UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA APROXIMADA PERMITIENDO ERRORES

CLAUDIO ANDRÉS TELHA CORNEJO

2007

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA APROXIMADA PERMITIENDO ERRORES

CLAUDIO ANDRÉS TELHA CORNEJO

COMISIÓN EXAMINADORA		CALIFICACIONES: NOTA (n°)	(Letras)	FIRMA
PROFESOR GUÍA MARCOS KIWI	:		.....	.....
PROFESOR CO-GUÍA GONZALO NAVARRO	:		.....	.....
PROFESOR INTEGRANTE SR.	:		.....	.....
NOTA FINAL EXAMEN DE GRADO	:		.....	.....

TESIS PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL MATEMÁTICO Y AL GRADO DE MAGÍSTER EN CIENCIAS,
MENCIÓN COMPUTACIÓN.

SANTIAGO - CHILE
OCTUBRE - 2007

RESUMEN DE LA TESIS PARA OPTAR AL
TÍTULO DE INGENIERO CIVIL MATEMÁTICO
Y GRADO DE MAGÍSTER EN CIENCIAS,
MENCIÓN COMPUTACIÓN
POR: CLAUDIO ANDRÉS TELHA CORNEJO
FECHA: 25 DE SEPTIEMBRE DE 2006
PROF. GUÍA: MARCOS KIWI

BÚSQUEDA APROXIMADA PERMITIENDO ERRORES

El problema de la búsqueda aproximada en texto consiste en buscar las ocurrencias de un patrón en un texto, permitiendo que las ocurrencias no sean necesariamente copias exactas del patrón, sino que sean suficientemente próximas de acuerdo a alguna métrica particular. El problema tiene gran importancia en áreas como recuperación de la información, biología computacional y bases de datos de texto.

El objetivo de este trabajo es estudiar este problema en un contexto de recursos muy limitados. En este trabajo supondremos que no hay tiempo suficiente para leer completamente el texto y entregar una respuesta. Esto tiene mucho sentido en la actualidad: desde el punto de vista práctico, la existencia de bases de datos de tamaños muy grandes ha motivado el desarrollo de algoritmos que evitan la lectura completa de la entrada por tomar tiempos inaceptables. Desde el punto de vista teórico, la comprensión de lo que puede determinarse a tiempo sublineal en el tamaño de la entrada es tema de investigación reciente pero muy activa.

El aspecto innovador de la tesis es enfrentar el problema de búsqueda utilizando una formulación débil que tiene potenciales aplicaciones prácticas y admite soluciones más eficientes que aquellas que se obtienen para el problema original, a cambio de posibles errores en la respuesta. Denominamos nuestra formulación búsqueda aproximada permitiendo errores.

La principal contribución de este trabajo es la introducción y definición formal del problema de búsqueda aproximada permitiendo errores para el caso en-línea, es decir, cuando se asume que no hay tiempo o espacio suficiente como para preprocesar el texto. Se presentan algoritmos para esta formulación, apoyados por análisis teóricos y experimentales que permiten entender su competitividad con respecto a algoritmos tradicionales para búsqueda aproximada.

El trabajo se complementa con algunas extensiones de las ideas desarrolladas para el caso en línea a otros problemas relacionados. En particular, se estudia como adaptar las ideas planteadas al problema de búsqueda aproximada de múltiple, donde varios patrones se buscan sobre un mismo texto y a la búsqueda fuera de línea, en la cual se permite preprocesar el texto. Ambas extensiones muestran la robustez de los conceptos introducidos en este trabajo.

frase:

Este trabajo está dedicado a

Índice general

Capítulo 1

Introducción

Hasta hace algunos años atrás, encontrar un algoritmo para un problema a tiempo lineal en el tamaño de la entrada se consideraba óptimo. En realidad, cuesta imaginarse algo mejor, pues pareciera que ante cualquier problema no trivial lo mínimo que se tiene que hacer es leer la entrada completa. Sin embargo, la creciente demanda por procesar datos cada vez más grandes motiva el cuestionamiento de qué es lo que se puede determinar algorítmicamente en tiempo sublineal.

Aquí nos preguntamos: ¿Qué se puede hacer en tiempo sublineal para el problema de buscar una frase dentro de un documento?

1.1. Motivación

1.1.1. Una introducción a la búsqueda aproximada en texto

El problema de la *búsqueda exacta en texto* consiste en encontrar las apariciones de un patrón (corto) dentro de un texto (largo). Por ejemplo, una búsqueda del patrón `abra` en el texto `abracadabra` arrojaría dos ocurrencias, mientras que una búsqueda de `cabra` en el mismo texto no arrojaría ninguna ocurrencia. Este problema es un clásico de las ciencias de la computación.

Una variante interesante del problema anterior es la *búsqueda aproximada en texto*, que consiste en buscar todas las apariciones de un patrón en un texto, pero permitiendo que el calce no sea exacto. Por ejemplo, si ahora hacemos una búsqueda de `cabra` en el texto `abracadabra`, pero permitiendo que a lo más una letra del patrón no se corresponda con el texto, encontraríamos dos ocurrencias: `cabra` con `abra` y `cabra` con `dabra`.

Búsqueda aproximada es de utilidad cuando no se puede confiar en la exactitud del texto y/o el patrón. Por ejemplo:

- **Búsqueda en textos con errores de tipeo, ortografía o de reconocimiento óptico de caracteres**: variantes de este problema podemos citar muchas, tales como la búsqueda en la Web o la búsqueda en libros digitalizados vía reconocimiento

óptico[1] [TNB04]. En ambos casos, si una palabra no está correctamente ingresada/recuperada, ningún método de búsqueda exacta podría encontrarla. También existe el problema inverso: la palabra puede estar correctamente ingresada, pero la búsqueda puede estar mal escrita, como típicamente ocurre al buscar algún nombre de origen extranjero.

Las aplicaciones de este tipo no están restringidas sólo a efectuar búsqueda directas. Corregir errores de ortografía con algún procesador de texto es una aplicación típica en la que se utiliza búsqueda aproximada. De hecho, podemos ver el problema de corregir ortográficamente una palabra como encontrar la palabra de un diccionario más similar a ella.

Otro nicho importante de aplicaciones de búsqueda aproximada se encuentra en el área de Recuperación de la Información [BYRN99, FPS97, ZD96], que trata de la búsqueda de información o contenido relevante en textos de gran tamaño (por ejemplo, la Web). Debido a corrupción en el texto, las herramientas de búsqueda aproximada son muy utilizadas en esta área.

- **Búsqueda de secuencias de ADN**: la búsqueda de secuencias específicas de ADN (o proteínas) en otras secuencias aún más largas es una operacion fundamental en biología computacional. Podemos representar una secuencia de ADN como un texto sobre el alfabeto {`A,C,G,T`}, de modo que el problema de búsqueda de secuencias se puede modelar como encontrar las apariciones de un patrón en un texto. Sin embargo, realizar esto en forma exacta no resulta ser práctico debido, entre otras razones, a las alteraciones que sufren las secuencias debido a ciertas mutaciones. Para manejar esto, una alternativa es introducir una medida de similaridad de modo que dos secuencias, una de las cuales es una mutación de la otra, se consideren cercanas. Con esto, el problema se puede enfrentar con herramientas de búsqueda aproximada [Gus97, Wat95, CCY03].

- **Búsqueda de patrones cuando el texto o el patrón son recibidos a través de un canal que no es confiable**: la transmisión física de datos es, en general, susceptible a errores. Esto se debe a limitaciones intrínsecas de los canales de transmisión. En esta situación, si el patrón o el texto son recibidos con errores, una búsqueda exacta puede no resultar de mucha utilidad. Sin embargo, con una definición de similaridad de palabras adecuada, podemos buscar el patrón en el texto en forma aproximada y obtener, posiblemente, las ocurrencias esperadas.

Es preciso notar que para cada aplicación se debe definir una noción específica de similaridad. Por ejemplo, en textos escritos con teclado, la transposición (intercambiar dos caracteres consecutivos) es un error muy frecuente. Pero en textos escritos con lápiz y papel este error es prácticamente inexistente. Se puede decir que `cabra` y `carba` son cercanas para el primer caso, pero no así en el segundo. Por eso, la mayoría de las estrategias de búsqueda con errores incorporan una medida de distancia: para cada par de palabras existe un número indicando cuán cercanas son. Normalmente, la definición de distancia se basa en operaciones: cada error típico (transposición, por ejemplo) se ve como una operación y

[1] Más conocido como OCR.

tiene asignado un costo (más bajo mientras más frecuente es), y la distancia entre palabras se define como la suma de los costos más económica posible de una secuencia de operaciones que transforme una palabra en la otra.

1.1.2. Desafíos computacionales en búsqueda aproximada en texto

Computacionalmente hablando, la búsqueda aproximada ha sido significativamente más difícil de enfrentar que la exacta. En el caso de la búsqueda exacta, si nos restringimos a algoritmos basados en comparaciones de caracteres, una medida de complejidad es el número de comparaciones que éste realiza y ya desde fines de los setenta que se conocen algoritmos que alcanzan la cota inferior en esta medida [KMP77]. En el caso de la búsqueda aproximada, para los modelos de error más comunes el mismo logro se obtuvo mucho más tarde, a principios de los '90 [CM94]. Cabe destacar que esta área ha sido objeto de investigación continua por muchos años, y lo sigue siendo, aunque en menor medida, en la actualidad.

Por estos días, un nuevo elemento está agregando nuevos desafíos a este problema. A pesar de los buenos algoritmos existentes, el tamaño de las bases de texto actuales está llegando a volúmenes tan grandes[2] que leer el texto completo para responder una búsqueda resulta ser inaceptable. Actualmente, las soluciones a este problema típicamente caen en las siguientes categorías:

- **Algoritmos de indexado** [NBY99a, NBY00, NdMN+00, NBYST01]: se caracterizan por leer el texto una sola vez para construir una estructura (costosa de generar en tiempo y/o espacio), denominada *índice*. Ante cualquier búsqueda sobre ese texto, el algoritmo se apoya en el índice para entregar una respuesta en forma más eficiente. Esta alternativa es viable cuando se desea realizar búsquedas sobre textos que son consultados muchas veces (para amortizar el costo de la construcción del índice), y bajo el supuesto de que se dispone del texto y del tiempo necesario como para precomputar el índice.
- **Algoritmos sublineales** [CL90, Ukk92, TU93, CM94, NBY99b, BYN99]: se caracterizan por el hecho de que en una gran mayoría de las búsquedas leen el texto sólo parcialmente para responder una consulta. La respuesta suele ser entregada rápidamente en esos casos, aunque pueden ser lentos en el resto.

Sin embargo, ambas soluciones no son satisfactorias en todas las situaciones. Y aunque aún hay espacio para optimizaciones, éste cada vez es menor.

1.1.3. Un nuevo enfoque para el problema de búsqueda aproximada en texto

Desde hace mucho tiempo, varias propuestas en diferentes campos de las ciencias de la computación sugieren la búsqueda de soluciónes más débiles como alternativa a lo

[2] Por ejemplo, en el contexto de la bioinformática, la búsqueda de una secuencia particular en el ADN humano equivale a una búsqueda sobre un texto de 3Gb [CCY03].

que en su momento se considera uso "excesivo" de recursos de computador. Por ejemplo, las ρ-aproximaciones para problemas NP duros [Hoc97] son una alternativa donde no se pueden encontrar algoritmos eficientes. Otro ejemplo son los algoritmos tipo Montecarlo [Rub81, Fis96] que muchas veces ofrecen más velocidad a cambio de posibles errores en las respuestas. Lo que se define como excesivo uso de recursos varía de caso a caso, y en algunas aplicaciones esta definición tiende a ser cada vez más restrictiva con el paso del tiempo. Ello porque si bien tenemos computadores cada vez más rápidos, las demandas crecen a velocidades muy superiores. Por ejemplo, hoy en día algunas bases de datos en sí mismas resultan ser excesivamente grandes, y la búsqueda por algoritmos sublineales en el tamaño de la entrada ha empezado a cobrar importancia cuando se trata de manejarlas.

En búsqueda en texto, las técnicas para manejar los cada vez más crecientes volúmenes de datos parten del supuesto que la salida de cada algoritmo propuesto sea exactamente el conjunto de las ocurrencias buscadas. Aunque esta propiedad es del todo deseable, esto impone ciertos requerimientos de tiempo y/o espacio inaceptables en ciertas situaciones. Por ello, explorar la alternativa de utilizar algoritmos más eficientes en uso de recursos, pero que entreguen la respuesta con algún tipo de error, resulta una posibilidad interesante.

El problema de los datos de gran tamaño no es la única razón práctica para estudiar este tipo de algoritmos "débiles". En algunas aplicaciones de búsqueda en texto, una respuesta aproximada puede ser casi tan buena como una respuesta exacta. Por ejemplo, si nos interesa saber el área o categoría de cierto documento de la web mediante el conteo de las apariciones de algunas palabras clave, normalmente lo que sucede es que o bien el texto tiene escasas apariciones de esas palabras clave, o bien tiene muchas. Y saber si un patrón aparece pocas veces o muchas sólo requiere una estimación razonable del número de apariciones, algo que podría eventualmente ser entregado por un algoritmo más eficiente que uno de búsqueda aproximada.

Junto con lo anterior, en problemas muy relacionados con búsqueda en texto hay evidencia de algoritmos que comparten esta filosofía. Por ejemplo, en el problema de alineamiento de secuencias de ADN, una heurística (denominada BLAST [CM94]) se utiliza comúnmente en la práctica. Aunque existe un algoritmo exacto basado en búsqueda aproximada, este toma un tiempo inaceptable. Un segundo ejemplo es un un trabajo reciente acerca de la posibilidad de estimar débilmente la distancia de edición a tiempo sublineal en el tamaño de la entrada [BEK$^+$03].

Apoyados por lo anterior, en el Capítulo 4 definiremos una variante del problema de búsqueda aproximada, que denominaremos *búsqueda aproximada permitiendo errores*, para el cual veremos que es posible encontrar algoritmos significativamente más rápidos que los existentes para la búsqueda aproximada tradicional, a cambio de posibles errores en la respuesta.

1.2. Resumen de principales resultados

En este trabajo abordamos el problema de búsqueda aproximada en un contexto nuevo, que denominamos *búsqueda aproximada permitiendo errores*. Nuestra contribución es un estudio algorítmico de este problema.

En el Capítulo 4 se define la noción de algoritmo para búsqueda aproximada permitiendo errores, que esencialmente corresponde a un algoritmo probabilista que reporta un subconjunto de las ocurrencias aproximadas de un patrón en el texto, con cierta probabilidad mínima (en la aleatoriedad del algoritmo) de que cada ocurrencia se encuentre presente en el subconjunto entregado.

Los Capítulos 4, 5 y 6 presentan algoritmos que resuelven el problema de búsqueda en el sentido de la definición anterior en tres diferentes contextos. En el Capítulo 4 se estudia el problema de búsqueda aproximada en línea de un sólo patrón. Aquí se presenta la mayor parte de las ideas nuevas de este trabajo. En el Capítulo 5 se extienden los algoritmos obtenidos en el capítulo anterior al caso en que múltiples patrones son buscados simultáneamente en un mismo texto. Finalmente, en el Capítulo 6 se estudia la búsqueda fuera de línea, es decir, cuando se permite preprocesar el texto. Aquí se introduce una noción de búsqueda con errores algo diferente a los casos anteriores, ya que ahora el espacio utilizado por los algoritmos juega un papel fundamental.

Cada algoritmo propuesto va acompañado de un análisis que entrega algunas garantías teóricas que éstos ofrecen. En particular, se estudia el tiempo esperado de ejecución de cada algoritmo dada una cierta probabilidad de perder cada ocurrencia. En algunos algoritmos, cuando es relevante, también se estudia el espacio utilizado. El Cuadro 1.1 resume las propiedades principales.

Finalmente, el Capítulo 7 contiene una evaluación práctica de los algoritmos propuestos. Experimentalmente se mide el comportamiento práctico de un algoritmo para búsqueda con errores basado en q-gramas (**Q-PE**) y de un algoritmo para búsqueda con errores basado en la técnica de Chang y Marr (**CM-PE**).

	Algoritmo Sección	Tiempo	Tolerancia	Pérdida	Observaciones
En línea, un patrón	Caracteres lejanos 3.2	k^2n/σ	$me^{-(2k+1)\sigma}$		
	CL-PE 4.3	$kn\log m/m^{1-\alpha}$	$\log m$	$m^{-\log m}$	$\alpha \in (0,1)$ fijo
	q-gramas [Ukk92] 3.3	n	$m/\log_\sigma m$		
	Q-PE 4.4	$n\log_\sigma m/m$	$m/\log_\sigma m$	$(k\log_\sigma m/m)^t$	$t>0$ ajustable
	Chang y Marr [CM94] 3.4	$n(k+\log_\sigma m)/m$	m		
	CM-PE 4.5	$n\log_\sigma m/m$	m	$(k/m)^t$	$t>0$ ajustable
En línea múltiple	**QMP-PE** 5.2.1	$n\log_\sigma(rm)/m$	$m/\log_\sigma m$	$(k\log_\sigma(rm)/m)^t$	r número de patrones $t>0$ ajustable
	[FN04] 5.1	$n(k+\log_\sigma(rm))/m$	m		r número de patrones
	CMMP-PE 5.2.2	$n\log_\sigma m/m$	m	$(k/m)^t$	r número de patrones $t>0$ ajustable
Fuera de línea	Índice q-gramas [NBY98] 6.2	kn/σ^q	$m/\log_\sigma m$		
	Q-FL 6.2.1	$tn\log_\sigma k/\sigma^q + tnm^2\log_\sigma k/\sigma^{m/k\sigma}$	m	$1/k^t$	$t>0$ ajustable
	Q-FL2 6.2.3	$n\log k/\sigma^q + nm^2\log k/\sigma^{m/2k}$	m	$1/k$	

Tabla 1.1: Resumen de las propiedades de los algoritmos propuestos y comparación con algoritmos tradicionales similares. Se comparan los algoritmos propuestos (**CL-PE, Q-PE, CM-PE, QMP-PE, CMMP-PE, Q-FL, Q-FL2**) con respecto a algoritmos similares para búsqueda aproximada tradicional. Los algoritmos similares, es decir, basados en la misma idea, se encuentran indicados a través de filas consecutivas del mismo color. El parámetro n denota el largo del texto, m es el largo de cada patrón, k es el error permitido y σ es el tamaño del alfabeto. Todos los valores son asintóticos, se omitió la notación $O(\cdot)$ por problemas de espacio.

Capítulo 2

Conceptos y preliminares

2.1. Búsqueda aproximada en texto

En la introducción presentamos el problema de búsqueda aproximada como la búsqueda de ocurrencias aproximadas de una palabra corta (patrón) dentro de una palabra larga (texto). Ahora daremos una definición más rigurosa, adecuada para los propósitos de este trabajo.

Comenzaremos con una revisión de la notación estándar para lenguajes formales: asumiremos que las palabras utilizan siempre un conjunto de *caracteres* fijo y finito, que denominamos *alfabeto*. Usualmente, denotaremos el alfabeto por Σ y a su cardinal por σ. Las *palabras* se definen como secuencias de 0 o más caracteres y el conjunto de todas las palabras se denota por Σ^*.

Para escribir una palabra s explícitamente, utilizaremos la notación $s = s_1 s_2 \dots s_n$, $s_i \in \Sigma$. Denotamos el largo de s por $|s| = n$ y a s_i lo denominamos el *i-ésimo carácter de s*. En el caso particular de la palabra de 0 caracteres, nos referiremos a ella como ε.

Definimos también la *concatenación de palabras* como sigue: si $s = s_1 s_2 \dots s_n$ y $t = t_1 t_2 \dots t_m$, la concatenación de s y t, denotada st es la palabra $st = s_1 s_2 \dots s_n t_1 t_2 \dots t_m$. Finalmente, escribiremos $s_{i..j} = s_i s_{i+1} \dots s_j$ para referirnos a la *subpalabra* de s formada por los caracteres entre las posiciones i y j (ambas incluidas).

En el problema de búsqueda en texto aproximada, el esquema general es el siguiente:

- $d : \Sigma^* \times \Sigma^* \to \mathbb{R} \cup \{\infty\}$ es una *función de distancia* entre palabras.
- $T \in \Sigma^*$ es el *texto* donde se desea encontrar las apariciones aproximadas de un patrón. Denotamos su largo por $n = |T|$.
- $P \in \Sigma^*$ es el *patrón* a buscar. Denotamos su largo por $m = |P|$.
- $k \in \mathbb{R}$ es el máximo error permitido.

y el problema se escribe formalmente como sigue:

PROBLEMA: Búsqueda aproximada

ENTRADA: $T \in \Sigma^n$, $P \in \Sigma^m$, $k \in \mathbb{N}$ y $d(\cdot,\cdot)$

SALIDA: Posiciones j en el texto T, tal que existe i con $d(P, T_{i..j}) \leq k$.

La salida se restringe a entregar sólo las posiciones de término para asegurar que el tamaño de la respuesta sea a lo más lineal en el tamaño del texto. Esto deja de ser cierto si se entregan todos los pares (inicio, término), impidiéndonos de inmediato hablar de algoritmos lineales o sublineales para este problema.

Por lo general, utilizaremos el término *calce* u *ocurrencia* para referirnos a las subpalabras $T_{i..j}$ tal que $d(P, T_{i..j}) \leq k$ y el término *ventana de texto* para referirnos a una subpalabra de texto $T_{i..j}$ junto a sus posiciones de inicio i y de término j.[1]

En la literatura normalmente se escogen funciones de distancia $d(\cdot,\cdot)$ que evalúan el costo de convertir una palabra en la otra a través de transformaciones básicas que actúan sobre subpalabras. Por ejemplo, la operación de sustitución cambia un carácter a por otro distinto b. Si se desea tranformar `barco` en `tarso` se necesitan 2 sustituciones (`b` por `t` y `c` por `s`). Podríamos decir entonces que la distancia entre ambas palabras es 2 (suponiendo que cada sustitución cuesta 1 y que es la única operación permitida).

Más precisamente, las distancias que utilizaremos siempre estarán definidas de la siguiente forma: $d(A, B)$ es el mínimo costo de una secuencia de operaciones que transforman A en B (o ∞ si no existe secuencia alguna). El costo de una secuencia de operaciones es el costo de la suma de las operaciones individuales, donde una operación es una regla de la forma $\delta(r, s) = t$, $r \neq s$, que denota la sustitución de la subpalabra r por la subpalabra s con un costo $t \in \mathbb{N}$. Una vez que la operación ha transformado r en s, ninguna operación adicional puede ser realizada sobre s.

Las operaciones usadas más comúnmente son:

- *Inserción*: $\delta(\varepsilon, a)$, inserta el carácter a.
- *Borrado*: $\delta(a, \varepsilon)$, borra el carácter a.
- *Sustitución*: $\delta(a, b)$, con $a \neq b$, reemplaza el carácter a por el carácter b.
- *Transposición*: $\delta(ab, ba)$ con $a \neq b$, intercambia los caracteres consecutivos a y b.

y en base a ellas se definen tres de las distancias más comunes:

- *Distancia de Edición* [Lev65]: permite cualquier inserción, borrado y sustitución, todas con costo unitario.
- *Distancia de Hamming* [SK83]: permite cualquier sustitución, todas con costo unitario.

[1] No es lo mismo subpalabra que ventana. Por ejemplo, dos ventanas distintas pueden corresponder a la misma subpalabra.

- *Distancia de Subsecuencia Común más Larga* [NW70]: permite cualquier sustitución y borrado, todas con costo unitario. Se suele denotar LCS.

En los tres casos, la distancia es efectivamente simétrica (pues para cada regla $\delta(r,s) = t$ hay una regla $\delta(s,r) = t$ del mismo costo). Generalizaciones de estas distancias se obtienen asignando costos individuales (por ejemplo, asignar un costo de 1 a sustituir `s` por `z` y un costo de 2 a sustituir `s` por `b`).

En este trabajo, utilizaremos exclusivamente la distancia de edición. Esta distancia se utiliza comúnmente en la práctica (corrección ortográfica, análisis de ADN). Señalamos eso sí que muchos de los algoritmos existentes y varios de los algoritmos propuestos en este trabajo pueden adaptarse fácilmente a las otras distancias mencionadas.

2.2. Alineaciones

Cuando revisemos ciertos algoritmos de búsqueda aproximada nos serán de utilidad dos conceptos intuitivos, aunque algo complicados de definir.

El primer concepto es el de *alineación*. Supongamos que queremos transformar $A =$`abra` en $B =$`abracadabra` usando la mínima cantidad de operaciones de edición. El ejemplo es muy simple y claramente hay dos alternativas:

1. Considerar que A es el "primer" `abra` de B e insertar `cadabra` a la derecha de A.
2. Considerar que A es el "segundo" `abra` de B e insertar `abracad` a la izquierda de A.

Notar que hay muchas maneras distintas de realizar las inserciones en cada caso. En la Figura 2.1 se muestra una de esas formas para cada alternativa. Observe que al realizar cada inserción no movemos lo que hemos obtenido en el paso anterior. La alineación es el desplazamiento *desp* que hay entre el primer carácter de A y el primer carácter de B, es decir, cuanto hay que mover A para que A y B comiencen en la misma posición (con la convención de que mover hacia la derecha es positivo). Diremos también que el carácter A_i de A está alineado con el carácter B_{i-desp} de B.

```
A= abra                    abra =A
   abrac                  dabra
   abraca                adabra
   abracad              cadabra
   abracada           acadabra
   abracadab         racadabra
   abracadabr       bracadabra
B= abracadabra    abracadabra =B
```

Figura 2.1: Dos maneras de transformar A en B. La alineación en el caso de la izquierda es 0, en el caso de la derecha es -7.

El ejemplo anterior es muy simple, dado que las inserciones se realizaron siempre al final o al inicio. Cuando se realiza una inserción en posiciones intermedias podemos escoger entre mover hacia la derecha la subpalabra que comienza en la posición a insertar, o mover hacia la izquierda la subpalabra que termina en la posición a insertar. La Figura 2.2 ilustra lo anterior y también lo que ocurre en los casos de borrado y sustitución de caracteres. Una secuencia de operaciones de edición puede generar distintas alineaciones.

```
A=    azar          A= cobrar        A= acometer
      alzar            conrar           cometer
      alizar        B= contar           cometr
     nalizar                        B=  comer
    inalizar
B= finalizar
```

Figura 2.2: Tres ejemplos que ilustran el concepto de alineación. A la izquierda, sólo se realizan inserciones, con una alineación de -3; al centro, sólo se realizan sustituciones, con una alineación de 0; finalmente, a la derecha sólo se realizan borrados, con una alineación de 1.

El segundo concepto es el de *asignación de operaciones*. Consideremos una secuencia de operaciones de edición $o_1, o_2, \ldots, o_u$, con $u \leq k$, que transforman una palabra A en otra palabra B. Sea $A = A^1 A^2 \ldots A^v$ una escritura de A como concatenación de palabras. A cada A^j le asignaremos un subconjunto $O_j \subseteq \{o_1, o_2, \ldots, o_u\}$ de acuerdo al procedimiento indicado en la Figura 2.3.

$O_j = \varnothing$
for $i = 1$ **to** u
  for $j = 1$ **to** v
    if o_i es una sustitución o borrado en posiciones incluidas en A^j **or**
      o_i es una inserción en posiciones inmediatamente previas a un carácter en A^j
      then Agregar o_i a O_j y redefinir A^j como el resultado de aplicar o_i a A^j
        break
    end
  end
end

Figura 2.3: Asignación de operaciones.

Observemos que:

1. Toda operación o_i está asociada a un y sólo un conjunto O_j, es decir, los O_j forman una partición del conjunto de operaciones[2].

[2]Técnicamente, para que esto sea cierto, las inserciones realizadas después del fin del texto deben ser asociadas a A^v.

2. Para cada subpalabra A^j (previo a la ejecución del algoritmo), el resultado de aplicar las operaciones en O_j (en el suborden inducido) generan una subpalabra de B, que denominamos B^j, cuya distancia a A^j no supera el número de operaciones en O_j.

La asignación de operaciones permite descomponer las operaciones realizadas sobre el texto o el patrón, lo que es útil para demostrar ciertas propiedades simples que se utilizan en búsqueda aproximada.

2.3. Algoritmos en línea y fuera de línea

Los algoritmos para el problema de búsqueda aproximada se suelen dividir en dos grandes categorías:

- Los *algoritmos en línea* deben responder las búsquedas sin conocimiento previo sobre el texto. Normalmente este tipo de algoritmos son capaces de entregar ocurrencias a medida que se va leyendo el texto, de aquí la razón de su nombre. El tiempo de búsqueda es el tiempo total del algoritmo, aunque las labores de precómputo sobre el patrón (que se asume pequeño respecto al texto) se consideran de menor importancia a nivel de análisis.
- Los *algoritmos fuera de línea* permiten la construcción de un índice sobre el texto, es decir, una estructura que permite responder búsquedas sobre el texto de manera más eficiente. Este tipo de algoritmos está pensado para ser utilizado con consultas masivas sobre un mismo texto. La idea es que el costo de construcción del índice sobre un texto se va amortizando con el costo menor de las búsquedas posteriores. Normalmente, en el tiempo de búsqueda no se considera el costo de la construcción del índice. El costo en espacio y tiempo de construcción de este último se analiza por separado.

Este trabajo trata principalmente sobre algoritmos en línea. A menos que explícitamente se diga lo contrario, el término búsqueda aproximada lo utilizaremos únicamente para referirnos al contexto en línea de este problema.

2.4. Algoritmos para búsqueda aproximada en texto

En esta sección revisaremos algunos aspectos algorítmicos de la búsqueda aproximada. Desde comienzos de los '80, diversos enfoques se han propuesto para enfrentar este problema (ver Figura 2.4):

- Programación dinámica [Sel80, LV88, LV89, Mye86, CH98].
- Autómatas [Ukk85, Mel96, Kur96, WMM96, Nav97].
- Paralelismo de bits [BY92, WM92, BYN99, Mye99].

- Técnicas de filtrado [CL90, Ukk92, WM92, TU93, CM94, NBY99b, Shi96, BYN99, ST04]).

A continuación veremos los fundamentos de cada una de estas técnicas, para futuras referencias. Para una revisión más detallada, el lector puede consultar [Nav01].

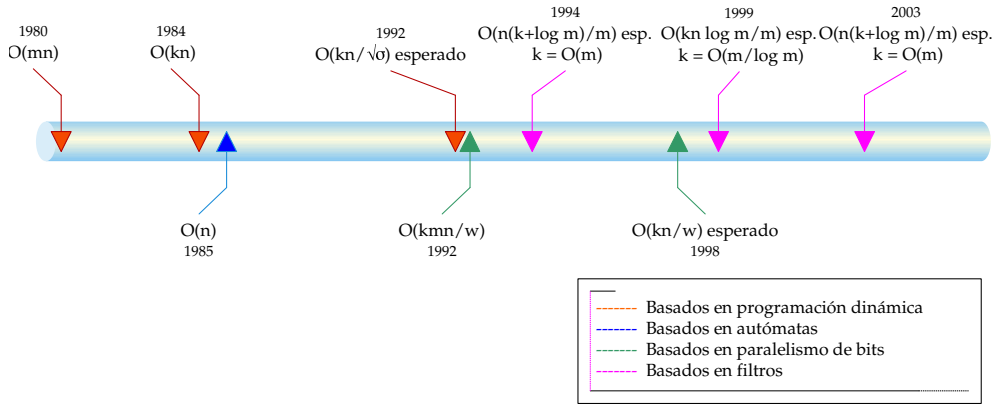

Figura 2.4: Contexto histórico. Algunos algoritmos representativos de búsqueda aproximada, divididos por categoría.

2.4.1. Programación dinámica

Comenzaremos revisando la más antigua de las técnicas para búsqueda aproximada. Los orígenes de esta técnica son algoritmos clásicos para calcular la distancia de edición y LCS. En esta sección veremos el método básico [Sel80] y señalaremos algunas extensiones.

Para $i = 1, \ldots, m$; $j = 1, \ldots, n$, definamos $D_{i,j}$ como la distancia más pequeña entre $P_{1..i}$ y alguna subpalabra de T terminada en la posición j. Observemos que los $D_{i,j}$ permiten resolver el problema de búsqueda aproximada directamente: si $D_{m,j} \leq k$, entonces existe una subpalabra de T terminada en j cuya distancia al patrón es a lo más k y viceversa. Normalmente, los $D_{i,j}$ se ven como los elementos de una matriz D, de m filas y n columnas.

El primer método para calcular la matriz D se basa en la idea siguiente: nos proponemos buscar la secuencia más corta de operaciones que transforma $P_{1..i}$ en alguna subpalabra de T terminada en j. Dividamos la búsqueda en dos casos, y utilicemos recursividad como sigue:

- Si $P_i = T_j$, entonces es claro que $D_{i,j} = D_{i-1,j-1}$: a partir de una secuencia óptima para $D_{i,j}$ se obtiene una secuencia óptima para $D_{i-1,j-1}$ del mismo costo.
- Si $P_i \neq T_j$, entonces de algún modo debemos reparar el hecho que el último carácter de las palabras sean distintos. Esto requiere siempre alguna de estas tres operaciones:
  1. Sustituir el i-ésimo carácter de $P_{1..i}$ por T_j. Si se ejecuta esta operación, el problema se reduce ahora a encontrar una transformación óptima entre $P_{1..i-1}$

y alguna subpalabra de T terminada en la posición $j-1$. El costo óptimo sería $1 + D_{i-1,j-1}$.

2. Insertar T_j al final de $P_{1..i}$. Si se ejecuta esta operación, el problema se reduce ahora a encontrar una transformación óptima entre $P_{1..i}$ y alguna subpalabra de T terminada en la posición $j-1$. El costo óptimo sería $1 + D_{i,j-1}$.
3. Remover el i-ésimo carácter de $P_{1..i}$. Si se ejecuta esta operación, el problema se reduce ahora a encontrar una transformación óptima entre $P_{1..i-1}$ y alguna subpalabra de T terminada en la posición j. El costo óptimo sería $1 + D_{i-1,j}$.

De lo anterior se desprende la siguiente recurrencia:

$$D_{i,j} = \begin{cases} D_{i-1,j-1}, & \text{si } P_i = T_j \\ 1 + \text{mín}\{D_{i-1,j-1}, D_{i,j-1}, D_{i-1,j}\}, & \text{en caso contrario.} \end{cases}$$

Es claro que si conocemos la primera fila y la primera columna de la matriz D, el resto de sus valores se puede calcular utilizando únicamente la fórmula anterior. Sin embargo, por simplicidad es preferible añadir una fila y columna adicional definida por:

$$D_{0,j} = 0, \quad j = 0, \ldots, n,$$

$$D_{i,0} = i, \quad i = 0, \ldots, m.$$

Es fácil ver que esta inicialización, junto a la recurrencia, permiten obtener los valores $D_{i,j}$. Esto justifica la correctitud del algoritmo descrito en la Figura 2.5 para resolver el problema de búsqueda aproximada.

Claramente el algoritmo se ejecuta en tiempo $O(mn)$ en el peor caso y se puede implementar con pequeñas modificaciones para que utilice espacio $O(m)$ (basta guardar únicamente la última columna calculada).

Aprovechándose de propiedades más complejas de la matriz D, posteriormente se encontraron maneras más eficientes de calcular sus coeficientes. Completando la matriz por diagonales, el algoritmo de Landau y Vishkin [LV89] calcula la matriz D en $O(kn)$ utilizando $O(n)$ espacio. Otra variante [LV88] calcula la matriz en $O(k^2 n)$ utilizando $O(m)$ espacio. Todos los análisis señalados son de peor caso. A pesar de ser buenos resultados teóricos, estos algoritmos (y muchos otros basados en programación dinámica) no son competitivos en la práctica.

2.4.2. Autómatas

Los *algoritmos basados en autómatas* generan un autómata finito no determinista a partir del el patrón, que luego se utiliza para leer el texto. El autómata detecta las posiciones que son término de algún calce a través de sus estados.

Veamos un ejemplo simple. Para buscar la palabra `volver` en un texto cualquiera con a lo más 2 errores, utilizaremos el autómata de la Figura 2.6. La idea central es que cada fila representa el número de errores (si nos encontramos en la fila i es porque hemos

```
proc PROG-DIN (T, P, n, m, k)
  comment: T = texto; P = patrón, n = |T|, m = |P|, k = error.
  for i = 1 to m
      D_{i,0} = i
  end
  for j = 1 to n
      D_{0,j} = 0
  end
  for j = 1 to n
      for i = 1 to m
          if P_i = T_j
            then D_{i,j} = D_{i-1,j-1}
            else D_{i,j} = 1 + mín{D_{i-1,j-1}, D_{i,j-1}, D_{i-1,j}}
          end
      end
  end
  return {j tal que D_{m,j} ≤ k}
end
```

Figura 2.5: Algoritmo de programación dinámica.

encontrado $i-1$ errores) y cada columna representa el hecho de tener un calce contra un prefijo distinto del patrón (si nos encontramos en la columna j es porque hemos logrado hacer calzar los $j-1$ primeros caracteres del patrón).

Las flechas representan principalmente las operaciones que utilizamos para transformar el patrón (en realidad un prefijo cada vez más grande de él) en el texto: una flecha hacia la derecha indica que los caracteres del texto y patrón coinciden y por lo tanto podemos avanzar en el texto y el patrón sin aumentar el número de errores (sin aplicar ninguna operación). Las flechas verticales indican inserción de un carácter (avanzamos en el texto pero no en el patrón). Las flechas en diagonal representan el borrado de un carácter (avanzamos en el patrón, pero no en el texto) o una sustitución de un carácter (avanzamos en patrón y texto). El loop inicial permite que un calce empiece en cualquier parte del texto y los estados finales indican que la posición del carácter recién leído es una posición de término de un calce.

Para traducir esta idea a un algoritmo, una alternativa es transformar el autómata a su versión determinista y luego ejecutar el nuevo autómata sobre el texto. Claramente esto hace que el recorrido del texto buscando los calces se ejecute en tiempo $O(n)$ (óptimo en el peor caso), pero a costa de utilizar espacio $O(\text{mín}(3^m, m(2m\sigma)^k))$ para los estados del autómata [Ukk85][3]. Esto lo hace inaplicable en la práctica. Otras alternativas reducen el espacio significativamente ($O(mn)$ en [Kur96]), pero a costa de mezclar la ejecución del autómata con su construcción[4].

[3] En particular, esto significa que el tiempo de construcción es al menos de ese orden.

[4] Esto es algo que tiene relevancia: cuando un proceso depende únicamente del patrón y éste se conoce

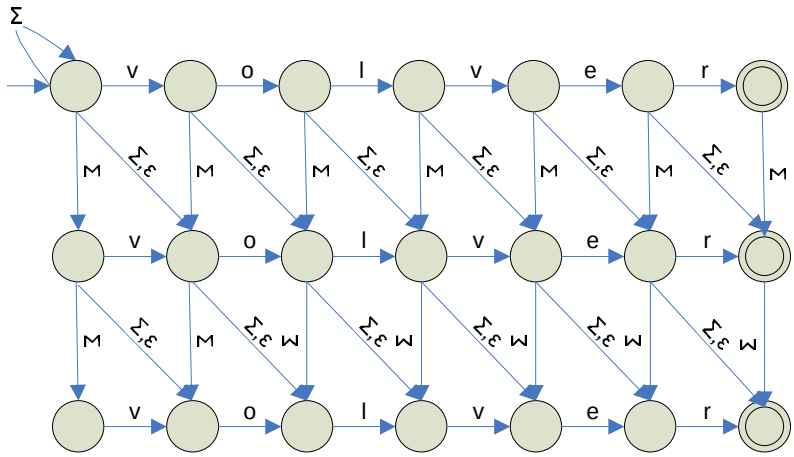

Figura 2.6: Autómata para la búsqueda del patrón `volver` con no más de dos errores.

2.4.3. Paralelismo de bits

Los algoritmos basados en *paralelismo de bits* reducen el tiempo de ejecución de algoritmos ya existentes en un factor de hasta w, donde w es el tamaño de palabra del computador. El valor de w es de 32 o 64 en las arquitecturas actuales, por lo que la reducción es significativa en la práctica. Básicamente, la idea es realizar alguna sección de un algoritmo en forma paralela utilizando operaciones básicas con bits.

La siguiente notación adicional nos será útil (sólo en esta sección):

- Para un bit b y un entero positivo n, b^n denota el número binario obtenido al concatenar n veces el bit b.
- Los símbolos $\wedge$ y $\vee$ denotan los operadores binarios AND y OR, respectivamente.
- El símbolo $\gg$ denota el operador SHIFT-RIGHT. Si $B = b_1 b_2 \dots b_n$ es un número binario y r un entero positivo, entonces, $B \gg r = 0^r b_1 b_2 \dots b_{m-r}$.

Ilustraremos la idea básica del método de paralelismo de bits aplicándola a un algoritmo basado en autómatas para búsqueda exacta. Supongamos que queremos buscar el patrón $P =$ `volver` en un texto cualquiera. Entonces podemos construir el autómata no determinista de la Figura 2.7 para buscar ese patrón en forma exacta. Representaremos la ejecución no determinista del autómata a través de bits. Más precisamente, representaremos los estados del autómata (excepto el estado inicial) como un número binario de m bits $D = d_1 d_2 \dots d_m$, cuyo i-esimo bit es 1 si y sólo si el estado i se encuentra activo. Para cada

de antemano, es posible ejecutar este proceso de inmediato. Luego, cuando el texto se encuentre disponible, se puede ejecutar el resto del algoritmo. Otra ventaja es que, si buscamos un mismo patrón en diferentes textos el proceso de creación del autómata se realiza una sola vez para todas las búsquedas. Usualmente denominaremos procesos de precómputo a rutinas que dependen únicamente del patrón y les adjudicaremos una importancia menor que a los procesos que actúan sobre el texto.

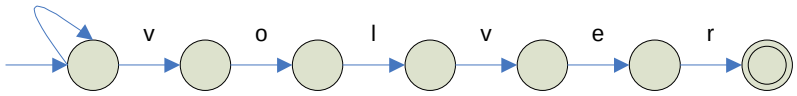

Figura 2.7: Autómata para la búsqueda del patrón `volver` en forma exacta.

carácter $c \in \Sigma$ construimos un número binario B_c de m bits, tal que su i-esimo bit es 1 si y sólo si $P_i = c$.

Inicialmente, $D = 0^m$ (sólo el estado inicial se encuentra activo). Además, si D representa los estados del autómata tras leer los primeros $j-1$ caracteres del texto, entonces:

$$((D \gg 1) \vee 10^{m-1}) \wedge B_{T_j}$$

es la representación de los estados del autómata tras leer el carácter j-ésimo del texto. En efecto, tras leer el carácter j del texto, el i-esimo estado del autómata se activa si y sólo si el $i-1$-ésimo estado se encontraba activo y el carácter T_j coincide con el carácter del patrón en la posición i.

El modelo permite simular las transiciones simúltaneas del autómata no determinista utilizando sólo dos operaciones binarias, con lo que la ejecución del autómata sobre un texto de largo n se puede realizar en $O(n)$, si es que el autómata se puede representar en una sola palabra de computador. En el caso general, es posible representar el autómata con $O(m/w)$ palabras de computador.

Aunque el ejemplo es para búsqueda exacta, la misma idea se puede aplicar para paralelizar los algoritmos basados en autómatas de la sección anterior [WM92]. También es posible (aunque con una idea diferente a la explicada aquí) paralelizar el cálculo de valores de la matriz de programación dinámica [Mye99].

2.4.4. Filtros

Supongamos que queremos buscar en forma exacta el patrón

$$P = \texttt{cabra}$$

en el texto

$$T = \texttt{abradabradacabra}.$$

Un enfoque simple es ver si P es igual o no a cada una de las 12 palabras de 5 letras consecutivas que se forman en T: `abrad`, `brada`, `radab`, ... (las denominaremos *subpalabras*). Observemos que son revisados al menos 12 caracteres distintos del texto, si se comparan las subpalabras de izquierda a derecha y se ignoran las subpalabras apenas se encuentra alguna diferencia con el patrón. Notemos que en general, si el texto tiene n caracteres y el patrón tiene m caracteres, entonces esta estrategia requiere revisar al menos $n-m+1$

caracteres diferentes del texto. Si m es pequeño en relación a n, entonces esta estrategia es esencialmente tan costosa como leer el texto completo.

Pero basta observar que la primera letra `d` del texto T no se encuentra en el patrón P para garantizar inmediatamente que el patrón no es igual a ninguna de las 5 primeras subpalabras del texto. De la misma manera, la segunda letra `d` impide que el patrón sea igual a alguna de las 5 siguientes subpalabras del texto. En este caso, sólo se revisaron dos caracteres del texto para descartar 10 de las 12 posibles subpalabras. Aunque el ejemplo es engañoso (ningún algoritmo podría adivinar que las posiciones donde hay letras `d` es lo que hay que revisar), si ilustra algo muy significativo: para garantizar que un patrón no se encuentra en un pedazo del texto no es necesario revisarlo completamente. Notar que no es cierto el reverso: para garantizar que el patrón se encuentra en una cierta posición del texto, hay que revisar todos los caracteres de la subpalabra empezando en dicha posición[5].

Aunque el ejemplo anterior es para búsqueda exacta, los *algoritmos de filtrado* para búsqueda aproximada se aprovechan de una idea similar: es más fácil garantizar que en un pedazo de texto no aparece el patrón en forma aproximada que garantizar que se encuentra. Esto se utiliza para descartar rápidamente porciones del texto donde no es posible que el patrón se encuentre en forma aproximada.

Es importante destacar que los algoritmos que utilizan esta técnica normalmente son incapaces de detectar las apariciones del patrón en el texto, sólo reducen la porción de texto donde se debe buscar. Por eso, cada algoritmo de filtro se utiliza en conjunto con un algoritmo de búsqueda aproximada (en este contexto se denomina *algoritmo verificador*) que busca apariciones del patrón en las porciones de texto no descartadas por el filtro. Podemos ver una idea del proceso completo en la Figura 2.8.

Resulta evidente que en ciertos casos el proceso completo de filtrado y verificación puede ser más costoso que ejecutar directamente el verificador sobre el texto completo: si el texto está lleno de ocurrencias del patrón (por ejemplo, $P =$ `abra`, $T =$ `abrabrabra`), el filtro no puede descartar ninguna parte del texto y por lo tanto es sólo un sobrecosto. Por eso, el análisis de este tipo de algoritmos se hace sólo para el caso promedio, asumiendo que el texto es generado bajo una cierta distribución de probabilidad y mostrando que bajo esa distribución el filtro "normalmente" aumenta la velocidad del algoritmo completo, es decir, filtro y verificador.

Los filtros son sublineales en promedio, descartando porciones de texto leyendo muy pocos de sus caracteres. Sin embargo, en el peor caso (es decir, para ciertos textos y patrón específicos) los filtros revisan todos los caracteres.

2.5. Extensiones al problema de búsqueda aproximada en texto

Muchas extensiones o problemas relacionados se han propuesto en búsqueda aproximada. En realidad la palabra extensión puede resultar algo mezquina por cuanto algunos de

[5] Asumiendo que no se sabe nada del texto excepto los caracteres consultados.

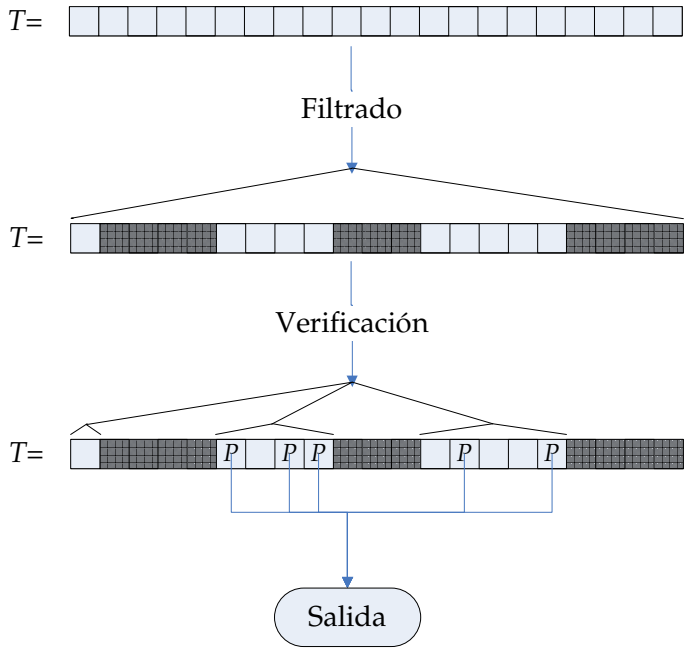

Figura 2.8: Esquema de un algoritmo basado en filtros: durante el proceso de filtrado, ciertas áreas de texto (en negro) son descartadas. Luego, el algoritmo verificador se ejecuta sobre las áreas no descartadas, reportando las ocurrencias (marcadas con la letra P).

estos problemas tienen vida propia. Parte de este trabajo podría eventualmente extenderse a este tipo de problemas:

- Alineamiento local de secuencias (biología). La similitud de dos secuencias de proteínas puede verse como la búsqueda de alineamientos locales, es decir, la búsqueda de subsecuencias que son de algún modo parecidas. Las medidas de similitud son parecidas a la distancia de edición.
- Búsqueda aproximada de múltiples patrones, en la cual se desean encontrar las ocurrencias de varios patrones en un mismo texto. Veremos algunos aspectos de este tema en el Capítulo 5.
- Búsqueda con caracteres comodines. Es el problema de búsqueda aproximada, permitiendo un carácter especial "*" que puede representar a cualquier carácter.

2.6. Herramientas probabilistas

Para demostrar las garantías probabilistas que alcanzarán los algoritmos propuestos en este trabajo, utilizaremos frecuentemente cotas para la *distribución binomial de probabilidades*. El lector no familiarizado con este tema puede consultar [Fel68]. Esta sección solamente presenta la notación y resultados que necesitaremos en este trabajo.

Para una moneda cuya probabilidad de obtener cara es p, consideremos el experimento de lanzarla n veces de manera independiente. Denotemos por X_i a la variable indicadora del resultado obtenido en cada lanzamiento, es decir:

$$X_i = \begin{cases} 1, & \text{si el } i\text{-ésimo lanzamiento sale cara,} \\ 0, & \text{si el } i\text{-ésimo lanzamiento sale sello.} \end{cases}$$

Se define la *distribución binomial de parámetros n y p*, denotada por $\text{Bin}(n,p)$, como la variable aleatoria que cuenta el número de caras obtenidas, es decir,

$$\text{Bin}(n,p) = \sum_{i=1}^{n} X_i.$$

Algunos resultados básicos respecto a esta variable aleatoria son los siguientes:

- La *esperanza* o *valor esperado* de la binomial, que denotaremos comúnmente por μ o $\mathbb{E}(\text{Bin}(n,p))$ vale $\mu = np$.
- $\mathbb{P}(\text{Bin}(n,p) \leq k) = \mathbb{P}(\text{Bin}(n,1-p) \geq n-k)$.

A lo largo de este trabajo frecuentemente necesitaremos acotar superiormente la probabilidad que el resultado del experimento binomial se encuentre lejos de la media. Específicamente, necesitaremos acotar superiormente probabilidades de eventos de la forma $\text{Bin}(n,p) \leq k$ con $k \leq \mu$, o $\text{Bin}(n,p) \geq k$ con $k \geq \mu$. Para ello utilizaremos las siguientes desigualdades, conocidas como cotas de Chernoff [Che52] .

Teorema 2.6.1. *Cotas de Chernoff para la distribución binomial*

$$\mathbb{P}(\text{Bin}(n,p) \geq k) \leq e^{k-\mu}\left(\frac{\mu}{k}\right)^k, \qquad \forall k > \mu,$$

$$\mathbb{P}(\text{Bin}(n,p) \leq k) \leq e^{k-\mu}\left(\frac{\mu}{k}\right)^k, \qquad \forall 0 < k \leq \mu.$$

Aunque más débiles que las cotas anteriores, por simplicidad a veces utilizaremos:

Teorema 2.6.2. *Cotas de Chernoff simplificadas para la distribución binomial*

$$\mathbb{P}(\text{Bin}(n,p) \geq (1+\delta)\mu) \leq 2^{-\delta\mu}, \qquad \forall \delta > 0,$$

$$\mathbb{P}(\text{Bin}(n,p) \geq (1+\delta)\mu) \leq e^{-\delta^2\mu/2}, \qquad \forall 0 < \delta < 2e-1,$$

$$\mathbb{P}(\text{Bin}(n,p) \leq (1-\delta)\mu) \leq e^{-\delta^2\mu/2}, \qquad \forall 0 \leq \delta < 1.$$

Para que $\sum_{i=1}^{n} X_i$ sea una binomial es necesario que los X_i sean independientes. Existen desigualdades muy similares a las indicadas en el Teorema 2.6.2 cuando existe "escasa" dependencia. Introduciremos la siguiente definición:

Definición 2.6.1. *Sean X_i, $i = 1, \ldots, n$ variables aleatorias. Diremos que los X_i son m-independientes si existe una familia $\{A_1, \ldots, A_m\}$ de subconjuntos de $A = \{1, \ldots, n\}$ tales que:*

1. $A = \cup_{i=1}^{m} A_i$
2. *Para cada $i = 1, \ldots, m$, las variables X_j con $j \in A_i$ son independientes.*

El siguiente resultado [Jan04] es el análogo del Teorema 2.6.2 para variables m-independientes.

Teorema 2.6.3. *Cotas para suma de variables m-independientes.*

Sean $X_1, \ldots, X_n$ variables aleatorias m-independientes a valores en $\{0, 1\}$, con $\mathbb{P}(X_i = 1) = p \in (0, 1)$ para todo i. Entonces la esperanza de $\sum_{i=1}^{n} X_i$, denotada por μ, vale $\mu = np$ y para todo $t > 0$ se tienen las siguientes desigualdades:

$$\mathbb{P}\left(\sum_{i=1}^{n} X_i \geq \mu + t\right) \leq \exp\left(-\frac{8t^2}{25m(\mu + t/3)}\right),$$

$$\mathbb{P}\left(\sum_{i=1}^{n} X_i \leq \mu - t\right) \leq \exp\left(-\frac{8t^2}{25m\mu}\right).$$

Esencialmente, todas estas desigualdades indican un decrecimiento exponencial en la cola de la distribución $\sum_{i=1}^{n} X_i$ cuando nos alejamos de la media.

2.7. Análisis promedio

En esta sección definiremos el esquema aleatorio en el que trabajaremos. El modelo de texto aleatorio que consideramos principalmente es el *Bernoulli uniforme*, o simplemente *Bernoulli* que puede verse como una variable aleatoria T, uniforme e independiente, a valores en Σ^t. Desde un punto de vista constructivo, este modelo también puede verse como un texto aletorio generado como la concatenación de t caracteres en Σ, extraídos uniformemente e independientemente.

Cuando un texto es generado según la distribución Bernoulli uniforme, lo denotaremos por $T \leftarrow \Sigma^t$. En general, utilizaremos la notación $x \leftarrow X$ para denotar cualquier elección uniforme e independiente de un elemento x en un conjunto X. El tiempo promedio de un algoritmo determinista $\mathcal{A}$ cuya entrada es un texto T de largo n y un patrón P de largo m lo definiremos como

$$\mathop{\mathbb{E}}_{T \leftarrow \Sigma^n, P \leftarrow \Sigma^m} \text{time}(\mathcal{A}, T, P), \tag{2.1}$$

donde $\text{time}(\mathcal{A}, T, P)$ denota el tiempo de ejecución del algoritmo $\mathcal{A}$ sobre el texto T y el patrón P. Ya que para la mayoría de los algoritmos que propondremos en este trabajo la aleatoriedad del patrón no es esencial en el análisis,[6] normalmente utilizaremos

[6] Esto no es cierto para algunos algoritmos de búsqueda aproximada

$$\max_{P} \mathop{\mathbb{E}}_{T \leftarrow \Sigma^n} \text{time}(\mathcal{A}, T, P)$$

como una cota superior en el tiempo promedio de ejecución del algoritmo.

En esta tesis los algoritmos propuestos serán probabilistas. El tiempo de ejecución de un algoritmo sobre una entrada ya no es único, y lo que normalmente se mide es el tiempo de ejecución esperado. Para ello, se asume que la parte aleatoria del algoritmo se puede ver como un elemento aleatorio r extraído uniforme e independientemente sobre un conjunto R y que una vez conocido el valor de r el algoritmo se vuelve determinista. El tiempo esperado de un algoritmo probabilista $\mathcal{A}$ cuya entrada es un texto T de largo n y un patrón P de largo m lo definiremos como

$$\mathop{\mathbb{E}}_{r \leftarrow R} \text{time}(\mathcal{A}, T, P, r)$$

donde $\text{time}(\mathcal{A}, T, P, r)$ denota el tiempo de ejecución del algoritmo $\mathcal{A}$ sobre el texto T y el patrón P, cuando el elemento aleatorio del algoritmo es r.

Podemos definir el tiempo promedio de ejecución de un algoritmo probabilista con la expresión 2.1, pero considerando $\text{time}(\mathcal{A}, T, P)$ como el tiempo esperado. A este tiempo lo denominaremos tiempo promedio esperado, o simplemente tiempo esperado.

Observación. En este trabajo sólo veremos algoritmos deterministas sobre texto aleatorio y algoritmos probabilistas sobre texto aleatorio. Utilizaremos la siguiente convención: los términos *tiempo promedio esperado* y *tiempo esperado* se refieren al tiempo promedio de ejecución de algoritmos probabilistas, mientras que el *tiempo promedio* se refiere al tiempo promedio de algoritmos deterministas.

Capítulo 3

Algoritmos de filtrado

En este capítulo revisaremos una familia de algoritmos para búsqueda aproximada. Dedicamos una sección completa a ellos porque son la base para los métodos que propondremos en el siguiente capítulo.

3.1. Introducción

Los *algoritmos de filtrado para búsqueda aproximada* se basan en el hecho de que puede ser mucho más fácil garantizar que en una posición del texto no existe un calce, que garantizar que existe. Por ejemplo, supongamos que en una parte del texto las palabras `abra`, `cada` y `bra` no aparecen como subpalabras. Entonces podemos estar seguros que la palabra `abracadabra` no aparece allí con 2 errores bajo la distancia de edición (cada operación de edición puede alterar a lo más una de las tres palabras antes mencionadas).

Los algoritmos de filtrado aprovechan este hecho para descartar (filtrar) partes del texto en las que no es posible que el patrón se encuentre en forma aproximada. Si queremos buscar `abracadabra` en un texto de n caracteres y las palabras `abra`, `cada` y `bra` aparecen como subpalabras sólo en los últimos 50 caracteres, podemos usar programación dinámica sobre los últimos 50+11 caracteres, y no sobre el texto completo. Esto se traduce en mayor velocidad de búsqueda si podemos efectuar la búsqueda (exacta) de las tres palabras en forma suficientemente rápida.

Para el ejemplo anterior, notemos que aunque las tres palabras aparecieran en el texto nada garantiza que exista un calce. Esto ilustra algo característico de estos algoritmos: filtrar es un proceso rápido que permite descartar algunas porciones del texto donde no hay calce, pero no permite encontrarlos. Por ello, todos los algoritmos de filtrado utilizan un algoritmo de búsqueda aproximada para encontrar las ocurrencias en todas aquellas porciones de texto que no fueron descartadas. En lo que sigue, denominaremos *filtro* al procedimiento que descarta partes del texto donde no hay calces y *rutina verificadora* o *verificador* al procedimiento que encuentra los calces. El *algoritmo de filtrado* es el proceso completo, es decir, filtrado y verificación.

Otra característica importante de los algoritmos de filtrado es que son rápidos en la

mayoría de los textos y patrones, pero siempre se pueden encontrar casos específicos en los que el filtro no es capaz de descartar ninguna porción de texto, y por lo tanto el filtro es sólo sobrecosto. Por eso, lo interesante es ver como se comportan estos algoritmos en promedio. En la literatura, el análisis de este tipo de algoritmos normalmente sigue estas directrices:

- Desde el punto de vista experimental, se consideran un conjunto de textos y patrones provenientes de aplicaciones donde búsqueda en texto aproximada tiene potenciales aplicaciones prácticas, como bases de datos de ADN o texto. Se toma un promedio del tiempo/espacio medidos.
- Desde el punto de vista teórico, el análisis que se realiza comúnmente es el de tiempo promedio bajo el supuesto de que el texto y el patrón siguen una distribución Bernoulli uniforme. Aunque el modelo es muy simple, los resultados que se obtienen a partir de él usualmente son representativos de lo que se obtiene en forma experimental.

En todo este capítulo, a menos que se indique explícitamente lo contrario, el tiempo promedio se considera bajo el supuesto de que el texto y el patrón son aleatorios siguiendo una distribución Bernoulli uniforme.

Finalmente, una característica distintiva de los algoritmos de filtrado es que su eficiencia (en tiempo esperado) es válida sólo para valores del error k no superiores a cierto umbral. A este valor se le denomina la tolerancia del filtro. Dependiendo de la capacidad de filtro, cada algoritmo tiene su propia tolerancia máxima. El problema es que a medida que k crece, el número de ocurrencias crece y descartar porciones de texto se hace más difícil. Se sabe además [Nav01] que los filtros no funcionan bien en el caso promedio cuando $k > c_\sigma m$, donde $c_\sigma < 1$ depende únicamente del tamaño del alfabeto σ.

3.2. El algoritmo de Caracteres Lejanos

En 1990, Tarhio y Ukkonen [TU93] diseñaron el primer algoritmo de filtrado. La idea básica es alinear el patrón con el texto y revisar el texto contando el número de *caracteres lejanos*. Un carácter del texto se dirá lejano si no coincide con el carácter del patrón con el que está alineado, y tampoco coincide con cualquier carácter del patrón a distancia de k caracteres o menos. Para ilustrar esto veamos un ejemplo concreto: si $T =$ `abracadabra`, $P =$ `abrir`, y $k = 2$, entonces los caracteres lejanos para algunas alineaciones entre el texto y el patrón se ilustran en la Figura 3.1:

La observación clave del algoritmo es que si un cáracter de una ocurrencia para una alineación es lejano, entonces necesariamente tenemos que borrarlo en una secuencia de operaciones de edición que transforme una posible ocurrencia (con esa alineación) en el patrón. Por lo tanto, cuando más de k caracteres lejanos han sido encontrados, la alineación puede ser descartada. En caso contrario, debe ejecutarse el algoritmo verificador.

Formalmente, consideremos una alineación j entre el texto y el patrón, es decir, T_{j+i}

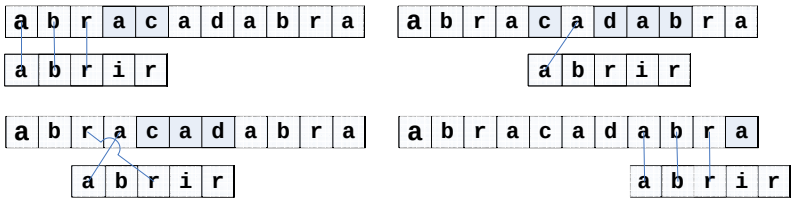

Figura 3.1: En gris oscuro, caracteres lejanos (con $k = 2$) para algunas alineaciones.

está alineado con P_i. Diremos que T_{j+i} es lejano si[1]

$$T_{j+i} \notin \{P_{i-k}, P_{i-k+1}, \ldots, P_{i+k}\}.$$

Notemos que si un carácter T_{j+i} es lejano para algún $k < i \leq m - k$, y existe una ocurrencia con esa alineación j, entonces:

- El carácter es parte de la ocurrencia.
- El carácter es borrado en cualquier secuencia de a lo más k operaciones que transforme la ocurrencia en el patrón. Ello pues no es posible alinear el carácter con ningún cáracter del patrón a distancia mayor que k.

Luego, encontrar más de k caracteres lejanos para una alineación garantiza que la alineación no puede corresponder a un calce y por lo tanto se puede descartar.[2] El filtro aprovecha esta propiedad alineando el patrón con el texto y recorriendo los caracteres antes señalados. Cuando más de k caracteres lejanos han sido encontrados, la alineación se descarta. Si lo anterior no ocurre, entonces se buscan todos los posibles calces en $T_{j-k..j+m+k}$ con un algoritmo tradicional.

Por eficiencia, la propiedad de ser carácter lejano se precomputa, es decir, para cada carácter c y cada posición i del patrón, el resultado de

$$c \notin \{P_{i-k}, P_{i-k+1}, \ldots, P_{i+k}\}$$

se almacena en una tabla. Además, para cada posición i del patrón y cada posible carácter c, el filtro almacena el valor de

$$\text{Desp}(i, c) \equiv \min_{s>0}\{P_{i-s} = c\}$$

con lo que el número de alineaciones que pueden ser saltadas se calcula de manera simple como

$$\min_{i=m-k..m} \text{Desp}(i, T_{j+i}).$$

[1] Si $\{P_{i-k}, P_{i-k+1}, \ldots, P_{i+k}\}$ contiene elementos fuera de rango, por ejemplo, P_{-1}, simplemente no se consideran parte del conjunto.

[2] Técnicamente, el filtro descarta alineaciones, lo que a su vez genera que ciertas posiciones del texto no puedan ser término de algún calce.

Con lo que hemos presentado, el filtro debe considerar todas las alineaciones. Para evitar esto, los autores del algoritmo utilizan un enfoque muy similar al algoritmo de Boyer-Moore [BM77] para búsqueda exacta en texto. No detallaremos este procedimiento aquí.

El tiempo de este algoritmo depende principalmente de la rapidez con que en promedio se encuentran más de k caracteres lejanos para una alineación. Los autores muestran que, bajo el supuesto de que el texto y el patrón son aleatorios, el tiempo promedio de ejecución del filtro es $O(k^2 n/\sigma)$. Esto no incluye el tiempo de verificación, es decir, el tiempo invertido en revisar las posiciones no descartadas. Un análisis simple en [Nav01] indica que el costo de verificación promedio es no superior al costo de filtro promedio para $k/m < e^{-(2k+1)/\sigma}$. En la práctica, aunque el algoritmo resulta competivo para valores pequeños de k, su baja tolerancia limita su aplicabilidad.

3.3. El algoritmo de q-gramas

Muchos filtros se basan en la idea de buscar piezas del patrón en el texto en forma exacta (SET [CL90], h-samples [Tak94]). En esta sección veremos uno de los más representativos de esta familia, el algoritmo de q-gramas [Ukk92].

Un q-grama es una palabra de largo q. El conjunto de q-gramas de una palabra se define como:

Definición 3.3.1. *Para una palabra V de largo v y para $q < v$, el multiconjunto*[3] Q_q^V *de q-gramas de V es:*

$$Q_q^V = \{\{V_{i..i+q-1} \mid i = 1, \ldots, v-q+1\}\}$$

contando repeticiones de elementos.

La idea básica de todos los algoritmos basados en q-gramas es la siguiente. Supongamos que P está a distancia menor o igual que k de la subpalabra $T_{i..j}$ y consideremos una secuencia de a lo más k operaciones que transformen P en $T_{i..j}$. Cada operación altera a lo más q de los q-gramas de P, por lo que al menos $m-q+1-kq$ de los q-gramas del patrón deben aparecer en cualquier ocurrencia. En lo que sigue de esta sección, denotaremos por $m' = m-q+1$ el número de q-gramas de P.

El algoritmo de q-gramas recorre cada ventana V de tamaño $m-k$ (que es el tamaño mínimo de un calce), contando el número de q-gramas del patrón presentes en V. Existen dos posibilidades:

- Si se encuentran menos de $m'-kq-2k$ de los q-gramas de Q_q^P, entonces ningún calce puede contener a la ventana. Ello porque el largo de un calce es a lo más $m+k$, por lo que como máximo $2k$ de los m' q-gramas no han sido considerados en el conteo. Diremos en este caso que la ventana es descartada por el filtro.

[3] Usaremos $\{\{\ldots\}\}$ para los multiconjuntos.

- Si se encuentran al menos $m' - kq - 2k$ de los q-gramas de Q_q^P, el filtro no descarta la ventana y todos los calces que contienen a la ventana son encontrados con un algoritmo tradicional. Ello requiere revisar calces en una ventana de texto de tamaño $m + 3k$. Diremos que la ventana es verificada por el filtro.

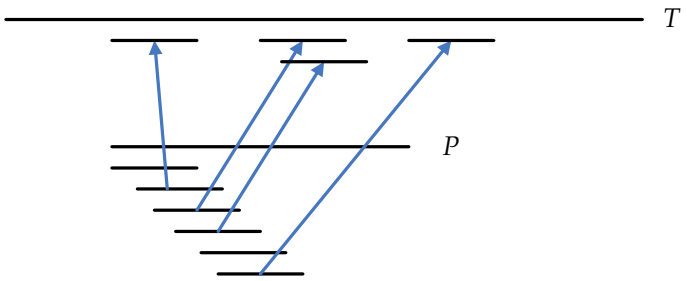

Figura 3.2: Conteo de q-gramas.

Note que no es necesario repetir el conteo de q-gramas completamente en cada ventana. En efecto, dos ventanas consecutivas sólo difieren en dos q-gramas. Más aún, utilizando un árbol de sufijos, el conteo de q-gramas (y por lo tanto el costo del filtro) puede realizarse en tiempo $\Theta(n)$. Un análisis grueso en [Nav01] muestra que para $k = O(m/\log m)$ el tiempo de verificación es a lo más tiempo de filtro (asintóticamente). Como en todo algoritmo de filtro, este análisis es bajo el supuesto de texto aleatorio.

3.4. El algoritmo de Chang y Marr

El primer resultado teórico óptimo en algoritmos de filtrado corresponde al algoritmo de Chang y Marr [CM94]. La idea base de este algoritmo es realizar una búsqueda aproximada de algunas subpalabras del texto en el patrón.

Veamos primero los fundamentos de este algoritmo. Introduzcamos la siguiente:

Definición 3.4.1. *Sean R y S dos palabras cualesquiera. Definamos* $\mathrm{asm}(R, S)$ *como la mínima distancia de la palabra R a las subpalabras de S. Es decir:*

$$\mathrm{asm}(R, S) = \min_{i,j} d(R, S_{i..j}).$$

La siguiente propiedad [NBYST01] es la base del algoritmo de Chang y Marr:

Lema 3.4.1. *Sea S una ocurrencia del patrón P en el texto T. Consideremos una partición cualquiera de la ocurrencia en subpalabras:*

$$S = S^1 S^2 \dots S^j.$$

Entonces,

$$\sum_{i=1}^{j} \mathrm{asm}(S^i, P) \leq k.$$

Veamos una descripción del algoritmo. Para encontrar los calces, el algoritmo divide el texto en ventanas disjuntas de largo $v = (m-k)/2$. Esta división garantiza que cada calce contiene necesariamente a una de esas ventanas. Cada una de esas ventanas es candidata a ser descartada por el filtro del algoritmo. Una ventana descartada significa que no existe ningún calce conteniendo completamente esa ventana. Cuando el filtro no descarta una ventana, una porción de texto de tamaño $(m+3k)/2$ debe ser verificada por un algoritmo tradicional.

Ahora veamos cómo opera el filtro. Introduzcamos para ello la siguiente definición:

Definición 3.4.2. *Para una palabra V de largo v y para $l < v$, el multiconjunto D_l^V de l-gramas disjuntos de V es:*

$$D_l^V = \{\{V_{il+1..(i+1)l} \mid i = 0, \ldots, \lfloor v/l \rfloor - 1\}\}$$

contando repeticiones de elementos. Dicho de otra forma, D_l^V son los l-gramas obtenidos al particionar la ventana V en subpalabras de largo l.

El filtro descarta V si

$$\sum_{s \in D_l^V} \mathrm{asm}(s, P) > k,$$

en caso contrario, verifica la ventana. Para hacer este cálculo eficientemente, en una etapa de preprocesamiento el algoritmo construye una tabla con los valores de $\mathrm{asm}(S, P)$, para todas las palabras S de largo $l = \lfloor t \log m \rfloor$, donde t es un valor que depende únicamente de σ.

Los autores del algoritmo demuestran que, para t adecuado, existe ϵ dependiendo únicamente de σ, tal que $\mathbb{E}(\mathrm{asm}(S, P)) \geq \epsilon l$, donde la esperanza se toma sobre el conjunto de las palabras S de largo l. De aquí se deduce fácilmente que para detectar $k+1$ errores con este filtro, basta revisar $O(k)$ caracteres en promedio, con un mínimo de l caracteres. Usando esto, los autores demuestran que, para $k < \epsilon m$, el tiempo promedio de ejecución de este algoritmo es

$$O\left(\frac{k + \log_\sigma m}{m} n\right).$$

Está demostrado que este es el mejor resultado asintótico que se puede obtener para el caso promedio [CM94]. Más aún, el rango de valores de k para los cuales este filtro tiene la garantía anterior es mucho mayor que en otros algoritmos conocidos para el mismo problema.

Capítulo 4

Búsqueda aproximada permitiendo errores

Este capítulo presenta una manera alternativa de enfrentar el problema de búsqueda aproximada en línea, lo que creemos es la principal contribución de este trabajo. En esencia, este capítulo muestra que se pueden encontrar algoritmos significativamente más eficientes relajando la condición de encontrar todas las ocurrencias del patrón en un texto a sólo encontrar cada ocurrencia con alta probabilidad.

Esta variación del problema sigue teniendo sentido por cuanto búsqueda aproximada implica, en casi todas las aplicaciones, un modelo no exacto de alguna situación.

4.1. Introducción

Comenzaremos este capítulo con una observación general acerca de los algoritmos de filtrado que es la base para todo lo que sigue. Recordemos que esta familia de algoritmos se basa en el uso de alguna propiedad que garantiza el descarte de una porción de texto. En el caso de los algoritmos revisados en el capítulo anterior, podemos describir estas propiedades en forma sucinta como sigue:

- El algoritmo de caracteres lejanos calcula la fracción de caracteres de una ventana de texto que son lejanos para una alineación. Si esa fracción sobrepasa cierto umbral, se descarta la alineación.
- El algoritmo de q-gramas calcula la fracción de q-gramas de una ventana del texto que se encuentran en el patrón. Si esa fracción sobrepasa cierto umbral, se verifica la ventana.
- El algoritmo de Chang y Marr calcula la fracción de l-gramas de una ventana de texto que se encuentran en el patrón en forma suficientemente aproximada. Si esa fracción sobrepasa cierto umbral, se verifica la ventana.

Notamos inmediatamente una subestructura similar de la siguiente forma:

1. Se cuenta la fracción F de los elementos de un conjunto C que cumplen determinada propiedad P (por ejemplo, la fracción de caracteres de una porción de texto que son lejanos).

2. Si F sobrepasa cierto umbral ν, entonces se descarta (o verifica) ciertas posiciones del texto.

Esto resulta interesante, ya que en general esta fracción F se puede estimar fácilmente mediante muestreo aleatorio. Más precisamente, si se eligen algunos elementos al azar del conjunto C, la fracción de esos elementos que cumplen la propiedad P será probablemente muy parecida a F. Esto sugiere la posibilidad de modificar un algoritmo de filtrado, cambiando la determinación exacta de la fracción por una aproximación. En principio, resulta razonable utilizar el mismo criterio para descartar, es decir, si la estimación sobrepasa cierto umbral ν', se descarta (o verifica) ciertas posiciones del texto.

La modificación del filtro puede gatillar dos posibles errores:

1. El filtro modificado puede descartar posiciones del texto que el filtro original habría verificado.

2. El filtro modificado puede verificar posiciones del texto que el filtro original habría descartado.

Informalmente, podemos asociar el error 1 con pérdida de correctitud y el error 2 con pérdida de velocidad. En efecto, cada vez que el error 1 se produce podríamos eventualmente perder uno o más calces y cada vez que el error 2 se produce perdemos tiempo de verificación en ventanas que habrían sido descartadas (con certeza) por el algoritmo original. Es interesante notar también que el error 1 no genera disminución de velocidad así como el error 2 no genera error de correctitud.[1]

Veremos que esta idea se puede aprovechar (introduciendo modificaciones adecuadas para cada algoritmo) de manera de lograr algoritmos eficientes con una pequeña probabilidad de perder cada ocurrencia.

4.2. Definición de búsqueda aproximada permitiendo errores

Los algoritmos que propondremos a lo largo de este trabajo están motivados principalmente por la idea señalada en la sección anterior. Para dejar estos algoritmos bajo un esquema unificado, definiremos el tipo de algoritmos que nos interesa encontrar a través de propiedades. Para este fin, en toda esta sección denotaremos por $\mathcal{A}(T, P, n, m, k)$ a cualquier algoritmo probabilista cuya entrada es el error k y dos palabras T y P de largos n y m. Su salida es un conjunto de valores en $\{1, \ldots, n\}$ que lo representaremos como la variable aleatoria $S_{prob}(\mathcal{A}, T, P, n, m, k)$. Además definimos $S_{det}(T, P, n, m, k)$ como la salida del problema de búsqueda aproximada en T, P, n, m, k. Si no hay confusión, normalmente escribiremos S_{prob}, S_{det} y $\mathcal{A}$, sin especificar los parámetros.

[1] Asumiendo que el tiempo de filtro es menor que el tiempo de verificación.

Comenzaremos definiendo dos propiedades que exigiremos a los algoritmos propuestos. La primera obliga a cada algoritmo a no entregar ocurrencias inexistentes y la segunda obliga a entregar cada ocurrencia con cierta probabilidad mínima.

Definición 4.2.1. *Diremos que un algoritmo probabilista $\mathcal{A}(T, P, n, m, k)$ encuentra calces correctamente si cada elemento de su salida S_{prob} es una posición de término de una ocurrencia del patrón P en el texto T, a distancia no superior a k.*

Definición 4.2.2. *Diremos que un algoritmo probabilista $\mathcal{A}(T, P, n, m, k)$ tiene una probabilidad de a lo más p de perder cada calce si para todo $i \in S_{det}$ la probabilidad del evento $i \notin S_{prob}$ es a lo más p.*

En este trabajo, estas dos propiedades serán las mínimas exigidas y se resumen en la siguiente definición:

Definición 4.2.3. *Diremos que un algoritmo probabilista $\mathcal{A}(T, P, n, m, k)$ resuelve débilmente el problema de búsqueda aproximada permitiendo errores con probabilidad p si*

1. *Encuentra calces correctamente.*
2. *Tiene una probabilidad de a lo más p de perder cada calce.*

En los algoritmos propuestos, el valor de la cota p típicamente dependerá únicamente de n, m y k. En particular, no depende ni del texto ni del patrón, sólo de su largos. Un algoritmo $\mathcal{A}$ que resuelve débilmente el problema de búsqueda aproximada permitiendo errores permite resolver[2] problemas como:

- Existencia de ocurrencias: Dados T, P y k, decidir si existen ocurrencias del patrón P en el texto T a distancia k.
- Encontrar una fracción de ocurrencias: Dados T, P, k y $f \in [0, 1]$, encontrar al menos una fracción f del total de las ocurrencias del patrón P en el texto T a distancia k.

Para el primer problema, decidir si hay ocurrencias basado en si la salida del algoritmo $\mathcal{A}$ es o no vacía garantiza una probabilidad p de ser correcto. Para el segundo problema, a través de la desigualdad de Markov es fácil probar que la ejecución del algoritmo garantiza encontrar con probabilidad al menos $1-p/(1-f)$ una fracción al menos f de las ocurrencias, para $p > f$. Además, estas garantías pueden ser mejoradas repitiendo el algoritmo.

Como veremos en las secciones siguientes una de las características de muchos de los algoritmos propuestos es que dividen el texto en ventanas disjuntas de tamaño $\Theta(m)$ y luego realizan un mismo proceso en cada ventana. Usualmente esto se traducirá en que la detección de ocurrencias en ventanas alejadas sea realizada en forma independiente. Esta propiedad será una característica adicional que exigiremos a la mayoría de los algoritmos propuestos en este trabajo.

[2]En forma aproximada.

Definición 4.2.4. *Diremos que un algoritmo probabilista $\mathcal{A}(T, P, n, m, k)$ es independiente por ventanas si existe una constante c tal que para todo $i_1, i_2 \in S_{\text{det}}$ con $i_1 - i_2 > cm$, los eventos $i_1 \in S_{\text{prob}}$ e $i_2 \in S_{\text{prob}}$ son independientes.*

Definición 4.2.5. *Diremos que un algoritmo probabilista $\mathcal{A}(T, P, n, m, k)$ resuelve el problema de búsqueda aproximada permitiendo errores con probabilidad p si*

1. *Encuentra calces correctamente.*
2. *Tiene una probabilidad de a lo más p de perder cada calce.*
3. *Es independiente por ventanas.*

Las siguientes secciones presentan tres algoritmos. Cada uno de ellos se basa en uno de los algoritmos de filtro presentados en el Capítulo 3. Para cada uno mostramos un análisis de su complejidad temporal, dada una cota para la probabilidad de perder un calce. Demostraremos que los algoritmos obtenidos son en teoría y práctica, significativamente más rápidos que los algoritmos de filtro en los cuales se basan. La técnica para analizar cada uno de estos algoritmos es muy similar. Presentaremos un desarrollo detallado para el primero de ellos (caracteres lejanos). En el resto de los casos, sólo entregaremos los resultados principales.

4.3. El algoritmo de caracteres lejanos-PE

En esta sección presentaremos un primer algoritmo para búsqueda aproximada permitiendo errores, basado en el conteo de caracteres lejanos. La mayoría de las ideas presentadas aquí se utilizarán también en el resto de los algoritmos que introduciremos. En todos los casos, supondremos[3] que $k < m/2$.

Comenzaremos con algunas ideas generales. El algoritmo de Chang y Marr visto en la Sección 3.4 muestra que se puede extraer información acerca de la distancia entre una subpalabra de texto y el patrón conociendo únicamente una ventana de ella. Utilizaremos esta idea aquí, pero con un enfoque diferente. El algoritmo que proponemos comienza dividiendo el texto en ventanas disjuntas de tamaño $v = (m - k)/2$ y luego ejecuta una misma rutina sobre cada ventana. Esta rutina, que denominaremos filtro probabilista, o simplemente filtro si no hay confusión, es un procedimiento probabilista que puede tener dos posibles salidas: *descartar* y *verificar*. Cada vez que el filtro verifica una ventana, un procedimiento determinístico encuentra todos los posibles calces que pueden contener completamente a la ventana, lo que se puede hacer en $O(km)$ con programación dinámica.

El procedimiento queda así descrito con excepción del filtro probabilista. Notemos que si el filtro fuese correcto, es decir, sólo descarta ventanas en las que no hay calce, entonces el algoritmo resuelve el problema de búsqueda aproximada. Aunque este no será el caso, el error en la respuesta del filtro será pequeño, lo que implicará que con alta probabilidad los calces sean entregados por el algoritmo.

[3] Es un supuesto razonable, ya que ningún algoritmo de filtro conocido funciona para otros valores de k.

Ahora veamos como definir un filtro con las características citadas basado en el algoritmo de caracteres lejanos. Recordemos que si se encuentran en el texto más de k caracteres lejanos para una alineación, entonces la alineación puede ser descartada. De la misma manera, si se encuentran en el patrón más de k caracteres lejanos para una alineación, entonces ésta se puede descartar. La situación se ilustra en la Figura 4.1.

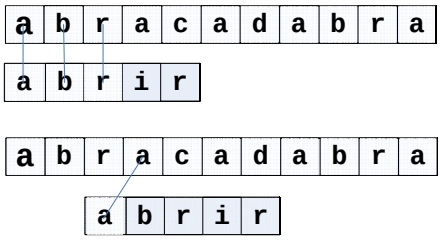

Figura 4.1: A diferencia del algoritmo original, ahora los caracteres lejanos se encuentran en el patrón, no en el texto. La figura muestra qué caracteres (fondo gris) son lejanos para dos alineaciones posibles con $k = 1$.

En todo este capítulo, entenderemos por alineación entre patrón y ventana cualquier alineación tal que la ventana queda incluida completamente en el patrón. En este contexto, los caracteres del patrón para los cuales tiene sentido la propiedad de ser o no carácter lejano se señalan en la Figura 4.2.

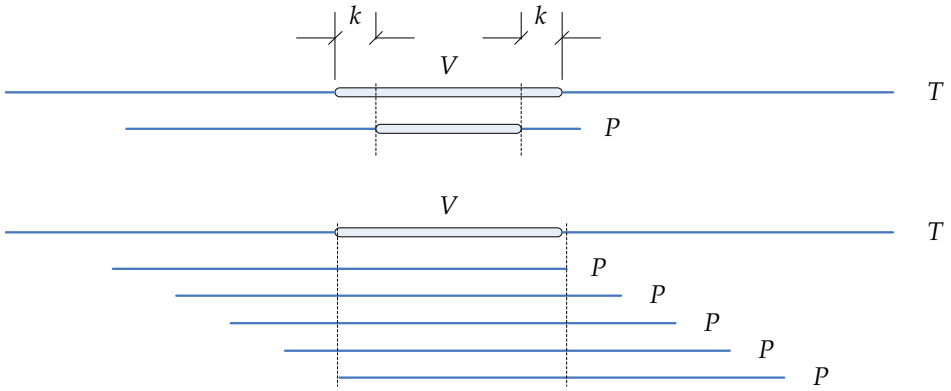

Figura 4.2: Arriba: Caracteres del patrón (en línea gruesa) para los cuales el algoritmo propuesto considera la propiedad ser carácter lejano, dada una alineación con una ventana. Abajo: Posibles alineaciones entre patrón y ventana.

La adaptación del filtro a nuestro algoritmo consiste en, dada una alineación cualquiera entre patrón y ventana, elegir al azar una cantidad (pequeña) de los caracteres del patrón y ver cual es la fracción de esos caracteres que son lejanos. Si exceden cierto valor umbral, se descarta la alineación. Los caracteres revisados por alineación no serán escogidos de manera independiente. Esto nos permitirá acelerar el descarte de ventanas. Definamos formalmente la función a precomputar:

Definición 4.3.1. *Dados:*

- $P \in \Sigma^m$ *un patrón fijo.*
- k *error de búsqueda.*
- $c \in \mathbb{N}$ *indicador del número de parámetros de la función.*
- $0 < F < 1$ *tolerancia máxima.*
- $j_1, j_2, \ldots, j_c \in \{k+1, \ldots, (m-3k)/2\}$ *indicador de posiciones a revisar.*

definimos la función de descarte

$$f : (\Sigma^{2k+1})^c \to \{\text{DESCARTAR}, \text{VERIFICAR}\}$$

como sigue: para cada $(s_1, \ldots, s_c) \in \Sigma^{(2k+1)c}$*, escribimos* $f(s_1, \ldots, s_c) = \text{VERIFICAR}$ *si existe una alineación* $j \in \{1, \ldots, (m+k)/2\}$ *tal que la cantidad de índices* i *en los que* P_{j+j_i} *no aparece como carácter en* s_i *es a lo más* Fc*. En otro caso, escribimos* $f(s_1, \ldots, s_c) =$ DESCARTAR.

La función de descarte es utilizada por el algoritmo para descartar todas las alineaciones entre el patrón y una ventana leyendo sólo una fracción de los caracteres de esta última. La función revisa c caracteres del patrón por cada alineación, pero los caracteres son escogidos de manera que sólo $c(2k+1)$ posiciones de la ventana son leídos en total (Ver Figura 4.3). En esencia, la función de descarte indica verificar una ventana cuando existe una alineación tal que la fracción de caracteres lejanos revisados para una alineación es menor o igual que F. La función se puede precomputar en tiempo $O\left(cm\sigma^{(2k+1)c}\right)$ usando un algoritmo como el que se indica en la Figura 4.4. Ésto restringirá la aplicabilidad práctica de este algoritmo sólo a pequeños valores de m.

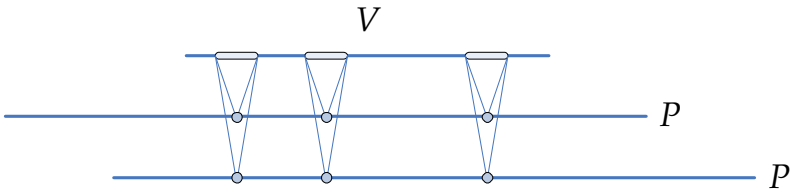

Figura 4.3: En cada ventana V, el algoritmo sólo revisa c subpalabras de largo $2k+1$ (en línea gruesa). Para cada alineación, esas subpalabras permiten determinar si c caracteres del patrón (en círculo) son lejanos.

El algoritmo completo de filtrado permitiendo errores basado en caracteres lejanos, **CL-PE**, queda como se indica en la Figura 4.5.

En la próxima sección, veremos que para elecciones adecuadas de las constantes c y F, el algoritmo tiene una probabilidad pequeña de perder cada calce y, simultáneamente, el tiempo de ejecución es bajo.

```
proc PRECÓMPUTO_{j_1,j_2,...,j_c}
  foreach (s_1,...,s_c) ∈ (Σ^{2k+1})^c
    for j = 1 to (m+k)/2
      if #{i = 1,...,c | P_{j+j_i} no aparece como carácter en s_i} ≤ Fc
        then almacenar f(s_1,...,s_c) = VERIFICAR
             break
      fi
    end
    if f(s_1,...,s_c) no ha sido almacenado
      then almacenar f(s_1,...,s_c) = DESCARTAR
    fi
  end
end
```

Figura 4.4: Precómputo de la función de descarte f.

```
proc CL-PE_{c,F}
  for i = 1 to c
      j_i ← {k+1,..., (m−3k)/2}.
  end
  Precomputar f : (Σ^{2k+1})^c → {DESCARTAR, VERIFICAR} asociada a los j_i
  foreach ventana disjunta V de largo (m−k)/2
    if f(V_{j_1−k..j_1+k} ... V_{j_c−k..j_c+k}) = DESCARTAR
      then descartar V
      else verificar V
    fi
  end
end
```

Figura 4.5: Algoritmo **CL-PE**.

4.3.1. Análisis

En todo lo que sigue asumiremos que P es un patrón fijo y sólo el texto T es aleatorio. Los resultados se extienden sin modificación alguna al caso en que adicionalmente el patrón es aleatorio, pues todas las probabilidades involucradas no dependen de cual sea el patrón. Recordemos que el modelo de aleatoriedad en el texto es Bernoulli uniforme.

Notemos que hay dos fuentes de aleatoriedad mezcladas en el análisis: la que proviene directamente del algoritmo a través de la elección de las posiciones, y aquella que se genera bajo el supuesto que el texto es aleatorio. Para facilitar la lectura, utilizaremos $\mathbb{P}_{Alg}$ para referirnos exclusivamente a la aleatoriedad del algoritmo.

En lo que sigue, denotamos por $v = (m-k)/2$ al largo común de todas las ventanas V con las que trabajaremos en esta sección y por $v' = (m-5k)/2$ el número de caracteres

del patrón en los que la propiedad carácter lejano tiene sentido, para una alineación con una ventana de largo v. Por simplicidad, a estos caracteres los denominaremos caracteres *válidos*.

Una de las restricciones impuestas en los algoritmos para búsqueda aproximada permitiendo errores es que la probabilidad de perder una ocurrencia es baja. Claramente esta probabilidad está acotada superiormente por la probabilidad de descartar una ventana V contenida completamente en la ocurrencia, que denotaremos por

$$\mathbb{P}_{Alg}\left(\text{Descartar } V\right).$$

Luego, la probabilidad de perder un calce cualquiera es a lo más

$$\max_{V \text{ admite calce}} \mathbb{P}_{Alg}\left(\text{Descartar } V\right), \tag{4.1}$$

donde "V admite calce" significa que existe una ocurrencia del texto conteniendo completamente la ventana. Usualmente la expresión 4.1 la escribiremos como

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right).$$

En el caso de este algoritmo, la probabilidad antes citada satisface la siguiente desigualdad:

Lema 4.3.1. *Para todo $F \geq k/v'$:*

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \exp\left(Fc - \frac{ck}{v'}\right)\left(\frac{k}{Fv'}\right)^{Fc}.$$

Demostración. Sea V una ventana que admite calce. Dado que existe calce, a lo más k caracteres del patrón son lejanos para una alineación j con el texto. Es decir, si P_i está alineado con T_{j+i}, el número de caracteres P_i tales que $P_i \notin \{T_{i+j-k}, \ldots, T_{i+j+k}\}$ es a lo más k y, en particular, cada carácter del patrón que es muestreado por el algoritmo tiene probabilidad a lo más k/v' de ser un carácter lejano. Luego, la probabilidad de que el algoritmo descarte la ventana es menor o igual que la probabilidad de obtener más de Fc caracteres lejanos de entre los c caracteres muestreados, lo que se puede acotar fácilmente utilizando el Teorema 2.6.1 como sigue:

$$\begin{aligned}
\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) &\leq \mathbb{P}\left(\text{Bin}\left(c, \frac{k}{v'}\right) > Fc\right) \\
&\leq \exp\left(Fc - \frac{ck}{v'}\right)\left(\frac{k}{Fv'}\right)^{Fc}.
\end{aligned}$$

Esto concluye la demostración. ∎

Ya que en cada ventana revisamos las mismas posiciones, nuestro algoritmo no es independiente por ventanas.

A continuación veremos el análisis temporal. La estrategia del análisis es la misma para todos los algoritmos que revisaremos, así que veamos primero la idea general. Queremos acotar el tiempo promedio esperado de ejecución del algoritmo. Como los algoritmos están basados en ventanas y el modelo de texto aleatorio es Bernoulli uniforme, podemos calcular el tiempo promedio esperado para una ventana, que denotaremos $\mathbb{T}(V)$ y luego multiplicar por $\Theta(n/m)$ para obtener el tiempo total.

Para acotar $\mathbb{T}(V)$, particionaremos el conjunto de todas las ventanas $V \in \Sigma^v$ de tamaño v en dos grupos A_β, y B_β, donde β es un parámetro que elegiremos posteriormente. La idea central es escoger estos conjuntos de manera que:

- La probabilidad del grupo A_β sea baja.
- La probabilidad de verificar una ventana, dado que ésta pertenece al grupo B_β sea baja.

La razón de esta elección es la siguiente. El tiempo promedio esperado por ventana puede calcularse como:

$$\mathbb{T}(V) = \mathbb{P}(A_\beta)\,\mathbb{T}_{Alg}(A_\beta) + \mathbb{P}(B_\beta)\,\mathbb{T}_{Alg}(B_\beta),$$

donde, para $S = A_\beta$ o $S = B_\beta$:

- $\mathbb{P}(S)$ denota la probabilidad que una ventana aleatoria pertenezca al grupo S.
- $\mathbb{T}_{Alg}(S)$ denota el tiempo promedio esperado del algoritmo dado que la ventana pertenece al grupo S.

Estimaremos el tiempo $\mathbb{T}(V)$ acotando los términos recién mencionados. Siempre acotaremos superiormente $\mathbb{T}(A_\beta)$ por el tiempo de filtrado y verificación ($O(km)$ utilizando el algoritmo de programación dinámica en [LV89]) y $\mathbb{P}(B_\beta)$ por 1, luego:

$$\mathbb{T}(V) = \mathbb{P}(A_\beta)\,O(km) + \mathbb{T}_{Alg}(B_\beta).$$

Por otro lado $\mathbb{T}_{Alg}(B_\beta)$ se puede acotar de la siguiente forma. Sea

- T_{filtro} una cota superior en el máximo tiempo de filtrado por ventana. El máximo está tomado sobre todas las ventanas y sobre todas las posibles instancias del algoritmo probabilista.
- $\mathbb{P}_{Alg}(\text{Verificar} \parallel B_\beta)$ es la máxima probabilidad de verificar una ventana, tomada sobre las ventanas que pertenecen al grupo B_β.
- $\mathbb{P}_{Alg}(\text{Descartar} \parallel B_\beta)$ el la máxima probabilidad de descartar una ventana, tomada sobre las ventanas que pertenecen al grupo B_β.

Entonces:

$$\begin{aligned}\mathbb{T}_{Alg}(B_\beta) &= O(T_{filtro})\mathbb{P}_{Alg}\left(\text{Descartar} \parallel B_\beta\right) + O(km)\mathbb{P}_{Alg}\left(\text{Verificar} \parallel B_\beta\right) \\ &= O(T_{filtro}) + O(km)\mathbb{P}_{Alg}\left(\text{Verificar} \parallel B_\beta\right)\end{aligned}$$

Juntando todo, nos queda la siguiente cota para $\mathbb{T}(V)$:

$$\mathbb{T}(V) = O(T_{filtro}) + \mathbb{P}\left(A_\beta\right) O(km) + \mathbb{P}\left(\text{Verificar} \parallel B_\beta\right) O(km).$$

Finalmente, observemos que si

$$\mathbb{P}\left(A_\beta\right) = O\left(\frac{1}{m^2}\right) \quad \text{y} \quad \mathbb{P}\left(\text{Verificar} \parallel B_\beta\right) = O\left(\frac{1}{m^2}\right)$$

entonces el tiempo de ejecución del algoritmo es $O(T_{filtro})$. Este argumento será utilizado recurrentemente a lo largo de este capítulo.

Veamos como aplicar lo anterior al algoritmo de caracteres lejanos. Denotemos por p la probabilidad de que un carácter del patrón sea lejano, para una alineación (fija) con una ventana aleatoria. Es fácil ver que

$$p = \left(1 - \frac{1}{\sigma}\right)^{2k+1}.$$

Introduzcamos la siguiente definición:

Definición 4.3.2. *Sea $\beta \geq 0$ un número real. Para una ventana V diremos que se cumple la propiedad* MAYOR(β) *si para cada alineación del patrón con la ventana el número de caracteres lejanos en el patrón es mayor que β.*

Definamos ahora los conjuntos:

$$A_\beta = \{V \mid V \text{ tiene tamaño } v \text{ y no cumple MAYOR}(\beta)\},$$
$$B_\beta = \{V \mid V \text{ tiene tamaño } v \text{ y cumple MAYOR}(\beta)\}.$$

Por claridad, escribiremos $\overline{\text{MAYOR}(\beta)}$ para referirnos al conjunto A_β y MAYOR(β) para referirnos al conjunto B_β.

Primero acotaremos superiormente $\mathbb{P}\left(\text{Verificar} \parallel \text{MAYOR}(\beta)\right)$.

Lema 4.3.2. *Si $Fv' \leq \beta$, entonces:*

$$\mathbb{P}\left(\text{Verificar} \parallel \text{MAYOR}(\beta)\right) \leq \frac{m+k}{2} \exp\left(\frac{-c\beta}{2v'}\left(1 - \frac{Fv'}{\beta}\right)^2\right).$$

Demostración. Sea V una ventana en MAYOR(β). Para una alineación fija j, cada P_{j+j_i} escogido por el algoritmo tiene probabilidad $q_j \geq \beta/v'$ de corresponder a un carácter lejano. Los j_i se escogen independientemente, por lo que la distribución del número de caracteres

lejanos en $\{P_{j+j_1}, \ldots, P_{j+j_c}\}$ para esa alineación es una binomial de c lanzamientos y probabilidad de éxito q_j. Luego, la probabilidad de que en una alineación cualquiera el algoritmo encuentre menos de Fc caracteres lejanos es como máximo

$$\mathbb{P}\left(\mathrm{Bin}\left(c, \frac{\beta}{v'}\right) \leq Fc\right).$$

Finalmente, la probabilidad de verificar una ventana es la probabilidad de que en alguna de las $(m+k)/2$ alineaciones se encuentren menos de Fc caracteres lejanos, lo que se puede acotar directamente usando lo anterior y una cota para la probabilidad de la unión de eventos:

$$\mathbb{P}_{Alg}\left(\text{Verificar} \parallel \text{MAYOR}(\beta)\right) \leq \frac{m+k}{2}\mathbb{P}\left(\mathrm{Bin}\left(c, \frac{\beta}{v'}\right) \leq Fc\right).$$

Usando la cota de Chernoff (Teorema 2.6.2) para acotar la binomial se concluye directamente el resultado.

■

Lema 4.3.3. *Si el texto T es aleatorio y V es una ventana del texto de tamaño v, entonces para todo β con $\beta \leq pv'$:*

$$\mathbb{P}\left(\overline{\text{MAYOR}(\beta)}\right) \leq \frac{m+k}{2}\exp\left(\frac{-8pv'}{25(2k+1)}\left(1-\frac{\beta}{pv'}\right)^2\right).$$

Demostración. Es claro que dos caracteres del patrón separados por distancia mayor que $2k$ son o no lejanos de manera independiente. Luego, para cada $i \in \{0, \ldots 2k\}$, tenemos que $P_{j+i}, P_{j+i+(2k+1)}, P_{j+i+2(2k+1)}, \ldots$ son o no lejanos de manera independiente. Cada carácter válido del patrón pertenece a uno de estos $2k+1$ conjuntos independientes.

Denotemos por $N_j(V)$ a la variable aleatoria que indica el número de caracteres lejanos del patrón en la ventana V del texto, para la alineación del patrón con la ventana en la que el primer carácter de la ventana está alineado con la posición j del patrón. La variable $N_j(V)$ se puede escribir como suma de variables indicatrices $Y_i^j, i = 1, \ldots v'$, que indican si el i-ésimo carácter válido del patrón es lejano. Lo anterior muestra que los Y_i^j son $(2k+1)$-independientes. Además, $\mathbb{P}\left(Y_i^j = 1\right) = p$, y usando Teorema 2.6.3:

$$\mathbb{P}\left(N_j(V) \leq \beta\right) = \mathbb{P}\left(\sum_j Y_i^j \leq \beta\right) \leq \exp\left(-\frac{8}{25}\frac{(pv'-\beta)^2}{pv'(2k+1)}\right)$$

Finalmente, si se cumple $\overline{\text{MAYOR}(\beta)}$, entonces existe una alineación j tal que existen menos de β caracteres lejanos. Usando una cota para la probabilidad de la unión de eventos, se concluye directamente que:

$$\mathbb{P}\left(\overline{\text{MAYOR}(\beta)}\right) \leq \frac{m+k}{2}\exp\left(\frac{-8pv'}{25(2k+1)}\left(1-\frac{\beta}{pv'}\right)^2\right).$$

■

La siguiente proposición finalmente muestra que si p no es demasiado pequeño, entonces existen valores de F y c tales que el algoritmo resuelve débilmente el problema de búsqueda aproximada permitiendo errores:

Proposición 4.3.1. *Sea $0 \leq \alpha < 1$. Si $p = \Omega(1/m^\alpha)$ y $k = O(\log m)$, entonces* **CL-PE**, *con $F = p/4$ y $c = \Theta(m^\alpha \log m)$ adecuado, cumple las siguientes propiedades:*

(1) La probabilidad de descartar cada calce es $O\left(m^{-\log m}\right)$.

(2) Si el texto y patrón son aleatorios, el algoritmo se ejecuta en tiempo promedio esperado $O(kn \log m/m^{1-\alpha})$, sin considerar el precómputo.

Demostración. La demostración es simplemente utilizar los Lemas 4.3.1, 4.3.2 y 4.3.3 en un caso particular. Fijemos $F = p/4$ y $c = \Theta(m^\alpha \log m)$ con constante suficientemente grande tal que, por Lema 4.3.1:

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) = O\left(m^{-\log m}\right).$$

Además, fijando $\beta = 2Fv'$ es fácil ver que, por Lema 4.3.3:

$$\mathbb{P}\left(\overline{\text{MAYOR}(\beta)}\right) = \frac{m+k}{2} \exp\left(-\Omega\left(m^{1-\alpha}/\log m\right)\right) = O\left(\frac{1}{m^2}\right).$$

Finalmente, agrandando la constante en $c = \Theta(m^\alpha \log m)$ si es necesario, y aplicando el Lema 4.3.2:

$$\mathbb{P}\left(\text{Verificar} \parallel \text{MAYOR}(\beta)\right) = O\left(\frac{1}{m^2}\right).$$

Se concluye de lo anterior que el tiempo promedio esperado es esencialmente el tiempo de filtrado, es decir $O(ckn/m)$, lo que prueba la segunda afirmación. ■

Aunque la implementación del algoritmo es sencillo, claramente no es práctico para valores grandes de m, σ y k, debido al espacio utilizado para almacenar la función de descarte precomputada. Además, el algoritmo no es independiente por ventanas, ya que siempre se revisan las mismas posiciones de cada ventana.

4.4. El algoritmo de q-gramas-PE

La idea central del algoritmo de q-gramas es que mientras en un texto aleatorio es difícil encontrar subpalabras del patrón de tamaño $\Omega(\log m)$, el fenómeno opuesto ocurre al restringirse a zonas del texto donde existen ocurrencias: en cada calce necesariamente se encuentran muchas de esas subpalabras. En esta sección utilizamos esta propiedad para definir un algoritmo para búsqueda aproximada permitiendo errores. Dividiremos el texto en ventanas disjuntas de tamaño $v = (m-k)/2$, en cada una de las cuales elegiremos una cierta cantidad de q-gramas al azar. Luego calculamos la fracción de aquellos q-gramas que son subpalabras del patrón y si esa fracción es menor que cierto umbral, se descarta

la ventana. Las ventanas no descartadas son verificadas con un algoritmo tradicional. Recordemos que Q_q^V denota el conjunto de q-gramas de la ventana V.

El algoritmo propuesto, **Q-PE**, es el que se encuentra descrito en la Figura 4.6.

```
proc Q-PE_{c,F,q}
  foreach ventana disjunta V de largo (m − k)/2
    for i = 1 to c
        s_i ← Q_q^V.
    end
    if #{i | s_i es subpalabra de P} ≤ Fc
      then descartar V
      else verificar V
    fi
  end
end
```

Figura 4.6: Algoritmo **Q-PE**.

4.4.1. Análisis

Esencialmente, acotaremos el tiempo promedio esperado del algoritmo de manera muy similar al caso de **CL-PE**. Utilizaremos la misma notación, a menos que específicamente se indique lo contrario. Nos limitaremos a entregar sólo los resultados principales.

Denotemos $v' = v - q + 1$, el número de q-gramas de la ventana, contando repeticiones. El siguiente lema acota la probabilidad de perder una ocurrencia.

Lema 4.4.1. *Si* $kq \leq v'(1-F)$, *entonces*

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \exp\left(c(1-F) - \frac{ckq}{v'}\right)\left(\frac{kq}{v'(1-F)}\right)^{c(1-F)}.$$

Demostración. Si V admite un calce, entonces existen al menos $v' - kq$ q-gramas que son subpalabras del patrón (ello pues cada error puede alterar a lo más q q-gramas). Luego,

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \mathbb{P}\left(\text{Bin}\left(c, \frac{v'-kq}{v'}\right) \leq Fc\right).$$

El lado derecho de la desigualdad es equivalente a:

$$\mathbb{P}\left(\text{Bin}\left(c, \frac{kq}{v'}\right) \geq c - Fc\right).$$

Usando una cota de Chernoff (Teorema 2.6.1) se concluye directamente el resultado.

■

Para el análisis temporal, utilizaremos la misma técnica de la sección anterior. El primer paso es definir los conjuntos A_β y B_β. Introduzcamos la siguiente

Definición 4.4.1. *Para una ventana V diremos que se cumple la propiedad* MENOR(β), *si menos de β de los q-gramas en Q_q^V son subpalabras del patrón.*

Definamos $A_\beta = \overline{\text{MENOR}(\beta)}$ y $B_\beta = \text{MENOR}(\beta)$. Los siguientes dos lemas son los análogos de los Lemas 4.3.2 y 4.3.3 de la sección anterior:

Lema 4.4.2. *Para todo $\beta \leq Fv'$,*

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) \leq \exp\left(Fc - \frac{c\beta}{v'}\right)\left(\frac{\beta}{Fv'}\right)^{Fc}.$$

Demostración. Si una ventana V cumple MENOR(β), la probabilidad de que un q-grama escogido al azar se encuentre en el patrón es menor que β/v'. Luego,

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) \leq \mathbb{P}\left(\text{Bin}\left(c, \frac{\beta}{v'}\right) \geq Fc\right).$$

Utilizando una cota de Chernoff (2.6.1) se concluye el resultado. ■

Lema 4.4.3. *Bajo el supuesto de texto aleatorio, sea p la probabilidad de que un q-grama de una ventana sea subpalabra del patrón. Entonces, para $\beta \geq pv'$:*

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) \leq \exp\left(-\frac{24(\beta - pv')^2}{25q(\beta + 2pv')}\right).$$

Demostración. Para $i = 1, \ldots, v'$ definamos la variable aleatoria X_i como

$$X_i(V) = \begin{cases} 1, & \text{si el } q\text{-grama } V_{i..i+q} \text{ es subpalabra del patrón,} \\ 0, & \text{en caso contrario.} \end{cases}$$

Si una ventana no cumple MENOR(β), entonces al menos β de los q-gramas de la ventana son subpalabras del patrón, luego:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) = \mathbb{P}\left(\sum X_i \geq \beta\right).$$

Finalmente, observemos que X_i es independiente de X_j si $|i-j| \geq q$, así que $X_1, \ldots, X_{v'}$ son q-independientes, con $\mathbb{P}(X_i = 1) = p$. Usando una cota tipo Chernoff (Teorema 2.6.3) se concluye que:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) \leq \exp\left(-\frac{8(\beta - pv')^2}{25q(pv' + (\beta - pv')/3)}\right).$$

A partir de esto se obtiene directamente el resultado deseado. ■

Los lemas anteriores son, al igual que en el caso de **CL-PE**, todo lo necesario para demostrar el resultado principal de esta sección. En este caso, el algoritmo **Q-PE** es independiente por ventanas y resuelve el problema de búsqueda aproximada permitiendo errores.

Proposición 4.4.1. *Existe $k^* = O(m/\log_\sigma m)$ tal que para todo $k < k^*$ y para todo $t > 0$ fijo, existen c, F y q tales que* **Q-PE**$_{c,F,q}$ *cumple las siguientes propiedades:*

(1) La probabilidad de descartar un calce es $O((k \log_\sigma m/m)^t)$.

(2) El tiempo promedio esperado de ejecución es $O(n \log_\sigma m/m)$.

(3) El algoritmo es independiente por ventanas.

Demostración. La independencia por ventanas se deduce directamente del hecho que el muestreo de q-gramas por ventana se realiza independientemente. Para el resto del resultado, el argumento de la demostración sigue el esquema general visto en la sección anterior. En primer lugar, notemos que si h es el número de q-gramas distintos del patrón, entonces

$$p = h/\sigma^q \leq m/\sigma^q.$$

En particular, podemos fijar $q = 2\log_\sigma m$, con lo que $p \leq 1/m$. Con esta elección de q es claro que si escogemos $\beta = \Theta(\log^2 m)$ con constante suficientemente grande, entonces, por Lema 4.4.3:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) = O\left(\frac{1}{m^2}\right).$$

Finalmente, fijemos $F = 1/2$ y $c = O(1)$ suficientemente grande tal que, por Lema 4.4.2:

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) = O\left(\frac{1}{m^2}\right).$$

Lo anterior implica que el tiempo de verificación es despreciable. Para concluir, debemos acotar el tiempo de filtrado. Esencialmente, el filtro elige c de los q-gramas en Q_q^V y determina cuántos de ellos son subpalabras de P. Ya que determinar si un q-grama es subpalabra de P puede hacerse en $O(1)$, el tiempo de filtrado es $O(cq) = O(\log_\sigma m)$. Esto demuestra *(2)*.

Ahora veamos *(1)*. Sea $k^* = v'(1-F)/q$, con lo que la hipótesis del Lema 4.4.1 se cumple. Escojamos c de modo de cumplir además $c(1-F) \geq t$. Notando que

$$\frac{kq}{(1-F)v'}$$

es $O(k \log_\sigma m/m)$ y aplicando el lema se concluye el resultado. ■

Observación. En la demostración, el parámetro t se considera constante. Evidentemente existe una dependencia en t del tiempo esperado de ejecución (es lineal en t) y de la probabilidad de descartar un calce no explicitada en el enunciado de la proposición.

En este algoritmo no aparecen los problemas del algoritmo de caracteres lejanos visto en la sección anterior. El algoritmo es fácilmente implementable[4], es independiente por ventanas y utiliza poco espacio ($O(mk)$ para guardar la tabla de q-gramas presentes en el patrón). En el capítulo 7 se evalúa este algoritmo desde el punto de vista práctico.

4.5. El algoritmo de Chang-Marr-PE

La adaptación del algoritmo de filtrado de Chang y Marr a nuestro esquema es muy natural: dividimos el texto en ventanas de tamaño $(m-k)/2$. Cada ventana es dividida en l-gramas disjuntos. Luego se eligen algunos l-gramas al azar y se examina la fracción de aquellos que aparecen en el patrón en forma suficientemente aproximada. Si esta fracción está por debajo de cierto umbral, se descarta la ventana. En otro caso se verifica.

Recordemos que D_l^V denota el conjunto de l-gramas disjuntos de V y $\mathrm{asm}(\cdot, P)$ es la función que a cada palabra le asigna la mínima distancia a las subpalabras de P. Utilizando esta notación, el algoritmo propuesto, que denominaremos **CM-PE**, queda como se indica en la Figura 4.7. El precómputo de la función $\mathrm{asm}(\cdot, P)$ para palabras de largo l se encuentra detallado en la Figura 4.8.

```
proc CM-PE_{c,F,l,g}
  Calcular asm(·, P) (sólo para palabras de largo l).
  foreach ventana disjunta V de largo (m − k)/2
    for i = 1 to c
        s_i ← D_l^V
    end
    if #{i | asm(s_i, P) ≤ g} ≤ Fc
      then descartar V
      else verificar V
    fi
  end
end
```

Figura 4.7: Algoritmo **CM-PE**.

4.5.1. Análisis

En todo lo que sigue, denotaremos el cardinal de D_l^V por d. Es decir, $d = v/l$. Veamos como acotar la probabilidad de perder un calce para el algoritmo en discusión.

Lema 4.5.1. *Si $k \leq dg(1-F)$, entonces*

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \exp\left(c(1-F) - c\frac{k}{dg}\right)\left(\frac{k}{dg(1-F)}\right)^{c(1-F)}$$

[4]Incluso más fácil que el algoritmo original, ya que no requiere el uso de un suffix trie para la lectura de los q-gramas.

```
proc ASM_{P,l}
  foreach w ∈ Σ^l
    Definir asm(w, P) = ∞
    for i = −k to k
      foreach s subpalabra de P de largo m + i
        if d(w, s) < asm(w, P)
          then definir asm(w, P) = d(w, s)
        fi
      end
    end
  end
end
```

Figura 4.8: Precómputo de la función $\mathrm{asm}(\cdot, P)$, restringido a palabras de largo l.

Demostración. Supongamos que V admite un calce. Entonces, para cualquier g:

$$\#\{s \in D_l^V \mid \mathrm{asm}(s, P) > g\} < \frac{k}{g}$$

Luego, la probabilidad de descartar una ventana V que admite calce puede acotarse superiormente por la probabilidad de que una binomial de c lanzamientos y probabilidad de éxito $1 - k/dg$ tenga a lo más Fc éxitos, es decir,

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \mathbb{P}\left(\mathrm{Bin}\left(c, 1 - \frac{k}{dg}\right) \leq Fc\right).$$

Notando que

$$\mathbb{P}\left(\mathrm{Bin}\left(c, 1 - \frac{k}{dg}\right) \leq Fc\right) = \mathbb{P}\left(\mathrm{Bin}\left(c, \frac{k}{dg}\right) \geq c - Fc\right)$$

y utilizando una cota de Chernoff (Teorema 2.6.1) para el lado derecho de la igualdad se concluye la demostración.

■

Para establecer una cota superior en el tiempo esperado de **CM-PE**, definamos $A_\beta = \overline{\mathrm{MENOR}}(\beta)$ y $B_\beta = \mathrm{MENOR}(\beta)$, donde el conjunto $\mathrm{MENOR}(\beta)$ viene dado por la siguiente definición:

Definición 4.5.1. *Para una ventana V diremos que se cumple la propiedad* $\mathrm{MENOR}(\beta)$ *si*

$$\#\{s \in D_l^V \mid \mathrm{asm}(s, P) \leq g\} < \beta.$$

Como en los algoritmos anteriores, probaremos los siguientes lemas:

Lema 4.5.2. *Si $\beta \leq Fd$ entonces,*

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) \leq \exp\left(Fc - \frac{c\beta}{d}\right)\left(\frac{\beta}{Fd}\right)^{Fc}.$$

Demostración. Dado que V cumple MENOR(β), la probabilidad de que una palabra elegida al azar en D_l^V se encuentre a distancia menor que g del patrón es menor o igual que β/d. Luego,

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) \leq \mathbb{P}\left(\text{Bin}\left(c, \frac{\beta}{d}\right) \geq Fc\right).$$

Utilizando una cota de Chernoff (Teorema 2.6.1) se concluye el resultado. ■

Lema 4.5.3. *Existen constantes $\alpha > 0$ y $0 < \epsilon < 1$ tal que, para $l = \alpha \log_\sigma m$, $g = \epsilon l$, $d \leq \beta m^3$ y m suficientemente grande*

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) \leq \exp\left(\beta - dm^{-3}\right)\left(\frac{dm^{-3}}{\beta}\right)^{\beta}.$$

Demostración. Como las subpalabras de V en D_l^V son disjuntas entre sí (no comparten posiciones del texto), es claro que los eventos $\{\text{asm}(s, P) \leq g\}$, con $s \in D_l^V$ son independientes entre sí. Además, tienen la misma probabilidad, que denotaremos $p(g, l)$ o simplemente p. De aquí se deduce que:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) = \mathbb{P}\left(\text{Bin}(d, p) \geq \beta\right).$$

En [CM94] se demuestra que existen constantes $\alpha > 0$ y $0 < \epsilon < 1$, tal que para $l = \alpha \log_\sigma m$ se tiene $p(\epsilon l, l) = O(m^{-3})$. Luego, dado este resultado,[5] basta aplicar una cota de Chernoff (Teorema 2.6.1) para demostrar el lema.

■

Proposición 4.5.1. *Existe $k^* = \Theta(m)$ tal que para todo $k < k^*$ y para todo $t > 0$, existen l, c, g y F tales que* **CM-PE**$_{c,F,l,g}$ *cumple las siguientes propiedades:*

(1) La probabilidad de descartar un calce es $O((k/m)^t)$

(2) El tiempo promedio esperado de ejecución es $O(n \log_\sigma m/m)$

(3) El algoritmo es independiente por ventanas.

Demostración. Sea $t > 0$. Fijemos $F = 1/2$, $c = \text{máx}\{2t, 6\}$, $g = \epsilon l$ y $k^* = v\epsilon/2$, donde ϵ es la constante del Lema 4.5.3. Si $k < k^*$ podemos utilizar el Lema 4.5.1, que permite concluir directamente *(1)*.

[5] Por completitud, en el apéndice se incluye una demostración corta de este resultado válida sólo para $\sigma \geq 8$.

Fijemos ahora $l = \alpha \log_\sigma m$, con α la constante del Lema 4.5.3. Fijando $\beta = 2$ y aplicando este último resultado se obtiene:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) = O\left(\frac{1}{m^4}\right) = O\left(\frac{1}{m^2}\right).$$

Finalmente, usando 4.5.2 y notando que $Fc \geq 3$ concluimos que:

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) = O\left(\frac{\log_\sigma^3 m}{m^3}\right) = O\left(\frac{1}{m^2}\right).$$

Así, el tiempo de verificación es despreciable. El tiempo de filtrado en una ventana V esencialmente es el tiempo invertido en leer c de los l-gramas en D_l^V. Luego, el tiempo total de filtrado, y por lo tanto el tiempo promedio esperado de ejecución del algoritmo es $O(ncl/m) = O(n \log_\sigma m/m)$, lo que prueba *(2)*.

La independencia por ventana se deduce del hecho que los l-gramas elegidos por ventana se escogen independientemente.

■

Observación. En la demostración, el parámetro t se considera constante. Evidentemente existe una dependencia en t del tiempo esperado de ejecución (es lineal en t) y de la probabilidad de descartar un calce no explicitada en el enunciado de la proposición.

Notemos que el espacio utilizado por este algoritmo esencialmente es el mismo que en el algoritmo original. Se necesita precalcular la función $asm(\cdot, P)$, lo que de acuerdo a los parámetros utilizados en la proposición 4.5.1, requiere espacio polinomial en m. Una evaluación práctica de este algoritmo se encuentra detallada en el capítulo 7.

Capítulo 5

El caso de la búsqueda de múltiples patrones

En este capítulo veremos que gran parte de la formalización y algoritmos presentados para búsqueda aproximada permitiendo errores se pueden extender de una manera simple al caso de la búsqueda aproximada múltiple, en el cual nos interesa reportar las ocurrencias aproximadas de varios patrones en un mismo texto.

Más que una presentación de nuevos algoritmos, el objetivo de este capítulo, y gran parte del capítulo siguiente, es mostrar que los conceptos introducidos en este trabajo se pueden estudiar en un esquema amplio incluyendo gran parte de los problemas de búsqueda aproximada.

5.1. Introducción

El problema de la búsqueda aproximada múltiple se define formalmente como sigue: dados r patrones que los escribiremos como $\vec{P} = (P^1, P^2, \ldots, P^r)$, un texto T y un error máximo permitido k, queremos reportar todas las ocurrencias de todos los patrones en el texto T. Por simplicidad, en todo lo que sigue supondremos que todos los patrones son del mismo largo m. El problema es una extensión de la versión exacta de búsqueda múltiple ($k = 0$) que ha sido estudiado extensamente [DM79, KS94, BYN00].

La existencia de algoritmos optimales para el problema de búsqueda aproximada múltiple ha sido resuelta recientemente. En el 2004, Navarro y Fredriksson [FN04] muestran un algoritmo que resulta optimal bajo el supuesto de texto y patrón aleatorios. Más precisamente, ellos demuestran que la complejidad promedio del problema de búsqueda aproximada múltiple es $O\left((k + \log_\sigma(rm))n/m\right)$ cuando $k < c_\sigma m$ (aquí, c_σ es una valor cercano a $1/2$ dependiente de σ). Para r grande, esto es notoriamente superior al algoritmo básico que consiste en repetir r ejecuciones de un algoritmo de búsqueda aproximada para buscar cada patrón por separado. El tiempo de ejecución promedio con esta estrategia es $\Omega((k + \log_\sigma m)rn/m)$.

Al igual que en el caso de un solo patrón, aquí podemos introducir algoritmos que admiten errores. Podemos definir el problema de búsqueda aproximada permitiendo errores para múltiples patrones a través de algoritmos que admiten una pequeña probabilidad de perder cada ocurrencia de cada uno de los r patrones.

En este caso, la formalización sigue el siguiente esquema. Los algoritmos para búsqueda aproximada múltiple permitiendo errores los denotaremos por $\mathcal{A}(T, \vec{P}, r, n, m, k)$, donde T es un texto de largo n, $\vec{P}$ es un vector de r palabras de largo m, y k es el error permitido. La salida de estos algoritmos es un conjunto de pares $(i, j) \in \{1, \ldots, r\} \times \{1, \ldots, n\}$ donde i indica un patrón en particular y j indica una posición en el texto donde habría un calce para ese patrón. Denotaremos por $S_{det}(T, \vec{P}, r, n, m, k)$ la salida del algoritmo determinista de búsqueda aproximada múltiple (es decir, el conjunto de todos los (i, j) correspondientes a ocurrencias) y por $S_{prob}(\mathcal{A}, T, \vec{P}, r, n, m, k)$ la variable aleatoria correspondiente a la salida del algoritmo probabilista $\mathcal{A}(T, \vec{P}, r, n, m, k)$.

Definición 5.1.1. *Diremos que un algoritmo probabilista $\mathcal{A}(T, \vec{P}, r, n, m, k)$ resuelve el problema de búsqueda aproximada múltiple permitiendo errores con probabilidad p si:*

1. *Encuentra calces correctamente, es decir, cada (i, j) entregado por el algoritmo corresponde a la posición de término j de una ocurrencia del patrón P^i en el texto T.*
2. *Tiene una probabilidad p de perder cada calce, es decir, para cada $(i, j) \in S_{det}$, la probabilidad del evento $(i, j) \notin S_{prob}$ es a lo más p.*
3. *Es independiente por ventanas, es decir, existe una constante c tal que para todo $(i_1, j_1), (i_2, j_2) \in S_{det}$, si $j_1 - j_2 > cm$, los eventos $(i_1, j_1) \in S_{prob}$ e $(i_2, j_2) \in S_{prob}$ son independientes.*

Si sólo 1 y 2 se cumplen, diremos que el algoritmo resuelve débilmente el problema de búsqueda aproximada con probabilidad p.

La siguiente sección muestra como los algoritmos vistos en capítulos anteriores se pueden extender para resolver el problema de búsqueda aproximada múltiple permitiendo errores.

5.2. Algunos algoritmos

Los algoritmos para búsqueda aproximada permitiendo errores pueden extenderse al caso de de búsqueda aproximada múltiple. En general, el esquema central se conserva. Los algoritmos se basan en un filtro que descarta o verifica ventanas del texto. Cada vez que una ventana se verifica, se revisa la vecindad de ésta en busca de ocurrencias de cualquiera de los patrones. La única adaptación ocurre en el filtro, donde la condición de descarte será levemente modificada para ser consistente con lo anterior. Veremos como realizar esta extensión para el caso de los algoritmos de q-gramas y de Chang y Marr.

5.2.1. El algoritmo de q-gramas

La idea del algoritmo de q-gramas permitiendo errores es dividir el texto en ventanas de tamaño $v = (m-k)/2$. En cada ventana se eligen algunos q-gramas al azar y se determina la fracción de ellos que se encuentra contenido en forma exacta en el patrón. Si esa fracción es grande, se verifica la ventana, en caso contrario, se descarta.

La extensión al caso de múltiples patrones consiste en muestrear q-gramas de la misma forma que el caso anterior, pero esta vez se determina la fracción de ellos que se encuentra contenido en forma exacta en alguno de los patrones. Nuevamente, si esa fracción supera cierto umbral, se verifica la ventana (buscando ocurrencias contenidas en la ventana de cualquiera de los patrones), en caso contrario, se descarta.

```
proc QMP-PE_{c,F,q}
  foreach ventana disjunta V de largo (m − k)/2
    for i = 1 to c
        s_i ← Q_q^V .
    end
    if #{i | s_i es subpalabra de algún P^j} ≤ Fc
      then descartar V
      else verificar V
    fi
  end
end
```

Figura 5.1: Algoritmo de q-gramas para múltiples patrones.

Esencialmente, el tiempo promedio esperado del algoritmo se analiza de manera muy similar al caso de **Q-PE** visto en la Sección 4.4. Utilizaremos la misma notación, a menos que específicamente se indique lo contrario.

El costo de verificación es $O(rm^2)$ si verificamos las ocurrencias de cada uno de los r patrones por separado con programación dinámica.

Denotemos $v' = v - q + 1$ el número de q-gramas de la ventana. Diremos que V admite calce si existe una ocurrencia de algún patrón que se encuentre completamente contenida en la ventana. El siguiente resultado acota la probabilidad de perder una ocurrencia.

Lema 5.2.1. *Si* $kq \leq v'(1-F)$, *entonces*

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \exp\left(c(1-F) - \frac{ckq}{v'}\right)\left(\frac{kq}{v'(1-F)}\right)^{c(1-F)}.$$

Demostración. Análoga a la demostración del Lema 4.4.1. ∎

Para el análisis temporal, utilizaremos la misma técnica del Capítulo 4. Introduzcamos la siguiente

Definición 5.2.1. *Para una ventana V diremos que se cumple la propiedad* MENOR(β)*, si menos de β q-gramas de la ventana V son subpalabra de algún patrón.*

Lema 5.2.2. *Para todo $\beta \leq Fv'$:*

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) \leq \exp\left(Fc - \frac{c\beta}{v'}\right)\left(\frac{\beta}{Fv'}\right)^{Fc}.$$

Demostración. Análoga a la demostración del Lema 4.4.2. ■

Lema 5.2.3. *Bajo el supuesto de texto aleatorio, sea p la probabilidad de que un q-grama de una ventana sea subpalabra de algún patrón. Entonces, para $\beta \geq pv'$:*

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) \leq \exp\left(-\frac{24(\beta - pv')^2}{25q(\beta + 2pv')}\right).$$

Demostración. Análoga a la demostración del Lema 4.4.3. ■

Finalmente, el algoritmo **QMP-PE**$_{c,F,q}$ es independiente por ventanas y resuelve el problema de búsqueda aproximada múltiple permitiendo errores.

Proposición 5.2.1. *Sean $u > 0$, $t > 0$ fijos y sea $r = O(m^u)$. Entonces existe $k^* = O(m/\log_\sigma(rm))$ tal que para todo $k < k^*$ existen c, F y q tales que* **QMP-PE**$_{c,F,q}$ *cumple las siguientes propiedades:*

(1) La probabilidad de descartar un calce es $O((k\log_\sigma(rm)/m)^t)$.

(2) El tiempo promedio esperado de ejecución es $O(n\log_\sigma(rm)/m)$.

(3) El algoritmo es independiente por ventanas.

Demostración. La independencia por ventanas se deduce directamente del hecho que el muestreo de q-gramas por ventana se realiza independientemente. Si h es el número de q-gramas distintos en todos los patrones, entonces

$$p = \frac{h}{\sigma^q} \leq \frac{rm}{\sigma^q}.$$

En particular, podemos fijar $q = 2\log_\sigma(rm)$, con lo que $p \leq 1/rm$. Con esta elección de q es claro que si escogemos $\beta = \Theta(\log^2(rm))$ suficientemente grande, entonces:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) = O\left(\frac{1}{rm^2}\right).$$

Finalmente, fijemos $F = 1/2$ y $c = O(1)$ suficientemente grande tal que:

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) = O\left(\frac{1}{rm^2}\right).$$

Lo anterior implica que el tiempo de verificación es despreciable,[1] y por lo tanto el tiempo esperado por ventana se reduce esencialmente al tiempo de filtrado $cq = O(\log_\sigma(rm))$, lo que prueba *(2)*.

Ahora veamos *(1)*. Sea $k^* = v'(1-F)/q$, con lo que la hipótesis del Lema 5.2.1 se cumple. Escojamos c de modo de cumplir además $c(1-F) \geq t$. Notando que

$$\frac{kq}{(1-F)v'}$$

es $O(k \log_\sigma(rm)/m)$ y aplicando el lema se concluye el resultado. ■

Observación. En la demostración, los parámetros t y u se consideran constantes.

5.2.2. El algoritmo de Chang y Marr

Otro algoritmo para este problema consiste en usar las ideas del algoritmo de Chang y Marr visto en la Seccion 4.5. Para cada ventana de texto de tamaño $(m-k)/2$, muestreamos algunos l-gramas. Si r no es excesivamente grande, en un texto aleatorio la mayoría de esos l-gramas se encontrarán a gran distancia (respecto a l) de todas las subpalabras de cada patrón. Por otro lado, cuando la ventana está contenida en una ocurrencia de algún patrón, la mayoría de los l-gramas muestreados se encontrarán a pequeña distancia de alguna subpalabra de ese patrón. Por lo tanto, podemos utilizar la cantidad de l-gramas muestreados que se encuentran a gran distancia de todos los patrones como criterio de filtro.

Definición 5.2.2. *Sean R una palabra cualquiera y $\vec{S}$ un vector de palabras cualesquiera. Definimos* $\mathrm{minAsm}(R, \vec{S})$ *como la mínima distancia de la palabra R a las subpalabras de las componentes de S. Es decir:*

$$\mathrm{minAsm}(R, \vec{S}) = \min_{i,j,k} d(R, S^k_{i..j})$$

El algoritmo propuesto se describe en la Figura 5.2. De manera similar a **CM-PE**, la distancia de los l-gramas a los patrones se precomputa. Dicho cálculo se encuentra detallado en la Figura 5.3.

El análisis es similar al realizado para **CM-PE**. En todo lo que sigue, denotaremos el cardinal de D_l^V por d. Es decir, $d = v/l$. Veamos como acotar la probabilidad de perder un calce para este algoritmo.

Lema 5.2.4. *Si $k \leq dg(1-F)$, entonces*

$$\mathbb{P}_{Alg}\left(\text{Descartar } V \parallel V \text{ admite calce}\right) \leq \exp\left(c(1-F) - c\frac{k}{dg}\right)\left(\frac{k}{dg(1-F)}\right)^{c(1-F)}.$$

Demostración. Análoga a la demostración del Lema 4.5.1 ■

[1] El tiempo de verificación ahora es $O(1/rm^2)$ en vez de $O(1/m^2)$.

```
proc CMMP-PE_{c,F,l,g}
  Calcular minAsm(·, P⃗) (sólo palabras de largo l).
  foreach ventana disjunta V de largo (m − k)/2
    for i = 1 to c
        s_i ← D_l^V
    end
    if #{i | minAsm(s_i, P⃗) ≤ g} ≤ Fc
      then descartar V
      else verificar V
    fi
  end
end
```

Figura 5.2: Algoritmo tipo Chang y Marr para múltiples patrones.

Para establecer una cota superior en el tiempo promedio esperado de **CMMP-PE** definimos:

Definición 5.2.3. *Para una ventana V diremos que se cumple la propiedad* MENOR(β) *si*

$$\#\{s \in D_l^V \text{ tal que } \operatorname{minAsm}(s, \vec{P}) \leq g\} < \beta.$$

Como en los algoritmos anteriores, probaremos los siguientes lemas:

Lema 5.2.5. *Si $\beta \leq Fd$ entonces,*

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) \leq \exp\left(Fc - \frac{c\beta}{d}\right)\left(\frac{\beta}{Fd}\right)^{Fc}$$

Demostración. Análoga a la demostración del Lema 4.5.2 ■

Lema 5.2.6. *Para $r = O(m^u)$ con $u > 0$ fijo, existen constantes $\alpha > 0$ y $0 < \epsilon < 1$ tal que, para $l = \alpha \log_\sigma m$, $g = \epsilon l$ y $d \leq \beta m^3$:*

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) \leq \exp\left(\beta - dm^{-3}\right)\left(\frac{dm^{-3}}{\beta}\right)^{\beta}$$

Demostración. Como las subpalabras de V en D_l^V son disjuntas entre sí (es decir, no comparten posiciones del texto), es claro que los eventos $\{\operatorname{minAsm}(s, \vec{P}) \leq g\}$, con $s \in D_l^V$ son independientes entre sí. Además, los eventos tienen la misma probabilidad, que denotaremos $p(g, l, r)$ o simplemente p. De aquí se deduce que:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) \leq \mathbb{P}\left(\text{Bin}(d, p) \geq \beta\right)$$

```
proc MINASM_{P⃗,r,l}
  foreach w ∈ Σ^l
    Definir minAsm(w, P⃗) = ∞
    for i = 1 to r
      for j = −k to k
        foreach s subpalabra de P_i de largo m + j
          if d(w, s) < minAsm(w, P⃗)
            then definir minAsm(w, P⃗) = d(w, s)
          fi
        end
      end
    end
  end
end
```

Figura 5.3: Precómputo de la función $\text{minAsm}(\cdot, \vec{P})$ restringido a palabras de largo l. El número de patrones es r.

Notemos que $\mathbb{P}\left(\text{minAsm}(s, \vec{P}) \leq g\right) = \mathbb{P}\left(\exists i \mid \text{asm}(s, P_i) \leq g\right) \leq rp(g, l)$, donde $p(g, l)$ se definió en el Lema 4.5.3.

En el Apéndice A se demuestra (basado en un resultado de [CM94]) que existen constantes $\alpha > 0$ y $0 < \epsilon < 1$, tal que para $l = \alpha \log_\sigma m$ se tiene $p(\epsilon l, l) = O(m^{-u-3})$. Aplicando una cota de Chernoff se concluye el resultado.

■

Proposición 5.2.2. *Sean $u > 0$ y $t > 0$ fijos. Para $r = O(m^u)$ existe $k^* = \Theta(m)$ tal que para todo $k < k^*$, existen l, c g y F tales que* **CMMP-PE**$_{c,F,l,g}$ *cumple las siguientes propiedades:*

(1) La probabilidad de descartar un calce es $O((k/m)^t)$

(2) El tiempo promedio esperado de ejecución es $O(n \log_\sigma m/m)$

(3) El algoritmo es independiente por ventanas.

Demostración. Sea $t > 0$. Fijemos $F = 1/2$, $g = \epsilon l$ y $k^* = v\epsilon/2$, donde ϵ es la constante del Lema 5.2.6. Si $k < k^*$ y $c \geq 2t$ podemos aplicar el Lema 5.2.4 y concluir directamente *(1)*.

Fijemos ahora $l = \alpha \log_\sigma m$, con α la constante del Lema 5.2.6. Fijando $\beta = u/2 + 1$ y aplicando este resultado se obtiene:

$$\mathbb{P}\left(\overline{\text{MENOR}(\beta)}\right) = O\left(\frac{1}{rm^2}\right).$$

Finalmente, escogiendo $c = \text{máx}(2t, (u+3)/2)$ y usando el Lema 5.2.5 concluimos que:

$$\mathbb{P}_{Alg}\left(\text{Verificar } V \parallel \text{MENOR}(\beta)\right) = O\left(\frac{\log_\sigma^{u+3} m}{m^{u+3}}\right) = O\left(\frac{1}{rm^2}\right).$$

Así, el tiempo de verificación es despreciable y por lo tanto el tiempo promedio esperado se puede reducir al tiempo de filtrado $O(ncl/m) = O(n \log_\sigma m/m)$, lo que prueba *(2)*.

La independencia por ventanas se deduce del hecho que los l-gramas elegidos por ventana se escogen independientemente.

■

Observación. En la demostración, los parámetros t y u se consideran constantes.

Capítulo 6

El caso de la búsqueda fuera de línea

Hasta ahora, todos los algoritmos considerados tratan el caso *en línea* del problema de búsqueda aproximada, donde no se asume ningún conocimiento previo sobre el texto para responder una búsqueda. En particular, esto obliga a que cada algoritmo para el problema de búsqueda aproximada y para la versión permitiendo errores tengan complejidad temporal $\Omega(n/m)$, ya que en cada ventana de tamaño m se debe leer al menos un carácter para tomar una decisión acerca de la existencia de calces en esa ventana. Sublinealidad en n resulta por tanto inviable con este esquema.

Un enfoque alternativo es el *fuera de línea*, que se basa en la construcción de una estructura sobre el texto que luego permite responder búsquedas de patrones más eficientemente. La construcción de esta estructura (o *índice*) puede ser muy costosa (normalmente leen el texto completo una vez, y usan espacio proporcional al tamaño del texto), pero ese costo se compensa con las rápidas búsquedas (incluso sublineales en n) sobre ese mismo texto.

Elegir entre algoritmos en línea o fuera de línea depende principalmente de la aplicación. Nuestro interés en esta sección es mostrar que el concepto de búsqueda aproximada permitiendo errores se puede extender a este último caso.

6.1. Introducción

En muchas situaciones, la búsqueda en línea no resulta útil en la práctica. Un ejemplo son las grandes bases de datos de texto actuales, donde sólo el tiempo invertido en leer el texto resulta ya demasiado costoso en muchas aplicaciones. Otro ejemplo son los sistemas de consulta, dónde el costo de búsqueda se multiplica por el enorme número de búsquedas que los usuarios demandan. En ambos casos, se necesita un sistema de búsqueda más eficiente.

6.1.1. Indíces

El término *índice* alude a cualquier estructura que se construye sobre el texto que permite luego acelerar alguna operación, en nuestro caso, realizar búsquedas. Por ejemplo, un índice para realizar búsquedas sobre un texto muy largo T escrito en español podría almacenar todas las palabras del diccionario y las ubicaciones en que cada palabra aparece en el texto. Este procedimiento es muy costoso, de hecho, al menos requiere leer el texto completo. Pero supongamos que después de haber construido el índice recibimos una búsqueda por una palabra P en forma exacta: la respuesta sería inmediata, pues bastaría con utilizar el índice para retornar la lista de sus apariciones. En general, no es difícil utilizar el índice anterior para efectuar eficientemente búsquedas más complejas, por ejemplo, frases.

En el párrafo anterior se muestra una situación donde la construcción de un índice puede mejorar significativamente los tiempos de búsqueda. Sin embargo, estamos realizando tácitamente dos supuestos no requeridos por la búsqueda en línea:

1. El texto es completamente conocido antes de efectuar las búsquedas. Normalmente un índice requiere conocer el texto completo para su construcción.

2. Se realizan muchas búsquedas sobre el texto. Si se hicieran pocas búsquedas, el tiempo invertido en la construcción del índice sería un sobrecosto altísimo que probablemente no vale la pena pagar.

Además de estos factores, el uso de índices tiene ciertas limitaciones particulares:

- Requerimientos de espacio: Usualmente los índices utilizan un tamaño proporcional al texto. En algunos casos este tamaño resulta inaceptable en las aplicaciones prácticas.

- Tolerancia a pequeñas modificaciones: En muchas aplicaciones, el texto a indexar puede sufrir modificaciones pequeñas en el tiempo. Esto requiere que el índice pueda adaptarse a estas pequeñas modificaciones sin tener que reconstruirlo completamente.

- Ciertos índices utilizan una estructura conocida del texto. El ejemplo del índice para textos en español es uno de ellos.[1] Otros son más generales, pero suelen ser más lentos que aquellos construidos directamente para una aplicación particular.

En este trabajo sólo nos concentraremos en índices para texto en general. El lector puede consultar [NBYST01] para un revisión de las técnicas más comunes de indexado de este tipo.

En lo que sigue, veremos propuestas de algoritmos para búsqueda aproximada permitiendo errores para el caso fuera de línea. Todos los algoritmos propuestos irán precedidos de una breve descripción de los algoritmos que utilizamos como base.

[1] Se basan en que el texto es una concatenación de palabras

6.2. Algoritmos basados en el índice de q-gramas

Dentro de aquellos índices que permiten acceder más eficientemente a las subpalabras del texto se encuentra el de q-gramas. Este índice guarda, para cada q-grama en Σ^q, las posiciones del texto en las cuales aparece.

La idea básica [NBY98] para utilizar el índice de q-gramas en búsqueda aproximada es dividir el patrón en $k+1$ subpalabras $P = P^1 P^2 \dots P^{k+1}$, cada una de las cuales se busca en forma exacta dentro del texto. Luego se revisa la vecindad de todas esas ocurrencias para ver si el patrón se encuentra en forma aproximada. Este procedimiento permite encontrar todas las ocurrencias, debido al siguiente resultado.

Lema 6.2.1. *Si el patrón es dividido en $k+1$ subpalabras, al menos una de ellas debe aparecer en cada ocurrencia a distancia no superior a k del patrón en un texto.*

La utilidad del índice de q-gramas es que permite hacer la búsqueda de las palabras P^i de manera más eficiente que el equivalente a una lectura completa del texto. Por simplicidad, supongamos que todas las palabras son del mismo tamaño $l = m/(k+1)$, y por lo tanto los P^i son los elementos de D_l^P. Dependiendo de l y q, hay dos casos:

- Si $q \leq l$ podemos utilizar el índice para localizar los prefijos de largo q de cada l-grama en D_l^P. Luego, una revisión de una vecindad de esos prefijos permite saber si esos prefijos corresponden a ocurrencias efectivas de los P^i.
- Si $q > l$, el índice tiene directamente todas las ocurrencias de cada l-grama en D_l^P. Basta retornar la unión de todas las ocurrencias de los q-gramas de los cuales P^i es un prefijo.

En el caso de texto aleatorio, el algoritmo utilizando el índice de q-gramas siguiendo el proceso antes mencionado es más eficiente que cualquier algoritmo en línea. Su tiempo promedio de ejecución es:

$$O\left(\frac{km^2n}{\sigma^{m/k}} + \frac{kn}{\sigma^q}\right),$$

como se encuentra demostrado en [NBY98].

6.2.1. Adaptando el índice para búsqueda permitiendo errores

En esta sección presentaremos un primer algoritmo para búsqueda fuera de línea permitiendo errores. La idea es usar el índice de q-gramas del texto de la sección anterior para responder a las búsquedas. Sin embargo, dado que sólo queremos encontrar cada ocurrencia con alta probabilidad, el tiempo que necesitaremos en una búsqueda será menor.

Veamos una descripción del algoritmo en su forma más simple (luego introduciremos algunas modificaciones). Construiremos un índice en el que para cada posible q-grama del texto guardaremos las posiciones en las que aparece.

En el algoritmo original descrito al inicio de esta sección, particionar el patrón en $k+1$ subpalabras es suficiente para garantizar la existencia de al menos una de esas palabras en cada ocurrencia. Muestrear un pequeño número de esas subpalabras y buscar en la vecindad de sus ocurrencias resulta insuficiente para encontrar cada calce con probabilidad al menos constante. Para solucionar esto, utilizaremos la siguiente extensión del Lema 6.2.1

Lema 6.2.2. *Si el patrón es dividido en $k+s$ subpalabras, al menos s deben aparecer en cada ocurrencia del patrón en un texto a distancia no superior a k.*

Motivados por este lema, podemos dividir el patrón en $k+s$ subpalabras del mismo largo, muestrear una cierta cantidad c de esas subpalabras y buscar en la vecindad de sus apariciones en el texto. En todo lo que sigue de este capítulo, denotemos $l \equiv m/(k+s)$, con lo que el muestreo puede verse como la elección de c elementos escogidos al azar en D_l^P.

Utilizaremos el índice de q-gramas en la misma forma que el algoritmo original, pero restringido a los l-gramas muestreados. Observemos que si el muestreo de patrones se realiza una sóla vez para todo el texto, entonces las ocurrencias no son reportadas de manera independiente.

El índice, que denotaremos I, se construye como se indica en la Figura 6.1. Por simplicidad, veremos el índice como un conjunto de listas: para cada q-grama Q, la lista I_Q contiene todas las posiciones donde aparece el q-grama Q dentro del texto. Señalamos eso sí que el índice descrito en [NBY98] utiliza indexación por bloques, donde para cada q-grama se guarda no la posición i, sino i mod b, para algún $b \in \mathbb{N}$, de modo de reducir el espacio del índice.

```
proc Q-FL-IND_{c,F,q}(T)
  comment: Creación del índice
  foreach q-grama Q
    for i = 1 to n − q
      if Q = T_{i..i+q}
        then Agregar i a la lista I_Q
      fi
    end
  end
end
```

Figura 6.1: Índice de q-gramas.

El algoritmo para realizar búsqueda aproximada permitiendo errores usando el índice I lo denominaremos **Q-FL** y se describe en la Figura 6.2. En la descripción del algoritmo sólo consideramos el caso en que los l-gramas tienen largo mayor o igual que q. Para una palabra s de largo $|s| \geq q$, utilizaremos la notación $\text{pref}_q(s)$ para denotar el prefijo de largo q de s.

El proceso de verificar es el mismo de los capítulos anteriores, pero aplicado a una

ventana de tamaño distinto. En el algoritmo **Q-FL**, el proceso de verificar $T_{j..j+l-1}$ corresponde a buscar todas las ocurrencias que puedan contener completamente esa porción del texto. Para ello basta buscar ocurrencias en $T_{j-(m+k-l)..j+l-1+(m+k-l)}$, es decir, en una ventana de largo $O(m)$ al igual que en todos los algoritmos propuestos en los capítulos anteriores.

```
proc Q-FL_{c,s,q,I}
  comment: responder consultas
  l = m/(k + s)
  for i = 1 to c
      s_i ← D_l^P
      foreach j ∈ I_{pref_q(s_i)}
        if T_{j..j+l-1} = s_i
          then verificar T_{j..j+l-1}
        fi
      end
  end
end
```

Figura 6.2: Algoritmo de indexado basado en el índice de q-gramas.

6.2.2. Análisis

La adaptación del algoritmo usa la idea de muestrear q-gramas en una manera diferente a la utilizada para algoritmos en línea. Aunque estamos interesados en las mismas variables del caso en línea, es decir, probabilidad de perder una ocurrencia y tiempo promedio esperado de ejecución, el análisis es bastante más simple en este caso. Además de estas dos magnitudes, aquí resulta de vital importancia el espacio utilizado por el índice. Los siguientes dos lemas acotan directamente la probabilidad de perder una ocurrencia y el tiempo promedio esperado de ejecución:

Lema 6.2.3. *La probabilidad que el algoritmo descarte una ocurrencia es a lo más* $(\frac{k}{k+s})^c$.

Demostración. En una ocurrencia aparecen al menos s de los $k+s$ elementos en D_l^P. Si al menos uno de esos l-gramas es seleccionado entonces la ocurrencia es reportada. La probabilidad de descartarla es por lo tanto a lo más $(\frac{k}{k+s})^c$. ■

Lema 6.2.4. *El tiempo promedio esperado de ejecución del algoritmo es*

$$\frac{2cn}{\sigma^q} + \frac{cn}{\sigma^l} O\left(m^2\right).$$

Demostración. El tiempo promedio esperado depende esencialmente del tiempo empleado en las comprobaciones $T_{j..j+l-1} = s_i$ y del tiempo empleado en las verificaciones.

Ya que el número promedio de apariciones de los l-gramas elegidos en el patrón es a lo más cn/σ^l, el tiempo promedio esperado de verificación usando programación dinámica es

$$\frac{cn}{\sigma^l} O\left(m^2\right).$$

Respecto al tiempo utilizado al revisar que los q-gramas entregados por el índice sean ocurrencias de los l-gramas, es fácil demostrar que el tiempo esperado de esta revisión por cada posición revisada es básicamente constante. En efecto, la probabilidad de que se lean i caracteres al efectuar esta revisión es

$$\frac{1}{\sigma^{i-1}}\left(1-\frac{1}{\sigma}\right)$$

y por lo tanto el tiempo esperado es a lo más

$$\frac{1}{1-1/\sigma} \leq 2.$$

El número esperado de posiciones revisadas puede acotarse por el número esperado de apariciones en el texto de los c q-gramas muestreados, que es cn/σ^q. Luego, el tiempo esperado de revisión es a lo más el doble de esta cantidad.

De aquí, el tiempo esperado del algoritmo está acotado superiormente por

$$\frac{2cn}{\sigma^q} + \frac{cn}{\sigma^l} O\left(m^2\right).$$

■

El espacio utilizado por el índice, tal como se encuentra descrito, es esencialmente $O(n \log n)$. Sin embargo, el índice descrito en [NBY98], que cumple la misma funcionalidad, puede utilizarse en este esquema sin modificaciones. El espacio es $o(n)$ bajo ciertas condiciones que no detallaremos.

Finalmente, al igual que en los casos anteriores, mostraremos valores que permiten obtener un resultado interesante desde el punto de vista teórico. Naturalmente, otros valores pueden establecerse de manera de cambiar la relación entre pérdida de calces y tiempo esperado de ejecución. En cuanto al espacio utilizado, este es el mismo que el algoritmo original. Como diferencia importante respecto a los algoritmos en línea, no es razonable fijar el tamaño q de los q-gramas, puesto que el índice se crea una sola vez para responder consultas.

Proposición 6.2.1. *Sea $t > 0$ una constante fija. Si $q < m/k\sigma$, entonces existen s y c tales que* **Q-FL**$_{c,s,q}$ *cumple las siguientes propiedades:*

(1) La probabilidad de descartar un calce es $O(1/k^t)$.

(2) El tiempo promedio esperado de ejecución es

$$\frac{2tn \log_\sigma k}{\sigma^q} + \frac{tn \log_\sigma k}{\sigma^{m/k\sigma}} O\left(m^2\right).$$

Demostración. Basta fijar $s = k\sigma - k$, $c = t \log_\sigma k$ y aplicar los Lemas 6.2.3 y 6.2.4. ■

6.2.3. Un segundo algoritmo basado en el índice de q-gramas

Una posible modificación al algoritmo visto en la sección anterior consiste en revisar todos los l-gramas en D_l^P, pero ahora para cada l-grama se verifica la vecindad de sólo algunas de sus apariciones en el texto. Este algoritmo se encuentra descrito en la Figura 6.3.

```
proc Q-FL2_{f,s,q,I}
  comment: responder consultas.
  l = m/(k + s)
  foreach s_i ∈ D_l^P
     Elegir una sublista I' de I_{pref_q(s_i)} de tamaño f · #I_{pref_q(s_i)} uniformemente.
    foreach j ∈ I'
      if T_{j..j+l-1} = s_i
        then verificar T_{j..j+l-1}
      fi
    end
  end
end
```

Figura 6.3: Un segundo algoritmo de indexado basado en el índice de q-gramas.

La generación del conjunto I' (ver algoritmo) en cada una de las $k+s$ iteraciones puede llegar a ser muy costosa, pero desde el punto de vista práctico esto lo podemos reemplazar por un procedimiento que elige una cantidad fija de elementos de la lista $I_{\mathrm{pref}_q(s_i)}$ uniformemente, de manera tal que el número esperado de elementos distintos reportados coincida con el tamaño de la lista I'.

El análisis es muy similar al caso anterior. Si hay una ocurrencia, entonces al menos s de los l-gramas se encuentran en el texto. Cada uno de esos l-gramas es reportado con probabilidad al menos f y por lo tanto la probabilidad de descartar una ocurrencia es a lo más $(1-f)^s$. Además, el tiempo esperado de verificación es $f(k+s)n/\sigma^l O(m^2)$ y el tiempo esperado de filtro es $f(k+s)n/\sigma^q$.

Así, obtenemos el siguiente resultado:

Proposición 6.2.2. *Si $q < m/k\sigma$, entonces existen s y c tales que* **Q-FL2**$_{c,s,q}$ *cumple las siguientes propiedades:*

(1) La probabilidad de descartar un calce es $O(1/k)$.

(2) El tiempo esperado de ejecución (sin considerar la elección de la sublista I') es

$$\frac{2n\ln k}{\sigma^q} + \frac{n\ln k}{\sigma^{m/2k}}O(m^2).$$

Demostración. Basta fijar $f = \ln k/k$, $s = k$, y aplicar los lemas anteriores. Se puede demostrar que:

$$\left(1 - \frac{\ln x}{x}\right)^x < \frac{1}{x}$$

para concluir el resultado de la parte *(1)*. ■

Capítulo 7

Evaluación práctica de los algoritmos

A lo largo de este trabajo hemos propuesto algunos algoritmos para el problema de búsqueda aproximada permitiendo errores y demostrado algunas garantías teóricas que ellos ofrecen. Cerraremos este trabajo revisando, a través de experimentos, el aspecto práctico de estos algoritmos.

Nuestro principal objetivo no es proveer la mejor implementación de cada algoritmo propuesto, sino dar información relevante que permita entender la competitividad de los algoritmos propuestos en relación a algoritmos existentes para el problema de búsqueda aproximada.

7.1. Los algoritmos

En este trabajo nos restringimos a evaluar los algoritmos **Q-PE** (Sección 4.4) y **CM-PE** (Sección 4.5). Los algoritmos fueron implementados en C++, compilados usando `g++`. Los experimentos fueron ejecutados en un computador 1.86Ghz Core Duo, 1GB de RAM, utilizando linux 2.6.15.

Dado que los algoritmos propuestos funcionan bajo un esquema distinto respecto a algoritmos tradicionales, optamos por efectuar experimentos cuyo objetivo es medir la relación entre tiempo de ejecución y pérdida de ocurrencias. También efectuamos una comparación entre los algoritmos nuevos y aquellos en los cuales se basan. Los experimentos están realizados con algoritmos sin optimizar, todos implementados por nosotros mismos.

Los detalles de implementación se encuentran adecuadamente comentados en el código fuente (incluido en el Apéndice B). Algunos detalles importantes:

- La rutina para verificar ventanas es un algoritmo muy eficiente basado en paralelismo de bits, de manera de evitar que el tiempo total de verificación predominara sobre el tiempo de total de filtrado.
- Comparamos el algoritmo **CM-PE** con el algoritmo de Chang y Marr visto en la Sección 3.4 (que en lo que sigue denominaremos **CM**) y el algoritmo **Q-PE** con el

algoritmo de q-gramas visto en Sección 3.3 (que en lo que sigue denominaremos **Q**).

- Todos los algoritmos leen primero el texto completo y lo almacenan en un buffer de memoria RAM. El algoritmo se ejecuta posteriormente sobre dicho buffer. Se utilizó la memoria primaria en vez de la secundaria debido a que esta última no es eficiente bajo acceso aleatorio.

- En el algoritmo de q-gramas original utilizamos una estructura diferente a la señalada por los autores del algoritmo para realizar el conteo de los q-gramas en el texto. Esencialmente, precomputamos una función que representa cada q-grama Q con un número entero n_Q y cada caracter c con un número entero n_c, de manera que para cada caracter c, el q-grama obtenido al remover el primer carácter de Q y agregar c como último carácter es una función sencilla de n_Q y n_c. Esta función puede utilizarse para efectuar el conteo de q-gramas en $O(n)$ tal como en el algoritmo original.

- Se ignoraron los tiempos de precómputo en todos los algoritmos. El tiempo medido es el invertido al procesar el texto.

7.2. Datos

Utilizamos las colecciones de texto disponibles en Pizzachili [FN05]. Pizzachili es un repositorio (desarrollado y mantenido por académicos de la Universidad de Chile y Universidad de Pisa) cuyo objetivo es la evaluación de índices comprimidos, es decir, estructuras de tamaño pequeño[1] creadas a partir de un texto y que permiten realizar búsquedas sobre él de manera más eficiente. Los textos disponibles en Pizzachili (denominadas colecciones) provienen de concatenación de datos reales provenientes de distintas fuentes, entre otras, códigos fuentes, audio en formato MIDI, cadenas de proteínas, cadenas de ADN, texto en inglés y datos estructurados de XML.

La mayoría de las colecciones se encuentra almacenada en un archivo de texto ASCII de al menos 50MB. Para cada colección se encuentra calculada la probabilidad de coincidencia de caracteres. Representando una colección como un texto T de largo n, esta probabilidad equivale a

$$\mathbb{P}\left(T_i = T_j\right)$$

cuando i y j son elegidos independiente y uniformemente en el conjunto $\{1, \ldots, n\}$. En general, el inverso de esta probabilidad se asocia al número efectivo de caracteres presentes en la colección, ya que desprecia a los caracteres cuya frecuencia de aparición es baja. La Tabla 7.1 muestra el tamaño de cada colección, el número de símbolos diferentes usados en el texto y el inverso de la probabilidad de coincidencia.

Evaluamos nuestros algoritmos en un subconjunto de las colecciones disponibles en el repositorio Pizzachili, específicamente, las colecciones de cadenas de ADN, texto en inglés y archivos MIDI. Estas tres colecciones no sólo provienen de fuentes completamente distintas, sino también tienen una cantidad muy diferente de caracteres efectivos. Además,

[1] Relativo al tamaño del texto

Colección	Tamaño(MB)	Alfabeto	Inv. Prob. Coincid.
Código fuente	210	230	24.77
Archivos MIDI	55	133	39.75
Cadenas de Proteínas	66	24	16.98
Cadenas de ADN	403	16	3.91
Texto en inglés	2210	239	15.25
Archivos XML	294	97	28.73

Tabla 7.1: Colecciones de texto disponibles en el repositorio PizzaChili (Nov. 2006).

estas colecciones son las mismas que aquellas que típicamente se utilizan para evaluar algoritmos para búsqueda aproximada [Nav01, Mye94].

Para el experimento, se preprocesó el texto de la siguiente manera: en cada colección se removieron todos los caracteres cuya frecuencia de aparición es inferior a 0.001, de manera que el tamaño del alfabeto fuese más representativo del número de caracteres utilizados en el texto. También, por motivos de tiempo, los datos fueron reducidos a 50MB por colección, simplemente truncando el archivo. Tras estas modificaciones se obtuvieron tres archivos de texto con las propiedades indicadas en la Tabla 7.2.

Texto	Tamaño(MB)	Alfabeto	Inv. Prob. Coincid.
Archivos MIDI	50	65	39.75
Cadenas de ADN	50	5	3.91
Texto en inglés	50	37	15.25

Tabla 7.2: Archivos de texto usados en el experimento.

7.3. Procedimiento

En este experimento, al igual que en el caso teórico, nos interesa medir el comportamiento esperado bajo el supuesto de texto promedio. Tratándose de una evaluación práctica, esto lo haremos como sigue:

- Para estimar el valor esperado de una medición asociada a la ejecución de un algoritmo probabilista(por ejemplo el tiempo), cada ejecución del algoritmo se repetirá 50 veces y se tomará el promedio de las mediciones asociadas.
- El sentido de texto promedio que utilizaremos es el de "texto representativo para alguna aplicación". Los textos del repositorio son, en ese sentido, una muestra adecuada, ya que contienen concatenaciones de muchas muestras representativas.

Además, para evitar la elección de patrones ventajosos, se repetirá la búsqueda sobre varios patrones. Ante la falta de un conjunto de patrones razonables para búsqueda, se

eligieron varios patrones presentes en forma exacta en el texto y varios patrones generados aleatoriamente (tipo Bernoulli uniforme, utilizando el alfabeto del texto como alfabeto de muestreo) para cada medición realizada. El número de patrones será el suficiente como para que las mediciones promedio sean razonablemente "suaves".

Luego, si $\mathcal{A}$ es un algoritmo para búsqueda aproximada (en cualquiera de sus versiones) y *params* representa una configuración de sus parámetros, los valores que mediremos son:

- $T_{\text{text}}(\mathcal{A}, m, p, params)$: tiempo promedio de búsqueda de p patrones de tamaño m elegidos al azar como subpalabras del texto. Cada búsqueda de un patrón se repite 50 veces.
- $T_{\text{rnd}}(\mathcal{A}, m, p, params)$: tiempo promedio de búsqueda de p patrones de tamaño m elegidos al azar como palabras de largo m. Cada búsqueda de un patrón se repite 50 veces.
- $Acc_{\text{text}}(\mathcal{A}, p, m, params)$: fracción promedio de las ocurrencias reportadas al realizar búsqueda de p patrones de tamaño m elegidos al azar como subpalabras del texto. Cada búsqueda de un patrón se repite 50 veces. Esta medición sólo tiene sentido para los algoritmos propuestos, no para sus versiones originales.
- $Acc_{\text{rnd}}(\mathcal{A}, m, p, params)$: fracción promedio de las ocurrencias reportadas al realizar búsqueda de p patrones de tamaño m elegidos al azar como palabras de largo m. Cada búsqueda de un patrón se repite 50 veces. Esta medición sólo tiene sentido para los algoritmos propuestos, no para sus versiones originales. Si no hay ocurrencias, entonces este valor no está definido.

Utilizamos patrones de tamaños $m = 50$ y $m = 100$ para la búsqueda. Además, fijamos $p = 50$. Las siguientes fueron las configuraciones utilizadas en cada algoritmo:

1. **Q**: fijamos $q = 4$ como tamaño de los q-gramas. Normalmente, $q = 4$ ó $q = 5$ se utilizan en la práctica.
2. **Q-PE**: fijamos $q = 4$. El número de q-gramas muestreados por ventana varía desde $c = 2$ hasta $c = 5$ para $m = 50$ y desde $c = 2$ hasta $c = 9$ para $m = 100$. Utilizamos $F = 0.7$. Este último parámetro fue determinado a partir de una exploración preliminar.
3. **CM**: el único parámetro configurable es el tamaño de los l-gramas. Fijamos $l = 4$. Normalmente, $l = 4$ ó $l = 5$ se utilizan en la práctica.
4. **CM-PE**: fijamos $l = 4$. El número de l-gramas muestreados por ventana varía desde $c = 2$ hasta $c = 5$ para $m = 50$ y desde $c = 2$ hasta $c = 10$ para $m = 100$. Utilizamos $\epsilon = 0.2$ y $F = 0.3$. Estos últimos parámetros fueron determinados a partir de una exploración preliminar, donde se evaluaron otros valores de ϵ y F.

7.4. Evaluación del algoritmo de q-gramas

En el caso del algoritmo de q-gramas, tanto **Q** como **Q-PE** son eficientes para el rango de valores utilizados (es decir, no producen muchas verificaciones). La única excepción es el caso de cadenas de ADN, donde sólo para $m = 50$ y $k = 5$ el algoritmo resulta eficiente. Esto se debe a la tolerancia del filtro de q-gramas.

Cuando consideramos patrones aleatorios, en todos los experimentos realizados el patrón no aparece en el texto. Por ello, en este caso sólo tiene sentido medir el tiempo de ejecución.

7.4.1. Dependencia sobre los parámetros

Tanto el tiempo de ejecución como las ocurrencias no reportadas cambian drásticamente en función del parámetro F. Para dar una idea de esto, la Tabla 7.3 muestra que sucede con el tiempo de ejecución y la pérdida de ocurrencias cuando fijamos $c = 2, m = 50$. Para las colecciones MIDI e inglés fijamos $k = 8$ y para la colección ADN fijamos $k = 5$.

	T_{rnd}/T_{text} **Q-PE** $F = 0.3$	T_{rnd}/T_{text} **Q-PE** $F = 0.7$	T_{rnd}/T_{text} **Q**	*Acc* **Q-PE** $F = 0.3$	*Acc* **Q-PE** $F = 0.7$
ADN	41.6/46	7.1/7.3	26.5/28	100 %	94.6 %
Inglés	5.4/6.7	2.4/2.6	49.8/45	100 %	99.4 %
MIDI	2.4/2.6	2.5/2.3	63.4/68.4	100 %	99.8 %

Tabla 7.3: Dependencia en F. El único valor de *Acc* reportado corresponde al de patrones no aleatorios. El tiempo está medido en segundos.

La dependencia sigue siendo de la misma naturaleza para otros valores de m y k. En todo lo que sigue, fijaremos $F = 0.7$. Este valor es óptimo en en el sentido de tiempo de ejecución para todas las colecciones. Moviendo el parámetro c podemos cambiar la pérdida de calces.

En general, la diferencia de tiempo de ejecución del algoritmo no varía más allá de un 20 % si se considera texto aleatorio o no. La Tabla 7.3 muestra lo que sucede con el tiempo de ejecución y la pérdida de calces para patrones aleatorios y no aleatorios, cuando $F = 0.7$.

El tiempo del algoritmo crece a medida que aumenta c. En cuanto a la pérdida de ocurrencias, ésta tiende a disminuir a medida que aumenta c. La tabla 7.4 muestra lo que sucede con el tiempo de ejecución y la pérdida de ocurrencias cuando fijamos $F = 0.7$, $m = 50$ y movemos c. Para las colecciones MIDI e inglés fijamos $k = 8$ y para la colección ADN fijamos $k = 5$.

	T_{text} ADN	T_{text} Inglés	T_{text} MIDI	*Acc* ADN	*Acc* Inglés	*Acc* MIDI
$c = 2$	7.3	3.5	2.3	95.5 %	99.5 %	99.9 %
$c = 3$	9.2	4.1	4	97 %	100 %	100 %
$c = 4$	10	4.9	4.7	100 %	100 %	100 %

Tabla 7.4: Dependencia en c. El tiempo está medido en segundos.

7.4.2. Resultados

Para el algoritmo **Q-PE**, la figura 7.1 muestra la relación existente entre tiempo de ejecución y pérdida de ocurrencias para diversas combinaciones de k y m, cuando se utilizan patrones no aleatorios. También se muestran los tiempos de ejecución del algoritmo **Q**.

En general, se observa que para los tamaños de patrones m y errores k utilizados, el algoritmo pierde ninguna o muy pocas ocurrencias.

Uno de los resultados más interesantes es que un valor pequeño de c produce muy buenos resultados en general. Ya con $c = 2$, y $F = 0.7$ se obtiene una pérdida de calces inferior al 1 % en el caso de texto en inglés y archivos MIDI.

Otro resultado interesante es que el tiempo de ejecución no varía drásticamente si se considera el caso de patrones aleatorios. Las diferencias son menores al 20 % en todos los casos. Además, desde el punto de vista de la pérdida de calces, estos patrones no presentan interés, ya que en todos los casos considerados un patrón aleatorio no registra apariciones aproximadas.

Finalmente, el tiempo de ejecución de **Q-PE** se reduce significativamente con respecto al tiempo de ejecución de **Q**, lo cual es provocado por el menor número de verificaciones realizadas.

7.5. Evaluación del algoritmo Chang y Marr

En el caso del algoritmo de Chang y Marr, tanto **CM** como **CM-PE** son eficientes para el rango de valores utilizados.

Cuando consideramos patrones aleatorios, en todos los casos el patrón no aparece en el texto. Por ello, en este caso sólo tiene sentido medir el tiempo de ejecución.

7.5.1. Dependencia sobre los parámetros

Tanto el tiempo de ejecución como las ocurrencias no reportadas cambian significativamente en en función del parámetro F. En este caso se produce un fenómeno muy similar al caso de q-gramas. En todo lo que sigue, fijaremos $F = 0.7$. Este valor es óptimo en en el sentido de tiempo de ejecución para todas las colecciones. Moviendo el parámetro c podemos cambiar la pérdida de calces.

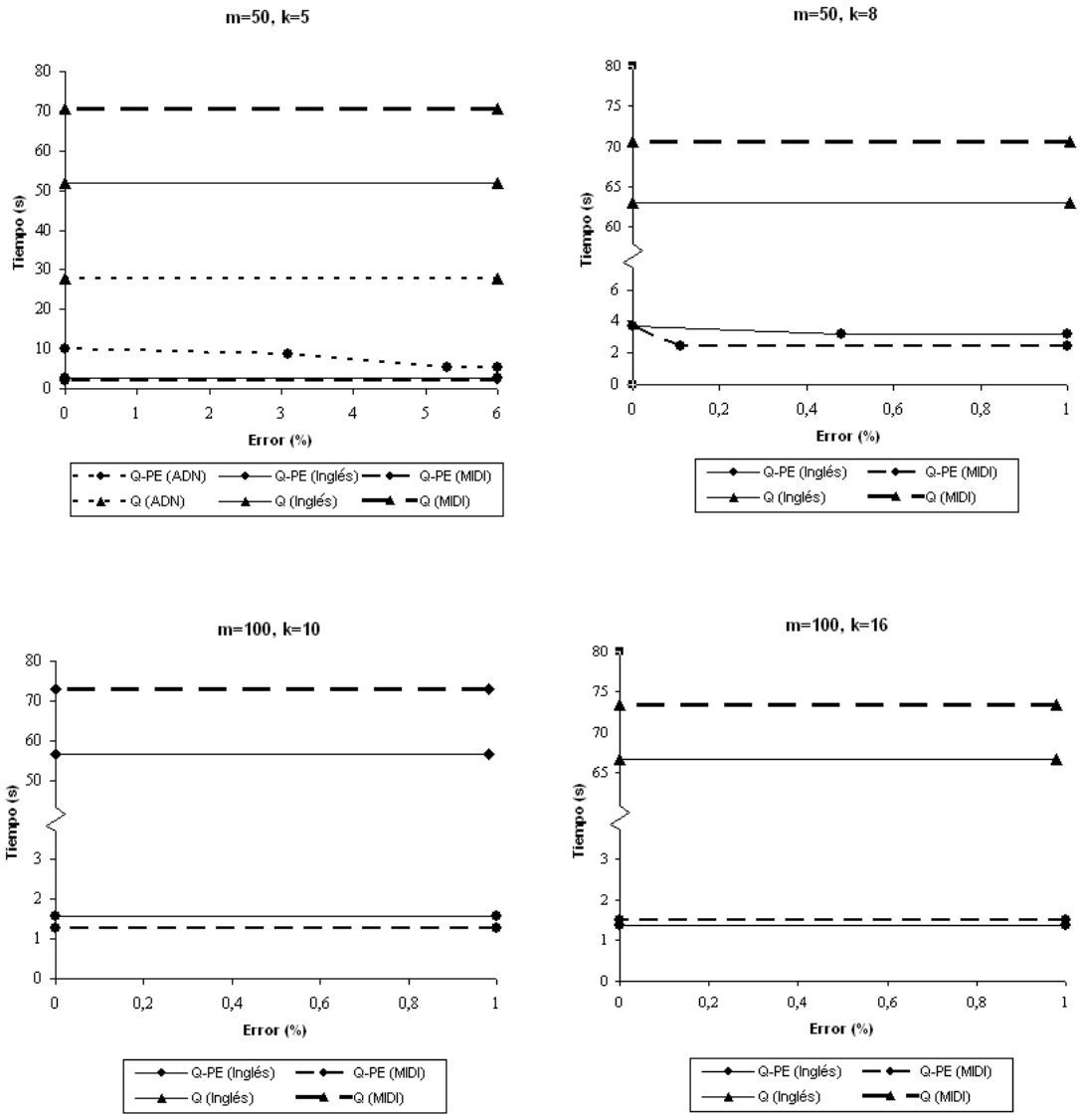

Figura 7.1: Tiempo de ejecución requerido por el algoritmo **Q-PE** para lograr un nivel de pérdida de ocurrencias (error) dado. También se indica el tiempo de ejecución del algoritmo **Q**.

En cuanto a la dependencia en el parámetro ϵ, dado que fijamos $l = 4$, no hay muchas formas de elegir ϵ. Además, si $\epsilon > 0.4$ el algoritmo **CM-PE** toma un tiempo muy superior a **CM**. Por ello, tomamos $\epsilon = 0.2$, que es el único valor que entrega resultados interesantes.

El tiempo del algoritmo tiene un comportamiento creciente al variar c. En general, la diferencia de tiempo de ejecución del algoritmo no varía más allá de un 20 % si se considera texto aleatorio o no. En cuanto a la pérdida de ocurrencias (o error), éste tiende a disminuir a medida que aumenta c.

7.5.2. Resultados

Para el algoritmo **CM-PE**, la Figura 7.2 muestra la relación existente entre tiempo de ejecución y pérdida de ocurrencias para diversas combinaciones de k y m, cuando se utilizan patrones no aleatorios. También se muestran los tiempos de ejecución del algoritmo **CM**.

Este algoritmo no resulta tan eficiente como **Q-PE**. Además del hecho que la probabilidad de perder calces es mucho más alta (incluso para valores grandes de c, su tiempo de ejecución y probabilidad de perder ocurrencias es mucho menos predecible. Una ventaja de este algoritmo es que es tolerante a todos los parámetros de m y k utilizados.

Otro resultado interesante es que el tiempo de ejecución no varía drásticamente si se considera el caso de patrones aleatorios. Las diferencias son menores al 20 % en todos los casos. Además, desde el punto de vista de la pérdida de calces, estos patrones no presentan interés, ya que en todos los casos considerados un patrón aleatorio no registra apariciones aproximadas.

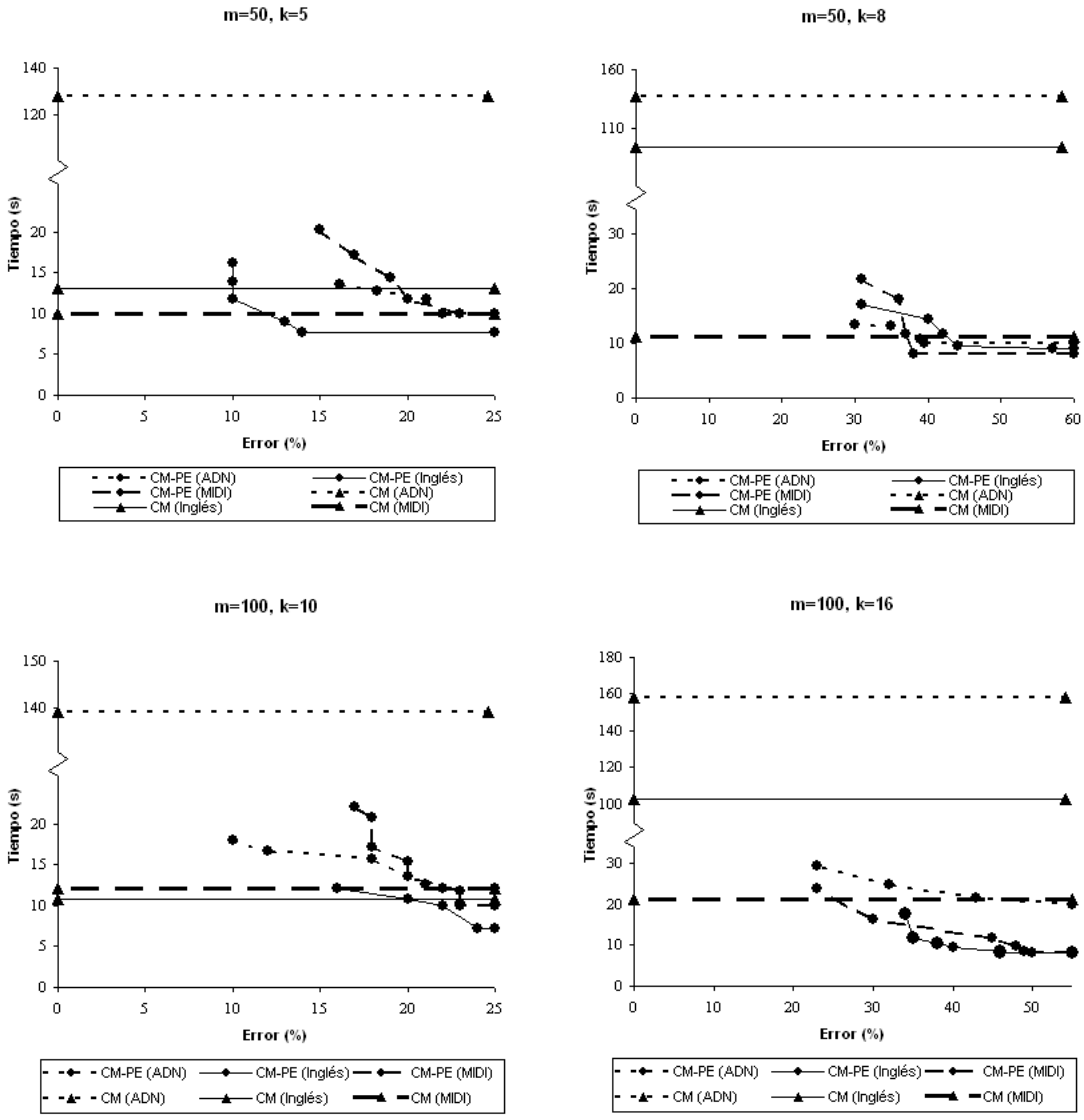

Figura 7.2: Tiempo de ejecución requerido por el algoritmo **CM-PE** para lograr un nivel de pérdida de ocurrencias (error) dado. También se indica el tiempo de ejecución del algoritmo **CM**.

Capítulo 8

Trabajo futuro

En este capítulo resumimos algunas ideas que creemos podrían resultar interesantes como tema de estudio. Algunas de ellas fueron estudiadas brevemente durante este trabajo; otras, simplemente quedaron como interrogantes.

8.1. Complejidad del problema

Un aspecto importante acerca de cualquier problema algorítmico es el estudio de su complejidad. En el caso de búsqueda aproximada, la medida usual de complejidad es la denominada complejidad promedio, entendida como el mínimo tiempo promedio de ejecución de un algoritmo determinista que resuelve el problema de búsqueda aproximada.

Esencialmente, la técnica para demostrar cotas inferiores en la complejidad promedio del problema de búsqueda aproximada en texto y muchas de sus variantes se basa directamente en los resultados de Yao para búsqueda exacta. En [Yao77] se demuestra que la complejidad promedio de la búsqueda exacta de un patrón aleatorio de largo m sobre un texto aleatorio de largo n asintóticamente coincide con la complejidad promedio de consulta, es decir, con la mínima cantidad de caracteres del texto que son consultados para entregar la respuesta, que es $O(n \log m/m)$. La demostración de este teorema es muy técnica, sin embargo, un argumento muy simple basado en este resultado permite determinar la complejidad de consulta del problema de búsqueda aproximada [CM94] y del problema de búsqueda aproximada múltiple [FN04]. En ambos casos se verifica que la complejidad de consulta coincide con la complejidad temporal.

La complejidad promedio del problema de búsqueda aproximada permitiendo errores no puede tratarse directamente bajo el esquema descrito en el párrafo anterior. Una razón es el hecho que los algoritmos en este contexto son probabilistas, por lo que es necesario definir una noción de complejidad promedio esperada. Además, una noción de complejidad para este problema necesariamente debe incluir el hecho que la probabilidad de perder calces está acotada.

Para subsanar estas dificultades, pero preservando las ideas utilizadas en otros trabajos relacionados, proponemos utilizar una medida de complejidad basada en el número

de caracteres del texto que son consultados por los algoritmos. Si $\mathcal{A}$ es un algoritmo probabilista para búsqueda aproximada permitiendo errores denotamos por $N_{\mathcal{A}}(n,m,k)$ el promedio, sobre todos los textos de tamaño n y todos los patrones de tamaño m, del número esperado de caracteres revisados del texto al realizar una búsqueda aproximada de distancia k. Además, denotamos por $E_{\mathcal{A}}(n,m,k)$ a la máxima probabilidad de perder un calce, sobre todas las búsquedas posibles. Finalmente, definimos $C(n,m,k,\epsilon)$ como el mínimo de $N_{\mathcal{A}}(n,m,k)$ sobre todos los algoritmos $\mathcal{A}$ tal que $E_{\mathcal{A}}(n,m,k) \leq \epsilon$.

En nuestro modelo, el tiempo promedio esperado de cualquier algoritmo de búsqueda aproximada permitiendo errores, con una probabilidad de perder un calce no superior a ϵ es al menos $C(n,m,k,\epsilon)$. Además, por los resultados vistos en este trabajo, $C(n,m,k,p) = O(n \log m/m)$ para un gran espectro de valores de p, en particular para p constante.

Algunos resultados útiles en demostraciones de complejidad para problemas de búsqueda en texto se verifican para la definición de complejidad introducida, por ejemplo, que podemos reducirnos al caso en que el texto es de tamaño $2m$.[1]

Lema 8.1.1. $C(n,m,k,\epsilon) \geq \frac{n}{2m} C(2m,m,k,\epsilon)$.

Demostración. Consideremos el problema más sencillo de encontrar sólo las ocurrencias incluidas completamente en las ventanas $T_{1..2m}$, $T_{2m+1..4m}$, $T_{4m+1..6m}$, ... Dado que el modelo de texto aleatorio es Bernoulli uniforme, la consulta de un carácter incluida en una ventana sólo aporta información para encontrar las ocurrencias en dicha ventana. Luego, sin perder generalidad podemos asumir que un algoritmo para este problema realiza un trabajo independiente por ventana. Su costo por lo tanto es al menos $\frac{n}{2m} C(2m,m,k,\epsilon)$. ■

También podemos relacionar la complejidad de la búsqueda aproximada permitiendo errores con la complejidad de la búsqueda exacta permitiendo errores:

Lema 8.1.2. *Existe c_σ dependiendo únicamente de σ tal que para todo $k < c_\sigma m$, $s = O(m)$*

$$C(2m,m,k,\epsilon) \geq \frac{1}{s} C(2m,m,0,\epsilon^s)$$

cuando m es suficientemente grande.

Demostración. Sea $\mathcal{A}$ un algoritmo para búsqueda aproximada permitiendo errores. Definamos el siguiente algoritmo $\mathcal{A}'$, que busca ocurrencias exactas del patrón en el texto.

proc $\mathcal{A}'(T,P,n,m)$
  $S = \varnothing$
  for $i = 1$ **to** s
    Ejecutar $\mathcal{A}(T,P,n,m,k)$. Agregar a S las ocurrencias reportadas
  end
  Verificar cada posición en S, reportando aquellas que correspondan a $k = 0$.
end

[1] Resultados similares en problemas de búsqueda en texto se encuentran, por ejemplo, en [Yao77].

Es claro que $E_{\mathcal{A}'}(n,m,0) \leq E_{\mathcal{A}}(n,m,k)^s$. Estudiemos $N_{\mathcal{A}'}(2m,m,0)$. Para ello, consideremos separadamente las dos partes principales del algoritmo $\mathcal{A}'$, específicamente, el ciclo de llamadas al algoritmo $\mathcal{A}$ y las verificaciones de los elementos en S:

- Es claro que el número de caracteres promedio esperado que son revisados al realizar las s llamadas al algoritmo $\mathcal{A}$ es $sN_{\mathcal{A}}(2m,m,k)$.
- En [Nav01] se demuestra que para $k \leq c_\sigma m$, con $c_\sigma = 1 - e/\sigma$, la probabilidad que una posición de texto sea término de una ocurrencia decrece exponencialmente con m. Luego, para $n = 2m$, el número promedio esperado de elementos de S es exponencialmente decreciente en m, y por lo tanto el número de caracteres promedio revisados en las verificaciones es $o(1)$.

De lo anterior se concluye que $N_{\mathcal{A}'}(2m,m,0) = sN_{\mathcal{A}}(2m,m,k) + o(1)$, y por lo tanto $sC(2m,m,k,\epsilon) \geq C(2m,m,0,\epsilon^s)$ para todo m suficientemente grande. De aquí sigue directamente el resultado. ■

Muchas demostraciones acerca de cotas inferiores para algoritmos probabilistas utilizan el método de Yao. Esta técnica permite hallar cotas inferiores para algoritmos probabilistas a partir de cotas inferiores para algoritmos deterministas. En nuestro caso, el uso de esta técnica da origen a un problema que podría ser interesante.

Nos limitaremos a estudiar el caso $k = 0$ y $n = 2m$. La idea de la reducción de Yao es que todo algoritmo probabilista que entrega un subconjunto de calces válidos se puede ver como una distribución sobre algoritmos deterministas que entregan un subconjunto de calces. Ello pues, si fijamos la parte aleatoria (es decir, los valores de sus lanzamientos de moneda) se obtiene un algoritmo determinista que entrega un subconjunto de calces.

Nuevamente introduciremos algunas definiciones. Para P un patrón fijo de largo m, definamos $\mathcal{A}^P = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_d\}$ como el conjunto de todos los algoritmos deterministas tales que en cada texto de largo $2m$ entregan un subconjunto de $\{1,2,\ldots,2m\}$ en el que cada uno de sus elementos es posición de término de un calce exacto. Sea además $\{T_1, T_2, \ldots, T_t\}$, con $t = \sigma^{2m}$, el conjunto de todos los textos de largo $2m$.

Observación. El conjunto $\mathcal{A}^P$ se puede asumir finito si consideramos algoritmos cuyo tiempo de ejecución de peor caso es a lo más 2^t. Esto no afecta el argumento.

Definamos $T(\mathcal{A}_i, T_j)$ como el tiempo del algoritmo $\mathcal{A}_i$ en el texto T_j, $\mathrm{N}(\mathcal{A}_i, T_j)$ como el número de ocurrencias de P en T_j no reportadas por $\mathcal{A}_i$ y *occ* como el número total de ocurrencias del patrón en todos los textos de largo $2m$.

Finalmente, definamos también

$$E(\mathcal{A}_i, T_j) = \frac{\mathrm{N}(\mathcal{A}_i, T_j)t}{occ}$$

y

$$F(\mathcal{A}_i, T_j) = T(\mathcal{A}_i, T_j) + E(\mathcal{A}_i, T_j) \log m.$$

Sea D una distribución de probabilidad sobre los algoritmos deterministas en $\mathcal{A}^P$. Esta distribución genera un algoritmo probabilista para el problema de búsqueda aproximada permitiendo errores (con un patrón fijo), que por comodidad seguiremos llamando D. Entonces $\text{Avg}_j \mathbb{E}_{\mathcal{A}_i \sim D} T(\mathcal{A}_i, T_j)$ es el tiempo promedio esperado del algoritmo probabilista D y $\text{Avg}_j \mathbb{E}_{\mathcal{A}_i \sim D} E(\mathcal{A}_i, T_j)$ es el valor esperado de $\sum_j N(\mathcal{A}_i, T_j)/occ$, el cual corresponde a la fracción esperada de los calces no reportados por el algoritmo probabilista D cuando se realiza una búsqueda sobre cada texto de largo $2m$. Entonces se tiene el siguiente lema, cuya demostración es directa.

Lema 8.1.3. *Supongamos que existe $a > 0$ tal que para todo $i \in \{1, \ldots, d\}$ y todo m suficientemente grande se cumple $\text{Avg}_j F(\mathcal{A}_i, T_j) \geq a \log m$. Entonces, para todo $p \leq a/2$ y para toda distribución de probabilidad D sobre $\{\mathcal{A}_1, \ldots, \mathcal{A}_d\}$:*

$$\operatorname*{Avg}_j \mathbb{E}_{\mathcal{A}_i \sim D} E(\mathcal{A}_i, T_j) \leq p \Rightarrow \operatorname*{Avg}_j \mathbb{E}_{\mathcal{A}_i \sim D} T(\mathcal{A}_i, T_j) \geq \frac{a}{2} \log m.$$

Ahora supongamos cierta la hipótesis del Lema 8.1.3. Sea D un algoritmo probabilista cuya probabilidad de perder un calce es a lo más $p \leq a/2$. Entonces, la fracción esperada de calces no reportados (y en consecuencia su promedio sobre todos los textos) es a lo más p. Luego, $\text{Avg}_j \mathbb{E}_{\mathcal{A}_i \sim D} E(\mathcal{A}_i, T_j) \leq p$. Aplicando el lema se concluye que el tiempo esperado es $\Omega(\log m)$, lo que resultaría ser una cota inferior en el número de caracteres consultados para un algoritmo que resuelve el problema de búsqueda con errores con $k = 0$ y probabilidad de perder calces a lo más $p \leq a/2$.

Finalmente, es directo ver que la hipótesis del Lema 8.1.3

$$\forall i \in \{1, \ldots, d\} : \operatorname*{Avg}_j F(\mathcal{A}_i, T_j) \geq a \log m$$

es equivalente a demostrar que todo algoritmo determinista $\mathcal{A}$ que reporta una fracción al menos s de la totalidad de ocurrencias (es decir, sumando las ocurrencias para cada patrón) debe tener una complejidad temporal al menos $(a - (1 - s)) \log m$. En efecto, la hipótesis es equivalente a demostrar que para todo algoritmo determinista $\mathcal{A}$

$$\operatorname*{Avg}_j T(\mathcal{A}, T_j) \geq a \log m - \operatorname*{Avg}_j E(\mathcal{A}, T_j),$$

donde el lado izquierdo no es más que el tiempo promedio del algoritmo $\mathcal{A}$ y

$$\operatorname*{Avg}_j E(\mathcal{A}, T_j) = \sum_j \frac{\text{N}(\mathcal{A}_i, T_j)}{occ}$$

es la fracción de la totalidad de las ocurrencias que no son reportadas.

Si la hipótesis del Lema 8.1.3 es cierta, entonces se reduciría la demostración de una cota inferior para algoritmos (probabilistas) de búsqueda con errores a una cota inferior para algoritmos de búsqueda deterministas que entregan sólo un subconjunto de ocurrencias.

8.2. Otra posible perspectiva

En este trabajo hemos planteado la búsqueda aproximada permitiendo errores esencialmente como el sacrificio de precisión en función del tiempo. Sin embargo, el carácter sublineal de los algoritmos propuestos nos permiten ver esto de una manera alternativa, que es, entregar una respuesta aproximada al problema de búsqueda dado que tenemos una capacidad limitada de explorar la entrada. Esto tiene relación con el concepto de testeo de propiedades [GGR98], donde se estudia lo que es posible decir cuando sólo se puede consultar una pequeña fracción de la entrada.

El problema que dio origen a esta área es el de funciones no explícitas. Supongamos que se dispone de un programa que entrega los valores de una función ante cualquiera de sus entradas. El problema consiste en determinar propiedades de esta función que pueden determinarse sólo a través de consultas al programa. En particular, Rubinfeld y Sudan [RS96] estudiaron el test de polinomialidad de una función de varias variables sobre un cuerpo finito.

En testeo de propiedades, el desconocimiento de la entrada completa se compensa normalmente con aleatoriedad: los algoritmos que se generan consultan posiciones aleatorias de la entrada. Si bien la definición del problema de testeo de propiedades no la revisaremos aquí, sí señalamos que una de sus limitaciones es que sólo estudia problemas de decisión. En particular, el problema de búsqueda aproximada no cae directamente en este esquema. Sería interesante plantear una extensión de este concepto a problemas más generales y estudiar problemas como el de búsqueda aproximada de acuerdo a esta extensión.

8.3. Aspectos prácticos

Los experimentos realizados en este trabajo muestran resultados interesantes respecto al tiempo de ejecución y la pérdida de ocurrencias de dos de los algoritmos propuestos. Sin embargo, los resultados no nos permiten concluir, por ejemplo, qué valores debemos asignar a los parámetros de cada algoritmo cuando se toma un patrón de largo m, un error k y un alfabeto de tamaño σ. Dado que la probabilidad de perder calces depende del tipo de texto usado en la aplicación, un estimador de los parámetros podría obtenerse probando el algoritmo con diversos parámetros sobre textos y patrones representativos de la aplicación.[2] La probabilidad de perder calces puede aproximarse como el promedio de la fracción de calces reportados en cada búsqueda.

Otro aspecto no estudiado directamente es la tolerancia de los filtros modificados, es decir, para qué valores de k el filtro utilizado es efectivo para descartar posiciones del texto. Desde el punto de vista teórico, en este trabajo se muestra que las cotas conocidas para la tolerancia de cada filtro se extienden a los algoritmos estudiados, pero desde el punto de vista práctico, no es claro si la tolerancia experimental coincide.

Finalmente, los algoritmos implementados en este trabajo carecen, intencionalmente, de optimizaciones. Una aplicación práctica no sólo debe incluir una programación cuida-

[2] Los textos no tienen que ser necesariamente tan grandes como en la aplicación real.

dosa, sino también un diseño adecuado de partes de los algoritmos que, por simplicidad, no fueron detallados en este trabajo.

Capítulo 9

Conclusiones

En este trabajo se estudió el problema de búsqueda aproximada en un contexto de recursos muy limitados, con la motivación práctica de encontrar algoritmos eficientes para el problema y la motivación teórica de avanzar en la comprensión de lo que es posible hacer en este problema cuando limitamos el uso de recursos computacionales.

Nuestro trabajo propone un nuevo marco para resolver el problema de búsqueda aproximada, que denominamos "búsqueda aproximada permitiendo errores". En este esquema, la salida de cualquier algoritmo de búsqueda debe entregar sólo un subconjunto de las ocurrencias, pero con la garantía de que la probabilidad del algoritmo de reportar cada ocurrencia sea mayor que algún valor a precisar.

Esta relajación admite potenciales aplicaciones y permite encontrar algoritmos muchos más eficientes que los tradicionales. A nivel práctico, con una probabilidad de perder calces ajustada adecuadamente, creemos que muchas de las aplicaciones que utilizan algoritmos tradicionales de búsqueda podrían recurrir a estos algoritmos: la búsqueda aproximada ya de por sí implica un modelo aproximado de la realidad, por lo que la adición de un error extra puede ser admisible. Además, a diferencia de heurísticas y otras aproximaciones, el error derivado de la pérdida de calces es controlable.

Para el caso en línea del problema de búsqueda aproximada, presentamos tres algoritmos **CL-PE**, **Q-PE** y **CM-PE**. Todos ellos se encuentran motivados por las ideas de filtros para búsqueda, algoritmos que usan una rutina rápida (filtro) para descartar porciones de texto donde no pueden existir ocurrencias. El primero de los algorimos **CL-PE**, o algoritmo de caracteres lejanos, muestra la idea básica que luego se desarrolla múltiples veces en este trabajo. Esencialmente, esta idea es usar muestreo aleatorio para obtener estimaciones de valores calculados exactamente por los filtros. Esto conlleva una reducción de tiempo a cambio de errores producto del uso de estimaciones en reemplazo de valores exactos. Teóricamente, demostramos que el tiempo de ejecución promedio del filtro disminuye respecto al tiempo del filtro original, lo que se traduce a posteriori en una significativa disminución del tiempo de ejecución del algoritmo completo.

Los algoritmos **Q-PE** y **CM-PE** presentan un mayor interés desde el punto de vista teórico y práctico que **CL-PE**. Ambos se basan en ideas que han dado origen a algoritmos

muy eficientes para búsqueda aproximada. Comparando con algoritmos similares para búsqueda aproximada, nuestros algoritmos exhiben una considerable disminución del tiempo de ejecución (teórico) especialmente cuando k es grande. Por ejemplo, **CM-PE** tiene un tiempo promedio de ejecución $O(n \log_\sigma m/m)$ con una probabilidad de perder un calce de $O(k/m)$. En contraparte, el algoritmo de búsqueda aproximada más similar tiene un tiempo de ejecución promedio $O(n(k + \log_\sigma m)/m)$, que además es la complejidad promedio del problema de búsqueda aproximada. Desde el punto de vista práctico, nuestros experimentos corroboran lo obtenido teóricamente, en el sentido de que la reducción de tiempo de ejecución es significativa con muy pocas pérdidas de ocurrencias.

También extendimos las ideas vistas a otros problemas de búsqueda aproximada en texto. Formalizamos la noción de búsqueda aproximada permitiendo errores al caso de múltiples patrones. Aquí, utilizando las mismas ideas exhibidas en **Q-PE** y **CM-PE** obtenemos algoritmos que nuevamente exhiben una significativa disminución del tiempo de ejecución (teórico) respecto a algoritmos similares, y menores que la complejidad temporal promedio de este problema.

Finalmente, presentamos algoritmos para el caso de texto indexado. Aquí, el espacio utilizado por el índice juega un rol fundamental, por lo que las directrices seguidas para desarrollar los algoritmos necesariamente tienen que considerar este factor. Nuestra propuesta aquí fue modificar algoritmos para búsqueda aproximada de modo de acelerar las búsquedas a cambio de pérdidas de ocurrencias, pero con la restricción adicional de conservar el tamaño del índice.

Apéndice A

Una demostración complementaria

En este apéndice se demuestra un resultado que se utilizó en las Secciones 4.5 y 5.2.2. Este resultado es una extensión directa de un teorema demostrado en [CM94].

A.1. Enunciado y demostración de la proposición

Teorema A.1.1. *Sea P una palabra de largo m y $u > 0$. Entonces existen constantes $\alpha > 0$ y $\epsilon > 0$ tales que:*

$$\mathbb{P}_{S \leftarrow \Sigma^l}\left(\mathrm{asm}(S,P) \leq \epsilon l\right) \leq m^{-u},$$

con $l = \alpha \log_\sigma m$.

Demostración. (Sólo el caso $\sigma > 8$)

Ses S un l-grama tal que $\mathrm{asm}(S,P) \leq \epsilon l$, entonces existe alguna subpalabra W de P tal que $d(S,W) \leq \epsilon l$. Consideremos una secuencia de a lo más ϵl operaciones que transforma S en W. Es claro que los caracteres de S no borrados o sustituidos por estas operaciones aparecerán en W en las mismas posiciones relativas. Por lo tanto, S y W comparten una subsecuencia de tamaño $l - \epsilon l$. El mismo argumento demuestra que S y W comparten una subsecuencia de tamaño $l' - \epsilon l$, donde l' es el largo de W. Luego, existe un l-grama de P que comparte con S una subsecuencia de largo $l - \epsilon l$. En efecto, si $l' < l$, basta con extender W. Si por el contrario $l' > l$, entonces S y W comparten una subsecuencia de tamaño $l' - \epsilon l$ por lo que S y $W_{1..l}$ comparten una subsecuencia de tamaño $l' - \epsilon l - (l' - l) = l - \epsilon l$. De aquí se deduce que

$$\mathbb{P}_{S \leftarrow \Sigma^l}\left(\mathrm{asm}(S,P) \leq \epsilon l\right) \leq \sum_{i=1}^{m-l+1} \mathbb{P}_{S \leftarrow \Sigma^l}\left(\mathrm{lcs}(S, P_{i..i+l-1}) \geq l - \epsilon l\right),$$

donde $\mathrm{lcs}(A,B)$ denota la subsecuencia común más larga entre A y B. Es fácil demostrar que

$$\mathbb{P}_{S \leftarrow \Sigma^l}\left(\mathrm{lcs}(S, P_{i..i+l-1}) \geq l - \epsilon l\right) \leq \binom{l}{l-\epsilon l}^2 \sigma^{-(l-\epsilon l)}$$

y, usando la desigualdad $\binom{n}{k} < (ne/k)^k$ se obtiene que

$$\mathbb{P}_{S \leftarrow \Sigma^l}\left(\mathrm{asm}(S,P) \leq \epsilon l\right) \leq m\left(\frac{\sigma(1-\epsilon)^2}{e^2}\right)^{-(l-\epsilon l)}.$$

Finalmente, para $\sigma > e^2$, podemos fijar $0 < \epsilon < 1$ y $0 < v < 1$ tal que

$$\sigma^v \equiv \frac{\sigma(1-\epsilon)^2}{e^2} > 1$$

y por lo tanto

$$\mathbb{P}_{S \leftarrow \Sigma^l}\left(\mathrm{asm}(S,P) \leq \epsilon l\right) \leq m\sigma^{-vl(1-\epsilon)}.$$

Eligiendo α de manera de que esta probabilidad sea menor o igual que m^{-u} se concluye el resultado. ■

Apéndice B

Código fuente

B.1. Algoritmos Q y Q-PE

```
#include "stdafx.h"
#include "utils.h"

typedef map<string,int> Wordmap;

/* Algoritmo Q
filename = nombre del archivo que contiene el texto
pattern = patron
k = error permitido
q = tamano q-gramas
*/
int Q(char* filename, char* pattern, int k, int q)
{
  char character;
  int filesize;

  //carga archivo de texto en memoria
  char* buffer = loadFile(filename, filesize);

  string text(buffer); //texto
  string pat(pattern); //patron
  int m = pat.length();
  int outputCount; //numero de ocurrencias entregadas por el algoritmo
  int numChar; //numero de caracteres
  long last_qgram;//id del ultimo qgrama leido
  long first_qgram;//id del primer qgrama leido en la ventana de largo n

  /* generacion de id's de q-gramas
  Cada q-grama Q es visto como un numero entero en base Sigma
  Q=Q1Q2..Qq es convertido en el numero idChar(Q1)idChar(Q2)..idChar(Qq)
  */
  vector<int> id; //id de cada q-grama
  numChar = idChar(text, pat, id);  //cada caracter tiene un id asociado

  char* prog = "myers";
  char* kStr = (char *) malloc( 10 );

  int numIter=1;
  for(int cont = 0; cont < numIter ; cont++)
  {
    set <int> output;

    //informacion acerca de los qgramas almacenados
    map<long,int> nTotalq_grams, nUsedq_grams, nNotUsedq_grams;

    //inicializacion: creacion del hash de q-gramas.
    for(int i = 0; i <= m-q; i++)
    {
      nTotalq_grams[ id_qgrams(pat.substr(i, q), id, numChar, q) ]++;
    }

    //lectura de qgramas.
    int count = 0;
    for(int s = 0; s <= filesize - q + 1; s++)
    {
```

```
      if( s >= m-k )
      {
        if( s == m-k )
        {
          last_qgram = id_qgrams(text.substr(0, q), id, numChar, q);
        }
        else
        {
          last_qgram = stepFunc2(last_qgram , id[ text[ s-(m-k)+q-1 ] ] , q , numChar);
        }
        if( nUsedq_grams [last_qgram] > 0 )
        {
          if( nNotUsedq_grams[ last_qgram ] > 0)
          {
            nNotUsedq_grams[ last_qgram ]--;
          }
          else
          {
            nUsedq_grams[ last_qgram ]--;
            count--;
          }
        }
      }
      if( s == 0 )
      {
        first_qgram = id_qgrams(text.substr(s, q), id, numChar, q);
      }
      else
      {
        first_qgram=stepFunc2(first_qgram , id[ text[s+q-1] ] , q, numChar);
      }
      if( nUsedq_grams[ first_qgram ] < nTotalq_grams[ first_qgram ] )
      {
        nUsedq_grams[ first_qgram ]++;
        count++;
      }
      else if( nTotalq_grams[ first_qgram ] > 0)
      {
        nNotUsedq_grams[ first_qgram ]++;
      }
      if( count >= m - k - k*q -q +1)
      {
        //ejecucion del algoritmo verificador en la ventana.
        string str;
        sprintf(kStr, "%d", k);
        str = text.substr( max(s + q - m - k,0), m + 2*k );
        char* text2 = strdup( str.c_str() );
        char* argvs[ 4 ];
        argvs[ 0 ] =prog; argvs[ 1 ] = strdup(pattern); argvs[ 2 ] = kStr; argvs[ 3 ] = text2;
        myersVerifier( 4, argvs, output,max( s + q - m - k, 0 ) );
        free(text2);
      }
    }

    //entrega salida
    set < int > :: iterator s;
    outputCount = 0;
    for ( s = output.begin(); s != output.end(); s++)
    {
      outputCount++;
    }
  }
  printf("QGramasNumOcc=%d;",outputCount);
  return outputCount;
}

/* Algoritmo Q-PE.
filename = nombre del archivo que contiene el texto
pattern = patron
k = error permitido
F = constante asociada al algoritmo
c = constante asociada al algoritmo
q = tamano q-gramas
numIter = numero de iteraciones que se ejecuta el algoritmo
realOutput = total de ocurrencias (se utiliza para determinar)
*/
void Q-PE(char* filename, char* pattern, int k, double F,
          int c, int q, int numIter, int realOutput)
{
  srand(time(NULL));
  int filesize;

  //carga archivo de texto en memoria
  char* buffer = loadFile(filename, filesize);

  string text(buffer);//texto
  string pat(pattern);  //patron
```

```
  int m = pat.length();
  int counter = 0;
  char* prog = "myers";
  char* kStr = ( char* ) malloc( 10 );

  for(int cont = 0; cont < numIter; cont++)
  {

    set <string> qgramsList; //lista de qgramas del patron
    set <int> output;

    for(int i = 0; i <= m-q; i++)
    {
      qgramsList.insert( pat.substr(i,q) );
    }

    int init_block_idx = 0; //inicio de la ventana
    int size = (m - k)/2;//tamano de la ventana
    int count;
    while (init_block_idx - size < filesize - m)
    {
      count = 0;
      for(int i = 0; i < c; i++)
      {
        //Se elige una posicion aleatoria de la ventana.
        int index = ( rand()%(size - q + 1) ) + 1;
        if ( qgramsList.find( text.substr( init_block_idx + index, q ) ) != qgramsList.end() )
        {
          count++;
        }
      }
      if( count > F * c )
      {
        //ejecucion del algoritmo verificador en la ventana.
        sprintf(kStr, "%d", k);
        string str = text.substr( max( init_block_idx - (int)( (m + 3*k + 1)/2 ) , 0 ),
        min( (3*m + 5*k)/2 + 1,  filesize - 2 - (int) max(init_block_idx - (m+3*k+1)/2 , 0) ) );
        char* text2 = strdup( str.c_str() );
        char* argvs[4];
        argvs[0] = prog; argvs[1] = strdup(pattern); argvs[2] = kStr; argvs[3] = text2;
        myersVerifier( 4, argvs, output,max(init_block_idx-(int)((m + k + 1)/2) , 0 )  );
        free(text2);
      }
      init_block_idx = init_block_idx + size;
    }
    //entrega salida
    set<int>::iterator s;
    for (s = output.begin() ; s != output.end() ; s++)
    {
      counter++;
    }
  }
  printf("QgramsPESuccess=%f%%",((counter)/(realOutput*numIter + 0.0) * 100));

}

long stepFunc2(long i, long j, int q, int numCharacter)
{
  return ((i%( (int) (pow(numCharacter,q-1)) ) )*numCharacter + j);
}
```

B.2. Algoritmos CM y CM-PE

```
#include "stdafx.h"
#include "utils.h"
#include "cm.h"

/* Algoritmo CM.
filename = nombre del archivo que contiene el texto
pattern = patron
k = error permitido
l = tamano l-gramas
*/
int CM(char* filename, char* pattern, int k, int l)
{
  int nResp;
  int filesize;

  //carga archivo de texto en memoria
  char* buffer = loadFile(filename, filesize);

  string text(buffer);  //texto
  string pat(pattern);  //patron
```

```
  int m = pat.length();

  int counter;//numero de ocurrencias
  int numChar;//numero total de caracteres distintos

  //generacion id's de l-gramas
  vector <int> id;
  vector <int> invId;
  numChar = idChar(text, pat, id);
  invIdChar(id, invId);
  map <string , int>  bestMatch;//matches aproximados (asm)

  //generacion funcion asm
  bestMatches(bestMatch, pat, l, id, invId, numChar);

  char* prog = "myers";
  char* kStr = ( char* ) malloc( 10 );
  int numIter=1;

  for(int cont = 0 ; cont < numIter ; cont++)
  {
    set< int > output;
    //contando el error acumulado en los intervalos
    int init_block_idx = 0;
    int size = (m - k)/2;
    int errorCount;
    while (init_block_idx + size < filesize )
    {
      errorCount = 0;
      int i;
      for(i = 0 ; i < ( size/l ) - 1 ; i++)
      {
        errorCount = errorCount + bestMatch[ text.substr(init_block_idx + i*l ,l) ];
        if( errorCount > k )
        {
          break;
        }
      }

      if( errorCount <= k )
      {
        //ejecucion del algoritmo verificador en la ventana.
        sprintf(kStr, "%d", k);
        string str = text.substr( max( init_block_idx - (int)( (m + 3*k + 1)/2 ) , 0 ),
        min( (3*m + 5*k)/2 + 1,  filesize - 2 - (int) max(init_block_idx - (m+3*k+1)/2 , 0) ) );
        char* text2 = strdup( str.c_str() );
        char* argvs[4];
        argvs[0] = prog; argvs[1] = strdup(pattern); argvs[2] = kStr; argvs[3] = text2;
        myersVerifier( 4, argvs, output,max(init_block_idx-(int)((m + k + 1)/2) , 0 ) );
        free(text2);
      }
      init_block_idx = init_block_idx + size;
    }

    //entrega salida
    set<int>::iterator s;
    counter = 0;
    for (s = output.begin() ; s != output.end() ; s++)
    {
      counter++;
    }
  }
  printf("cmNumOcc=%d;", counter);
  return counter/numIter;
}

/* Algoritmo CM-PE.
filename = nombre del archivo que contiene el texto
pattern = patron
k = error permitido
l = tamano l-gramas
numIter = numero de iteraciones que se ejecuta el algoritmo
epsilon = constante asociada al algoritmo
c = constante asociada al algoritmo
F = constante asociada al algoritmo
realOutput = total de ocurrencias (se utiliza para determinar)
*/
int CM-PE(char* filename, char* pattern, int k, int l, int numIter,
          double epsilon, int c, double F, int realOutput)
{
  int filesize;

  //carga archivo de texto en memoria
  char* buffer = loadFile(filename, filesize);

  string text(buffer);//texto
  string pat(pattern);//patron
```

```
  int m = pat.length();
  int counter;
  int numChar;

  //genero id's de l-gramas y la funcion de avance
  vector<int> id;
  vector<int> invId;
  numChar = idChar(text, pat, id);
  invIdChar(id, invId);
  map<string,int>  bestMatch;

  //genero funcion asm
  bestMatches(bestMatch, pat, l, id, invId, numChar);
  counter = 0;
  char* prog = "myers";
  char* kStr = ( char* ) malloc( 10 );
  for(int cont = 0 ; cont < numIter ; cont++)
  {
    set<int> output;
    //cuento el error en los l-gramas
    int init_block_idx = 0;
    int size = (m - k)/2;
    int count;
    while (init_block_idx +size < filesize )
    {
      count = 0;
      int i;
      for(i = 0 ; i < c  ; i++)
      {
        //Se elige una posicion aleatoria de la ventana.
        int idx = rand() %( size/l );
        if( bestMatch[ text.substr(init_block_idx + idx*l,l) ] <= epsilon*l )
        {
          count++;
        }
        if(count >= F * c)
        {
          break;
        }
      }
      if(count > F * c)
      {
        //ejecucion del algoritmo verificador en la ventana.
        sprintf(kStr, "%d", k);
        string str = text.substr( max( init_block_idx - (int)( (m + k + 1)/2 ) , 0 ),
        min( (3*(m - k))/2 + 1 ,  filesize - 2 - (int) max(init_block_idx - (m+k+1)/2 , 0) ) );
        char* text2 = strdup( str.c_str() );
        char* argvs[4];
        argvs[0] = prog; argvs[1] = strdup(pattern); argvs[2] = kStr; argvs[3] = text2;
        myersVerifier(4, argvs, output, max(init_block_idx-(int)((m + k + 1)/2) , 0 ) );
        free(text2);
      }
      init_block_idx = init_block_idx + size;
    }

    //entrega salida
    set<int>::iterator s;
    for ( s = output.begin() ; s != output.end() ; s++ )
    {
      counter++;
    }
  }

  printf("cmPESuccess=%f%%;",((counter)/(realOutput*numIter + 0.0) * 100));
  return counter/numIter;
}

void bestMatches (map<string,int>&  bestMatch, string pattern, int q,
                  vector<int>& id, vector<int>& invId,int numCharacter)
{

  int k1;
  int m = pattern.length();
  //generar todas las palabras de largo q
  for (long i = 0; i < pow(numCharacter,q); i++)
  {
    k1 = 2 * q;
    string s = q_gramOfId(i, invId, numCharacter, q);

    //calcular la minima distancia al patron
    int k2=editDistance2(pattern,s); //No incluida aqui.

    bestMatch[s] = k2;
  }
}
```

Bibliografía

[BEK+03] T. Batu, F. Ergun, J. Kilian, A. Magen, S. Raskhodnikova, R. Rubinfeld, and R. Sami. A sublinear algorithm for weakly approximating edit distance. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 316–324. ACM Press, 2003.

[BM77] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[BY92] R. Baeza-Yates. Text-retrieval: Theory and practice. In *IFIP 12th World Computer Congress*, volume 1, pages 465–476, 1992.

[BYN99] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[BYN00] R. Baeza-Yates and G. Navarro. New models and algorithms for multidimensional approximate pattern matching. *J. Discret. Algorithms*, 1(1):21–49, 2000.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[CCY03] L. Cheng, D. Cheung, and S. Yiu. Approximate string matching in DNA sequences. *In Proceedings of the 8th International Conference on Database Systems for Advanced Applications*, pages 303–310, 2003.

[CH98] R. Cole and R. Hariharan. Approximate string matching: a simpler faster algorithm. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 463–472, 1998.

[Che52] H. Chernoff. A measure of the asymptotic efficiency of tests of a hypothesis based on the sum of observations. *Annals of Mathmematical Statistics*, 23:493– 507, 1952.

[CL90] W. Chang and E. Lawler. Approximate string matching in sublinear expected time. In *Proceedings of the SIAM 31th Annual Symposium on Foundations of Computer Science*, volume 1, pages 116–124, 1990.

[CM94] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 259–273. Springer-Verlag, 1994.

[DM79] N. Dixon and T. Martin, editors. *Automatic Speech and Speaker Recognition.* John Wiley & Sons, Inc., 1979.

[Fel68] W. Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley and Sons, 1968.

[Fis96] G. Fishman, editor. *Monte Carlo Concepts, Algorithms and Applications.* Springer Verlag, 1996.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, 9, 2004.

[FN05] P. Ferragina and G. Navarro. Pizza & chili corpus compressed indexes and their testbeds. *http://pizzachili.dcc.uchile.cl/index.html*, 2005.

[FPS97] J. French, A. Powell, and E. Schulman. Applications of approximate word matching in information retrieval. In *Proceedings of the 6th International Conference on Knowledge and Information Management*, pages 9–15. ACM Press, 1997.

[GGR98] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of ACM*, 45(4):653–750, 1998.

[Gus97] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[Hoc97] D. Hochbaum. *Approximation algorithms for NP-hard problems.* PWS Publishing Co., Boston, MA, USA, 1997.

[Jan04] S. Janson. Large deviations for sums of partly dependent random variables. *Random Structure & Algorithms*, 24(3):234–248, 2004.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[KS94] S. Kumar and E. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.

[Kur96] S. Kurtz. Approximate string searching under weighted edit distance. In *Proceedings of the 3rd South American Workshop on String Processing (WSP '96)*, pages 156– 170. Carleton Univ. Press, 1996.

[Lev65] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. In *Problems of Information Transmission*, pages 8–17, 1965.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and Systems Science*, 37(1):63–78, 1988.

[LV89] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[Mel96] B. Melichar. String matching with k differences by finite automata. In *International Conference of Pattern Recognition '96*, pages 256–260, 1996.

[Mye86] G. Myers. Incremental alignment algorithms and their applications. Technical report, Dept. of Computer Science, Univ. of Arizona, 1986.

[Mye94] E. Myers. A sublinear algorithms for approximate keyword searching. *Algorithmica*, 12(4-5):345–374, October 1994.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav97] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proceedings of the 4th South American Workshop on String Processing*, pages 112–124. Carleton Univ. Press, 1997.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[NBY98] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electron. J.*, 1(2):273–282, 1998.

[NBY99a] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, pages 163–185. Springer-Verlag, 1999.

[NBY99b] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. In *Information Processing Letters*, volume 72, pages 65–70. Carleton Univ. Press, 1999.

[NBY00] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. Discret. Algorithms*, 1(1):205–239, 2000.

[NBYST01] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[NdMN+00] G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 49(1):444–453, 1970.

[RS96] R. Rubinfeld and M. Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2):252–271, 1996.

[Rub81] R. Rubinstein, editor. *Simulation and the Monte Carlo Method*. Wiley, 1981.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[Shi96] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proceedings of the 3rd South American Workshop on String Processing*, volume 35, pages 257– 271. Carleton University Press, 1996.

[SK83] D. Sankoff and J. Kruskal. *Time warps, String Edits, and Macromolecules*. Addison-Wesley, 1983.

[ST04] E. Sutinen and J. Tarhio. Approximate string matching with ordered q-grams. *Nordic J. of Computing*, 11(4):321–343, 2004.

[Tak94] T. Takaoka. Approximate pattern matching with samples. In *Proceedings of the 5th International Symposium on Algorithms and Computation*, pages 234–242. Springer-Verlag, 1994.

[TNB04] K. Taghva, T. Nartker, and J. Borsack. Information access in the presence of OCR errors. In *Proceedings of the 1st ACM workshop on Hardcopy document processing*, pages 1–8. ACM Press, 2004.

[TU93] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM J. Comput.*, 22(2):243–260, 1993.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6:132–137, 1985.

[Ukk92] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92:191–211, 1992.

[Wat95] M. Waterman. *Introduction to computational biology: maps, sequences and genomes*. Chapman y Hall, 1995.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. In *Communications of the ACM*, volume 35, pages 83–91, 1992.

[WMM96] S. Wu, U. Mamber, and E. Myers. A subquadratic algorithm for approximate limited expression matching. *Journal of Algorithms*, 19(1):346–360, 1996.

[Yao77] A. Yao. The complexity of pattern matching for a random string. Technical report, Stanford University, 1977.

[ZD96] J. Zobel and P. Dart. Phonetic string matching: Lessons from information retrieval. In *Proceedings of the 19th International Conference on Research and Development in Information Retrieval*, pages 166–172. ACM Press, 1996.