau)$, donde:

- Q es un conjunto no vacío de estados.
- $\delta: Q \times \Sigma \times Q \to L$ es un subconjunto difuso de $Q \times \Sigma \times Q$, llamada función de transición difusa.
- $\sigma: Q \to L$ subconjunto difuso de Q, llamado conjunto de estados iniciales difusos.
- $\tau: Q \to L$ subconjunto difuso de Q, llamado conjunto de estados finales difusos.

Se interpreta $\delta(a, x, b)$ como el grado en el que una entrada $x \in \Sigma$ causa la transición del estado $a \in Q$ al estado $b \in Q$, se toma $\sigma(a)$ y $\tau(a)$ como el grado en que el estado a es inicial y final respectivamente.

Se toma Σ^* como un monoide libre sobre el alfabeto Σ, y sea $\lambda \in \Sigma^*$ la cadena vacía. La función δ se extiende a $\delta^*: Q \times \Sigma^* \times Q \to L$ como sigue:

$$\delta^*(a, \lambda, b) = \begin{cases} 1, & a = b \\ 0, & \text{en otro caso} \end{cases}$$

$a, b \in Q$ y si $u \in \Sigma^*$ y $x \in \Sigma$ entonces:

$$\delta^*(a, ux, b) = \bigvee_{c \in Q} \delta^*(a, u, c) \otimes \delta(c, x, b)$$

Por el teorema 3.5 se tiene que:

$$\delta^*(a, uv, b) = \bigvee_{c \in Q} \delta^*(a, u, c) \otimes \delta^*(c, v, b)$$

Para todo $a, b \in Q$ y $u, v \in \Sigma^*$. En otras palabras, si $w = x_1 x_2 \dots x_n$ es tal que $x_1, x_2, \dots, x_n \in \Sigma$, entonces:

$$\delta^*(a, w, b) = \bigvee_{c_1, \dots, C_{n-1} \in Q^{n-1}} \delta(a, x_1, c_1) \otimes \delta(c_1, x_2, c_2) \otimes \dots \otimes \delta(c_{n-1}, x_n, b)$$

Si para cualquier $u \in \Sigma^*$ definimos la relación difusa δ_u sobre Q por:

$$\delta_u(a, b) = \delta^*(a, u, b)$$

Para todo $a, b \in Q$, llamándola relación de transición difusa determinada por u, entonces $\delta^*(a, uv, b)$ puede escribirse así, para todo $u, v \in \Sigma^*$:

$$\delta_{uv} = \delta_u \circ \delta_v$$

Lo cual implica que, respecto a la composición de relaciones difusas, la relación de transición difusa es un semigrupo, el cual es llamado semigrupo de transición de un autómata difuso $\mathcal{A}$.

Sea $\mathcal{A} = (Q, \delta, \sigma, \tau)$ un autómata difuso. Así $\hat{\delta}$, la parte concisa de δ, es un subconjunto conciso de $Q \times \Sigma \times Q$, y de igual forma $\hat{\sigma}$ y $\hat{\tau}$ son subconjuntos concisos de σ y τ respectivamente. El autómata $\hat{\mathcal{A}} = (Q, \hat{\delta}, \hat{\sigma}, \hat{\tau})$ es un autómata no determinístico también llamado parte concisa de $\mathcal{A}$.

Un lenguaje difuso en Σ^* sobre $\mathcal{L}$, es cualquier subconjunto difuso de Σ^*, es decir, cualquier función de $\Sigma^* \to L$. El lenguaje difuso reconocido por un autómata difuso $\mathcal{A} = (Q, \delta, \sigma, \tau)$, denotado por $[\![\mathcal{A}]\!]$, es un lenguaje difuso en $\mathcal{F}(\Sigma^*)$ definido por:

$$[\![\mathcal{A}]\!](\lambda) = \sigma \circ \tau$$

$$[\![\mathcal{A}]\!](u) = \sigma \circ \delta_{x_1} \circ \delta_{x_2} \circ \ldots \circ \delta_{x_n} \circ \tau$$

Para cualquier $u = x_1 x_2 \ldots x_n \in \Sigma^+$, donde $x_1, x_2, \ldots, x_n \in \Sigma$. En otras palabras, el grado de pertenencia de la cadena u al lenguaje difuso $[\![\mathcal{A}]\!]$ es igual al grado con el cual $\mathcal{A}$ reconoce o acepta la cadena u.

4.2. DIFERENTES MODELOS DE AUTOMATAS DIFUSOS

Sea $\mathcal{A} = (Q, \delta, \sigma, \tau)$ un autómata difuso sobre Σ y $\mathcal{L}$. La función de transición difusa δ es llamada determinística concisa si $\forall x \in \Sigma$ y $\forall a \in Q, \exists a' \in Q$ tal que $\delta_x(a, a') = 1$, y $\delta_x(a, b) = 0$ para toda $b \in Q - \{a'\}$. El conjunto difuso de estados iniciales σ es llamado determinístico conciso si $\exists a_0 \in Q$ tal que $\sigma(a_0) = 1$ y $\sigma(a) = 0, \forall a \in Q - \{a_0\}$. Si

σ y δ son determinísticos concisos, entonces $\mathcal{A}$ se dice autómata finito difuso determinístico conciso.

Se define un autómata finito difuso determinístico conciso sobre Σ y $\mathcal{L}$ como un cuádrupla $\mathcal{A} = (Q, \delta, a_0, \tau)$, donde Q es un conjunto no vacío de estados, $\delta: Q \times \Sigma \to Q$ es la función de transición, $a_0 \in Q$ es un estado inicial y $\tau: Q \to L$ es un conjunto difuso de estados finales.

La función de transición δ puede ser extendida a la función $\delta^*: Q \times \Sigma^* \to Q$ con $\delta(a, \lambda) = a$, para cada $a \in Q$, y $\delta^*(a, ux) = \delta(\delta^*(a, u), x)$ para cualquier $a \in Q, u \in \Sigma^*$ y $x \in \Sigma$. El lenguaje aceptado por $\mathcal{A}$ es el lenguaje difuso $[\![\mathcal{A}]\!]: \Sigma^* \to L$ definido por:

$$[\![\mathcal{A}]\!](u) = \tau\big(\delta^*(a_0, u)\big), \text{ para cada } u \in \Sigma^*$$

Un autómata intuicionista difuso sobre Σ y $\mathcal{L}$ es una séptupla $\mathcal{A} = (Q, \xi, \gamma, \alpha, \beta, \chi, \rho)$ donde:

- Q denota el conjunto no vacío de estados.
- $\alpha, \beta: Q \to L$ denotan respectivamente, las funciones de pertenencia y no pertenencia de los estados iniciales difusos intuicionistas. Así, el conjunto de estados iniciales intuicionistas difusos es representado por $\big(\tilde{\alpha}, \tilde{\beta}\big) = \big\{\big(q, \alpha(q), \beta(q)\big) | q \in Q\big\}$.
- $\chi, \rho: Q \to L$ denotan respectivamente, las funciones de pertenencia y no pertenencia de los estados finales difusos intuicionistas. Así, el conjunto de

estados finales intuicionistas difusos es representado por $(\tilde{\chi}, \tilde{\rho}) = \{(q, \chi(q), \rho(q))|q \in Q\}$.

- $\xi, \gamma: Q \times (\Sigma \cup \{\lambda\}) \times Q \to L$ representan las funciones de pertenencia y no pertenencia de las transiciones difusas intuicionistas, respectivamente.

Sea $IF(Q)$ el posible conjunto difuso intuicionista en Q. Las funciones de pertenencia y no pertenencia$\tilde{\xi}, \tilde{\gamma}: IF(Q) \times \Sigma \to IF(Q)$ se definen:

$$\tilde{\xi}(P', x) = \left\{(p, \xi)|\xi = \max_{q \in Q}\left(\min\left(\xi(q, x, p), \xi_{P'}(q)\right)\right), p \in Q\right\}$$

$$\tilde{\gamma}(P', x) = \left\{(p, \gamma)|\gamma = \min_{q \in Q}\left(\max\left(\gamma(q, x, p), \gamma_{P'}(q)\right)\right), p \in Q\right\}$$

Para $P' \in IF(Q), x \in \Sigma$.

Se define las funciones de pertenencia y no pertenencia $\xi^*, \gamma^*: IF(Q) \times \Sigma^* \to IF(Q)$ como:

- $\xi^*(P', \lambda) = \xi^{\lambda}(P')$, y $\gamma^*(P', \lambda) = \gamma^{\lambda}(P')$ para $P' \in IF(Q)$.
- $\xi^*(P', ax) = \xi^{\lambda}\left(\tilde{\xi}(\xi^*(P', a), x)\right)$, y $\gamma^*(P', ax) = \gamma^{\lambda}\left(\tilde{\gamma}(\gamma^*(P', a), x)\right)$ para $P' \in IF(Q), x \in \Sigma, a \in \Sigma^*$.

Donde las funciones $\xi^{\lambda}, \gamma^{\lambda}: IF(Q) \to IF(Q)$ representan versiones difusas de la clausura difusa de la cadena vacía $(\lambda - clausura)$ para un autómata finito difuso no determinístico. Dichas funciones son calculadas:

$$\xi^{\lambda}(P') = \left\{(p, \xi)|\xi = \max\left(\xi_{P'}(p)\left(\max_{q \in Q}\left(\min\left(\xi_{E'(q)}(p), \xi_{P'}(q)\right)\right)\right)\right), p \in Q\right\}$$

$$\gamma^{\lambda}(P') = \left\{(p,\gamma)|\gamma = \min\left(\gamma_{P'}(p)\left(\min_{q\in Q}\left(\max\left(\gamma_{F'(q)}(p),\gamma_{P'}(q)\right)\right)\right)\right), p \in Q\right\}$$

Para $P' \in IF(Q)$, donde $E'(q)$ y $F'(q)$ pueden ser obtenidos mediante:

- Sea $\left(q_{k_0}, q_{k_1}, \ldots, q_{k_l}\right)$ una secuencia de estados $(l > 0)$arbitrarios y finito de Q tal que $\forall i, j, 0 \leq i, j \leq l, i \neq j$. Así, $q_{k_i} \neq q_{k_j}$.
- Sean $E'(q)$ y $F'(q)$ conjuntos intuicionistas difusos en Q representando el conjunto de estados alcanzables de q usando sólo transiciones por cadena vacía:

$$E'(q) = \left\{(p,\xi)|\xi = \max_{\left(q_{k_0},\ldots,q_{k_l}\right),q_{k_0}=q,q_{k_l}=p}\left(\min_{0<i\leq l}\xi\left(q_{k_{i-1}},\lambda,q_{k_i}\right)\right), p \in Q\right\}$$

$$F'(q) = \left\{(p,\gamma)|\gamma = \min_{\left(q_{k_0},\ldots,q_{k_l}\right),q_{k_0}=q,q_{k_l}=p}\left(\max_{0<i\leq l}\gamma\left(q_{k_{i-1}},\lambda,q_{k_i}\right)\right), p \in Q\right\}$$

Se define $\tilde{L}(\mathcal{A})$ o $[\![\mathcal{A}]\!]$ como el lenguaje aceptado por el autómata difuso intuicionista $\mathcal{A}$, el cual es un conjunto difuso en Σ^* definido como:

$$[\![\mathcal{A}]\!] = \tilde{L}(\mathcal{A}) = \left\{(\psi,\xi,\gamma)|\xi = \max_{q\in Q}\left(\min\left(\xi_{\xi^*(\tilde{\alpha},\psi)}(q)\right),\chi(q)\right),\gamma\right.$$
$$\left. = \min_{q\in Q}\left(\max\left(\gamma_{\gamma^*(\tilde{\beta},\psi)}(q)\right),\rho(q)\right), \psi \in \Sigma^*\right\}$$

5. MARCO APLICATIVO

Para el presente capítulo es necesaria la lectura de los capítulos anteriores por los conceptos que se manejan los cuales son nombrados y fueron introducidos y demostrados anteriormente. El capítulo se centra el desarrollo de los algoritmos junto a su respectivo análisis de complejidad y espacio utilizado, para la elaboración de definiciones y algoritmos se tomó como fuente de recopilación los trabajos (Ukkonen, 1985), (Masek & Paterson, 1978), (Wu, Manber, & Myers, 1992), (Navarro G., 1997), (Myers, 1999), (Navarro G., 2001), (Ravi, Choubey, & Tripati, 2014), (Echabone, Garitagoitia, & González de Mendívil), (Astrain, Garitagoitia, Gonzáles de Mendivíl, Villadangos, & Fariña, 2004) y (Andrejková, Almarimi, & Mahmoud, 2013).

5.1. ALGORITMO UKKONEN

Dado el patrón $P = a_1 \dots a_n$ y la máxima distancia de edición permitida k, el algoritmo construye un autómata finito determinístico $M_P = (Q, \Sigma, \delta, q_0, F)$ que recorre el texto $T = b_1 \dots b_n$ carácter a carácter y al llegar a un estado final que pertenece a F sí y solo si T contiene una subcadena con distancia de edición menor o igual a k, para lo cual cada estado del autómata corresponde a una posible columna de la matriz $C_{i,j}$.

Para presentar el algoritmo de pre cómputo de la figura 9, se utiliza $S = (S_0, \dots, S_m)$ y $S' = (S'_0, \dots, S'_m)$ como estados arbitrarios, es decir, columnas arbitrarias de $C_{i,j}$. El índex de un estado S es denotado por $INDEX(S)$. En particular, para el estado inicial q_0 se tiene $S = (0,1, \dots, m)$ e $INDEX\big((0,1, \dots, m)\big) = 0$. El alfabeto el cual aún es denotado

Algorithm Ukkonen's Algorithm

```
1: procedure UKKONEN
2:     S = (0, 1, ..., m)
3:     INDEX (S) = 0
4:     i = 0
5:     NEW = {S}
6:     STATES = {S}
7:     if k ⩾ m then
8:         F = {0}
9:     while NEW ≠ ∅ do
10:        S = NEW.pop()
11:        for each b ∈ Σ do
12:            S′ = Next − Column (S, b)
13:            if S′ ∉ STATES then
14:                NEW.push (S′)
15:                STATES.add (S′)
16:                i = i + 1
17:                INDEX (S′) = i
18:                if S′_m ⩽ k then
19:                    F = F ∪ INDEX (S′)
20:            δ (INDEX (S) , b) = INDEX (S′)
```

Figura 9 Algoritmo de Ukkonen para la construcción de un AFD.

por Σ y contiene los símbolos de P más el símbolo þ para denotar los símbolos del texto que no estén en el patrón.

Suponiendo que los estados S y S' son tal que siempre que $S_i \neq S'_i$ y $S_i, S'_i > k$, entonces ambos estados son equivalentes en el autómata, lo cual es que ambos estados aceptan un lenguaje idéntico. Esto ya que los valores de $C_{i,j}$ en cualquier secuencia del grafo de dependencia forma una secuencia no decreciente, es decir, las secuencias de edición dentro de la matriz siguen caminos no decrecientes.

Algorithm Ukkonen's Algorithm

```
1: procedure NEXT-COLUMN(S, b)
2:     S'_0 = 0
3:     for i ← 1, m do
4:         if x_{i-1} = b then
5:             S'_i = S_{i-1}
6:         else
7:             S'_i = 1 + min(S_{i-1}, S'_{i-1}, S_i)
8:         if S'_i > k then
9:             S'_i = k + 1
       return (S'_0, ..., S'_m)
```

Figura 10 Procedimiento de cálculo del siguiente estado, se envían como parámetro el estado S y el carácter b.

Por lo tanto, si para algún $C_{i,j} > k$, dicha posición no pertenece a una secuencia de edición que lleve a una ocurrencia. Es decir, los valores de las columnas (estados) S que sean mayores a k no tienen influencia en los caminos que llegan a una ocurrencia, por lo cual se realiza la modificación de los estados dentro el procedimiento $next - column$ en la línea 9, esta observación reduce el número de estados del autómata.

Tomando en cuenta el resultado del teorema 2.1 donde los estados se convierten en vectores compuestos por los valores $\{-1,0,1\}$, lo cual sugiere que una estructura de datos conveniente para representar los estados del autómata es un árbol ternario, el cual estaría compuesto por nodos y cada nodo tiene hasta tres nodos descendientes etiquetados por $-1,0$ o 1. El conjunto NEW puede ser representado como una cola de punteros a las hojas apropiadas del árbol de estados.

El espacio requerido para el árbol es $O(mt)$, donde t denota el número total de estados y cada revisión de pertenencia e inserción de nuevos estados se realiza en $O(m)$. La cola

NEW requiere un espacio de $O(t)$ y cada inserción y eliminación dentro de la cola tiene un tiempo constante. Cada llamada al procedimiento $next - column$ toma un tiempo de $O(m)$, por lo cual la complejidad del algoritmo es $O(m * |\Sigma| * t)$. Además, el espacio requerido para almacenar la función de transición δ es $O(|\Sigma| * t)$, por lo cual es espacio requerido es $O\big((|\Sigma| + m) * t\big)$.

Finalmente, para estimar t tomamos primero la idea de la representación del conjunto de estados en forma de árbol del cual es fácil deducir que se tiene a lo más $k = O(3^m)$. Para mejorar esta estimación, añadiendo la modificación donde las posiciones de la matriz mayores a k son representadas por $k + 1$. Se particiona todas las columnas S en clases disjuntas, la clase m contiene todas las columnas S tal que $S_m \leq k$. La clase i, $i < m$, contiene todas las columnas que tienen $S_i = k$ y $S_j = k + 1$ para $j > i$.

Como ya fue mencionado si la matriz C contiene una columna S en la clase m entonces el texto contiene una subcadena p', la cual tiene una distancia de edición a lo más k, tras contar el número de diferentes secuencias de edición de longitud hasta k, la clase m contiene a lo sumo $(|\Sigma| + 1)^k * (2n + 1)^k$ elementos. De igual forma, la clase i es limitada por el número de cadenas diferentes cuya distancia de edición es k con lo que se obtiene $(|\Sigma| + 1)^k * (2i + 1)^k$. Por lo que se concluye que $k = O(2^k * |\Sigma|^k * m^{k+1})$, combinando ambos casos:

$$k = \min(3^m, 2^k * |\Sigma|^k * m^{k+1})$$

5.2. ALGORITMO MASEK Y PATERSON

La clausura transitiva de un grafo dirigido con n nodos es fácilmente calculado con una matriz $n \times n$ con $O(n^2)$ operaciones, en (Arlazarov, Dinic, Kronrod, & Faradzev, 1970) se prueba que, si la matriz es dividida en submatrices con un número menor de columnas y pre calculando estas submatrices, el problema es resuelto con $O(n^2/\log n)$ operaciones, este algoritmo es comúnmente conocido como "*the four Russians' algorithm*".

El *four Russians' algorithm* realiza el cómputo más rápido dividiendo el problema en varios cómputos más pequeños. Realiza primero el cálculo de todos los cómputos pequeños, después los une para así obtener el cómputo del problema original. Siguiendo esta idea, primero se calculará todas las posibles submatrices $(t+1) \times (t+1)$ que puedan llegar a ocurrir en la matriz C para algún parámetro t; así estas submatrices combinadas dan como resultado la matriz original junto al cálculo de la distancia de edición.

Se define la submatriz (i, j, t) de la matriz de edición C como la matriz $t+1 \times t+1$ cuyo vértice superior izquierdo es $C_{i,j}$. Los valores en la submatriz (i, j, t) son determinados en base al vector inicial $C_{i,j}, C_{i,j+1}, \dots, C_{i,j+t}$ y $C_{i,j}, C_{i+1,j}, \dots, C_{i+t,j}$ junto a las subcadenas $a_{i+1 \dots i+t}$ y $b_{i+1 \dots i+t}$ del texto T y el patrón P. La primera parte del algoritmo construye todas las submatrices (i, j, t) que pueden ocurrir, y almacena los vectores finales de cada submatriz $C_{i+t,j+1} \dots C_{i+t,j+t}$ y $C_{i+1,j+t} \dots C_{i+t,j+t}$ para ser usadas posteriormente.

Para calcular los vectores finales para cada posible submatriz, primero es necesario enumerar las submatrices posibles. Como se tiene un alfabeto finito es posible listar todas las cadenas de longitud t, por otro lado, listar todos los posibles vectores iniciales podría tomar mucho tiempo, ya que los valores en la matriz tienden a crecer, no resulta óptimo tomarlos de esa manera.

Tomando como apoyo los resultados del teorema 2.1 y teorema 2.2, se toma los vectores d_j y d'_j llamándoles vectores *step* y como por el teorema 2.2 se tiene que sólo existen un número limitado de diferencias entre los valores de la matriz, operar con los valores d_j y d'_j representa una reducción significativa en los posibles vectores iniciales a considerar.

Algorithm Masek & Paterson's Algorithm

```
1:  procedure MASEK & PATERSON
2:      for each C, D ∈ Σ^t & R, S do
3:          for i ← 1, t do
4:              T_{i,0} = R_i
5:              U_{0,i} = S_i
6:          for i ← 1, t do
7:              for j ← 1, t do
8:                  if C_i = D_j then
9:                      val = 0
10:                 else
11:                     val = 1
12:                 T_{i,j} = min(val − U_{i−1,j}, 1, 1 + T_{i,j−1} − U_{i−1,j})
13:                 U_{i,j} = min(val − T_{i,j−1}, 1 + U_{i−1,j} − T_{i,j−1}, 1)
14:         R' = (T_{1,m}, ..., T_{m,m})
15:         S' = (U_{m,1}, ..., U_{m,m})
16:         Store(R', S', R, S, C, D)
```

Figura 11 Algoritmo Masek y Paterson para el cálculo de las submatrices, R y S son vectores de tamaño t y la función Store almacena los vectores R, S, las cadenas C, D y los vectores de respuesta R', S'.

Así cada submatriz (i, j, t) se determina por el valor inicial $C_{i,j}$, los vectores iniciales *step* $d_{j+1}[i], \dots, d_{j+t}[i]$ y $d_j[i+1], \dots, d_j[i+t]$ y las cadenas $a_{i+1\dots i+t}$ y $b_{j+1\dots j+t}$. Después se calcula los vectores *step* finales, y para enumerar todas las posibles submatrices se enumerará todos los posibles pares de cadenas y vectores de longitud t.

En la figura 11 se presenta el algoritmo que realiza el cálculo de todas las submatrices, se asume que el alfabeto esta ordenado y las cadenas y vectores tienen un orden lexicográfico. Así para cualquier par de cadenas C, D y par de vectores *step* R, S el algoritmo calcula una submatriz de diferencias de acuerdo con la fórmula del teorema 2.1.

Existen dos tipos de vectores a considerar, los horizontales y verticales, donde se separa el resultado del algoritmo estableciendo a T como la matriz que contiene las diferencias verticales y U como la matriz que contiene las diferencias horizontales, concluyendo guardando los vectores R', S' que representan los vectores *step* finales.

La última parte del algoritmo, junta las submatrices generadas anteriormente para así formar la matriz *step* ver figura. Para calcular la distancia de edición se suman los valores de los vectores *step* finales horizontales en la última fila de la matriz C. En la figura 12 se ilustra el recorrido del texto por las submatrices.

La primera parte del algoritmo tiene una complejidad de $O(t^2 \log t)$ para calcular y almacenar cada submatriz, las $O(t^2)$ operaciones para calcular cada submatriz requiere un tiempo de $O(\log t)$ para almacenarla. Al saber el número de vectores almacenados en

Algorithm Masek & Paterson's Algorithm

1: **procedure**
2: $r = |P|/t$
3: $s = |T|/t$
4: **for** $i \leftarrow 1, r$ **do**
5: $P_{i,0} = (1, 1, ..., 1)$
6: **for** $j \leftarrow 1, s$ **do**
7: $Q_{0,j} = (1, 1, ..., 1)$
8: **for** $i \leftarrow 1, r$ **do**
9: **for** $j \leftarrow 1, s$ **do**
10: $(P_{i,j}, Q_{i,j}) = get\left(P_{i,j-1}, Q_{i-1,j}, P_{(i-1)t+1...it}, T_{(j-1)t+1...jt}\right)$
11: $cost = m$
12: **for** $j \leftarrow 1, s$ **do**
13: $cost = cost + sum\left(Q_{r,j}\right)$

Figura 12 Recorrido del texto por las submatrices construidas, los vectores (1,1, … ,1) son de dimensión t, el procedimiento get devuelve los vectores R', S' en base a los parámetros enviados y procedimiento sum devuelve la suma del vector almacenado en Q.

la primera etapa del algoritmo, es posible recuperar los valores en un tiempo $O(t + \log|T|)$ indexándolos en la memoria RAM. Se usa un tiempo $O(\log|T|)$ para leer los vectores P, Q más un tiempo de $O(t)$ para almacenarlos. Por lo cual se requiere $O(|T| * |P|/t^2)$ para recuperar y asignar los vectores de longitud t y $O(|T| * |P| * (t + \log|T|)/t^2)$ diferencias básicas. Si se escoge $t = \lfloor \log_k|T| \rfloor$ el algoritmo tiene una complejidad $O(|T| * |P|/t)$.

5.3. ALGORITMO WU, MANBER, MAYERS

El algoritmo es diseñado para extender el enfoque de la programación dinámica a expresiones limitadas, más el algoritmo es aplicable a la búsqueda de cadenas aproximadas con mínimas alteraciones.

Algorithm Sellers with diferences Algorithm

```
1:  procedure SELLERS
2:      for i ← 0, m do
3:          D_{i,0} = 1
4:      for j ← 1, n do
5:          c_{0,j} = 0
6:          for i ← 1, m do
7:              if x_i = y_j then
8:                  val = 0
9:              else
10:                 val = 1
11:             D_{i,j} = min(1, val − c_{i−1,j}, D_{i,j−1} − c_{i−1,j} + 1)
12:             c_{i,j} = c_{i−1,j} + D_{i,j} − D_{i,j−1}
```

Figura 13 Algoritmo de Sellers modificado con los vectores diferencia del teorema 2.1, el algoritmo puede ser modificado para almacenar una simple columna de la matriz D.

Para el desarrollo de un algoritmo sub cuadrático se analiza el autómata generado por el algoritmo de Ukkonen, el cual consiste en dos fases, la primera en la se genera el autómata a partir del patrón y en después se recorre el texto con el autómata. El tiempo de pre procesamiento llega a ser exponencial en $|P|$, pero es lineal en el recorrido del texto.

Para empezar, el resultado del teorema 2.1 se aplica utilizando las diferencias d_j directamente en la recurrencia del algoritmo de Sellers, y se expresa la columna d_j en términos de la columna d_{j-1} junto a los valores $c_{i,j}$ cómo se muestra en la demostración del teorema 2.1. En la figura 13 se muestra el algoritmo de Sellers modificado.

El enfoque utilizado para el presente algoritmo es particionar cada columna o estado del autómata en regiones y construir un autómata más pequeño para cada región, y se utiliza este autómata más pequeño para simular el funcionamiento del original. Cada

vector de diferencias o estado es particionado en sub vectores, llamados regiones, de tamaño r.

Sea la región h el sub vector $d_j[(h-1)r+1, (h-1)r+2, \dots, hr]$ del vector $d_j[1 \dots m]$, donde $1 \leq h \leq m/r$, denotado por $d_j\langle h\rangle$, es importante resaltar que cada región puede ser uno de los 3^r posibles vectores de r diferencias, a los cuales se llamarán estados de ahora en adelante y es posible codificar cada una de estas regiones en un entero entre 0 y $3^r - 1$.

Estudiando el algoritmo modificado de Sellers se llega a las siguientes observaciones, el estado $d_j\langle h\rangle$ depende de tres factores, el estado $d_{j-1}\langle h\rangle$, el valor de $c_{(h-1)r,j}$ y los valores $s_{(h-1)r+1,j}, s_{(h-1)r+2,j}, \dots, S_{hr,j}$ llamado vector característico de la región h con la entrada b_j, es decir, $d_j\langle h\rangle$ depende de dos vectores de tamaño r y $c_{(h-1)r,j}$ llamado a partir de ahora *carry in*.

Es necesario el cálculo del *carry out* $c_{hr,j}$ como entrada para la región $h+1$, tanto las regiones como los vectores característicos son codificados por un entero, donde dado un vector característico cv este es codificado como un entero entre 0 y $2^r - 1$ y el valor del *carry in* $c \in \{-1,0,1\}$, así el autómata a partir de un estado s se mueve a un único estado siguiente y determina un único *carry out*.

Dicho autómata es modelado en dos tablas $GOTO[cv, s, c]$ y $COUT[cv, s, c]$ que tienen de parámetros valores enteros y retornan el siguiente estado y el *carry out* como valores enteros. Además, las tablas tienen exactamente $3m$ posibles índex si $r = \log_6 m$,

es decir, $2^{\log_6 m} * 3^{\log_6 m} * 3 = 3m$ y dichas tablas pueden ser pre calculadas y almacenadas en un tiempo $O(n)$.

Usando la tabla de transición pre calculada, es posible calcular $r = O(\log m)$ valor de la matriz de programación dinámica en un tiempo constante si la entrada para las transiciones puede ser obtenido en un tiempo constante. El estado o región previo junto al *carry in* son representados como enteros, ahora interesa una representación para los vectores característicos.

Cada vector característico consiste en r valores binarios y calcularlos y codificarlos en cada lectura de región y símbolo del preprocesamiento llevaría un tiempo de $O(r)$ lo cual incrementaría la complejidad del preprocesamiento y del algoritmo. Se soluciona este inconveniente pre calculando para cada región el entero que representa el vector característico que resulta al evaluar cada uno de los diferentes símbolos del alfabeto, lo cual lleva a construir una nueva tabla cv de dimensiones $n/r \times \Sigma$, una vez construida dicha tabla, el vector característico para cualquier región con cualquier carácter de entrada puede ser obtenido en un tiempo constante.

Posteriormente a la construcción de las tablas, el algoritmo encuentra las coincidencias aproximadas como es detallado en la Figura 14. La invariante principal es que antes de la ejecución del bucle, $D[h] = D_{j-1}[h]$ para toda $1 \leq h \leq m/r$ y $e = C_{m,j-1}$. Inicialmente, en las líneas 2 a la 4, el estado de cada región h es asignada con el entero que representa el estado $(1,1,1,\ldots,1)$ y $e = m$. El bucle de la línea 5 realiza la evaluación del texto carácter a carácter, al avanzar la invariante con el carácter b_j, se calcula la transición de

Algorithm Wu, Manber, Myers' Algorithm

```
1:  procedure
2:      for h ← 1, m/r do
3:          D[h] = (1, 1, ..., 1)
4:      e = m
5:      for j ← 1, n do
6:          c = 0
7:          for h ← 1, m/r do
8:              D[h] = GOTO(CV(h, y_j), D[h], c)
9:              c = COUT(CV(h, y_j), D[h], c)
10:         e = e + c
11:         if e ⩽ k then
12:             Ocurrencia en j
```

Figura 14 Algoritmo de Wu, Manber y Myers para la búsqueda de un patrón en un tiempo $O(mn/\log n)$.

cada región h, primero se recupera el entero que representa al vector característico para b_j correspondiente a la región h evaluada para después junto al valor de la variable *carry in* y el estado actual se realice la búsqueda en las tablas $GOTO$ y $COUT$ y obtener el nuevo estado y *carry out*. En la línea 8 se actualiza el valor de e con el último *carry out* de la columna ya que $C_{m,j} = C_{m,j-1} + c_{m,j}$ y así en el caso de que $e \leq k$ se reporta correctamente una ocurrencia.

La tabla de los vectores característicos requiere un espacio de $O(|\Sigma|m/\log n)$ ya que para cada símbolo del alfabeto se requieren $m/\log n$ enteros y se requiere un tiempo de $O(m)$ para pre calcular los vectores característicos para un símbolo por lo cual se tiene una complejidad total de $O(m|\Sigma|)$.

En la construcción de las tablas de transición se tiene una complejidad de $O(n)$ y ocupan un espacio de $O(n)$. El tiempo total para escanear un carácter es $O(m/\log n)$ ya

que cada región puede ser procesada en un tiempo constante y solo existen $m/\log n$ regiones en cada columna. Por tanto, el tiempo total de ejecución es de $O(nm/\log n)$ y el espacio requerido es de $O(n + m|\Sigma|/\log n)$.

5.4. ALGORITMO WU, MANBER, MAYERS $O(kn/log\ n)$

El algoritmo antes propuesto es mejorado aplicando el criterio descrito en (Ukkonen, 1985) explicado en la sección 5.1 del presente documento. Se ha probado que el algoritmo propuesto por Ukkonen tiene un tiempo esperado de $O(kn)$ en distribuciones adecuadas en la entrada (Chang & Lampe, 1992).

Ukkonen mejora el tiempo esperado del algoritmo de programación dinámica calculando solo una porción de la matriz, dicho enfoque es mantener para cada columna j el menor índex x_j tal que $C_{i,j} > k, \forall i > x_j$. Si se conoce el valor de x_j ya no es necesario el calculo de aquellas posiciones en la columna ya que no llevarían a una ocurrencia.

Para calcular las regiones relevantes en bloques de tamaño r se requiere mantener la última región w_j en la columna j que contiene una entrada o posición igual a k. Para esto es necesario más información acerca de los estados, sea un estado $s = [d_1, d_2, \dots, d_r]$ se define la suma de bloque de s como $d_1 + d_2 + \cdots + d_r$ y la suma de sufijo máximo de s como $\max(d_{k+1} + d_{k+2} + \cdots + d_r | k \in [1, r])$. Sean $BKS[s]$ y $MSS[s]$ dos tablas que dado un estado s retornan su suma de bloque y la suma de su sufijo máximo respectivamente. Ciertamente el cálculo de estas tablas adicionales durante la construcción del autómata no incrementa el tiempo de procesamiento ya que las tablas tienen una dimensión 3^r.

Algorithm Wu, Manber, Myers' Algorithm

```
1:  procedure
2:      w = ⌈k/r⌉
3:      for h ← 1, w do
4:          D[h] = (1, 1, ..., 1)
5:      e = w * r
6:      for j ← 1, n do
7:          if w < m/r then
8:              e = e + r
9:              w = w + 1
10:             D[w] = (1, 1, ..., 1)
11:         c = 0
12:         for h ← 1, w do
13:             D[h] = GOTO(CV(h, y_j), D[h], c)
14:             c = COUT(CV(h, y_j), D[h], c)
15:         e = e + c
16:         while w > 0 ∧ e − MSS[w] > k do
17:             e = e − BKS[D[w]]
18:             w = w − 1
19:         if w = m/r ∧ e ⩽ k then
20:             Ocurrencia en j
```

Figura 15 Algoritmo de Wu, Manber y Myers para la búsqueda de un patrón en un tiempo $O(nk/\log n)$.

Se presenta el algoritmo modificado en la figura 15, el algoritmo comienza calculando las regiones relevantes en la primera columna, es decir, hasta la región que contiene la última posición activa y asigna a e el valor de sumatoria de todas las suma bloque de las regiones relevantes.

Suponiendo que se lee b_j, en exacta analogía al algoritmo de Ukkonen en la columna se calcula todas las regiones hasta $\max(w_{j-1}+1, m/r)$ y se asigna a w_j la región más larga que contenga una entrada igual a k. Para ello, se asigna a la región $w_{j-1}+1$ el valor de $(1,1,\ldots,1)$ y se añade r a e si $w_{j-1} < m/r$. En las líneas de la 11 a la 15, las w_{j-1}

regiones de la columna j son calculadas a partir de la columna $j-1$ de la manera usual sumando el último *carry out* a e.

Las líneas 16 a la 18 sigue el siguiente criterio, si e menos la suma del sufijo máximo de la región w, que es igual a $w_{j-1}+1$ en este punto, es mayor a k entonces la región y no contiene una posición igual a k por lo que se decrementa e por la suma de bloque de la región w y se resta uno a w hasta que se cumpla la condición. Concluyendo, se pregunta si w se encuentra en la última región de la columna j y si $e \leq k$ para reportar una coincidencia.

En esencia el algoritmo realiza el mismo trabajo que el algoritmo de Ukkonen salvo que este realiza los cálculos en bloques de tamaño r, y al elegir un $r = \log_6 n$ lleva a un algoritmo que toma un espacio $O(n)$ para sus tablas, además como se espera que la última posición activa sea alcanzada en un tiempo $O(k)$ se espera que las regiones tomen $O(k/\log n)$ por lo cual la búsqueda tendría un tiempo aproximado de $O(nk/\log n)$.

Para mejorar el desempeño del algoritmo en la práctica se realizan las siguientes modificaciones al algoritmo de recorrido de la figura 15, las tablas $GOTO$ y $COUT$ las cuales tenían como parámetro de entrada tres enteros, el primero en el rango 0 a $2^r - 1$ representando el vector característico, el segundo en el rango 0 a $3^r - 1$ representando el estado actual de una región y el *carry in* un entero en el rango -1 a 1, son modificados transformando cada entero que representa el estado s en $3s+1$ y cada entero que representa el vector característico cv en $cv * 3^{r+1}$.

Algorithm Wu, Manber, Myers' Algorithm

```
1:  procedure
2:      Y = ⌈m/r⌉
3:      I = 3^{r+1} − 2
4:      K = k + r
5:      w = ⌈k/r⌉
6:      for h ← 1, w do
7:          D[h] = I
8:      e = w * r
9:      for j ← 1, n do
10:         c = 0
11:         for h ← 1, w do
12:             i = CV(h, y_j) + D[h] + c
13:             D[h] = GOTO(i)
14:             c = COUT(i)
15:         if e = k ∧ w < Y then
16:             i = CV(w + 1, y_j) + I + c
17:             w = w + 1
18:             D[w] = GOTO(i)
19:             e = e + r + COUT(i)
20:         else
21:             e = e + c
22:             while e > K do
23:                 e = e − BKS[D[w]]
24:                 w = w − 1
25:         if w = Y ∧ e ⩽ k then
26:             Ocurrencia en j
```

Figura 16 Algoritmo de Wu, Manber y Myers para la búsqueda de un patrón.

Esta modificación causa la modificación de los parámetros que reciben las tablas $GOTO$ y $COUT$ y además como los estados ahora tienen un rango de 1 a $3^{r+1} - 2$, el dominio de la tabla BKS también crece en un factor de 3. Con esa modificación las tablas tienen la forma $GOTO[cv + s + c]$ y $COUT[cv + s + c]$ con dominio de 0 a $3 * 6^r - 1$.

La siguiente modificación podría reducir cálculos innecesarios, en lugar de calcular las regiones 1 a la $w_{j-1} + 1$ en la columna j, se calcula solo hasta la w_{j-1} inicialmente, si

la suma de la columna $j-1$ en e es igual a k y w_{j-1} no es la última región entonces es necesario calcular $w_{j-1}+1$, dicha condición es verificada en la línea de la figura 16.

Finalmente, no se utilizará la tabla MSS para verificar si el valor de una región activa final es k o menos en su lugar se pregunta si e es mayor a $k+r$ y aunque sea menos preciso es más eficiente en tiempo.

5.5. ALGORITMO NAVARRO

Se define un autómata finito determinístico o AFD como un conjunto de estados conectados por transiciones, las transiciones llevan de un estado a otro en base a los símbolos del alfabeto y existe exactamente una transición en cada estado por cada uno de los símbolos de alfabeto. Un autómata finito determinístico parcial es un AFD donde algunas transiciones aún no han sido determinadas.

Navarro propone el uso de un AFD parcial para el problema del emparejamiento aproximado de cadenas, comenzando con un único estado sin transición alguna correspondiente a la configuración inicial de la matriz C, se recorre el texto exactamente como si se tuviera el AFD completo, con la diferencia que si se requiere una transición inexistente se la calcula inmediatamente, es decir, se toma el estado actual y se realiza el algoritmo clásico $O(m)$ para determinar el siguiente estado.

La ventaja de dicha construcción es que, a pesar de que el AFD para un determinado patrón puede ser muy grande, solo se requiere una pequeña porción de los estados los cuales son visitados conforme se recorre el texto. La desventaja es que una vez generado el AFD completo la configuración que corresponde a cada estado no necesita ser

Algorithm Navarro's Algorithm

```
1:  procedure NAVARRO
2:      iniState = C_0
3:      Automata.Add(iniState)
4:      state = iniState
5:      for i ← 0, n do
6:          nstate = transicion(state, y_i)
7:          if nstate = unknown then
8:              nstate = perform − step(state, y_i)
9:              if nstate ∉ STATES then
10:                 Automata.Add(nstate)
11:             transicion(state, y_i) = nstate
12:         state = nstate
13:         if state ∈ F then
14:             Ocurrencia en j
```

Figura 17 Algoritmo de Navarro que construye un autómata determinístico parcial para la búsqueda de un patrón en un texto.

almacenada, mientras que en un AFD parcial es necesario su almacenamiento para generar nuevos estados y transiciones. Este espacio extra que se acerca a un 25% extra por estado generado demuestra ser menor al número de estados no visitados en un AFD completo.

Los estados son representados como en los algoritmos clásicos donde se representa las columnas C_i como vectores de tamaño m, y para mejorar la complejidad del algoritmo se hace uso de la idea propuesta por Ukkonen, se trabaja solo con las posiciones activas por lo cual se tienen en promedio $O(k)$ valores activos.

Una vez que *uconf* es calculado no es necesario saber si corresponde a un estado ya existente en el autómata, por lo tanto, se realiza la búsqueda en el conjunto de configuraciones conocidas la cual puede ser realizada en un tiempo proporcional a la longitud de la configuración buscada.

Sea s y t el número total de estados y transiciones respectivamente, como es mostrado en Ukkonen y en la sección 5.1 el número de estados tiene un límite superior en $s = O(\min(3^m, 2^k * |\Sigma|^k * m^{k+1}))$, dado que se puede asignar Σ a $O(m)$ símbolos diferentes, se reemplaza $|\Sigma|$ por $\min(|\Sigma|, m)$, lo que se muestra en (Melichar, 1996) donde se describe un límite alternativo $s = O((k+2)^{m-k}(k+1)!)$.

El número de transiciones es entonces $t = O(s * \min(|\Sigma|, m))$. En el presente algoritmo se tiene que el número de estados puede ser $s' = O(\min(s, n))$ y las transiciones $t' = O(\min(t, n))$ ya que cada carácter del texto puede crear a lo más un estado y transición. Así, el espacio requerido por el algoritmo de Navarro es $O(s' * \min(m, |\Sigma|) + t')$. El algoritmo es lineal en su tiempo de ejecución a excepción de la generación de estados y nuevas transiciones, costando cada una $O(m)$, lo que lleva a un costo total para el algoritmo de $O(n + s' \min(m, |\Sigma|) + t'm)$.

Navarro propone un modelo probabilístico para calcular el número de estados promedio generados después de leer n símbolos aleatorios, asumiendo que cada nuevo carácter produce una transición aleatoria. La probabilidad de que dada una transición no sea generada en una posición particular del testo es $(1 - 1/t)$, por lo cual, el número promedio de estados generados después de leer n caracteres del texto es:

$$t' = t\left(1 - \left(1 - \frac{1}{t}\right)^n\right) = t\left(1 - e^{-\frac{n}{t}}\right) + O\left(\frac{1}{t}\right)$$

Y por lo cual la complejidad promedio es $O\left(n + mt\left(1 - e^{-n/t}\right)\right)$. Es importante notar que $\left(1 - e^{-n/t}\right)$ es el factor por el cual construir un AFD parcial es más eficiente que un AFD completo.

5.6. ALGORITMO RAVI, CHOUBEY, TRIPATI

Varios métodos se han propuesto para el tratamiento del emparejamiento aproximado de cadenas y problemas similares en los artículos (Echabone, Garitagoitia, & González de Mendívil), (Reina, González de Mendivil, & Garitagoitia, 1992), entre otros, dichos artículos tratan el problema de manera similar y tan solo difieren en el tipo de autómata difuso aplicado para el reconocimiento y en la estructura algebraica usada para la construcción del autómata.

Las transiciones son obtenidas principalmente por la fórmula:

$$\tilde{\vartheta}\left(\tilde{P}, \tilde{y}\right) = \left\{(p, \vartheta) | \vartheta = \oplus_{q \in Q}\left(\oplus_{x \in \Sigma}\left(\vartheta(q, p, x) \otimes \vartheta_{\tilde{y}}(x)\right) \otimes \vartheta_{\tilde{P}}(q)\right), p \in Q\right\}$$

Donde $\tilde{P} \in \mathcal{F}(Q)$, $\tilde{y} \in \mathcal{F}(\Sigma)$, es decir, pertenecen al conjunto de todos los posibles conjuntos difusos sobre los estados y el alfabeto respectivamente. Los operadores $\otimes$ y $\oplus$ representan la multiplicación o norma – t y la conorma – t de la estructura algebraica utilizada, la figura 22 ilustra un algoritmo general para la construcción de un autómata difuso deformado.

En el artículo (Ravi, Choubey, & Tripati, 2014) se propone un nuevo método para el problema del emparejamiento aproximado de cadenas, el uso de conjuntos intuicionistas

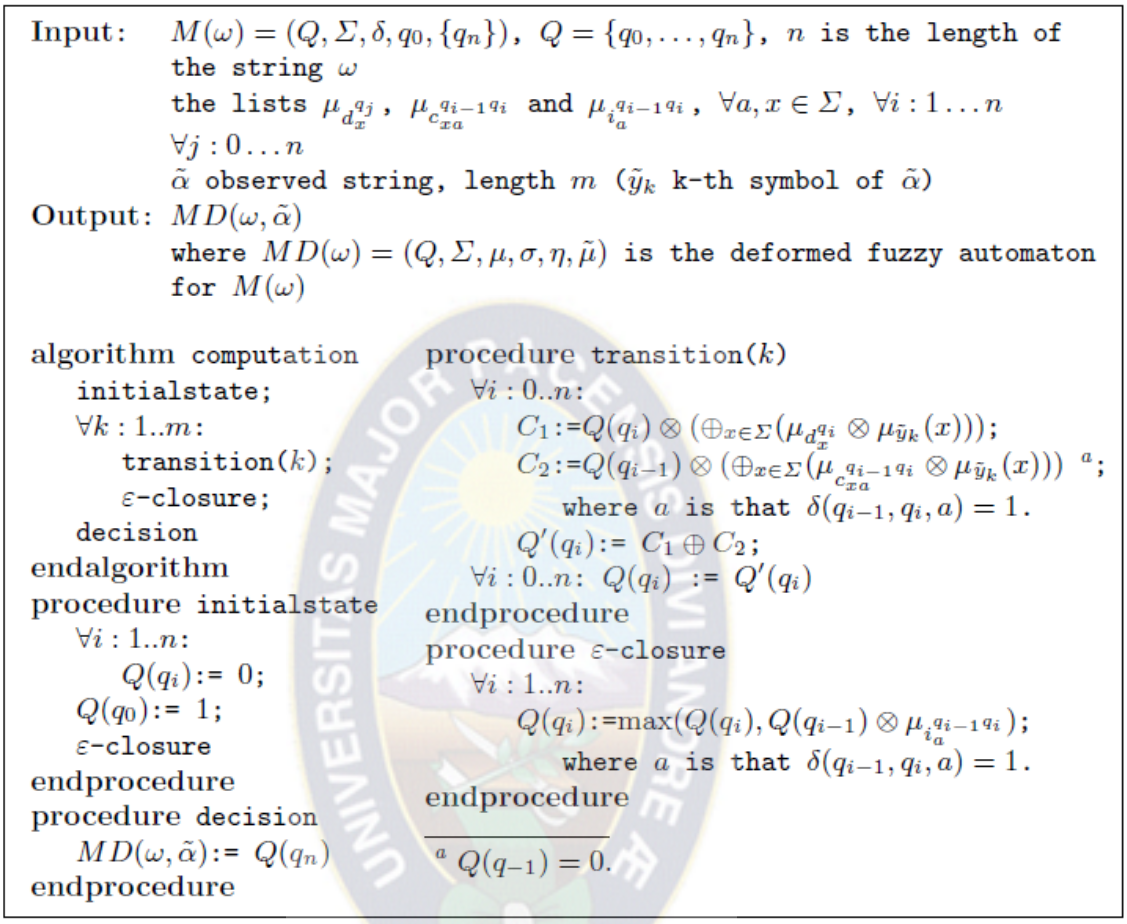

Figura 18 Algoritmo de Reina, González de Mendivil y Garitagoitia que construye un autómata difuso parcial para la búsqueda de un patrón en un texto.

difusos (Atanassov, 1999), la idea central es la construcción de un autómata intuicionista difuso para un patrón y con el cual recorrer un texto señalando aquellas cadenas que contengan un grado de pertenencia aceptable para el lenguaje que acepta el autómata intuicionista, dicho autómata es construido bajo el algebra de *Gödel* por lo cual las operaciones $\otimes$ y $\oplus$ son definidas como máximo y mínimo respectivamente.

Recordando la definición, un autómata intuicionista difuso sobre Σ y $\mathcal{L}$ es una séptupla $\mathcal{A} = (Q, \xi, \gamma, \alpha, \beta, \chi, \rho)$ donde:

- Q denota el conjunto no vacío de estados.
- $\alpha, \beta: Q \to L$ denotan respectivamente, las funciones de pertenencia y no pertenencia de los estados iniciales difusos intuicionistas. Así, el conjunto de estados iniciales intuicionistas difusos es representado por $(\tilde{\alpha}, \tilde{\beta}) = \{(q, \alpha(q), \beta(q)) | q \in Q\}$.
- $\chi, \rho: Q \to L$ denotan respectivamente, las funciones de pertenencia y no pertenencia de los estados finales difusos intuicionistas. Así, el conjunto de estados finales intuicionistas difusos es representado por $(\tilde{\chi}, \tilde{\rho}) = \{(q, \chi(q), \rho(q)) | q \in Q\}$.
- $\xi, \gamma: Q \times (\Sigma \cup \{\lambda\}) \times Q \to L$ representan las funciones de pertenencia y no pertenencia de las transiciones difusas intuicionistas, respectivamente.

Sea $P = a_1 a_2 \dots a_m$ y $T = b_1 b_2 \dots b_n$ el patrón y el texto observado, las transiciones $\xi(q_{t-1}, x, q_t)$ y $\gamma(q_{t-1}, x, q_t)$ para $1 \leq t \leq m$ tienen la siguiente interpretación:

- Si $\xi(q, b_r, q)$ y $\gamma(q, b_r, p)$ son aplicables, no existe error, es decir, el símbolo es el esperado por lo cual el valor asociado a las transiciones es $\xi(q, b_r, p) = 1$ y $\gamma(q, b_r, p) = 0$.
- Si $\xi(q, \lambda, p)$ y $\gamma(q, \lambda, p)$ son aplicables, las transiciones simulan la inserción de un símbolo b_r, teniendo en cuenta que el símbolo b_r es el correspondiente a la transición del estado q al estado p en la transición donde no existe error, así el valor asociado a las transiciones es $\left(\xi_{i_{b_r}^{qp}}, \gamma_{i_{b_r}^{qp}}\right)$.

algorithm computation
  initialdistrib;
  $\forall k : 1, 2, \cdots, n$:
    transition (k);
    ϵ-closure;
  decision
end algorithm
procedure initialdistrib
  $\forall i : 1, 2, \cdots, m$
    $Q(q_i) := (Q_1(q_i), Q_2(q_i))$
        $:= (0, 1)$;
    $Q(q_0) := (Q_1(q_0), Q_2(q_0))$
        $:= (1, 0)$;
  ϵ-closure
end procedure
procedure decision
  $IFA(\omega, \psi) := Q(q_m)$
end procedure

procedure transition (k)
  $\forall i : 0, 1, 2, \cdots, m$
    $R_1 := (C_1, D_1)$
      $:= (\min\{Q_1(q_i), \tau_{d_x^{q_i}}\}, \max\{Q_2(q_i), \gamma_{d_x^{q_i}}\})$;
    $R_2 := (C_2, D_2)$
      $:= (\min\{Q_1(q_{i-1}), \tau_{c_{xa}^{q_{i-1}q_i\ a}}\},$
        $\max\{Q_2(q_{i-1}), \gamma_{c_{xa}^{q_{i-1}q_i\ a}}\}),$
      where a is that $\delta(q_{i-1}, a, q_i) = 1$.
    $Q'(q_i) := (\max\{C_1, C_2\}, \min\{D_1, D_2\})$;
  $\forall i : 0, 1, 2, \cdots, m$
  $Q(q_i) := Q'(q_i) := (Q'_1(q_i), Q'_2(q_i))$
end procedure
procedure ϵ-closure
  $\forall i : 1, 2, \cdots, m$
  $Q(q_i) := (\max\{Q_1(q_i), \min(Q_1(q_{i-1}), \tau_{i_a^{q_{i-1}q_i}})\},$
      $\min\{Q_2(q_i), \max(Q_2(q_{i-1}), \gamma_{i_a^{q_{i-1}q_i}})\}),$
  where a is that $\delta(q_{i-1}, a, q_i) = 1$.
end procedure
.....
  $Q(q_{-1}) = (0, 1)$

Figura 19 Algoritmo de Ravi, Choubey y Tripati que construye un autómata difuso intuicionista para la búsqueda de un patrón en un texto. Nótese el parecido con el algoritmo de la figura 18, esto debido a que ambos son construidos de manera similar variando solo en la estructura algebraica utilizada para las operaciones.

- Si $\xi(q, x, p)$ y $\gamma(q, x, p)$ son aplicables para $x \neq b_r$ con b_r como el símbolo correspondiente a la transición del estado q al estado p donde no existe error, entonces las transiciones simulan la sustitución del símbolo x por b_r, el valor asociado a las transiciones es $\left(\xi_{c_{x,b_r}^{qp}}, \gamma_{c_{x,b_r}^{qp}}\right)$.

- Si $\xi(q, x, q)$ y $\gamma(q, x, q)$ son aplicables, las transiciones simulan la eliminación del símbolo x y el valor asociado a las transiciones es $\left(\xi_{d_x^q}, \gamma_{d_x^q}\right)$.

Es importante notar que los valores $\xi_{i_{b_r}^{qp}}, \gamma_{i_{b_r}^{qp}}, \xi_{c_{x,b_r}^{qp}}, \gamma_{c_{x,b_r}^{qp}}, \xi_{d_x^q}, \gamma_{d_x^q} \in L$.

El algoritmo para construir un autómata intuicionista difuso comienza construyendo los estados iniciales difusos en base a las funciones de pertenencia y no pertenencia α, β. La fórmula descrita en el capítulo 4 para el cálculo de la clausura – λ y para las transiciones son ejecutadas una tras otra para cada símbolo difuso intuicionista para finalmente evaluar por la función de pertenencia y no pertenencia de estados finales χ, ρ el grado con el cual el resultado es y no es aceptado por el autómata intuicionista difuso.

En el cálculo de $\tilde{\xi}(P', x)$ y $\tilde{\gamma}(P', x)$ solo existen dos posibles transiciones para alcanzar el estado q_i:

- Desde el estado q_{i-1}, mediante una sustitución o por un reemplazo de un carácter por sí mismo.
- Desde el mismo estado q_i, mediante una operación de eliminación.

En el cálculo de $\xi^{\lambda}(P')$ y $\gamma^{\lambda}(P')$ es fácil de probar que q_i es solo afectado por el valor de q_{i-1} y la transición que representa la operación de inserción entre los estados q_{i-1} y q_i.

El procedimiento *transition* tiene una complejidad $O(m)$ al igual que $\lambda - closure$, así la ejecución del algoritmo para una cadena observada es $O(n \times m)$.

5.7. ALGORTIMO ANDREJKOVÁ, ALMARIMI, MAHMOUD

En (Andrejková, Almarimi, & Mahmoud, 2013) se propone la utilización de un autómata construido bajo el algoritmo propuesto por (Aho & Corasick, 1975), el cual busca las ocurrencias exactas de un número de patrones en un texto. Su construcción es modificada usando una función de similitud entre símbolos, la cual evalúa cuán parecidos son dos símbolos, y aplicando operaciones de composición y multiplicación en una

Algorithm Knuth, Morris and Pratt's Algorithm

```
1: procedure PREFIX- FUNCTION(P)
2:     m = P.lenght
3:     prefixVec[0] = 0
4:     for q ← 1, m − 1 do
5:         while k > 0 ∧ P[k] ≠ P[q] do
6:             k = prefixVec[k − 1]
7:         if P[k] = P[q] then
8:             k = k + 1
9:         prefixVec[q] = k
```

Figura 20 Algoritmo KMP con el que se construye un vector de fallos en base al prefijo propio que iguala con un sufijo propio dentro del texto.

estructura algebraica determina las ocurrencias aproximadas de los patrones además de las ocurrencias exactas.

Dado que la búsqueda de varios patrones en un texto no está contemplado dentro de la problemática de la presente tesis, se realiza una pequeña modificación en el autómata construido originalmente, optando por la construcción descrita en (Knuth, Morris, & Pratt, 1977) llamado algoritmo KMP, con la cual se realiza la búsqueda de las ocurrencias exactas de un patrón en un texto donde se utilizarán la función de similitud y operaciones antes indicadas.

El algoritmo KMP original construye un vector de fallos de acuerdo al prefijo propio de mayor longitud que iguale con un sufijo propio hasta una cierta posición, en la figura 24 se detalla la construcción de dicho vector. Sin perdida de generalidad, es posible implementar un lugar de un vector de fallos un autómata donde los arcos o transiciones hacia adelante representarían comparaciones exitosas y los arcos hacia atrás denominados enlaces de fallos representan el fallo en alguna posición.

Dado el alfabeto Σ definimos la relación difusa $f(\Sigma, \Sigma)$ como la función de similitud, que evidentemente es una función $f: \Sigma \times \Sigma \to \mathcal{L}$ por los conceptos desarrollados en capítulos anteriores, y está definida como:

$$f(a,b) = \begin{cases} 1, & a = b \\ v, & a \neq b \end{cases}$$

Con $v \in \mathcal{L}$, el valor de v depende de cuan similares sean a y b.

Sea la relación difusa $\mu_q(Q, \Sigma)$ y la función de similitud, hallamos la función de transición difusa $\delta_q(Q, \Sigma)$ para un autómata difuso:

$$\delta_q(p,a) = \bigvee_{b \in \Sigma} \mu_q(p,b) \otimes f(b,a)$$

La relación $\mu_q(Q, \Sigma)$ representa las transiciones del autómata construido por el algoritmo KMP por lo cual $\mu_q(p,x) \in \mathcal{L}$. Usando la fórmula para el cálculo de $\delta_q(p,a)$ se obtiene la función de transición difusa. Para un estado difuso $S \in \mathcal{F}(Q)$ y la relación difusa $\delta_q(p,a)$, su composición produce el siguiente estado difuso y está definido por:

$$(S \circ \delta_q)(p) = \bigvee_{q \in Q} S(q) \otimes \delta_q(q,p)$$

Por último, recordando que el lenguaje aceptado por el autómata difuso A para cualquier cadena $u = x_1 x_2 \ldots x_n$ se define como: $[\![\mathcal{A}]\!](u) = \sigma \circ \delta_{x_1} \circ \delta_{x_2} \circ \ldots \circ \delta_{x_n} \circ \tau$ se tiene todos los elementos para el funcionamiento del autómata difuso.

Los resultados varían de acuerdo a la estructura algebraica y la norma – t utilizada, y para la presente tesis se tomará a la estructura de Gödel como estructura de valores de verdad, por lo cual se tiene las siguientes modificaciones.

La primera asignación se hará en la función de similitud que al igual que en el autómata intuicionista difuso se tomará un solo valor para denotar la similitud de los símbolos del alfabeto por lo cual la fórmula ahora está dada por:

$$(a,b) = \begin{cases} 1, & a = b \\ 0.3, & a \neq b \end{cases}$$

La función de transición difusa $\delta_q(Q,\Sigma)$ es modificada reemplazando $\otimes$ y $\vee$ por las operaciones respectivas dentro de la estructura de Gödel con lo que se tiene:

$$\delta_q(p,a) = \max_{b\in\Sigma}\left(\min\left(\mu_q(p,b), f(b,a)\right)\right)$$

De igual forma, en $(S \circ \delta_q)(p)$ se reemplazan las operaciones respectivas con lo que se tiene:

$$(S \circ \delta_q)(p) = \max_{q\in Q}\left(\min\left(S(q), \delta_q(q,p)\right)\right)$$

Por último modificamos la fórmula de aceptación de autómata:

$$[\![\mathcal{A}]\!](u) = \max_{q\in Q}\left(\min\left(\delta^*(\sigma,u)(q), \tau(q)\right)\right)$$

6. RESULTADOS Y ANÁLISIS

En el presente capítulo se realizan las pruebas empíricas de los algoritmos aplicables en la práctica descritos. Los algoritmos excluidos son los propuestos por (Sellers, 1980) y (Masek & Paterson, 1978) ya que no son competitivos en la práctica pero estos representan o representaron en su momento una contribución significativa en el desarrollo de nuevas técnicas para resolver el problema.

La plataforma de prueba fue una Notebook Samsung Intel® Core™ i5-2450M CPU 2.50GHz, RAM 6 GB. Los experimentos son evaluados en textos de 3 a 5 MB con distintos patrones y los mismos patrones son usados para todos los algoritmos.

Se utilizan los siguientes textos para los experimentos:

- Lenguaje Natural: La biblia en su versión en inglés de 4MB, el proyecto Gutenberg de 2.5MB, el proyecto Gutenberg en su versión francesa de 2.3MB y texto en italiano de 1.5MB.
- Proteína: El archivo está formado por una secuencia de letras representando una cadena de proteínas de tamaño 3.2MB.
- Genoma: El archivo de tamaño 4.5MB está formado por una secuencia de letras con el alfabeto de tamaño 4.

6.1. RESULTADOS

A continuación, se expone los resultados obtenidos en la búsqueda de distintos patrones tomando como texto la biblia en su versión en inglés de 4 MB y el proyecto Gutenberg de 2.5MB.

m	k	ukk			nav			Wu		
2	0	6,7976	8,8659	8,5347	11,7789	11,2668	12,4904	4,3251	4,2145	3,9912
2	0	11,8230	11,6605	14,4690	16,4986	18,0249	17,0683	5,2312	4,8932	4,9513
2	0	11,3334	11,5767	9,8011	11,6243	12,6548	10,9760	4,5082	4,8943	4,8837
8	3	6,6560	6,4686	6,3748	18,1662	17,3898	17,2590	5,5648	5,7811	5,6762
8	3	8,0537	8,6890	8,2496	16,9059	18,6813	18,3483	5,8732	5,8236	5,7799
8	3	5,0873	5,9220	5,7159	13,7450	13,7280	13,6986	5,6743	5,7749	5,6935
8	3	4,4685	5,8368	5,5195	11,6923	12,2329	12,2675	6,0023	5,9932	6,1294
32	5	75,0549	83,5855	77,7222	21,1551	21,5776	21,0776	15,1551	14,9993	15,2354
32	5	55,0208	56,7942	56,2701	19,9682	20,1401	20,4526	16,4043	16,4362	15,9954
32	5	56,9857	60,0841	59,3020	23,5709	21,8489	24,8946	16,4539	16,4693	16,4759

Tabla 4 Se presenta los resultados de los algoritmos Ukkonen, Navarro y Wu et. al. para distintos patrones y errores máximos permitidos con el texto La Biblia versión en inglés de 4MB.

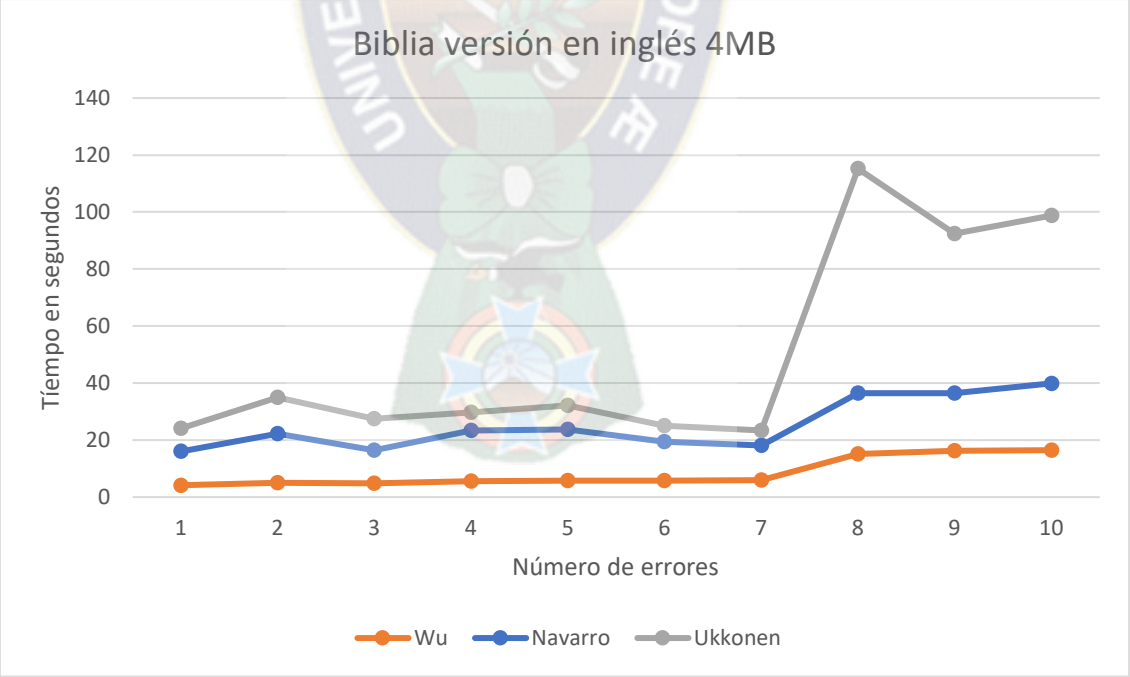

Figura 21 En la figura se muestra la comparativa de tiempos de ejecución entre los algoritmos descritos en la tabla 4, donde el algoritmo descrito por Wu tiene mejores tiempos de ejecución.

m	k	ravi			Andrejkova		
2	0	12,6174	11,9832	10,3982	28,963	23,2684	25,1281
2	0	10,9120	11,0932	11,2341	33,4576	32,5424	32,0396
2	0	12,8734	12,0512	11,8899	24,8361	24,5365	20,4837
8	3	29,9350	28,3992	27,8776	160,8160	162,9249	171,9197
8	3	21,9032	21,6733	23,4054	160,2376	157,5283	161,4123
8	3	23,0189	25,0002	25,8971	164,5960	160,3484	149,7151
8	3	24,3823	14,9873	25,3490	157,1421	154,7857	150,8394
32	5	83,8185	80,9992	79,9085	367,1324	372,2103	383,3240
32	5	72,6684	77,9231	79,9923	389,2310	383,2394	390,0023
32	5	85,2637	84,3218	82,9901	379,2341	378,9926	380,0312

Tabla 5 Se presentan los resultados obtenidos por los algoritmos Ravi et. al. y Andrejkova et. al. para los mismos patrones y texto de la tabla 4.

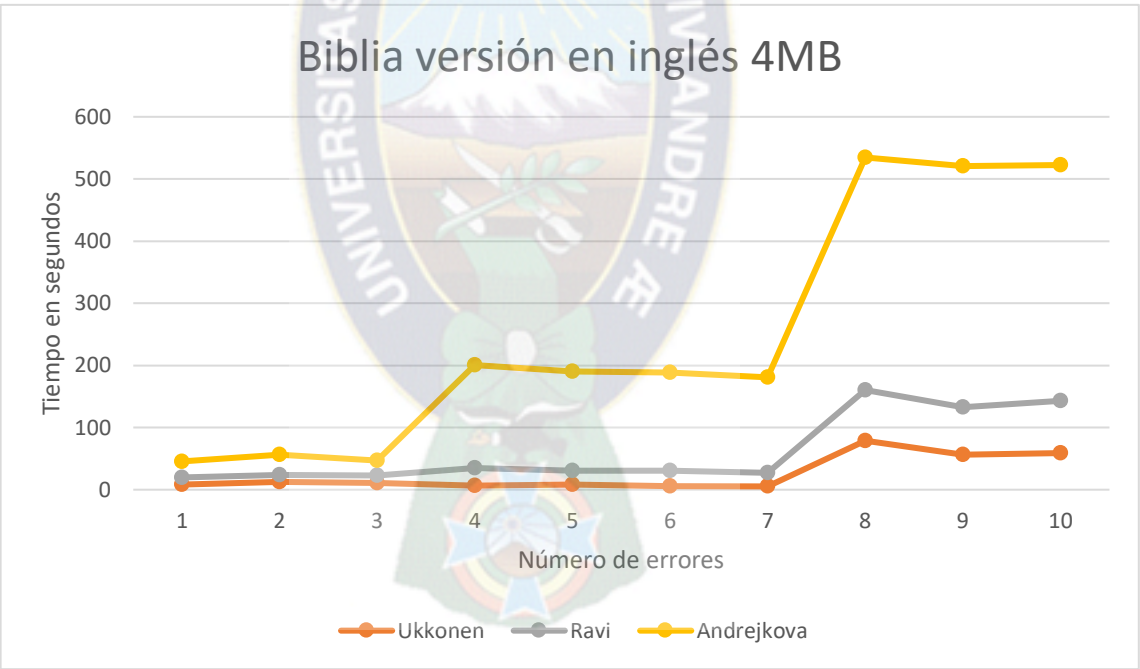

Figura 22 En la figura se muestra la comparativa de tiempos de ejecución entre los algoritmos descritos en la tabla 4 junto al algoritmo de Ukkonen que es el que tiene el peor tiempo en los algoritmos clásicos.

A continuación, se muestran los resultados obtenidos por los algoritmos en un autómata difuso y un autómata difuso intuicionista. La comparación de los algoritmos en autómatas difusos con el algoritmo de Ukkonen muestra que este a pesar de tener el peor

tiempo de ejecución entre los algoritmos clásicos, es aún mejor los modelados en autómatas difusos.

|P|	Ukkonen	Navarro	Ravi	Andrejkova	Wu
2	8,066066667	11,8453667	11,6662667	25,7865	4,176933333
2	12,6508	17,1972667	11,0797667	32,67986667	5,025233333
2	10,90373333	11,7517	12,2715	23,28543333	4,762066667
8	6,4998	17,605	28,7372667	165,2202	5,674033333
8	8,330766667	17,9785	22,3273	159,7260667	5,825566667
8	5,575066667	13,7238667	24,6387333	158,2198333	5,714233333
8	5,274933333	12,0642333	21,5728667	154,2557333	6,041633333
32	78,78753333	21,2701	81,5754	374,2222333	15,12993333
32	56,02836667	20,1869667	76,8612667	387,4909	16,27863333
32	58,7906	23,4381333	84,1918667	379,4193	16,46636667

Tabla 6 Tabla comparativa de tiempos promedios para distintos patrones y longitudes.

m	k	Ukkonen			Navarro			Wu		
2	0	16,7117	20,2532	20,7869	25,0629	25,5172	23,8998	20,4859	20,2145	21,9912
2	0	11,8230	11,6605	14,4690	31,3310	30,5580	31,4652	28,2312	29,4932	28,9313
2	0	11,3334	11,5767	9,8011	29,4381	30,0585	27,4808	24,492	24,8943	25,0048
8	3	16,6560	16,4686	16,4748	11,7818	10,3174	10,6525	9,5648	8,8821	9,6762
8	3	18,0537	18,6890	18,2496	12,2707	11,2398	12,3826	10,2394	10,8475	10,4847
8	3	15,0873	15,9220	15,7159	13,7450	13,7280	13,6986	11,3948	11,9488	12,0093
8	3	14,4685	15,8368	15,0195	11,6923	12,2329	12,2675	9,9491	9,4901	9,842
32	5	45,0549	43,5855	47,7222	26,6435	24,6738	25,9688	15,1551	14,9993	15,2354
32	5	55,0208	56,7942	56,2701	23,3261	23,9847	24,0930	16,4043	16,4362	15,9954
32	5	56,9857	60,0841	59,3020	24,9057	25,9345	25,7643	16,4539	16,4693	16,4759

Tabla 7 Algoritmos de Ukkonen, Navarro y Wu et. al. para distintos patrones y errores máximos permitidos con el texto representado un genoma de 4.5MB.

En la tabla 7 se expone los resultados obtenidos en la búsqueda de distintos patrones tomando como texto conformado por una secuencia de símbolos representando un genoma con un tamaño de 4.5MB y las cadena de proteínas de tamaño 3.2MB.

Se presentan las tablas y figuras comparativas de la búsqueda de patrones en el texto descrito anteriormente.

m	k	ravi		Andrejkova		
2	0	34,6174	33,9821	53,1249	58,3066	59,6424
2	0	45,7843	46,3363	54,5117	57,6906	49,4999
2	0	43,9452	44,3294	60,5605	65,2114	71,2875
8	3	29,9350	27,8776	224,7108	221,9207	225,6042
8	3	21,9032	23,4054	257,2468	257,5006	161,4123
8	3	33,0189	35,8971	164,5960	269,7811	149,7151
8	3	34,3823	35,3490	157,1421	263,1646	150,8394
32	5	83,8185	79,9085	367,1324	372,2103	383,3240
32	5	86,6684	88,0923	389,2310	383,2394	390,0023
32	5	85,2637	82,9901	379,2341	378,9926	380,0312

Tabla 9 Se presentan los resultados obtenidos por los algoritmos Ravi et. al. y Andrejkova et. al. para los mismos patrones y texto de la tabla 4.

|P|	Ukkonen	Navarro	Ravi	Andrejkova	Wu
2	19,2506	24,8266333	34,29975	57,0246333	13,5117
2	12,6508333	31,1180667	46,0603	53,9007333	10,5519
2	10,9037333	28,9924667	44,1373	65,6864667	24,7970333
8	16,5331333	10,9172333	28,9063	224,078567	9,37436667
8	18,3307667	11,9643667	22,6543	225,386567	10,5238667
8	15,5750667	13,7238667	34,458	194,6974	11,7843
8	15,1082667	12,0642333	34,86565	190,382033	9,7604
32	45,4542	25,7620333	81,8635	374,222233	15,1299333
32	56,0283667	23,8012667	87,38035	387,4909	16,2786333
32	58,7906	25,5348333	84,1269	379,4193	16,4663667

Tabla 8 Tabla comparativa de tiempos promedios para distintos patrones y longitudes.

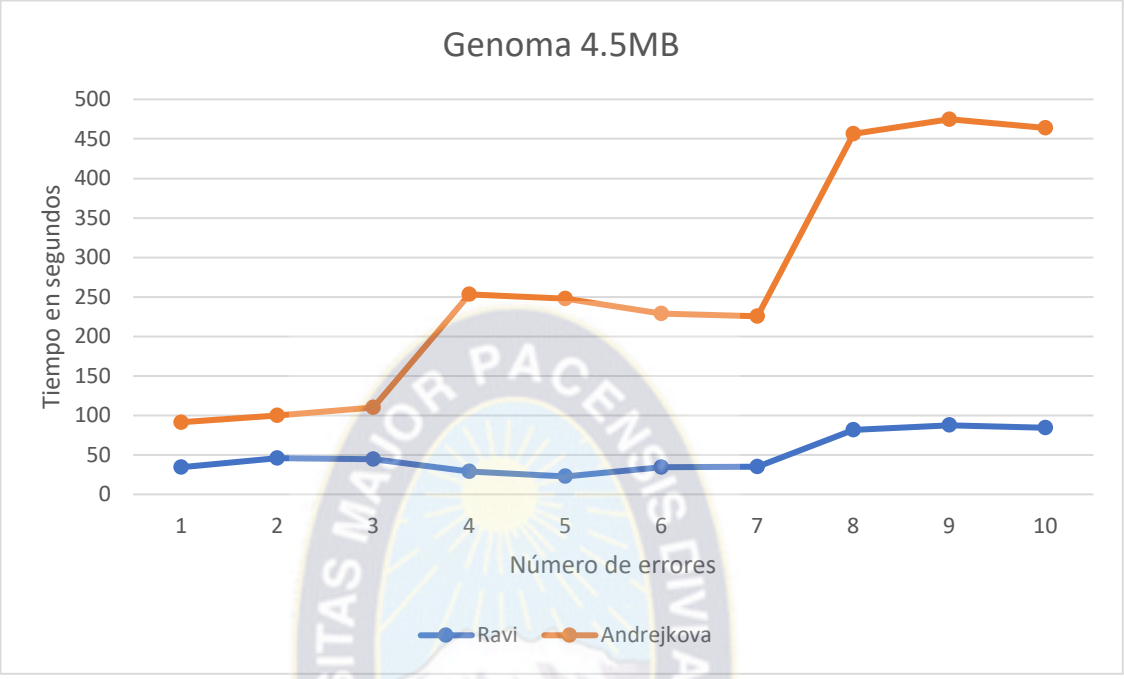

Figura 24 En la figura se muestra la comparativa de tiempos de ejecución entre los algoritmos modelados sobre autómatas difusos.

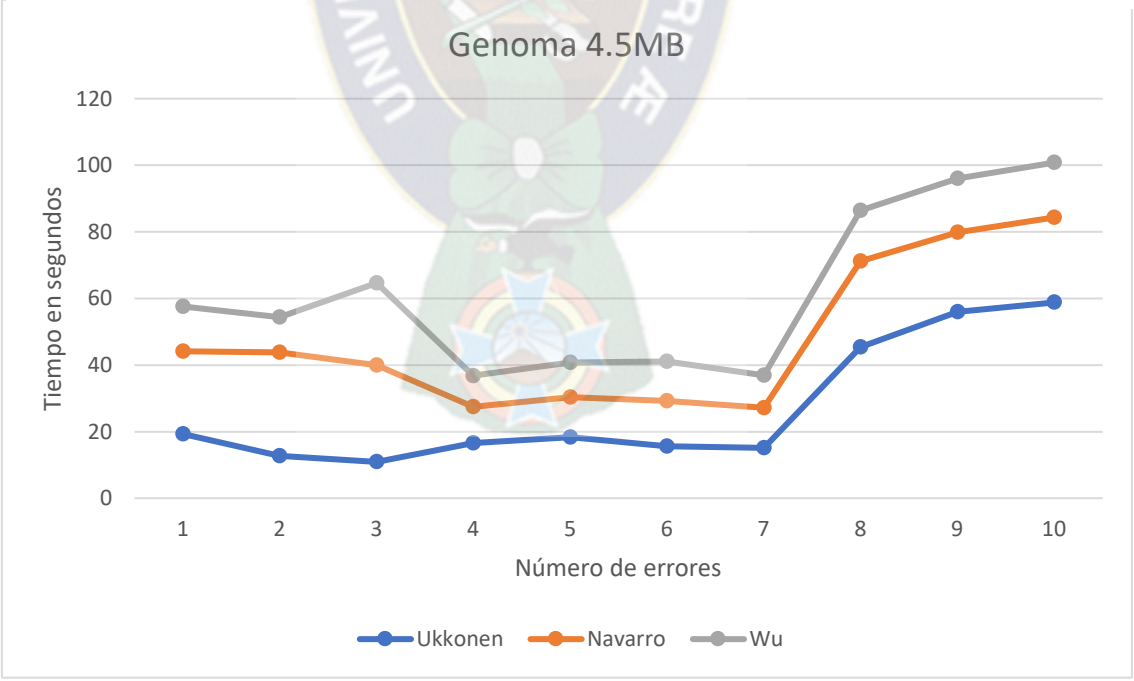

Figura 23 En la figura se muestra la comparativa de tiempos de ejecución entre los algoritmos clásicos.

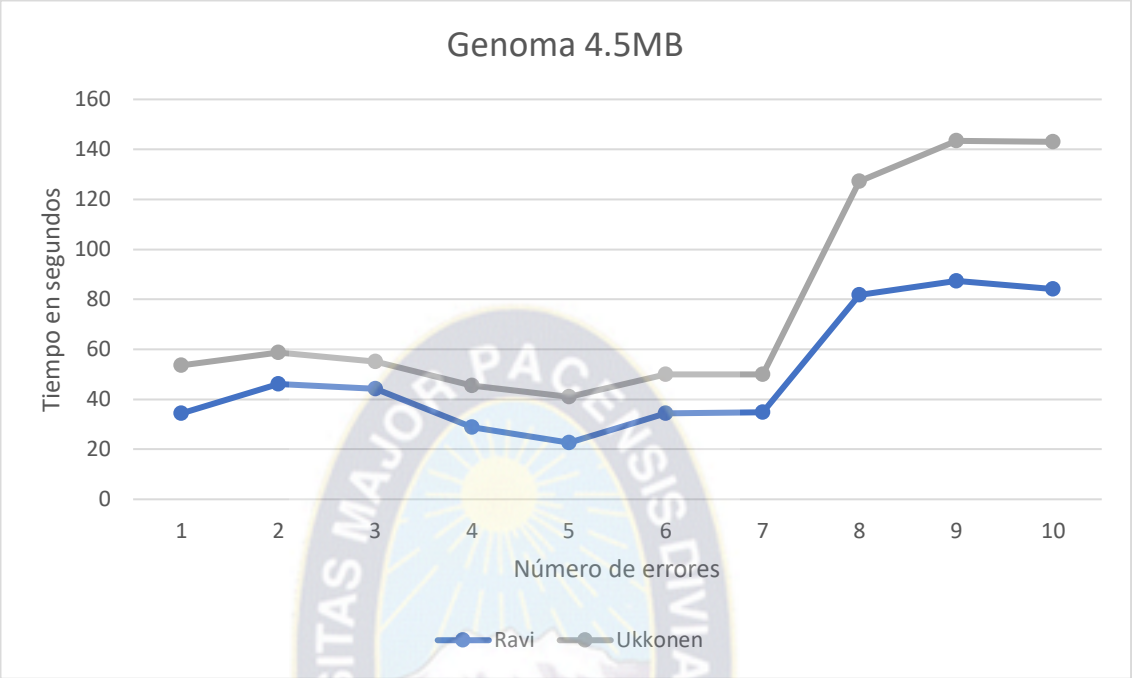

Figura 25 En la figura se muestra la comparativa de tiempos de ejecución entre los algoritmos descritos en la tabla 9 tomando al algoritmo de Ravi que tiene mejor tiempo de ejecución junto al algoritmo de Ukkonen de pero tiempo de ejecución entre los algoritmos clásicos.

7. CONCLUSIONES Y RECOMENDACIONES

7.1. CONCLUSIONES

Con base en los resultados de los experimentos realizados en el capítulo 6 se concluye lo siguiente:

Es importante señalar que a pesar de que los algoritmos (Echabone, Garitagoitia, & González de Mendívil), (Reina, González de Mendivil, & Garitagoitia, 1992) y principalmente el algoritmo (Ravi, Choubey, & Tripati, 2014) el cual fue evaluado en la experimentación en el capítulo 6, son descritos para el problema de emparejamiento aproximado de cadenas el desarrollo de los algoritmos se limita a comparar cadenas y dar como respuesta el grado de aproximación entre dichas cadenas la realizan una comparación entre cadenas, por lo cual para su aplicación las subcadenas del texto son obtenidas separándolas por los espacios en el texto para así comparar dichas subcadenas con el patrón y obtener su grado de pertenencia al autómata difuso intuicionista del patrón dado.

El algoritmo propuesto por (Andrejková, Almarimi, & Mahmoud, 2013) cumple hasta cierto punto el criterio del emparejamiento aproximado de cadenas lo cual será observado más adelante, a pesar de esto el algoritmo demuestra no ser competitivo en la práctica ya que queda muy por debajo de los resultados de otros algoritmos cuando se toman patrones largos, por otra parte tiene un tiempo de ejecución razonable en patrones cortos.

Es evidente por la experimentación que el emparejamiento aproximado de cadenas modelado en autómatas difusos de manera general no es competitivo en la práctica, puesto

que, para patrones cortos el tiempo de desempeño es alto en relación a los demás algoritmos y para patrones largos se vuelven inaplicables por su alto tiempo de ejecución superando los 5 minutos razón por la cual la experimentación se realiza sólo hasta patrones de longitud 32.

Por otra parte, un punto muy importante son las ocurrencias encontradas por los algoritmos, por parte de los algoritmos denominados clásicos el número de ocurrencias encontradas es siempre el mismo ya que todos están basados en la idea original de Sellers, en cuanto a los autómatas difusos el número de ocurrencias encontradas es tan solo del 63% en promedio esto debido a que en el algoritmo (Ravi, Choubey, & Tripati, 2014) al comparar subcadenas se pierden ocurrencias ya que solo se devuelve un valor por subcadena del texto evaluada, en (Andrejková, Almarimi, & Mahmoud, 2013) por la propia construcción del autómata y las operaciones realizadas ciertas ocurrencias son perdidas.

Los algoritmos clásicos demuestran ser muy superiores respecto a aquellos que son modelados en autómatas difusos tanto en tiempo de ejecución, espacio de memoria ocupado y ocurrencias encontradas.

Por último, respecto a la hipótesis se concluye que los algoritmos de emparejamiento aproximado de cadenas modelados sobre autómatas difusos no tienen mejor desempeño que los algoritmos clásicos en términos de tiempo de ejecución y ocurrencias encontradas por lo cual se rechaza la hipótesis.

7.2. RECOMENDACIONES

Una vez concluido el presente trabajo se sugiere la posibilidad de continuar con alguna de las líneas de investigación desarrolladas e incluso incursionar en otras relacionadas con los temas tratados, se recomienda seguir con las siguientes temáticas:

Continuar con el estudio e investigación de los autómatas difusos en su ámbito formal como también profundizar en sus aplicaciones en los campos de control de sistemas, bases de datos, redes neuronales entre otros.

Estudiar la aplicabilidad de los autómatas difusos intuicionistas en comparaciones en diccionarios frente a los métodos utilizados comúnmente como análisis estadísticos, algoritmos genéticos, redes neuronales por nombrar algunos.

Finalmente se plantea incorporar a dicho estudio distintos algoritmos y métodos de emparejamiento aproximado de cadenas como podrían ser por paralelismo de bit, mediante procesos estadísticos, redes neuronales entre otros.

8. BIBLIOGRAFÍA

Aho, A., & Corasick, M. (1975). Efficient string matching: and aid to bibliographic search. *ACM Vol. 18*, 333-340.

Amir, A., & Lewenstein, M. L. (1997). Pattern matching in hypertext. *5th International Workshop on Algorithms and Data Structures (WADS '97)* (págs. 160-173). Berlin: Springer-Verlag.

Amir, A., Aumann, Y., Landau, G., Lewenstein, M., & Lewenstein, N. (1997). Pattern matching with swaps. *Foundations of Computer Science (FOCS '97)*, (pp. 144-153).

Andrejková, G., Almarimi, A., & Mahmoud, A. (2013). Approximate Pattern Matching using Fuzzy Logic. *CEUR Workshop Proceedings Vol. 1003* (págs. 52-57). Slovakia: ITAT.

Apostolico, A., & Guerra, C. (1987). The Longest Common Subsequence problem revisited. *Algorithmica 2*, 315-336.

Arlazarov, V., Dinic, A., Kronrod, M., & Faradzev, I. (1970). On economic construction of the transitive closure of a directed graph. *Soviet Math. Dokl. 11 No. 5*, 1209-1210.

Astrain, J., Garitagoitia, J., Gonzáles de Mendivíl, J., Villadangos, J., & Fariña, F. (2004). Approximate String Matching Using Deformed Fuzzy Automata: A Learning Experience. *Fuzzy Optimization and Decision Making 3*, 141-155.

Atanassov, K. (1999). *Intuitionistic Fuzzy Sets.* Berlin, Heidelberg: Springer-Verlag.

Baeza-Yates, R. (1991). Some new results on approximate string matching. *Workshop on Data Structures.* Dagstuhl.

Baeza-Yates, R., & Navarro, G. (1999). Faster approximate string matching. *Algorithmica 23*, 127-158.

Belohlavek, R. (2002). Determinism and fuzzy automata. *Information Sciences*, 205 - 209.

Belohlávek, R. (2002). *Fuzzy Relational Systems: Foundations and Principles.* New York: Kluwer.

Birkhoff, G. (1964). *Lattice theory.* American mathematical Society.

Blyth, T. (2006). *Lattices and Ordered Algebraic Structures.* Springer.

Chang, W., & Lampe, J. (1992). Theoretical and Empirical Comparisons of Approximate String Matching Algorithms. *3rd Sysmp. on Combinatorial Pattern Matching*, (págs. 172-181). Tucson,AZ.

Chatterjee, K., Henzinger, T., Ibsen-Jensen, R., & Otop, J. (2017). Edit Distance for Pushdown Automata. *Logical Methods in Computer Science*, 1-23.

Chavátal, V., & Sankoff, D. (1975). Longest common subsequences of two random sequences. *J. Appl. Probab. 12*, 306-315.

Ciric, M., & Ignjatovic, J. (2012). Bisimulations for fuzzy automata. *Fuzzy Sets and Systems 186*, 100 - 139.

Ciric, M., Droste, M., Ignjatovic, J., & Vogler, H. (2010). Determinization of weighted finite automata over strong bimonoids. *Information Sciences*, 3497-3520.

Ciric, M., Ignjatovic, J., & Bogdanovic, S. (2009). Uniform fuzzy relations and fuzzy functions. *Fuzzy Sets and Systems*, 1054 - 1081.

Deken, J. (1979). Some limit results for longest common subsequences. *Discrete Math 26*, 17-31.

Droste, M., Stüber, T., & Vogler, H. (2010). Weigthed finite automata over strong bimonoids. *Information Sciences*, 156-166.

Dubonis, D., & Prade, H. (1980). *Fuzzy Sets and Systems: Theory and Applications.* New York: Academic Press.

Echabone, J., Garitagoitia, J., & González de Mendívil, J. (s.f.). Deformed Fuzzy Automata for the Text Error Correction Problem.

Galil, Z., & Giancarlo, R. (1986). Improved string matching with k mismatches. *SIGACT News*, 52-54.

Gupta, M., Sardis, N., & Gaines, B. (1977). *Fuzzy Automata and Decision Processes.* New York: North-Holland.

Ignjatovic, J., Ciric, M., & Bogdanovic, S. (2008). Determinization of fuzzy autmomata with memebership values in complete residuated lattices. *Information Sciences*, 164-180.

Ignjatovic, J., Ciric, M., Bogdanovic, S., & Petkovic, T. (2010). Myhill Nerode type theory for fuzzy languages and automata. *Fuzzy Sets and Systems,* 1288-1324.

Ignjatovic, J., Ciric, M., Bogdanovic, S., & Petkovic, T. (2010). Myhill-Nerode type theory for fuzzy languages and automata. *Fuzzy Sets and Systems 161,* 1288-1324.

Jancic, Z., Ignjatovic, J., & Ciric, M. (2011). An improved algorithm for determinization of weigthed and fuzzy automata. *Information Sciences,* 1358-1368.

Jokinen, P., Tarhio, J., & Ukkonen, E. (1988). A Comparison of Approximate String Matching Algorithms. *Software - Practice and Experience, Vol. 1*, 1-19.

Karloff, H. (1993). Fast algorithms for approximately conunting mismatches. *Information Processing Lett.*, 53-60.

Kececioglu, J., & Sankoff, D. (1995). Exact and approximation algorithms for the inversion distance between two permutations. *Algorithmica 13*, 180-210.

Knuth, D., Morris, J., & Pratt, V. (1977). Fast Pattern Matching in Strings. *Siam J. Comput. Vol. 6*, 323-350.

Landau, G., & Vishkin, U. (1986). Efficient string matching with k mistmatches. *Theoretical Computer Science,* 239-249.

Lee, E., & Zadeh, A. (1969). Note on fuzzy languages. *Information Sciences 1*, 421 - 434.

Levenshtein, V. (1965). Binary codes capable of correcting spurious insertions and deletions of ones. *Problem Information Transmission 1*, 8-17.

Li, P., & Li, Y. (2007). Minimization of states in automata theory based on finite lattice-ordered monoids. *Information Sciences*, 1413 - 1421.

Li, Y. M., & Pedrycz, W. (2005). Fuzzy finite automata and fuzzy regular expressions with membership values in lattice ordered monoids. *Fuzzy Sets and Systems*, 68-92.

Liu, F., & Qiu, D. (2008). Decentralized supervisory control of fuzzy discrete systems. *European Journal of Control*, 234-243.

Masek, W., & Paterson, M. (1978). A Faster Algorithm Computing String Edit Distances. *Journal of Computer and System Sciences*, 18-31.

Melichar, B. (1996). String matching with k differences by finite automata. *IEEE CS Press.*, 256-260.

Micic, I. Z. (2014). *Bisimulations for Fuzzy Automata.* Nis: Faculty of Sciences and Mathematics.

Muth, R., & Manber, U. (1996). Approximate multiple string search. *7th Annual Symposium on combinatorial Pattern Matching (CMP '96)* (pp. 75-86). Berlin: Springer-Verlag.

Myers, G. (1999). A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *Journal of the ACM Vol. 46*, 395-415.

Navarro, G. (1997). A Partial Deterministic Automaton for Approximate String Matching. *4th South American Workshop on String Processing* (págs. 112-124). Carleton Univ. Press.

Navarro, G. (2001). A Guided Tour to Approximate String Matching. *ACM Computing Surveys, Vol.33*, 31-88.

Navarro, G., & Baeza-Yates, R. (1999). A new indexing method for approximate string matching. *10th Annual symposium on Combinatorial Pattern Matching* (págs. 163-185). Berlin: Springer-verlag.

Navarro, G., & Baeza-Yates, R. (1999). Fast multi-dimensional approximate pattern matching. *10th Annual Symposium on Combinatorial Pattern Matching (CPM '99)* (pp. 243-257). Berlin: Springer-verlag.

Navarro, G., & Raffinot, M. (2002). *Flexible Pattern Matching in Strings.* New York: Cambridge University Press.

Needleman, S., & Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol. 48*, 444-453.

Qiu, D. (2001). Automata theory based on completed resiguated lattice-valued logic. *Science in China*, 419 - 429.

Qiu, D., & Liu, F. (2009). Fuzzy discrete-event systems under fuzzy observability and a test algorithm. *IEEE Transactions on Fuzzy Systems*, 578-589.

Ravi, K. M., Choubey, A., & Tripati, K. (2013). Intuitionestic Fuzzy Automaton for Approximate String Matching. *Fuzzy Information and Engineering*, 29-39.

Ravi, K. M., Choubey, A., & Tripati, K. K. (2014). Intuitionistic Fuzzy Automaton for Approximate String Matching. *Fuzzy Information and Engineering*, 29-39.

Réigner, M., & Szpankowski, W. (1997). On the approximate pattern occurrence in a text. *IEEE Press*.

Reina, R., González de Mendivil, J., & Garitagoitia, J. (1992). Improed Character Recognition System based on a Neural Network Invorporating the context via Fuzzy Automata. *2nd International Conference on Fuzzy Logic and Neural Networks*, (págs. 1143-1146). Iizuka - Japan.

Sankoff, D., & Kruskal, J. (1983). String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. *Addison - Wesley, Reading*, 363-365.

Sankoff, D., & Mainville, S. (1983). Common Subsequiences and Monotone Subsequences. *Addison - Wesley*, 363-365.

Santos, E. (1968). *Maximin automata, Information and Control.* Journal of Mathematical Analysis and Applications.

Sellers, P. (1980). The theory and computation of evolutionary distances: pattern recognition. *J. Algor. 1*, 359-373.

Stamenkovic, A., Ciric, M., & Ignjatovic, J. (2014). Reduction of fuzzy automata by means of fuzzy quasi-orders. *Information Sciences*.

Stamenkovic, A., Ciric, M., & Ignjatovic, J. (2015). Different Models of Automata with Fuzzy States. *Ser. Math. Inform. Vol. 30,* 235-253.

Tarhio, J., & Ukkonen, E. (1988). A greedy approximation algorithm for constructing shortest common superstrings. *Theorical Computer Science,* 131-145.

Tichy, W. (1984). The string-to-string correction problem with block moves. *ACM Trans. Computer Systems.,* 309-321.

Ukkonen, E. (1985). Finding approximate patterns in strings. *J. Algor. 6,* 132-137.

Ukkonen, E. (1992). Approximate string matching with q-grams and maximal matches. *Theorical Computer Science,* 191-211.

Wechler, W. (1978). *The Concept of Fuzziness in Automata and Language Theory*. Berlin: Akademie - Verlag.

Wee, W. (1969). A formulation of fuzzy automata and its application as a model of learning systems. *IEEE Transaction on Systems,* 215 - 223.

Wu, S., & Manber, U. (1992). Fast text searching allowing errors. *Commun. ACM 35, 10,* 83-91.

Wu, S., Manber, U., & Myers, G. (1992). A Sub-quadratic Algorithm for Approximate Limited Expression Matching. *Algorithmica 15,* 50-67.

Xing, H., Zhang, Q., & Huang, K. (2012). Analysis and control of fuzzy discrete event systems using a bisimulation equivalence. *Theoretical Computer Science,* 100-111.

9. ANEXOS

Código de la implementación del Algoritmo de Ukkonen.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UkkonenConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            //System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\Textos\bible.txt");
            //string P = "that day";

            System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\Textos\ecoli.txt");
            string P = Console.ReadLine();

            //System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\Textos\orlando_.txt");

            //System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\Textos\pg5423.txt");

            //System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual Studio\Textos\protein -
hs.txt");

            //System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual Studio\Textos\protein -
sc.txt");

            // Número de errores admitidos
            int k = 0;

            DateTime tiempostart = DateTime.Now;
            // creación del Autómata de Ukkonen
            Automata Ukk = new Automata(P, k);
            DateTime tiempofin = DateTime.Now;
            TimeSpan time1 = new TimeSpan(tiempofin.Ticks - tiempostart.Ticks);

            DateTime start = DateTime.Now;
```

```
            while (!file.EndOfStream)
            {
                string T = file.ReadToEnd();
                start = DateTime.Now;
                for (int i = 0; i < T.Length; i++)
                {
                    if (Ukk.transicion(T[i]))
                    {
                        Console.WriteLine("Ocurrencia en: " + i);
                    }
                }
            }
            DateTime fin = DateTime.Now;
            TimeSpan time2 = new TimeSpan(fin.Ticks - start.Ticks);


Console.WriteLine("============================================================");
            Console.WriteLine("construcción de autómata: " + time1.ToString());

Console.WriteLine("============================================================");

Console.WriteLine("============================================================");
            Console.WriteLine("recorrido del texto: " + time2.ToString());

Console.WriteLine("============================================================");
            TimeSpan total = new TimeSpan(time1.Ticks + time2.Ticks);

Console.WriteLine("============================================================");
            Console.WriteLine("tiempo total: " + total.ToString());

Console.WriteLine("============================================================");

            Console.ReadKey();
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UkkonenConsole
{
    class Automata
    {
        //Árbol de estados actuales
        private Arbol states;
        // conjunto de columnas, estados de la matriz
        private Dictionary<string, int> index;
        // Cola de estados
        private Queue<int[]> nuevos;
```

```
// Conjunto de estados finales
private HashSet<int> F;
// Alfabeto del autómata
private HashSet<char> Alfabeto;
// Conjunto de transiciones(estado+simbolo, estado)
private Dictionary<string, int> transiciones;
// Patron observado
private string patron;

private int estadoActual;

// Errores permitidos en la distancia de edición
private int errores;

public Automata(string P, int k)
{
    // Inicializacion de variables
    states = new Arbol();
    index = new Dictionary<string, int>();
    nuevos = new Queue<int[]>();

    // estados Finales vacio, alfabeto y errores = k
    F = new HashSet<int>();
    Alfabeto = setAlfabeto(P);
    transiciones = new Dictionary<string, int>();
    patron = P;
    estadoActual = 0;
    errores = k;

    ukkonenAlgorithm();
}

private void ukkonenAlgorithm()
{
    int m = patron.Length;
    int[] S = new int[m + 1];
    int i = 0;

    for (int j = 0; j <= errores; j++)
        S[j] = j;
    for (int j = errores + 1; j <= m; j++)
        S[j] = errores + 1;

    string cadenaS = aCadena(S);
    index.Add(cadenaS, i);
    nuevos.Enqueue(S);
    states.añadir(S);
    if (m <= errores)
        F.Add(i);

    while (nuevos.Count != 0)
    {
        S = nuevos.Dequeue();
```

```
                cadenaS = aCadena(S);
                foreach (char b in Alfabeto)
                {
                    int[] _S = nextColumn(S, b);
                    string cadena_S = aCadena(_S);
                    if (states.añadir(_S))
                    {
                        nuevos.Enqueue(_S);
                        i++;
                        index.Add(cadena_S, i);
                        if (_S[m] <= errores)
                            F.Add(i);
                    }
                    string a = index[cadenaS] + b.ToString();
                    transiciones.Add(a, index[cadena_S]);
                }
            }
        }

        private int[] nextColumn(int[] S, char b)
        {
            int[] _S = new int[S.Length];
            _S[0] = 0;

            for (int i = 1; i <= patron.Length; i++)
            {
                _S[i] = (patron[i - 1] == b) ? S[i - 1] : menor(S[i - 1], _S[i -
1], S[i]) + 1;
                if (_S[i] > errores + 1)
                    _S[i] = errores + 1;
            }

            return _S;
        }

        public bool transicion(char a)
        {
            a = (Alfabeto.Contains(a)) ? a : 'þ';
            estadoActual = transiciones[estadoActual.ToString() + a];
            if (F.Contains(estadoActual))
                return true;
            else
                return false;
        }

        private string aCadena(int[] S)
        {
            string aux = string.Empty;
            for (int i = 0; i < S.Length; i++)
                aux = aux + S[i];
            return aux;
        }

        private HashSet<char> setAlfabeto(string P)
```

```
        {
            HashSet<char> aux = new HashSet<char>();
            for (int i = 0; i < P.Length; i++)
                aux.Add(P[i]);
            aux.Add('þ');
            return aux;
        }

        private int menor(int a, int b, int c)
        {
            if (a < b)
            {
                if (a < c)
                    return a;
                else
                    return c;
            }
            else
            {
                if (b < c)
                    return b;
                else
                    return c;
            }
            return 0;
        }
    }
}
```

Código de implementación algoritmo de Wu.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleWuManberMyers2
{
    class Program
    {
        static void Main(string[] args)
        {
            // lectura del patrón

Console.WriteLine("============================================================");
            Console.WriteLine("=================== Ingresar el patrón
==================");

Console.WriteLine("============================================================");
            string P = "limpia";
```

```
            //string P = Console.ReadLine();
            // Número de errores admitidos
            int k = 2;
            // tamaño de los subvectores
            int r = 3;

            DateTime tiempostart = DateTime.Now;
            // Creacion del Automata Wu, Manber, Myers
            Automata WMM = new Automata(P, k, r);
            DateTime tiempofin = DateTime.Now;
            TimeSpan time1 = new TimeSpan(tiempofin.Ticks - tiempostart.Ticks);

            // lectura del texto
            DateTime start = DateTime.Now;
            System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\ConsoleWuManberMyers2\ConsoleWuManberMyers2\prueba.txt");

            while (!file.EndOfStream)
            {
                string T = file.ReadToEnd();
                start = DateTime.Now;
                int Y = (int)Math.Ceiling((double)P.Length / (double)r);
                int I = (int)Math.Pow(3, r + 1) - 2;
                int K = k + r;
                int y = (int)Math.Ceiling((double)k / (double)r);

                int e = 0;
                int c = 0;
                int i = 0;

                int[] D = new int[Y + 1];

                for (int p = 1; p <= y; p++)
                    D[p] = I;
                e = y * r;

                for (int j = 0; j < T.Length; j++)
                {
                    c = 0;
                    for (int p = 1; p <= y; p++)
                    {
                        i = WMM.getCV(p, T[j]) + D[p] + c;
                        D[p] = WMM.getGOTO(i);
                        c = WMM.getCOUT(i);
                    }
                    if (e == k && y < Y)
                    {
                        i = WMM.getCV(y + 1, P[j]) + I + c;
                        y++;
                        D[y] = WMM.getGOTO(i);
                        e = e + r + WMM.getCOUT(i);
                    }
                    else
```

```
                    {
                        e += c;
                        while (e > K)
                        {
                            e -= WMM.getBKS(D[y]);
                            y--;
                        }
                    }
                    if (y == Y && e <= k)
                        Console.WriteLine("Ocurrencia en: " + j);
                }
            }

            DateTime fin = DateTime.Now;
            TimeSpan time2 = new TimeSpan(fin.Ticks - start.Ticks);


Console.WriteLine("============================================================");
            Console.WriteLine("construcción de autómata: " + time1.ToString());

Console.WriteLine("============================================================");

Console.WriteLine("============================================================");
            Console.WriteLine("recorrido del texto: " + time2.ToString());

Console.WriteLine("============================================================");
            TimeSpan total = new TimeSpan(time1.Ticks + time2.Ticks);

Console.WriteLine("============================================================");
            Console.WriteLine("tiempo total: " + total.ToString());

Console.WriteLine("============================================================");

            Console.ReadKey();
        }
    }
}



using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleWuManberMyers2
{
    class Automata
    {
        // Alfabeto del autómata
        private HashSet<char> Alfabeto;
        // tamaño de los subvectores
        private int r;
```

```
// Conjunto de vectores característicos
private Dictionary<string, int> CV;
private HashSet<int> valueCV;

// Conjunto de proximos estados
int[] GOTO;

// Conjunto de carrys de salida
int[] COUT;

// Conjunto de sumas máximas de bloque
int[] BKS;

// Patrón
private string P;
// máximo de errores admitidos
private int k;

public Automata(string patron, int errores, int dimSubVectores)
{
    CV = new Dictionary<string, int>();
    valueCV = new HashSet<int>();

    P = patron;
    k = errores;
    r = dimSubVectores;
    GOTO = new int[3 * (int) Math.Pow(6, r)];
    COUT = new int[3 * (int) Math.Pow(6, r)];
    BKS = new int[(int) Math.Pow(3, r + 1) - 1];
    Alfabeto = setAlfabeto(patron);
    setCV();
    setBKS();
    setAutomata(r);
}
private HashSet<char> setAlfabeto(string P)
{
    HashSet<char> aux = new HashSet<char>();
    for (int i = 0; i < P.Length; i++)
        aux.Add(P[i]);
    aux.Add('þ');
    return aux;
}

private void setCV()
{
    for (int i = 1; i <= (P.Length / r); i++)
    {
        foreach (char a in Alfabeto)
        {
            int total = 0;
            for (int j = 0; j < r; j++)
            {
                int val = (P[r * (i - 1) + j] == a) ? 1 : 0;
```

```
                        total += val * (int)Math.Pow(2, r - (j + 1));
                    }
                    int num = total * (int)Math.Pow(3, r + 1);
                    CV.Add(i + a.ToString(), num);
                    valueCV.Add(num);
                }
            }
        }

        public int getCV(int p, char a)
        {
            a = (Alfabeto.Contains(a)) ? a : 'þ';
            return CV[p + a.ToString()];
        }
        public int getGOTO(int n)
        {
            return GOTO[n];
        }
        public int getCOUT(int n)
        {
            return COUT[n];
        }
        public int getBKS(int n)
        {
            return BKS[n];
        }
        private void setAutomata(int r)
        {
            for (int i = 0; i <= Math.Pow(3,r) - 1; i++)
            {
                int[] vecState = makeState(i);
                foreach (int valCV in valueCV)
                {
                    int[] vecCV = getCV(valCV);
                    int[] nextD = new int[r];
                    for (int c = -1; c <= 1; c++)
                    {
                        int carry = c;
                        int total = 0;
                        //int bks = 0;
                        for (int k = 0; k < r; k++)
                        {
                            nextD[k] = menor(1, vecCV[k] - carry, vecState[k] -
carry + 1);
                            switch (nextD[k])
                            {
                                case 0:
                                    total += (int) Math.Pow(3, r - (k + 1));
                                    break;
                                case 1:
                                    total += 2 * (int) Math.Pow(3, r - (k + 1));
                                    break;
                            }
                            carry = carry + nextD[k] - vecState[k];
```

```
                    //bks += nextD[k];
                }
                int a = 3 * i + 1;
                int val = a + valCV + c;
                GOTO[val] = 3 * total + 1;
                COUT[val] = carry;
                //BKS[3 * total + 1] = bks;
            }
        }
    }
}
private void setBKS()
{
    for (int i = 0; i <= Math.Pow(3, r) - 1; i++)
    {
        int total = 0;
        int[] vecState = makeState(i);
        for (int k = 0; k < r; k++)
            total += vecState[k];
        BKS[3 * i + 1] = total;
    }
}
private int[] getCV(int n)
{
    int[] aux = new int[r];
    int dig;
    for (int i = 0; i < r; i++)
    {
        dig = n % 2;
        n = n / 2;
        aux[r - 1 - i] = dig;
    }
    return aux;
}
private int[] makeState(int n)
{
    int[] aux = new int[r];
    int dig;
    for (int i = 0; i < r; i++)
    {
        dig = n % 3;
        n = n / 3;
        switch (dig)
        {
            case 0:
                aux[r - 1 - i] = -1;
                break;
            case 1:
                aux[r - 1 - i] = 0;
                break;
            case 2:
                aux[r - 1 - i] = 1;
                break;
        }
```

```
            }
            return aux;
        }
        private int menor(int a, int b, int c)
        {
            if (a < b)
            {
                if (a < c)
                    return a;
                else
                    return c;
            }
            else
            {
                if (b < c)
                    return b;
                else
                    return c;
            }
            return 0;
        }
    }
}
```

Código de implementación algoritmo de Navarro

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NavarroConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            //System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\Textos\bible.txt");
            //string P = "that day";

            System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual
Studio\Textos\ecoli.txt");
            string P = Console.ReadLine();

            // Número de errores admitidos
            int k = 0;
```

```
            DateTime tiempostart = DateTime.Now;
            // Creacion del Automata Navarro
            Automata Nav = new Automata(P, k);
            DateTime tiempofin = DateTime.Now;
            TimeSpan time1 = new TimeSpan(tiempofin.Ticks - tiempostart.Ticks);

            DateTime start = DateTime.Now;
            //int cnt = 0;
            while (!file.EndOfStream)
            {
                string T = file.ReadToEnd();
                start = DateTime.Now;
                for (int i = 0; i < T.Length; i++)
                {
                    if (Nav.transicion(T[i]))
                    {
                        Console.WriteLine("Ocurrencia en: " + i);
                        //cnt++;
                    }
                }
            }
            DateTime fin = DateTime.Now;
            TimeSpan time2 = new TimeSpan(fin.Ticks - start.Ticks);


Console.WriteLine("============================================================");
            Console.WriteLine("construcción de autómata: " + time1.ToString());

Console.WriteLine("============================================================");

Console.WriteLine("============================================================");
            Console.WriteLine("recorrido del texto: " + time2.ToString());

Console.WriteLine("============================================================");
            TimeSpan total = new TimeSpan(time1.Ticks + time2.Ticks);

Console.WriteLine("============================================================");
            Console.WriteLine("tiempo total: " + total.ToString());

Console.WriteLine("============================================================");
            //Console.WriteLine(cnt);

            Console.ReadKey();
        }
    }
}



using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace NavarroConsole
{
    class Automata
    {
        //Árbol de estados actuales
        private Arbol states;
        // conjunto de columnas, estados de la matriz
        private Dictionary<string, int> index;

        // Conjunto de estados finales
        private HashSet<int> F;
        // Alfabeto del autómata
        private HashSet<char> Alfabeto;
        // Conjunto de transiciones(estado+simbolo, estado)
        private Dictionary<string, int> transiciones;
        // Patron observado
        private string patron;

        private int[] vectorEstadoActual;
        private int estadoActual;
        private int i;

        // Errores permitidos en la distancia de edición
        private int errores;

        public Automata(string P, int k)
        {
            // Inicializacion de variables
            states = new Arbol();
            index = new Dictionary<string, int>();

            // estados Finales vacio, alfabeto y errores = k
            F = new HashSet<int>();
            Alfabeto = setAlfabeto(P);
            transiciones = new Dictionary<string, int>();
            patron = P;
            errores = k;

            navarroAlgorithm();
        }

        private void navarroAlgorithm()
        {
            int m = patron.Length;
            vectorEstadoActual = new int[m + 1];
            i = 0;

            for (int j = 0; j <= m; j++)
            {
                vectorEstadoActual[j] = j;
                if (vectorEstadoActual[j] > errores + 1)
                    vectorEstadoActual[j] = errores + 1;
```

```
            }

            index.Add(aCadena(vectorEstadoActual), i);
            states.añadir(vectorEstadoActual);
            if (m <= errores)
                F.Add(i);

            estadoActual = 0;
        }

        public bool transicion(char b)
        {
            b = (Alfabeto.Contains(b)) ? b : 'þ';
            int m = patron.Length;
            if (!transiciones.ContainsKey(estadoActual + b.ToString()))
            {
                vectorEstadoActual = nextColumn(vectorEstadoActual, b);
                if (states.añadir(vectorEstadoActual))
                {
                    i++;
                    index.Add(aCadena(vectorEstadoActual), i);
                    if (vectorEstadoActual[m] <= errores)
                        F.Add(i);
                }
                transiciones.Add(estadoActual + b.ToString(),
index[aCadena(vectorEstadoActual)]);
                estadoActual = index[aCadena(vectorEstadoActual)];
            }
            else
            {
                estadoActual = transiciones[estadoActual.ToString() + b];
                vectorEstadoActual = aVector(index.ElementAt(estadoActual));
            }


            if (F.Contains(estadoActual))
                return true;
            else
                return false;
        }

        private int[] aVector(KeyValuePair<string, int> keyValuePair)
        {
            string cadena = keyValuePair.Key;
            int[] aux = new int[cadena.Length];
            for (int i = 0; i < cadena.Length; i++)
                aux[i] = int.Parse(cadena[i].ToString());
            return aux;
        }

        private int[] nextColumn(int[] vectorEstadoActual, char b)
        {
            int[] _S = new int[vectorEstadoActual.Length];
            _S[0] = 0;
```

```
            for (int i = 1; i <= patron.Length; i++)
            {
                _S[i] = (patron[i - 1] == b) ? vectorEstadoActual[i - 1] :
menor(vectorEstadoActual[i - 1], _S[i - 1], vectorEstadoActual[i]) + 1;
                if (_S[i] > errores + 1)
                    _S[i] = errores + 1;
            }
            return _S;
        }

        private string aCadena(int[] S)
        {
            string aux = string.Empty;
            for (int i = 0; i < S.Length; i++)
                aux = aux + S[i];
            return aux;
        }

        private HashSet<char> setAlfabeto(string P)
        {
            HashSet<char> aux = new HashSet<char>();
            for (int i = 0; i < P.Length; i++)
                aux.Add(P[i]);
            aux.Add('þ');
            return aux;
        }

        private int menor(int a, int b, int c)
        {
            if (a < b)
            {
                if (a < c)
                    return a;
                else
                    return c;
            }
            else
            {
                if (b < c)
                    return b;
                else
                    return c;
            }
            return 0;
        }
    }
}
```

Código de implementación algoritmo de Andrejkova.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AndrejkovaAlmarimiMahmoudConsole
{
    class Program
    {
        static void Main(string[] args)
        {

            System.IO.StreamReader file = new
System.IO.StreamReader(@"D:\Documentos\Proyectos\Visual Studio\Textos\protein -
hs.txt");

            string P = Console.ReadLine();
            int k = 0;

            DateTime tiempostart = DateTime.Now;
            // Creacion del Automata
            Automata AAM = new Automata(P, k);
            DateTime tiempofin = DateTime.Now;
            TimeSpan time1 = new TimeSpan(tiempofin.Ticks - tiempostart.Ticks);

            DateTime start = DateTime.Now;

            while (!file.EndOfStream)
            {
                Console.WriteLine("empezando!");
                string T = file.ReadToEnd();
                start = DateTime.Now;
                for (int i = 0; i < T.Length; i++)
                {
                    AAM.transicion(T[i]);
                    AAM.estadoFinal(i);
                }
            }
            DateTime fin = DateTime.Now;
            TimeSpan time2 = new TimeSpan(fin.Ticks - start.Ticks);


Console.WriteLine("============================================================");
            Console.WriteLine("construcción de autómata: " + time1.ToString());

Console.WriteLine("============================================================");

Console.WriteLine("============================================================");
            Console.WriteLine("recorrido del texto: " + time2.ToString());
```

```
Console.WriteLine("============================================================");
            TimeSpan total = new TimeSpan(time1.Ticks + time2.Ticks);

Console.WriteLine("============================================================");
            Console.WriteLine("tiempo total: " + total.ToString());

Console.WriteLine("============================================================");
            //AAM.n();
            Console.ReadKey();
        }
    }
}



using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AndrejkovaAlmarimiMahmoudConsole
{
    class Automata
    {
        string P;
        int m;
        int k;
        int num = 0;
        double[] estadoActual;
        HashSet<char> alfabeto;
        Dictionary<string, double> fuzzyTrans;
        double[] F;
        Dictionary<string, int> tKMP;
        public Automata(string patron, int errores)
        {
            P = patron;
            k = errores;
            m = P.Length;
            estadoActual = new double[m + 1];
            setAlfabeto();
            setKMP();
            estadoInicial();
            F = new double[m + 1];
            estadosFinales();
            setFuzzyTransitions();
        }
        public void transicion(char b)
        {
            b = (alfabeto.Contains(b)) ? b : 'þ';
            double[] aux = new double[estadoActual.Length];
            for (int p = 0; p < estadoActual.Length; p++)
            {
```

```
                double mayor = 0;
                for (int q = 0; q < estadoActual.Length; q++)
                {
                    string cad = q + "," + p + b.ToString();
                    double val = (estadoActual[q] < fuzzyTrans[cad]) ?
estadoActual[p] : fuzzyTrans[cad];
                    if (mayor < val)
                        mayor = val;
                }
                aux[p] = mayor;
            }
            for (int i = 0; i < estadoActual.Length; i++)
            {
                estadoActual[i] = aux[i];
            }
        }
        public void estadoFinal(int pos)
        {
            double mayor = 0;
            for (int i = 0; i < estadoActual.Length; i++)
            {
                double val = (estadoActual[i] < F[i]) ? estadoActual[i] : F[i];
                if (mayor < val)
                    mayor = val;
            }
            if (mayor > 0.5f)
            {
                Console.WriteLine("Posible coincidencia en: " + pos + " con
valor: " + mayor);
                //num++;
            }
        }
        public void n()
        {
            Console.WriteLine(num);
        }
        private void estadosFinales()
        {
            for (int i = 0; i <= m; i++)
                F[i] = 0;
            F[m] = 1;
            //double[] e = { 1, 0.91f, 0.86f, 0.74f, 0.65f, 0.58f };
            //double[] e = { 1, 0.86f, 0.74f, 0.65f };
            /*
            for (int i = 1; i <= k; i++)
            {
                F[m - i] = e[i];
            }
            */
        }
        private void estadoInicial()
        {
            estadoActual[0] = 1;
            for (int i = 1; i < m + 1; i++)
```

```
        estadoActual[i] = 0;
}
private void setKMP()
{
    tKMP = new Dictionary<string, int>();
    int[] prefixVec = new int[m];

    prefixVec[0] = 0;
    int k = 0;
    for (int q = 1; q < m; q++)
    {
        while (k > 0 && P[k] != P[q])
            k = prefixVec[k - 1];
        if (P[k] == P[q])
            k++;
        prefixVec[q] = k;
    }
    int falla = 0;
    int estados = m + 1;
    for (int i = 0; i < estados; i++)
    {
        foreach (char x in alfabeto)
        {
            if (i != m)
            {
                if (x == P[i])
                    tKMP.Add(i + x.ToString(), i + 1);
                else
                    tKMP.Add(i + x.ToString(), falla);
            }
            else
                tKMP.Add(i + x.ToString(), 0);
        }
        if (i != m)
            falla = prefixVec[i];
    }
}
private void setFuzzyTransitions()
{
    fuzzyTrans = new Dictionary<string, double>();
    for (int i = 0; i < m + 1; i++)
    {
        for (int j = 0; j < m + 1; j++)
        {
            foreach (char a in alfabeto)
            {
                double mayor = 0;
                int conjuncion;
                foreach (char x in alfabeto)
                {
                    int t_ = (tKMP[i + x.ToString()] == j) ? 1 : 0;
                    double f = (x == a) ? 1 : 0.3f;
                    double val = (t_ < f) ? t_ : f;
                    if (mayor < val)
```

```
                                mayor = val;
                            }
                            fuzzyTrans.Add(i + "," + j + a.ToString(), mayor);
                        }
                    }
                }
            }
            private void setAlfabeto()
            {
                alfabeto = new HashSet<char>();
                for (int i = 0; i < P.Length; i++)
                    alfabeto.Add(P[i]);
                alfabeto.Add('þ');
            }
        }
    }
```