
Universidad Nacional de Rosario

Tesina de grado
para la obtención del grado de
Licenciado en Ciencias de la Computación

# String Matching Aproximado Mejorado con SIMD

*Autor:*
Fernando Jesús Fiori

*Director:*
Dr. Jorma Tarhio

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

9 de diciembre de 2019

# Resumen

En este documento consideraremos la versión de $k$ sustituciones de string matching aproximado (o búsqueda aproximada de cadenas) para uno solo y múltiples patrones. El problema básicamente consiste en encontrar todas las ocurrencias de uno o más patrones con a lo sumo $k$ sustituciones de caracteres en un texto.

Los algoritmos para este problema tienen varias aplicaciones como detección de virus e intrusiones, ortografía, reconocimiento de voz, reconocimiento óptico de caracteres, reconocimiento de escritura a mano, recuperación de texto bajo sinónimos o expansión de tesauro, entre otros. Con la disponibilidad de grandes cantidades de datos de ADN, la comparación de secuencias de nucleótidos en metagenómica en biología computacional se ha convertido en otra aplicación importante. Encontrar un gen en un nuevo organismo (por ejemplo, una planta de cultivo) con una secuencia similar a un gen de un organismo modelo (por ejemplo, levadura) proporciona una predicción de que el nuevo gen tiene la misma función que en el organismo modelo.

Dada la gran popularidad de las extensiones de conjuntos de instrucciones SIMD (Single Instruction Multiple Data) en las CPUs actuales, presentamos nuevos algoritmos eficientes para este problema que las aprovechan. Aplicamos instrucciones SIMD de dos maneras: contando sustituciones y en el cálculo de valores hash.

Medimos el rendimiento de cada nuevo algoritmo mediante pruebas exhaustivas en diferentes textos de la vida real comparándolo con los algoritmos más competitivos conocidos hasta la fecha. Tomamos un enfoque práctico al tratar de mejorar el tiempo promedio de cada algoritmo.

# Agradecimientos/Acknoledgements


- To Jorma, for being a great guy and welcoming me from the very beginning. Kiitos.
- A mis padres, hermanos y resto de la familia, por apoyarme desde todos los frentes.
- A la gente del grupo de programación competitiva. Muy especialmente a Margarita y Emilio, por su compañerismo, por incluirme aun cuando la situación estaba lejos de ser la ideal.

# Índice General

# Capítulo 1

# Introducción

## 1.1. Objetivos

El objetivo de esta tesis es mejorar los algoritmos para string matching de un solo patrón y múltiples patrones con a lo sumo $k$ sustituciones. Nuestro objetivo es obtener mejores resultados que los algoritmos más rápidos hasta la fecha, y así contribuir al estado del arte del tema. Investigaremos diferentes formas de utilizar instrucciones *Single Instruction Multiple Data* (SIMD) para conseguirlo, que incluirán el cálculo de valores hash (como se hace en [12]) y un método de recuento de sustituciones (como se hace en [14]). SIMD [20] es un tipo de arquitectura paralela que permite que una instrucción sea operada en múltiples elementos de datos al mismo tiempo.

Evaluaremos el rendimiento de cada nuevo algoritmo mediante pruebas exhaustivas en diferentes alfabetos de la vida real frente a los algoritmos más competitivos conocidos hasta la fecha. Tomamos un enfoque práctico al tratar de mejorar el tiempo promedio empleado por los algoritmos.

## 1.2. Resultados Principales

Presentamos dos nuevos algoritmos basados en SIMD para string matching con a lo sumo $k$ sustituciones para único y múltiples patrones: Baeza-Yates–Perleberg mejorado con SIMD (BYPS, Sección 3.1.2) y Baeza-Yates–Perleberg mejorado con SIMD para Múltiples patrones (MBYPS, Sección 3.2). Cada uno de ellos resultó ser el algoritmo más rápido para su problema correspondiente en muchos casos como se demuestra en el Capítulo 4.

También presentamos una versión modificada de Approximate Naive mejorada con SIMD [14] (ANS/ANS2, Sección A.4.2.1), que llamamos ANS2b (Sección 3.1.1), que mejora el rendimiento en patrones pequeños (no más de 16 caracteres) en comparación con ANS/ANS2. Además, sustituimos ANS2 por esta variación en BYPS y MBYPS, y también utilizamos una comparación de cadenas basada en SIMD, obteniendo algoritmos aún más rápidos como se muestra en el Capítulo 4. Los llamamos BYPSb y MBYPSb respectivamente. Además, implementamos versiones que omiten un paso de filtrado

intermedio en BYPSb y MBYPSb y asumen un hash perfecto. Se denominan BYPSc y MBYPSc, y son útiles en algunos casos.

## 1.3. Estructura de la Tesis

El resto de la tesis se divide en tres capítulos y un apéndice. En el Capítulo 2 se presentan los conceptos previos, definiciones y notación utilizados a lo largo de la tesis, así como algunas de las aplicaciones más importantes de este tipo de algoritmos. El Capítulo 3 introduce tres nuevos algoritmos basados en SIMD desarrollados en esta tesis para el problema de string matching aproximado para uno solo y múltiples patrones. El Capítulo 4 evalúa el rendimiento práctico de los algoritmos desarrollados en comparación con los ya existentes mediante experimentos prácticos. Finalmente, en el Capítulo 5 se resumen los aportes de la tesis y se exponen algunas líneas de trabajo interesantes para seguir explorando. En el Apéndice A hay una versión extendida de la tesina en idioma inglés, la cual a su vez contiene siete capítulos y dos apéndices, donde se analiza en profundidad el funcionamiento de los algoritmos testeados.

# Capítulo 2

# Conceptos Previos

## 2.1. Definición del Problema y Notación

El problema de *string matching* (o *búsqueda de cadenas*) se define de la siguiente manera: dado un patrón $P = p_0 \cdots p_{m-1}$ y un texto $T = t_0 \cdots t_{n-1}$ en un alfabeto $\Sigma$ de tamaño $\sigma$, encontrar todas las ocurrencias de $P$ en $T$.

Consideramos la variante de *k sustituciones* del problema, donde $P'$ es una ocurrencia de $P$ si $|P'| = |P|$ (usamos $|S|$ para indicar la longitud de una cadena $S$) y $P'$ tiene como máximo $k$ sustituciones de caracteres respecto a $P$. La distancia en términos de sustituciones entre dos cadenas de igual longitud también se denomina *Hamming distance* o *distancia Hamming*. Nos referiremos a *ratio de diferencia* o *relación de diferencia* (en inglés conocido como *difference ratio*) como $\alpha = k/m$, el cual es un indicador de cuán 'diferentes' pueden ser el patrón y sus ocurrencias aproximadas.

Existe una versión más general del problema que busca coincidencias de patrones con a lo sumo $k$ *diferencias* o $k$ *errores*, también conocido como con una *distancia de edición* de como máximo $k$. En este contexto, *diferencias* se definen como no sólo sustituciones, sino también inserciones y eliminaciones de caracteres. El análisis de este problema está fuera del alcance de la tesis.

Además del problema de patrón único, también consideramos la variante de múltiples patrones donde hay $r$ patrones $P^0, \cdots, P^{r-1}$ para buscar. El patrón $P^i$ tiene una longitud $m_i$ y denotamos $P^i = p_0^i \cdots p_{m_i-1}^i$.

Por simplicidad asumiremos que un byte representa un caracter. También asumiremos $m_i = m$ para todo $i$. Si hubieran diferentes longitudes, trabajaríamos con $m' = \min_{0 \le i < r} \{m_i\}$ y verificaríamos todos los 'candidatos de coincidencia' que sean un prefijo de longitud $m'$ de alguno de los patrones.

Trabajaremos con la versión *online* (o *en línea*) del problema, en la que los patrones se conocen de antemano pero el texto no. De esta manera, el único preprocesamiento que se puede hacer es sobre los patrones. Por otro lado, la variación *offline* (o *fuera de línea*) supone un texto fijo conocido y los patrones pueden variar, por lo que la fase de preprocesamiento puede centrarse en el texto.

Nos referiremos a operaciones bit a bit AND, OR, NOT, desplazamiento a la iz-

quierda y desplazamiento a la derecha con los mismos símbolos que en el lenguaje de programación C: '&', '|', '∼', '<<' y '>>', respectivamente. Los números binarios se distinguirán teniendo un subíndice 2 en su último dígito, y la notación $X_2^Y$, donde $X$ es un número binario e $Y \geq 0$ es un número entero, significa que $X$ se repite $Y$ veces. Por ejemplo, $01^3 0^2 1_2^2 = 01110011_2$ y $(01_2^3)_2^2 = 01110111_2$.

## 2.2. Aplicaciones

Los algoritmos para este problema tienen varias aplicaciones como detección de virus e intrusiones [23], ortografía [21], reconocimiento de voz [8], reconocimiento óptico de caracteres [10], reconocimiento de escritura a mano [10], recuperación de textos bajo sinónimos o expansión de tesauro [6], entre otros.

Con la disponibilidad de grandes cantidades de datos de ADN, la comparación de secuencias de nucleótidos en metagenómica en biología computacional se ha convertido en otra aplicación importante [1, 2, 13, 16]. Encontrar un gen en un nuevo organismo (por ejemplo, una planta de cultivo) con una secuencia similar a un gen de un organismo modelo (por ejemplo, levadura) proporciona una predicción de que el nuevo gen tiene la misma función que en el organismo modelo [25]. Además, en la evolución de las secuencias de proteínas, no todas las regiones mutan a la misma velocidad. Las regiones que son esenciales para la estructura y función de las proteínas están más conservadas. Por lo tanto, una similitud significativa en las secuencias de dos proteínas puede reflejar una función biológica cercana o un origen evolutivo en común. Cuando la similitud de las secuencias es estadísticamente significativa, se puede deducir con un alto nivel de confianza que las secuencias están relacionadas [17].

Además de los usos mencionados, algunos algoritmos de búsqueda aproximada de un solo patrón recurren a la búsqueda de múltiples patrones para buscar partes del patrón [5]. Dependiendo de la aplicación, $r$ puede variar de unos pocos a miles de patrones.

## 2.3. Arquitectura SIMD

*Single Instruction Multiple Data* (SIMD) [20] es un tipo de arquitectura paralela que permite que una instrucción sea operada en múltiples elementos al mismo tiempo como se muestra, por ejemplo, en la Figura 2.1. Inicialmente, SIMD se utilizaba en multimedia, especialmente en el procesamiento de imágenes o archivos de audio. Desde entonces, las instrucciones SIMD han encontrado aplicaciones en otras áreas como la criptografía y, más recientemente, también se han aplicado a string matching [7, 11, 14, 22, 24, 28]. Más detalles sobre este tipo de arquitectura y los conjuntos de extensiones SIMD desarrollados por Intel se encuentran en la Sección A.2.3. Sólo nos referimos a la arquitectura SIMD de Intel por razones de popularidad.

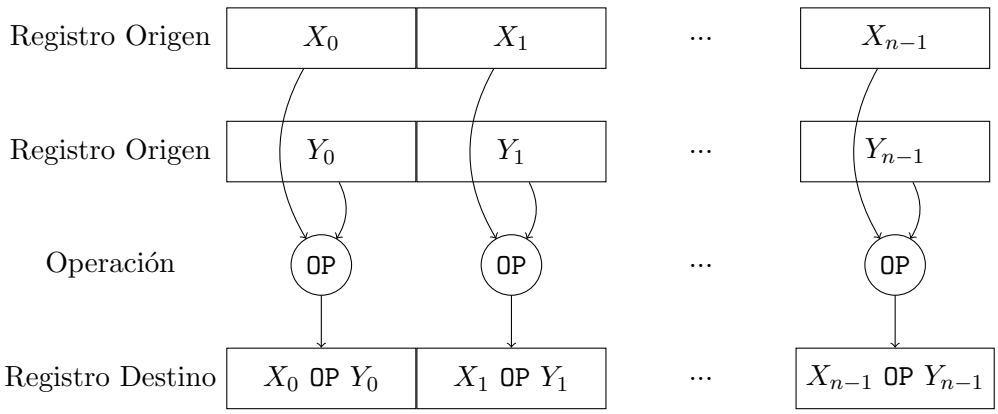



Figura 2.1: Ejemplo simple de operación SIMD.

# Capítulo 3

# Nuevos Algoritmos basados en SIMD

En este capítulo describimos tres nuevos algoritmos desarrollados en esta tesis: dos están diseñados para string matching aproximado de un solo patrón y el tercero para la variante de patrones múltiples. Dos de ellos utilizan SIMD incluyendo otros dos algoritmos basados en SIMD: MEPSM (Sección A.4.3) con algunas modificaciones y ANS2/ANS2b (Secciones A.4.2.1 y 3.1.1). El algoritmo que funciona para múltiples patrones es una extensión sencilla pero fructífera de uno de los nuevos algoritmos presentados para el caso de un único patrón.

## 3.1. String Matching Aproximado para Patrón Único

### 3.1.1. Approximate Naive mejorado SIMD (ANS2b)

Presentamos una variación del algoritmo Approximate Naive mejorado con SIMD (ANS2, Sección A.4.2.1). Como se indica en la sección mencionada, el rendimiento de ANS2 disminuye fuertemente cuando $k$ aumenta en los casos en los que $m \leq 16$ por fallas en el predictor de saltos condicionales. Una forma de evitar esta dependencia sobre $k$ sería cambiar la última línea del algoritmo ANS2 (ver Algoritmo 10)

$$\text{if } D[t] \text{ then } occ \leftarrow occ + 1$$

por

$$occ \leftarrow occ + D[t]$$

donde $D$ es el arreglo de booleanos usado en ANS2, almacenando valores *true* como números uno y valores $false$ como ceros. Esta modificación le obligaría a realizar siempre una adición sin una sentencia condicional. Ahorraría tiempo en comparación con el ANS2 original cuando hay muchas ocurrencias, lo que es más probable que ocurra cuando la relación de diferencia es alta y el tamaño del alfabeto es pequeño. Llamaremos a esta variación ANS2b. El pseudocódigo para su fase de búsqueda cuando $m \leq 16$ está escrito en el Algoritmo 1. ANS2b es igual que ANS2 para $m > 16$, por lo que se omite su

pseudocódigo para este caso. ANS2b resultó más rápido que ANS2 en todos nuestros experimentos en la Sección 4.1.

**Algoritmo 1**: ANS2b Búsqueda ($m \leq 16$)
$x \leftarrow simd\text{-}load(p_0 \cdots p_{m-1})$
for $i \leftarrow 0$ to $n - m$ do
    $y \leftarrow simd\text{-}load(t_i \cdots t_{i+m-1})$
    $t \leftarrow simd\text{-}cmpeq(x, y)$
    $occ \leftarrow occ + D[t]$
return $occ$

ANS2b tiene la misma complejidad temporal promedio de búsqueda que ANS y ANS2: $O(\lceil k/b \rceil n)$, donde $b$ es el número de caracteres que entran en un registro SIMD usado por las instrucciones empleadas en el algoritmo. Para el análisis de complejidad de ANS y ANS2 ver Sección A.4.2.1.

### 3.1.2. Baeza-Yates–Perleberg mejorado con SIMD

Presentamos un algoritmo basado en el algoritmo Baeza-Yates–Perleberg (BYP, Sección A.3.1.4), mejorado con instrucciones SIMD. El BYP original busca ocurrencias *exactas* de $k + 1$ subpatrones del patrón en el texto como se describe en la Sección A.3.1.4: $k + 1 - (m \mod (k + 1))$ de longitud $l$ y $m \mod (k + 1)$ de longitud $l + 1$, donde $l = \lfloor m/(k+1) \rfloor$. Para lograr esto, empleamos una versión modificada del algoritmo MEPSM para string matching exacto de múltiples cadenas (descripto en la Sección A.4.3). MEPSM reporta las ocurrencias de los subpatrones, que luego son verificadas por ANS2 (ver Sección A.4.2.1). Llamemos al algoritmo total BYPS.

Modificamos MEPSM ajustando los $q$-grams a lo más grande posible, es decir $q = \min(l, 8)$. El valor máximo para $q$ es 8 porque necesitamos calcular el valor hash de $q$-grams con $simd\text{-}crc(x)$. Para ello, utiliza una instrucción SIMD como la descrita en la Sección A.4.1.2, lo que significa que puede tomar un valor $x$ de hasta 64 bits como entrada, es decir, 8 bytes. Esto causa menos colisiones de valores hash, pero por otro lado, un $q$ más grande reduce los saltos entre alineaciones ($shift = l - q + 1$). Sin embargo, este intercambio demostró ser realmente satisfactorio en la práctica, especialmente en el caso de pequeños subpatrones (para los resultados de los experimentos ver Sección 4.1). Esta optimización se emplea en combinaciones de $m$ y $k$ que hacen que la longitud de los subpatrones buscados por MEPSM sea inferior a 8 caracteres (es decir, $l < 8$). Por otra parte, el valor mínimo admitido para $q$ (y por lo tanto para $l$) es 4 porque el número de colisiones durante el hashing aumentaría rápidamente de otra manera, disminuyendo críticamente su rendimiento.

También presentamos una variación de BYPS donde utilizamos el algoritmo ANS2b para buscar las ocurrencias aproximadas del patrón en las proximidades de las coincidencias exactas de los subpatrones. Dada la superioridad de ANS2b sobre ANS2, sería lógico obtener también una mejora en el algoritmo BYPS. En este algoritmo también reemplazamos el chequeo exacto trivial de subpatrones realizado por MEPSM después

de encontrar un valor hash coincidente de un $q$-gram en el texto con una función de comparación basada en SIMD *simd-memcmp*, la cual emplea la función *simd-cmpeq* definida en la Sección A.4.1.1. La función *simd-memcmp* se muestra en el Algoritmo 2 y funciona para cadenas de igual tamaño $|s|$ de hasta 32 caracteres[1] utilizando instrucciones de AVX2 en un tiempo de $O(\lceil |s|/b \rceil)$, en donde $b$ es la cantidad de caracteres que caben en un registro SIMD. A esta versión la llamamos BYPSb.

Las fases de preprocesamiento y búsqueda de BYPSb para los casos en los que $l < 8$ se muestran en pseudocódigo en los Algoritmos 3 y 4 respectivamente. En los casos en que $l \geq 8$ se utiliza el cálculo original de MEPSM de $q$. Los símbolos $\langle\rangle$ se utilizan para indicar el principio y el final de una lista de elementos, y $+\!\!+$ indica la concatenación de listas.

**Algoritmo 2**: *simd-memcmp*$(s_1, s_2)$ donde $|s_1| = |s_2|$ y $|s_1| \leq 32$
$eqmask \leftarrow (1 << |s_1|) - 1$
$x \leftarrow$ *simd-load*$(s_1)$
$y \leftarrow$ *simd-load*$(s_2)$
$mask \leftarrow$ *simd-cmpeq*$(x, y)$
return $(eqmask \;\&\; mask) = eqmask$

**Algoritmo 3**: BYPSb Preprocesamiento ($l < 8$)
$ANS2bPreprocess(m, k)$
$l \leftarrow \lfloor m/(k+1) \rfloor$
$rem \leftarrow m \bmod l$
$q \leftarrow \min(l, 8)$
*shift* $\leftarrow l - q + 1$
for $f \leftarrow 0$ to $2^{16} - 1$ do
  $H[f] \leftarrow \langle\rangle$
for *subpat* $\leftarrow 0$ to $m - l - rem * (l+1)$ step $l$ do
  for $j \leftarrow 0$ to *shift* $- 1$ do
    $f \leftarrow$ *simd-crc*$(p_{subpat+j} \ldots p_{subpat+j+q-1})$
    $H[f] \leftarrow \langle (subpat, j) \rangle +\!\!+ H[f]$
for *subpat* $\leftarrow m - rem * (l+1)$ to $m - (l+1)$ step $l + 1$ do
  for $j \leftarrow 0$ to *shift* $- 1$ do
    $f \leftarrow$ *simd-crc*$(p_{subpat+j} \ldots p_{subpat+j+q-1})$
    $H[f] \leftarrow \langle (subpat, j) \rangle +\!\!+ H[f]$

Hemos implementado otra versión de BYPSb que omite completamente el paso de verificar exactamente la ocurrencia de un subpatrón y asume una aparición del mismo cada vez que hay una coincidencia de valores hash en el texto. Dado que $q = \text{mín}(l, 8)$, este paso no sería necesario en los casos en que $q = l$ si la función hash funcionara

[1] Su extensión a tamaños generales es sencilla.

**Algoritmo 4**: BYPSb Búsqueda ($l < 8$)
$lastpos \leftarrow 0$
$occ \leftarrow 0$
for $i \leftarrow 0$ to $n - q$ step *shift* do
  $f \leftarrow simd\text{-}crc(t_i \ ... \ t_{i+q-1})$
  forall $(subpat, j)$ in $H[f]$ do
    if $simd\text{-}memcmp(t_{i-j} \ ... \ t_{i-j+l-1},\ p_{subpat} \ ... \ p_{subpat+l-1})$ then
      $begin \leftarrow \max\{i - j - (m - l),\ lastpos - m + 1,\ 0\}$
      $end \leftarrow \min\{i - j - subpat + m - 1,\ n - 1\}$
      if $end - begin + 1 \geq m$ then
        $lastpos \leftarrow end$
        $occ \leftarrow occ + ANS2bSearch(t_{begin} \ ... \ t_{end}, p, k)$
return *occ*

perfectamente. Tener en cuenta que si se realiza una suposición incorrecta de una coincidencia de un subpatrón, la eficacia del algoritmo no se verá afectada porque ANS2b analizará la ventana posteriormente. Llamaremos BYPSc a esta versión.

Para un análisis de complejidad temporal de estos algoritmos ver Sección A.5.1.2.

## 3.2. String Matching Aproximado para Múltiples Patrones

### Baeza-Yates–Perleberg mejorado con SIMD para Múltiples Patrones

Hemos extendido el algoritmo BYPS para trabajar con múltiples patrones. El nuevo algoritmo MBYPS funciona de la siguiente manera:

1. En el preprocesamiento, dividimos cada patrón en subpatrones de longitud $l$ o $l + 1$ de la misma manera que en BYPS (descrito en la Sección 3.1.2), donde $l = \lfloor m/(k + 1) \rfloor$. Luego calculamos el valor hash CRC de *shift* $q$-grams de cada subpatrón, donde $q \leq l$ es un parámetro modificado del algoritmo MEPSM, y $shift = l - q + 1$. El valor hash se utiliza para acceder a una tabla que almacena información sobre a cuáles subpatrones de qué patrones pertenece.

2. En la búsqueda, calculamos el valor hash de un $q$-gram en el texto, con el que obtenemos la información correspondiente de la tabla. Realizamos un desplazamiento de *shift* caracteres en el texto después de analizar cada $q$-gram, que es el número máximo de caracteres que podemos saltear.

3. Por cada subpatrón asociado con el valor hash, comprobamos de manera trivial si aparece *exactamente* en este punto. Si lo hace, se reporta una posible ocurrencia aproximada del patrón correspondiente.

4. Cada vez que se encuentra un candidato de ocurrencia de un patrón, usamos un algoritmo de string matching *aproximado* de patrón único para verificarlo.

Para la fase de string matching exacto de múltiples patrones usamos nuestra versión modificada de MEPSM como se describe en la Sección 3.1.2, excepto que ahora extendemos nuestro ajuste del cálculo de $q$ a todas las longitudes de subpatrones. Para la fase de string matching aproximado de una sola cadena usamos ANS2 para $m \leq 32$. Para patrones más largos, se debe utilizar otro algoritmo.

Para evitar volver a verificar una ocurrencia, hacemos un seguimiento de la posición hasta la cual el texto ya ha sido analizado para cada patrón. De esta forma, no es necesario comprobar si se han producido ocurrencias que comiencen en posiciones a la izquierda de la última analizada. Esto sería trivial si MEPSM garantizara orden al reportar las ocurrencias exactas de los subpatrones, pero no lo hace. Esto se ha resuelto ejecutando el algoritmo de string matching aproximado de patrón único en una ventana más grande. Si hay una ocurrencia en la posición $x$ en el texto de un subpatrón que comienza en la posición $sp$ del patrón $p$, comprobamos si hay una ocurrencia aproximada del patrón $p$ desde la posición $x - (m - l)$ a $x + m - sp$. Por lo tanto, una vez que se ha encontrado una ocurrencia de un patrón, una ocurrencia más nueva nunca la precederá posicionalmente.

Siguiendo el mismo razonamiento que en BYPSb, también consideramos una variación de MBYPS donde se utiliza ANS2b en lugar de ANS2 para buscar ocurrencias aproximadas de patrones y *simd-memcmp* para comparar exactamente las cadenas, que llamamos MBYPSb. Los algoritmos 5 y 6 muestran pseudocódigo para las fases de preprocesamiento de MBYPSb y de búsqueda, respectivamente. Los cambios con respecto a BYPSb se han resaltado en rojo. Tener en cuenta que estamos comparando MBYPSb con la versión BYPSb para $l < 8$ porque MBYPSb utiliza esta misma configuración para calcular $q$ para todos los valores de $l$.

Además, hemos implementado otra versión de MBYPSb análoga a BYPSc que omite el paso de comparar exactamente un subpatrón con una ventana de texto de $l$ caracteres después de una coincidencia de un valor hash. Lo llamamos MBYPSc.

Para un análisis de complejidad temporal de estos algoritmos ver Sección A.5.2.

**Algoritmo 5**: MBYPSb Preprocesamiento

$ANS2bPreprocess(m, k)$
$l \leftarrow \lfloor m/(k+1) \rfloor$
$rem \leftarrow m \bmod l$
$q \leftarrow \min(l, 8)$
$shift \leftarrow l - q + 1$
for $f \leftarrow 0$ to $2^{16} - 1$ do
  $H[f] \leftarrow \langle\rangle$
for $pat \leftarrow 0$ to $r - 1$ do
  for $subpat \leftarrow 0$ to $m - l - rem * (l + 1)$ step $l$ do
    for $j \leftarrow 0$ to $shift - 1$ do
      $f \leftarrow simd\text{-}crc(p^{pat}_{subpat+j} \ldots p^{pat}_{subpat+j+q-1})$
      $H[f] \leftarrow \langle (pat, subpat, j) \rangle +\!\!+ H[f]$
  for $subpat \leftarrow m - rem * (l + 1)$ to $m - (l + 1)$ step $l + 1$ do
    for $j \leftarrow 0$ to $shift$ do
      $f \leftarrow simd\text{-}crc(p^{pat}_{subpat+j} \ldots p^{pat}_{subpat+j+q-1})$
      $H[f] \leftarrow \langle (pat, subpat, j) \rangle +\!\!+ H[f]$

**Algoritmo 6**: MBYPSb Búsqueda

$occ \leftarrow 0$
$lastpos_{pat} \leftarrow 0 \ \forall \ pat$
for $i \leftarrow 0$ to $n - q$ step $shift$ do
  $f \leftarrow simd\text{-}crc(t_i \ldots t_{i+q-1})$
  forall $(pat, subpat, j)$ in $H[f]$ do
    if $simd\text{-}memcmp(t_{i-j} \ldots t_{i-j+l-1},\ p^{pat}_{subpat} \ldots p^{pat}_{subpat+l-1})$ then
      $begin \leftarrow \max\{i - j - (m - l),\ lastpos_{pat} - m + 1,\ 0\}$
      $end \leftarrow \min\{i - j + m - subpat,\ n - 1\}$
      if $end - begin \geq m$ then
        $lastpos_{pat} \leftarrow end$
        $occ \leftarrow occ + ANS2bSearch(t_{begin} \ldots t_{end}, p^{pat}, k)$
return $occ$

# Capítulo 4

# Experimentos

Los testeos han sido ejecutados en un CPU Intel Xeon E5-2603 v4 1.70 GHz con 8 GiB DDR3 de memoria RAM, usando Ubuntu 17.10 64-bit. Este procesador tiene una microarquitectura Broadwell y posee extensiones SIMD SSE4.2 y AVX2, pero no AVX-512.

Los programas fueron escritos en el lenguaje de programación C y compilados con gcc 7.2.0 usando el nivel de optimización -O3. El código para los algoritmos que no han sido desarrollados en este documento (es decir, aquéllos que no sean ANS2b, HBYN, BYPS/BYPSb/BYPSc y MBYPS/MBYPSb/MBYPSc) fue proporcionado por sus autores. Todos los algoritmos fueron implementados y testeados en el framework de Hume y Sunday [19]. Éste permite medir los tiempos de preprocesamiento y búsqueda por separado, realiza todas las operaciones de lectura antes de iniciar la fase de búsqueda de un algoritmo y sólo imprime los resultados después de medir los tiempos de un algoritmo. De esta manera, no hay tiempo de CPU invertido en I/O no sincronizado, lo que podría interferir con las mediciones. Todas las ocurrencias aproximadas de los patrones sólo se cuentan y no se imprimen.

Usamos dos textos de diferentes alfabetos para las pruebas: ADN (el genoma de Escherichia Coli, 4.6 MiB) e inglés (la versión King James de la Biblia, 4.0 MiB). Los textos fueron tomados del corpus Smart[1]. Cada cadena de los conjuntos de patrones fue generada seleccionando aleatoriamente una subcadena de longitud $m$ del texto, y luego sustituyendo un número aleatorio de sus caracteres (entre 0 y 8 incluidos) por caracteres aleatorios que aparecen en el texto con el fin de simular sustituciones de caracteres.

Se reportan los tiempos de ejecución de la fase de búsqueda de cada algoritmo, obtenidos a partir del promedio de 100 ejecuciones. Omitimos el tiempo de preprocesamiento porque nos enfocamos en la versión *online* del problema (ver Sección 2.1), por lo que se vuelve insignificante comparado con el tiempo de búsqueda en un texto arbitrariamente largo.

También realizamos pruebas en una secuencia de proteínas (de la secuencia del genoma humano, 3.1 MiB) tomada del corpus Smart. Los archivos de proteínas suelen estar codificados en un alfabeto de tamaño 20, donde cada caracter corresponde a un

[1] http://www.dmi.unict.it/~faro/smart/corpus.php

aminoácido diferente. Se obtuvieron resultados similares a los reportados en este capítulo para el alfabeto inglés para la búsqueda de un solo patrón y para el alfabeto de ADN en el caso de múltiples patrones. Omitimos estos resultados para evitar ser redundantes.

Realizamos los mismos conjuntos de pruebas en una máquina más moderna en el Apéndice A.A, obteniendo resultados aún más positivos para los algoritmos basados en SIMD que los presentados en este capítulo.

**Palabras de advertencia.** Nuestros resultados experimentales se basan en los procesadores que utilizamos en nuestras pruebas. Es posible que en un futuro los procesadores produzcan resultados diferentes si cambia la velocidad relativa de las instrucciones.

## 4.1. String Matching Aproximado para Patrón Único

Los siguientes algoritmos han sido comparados para este problema:

- SA: Shift-Add [3] (Sección A.3.1.2).
- TuSA: Tuned Shift-Add [9] (Sección A.3.1.2).
- TwSA: Two-way Shift-Add [9] (Sección A.3.1.2).
- EF: Enhanced FAAST [27] (Sección A.3.1.3).
- EFS: Enhanced FAAST with SIMD [14] (Sección A.4.2.2).
- ANS, ANS2 y ANS2b: Approximate Naive mejorado con SIMD [14] (Secciones A.4.2.1 y 3.1.1).
- BYP: algoritmo Baeza-Yates–Perleberg original [5] (Sección A.3.1.4).
- BYPS, BYPSb y BYPSc: Baeza-Yates–Perleberg mejorado con SIMD (Sección 3.1.2).

De ellos, ANS2b, BYPS, BYPSb y BYPSc fueron desarrollados en este trabajo. Usamos conjuntos de 100 patrones diferentes para testear. Para cada algoritmo mostramos la suma de los tiempos necesarios para buscar cada patrón del conjunto.

Según las pruebas de Hirvola [18], TwSA era el mejor para los textos en inglés. Según las pruebas de Salmela et al. [27], EF era el mejor para datos de ADN. Nuestras pruebas confirman las tendencias reportadas en [14], con un dominio de los algoritmos basados en SIMD.

Los resultados se muestran en la Tabla 4.1 con los mejores tiempos resaltados. Hay gráficas de tiempos de búsqueda de las variantes más rápidas de los algoritmos testeados en función de $k$ para $m \in \{16, 32\}$ y ambos alfabetos en las Figuras 4.1, 4.2, 4.3 y 4.4. Podemos observar que ANS2b y BYPSb/BYPSc son casi siempre los más rápidos para todas las combinaciones de parámetros tanto en ADN como en inglés. Sólo son superados

| | $m = 8$ | | | $m = 16$ | | | $m = 24$ | | | $m = 32$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | $\Sigma$ |
| SA | 1.68 | 1.69 | 1.69 | 1.68 | 1.69 | 1.68 | 1.68 | -[a] | -[a] | 1.68 | -[a] | -[a] | ADN |
| TuSA | 1.31 | 1.31 | 1.31 | 1.31 | 1.31 | 1.31 | 1.31 | -[a] | -[a] | 1.31 | -[a] | -[a] | |
| TwSA | 1.62 | 2.13 | 2.40 | 0.81 | 1.07 | 1.29 | 0.55 | -[a] | -[a] | 0.41 | -[a] | -[a] | |
| ANS | 1.25 | 1.25 | 1.25 | 1.25 | 1.25 | 1.25 | 1.35 | 1.35 | 1.34 | 1.34 | 1.34 | 1.34 | |
| ANS2 | 0.86 | 0.91 | 1.16 | 0.88 | 0.88 | 0.88 | -[b] | -[b] | -[b] | -[b] | -[b] | -[b] | |
| ANS2b | **0.75** | **0.75** | **0.75** | 0.78 | **0.78** | **0.78** | 1.05 | 1.05 | **1.07** | 1.05 | 1.05 | 1.07 | |
| EF | 1.46 | 2.28 | 3.90 | 0.70 | 1.04 | 1.75 | 0.49 | 0.77 | 1.37 | 0.39 | 0.64 | 1.20 | |
| EFS | 1.41 | 2.14 | 3.92 | 0.67 | 0.97 | 1.59 | 0.48 | 0.71 | 1.24 | 0.38 | 0.59 | 1.10 | |
| BYP | 4.18 | 10.13 | 14.82 | 3.56 | 5.32 | 8.23 | 3.62 | 5.16 | 6.36 | 3.67 | 4.99 | 5.83 | |
| BYPS | 1.60 | -[a] | -[a] | 0.35 | 1.56 | 1.93 | 0.25 | 0.42 | 1.60 | 0.19 | 0.35 | 0.48 | |
| BYPSb | 1.43 | -[a] | -[a] | **0.30** | 1.36 | 1.84 | **0.20** | **0.35** | 1.42 | 0.18 | **0.28** | **0.40** | |
| BYPSc | 1.16 | -[a] | -[a] | 0.37 | 1.10 | 1.42 | 0.42 | 0.63 | 1.18 | **0.15** | 0.65 | 0.84 | |
| SA | 1.47 | 1.47 | 1.47 | 1.47 | 1.47 | 1.47 | 1.47 | -[a] | -[a] | 1.47 | -[a] | -[a] | Inglés |
| TuSA | 1.14 | 1.14 | 1.14 | 1.14 | 1.14 | 1.14 | 1.14 | -[a] | -[a] | 1.14 | -[a] | -[a] | |
| TwSA | 0.83 | 1.17 | 1.53 | 0.48 | **0.62** | 0.79 | 0.33 | -[a] | -[a] | 0.26 | -[a] | -[a] | |
| ANS | 1.09 | 1.09 | 1.09 | 1.09 | 1.09 | 1.09 | 1.17 | 1.17 | 1.17 | 1.17 | 1.17 | 1.17 | |
| ANS2 | 0.75 | 0.75 | 0.76 | 0.75 | 0.75 | 0.75 | -[b] | -[b] | -[b] | -[b] | -[b] | -[b] | |
| ANS2b | **0.65** | **0.65** | **0.65** | 0.66 | 0.66 | **0.66** | 0.91 | 0.91 | **0.91** | 0.91 | 0.91 | 0.91 | |
| BYP | 1.19 | 2.29 | 3.24 | 0.77 | 1.33 | 1.87 | 0.54 | 0.92 | 1.31 | 0.49 | 0.80 | 1.08 | |
| BYPS | 1.37 | -[a] | -[a] | 0.27 | 1.43 | 1.42 | 0.16 | 0.28 | 1.44 | 0.17 | 0.20 | 0.28 | |
| BYPSb | 1.19 | -[a] | -[a] | 0.24 | 1.25 | 1.23 | **0.14** | 0.25 | 1.27 | 0.15 | **0.18** | 0.25 | |
| BYPSc | 1.04 | -[a] | -[a] | **0.20** | 1.10 | 1.07 | 0.15 | **0.23** | 1.12 | **0.13** | **0.18** | **0.24** | |

[a] El algoritmo no fue diseñado para funcionar en este caso.
[b] El algoritmo es el mismo que ANS2b.

Tabla 4.1: Tiempos de búsqueda en segundos de algoritmos para string matching aproximado de un solo patrón con hasta $k$ sustituciones ejecutados 100 veces con diferentes patrones.

por TwSA en el alfabeto inglés cuando $m = 16$ y $k = 2$. BYPSb y BYPSc son los mejores para casos con bajo ratio de diferencia. BYPSb funciona mejor en ADN mientras que BYPSc lo hace en inglés (ver complejidades teóricas en Tabla A.B.1). Por otro lado, ANS2b funciona mejor que BYPSb/BYPSc cuando el ratio de diferencia es más bien alto.

ANS, ANS2 y ANS2b funcionan para todos los valores posibles de $k$. ANS lo hace a una velocidad casi constante, independientemente del valor de $k$. La velocidad de ANS2 disminuye a medida que crece $k$, lo que es más notorio cuando $m = 8$ en ADN debido al alto número de ocurrencias. En cuanto a ANS2b, su desempeño es independiente de $k$ para $m \leq 16$ como se explica en la Sección 3.1.1. Éste mostró una clara mejoría en comparación con ANS2, superándolo en todas las pruebas, incluso en casos de pocas ocurrencias (bajos ratios de diferencia). Se han omitido los tiempos para ANS2 cuando

$m > 16$ ya que funciona de la misma manera que ANS2b y habría sido redundante (ver Sección 3.1.1 para más detalles).

También realizamos pruebas de ANS2 para todos los valores posibles de $k$ para el alfabeto de ADN y $m = 16$, obteniendo los tiempos graficados en la Figura 4.5. Usando la herramienta de análisis de rendimiento `perf`[2], recogimos información sobre esas ejecuciones. En particular, el porcentaje de saltos condicionales que se predijeron erróneamente y el número de instrucciones que empleó cada ejecución. Cada característica se reporta en las Figuras 4.6 y 4.7 respectivamente como una función de $k$. Es fácil ver que el pico en el tiempo de ejecución de ANS2 se basa en un incremento en el número de predicciones erróneas de saltos condicionales. El número de instrucciones ejecutadas por el programa también aumenta a medida que crece $k$ debido a que hay más ocurrencias aproximadas de patrones, por lo que necesita realizar más incrementos al contador de ocurrencias. Sin embargo, no es tan significativo como la variación en la cantidad de predicciones equívocas. Se han realizado las mismas pruebas en ANS2b, que mostraron un comportamiento constante para todos los $k$ como se predice en la Sección 3.1.1. Éste mantiene el mismo tiempo de ejecución, número de instrucciones ejecutadas y porcentaje de predicciones erróneas de saltos condicionales independientemente del valor de $k$. Esta característica también es evidente en sus tiempos en la Tabla 4.1.

SA, TuSA y TwSA están limitados a valores pequeños de $k$ para patrones largos. Por ejemplo, sólo trabajan para $k = 1$ en el caso de $m \in \{24, 32\}$. Además, la velocidad de TwSA se degrada a medida que crece $k$, como se muestra en su complejidad teórica en la Tabla A.B.1. EF, EFS, BYP, BYPS, BYPSb y BYPSc muestran un comportamiento similar, con $k$ afectando su velocidad. A pesar de esto, el crecimiento de $k$ puede ser tolerado dado que $m$ es lo suficientemente grande, es decir, cuando tenemos pequeños ratios de diferencia (ver Tabla A.B.1 para detalles sobre la complejidad de BYP). Se han omitido algunos tiempos de BYPS y sus variantes porque no funcionan para subpatrones de longitud $l = \lfloor m/(k+1) \rfloor < 4$ (ver Sección 3.1.2).

BYPS también ha sido testado para patrones más largos. De acuerdo con nuestros experimentos y siguiendo la misma línea que señalan Baeza-Yates y Perleberg en [5], BYP, BYPS y sus variantes obtienen sus mejores resultados con bajos ratios de diferencia. En cuanto a BYPSb, encontramos que se comporta siempre más rápido que BYPS. Por otro lado, BYPSc a veces funciona más rápido que BYPSb (particularmente en el alfabeto inglés) mientras que otras veces es incluso más lento que BYPS. También es importante destacar la mejora del rendimiento de BYPS y sus variantes con respecto a BYP original, especialmente en casos de bajos ratios de diferencia.

[2]Performance analysis tool for Linux. `https://perf.wiki.kernel.org/index.php/Main_Page`

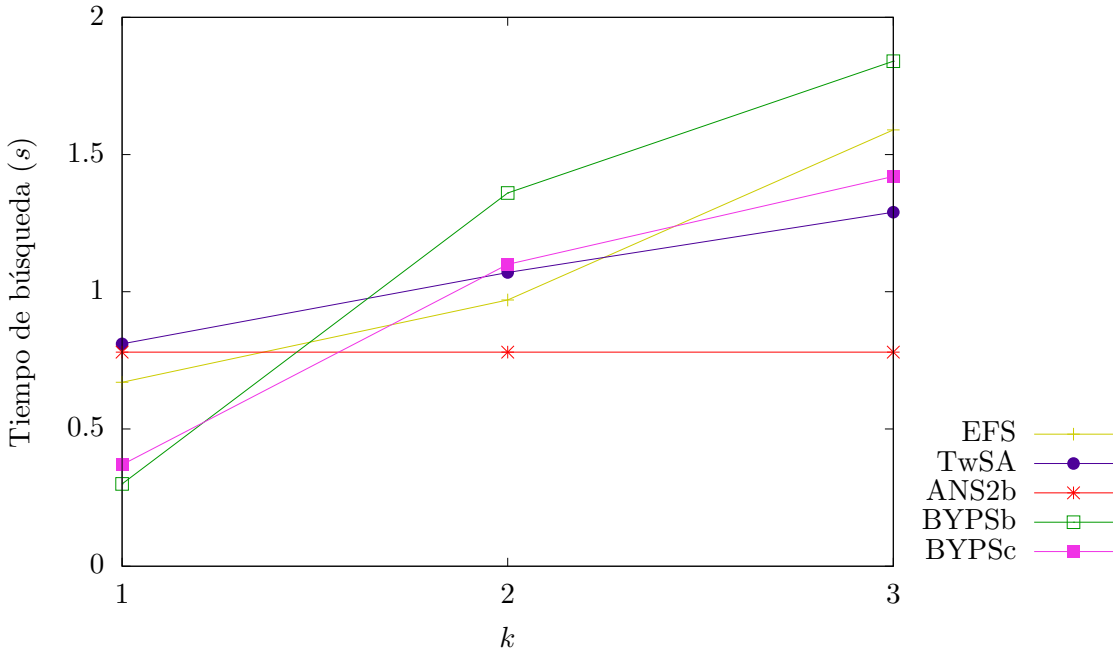



Figura 4.1: Tiempos de búsqueda de los algoritmos más rápidos testeados en función de $k$ para $m = 16$ en alfabeto de ADN.

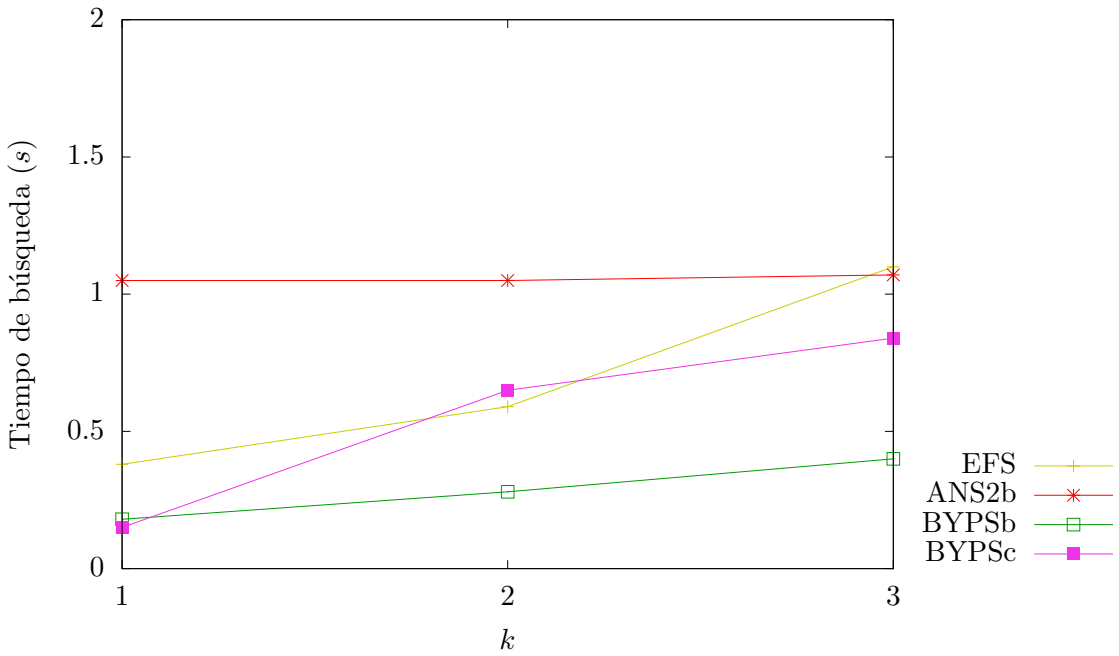



Figura 4.2: Tiempos de búsqueda de los algoritmos más rápidos testeados en función de $k$ para $m = 32$ en alfabeto de ADN.

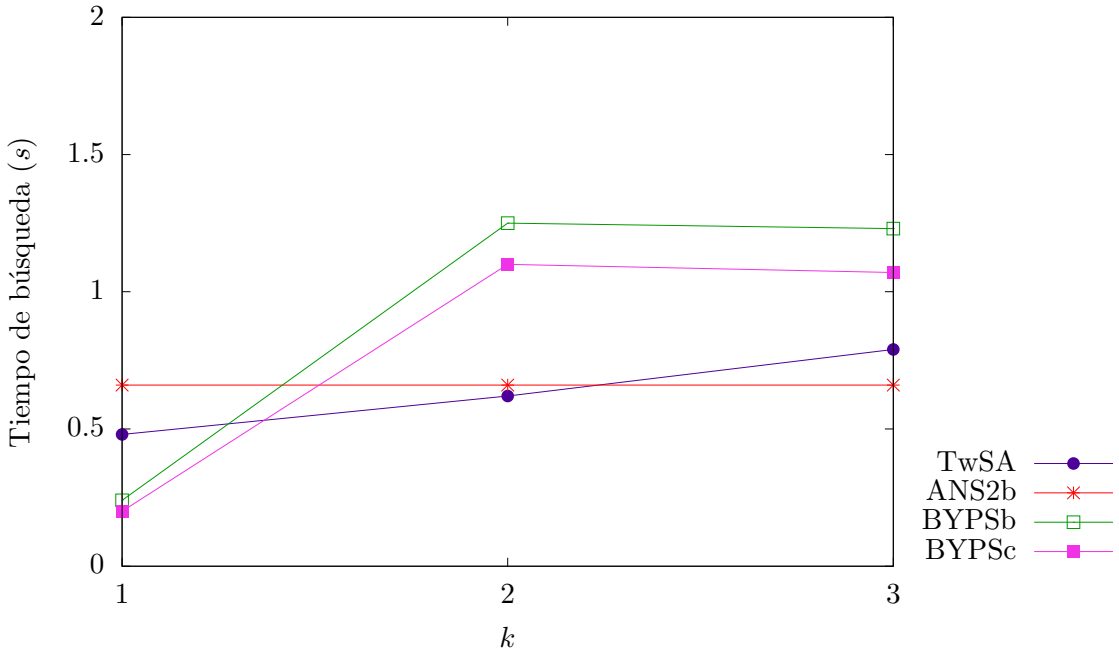



Figura 4.3: Tiempos de búsqueda de los algoritmos más rápidos testeados en función de $k$ para $m = 16$ en alfabeto inglés.

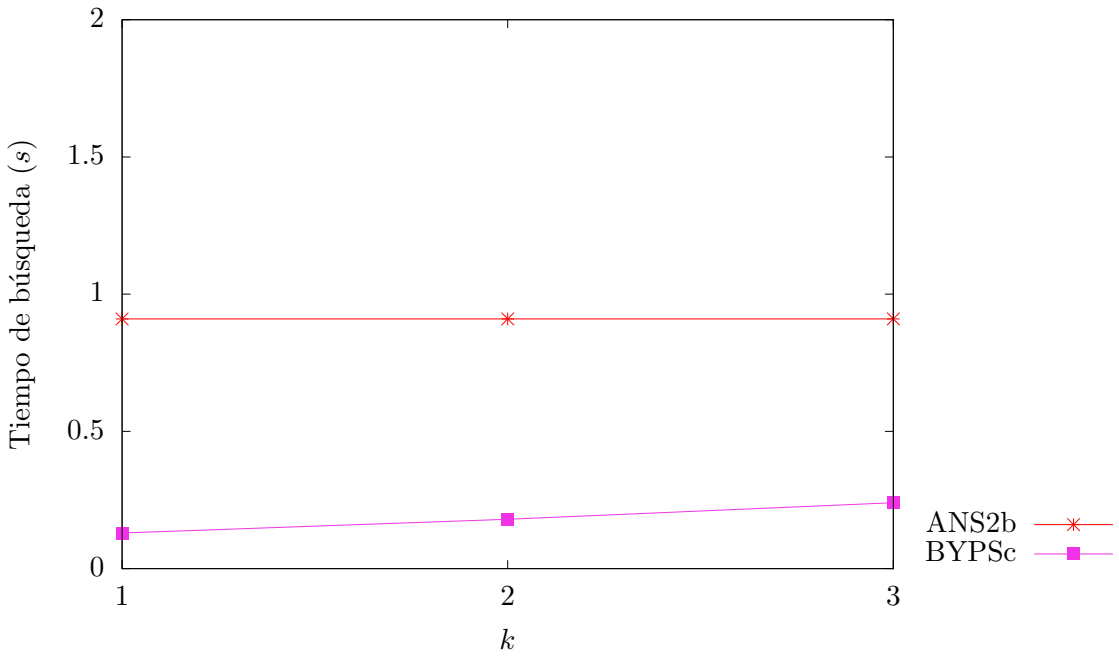



Figura 4.4: Tiempos de búsqueda de los algoritmos más rápidos testeados en función de $k$ para $m = 32$ en alfabeto inglés.

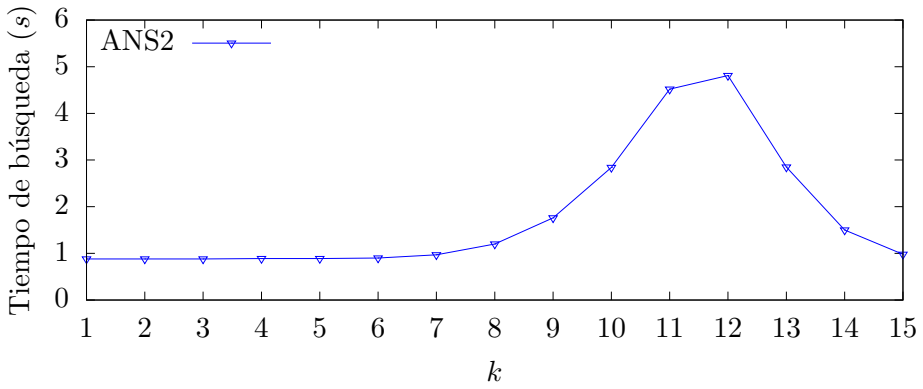



Figura 4.5: Tiempos de ejecución de ANS2 en función de $k$ para $m = 16$ en alfabeto de ADN.

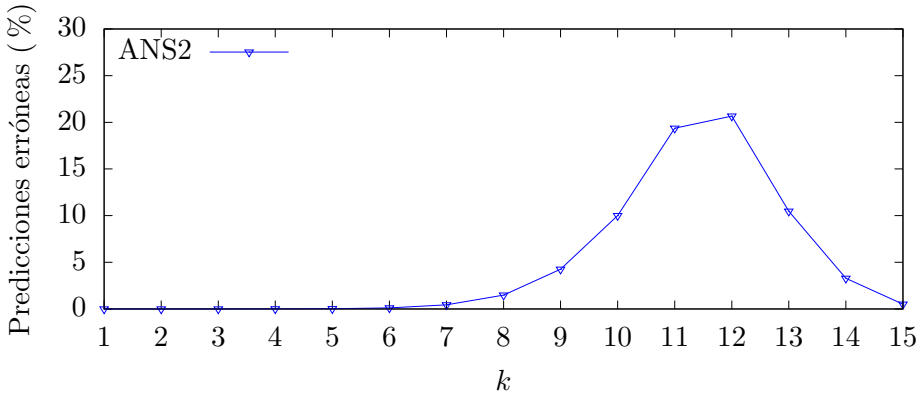



Figura 4.6: Porcentaje de errores de predicción de saltos condicionales del total de predicciones ejecutadas en ANS2 en función de $k$ para $m = 16$ en alfabeto de ADN.

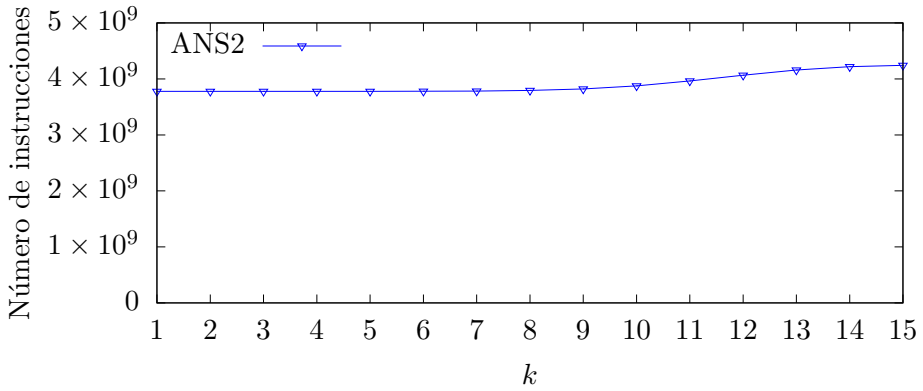



Figura 4.7: Número de instrucciones ejecutadas en ANS2 en función de $k$ para $m = 16$ en alfabeto de ADN.

## 4.2. String Matching Aproximado para Múltiples Patrones

Se usaron conjuntos de 10, 100 y 1000 patrones y $k$ pequeño para testear algoritmos para la variante del problema de buscar múltiples patrones. Los siguientes algoritmos han sido testeados:

- MM: algoritmo Muth–Manber [26] (Sección A.3.2.2).
- FN: algoritmo de Fredriksson y Navarro [15] (Sección A.3.2.4).
- BYN: algoritmo Baeza-Yates–Navarro original que detecta ocurrencias bajo *distancia de edición* [4] (Sección A.3.2.3).
- HBYN: algoritmo Baeza-Yates–Navarro adaptado a distancia Hamming (Sección A.3.2.3).
- MBYPS, MBYPSb y MBYPSc: algoritmo Baeza-Yates–Perleberg mejorado con SIMD para Múltiples patrones (Sección 3.2).

De ellos, HBYN, MBYPS, MBYPSb y MBYPSc fueron desarrollados en este trabajo. En primer lugar, vale la pena mencionar que HBYN nunca es significativamente más lento que el algoritmo BYN original. Esto significa que HBYN puede compararse con otros algoritmos, ya que aborda el mismo problema que ellos (a diferencia de BYN), y no se ha degradado en su rendimiento.

En cuanto al algoritmo Fredriksson–Navarro, se han realizado pruebas exhaustivas para elegir los mejores parámetros para cada caso. Para ADN obtuvimos la misma configuración mencionada en [15] como la mejor opción.

Los resultados se muestran en la Tabla 4.2 con los mejores tiempos resaltados. Hay gráficas de tiempos de búsqueda de los algoritmos testeados como una función de $k$ para $m \in \{16, 32\}$, $r = 100$ y ambos alfabetos en Figuras 4.8, 4.9, 4.10 y 4.11. También hay gráficas de tiempos de búsqueda en función de $r$ para $m = 16$ y $k = 3$, y $m = 32$ y $k = 1$, que representan ratios de diferencia alto y bajo respectivamente, también para ambos alfabetos, en Figuras 4.12, 4.13, 4.14 y 4.15. Para los gráficos de tiempos de búsqueda en relación a $r$ se han utilizado escalas logarítmicas en ambos ejes.

Se han omitido algunas mediciones de los siguientes algoritmos:

- MM: este algoritmo está diseñado para trabajar sólo para $k = 1$.
- HBYN: utiliza una adaptación de SA (ver Sección A.3.2.3), que se limita a valores pequeños de $k$ para patrones largos y sólo funciona para $k = 1$ en el caso de $m \in \{24, 32\}$.

MBYPSb y MBYPSc casi siempre superan a todos los demás algoritmos, sólo sobrepasados por MM cuando $m = 8$, $k = 1$ y $r = 1000$ en el alfabeto inglés. En general, hay

| | m=8 | $m = 16$ | | | $m = 24$ | | | $m = 32$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | 1 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | $r$ | $\Sigma$ |
| MM | 0.105 | 0.108 | -[a] | -[a] | 0.109 | -[a] | -[a] | 0.115 | -[a] | -[a] | 10 | ADN |
| FN | 0.246 | 0.033 | 0.249 | 0.873 | 0.012 | 0.015 | 0.028 | 0.008 | 0.010 | 0.016 | | |
| BYN | 0.138 | 0.115 | 0.132 | 0.196 | 0.114 | 0.131 | 0.135 | 0.117 | 0.126 | 0.140 | | |
| HBYN | 0.136 | 0.115 | 0.133 | 0.189 | 0.113 | -[a] | -[a] | 0.110 | -[a] | -[a] | | |
| MBYPS | 0.043 | 0.017 | 0.029 | 0.087 | 0.004 | 0.017 | 0.022 | **0.002** | 0.006 | 0.018 | | |
| MBYPSb | 0.038 | 0.016 | 0.027 | 0.076 | 0.004 | 0.016 | 0.021 | **0.002** | 0.006 | 0.016 | | |
| MBYPSc | **0.028** | **0.012** | **0.020** | **0.060** | **0.003** | **0.012** | **0.016** | **0.002** | **0.005** | **0.013** | | |
| MM | 0.708 | 0.738 | -[a] | -[a] | 0.730 | -[a] | -[a] | 0.758 | -[a] | -[a] | 100 | |
| FN | 2.215 | 0.306 | 1.977 | 9.952 | 0.023 | 0.062 | 0.164 | 0.015 | 0.027 | 0.065 | | |
| BYN | 0.510 | 0.187 | 0.313 | 0.972 | 0.187 | 0.205 | 0.278 | 0.186 | 0.198 | 0.220 | | |
| HBYN | 0.494 | 0.191 | 0.308 | 0.898 | 0.192 | -[a] | -[a] | 0.188 | -[a] | -[a] | | |
| MBYPS | 0.300 | 0.020 | 0.160 | 0.612 | **0.005** | 0.022 | 0.084 | **0.003** | 0.009 | 0.025 | | |
| MBYPSb | 0.261 | 0.018 | 0.137 | 0.541 | **0.005** | 0.021 | 0.077 | **0.003** | **0.007** | 0.022 | | |
| MBYPSc | **0.221** | **0.014** | **0.121** | **0.488** | **0.005** | **0.017** | **0.066** | 0.005 | 0.009 | **0.019** | | |
| MM | 4.763 | 4.931 | -[a] | -[a] | 4.985 | -[a] | -[a] | 5.151 | -[a] | -[a] | 1000 | |
| FN | 22.283 | 3.094 | 21.133 | 97.438 | 0.183 | 0.678 | 1.999 | 0.111 | 0.301 | 0.668 | | |
| BYN | 3.855 | 0.334 | 1.816 | 8.742 | 0.319 | 0.395 | 1.244 | 0.325 | 0.357 | 0.435 | | |
| HBYN | 3.649 | 0.335 | 1.720 | 7.931 | 0.308 | -[a] | -[a] | 0.324 | -[a] | -[a] | | |
| MBYPS | 2.811 | 0.054 | 1.526 | 6.994 | 0.018 | 0.077 | 0.766 | 0.014 | 0.034 | 0.102 | | |
| MBYPSb | 2.463 | 0.049 | 1.438 | 6.496 | **0.013** | 0.070 | 0.744 | **0.012** | **0.024** | 0.093 | | |
| MBYPSc | **2.216** | **0.041** | **1.313** | **5.808** | 0.033 | **0.065** | **0.681** | 0.035 | 0.059 | **0.090** | | |
| MM | 0.053 | 0.054 | -[a] | -[a] | 0.053 | -[a] | -[a] | 0.052 | -[a] | -[a] | 10 | Inglés |
| FN | 0.048 | 0.016 | 0.027 | 0.052 | 0.010 | 0.015 | 0.023 | 0.006 | 0.012 | 0.017 | | |
| BYN | 0.061 | 0.041 | 0.050 | 0.055 | 0.036 | 0.045 | 0.059 | 0.032 | 0.041 | 0.047 | | |
| HBYN | 0.061 | 0.042 | 0.050 | 0.056 | 0.037 | -[a] | -[a] | 0.031 | -[a] | -[a] | | |
| MBYPS | 0.022 | 0.015 | 0.016 | 0.016 | 0.003 | 0.015 | 0.015 | **0.002** | 0.005 | 0.015 | | |
| MBYPSb | 0.020 | 0.013 | 0.015 | 0.015 | 0.003 | 0.013 | 0.014 | **0.002** | **0.004** | 0.013 | | |
| MBYPSc | **0.015** | **0.010** | **0.012** | **0.011** | **0.002** | **0.010** | **0.011** | **0.002** | **0.004** | **0.010** | | |
| MM | 0.068 | 0.075 | -[a] | -[a] | 0.072 | -[a] | -[a] | 0.066 | -[a] | -[a] | 100 | |
| FN | 0.273 | 0.093 | 0.174 | 0.397 | 0.063 | 0.095 | 0.159 | 0.048 | 0.069 | 0.105 | | |
| BYN | 0.146 | 0.111 | 0.146 | 0.224 | 0.104 | 0.115 | 0.134 | 0.093 | 0.109 | 0.117 | | |
| HBYN | 0.146 | 0.110 | 0.146 | 0.218 | 0.094 | -[a] | -[a] | 0.096 | -[a] | -[a] | | |
| MBYPS | 0.063 | 0.018 | 0.049 | 0.096 | 0.005 | 0.018 | 0.032 | 0.003 | **0.006** | 0.019 | | |
| MBYPSb | 0.054 | 0.016 | 0.043 | 0.089 | **0.004** | 0.016 | 0.029 | **0.002** | **0.006** | 0.017 | | |
| MBYPSc | **0.040** | **0.012** | **0.033** | **0.068** | **0.004** | **0.013** | **0.024** | 0.003 | **0.006** | **0.014** | | |
| MM | **0.242** | 0.272 | -[a] | -[a] | 0.286 | -[a] | -[a] | 0.274 | -[a] | -[a] | 1000 | |
| FN | 3.235 | 0.679 | 2.299 | 7.809 | 0.368 | 0.910 | 1.887 | 0.270 | 0.613 | 1.047 | | |
| BYN | 0.616 | 0.208 | 0.563 | 1.512 | 0.186 | 0.238 | 0.413 | 0.178 | 0.212 | 0.258 | | |
| HBYN | 0.602 | 0.203 | 0.541 | 1.436 | 0.182 | -[a] | -[a] | 0.174 | -[a] | -[a] | | |
| MBYPS | 0.372 | 0.037 | 0.332 | 0.920 | 0.014 | 0.048 | 0.192 | 0.010 | 0.025 | 0.060 | | |
| MBYPSb | 0.327 | 0.032 | 0.299 | 0.823 | **0.011** | **0.041** | 0.162 | **0.008** | **0.019** | **0.051** | | |
| MBYPSc | 0.251 | **0.029** | **0.225** | **0.688** | 0.024 | 0.045 | **0.152** | 0.023 | 0.039 | 0.061 | | |

[a] El algoritmo no fue diseñado para funcionar en este caso.

Tabla 4.2: Tiempos de búsqueda en segundos de algoritmos para string matching aproximado de múltiples patrones con hasta $k$ sustituciones.

una mayor diferencia en el tiempo de ejecución para relaciones de diferencia más bajas y un alfabeto más extenso (es decir, inglés), como se muestra en sus complejidades teóricas en la Tabla A.B.1. A veces incluso superan a otros algoritmos por un orden de magnitud, siendo el mejor rendimiento relativo de MBYPSb alrededor de 24 veces (0.048/0.002) más rápido que el segundo algoritmo más eficiente distinto de MBYPS/MBYPSc (que es FN) para el alfabeto inglés, $r = 100$, $m = 32$ y $k = 1$. MBYPSc es más dominante que MBYPSb, a diferencia del caso de BYPSc comparado con BYPSb, aunque a veces es inferior en los casos de grandes $r$ y grandes longitudes de subpatrones ($l \geq 8$), lo que se corresponde con bajos ratios de diferencia debido a que $l = \lfloor m/(k+1) \rfloor$ (véase la Sección 3.2 para más información sobre los detalles del algoritmo).

En cuanto a otros algoritmos, cabe mencionar lo siguiente::

- El algoritmo de Muth–Manber sigue siendo competitivo sólo para grandes conjuntos de pequeños patrones en el alfabeto inglés, específicamente para $r = 1000$ y $m = 8$ en nuestras pruebas, mostrando su tolerancia a grandes conjuntos de patrones, como se expresa en la Sección A.3.2.2 y en su complejidad en la Tabla A.B.1.

- El algoritmo de Fredriksson–Navarro a veces se comporta de forma similar a MBYPS en el alfabeto de ADN para pequeños grupos de patrones largos, aunque nunca lo supera.

- El algoritmo de Baeza-Yates–Navarro adaptado a la distancia de Hamming siempre fue superado por MBYPS o sus variantes, pero ha sido el algoritmo no basado en SIMD más rápido en muchos casos de altas relaciones de diferencia o grandes conjuntos de patrones.

Un caso curioso que vale la pena analizar se presenta cuando se ejecuta un algoritmo para $r_1$ y $r_2$ donde $r_2 > r_1$, y manteniendo los mismos $m$, $k$ y alfabeto, produce tiempos de búsqueda $e_1$ y $e_2$ respectivamente donde resulta que $e_2 > \frac{r_2}{r_1} e_1$. Por ejemplo, en la Tabla 4.2, podemos ver que MBYPSc tarda 0.488 segundos en buscar las ocurrencias de patrones de $m = 16$, $k = 3$ y $r = 100$ en el alfabeto de ADN, mientras que se necesitan 5.808 segundos para llevar a cabo la misma tarea para un valor de $r = 1000$, alrededor de doce veces más de lo que se tardó en la búsqueda anterior. Se han utilizado dos técnicas de agrupado de patrones diferentes para abordar este problema en el algoritmo FN en [15] que se denominan *pattern grouping* y *pattern clustering*. La primera consiste simplemente en dividir el conjunto de patrones en conjuntos más pequeños y buscarlos por separado. La segunda es una extensión de la primera, donde la forma en que se dividen los subconjuntos es beneficiosa para el mecanismo de búsqueda, invirtiendo más tiempo en la fase de preprocesamiento. Desafortunadamente, estas mejoras no funcionarían para MBYPS o sus variantes, ya que la razón de su disminución de velocidad depende de su técnica de filtrado. Cuando el número de subpatrones a buscar comienza a crecer, los mismos subpatrones comienzan a aparecer en muchos patrones. Así que encontrarlos en el texto no filtraría mucho, ya que todavía necesitamos verificar una ocurrencia aproximada de todos los patrones que los contienen. En conclusión, la

aplicación del agrupamiento de patrones en cualquiera de estos algoritmos sólo los haría recorrer el texto muchas veces, aumentando su tiempo de búsqueda.

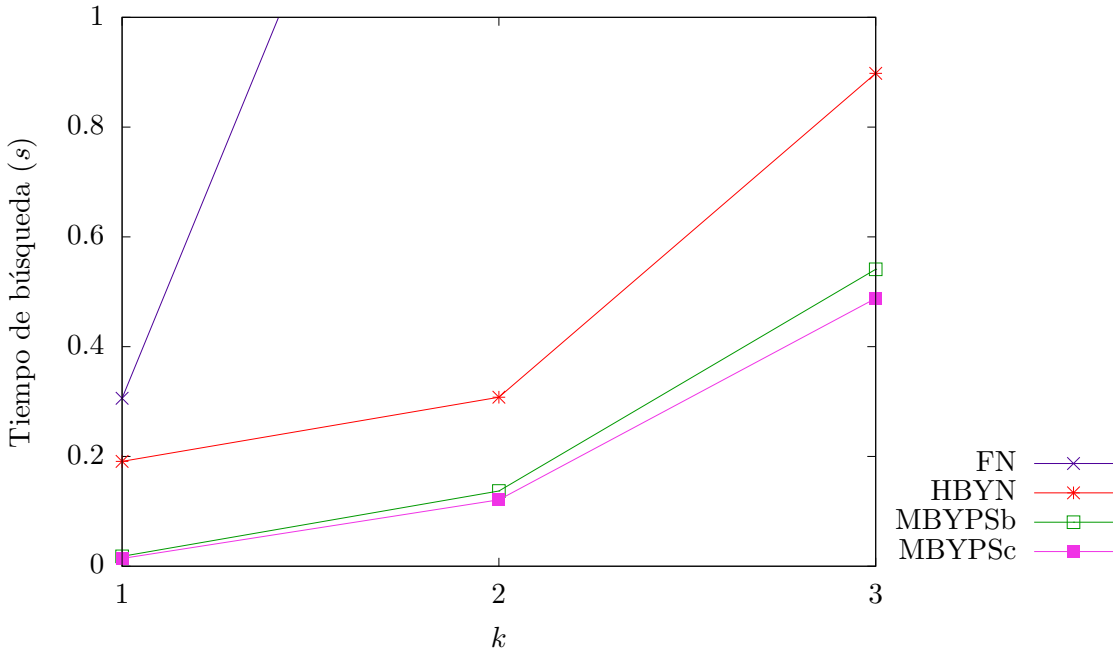



Figura 4.8: Tiempos de búsqueda de los algoritmos testeados en función de $k$ para $m = 16$ y $r = 100$ en alfabeto de ADN.

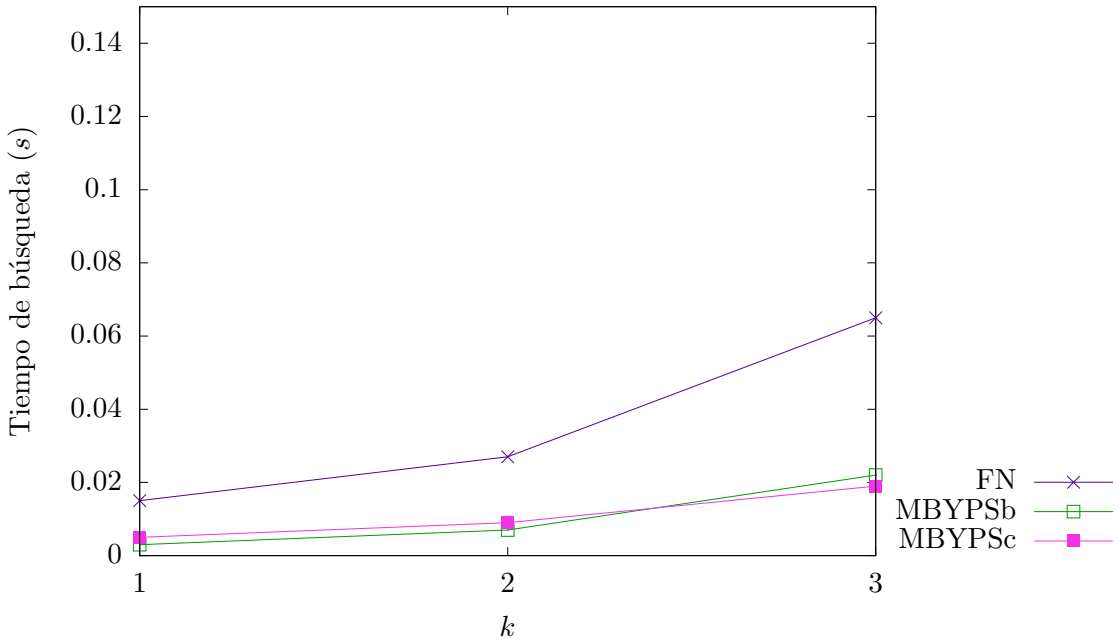



Figura 4.9: Tiempos de búsqueda de los algoritmos testeados en función de $k$ para $m = 32$ y $r = 100$ en alfabeto de ADN.

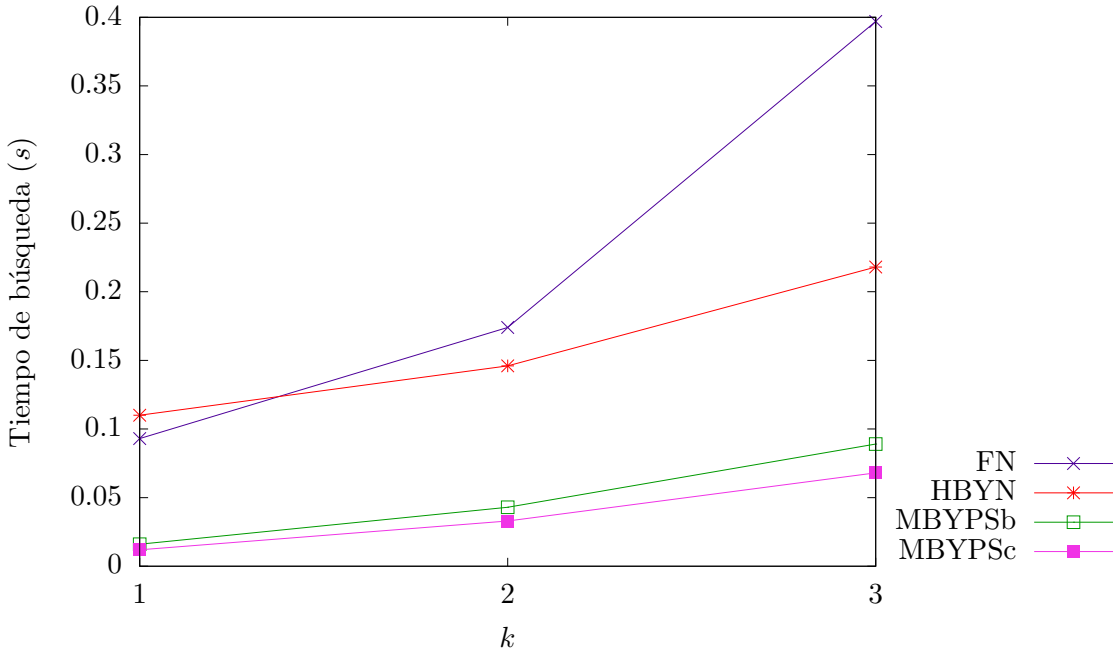



Figura 4.10: Tiempos de búsqueda de los algoritmos testeados en función de $k$ para $m = 16$ y $r = 100$ en alfabeto inglés.

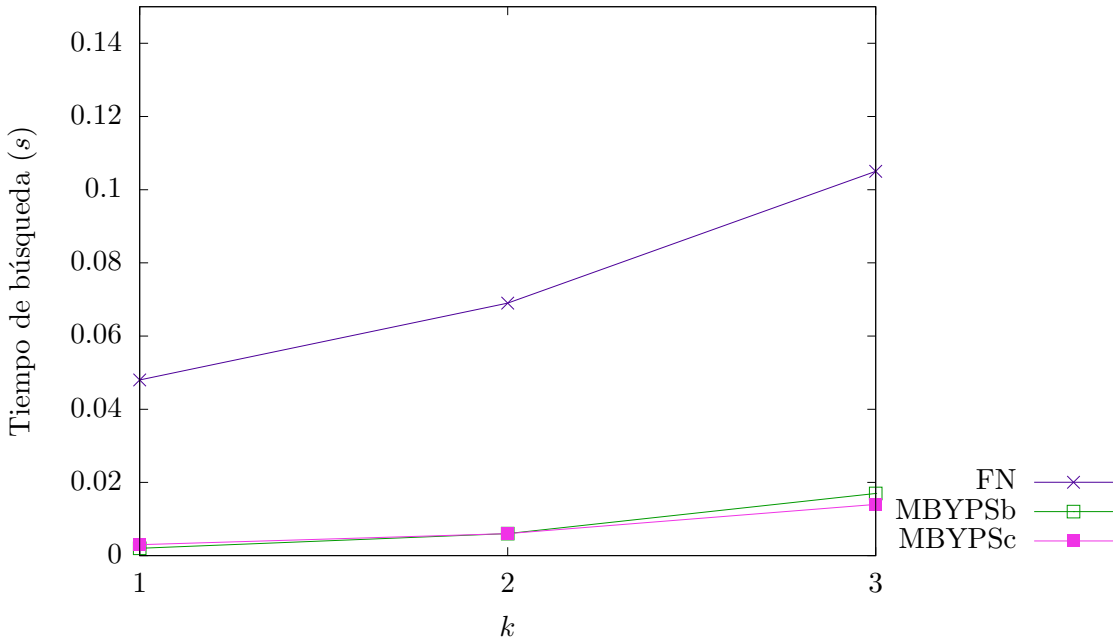



Figura 4.11: Tiempos de búsqueda de los algoritmos testeados en función de $k$ para $m = 32$ y $r = 100$ en alfabeto inglés.

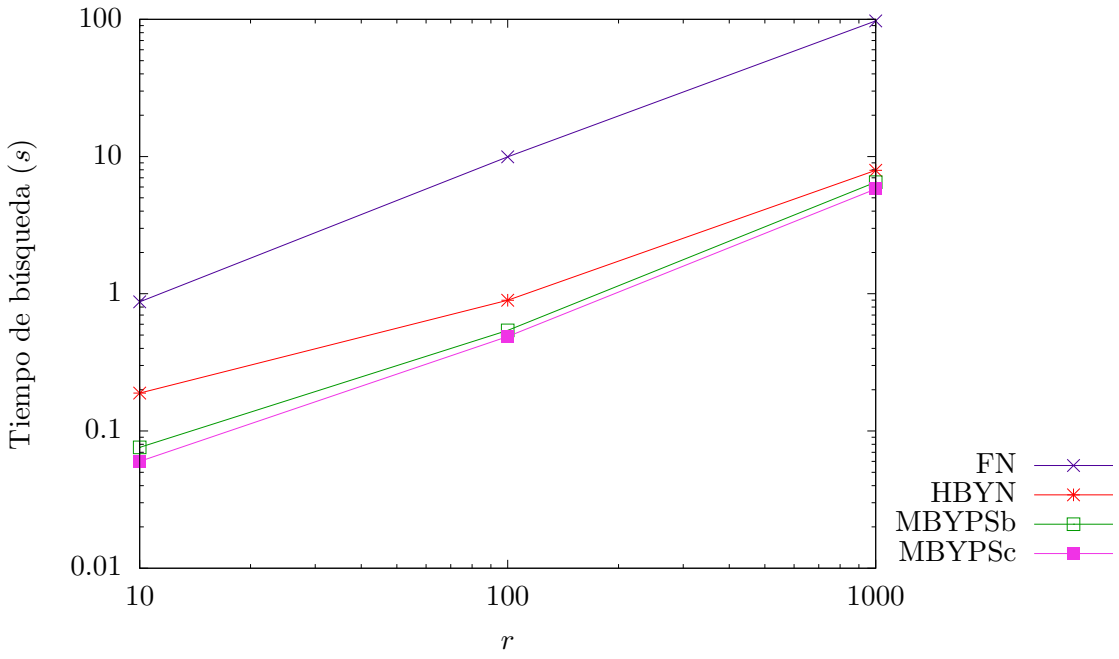



Figura 4.12: Tiempos de búsqueda de los algoritmos testeados en función de $r$ para $m = 16$ y $k = 3$ (ratio de diferencia alto) en alfabeto de ADN.

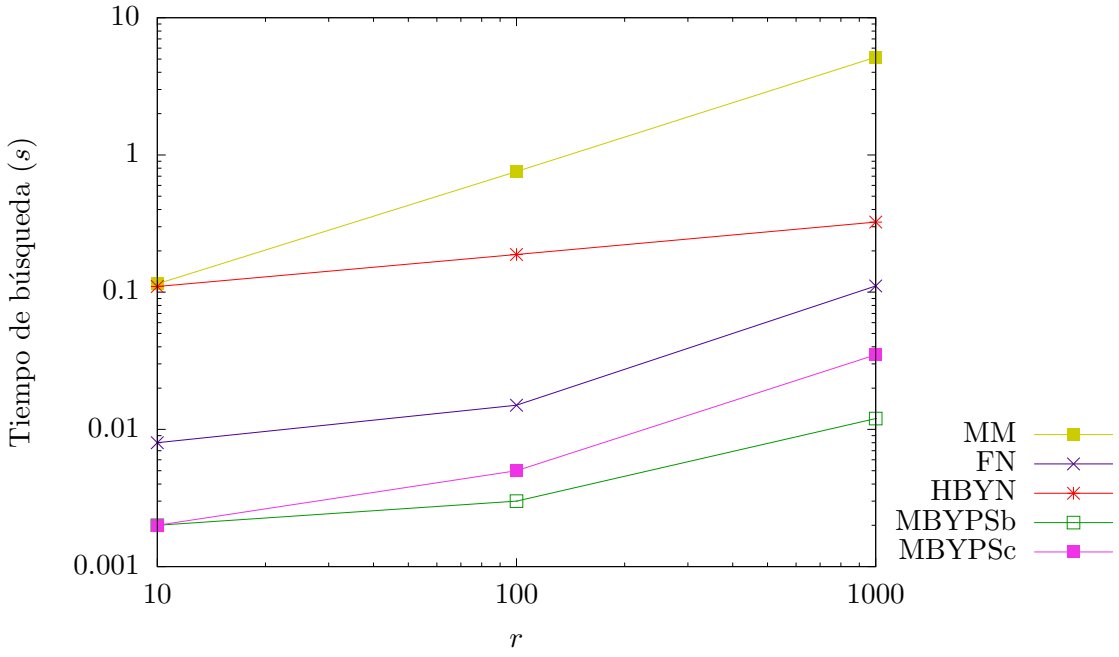



Figura 4.13: Tiempos de búsqueda de los algoritmos testeados en función de $r$ para $m = 32$ y $k = 1$ (ratio de diferencia bajo) en alfabeto de ADN.

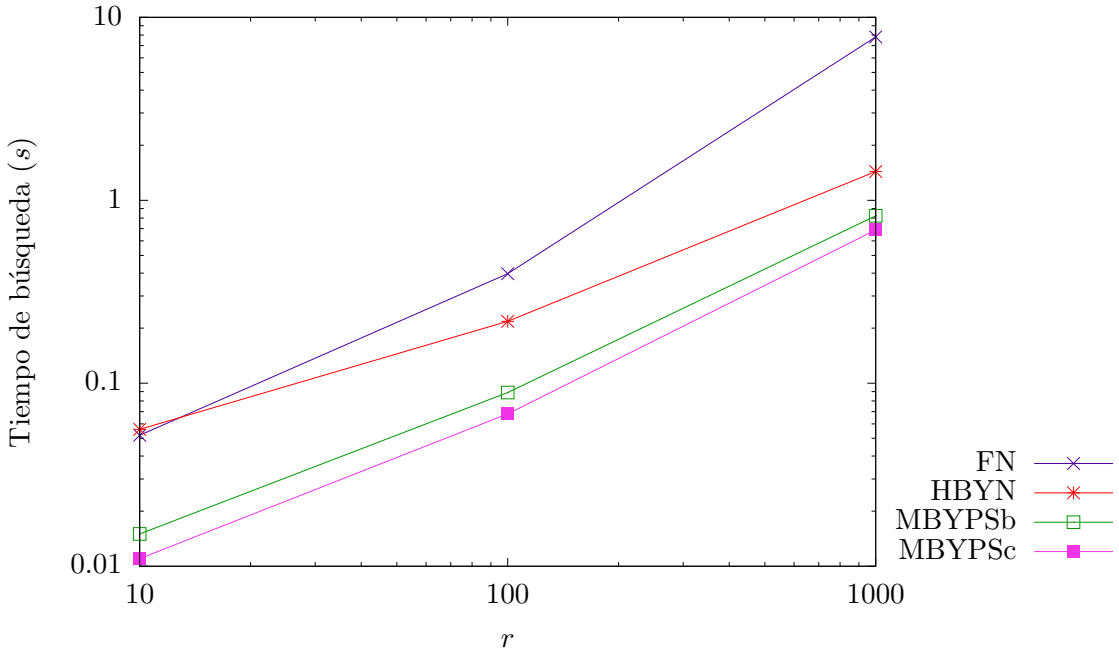



Figura 4.14: Tiempos de búsqueda de los algoritmos testeados en función de $r$ para $m = 16$ y $k = 3$ (ratio de diferencia alto) en alfabeto inglés.

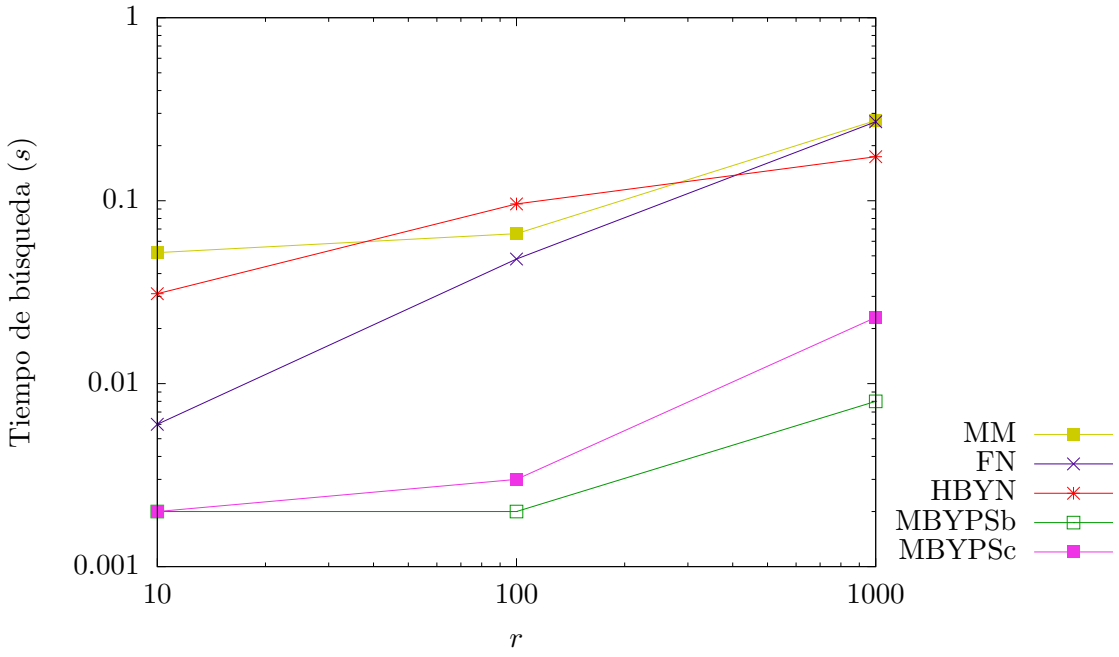



Figura 4.15: Tiempos de búsqueda de los algoritmos testeados en función de $r$ para $m = 32$ y $k = 1$ (ratio de diferencia bajo) en alfabeto inglés.

# Capítulo 5

# Conclusiones

Hemos demostrado que las soluciones SIMD simples son competitivas para buscar ocurrencias aproximadas de uno o varios patrones bajo distancia Hamming para patrones de un máximo de 32 caracteres de longitud. Nuestros experimentos en el Capítulo 4 y el Apéndice A.A muestran que los algoritmos basados en SIMD son las opciones más rápidas para un pequeño número de sustituciones ($k \leq 3$) y conjuntos de un máximo de 1000 patrones en ADN y alfabeto inglés usando textos de la vida real. Es importante notar que aunque los algoritmos basados en SIMD presentados en esta tesis tienen peores complejidades de tiempo teóricas que los algoritmos clásicos, funcionan mejor en la práctica.

## 5.1. Contribuciones

Parte de los resultados presentados en esta tesis ya han sido publicados en un artículo de conferencia en coautoría con Waltteri Pakalén y Jorma Tarhio [14]. Más precisamente, se presentó el trabajo realizado por Waltteri Pakalén en el desarrollo de ANS/ANS2 (posteriormente mejorado por mí en ANS2b), y mi desarrollo de BYPS y MBYPS. Jorma Tarhio cumplió la función de ser mi tutor durante mi trabajo.

### ANS2b

Introdujimos una modificación en Approximate Naive mejorado con SIMD [14] (ANS/ANS2, Sección A.4.2.1) llamado ANS2b (Sección 3.1.1), que mejora el rendimiento en patrones pequeños (de un máximo de 16 caracteres) en comparación con ANS2. ANS2 tiene una dependencia de rendimiento sobre $k$ para $m \leq 16$ causada por predicciones erróneas de saltos condicionales, como se muestra en las Figuras 4.5 y 4.6. ANS2b no tiene este problema porque reemplaza su sentencia condicional más evaluada por una sola instrucción, obteniendo un tiempo de ejecución constante para todos los $k$ para $m \leq 16$. Esta variante resultó en el algoritmo más eficiente para el problema de buscar un solo patrón en casos de alto ratio de diferencia, como se muestra en la Sección 4.1.

### BYPS/BYPSb/BYPSc

Presentamos un nuevo algoritmo para string matching de patrón único con a lo sumo $k$ sustituciones llamado Baeza-Yates–Perleberg mejorado con SIMD (BYPS, Sección 3.1.2). Éste emplea un método de filtrado basado en la técnica SIMD descripta en la Sección A.4.1.2 para calcular los valores hash.

También introdujimos una variante del mismo que utiliza ANS2b en lugar de ANS2 y una función de comparación de cadenas basada en SIMD, que llamamos BYPSb. Además, presentamos una tercera versión de BYPS que omite el paso de filtrado de la comparación exacta de cadenas y asume una función de hash perfecta. Lo llamamos BYPSc.

BYPSb y BYPSc demostraron ser los algoritmos más rápidos para el problema de patrón único en casos de bajo ratio de diferencia, siendo BYPSb más útil en el alfabeto de ADN y BYPSc en el alfabeto inglés.

### MBYPS/MBYPSb/MBYPSc

Desarrollamos un nuevo algoritmo para string matching de múltiples patrones bajo distancia Hamming llamado Baeza-Yates–Perleberg mejorado con SIMD para Múltiples patrones (MBYPS, Sección 3.2), que es una extensión de BYPS. De la misma manera que en BYPS, propusimos una variación del mismo que emplea ANS2b en lugar de ANS2 y una comparación de cadenas basada en SIMD llamada MBYPSb, y otra que omite el paso comparación exacta de cadenas llamado MBYPSc. En este caso, la última versión resultó ser más rápida que las anteriores más a menudo que lo observado en el caso de BYPSc.

Como resultado, MBYPSb y MBYPSc fueron los algoritmos más rápidos para el problema en casi todos los casos testeados. Cabe destacar su importancia como un algoritmo que se ejecuta más rápido que todos los algoritmos analizados en esta tesis, que son los más relevantes en el campo hoy en día.

### HBYN

Modificamos el algoritmo de Baeza-Yates–Navarro, que originalmente fue diseñado para trabajar bajo distancia de edición, para trabajar bajo distancia de Hamming (ver Sección A.3.2.3). Si bien no está basado en SIMD, obtuvimos un algoritmo razonablemente competitivo para este problema, quedando sólo por detrás del algoritmo basado en SIMD MBYPS y sus variantes en muchos casos.

## 5.2. Trabajo Futuro

En los siguientes puntos se exponen algunas líneas de trabajo interesantes para continuar la investigación en el tema.

- Analizar la utilidad de los algoritmos para $k$ más grandes, mayores ratios de diferencia (especialmente ANS2b) y mayores longitudes de patrones (especialmente BYPS y sus variantes y MBYPS y sus variantes).
- Extender los algoritmos basados en SIMD para que funcionen con AVX-512. Puede ser posible lograr mejores velocidades porque las instrucciones de comparación y enmascaramiento se han fusionado en una sola operación `VPCMPB` en AVX-512BW, como se describe en la Sección A.2.3.3. De esta manera, la función *simd-cmpeq*$(a, b)$ de la Sección A.4.1.1 podría definirse simplemente usando sólo

  ```
  __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b)
  ```

  la cual compara 64 bytes almacenados en $a$ y $b$ por igualdad, y devuelve el resultado como una máscara en un valor de 64 bits. Tener en cuenta que esta función también permite extender ANS, ANS2 y ANS2b para trabajar eficientemente con patrones de hasta 64 caracteres.
- Investigar más formas de explotar las instrucciones SIMD. Tener en cuenta que la latencia de las instrucciones STTNI (ver Sección A.2.3.2) puede cambiar en futuras arquitecturas, haciéndolas potencialmente adecuadas para el desarrollo de algoritmos de alto rendimiento.
- Extender el uso de SIMD a los algoritmos para string matching aproximado bajo la *distancia de edición*.
- Estudiar más formas de adaptar el autómata del algoritmo Baeza-Yates–Navarro para que funcione bajo la distancia de Hamming, dado que el algoritmo obtenido en la Sección A.3.2.3 (HBYN) demostró ser una solución competitiva no basada en SIMD para el caso de patrones múltiples.

# Bibliografía


[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers y D. J. Lipman. «Basic local alignment search tool». En: *Journal of Molecular Biology* 215.3 (1990), págs. 403-410. ISSN: 0022-2836. DOI: https://doi.org/10.1016/S0022-2836(05)80360-2.

[2] S. Altschul, T. Madden, A. Schäffer, J. Zhang, Z. Zhang, W. Miller y D. Lipman. «Gapped BLAST and PSI-BLAST: a new generation of protein databases search programs». En: *Nucleic acids research* 25 (oct. de 1997), págs. 3389-3402. DOI: 10.1093/nar/25.17.3389.

[3] R. Baeza-Yates y G. Gonnet. «A new approach to text searching». En: *Communications of the ACM* 35.10 (1992), págs. 74-82.

[4] R. Baeza-Yates y G. Navarro. «New and faster filters for multiple approximate string matching». En: *Random Structures & Algorithms* 20.1 (2002), págs. 23-49.

[5] R. Baeza-Yates y C. Perleberg. «Fast and practical approximate string matching». En: *Information Processing Letters* 59.1 (1996), págs. 21-27.

[6] R. Baeza-Yates y B. Ribeiro-Neto. «Modern Information Retrieval». En: *Addison-Wesley* (1999).

[7] T. Chhabra, S. Faro, M. O. Külekci y J. Tarhio. «Engineering order-preserving pattern matching with SIMD parallelism». En: *Software: Practice and Experience* 47.5 (2017), págs. 731-739.

[8] R. Dixon y T. Martin. «Automatic speech and speaker recognition». En: *IEEE Press* (1979).

[9] B. Ďurian, T. Chhabra, S. Guman, T. Hirvola, H. Peltola y J. Tarhio. «Improved two-way bit-parallel search». En: *Proceedings of the Prague Stringology Conference* (2014), págs. 71-83.

[10] D. Elliman e I. Lancaster. «A review of segmentation and contextual analysis techniques for text recognition». En: *Pattern Recognition* 23.3/4 (1990), págs. 337-346.

[11] S. Faro y M. O. Külekci. «Fast packed string matching for short patterns». En: *Proc. 15th Meeting on Algorithm Engineering and Experiments, SIAM* (2013), págs. 113-121.

[12] S. Faro y M. O. Külekci. «Towards a Very Fast Multiple String Matching Algorithm for Short Patterns». En: *Proceedings of the Prague Stringology Conference* (2013), págs. 78-91.

[13] S. Faro y E. Pappalardo. «Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem». En: *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science. Lecture Notes in Computer Science* 5901 (2010), págs. 260-281.

[14] F. J. Fiori, W. Pakalén y J. Tarhio. «Counting Mismatches with SIMD». En: *Proceedings of the Prague Stringology Conference* (2017), págs. 51-61.

[15] K. Fredriksson y G. Navarro. «Average-optimal Single and Multiple Approximate String Matching». En: *J. Exp. Algorithmics* 9 (2004).

[16] S. Gog, K. Karhu, J. Karkkainen, V. Makinen y N. Valimaki. «Multi-pattern matching with bidirectional indexes». En: *Computing and Combinatorics, J. Gudmundsson, J. Mestre, and T. Viglas. Lecture Notes in Computer Science* 7434 (2012), págs. 384-395.

[17] D. Higgins y W. R. Taylor. *Bioinformatics: Sequence, Structure, and Databanks: a Practical Approach*. Oxford University Press, 2000. Cap. 3, págs. 173-174. ISBN: 9780199637904.

[18] T. Hirvola. «Bit-parallel approximate string matching under Hamming distance». Master's Thesis. Aalto University, 2016.

[19] A. Hume y D. Sunday. «Fast string searching». En: *Software: Practice and Experience* 21.11 (1991), págs. 1221-1248.

[20] Intel. *Intel (R) 64 and IA-32 Architectures Software Developer's Manual*. December 2017. URL: `https://software.intel.com/en-us/articles/intel-sdm`.

[21] K. Kukich. «Techniques for automatically correcting words in text». En: *ACM Computing Surveys* 24.4 (1992), págs. 377-439.

[22] M. O. Külekci. «Filter based fast matching of long patterns by using SIMD instructions». En: *Proceedings of the Prague Stringology Conference* (2009), págs. 118-128.

[23] S. Kumar y E. Spafford. «A pattern-matching model for intrusion detection». En: *Proc. National Computer Security Conference* (1994), págs. 11-21.

[24] S. Ladra, O. Pedreira, J. Duato y N. R. Brisaboa. «Exploiting SIMD instructions in current processors to improve classical string algorithms». En: *Proc. 16th East European Conference on Advances in Databases and Information Systems, LNCS* 7503 (2012), págs. 254-267.

[25] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001, pág. 11. ISBN: 9780879697129.

[26] R. Muth y U. Manber. «Approximate multiple string search». En: *Proc. 7th Symposium on Combinatorial Pattern Matching, LNCS* 1075 (1996), págs. 75-86.

[27] L. Salmela, J. Tarhio y P. Kalsi. «Approximate Boyer-Moore string matching for small alphabets». En: *Algorithmica* 58.3 (2010), págs. 591-609.

[28] J. Tarhio, J. Holub y E. Giaquinta. «Technology beats algorithms (in exact string matching)». En: *Software: Practice and Experience* (2017).

# Apéndice A

# Versión en Inglés

En este apéndice presentamos la versión extendida de la tesina en el idioma inglés.


Universidad Nacional de Rosario

Graduate Thesis

# Approximate String Matching Improved with SIMD

*Author:*
Fernando Jesús Fiori

*Advisor:*
Dr. Jorma Tarhio

Computer Science Department
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
250 Pellegrini Av., Rosario, Santa Fe, Argentina

December 9, 2019

# Abstract

We consider the $k$ mismatches version of approximate string matching for a single pattern and multiple patterns. The problem basically consists of finding all occurrences of one or more patterns with at most $k$ mismatches in a text.

Algorithms for this problem have several applications such as virus and intrusion detection, spelling, speech recognition, optical character recognition, handwriting recognition, text retrieval under synonym or thesaurus expansion, among others. With the availability of large amounts of DNA data, matching of nucleotide sequences in metagenomics in computational biology has become another important application. Finding a gene in a new organism (e.g., a crop plant) with a sequence similar to a model organism gene (e.g., yeast) provides a prediction that the new gene has the same function as in the model organism.

Given the high popularity of SIMD (Single Instruction Multiple Data) instruction set extensions in nowadays CPUs, we present new efficient algorithms for this problem that take advantage of them. We apply SIMD computation in two ways: in counting of mismatches and in calculation of fingerprints.

We measure the performance of each new algorithm by thorough testing on different real life texts against the most competitive algorithms known to date. We take a practical approach by trying to improve the average time spent by each algorithm.

# Contents

# Chapter A.1

# Introduction

## A.1.1. Goals and Scope

The aim of this thesis is to improve algorithms for single and multiple string matching under $k$ mismatches. Our goal is to obtain better results than the fastest algorithms up to date, and thus contribute to the state of the art of the topic. We will investigate different ways of using *Single Instruction Multiple Data* (SIMD) instructions in order to achieve it, which will include calculation of fingerprints (as done in [12]) and a mismatch counting method (as done in [14]). SIMD [23] is a type of parallel architecture that allows one instruction to be operated on multiple data items at the same time.

We will measure the performance of each new algorithm by thorough testing on different real life alphabets against the most competitive algorithms known to date. We take a practical approach by trying to improve the average time spent by the algorithms.

## A.1.2. Main Results

We present two new SIMD-based algorithms for single and multiple pattern matching under $k$ mismatches: Baeza-Yates–Perleberg enhanced with SIMD (BYPS, Section A.5.1.2) and Multiple-pattern Baeza-Yates–Perleberg enhanced with SIMD (MBYPS, Section A.5.2). Each of them resulted in the fastest algorithm for their corresponding problem in many cases as demonstrated in Chapter A.6.

We also introduce a modified version of Approximate Naive enhanced with SIMD [14] (ANS/ANS2, Section A.4.2.1), which we call ANS2b (Section A.5.1.1), that improves performance on small patterns (no longer than 16 characters) compared to ANS/ANS2. Furthermore, we replace ANS2 with this variation in BYPS and MBYPS, and we also use a SIMD-based exact string comparison in them, obtaining even faster algorithms as shown in Chapter A.6. We call them BYPSb and MBYPSb respectively. In addition, we implement versions that skip an intermediate filtering step in BYPSb and MBYPSb and assume perfect hashing. They are named BYPSc and MBYPSc, and are useful in some cases.

### A.1.3. Structure of the Thesis

The rest of the thesis is divided into six chapters. Chapter A.2 provides background knowledge, definitions and notation used throughout the thesis, as well as some of the most important applications of these kind of algorithms. Chapter A.3 describes the existing most relevant solutions to the single and multiple pattern matching under $k$ mismatches problem that do not use SIMD instructions. Chapter A.4 presents two techniques to take advantage of SIMD instructions for the problem addressed. It also describes two solutions to the single pattern matching under $k$ mismatches problem that employ SIMD instructions, and a solution to the problem of exact multiple string matching that makes use of SIMD instructions which has been useful to develop other algorithms. The algorithms presented in this chapter are already known and were not developed over the course of this thesis. Chapter A.5 introduces three new competitive SIMD-based algorithms for single and multiple pattern matching with up to $k$ mismatches. Chapter A.6 evaluates the practical performance of developed algorithms compared to the existing ones described throughout the thesis with experiments. Finally, Chapter A.7 summarizes the contributions of the thesis and states some interesting lines of work to continue exploring.

# Chapter A.2

# Background

## A.2.1. Problem definition and Notation

The *string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ in an alphabet $\Sigma$ of size $\sigma$, find all the occurrences of $P$ in $T$.

We consider the $k$ *mismatches* variation of the problem, where $P'$ is an occurrence of $P$ if $|P'| = |P|$ holds (we use $|S|$ to denote the length of a string $S$) and $P'$ has at most $k$ mismatches with $P$. The mismatch distance between two strings of equal length is also called the *Hamming distance*. We will refer to *difference ratio* as $\alpha = k/m$, which is an indicator of how 'different' the pattern and its approximate occurrences can be.

There is also a more general version of the problem which searches pattern matches with at most $k$ *differences* or $k$ *errors*, also known as having an *edit distance* of at most $k$. In this context, *differences* are defined as not only mismatches, but also insertions and deletions of characters. Analysis of this problem is out of the scope of the thesis.

Besides the single pattern problem, we also consider the multiple pattern variation where there are $r$ patterns $P^0, \cdots, P^{r-1}$ to search. Pattern $P^i$ has length $m_i$ and we denote $P^i = p_0^i \cdots p_{m_i-1}^i$.

For the sake of simplicity we will assume that one byte represents one character. We will also assume $m_i = m$ for all $i$. If there were different lengths, we would work with $m' = \min_{0 \leq i < r} \{m_i\}$ and check every match candidate for the prefixes of length $m'$ of all patterns.

We will work with the *online* version of the problem, in which the patterns are known beforehand but the text is not. In this way the only preprocessing that can be done is on the patterns. On the other hand, the *offline* variation assumes a known, fixed text and patterns can vary, so the preprocessing phase may focus on the text.

We will refer to bitwise operations AND, OR, NOT, left shift and right shift with the same symbols as in programming language C: '&', '|', '$\sim$', '<<' and '>>', respectively. Binary numbers will be distinguished by having a subscript 2 in its last digit, and the notation $X_2^Y$, where $X$ is a binary number and $Y \geq 0$ is an integer, means that $X$ is repeated $Y$ times. For example, $01^3 0^2 1_2^2 = 01110011_2$ and $(01_2^3)_2^2 = 01110111_2$.

## A.2.2. Applications

Algorithms for this problem have several applications such as virus and intrusion detection [26], spelling [24], speech recognition [8], optical character recognition [10], handwriting recognition [10], text retrieval under synonym or thesaurus expansion [6], among others.

With the availability of large amounts of DNA data, matching of nucleotide sequences in metagenomics in computational biology [1, 2, 13, 16] has become another important application. Finding a gene in a new organism (e.g., a crop plant) with a sequence similar to a model organism gene (e.g., yeast) provides a prediction that the new gene has the same function as in the model organism [29]. Also, in the evolution of protein sequences, not all regions mutate at the same rate. Regions which are essential for the structure and function of proteins are more conserved. Therefore, significant sequence similarity of two proteins may reflect a close biological function or a common evolutionary origin. When the sequence similarity is statistically significant, it can be deduced at a high confidence level that the sequences are related [18].

Moreover, some single-pattern approximate search algorithms resort to multipattern searching of pattern pieces [5]. Depending on the application, $r$ may vary from a few to thousands of patterns.

## A.2.3. SIMD computation

*Single Instruction Multiple Data* (SIMD) [23] is a type of parallel architecture that allows one instruction to be operated on multiple data items at the same time, as depicted in Figure A.2.1 for example. Initially, SIMD was used in multimedia, especially in processing images or audio files. SIMD instructions have since found applications in other areas such as cryptography, and more recently, they have also been applied to string matching [7, 11, 25, 27, 33].

This section describes the most important features implemented by each set of instructions that involve SIMD in Intel processors, emphasizing those relevant to our problem. There is a summary of registers introduced by each SIMD extension and data types available to use at the end of the section in Table A.2.2. We only focus on Intel's SIMD architecture for popularity reasons.

### A.2.3.1. MMX

*MMX technology* was released in 1997. It defines a simple SIMD execution model to handle 64-bit packed integer data.

It introduces eight registers, called MM0 through MM7, and operations that employ them. Each register is 64 bits wide and can be used to hold either 64-bit integers or multiple smaller integers in a "packed" format, thus defining three new data types:

- 64-bit packed byte integers.

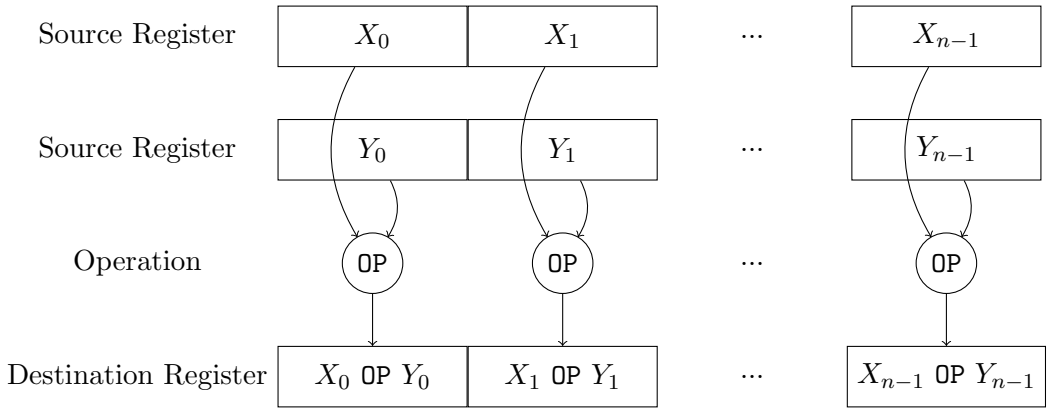



Figure A.2.1: Simple SIMD operation example.

- 64-bit packed word[1] integers.
- 64-bit packed doubleword integers.

A single MMX instruction can then be applied to two 32-bit integers, four 16-bit integers, or eight 8-bit integers at once.

#### A.2.3.2. SSE

*Streaming SIMD Extensions* (SSE) was made publicly available in 1999. It expands the capabilities of MMX incorporating the following:

- Eight 128-bit data registers (called XMM registers) in non-64-bit modes, whereas there are sixteen XMM registers in 64-bit mode.
- The 128-bit packed single-precision floating-point data type.
- Instructions that perform SIMD operations on single-precision floating-point values, and that extend SIMD operations on integers:
  - 128-bit packed and scalar single-precision floating-point instructions that operate on data located in MMX registers.
  - 64-bit SIMD integer instructions that support additional operations on packed integer operands located in MMX registers.

SSE2 extensions (2001) use the SIMD execution model that is used with MMX technology and SSE extensions. The following data types were added:

- 128-bit packed double-precision floating-point.

[1] A *word* consists of 16 bits in Intel terminology, which will be used throughout this document.

- 128-bit packed byte integers.
- 128-bit packed word integers.
- 128-bit packed doubleword integers.
- 128-bit packed quadword integers.

SSE3, SSSE3, SSE4, SSE4.1 and SSE4.2 do not introduce new data types, but provide new instructions. XMM registers are used to operate on packed integer data, single-precision floating-point data, or double-precision floating-point data.

SSE4.2 (2008) incorporates a family of CRC32 instructions which provide hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols. They have already been used in string matching algorithms as a fast method for calculating fingerprints in [12]. They operate on general-purpose registers and can take source operand of different sizes. There are four instructions in total, which can be emitted with intrinsics functions with the following form:

```
unsigned Z _mm_crc32_uX(unsigned int seed, unsigned Y data)
```

where *seed* is the initial value used to calculate the CRC32, $X$ refers to the size of the data argument in bits, $Y$ is the type of the data argument and $Z$ is the return type. All the available functions are listed in Table A.2.1.

| Function name | Data argument size (bits) | Return value size (bits) |
|---|---|---|
| _mm_crc32_u8 | 8 | 32 |
| _mm_crc32_u16 | 16 | 32 |
| _mm_crc32_u32 | 32 | 32 |
| _mm_crc32_u64 | 64 | 64 |

Table A.2.1: CRC32 family of intrinsic functions in SSE4.2.

The algorithm used by `_mm_crc32_u64` for calculating CRC32 values works as described in Algorithm 1. In this algorithm, operations are used with the following meanings:

- `BIT_REFLECTx(SOURCE[x-0])`: reverses the $x$ bit values in `SOURCE` so the return value's $i$-th bit has the same value as the source's $(x - i)$-th bit, for all $i$ from 0 to $x$.
- `T1 MOD2 T2`: calculates the remainder from polynomial division modulo 2 between `T1` and `T2`. Each bit in the operands and return value is interpreted as a coefficient of a polynomial.

SSE4.2 also allocates four operation codes to provide string and text processing capabilities (called *STring and Text New Instructions*, or STTNI) that traditionally required many more operations:

**Algorithm 1**: `_mm_crc32_u64(DST, SRC)`

```
TEMP1[63-0] ← BIT_REFLECT64(SRC[63-0])
TEMP2[31-0] ← BIT_REFLECT32(DST[31-0])
TEMP3[95-0] ← TEMP1[63-0] << 32
TEMP4[95-0] ← TEMP2[31-0] << 64
TEMP5[95-0] ← TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] ← TEMP5[95-0] MOD2 0x11EDC6F41
DST[31-0] ← BIT_REFLECT32(TEMP6[31-0])
DST[63-32] ← 0x00000000
```

- `PCMPESTRI`: Packed compare explicit-length strings, return index in ECX/RCX.
- `PCMPESTRM`: Packed compare explicit-length strings, return mask in XMM0.
- `PCMPISTRI`: Packed compare implicit-length strings, return index in ECX/RCX.
- `PCMPISTRM`: Packed compare implicit-length strings, return mask in XMM0.

They take three operands in this order: a string in a 128-bit XMM register, a second string in a 128-bit XMM register or memory location, and an 8-bit mode mask. The string operands are either `NULL`-terminated implicit-length strings (`PCMPISTRI`, `PCMPISTRM`) or explicit-length strings (`PCMPESTRI`, `PCMPESTRM`). The instructions return either a bit-mask (`PCMPESTRM`, `PCMPISTRM`) or the index of the first 1 bit in this mask (`PCMPESTRI`, `PCMPISTRI`). The mode mask is a constant consisting of several fields that affect the instruction behaviour. It specifies: whether characters are interpreted as signed or unsigned 8-bit or 16-bit values, an aggregation operation to compare characters in different ways, a processing to be performed on the intermediate result of the operation, and a final operation to produce the output from the intermediate result. For specific details see Intel documentation at [23].

However, they have not been found useful for the $k$ mismatches problem yet due to a much higher latency compared to the SSE2 instructions that would replace them [19, 22]. This issue is further analyzed in Section A.4.1.1.

#### A.2.3.3. AVX

*Intel Advanced Vector Extensions* (AVX) was released in 2011, introducing 256-bit vector processing capability. AVX instruction set extends 128-bit SIMD instruction sets by employing a new instruction encoding scheme via a vector extension prefix (VEX). With the exception of SIMD instructions operating on MMX registers, almost all legacy 128-bit SIMD instructions have AVX equivalents that support three operand syntax. 256-bit AVX instructions employ three-operand syntax and four-operand syntax.

AVX introduces support for 256-bit wide SIMD registers (YMM0-YMM7 in operating modes that are 32-bit or less, YMM0-YMM15 in 64-bit mode). The lower 128-bits of the YMM registers are aliased to the respective 128-bit XMM registers.

AVX2 (2013) extends AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. AVX2 instructions follow the same programming model as AVX instructions.

AVX-512 was partially released in 2016 in Knights Landing processors and it has been gaining presence in subsequent families of processors. AVX-512 is a family that comprises a collection of instruction set extensions, including for example AVX-512 Foundation (AVX-512F), AVX-512 Exponential and Reciprocal instructions (AVX-512ER), AVX-512 Conflict Detection instructions (AVX-512CD), AVX-512 Prefetch instructions (AVX-512PF), among others. It introduces the following architectural enhancements:

- Support for sixteen new 512-bit SIMD registers in 64-bit mode for a total of thirty two SIMD registers: ZMM0 through ZMM31. The lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers and the lower 128-bit are aliased to the respective 128-bit XMM registers.
- Support for eight new 'opmask' registers (k0 through k7) used for conditional execution and efficient merging of destination operands.
- A new encoding prefix (referred to as EVEX) is used to support additional vector length encoding up to 512 bits.

AVX-512 instructions are composed of natural extensions to AVX and AVX2, and many new different ones. Among upgrades relevant to our problem, AVX-512F implements four new compare instructions. They use an immediate field to select between eight different comparisons and save the result to a mask register. They only support doubleword and quadword comparisons, but AVX-512 Byte and Word Instructions extension (AVX-512BW) provides the byte and word versions: `VPCMPB` and `VPCMPW` respectively.

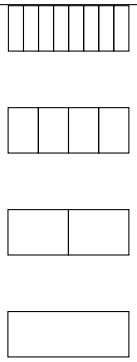
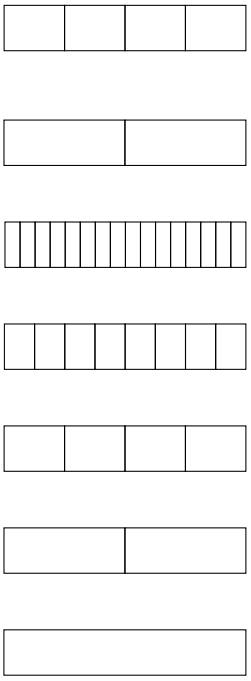
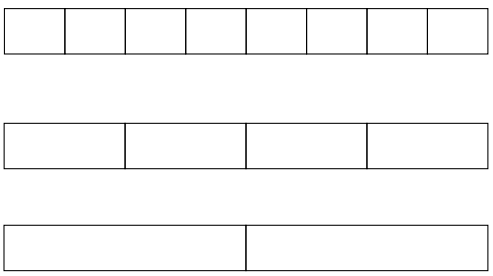
| **SIMD Extension** | **Register Layout** | **Data Type** |
|---|---|---|
| MMX<br>MMX registers |  | 8 Packed Byte Integers<br>4 Packed Word Integers<br>2 Packed Doubleword Integers<br>1 Quadword |
| SSE<br>XMM registers |  | 4 Packed Single-Precision Floating-Point Values<br>2 Packed Double-Precision Floating-Point Values<br>16 Packed Byte Integers<br>8 Packed Word Integers<br>4 Packed Doubleword Integers<br>2 Quadword Integers<br>1 Double Quadword |
| AVX<br>YMM registers |  | 8 Packed Single-Precision Floating-Point Values<br>4 Packed Double-Precision Floating-Point Values<br>2 128-bit Data |

Table A.2.2: Register layouts and available datatypes for each SIMD extension family. AVX-512 has been left out.

# Chapter A.3

# Known Non-SIMD based Solutions

This chapter describes the existing most relevant algorithms that solve the problem of string matching with up to $k$ mismatches for single and multiple patterns that do not use SIMD instructions. All of them were known at the moment of writing this thesis, with the exception of an adaptation of the Baeza-Yates–Navarro algorithm in Section A.3.2.3 to only detect mismatches.

## A.3.1. Single Pattern Matching with up to $k$ Mismatches

### A.3.1.1. Naive Algorithm

A naive algorithm works as follows: starting at the first text position, it compares character by character with the pattern until it detects more than $k$ mismatches or until it makes $m$ comparisons. If there are not more than $k$ mismatches in this text window of $m$ characters, it reports an occurrence. Then it moves on to the next text position, and repeats the comparison. It reiterates this procedure until it reaches index $n-m+1$, because a pattern occurrence cannot start at any index equal to or greater than it. Its pseudocode is shown in Algorithm 2.

This algorithm works in $O(mn)$ time in the worst case. However, its average case time complexity analysis yields an interesting result. Let us assume that individual characters in $P$ and $T$ are chosen independently and uniformly at random from the alphabet $\Sigma$. We will also make this assumption in all subsequent average time complexity analysis. Then the average time complexity of the naive algorithm is $n - m + 1$ times the expected number of comparisons it needs to perform at each text position. Let us call this expectation $E[C_x]$ where $C_x$ is the number of comparisons needed to get $x$ mismatches starting at a certain text position. If we call $C_{x_i}$ the number of comparisons needed to obtain $x$ mismatches given that we have already obtained $i - 1$ mismatches,

**Algorithm 2**: Naive
$occ \leftarrow 0$
for $i \leftarrow 0$ to $n - m$ do
  $misses \leftarrow 0$
  for $j \leftarrow 0$ to $m - 1$ do
    if $t_{i+j} \neq p_j$ then
      $misses \leftarrow misses + 1$
    if $misses > k$ then
      break
  if $misses \leq k$ then
    $occ \leftarrow occ + 1$
return $occ$

then $C_x$ can be rewritten as a sum of $C_{1_i}$ as follows

$$E[C_x] = E\left[\sum_{i=1}^{x} C_{1_i}\right] = \sum_{i=1}^{x} E[C_{1_i}] \tag{A.3.1}$$

Instantiating $x = k + 1$ in A.3.1 we then get

$$E[C_{k+1}] = \sum_{i=1}^{k+1} E[C_{1_i}] \tag{A.3.2}$$

Note that $E[C_1]$ corresponds to the expected number of comparisons needed to obtain one mismatch. Given that it is a geometric distribution, this expectation is equal to $1 + \frac{1-p}{p}$ where $p$ is the probability to get one mismatch. If we say that alphabet $\Sigma$ has size $\sigma > 1$, then

$$E[C_1] = 1 + \frac{1-p}{p} = 1 + \frac{1 - \frac{\sigma-1}{\sigma}}{\frac{\sigma-1}{\sigma}} = \frac{\sigma}{\sigma - 1}$$

Now, as each $C_{1_i}$ is independent from each other, we can replace each $E[C_{1_i}]$ in A.3.2 with the value of $E[C_1]$, obtaining

$$E[C_{k+1}] = \sum_{i=1}^{k+1} E[C_1] = \sum_{i=1}^{k+1} \frac{\sigma}{\sigma - 1} = (k+1)\frac{\sigma}{\sigma - 1}$$

Then we can affirm that $E[C_{k+1}] \in O(k)$, so the naive algorithm has an average time complexity of $O((n - m + 1)k) = O(kn)$.

### A.3.1.2. Shift-Add Algorithm

Baeza-Yates and Gonnet [3] presented Shift-Add (SA), the first bit-parallel algorithm for the $k$ mismatches problem.

It works with a bit-wise simulation of a non-deterministic finite automaton, which consists of $k+1$ rows and $m$ columns of states, as shown in an example in Figure A.3.1. Each row denotes the number of mismatches seen in the current alignment (from 0 to $k$), and each column which character of the pattern is being considered. Horizontal transitions represent matching a text character with the corresponding pattern character, and diagonal transitions represent a mismatch. This algorithm counts the number of mismatches seen per column. As there are at most $k$ mismatches, it employs counters of $\lceil \log_2(k+1) \rceil$ bits.

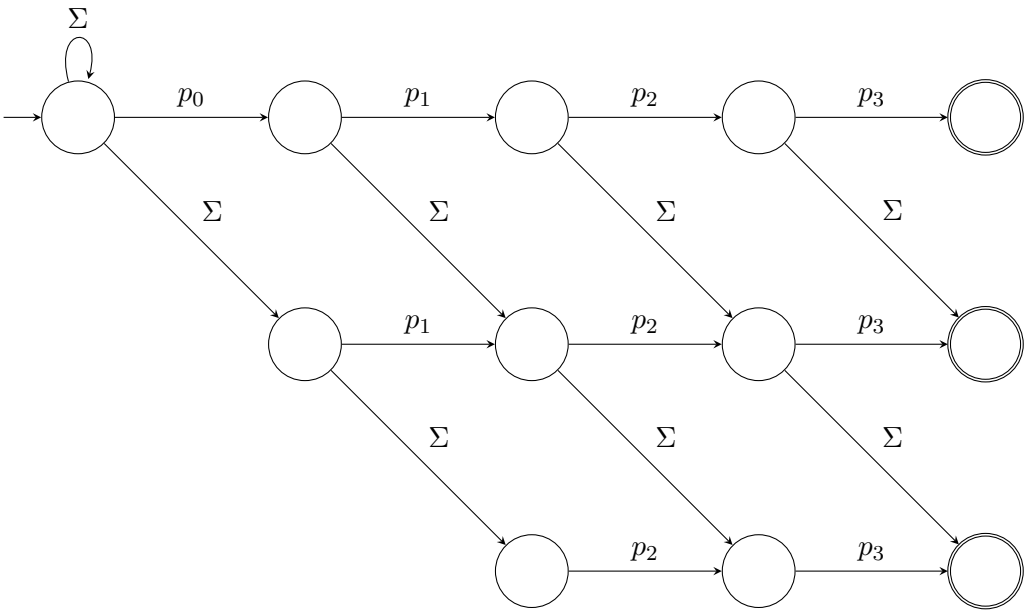



Figure A.3.1: Non-deterministic Finite Automaton used in Shift-Add for a pattern of length 4 and $k = 2$ mismatches.

In the search phase, all counters start with a value of 0. Then, the new value of the counter for column $j$ is the value of counter for column $j-1$ (or 0 if $j = 0$) plus 0 if the new character read in the text matches the $j$-th character of the pattern, or plus 1 otherwise. Whenever the counter of the last column (i.e. $m-1$) has a value less than or equal to $k$, a match is reported ending at the text position being processed.

Formally, let $D_i$ be the value of state vector that contains all $m$ counters at $i$-th text position. Let $D_i[j]$ be the $j$-th counter of the vector, $0 \leq j < m$. Then $D_i[j]$ encodes the number of mismatches in the alignment of $p_0...p_j$ with $t_{i-j}...t_i$. Consequently, $D_i[m-1] \leq k$ implies that there is an occurrence of the pattern at $t_{i-(m-1)}...t_i$ with less than or equal to $k$ mismatches.

These counters can be concatenated in a single bit-vector and added in parallel, allowing one bit between them to store possible overflow (to avoid modifying the neighboring counter). Let us call $L$ the number of bits required to represent one counter of the automaton, $L = \lceil \log_2(k+1) + 1 \rceil$. The automaton then needs $m \times L$ bits to

be simulated. Therefore we will assume that the state vector fits in a single computer register of $w$ bits, i.e. $m \times L \leq w$. In the algorithm implementation, the state vector $D_i$ is stored in a $w$-bit register which contains the $m$ counters in a little-endian fashion (i.e. $D_i[0]$ corresponds to the last column of the automaton).

In order to perform an update operation efficiently on all counters of $D_i$, Shift-Add parallelizes it as follows:

- Counter $j$ can get the value of counter $j-1$ just by shifting vector $D_i$ $L$ bits to the left.
- Counter $j$ adds 1 to its value if and only if $p_j \neq t_i$. This can be preprocessed in a vector $B$, such that $B[c]$ has its $j$-th $L$-bit field equal to $0^{L-1}1_2$ if $c \neq p_j$, and $0_2^L$ otherwise, for all $c \in \Sigma$. If $B$ contains this information for all $j = 0, ..., m-1$, then to update $D_i$ it suffices to add $B[t_i]$ to $D_i$ after shifting as described in the previous item.

However, to avoid possible overflows in counters, in the second step the overflow bit of each counter is kept in a separate vector $O$. The algorithm also cleans all garbage bits that remain to the left of vector $D$ after shifting by $L$ bits by performing a bitwise AND operation with a vector mask defined as $Vmask = 1_2^{m \times L}$. The complete search phase of Shift-Add for cases when $m \times L \leq w$ is described in Algorithm 3.

**Algorithm 3**: Shift-Add search
$occ \leftarrow 0$
$limit \leftarrow (k+1) << (m-1) \times L$
$Vmask \leftarrow 1_2^{m \times L}$
$Hmask \leftarrow (10^{L-1})_2^m$
$D \leftarrow Vmask \ \& \ \sim Hmask$
$O \leftarrow Hmask$
for $i \leftarrow 0$ to $n-1$ do
    $D \leftarrow ((D << L) + B[t_i]) \ \& \ Vmask$
    $O \leftarrow ((O << L) \mid (D \ \& \ Hmask)) \ \& \ Vmask$
    $D \leftarrow D \ \& \ \sim Hmask$
    if $(D \mid O) < limit$ then
        $occ \leftarrow occ + 1$
return $occ$

Shift-Add works in linear time for short patterns $m \leq w/\lceil \log_2(k+1) + 1 \rceil$ where $w$ is the width of the computer registers. Such patterns allow the automaton to fit in a single computer register, which makes it possible to shift and add the counters in constant time. However, for arbitrary $m$ and $k$, its time complexity is $O(n\lceil m \log(k)/b \rceil)$. SA is still competitive for short patterns and large $k$ [17]. Ďurian et al. [9] presented two variations of Shift-Add: Tuned Shift-Add and Two-way Shift-Add. They are both described next.

**Tuned Shift-Add**

Tuned Shift-Add (TuSA) improves Shift-Add performance by manipulating overflow bits in the same state vector $D$ instead of a separate vector $O$, such that $O = D \ \& \ (10^{L-1})_2^m$. To achieve this, TuSA initializes the counters so that they flip their overflow bit from 0 to 1 after exactly $k+1$ additions, i.e. it initializes them with a value of $init = (10_2^{L-1}) - (k+1)$. It prevents from performing more than $k+1$ additions to a counter by using its overflow bit as a mask by employing $(B[t_i] \ \& \ \sim (D << 1))$ as addition vector instead of just $B[t_i]$. Occurrences are then detected simply by checking the overflow bit of $D[m-1]$. TuSA search phase is written in pseudocode in Algorithm 4.

**Algorithm 4**: Tuned Shift-Add search

```
occ ← 0
init ← (10_2^{L-1}) − (k + 1)
D ← ∼ 0
for i ← 0 to n − 1 do
    D ← ((D << L) | init) + (B[t_i] & ∼ (D << 1))
    if (D & (1 << (m × L − 1))) = 0 then
        occ ← occ + 1
return occ
```

Although these changes introduced by TuSA do not modify the theoretical time complexity of Shift-Add, practical experiments as those conducted in Section A.6.1 demonstrate a significant speed-up compared to the original SA.

**Two-way Shift-Add**

Two-way Shift-Add (TwSA) extends TuSA by reading characters simultaneously from both ends of the window. It allows the CPU to make use of instruction-level parallelism (ILP), overlapping the execution of multiple instructions.

First, it divides the text into $\lceil n/m \rceil$ overlapping windows of size $2m-1$. The substring $t_{i-(m-1)}...t_{i+(m-1)}$ corresponds to a window with center at text position $i$. For each window with center at $i = m, 2m, ..., \lceil n/m \rceil m$, TwSA initializes the counters of $D$ with value $10_2^{L-1} - (k+1)$ as in TuSA, and then updates $D$ with the mismatches caused by $t_i$. Then, TwSA enters the scanning stage which adds the mismatch vectors to the state vector $D$:

$$\begin{aligned} D \leftarrow D + ((\sim D >> (L-1)) \ \& \ (B[t_{i-j}] << j \times L)) \\ + ((\sim D >> (L-1)) \ \& \ (B[t_{i+j}] >> j \times L)) \end{aligned} \tag{A.3.3}$$

for $j = 1, 2, ..., m-1$, until all states have over $k$ mismatches (i.e. $\sim D \ \& \ Hmask = 0$, where $Hmask = (10_2^{L-1})_2^m$) or the whole window has been processed, in which case non-zero bits in $\sim D \ \& \ Hmask$ correspond to occurrences of the pattern with at most $k$ mismatches. Notice that this update formula keeps $D$ at a fixed position and shifts the

values of $B$ instead. Each window processing gets all occurrences of the pattern that end between $t_i$ and $t_{i+m-1}$. Its pseudocode of search phase is presented in Algorithm 5.

Given that TwSA stops looking for occurrences in the current alignment once all counters register more than $k$ mismatches, its average case time complexity improves respect to TuSA and SA. Under the same assumption as in SA analysis that $O(\lceil m \log(k)/b \rceil)$ is constant (i.e. the automaton fits into a single machine register), TwSA performs $O(k + log_\sigma m)$ expected work in each window (the proof can be found in [19]). As it processes $O(m/n)$ windows, it yields a total expected time complexity of $O(n(k + log_\sigma m)/m)$.

**Algorithm 5**: Two-way Shift-Add search
$occ \leftarrow 0$
$init \leftarrow ((10_2^{L-1}) - (k+1))_2^m$
$Hmask \leftarrow (10_2^{L-1})_2^m$
$D \leftarrow \sim 0$
for $i \leftarrow m$ to $n - (m-1) - 1$ step $m$ do
  $D \leftarrow init + B[t_i]$
  $j \leftarrow 1$
  while $j < m$ and $(\sim D \ \& \ Hmask) \neq 0$ do
    $D \leftarrow D + ((\sim D >> (L-1)) \ \& \ (B[t_{i-j}] << j \times L))$
      $+ ((\sim D >> (L-1)) \ \& \ (B[t_{i+j}] >> j \times L))$
    $j \leftarrow j + 1$
  if $(\sim D \ \& \ Hmask) \neq 0$ then
    Increment $occ$ by number of one bits in $\sim D \ \& \ Hmask$
return $occ$

### A.3.1.3. Enhanced FAAST

Approximate Boyer–Moore (ABM) by Tarhio and Ukkonen [34] is a generalization of the Boyer–Moore–Horspool algorithm [20] to approximate string matching. In ABM, shifting is based on a $q$-gram (substring of $q$ contiguous characters), with $q = k+1$. Liu et al. [28] tuned ABM for small alphabets. Their algorithm applies wider $q$-grams and is called FAAST.

Salmela et al. [31] designed an Enhanced version of FAAST (EF). For each $q$-gram, the preprocessing phase of EF computes the Hamming distance with the end of all prefixes of the pattern. With this information, a shift table $S_q$ can be constructed: given a $q$-gram fingerprint $f$, $S_q[f]$ returns $m - q - good_f$ where $good_f$ is the position of the first character of the rightmost $q$-gram of the pattern that has at most $k$ mismatches with a $q$-gram corresponding to $f$. If there are collisions and many $q$-grams have the same fingerprint $f$, then $S_q[f]$ stores the minimum value for all of them. If there are no pattern $q$-grams with at most $k$ mismatches with a $q$-gram corresponding to $f$, then $S_q[f] = m - q + 1$, which corresponds to the largest possible shift. This table is used to compute the jump to the next text position to be analyzed.

$M$ is another precomputed table which gives the Hamming distance of a $q$-gram against the last $q$-gram of the pattern. In the case of the DNA alphabet, both tables are accessed with the fingerprint $f \leftarrow \sum_{i=0}^{q-1} map(t_{s-i}) * 4^i$, where the function $map$ maps each DNA character to an integer in $\{0, 1, 2, 3\}$. Whenever $M[f] > k$ holds, the algorithm shifts forward without processing the alignment window further. In this way, the table $M$ is used as a filtration mechanism.

The complete algorithm for EF search phase is written in pseudocode in Algorithm 6.

**Algorithm 6**: EF (search)

```
s ← m − 1
while s < n do
    f ← Σ_{i=0}^{q−1} map(t_{s−i}) * 4^i
    if M[f] ≤ k then
        c ← M[f]
        for i ← 1 to m − q do
            if t_{s−q−i+1} ≠ p_{m−q−i} then
                c ← c + 1
                if c > k then
                    break
        if c ≤ k then
            occ ← occ + 1
    s ← s + S_q[f]
return occ
```

### A.3.1.4. Baeza-Yates–Perleberg Algorithm

The Baeza-Yates–Perleberg algorithm (BYP) [5] is based on a partitioning scheme, which makes use of the following lemma:

**Lemma A.3.1.1.** *Given a partition of the pattern into $k+1$ subpatterns, and a pattern occurrence with at most $k$ mismatches, at least one of these $k+1$ subpatterns occurs exactly in it.*

*Proof.* Let us assume the opposite, i.e. each member of the partition occurs with at least 1 mismatch. Then the whole pattern would occur with at least $k+1$ mismatches, which leads to a contradiction. □

In the preprocessing phase, BYP splits the pattern into $k+1$ subpatterns: $k+1-(m \mod (k+1))$ of length $l$ and $m \mod (k+1)$ of length $l+1$, where $l = \lfloor m/(k+1) \rfloor$. For example, if the pattern is $abcdefgh$ and $k = 2$, a possible partition is $\{ab, cde, fgh\}$.

By Lemma A.3.1.1, it is known that given an approximate occurrence of the pattern with at most $k$ mismatches, at least one of these subpatterns occurs exactly. So then it performs a multiple exact string matching search of them in the text with an algorithm

based on Boyer–Moore–Sunday algorithm [32]. Whenever one of them is found at position $i$, it checks if there is an approximate pattern match between $i-(m-l)$ and $i+m$ with Ukkonen's dynamic algorithm [35].

In a worst case scenario, BYP finds subpatterns at every text position, which forces it to apply the approximate pattern matching algorithm over all the text. Then it would perform at most as much work as the worst case for the exact string matching algorithm plus the work needed in the worst case scenario of the approximate pattern matching algorithm over all the text. The worst case time complexity for the Boyer–Moore–Sunday algorithm is $O(n+m)$, and the worst case complexity for Ukkonen's algorithm is $O(kn)$. Then, BYP performs $O(n+m+kn) = O(kn)$ operations in its worst case. However, it achieves expected linear running time for $k \in O(m/\log m)$, as proved in [5].

## A.3.2. Multiple Pattern Matching with up to $k$ Mismatches

### A.3.2.1. Naive Algorithm

A naive algorithm for the multiple pattern variation of the problem is the natural extension of the naive algorithm for the single pattern case presented in Section A.3.1.1. It suffices to perform $r$ times the character-by-character comparisons at each text position (once for each pattern). This procedure yields an algorithm which runs in $O(rmn)$ time in the worst case, and $O(rkn)$ time on average.

### A.3.2.2. Muth–Manber Algorithm

The first non-naive algorithm for multiple pattern matching with up to $k$ mismatches was presented by Muth and Manber [30] for $k = 1$. We will call it MM. In the preprocessing it uses an ad-hoc function to obtain hash values of all the strings of length $l-1$ that result from taking each character out of every prefix of size $l$ of each pattern, where $l$ is chosen empirically depending on problem parameters. Then it proceeds in the same way for each text window of $l$ characters, and naively verifies each hash coincidence. They tested which value for $l$ performed better and used $l = 6$ characters in [30]. MM has an average preprocessing time of $O(lr)$ and an average search time complexity of $O(ln)$. Its average total time is then $O(l(r+n))$, which makes it suitable for large sets of patterns.

It uses *two-level hashing* for optimization, which consists of having two hash tables. The first one is a bitmap, such that each bit is set to 1 if and only if something is mapped to the corresponding entry in the second hash table, which actually maps hash values to its corresponding patterns. If the bit in the first table is 0, it knows that the string is not in the hash table. At first sight, it would seem that this technique only adds extra work to the algorithm. However, it speeds up unsuccessful queries because only the first table is queried in these cases. Unsuccessful queries are far more common than successful ones in a randomly distributed text. If we call $w$ the size in bits of a

computer register that is big enough to store an index to the patterns corresponding to a given fingerprint, then the bitmap is $w$ times smaller than the original table, which makes it more cache-friendly, hence the speedup.

In the case when $k > 1$, it can be modified to compute all the strings that result from taking $k$ characters out of each pattern prefix and text window. Given that prefixes and windows have length $l$, it needs to generate $\frac{l!}{k!(l-k)!}$ strings for each text position and each pattern. Its complexity then limits its usage to only very small $k$ in practice. At first sight, when $k$ approaches $l$ one would think that its complexity is comparable to cases when $k$ is small. However, the filtration mechanism would start to fail reporting a large number of possible occurrences because of hash coincidences. For example, if $k = l - 1$ then every character that appears in at least one pattern prefix will be considered as a potential occurrence when found in the text. As these possible occurrences need to be naively verified, the algorithm would degenerate into the naive algorithm of Section A.3.2.1. Further research into extending this algorithm to work with a general $k$ is out of the scope of this thesis.

#### A.3.2.3. Baeza-Yates–Navarro Algorithm (Variation)

The Baeza-Yates–Navarro algorithm (BYN) [4] addresses the problem of searching occurrences of patterns in a text within an *edit distance*[1] of at most $k$. It is based on a partition scheme, similar to the one used in the BYP algorithm for a single pattern (see Section A.3.1.4). We will first explain the original algorithm and then the changes made so it works under Hamming distance only.

This algorithm splits every pattern into subpatterns of length $l = \lfloor m/(k+1) \rfloor$ and performs an exact multiple subpattern search (see Lemma A.3.1.1) using an extension of the Sunday algorithm [32]. Whenever there is a subpattern occurrence, it checks for the entire pattern with an approximate single pattern matching algorithm. This process is done by applying a *hierarchical piece verification*. Instead of checking for the entire pattern in the candidate area, it checks for the concatenation of two pieces containing the one that matched. If it matches, then it checks the concatenation of four pieces, and so on.

For the approximate matching phase, BYN employs a Non-deterministic Finite Automaton which exploits bit-parallelism. It is similar to the one used in the Shift-Add algorithm (see Section A.3.1.2), except for the addition of two new kinds of transitions:

– Dashed diagonal arrows corresponding to *deletions* in the pattern, since, as they are empty transitions, they advance in the pattern but not in the text.

– Vertical arrows representing *insertions* in the text, since they advance in the text but not in the pattern.

An example of this automaton is shown in Figure A.3.2. If the number of errors of the occurrences is not important, the states of the last full diagonal can be considered as final states due to the empty transitions. Furthermore, we can omit the states below the first full diagonal because they are always active. Thus, the only interesting states

[1] Formal definition in Section A.2.1.

are those belonging to the full diagonals. This observation allows to efficiently simulate the automaton representing it only by its full diagonals.

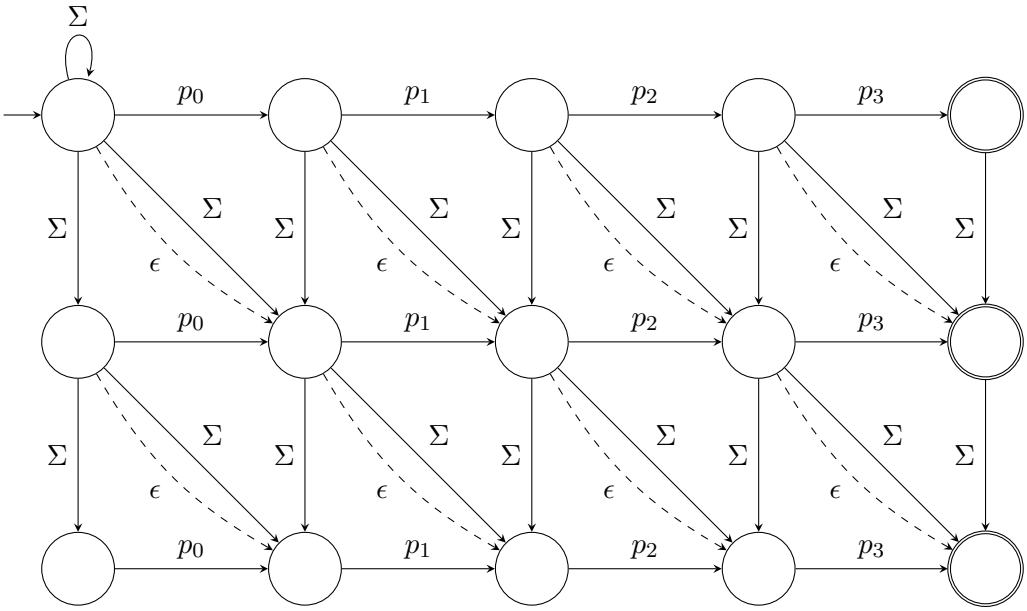



Figure A.3.2: Non-deterministic Finite Automaton used in Baeza-Yates–Navarro algorithm for a pattern of length 4 and $k = 2$ mismatches.

It is worth remarking the decomposition of searching multiple approximate occurrences into two more simple problems: a multiple exact search and a single approximate search in a reduced portion of the text.

Since this algorithm works under *edit distance*, it has been necessary to modify it to only detect mismatches. If we delete the unnecessary transitions from its automaton (i.e. the ones corresponding to insertions and deletions), we obtain the same automaton used in Shift-Add (example shown in figure A.3.1). We lose the ability to efficiently simulate it by diagonals, but it can be simulated by columns, as performed in Shift-Add algorithm. In this way, we modified the automaton, preserving as much of the original algorithm as was possible.

This algorithm adaptation could be further studied and improved, but it is out of the scope of the thesis.

### A.3.2.4. Fredriksson–Navarro Algorithm

The fastest algorithm to date in the case of low difference ratio is Fredriksson and Navarro's filtration algorithm [15]. It also works for *edit* distance with minor modifications.

In the preprocessing phase, it builds a table $D : \Sigma^q \to \mathbb{N}$ telling, for each possible $q$-gram, the minimum number of mismatches necessary to match a $q$-gram inside any of the patterns. Formally, given a $q$-gram $s$, and a function $dist(s_1, s_2)$ which gives the

Hamming distance between strings of equal length $s_1$ and $s_2$, then the table $D$ is defined as:

$$D(s) = \min_{\substack{0 \le j < r \\ 0 \le i \le len(p^j) - q}} \{dist(s, p_i^j ... p_{i+q-1}^j)\}$$

In the searching phase, the algorithm places a window over the text, in which $q$-grams are read in a backwards order. It accumulates the sum of the values in table $D$ corresponding to each read $q$-gram. Whenever an occurrence is impossible because of detecting more than $k$ mismatches with all patterns, the window is shifted to avoid analysing a window containing the read $q$-grams (i.e. it is shifted to start one character to the right of the beginning of the last $q$-gram read).

Several variations and improvements to this basic algorithm are proposed and tested such as hierarchical verification, packing counters to take advantage of bit-parallelism, and a method to speed up its preprocessing phase. For more details, see [15].

This algorithm is theoretically optimal in the average case. Its average search time complexity is $O((k + \log_\sigma(rm))n/m)$ for $\alpha < 1/2 - O(1/\sqrt{\sigma})$, where $\alpha = k/m$ is the difference ratio. However, its preprocessing time and space complexity of $O(rm\sigma^q)$ limits the size of $q$-grams that can be used in practice. For example, in DNA $|\sigma| = 4$, but for English ASCII text $|\sigma| = 96$ places a strong limitation in $q$. This is one of the reasons why FN works better in DNA and Proteins than in English texts. In [15] they use $q = 2$ for English and $q = 8$ for DNA, whereas the theoretically optimal values for $r \in \{1, 16, 64, 256\}$ are $q \in \{3, 4, 5\}$ and $q \in \{12, ..., 20\}$ respectively. This practical restriction of being unable to choose an optimal $q$ impacts negatively in its performance.

# Chapter A.4

# Known SIMD-based Solutions

In this chapter we analyze the most relevant SIMD-based algorithms that solve the problem of string matching with up to $k$ mismatches, as well as an algorithm that solves the exact multiple string matching problem because it was useful to build some algorithms presented in Chapter A.5. We also explain two common techniques that utilize SIMD instructions employed by these algorithms. All of the solutions included in this chapter have already been published before.

## A.4.1. SIMD Techniques

In this section we describe two ways of taking advantage of SIMD computation that have already been found useful in different algorithms for string matching.

### A.4.1.1. Counting of Mismatches

Counting mismatches is a usual operation in approximate string matching. It can be done with the intrinsic functions *simd-cmpeq*$(x, y)$ and *simd-popcount*$(x)$ explained below. In practice, we also need the function *simd-load*$(x)$, which is an intrinsic function of the compiler formally defined as

```
_mm_loadu_si128(x)
```

This function loads 16 bytes from the address $x$ to a SIMD register given as the left-hand side of an assignment statement. Function *simd-cmpeq*$(x, y)$ is formally

```
_mm_movemask_epi8(_mm_cmpeq_epi8(__m128i x, __m128i y))
```

The function `_mm_cmpeq_epi8` compares 16 bytes in $x$ and $y$ bytewise for equality and stores the result. Function `_mm_movemask_epi8` creates a bitvector from the most significant bit of each byte of the parameter. Function *simd-popcount*$(x)$ counts the number of on bits in $x$ and is formally

```
_mm_popcnt_u32(x)
```

The *simd-cmpeq*$(x, y)$ function, therefore, makes it possible to compare up to 16 characters at the same time. It can be extended to 32 characters by using AVX and AVX2 analogous instructions: `_mm256_loadu_si256`, `_mm256_movemask_epi8` and `_mm256_cmpeq_epi8`. The result is a bitvector of the pairwise comparisons. Lastly, a population count operation on the result tells the number of matching characters. An example for this technique is written in pseudocode in Algorithm 7.

**Algorithm 7**: Counting of mismatches between strings $s_1$ and $s_2$
$x \leftarrow$ *simd-load*$(s_1)$
$y \leftarrow$ *simd-load*$(s_2)$
$mask \leftarrow$ *simd-cmpeq*$(x, y)$
return *simd-popcount*$(mask)$

All instrinsic functions employed to calculate the bitmask stored in *mask* in Algorithm 7 are translated into instructions from SSE2 SIMD extension set, as shown in Table A.4.1. The table also shows latencies for each of these instructions for an Intel Broadwell microarchitecture [22], like the one used in Chapter A.6. Latency can be defined as the number of clock cycles that are required for the execution core to complete the execution of all of the micro-ops that form an instruction.

| Intrinsic function | Assembly Instruction | Extension Set | Latency |
|---|---|---|---|
| `_mm_loadu_si128` | `MOVDQU` | SSE2 | 1 |
| `_mm_cmpeq_epi8` | `PCMPEQB` | SSE2 | 1 |
| `_mm_movemask_epi8` | `PMOVMSKB` | SSE2 | 2 |
| `_mm_popcnt_u32` | `POPCNT` | POPCNT[1] | 3 |

Table A.4.1: Intrinsic functions used by Algorithm 7 and their corresponding assembly instruction, extension set and latency measured in CPU cycles for an Intel Broadwell microarchitecture.

We could define a new *simd-cmpeq* function so as to use an SSE4.2 STTNI instruction as follows:

```
_mm_cvtsi128_si32(_mm_cmpestrm(x, 16, y, 16, _SIDD_UBYTE_OPS |
_SIDD_CMP_EQUAL_EACH | _SIDD_BIT_MASK | _SIDD_NEGATIVE_POLARITY))
```

The function `_mm_cmpestrm` emits only one instruction: `PCMPESTRM`. Arguments used to call `_mm_cmpestrm` indicate the mode mask and that the length of strings $x$ and $y$ are 16 bytes (for more details see Section A.2.3.2). Then `_mm_cvtsi128_si32` truncates the return value of `_mm_cmpestrm`, which is a 128-bit mask, to the least significant 32 bits using `MOVD` instruction. This function is needed because *simd-popcount* cannot

[1]Population count instructions operate on integers rather than SSE registers. They are not SIMD instructions but they are counted as a separate extension because of being implemented by hardware. They were released by the same time as SSE4.

take a 128-bit XMM register as argument. However, these two instructions have latencies of 10 and 1 cycles respectively for an Intel Broadwell microarchitecture. Then it would increase total latency of Algorithm 7 by 8 cycles because original *simd-cmpeq* uses `PCMPEQB` and `PMOVMSKB` instead of them, which have a latency of 1 and 2 cycles respectively, making it less efficient. This effect has been further studied and tested in [19], with results matching this analysis.

#### A.4.1.2. CRC as a Fingerprint

There are many filtration methods for approximate string matching. Those methods contain two phases which are usually interleaved. The filtration phase selects match candidates and the checking phase verifies them. The former often entails the calculation of a fingerprint or a hash value from a $q$-gram, with which precomputed tables are accessed. Selecting a function to perform such calculation involves a tradeoff between the amount of collisions generated and its execution time [30]. A SIMD choice for this function can be the *simd-crc*$(x)$ intrinsic function. A similar function was first used by Faro and Külekci [11, 12] in exact string matching (see the MEPSM algorithm in Section A.4.3).

The function *simd-crc*$(x)$ returns a $b$-bit value by first calculating a 32-bit cyclic redundancy checksum (CRC) of a 64-bit value, and then taking the $b$ least significant bits of the CRC. It is formally

```
_mm_crc32_u64(x) & mask
```

where $x$ is a 64-bit integer, *mask* is $2^b - 1$, and '&' is bit-parallel AND. The best value of $b$ depends on the problem parameters. The family of functions `_mm_crc32_uX(y)` was introduced in SSE4.2 (see Section A.2.3.2), where $X$ can be 8, 16, 32 or 64, and refers to the size of $y$ in bits.

### A.4.2. Single Pattern Matching with up to $k$ Mismatches

This section presents the two most relevant algorithms known to the date of writing this thesis for approximate string matching that use SIMD instructions. Both have been developed in [14] with positive results.

#### A.4.2.1. Approximate Naive improved with SIMD

A straightforward approach to string matching with at most $k$ mismatches is the naive counting of mismatches, as described in Section A.3.1.1. A SIMD improved algorithm of this kind called ANS (see Algorithm 8) is presented in [14] for patterns up to 32 bytes long. It counts the character matches with $P$ starting from the $n - m + 1$ first positions of the text by using SIMD instructions.

They also presented a way to make ANS faster when $m \leq 16$ by preprocessing the condition *simd-popcount*$(t) \geq m - k$ to a boolean array $D$ for each vector $t$ of 16 bits.

**Algorithm 8**: ANS
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
for $i \leftarrow 0$ to $n - m$ do
    $y \leftarrow$ *simd-load*$(t_i \cdots t_{i+m-1})$
    $t \leftarrow$ *simd-cmpeq*$(x, y)$
    if *simd-popcount*$(t) \geq m - k$ then
        $occ \leftarrow occ + 1$
return *occ*

Then the last *if* statement of ANS is changed to

$$\text{if } D[t] \text{ then } occ \leftarrow occ + 1$$

For longer patterns, $16 < m \leq 32$, the last line is

$$\text{if } D[t \;\&\; mask] \text{ then if } \textit{simd-popcount}(t) \geq m - k \text{ then } occ \leftarrow occ + 1$$

where *mask* is $2^{16} - 1$. In other words, the first 16 characters of the pattern are tested first. This variation is called ANS2. Pseudocode for its preprocessing phase is shown in Algorithm 9, and psecudocode for its search phase is shown in Algorithms 10 and 11 for pattern sizes of $m \leq 16$ and $m > 16$ respectively.

**Algorithm 9**: ANS2 Preprocessing
for $i \leftarrow 0$ to $2^{16} - 1$ do
    if *simd-popcount*$(t) \geq m - k$ then
        $D[i] \leftarrow 1$
    else
        $D[i] \leftarrow 0$

**Algorithm 10**: ANS2 Search for $m \leq 16$
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
for $i \leftarrow 0$ to $n - m$ do
    $y \leftarrow$ *simd-load*$(t_i \cdots t_{i+m-1})$
    $t \leftarrow$ *simd-cmpeq*$(x, y)$
    if $D[t]$ then
        $occ \leftarrow occ + 1$
return *occ*

Note that the preprocessing time would grow exponentially if $D$ were extended for wider vectors. However, $D$ could be changed to preprocess shorter or longer patterns (different from 16 characters) according to the machine used to run an implementation of the algorithm. Different CPU cache and RAM memory sizes and speeds could yield different results.

**Algorithm 11**: ANS2 Search for $m > 16$

```
x ← simd-load(p_0 ··· p_{m−1})
mask ← 2^16 − 1
for i ← 0 to n − m do
    y ← simd-load(t_i ··· t_{i+m−1})
    t ← simd-cmpeq(x, y)
    if D[t & mask] then
        if simd-popcount(t) ≥ m − k then
            occ ← occ + 1
return occ
```

It is also important to remark that the speed of ANS does not depend on $k$ for values of $m$ that fit in a *simd-load* (i.e. 32 characters long). On the other hand, ANS2 performance degrades when $k$ approaches $m$ for $m > 16$ because it is more likely to use *simd-popcount*() function.

Furthermore, its speed also decreases for $m \leq 16$ when $k$ is rather high compared to $m$. For example, for $m = 16$ its execution time peaks when $k$ is around 11 in DNA alphabet, as shown in Figure A.6.5. The main reason for this decrement in performance is an increment in the number of CPU conditional branch mispredictions, as analyzed in Section A.6.1.

For the following complexity analysis, let us call $b$ the number of characters that fit in a single SIMD register used by the SIMD instructions employed in them. For example, $b$ is 16 and 32 for SSE4.2 and AVX2 instructions sets respectively. If ANS algorithm and its variations were extended to work with a general $m$, they would have a worst case time complexity in $O(\lceil m/b \rceil n)$. Performing the same analysis as in Section A.3.1.1, we obtain an average time complexity in $O(\lceil k/b \rceil n)$. Note that these bounds become 'linear' for cases when $m \leq b$ and $k \leq b$ respectively. It is worth remarking its improvement compared to the Naive algorithm from Section A.3.1.1.

### A.4.2.2. EF enhanced with SIMD

An improvement on the EF algorithm with SIMD computation (EFS) was presented in [14]. EF contains a filtration and a checking phase (see Section A.3.1.3). The checking method was replaced with ANS2 (see Section A.4.2.1), and the fingerprint computation of the filtration method was replaced with the already mentioned CRC fingerprint technique.

Algorithm 12 is the pseudocode of EFS for $m \leq 16$. The array $D$ is computed in the same way as for ANS2. For longer patterns, $16 < m \leq 32$, the required change is the same as in the case of ANS2.

**Algorithm 12**: EFS search for $m \leq 16$

```
x ← simd-load(p_0 ··· p_{m-1})
s ← m − 1
while s < n do
    f ← simd-crc(t_{s-q+1} ··· t_s)
    if M[f] ≤ k then
        y ← simd-load(t_{s-m+1} ··· t_s)
        t ← simd-cmpeq(x, y)
        if D[t] then
            occ ← occ + 1
    s ← s + S_q[f]
return occ
```

### A.4.3. Multiple Pattern Matching

In this section we present an algorithm that solves the *exact* string matching problem. Although it does not address the main problem analyzed in this thesis, its filtering technique based on SIMD computation has been useful for developing SIMD-based algorithms for approximate string matching as described later in Sections A.5.1.2 and A.5.2. When this thesis was planned, no relevant algorithms for solving the multiple string matching problem under Hamming distance using SIMD instructions have been found by the authors.

#### Multiple Exact Packed String Matching

The Multiple Exact Packed String Matching algorithm (MEPSM) [12] is based on a filter mechanism. It first computes the CRC fingerprint of some of the first $q$-grams of each pattern with SIMD instructions like those shown in Section A.4.1.2. $q$ is a value smaller than or equal to $m$ which is calculated as shown in Algorithm 13. The information about which $q$-gram the fingerprint belongs to is stored in a table $H$. Afterwards, the algorithm searches coincident fingerprints of $q$-grams in the text. Whenever a pattern occurrence candidate is found, it is naively verified and reported in case of a match. After each $q$-gram analysis, the algorithm shifts forwards by a fixed number of characters: $shift = (\lfloor m/q \rfloor - 1) \times q$.

It needs to precompute as many of the first $q$-grams from each pattern as the number of characters it shifts forwards after each text $q$-gram analysis. The reason for this is that any exact occurrence of a pattern $p^x$ in the text will contain all of its $q$-grams, so by shifting by $shift$ characters it is potentially omitting $shift - 1$ occurrences of $q$-grams of pattern $p^x$. This is fixed by precomputing one pattern $q$-gram more than the ignored $shift - 1$ text $q$-grams in order to find every occurrence of pattern $p^x$.

Pseudocode for its preprocessing and searching phases is shown in Algorithms 13 and 14 respectively.

MEPSM is designed to be effective on sets of short patterns, where the upper limit for the length of the shortest pattern of the set is 32 ($m' \leq 32$). It runs in $O(nm)$ worst case time complexity and uses $O(rm' + 2^b)$ additional space, where $b$ is the number of bits needed to store a single fingerprint value ($b \leq 16$ in [12]). Results obtained experimentally show that this algorithm is competitive for pattern lengths between 16 and 32 bytes.

**Algorithm 13**: MEPSM Preprocess
$q \leftarrow \min(8,\ 2^{\lfloor \log_2(m) \rfloor - 1})$
$shift \leftarrow (\lfloor m/q \rfloor - 1) * q$
for $f \leftarrow 0$ to $2^{16} - 1$ do
    $H[f] \leftarrow \langle\rangle$
for $pat \leftarrow 0$ to $r - 1$ do
    for $j \leftarrow 0$ to $shift - 1$ do
        $f \leftarrow simd\text{-}crc(p_j^{pat}\ ...\ p_{j+q-1}^{pat})$
        $H[f] \leftarrow \langle (pat, j) \rangle +\!\!+ H[f]$

**Algorithm 14**: MEPSM Search
$occ \leftarrow 0$
for $i \leftarrow shift$ to $n - q$ step $shift$ do
    $f \leftarrow simd\text{-}crc(t_i\ ...\ t_{i+q-1})$
    forall $(pat, j)$ in $H[f]$ do
        if $i - j + m - 1 < n$ and $t_{i-j}\ ...\ t_{i-j+m-1} = p^{pat}$ then
            $occ \leftarrow occ + 1$
return $occ$

# Chapter A.5

# New SIMD-based Algorithms

In this chapter we describe three new algorithms developed in this thesis: two are designed for approximate single pattern matching and the third one for the multiple pattern variation. Two of them make use of SIMD computation by including two other SIMD-based algorithms: MEPSM (Section A.4.3) with some modifications and ANS2/ANS2b (Sections A.4.2.1 and A.5.1.1). The algorithm that supports multiple pattern matching is a straightforward but fruitful extension of one of the new algorithms presented for the single pattern case.

## A.5.1. Single Pattern Matching with up to $k$ Mismatches

### A.5.1.1. Approximate Naive improved with SIMD (ANS2b)

We present a variation of ANS2 algorithm. As stated in Section A.4.2.1, ANS2 performance strongly drops when $k$ rises in cases when $m \leq 16$ due to misses in the branch predictor. A way to avoid this dependency on $k$ would be to change the last line of ANS2 algorithm (see Algorithm 10)

$$\text{if } D[t] \text{ then } occ \leftarrow occ + 1$$

into

$$occ \leftarrow occ + D[t]$$

where $D$ is the boolean array used in ANS2, storing $true$ values as number ones and $false$ values as zeroes. This modification would force it to always perform an addition without a conditional statement. It would save time compared to original ANS2 when there are many occurrences, which is more likely to happen when the difference ratio is high and the alphabet size is small. Let us call this variation ANS2b. Pseudocode for its search phase when $m \leq 16$ is written in Algorithm 15. ANS2b is the same as ANS2 for $m > 16$, so its pseudocode for this case is omitted. ANS2b resulted faster than ANS2 in all of our experiments in Section A.6.1.

ANS2b has got the same average search time complexity as ANS and ANS2: $O(\lceil k/b \rceil n)$, where $b$ is the number of characters that fit in a single SIMD register used by the SIMD instructions employed in the algorithm.

**Algorithm 15**: ANS2b Search for $m \leq 16$
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
for $i \leftarrow 0$ to $n - m$ do
    $y \leftarrow$ *simd-load*$(t_i \cdots t_{i+m-1})$
    $t \leftarrow$ *simd-cmpeq*$(x, y)$
    $occ \leftarrow occ + D[t]$
return $occ$

### A.5.1.2. Baeza-Yates–Perleberg enhanced with SIMD

We present an algorithm based on the Baeza-Yates–Perleberg algorithm (BYP, Section A.3.1.4), enhanced with SIMD computation. Original BYP looks for *exact* occurrences of $k + 1$ subpatterns of the pattern in the text as described in Section A.3.1.4: $k+1-(m \mod (k+1))$ of length $l$ and $m \mod (k+1)$ of length $l+1$, where $l = \lfloor m/(k+1) \rfloor$. To achieve this, we employed a tuned version of the MEPSM algorithm for exact multiple string matching (described in Section A.4.3). MEPSM reports subpattern occurences, which are later verified by ANS2 (see Section A.4.2.1). Let us call the total algorithm BYPS.

We tuned MEPSM by setting the $q$-grams as large as possible, that is $q = \min(l, 8)$. The maximum value for $q$ is 8 because we need to compute the fingerprint of $q$-grams with *simd-crc*$(x)$. To do so, it uses a SIMD instruction as described in Section A.4.1.2, which means it can take up to a 64-bit value $x$ as input, i.e. 8 bytes. This causes fewer fingerprint collisions, but on the other hand, larger $q$ reduces shifts between alignments ($shift = l - q + 1$). However, this trade-off showed to be really satisfactory in practice, especially in the case of small subpatterns (for experiments results see Section A.6.1). This tuning is then employed in combinations of $m$ and $k$ that make the length of subpatterns searched by MEPSM smaller than 8 characters (i.e. $l < 8$). On the other hand, the minimum value admitted for $q$ (and thus for $l$) is 4 because the number of collisions while hashing would quickly rise otherwise, critically lowering its performance.

We also present a variation of BYPS where we use the ANS2b algorithm for searching approximate occurrences of the pattern in the neighborhood of exact subpattern matches. Given ANS2b superiority over ANS2, it would be logical to obtain an improvement in the BYPS algorithm as well. In this algorithm we also replaced the naive exact subpattern check done by MEPSM after finding a $q$-gram matching fingerprint in the text with a SIMD-based comparison function *simd-memcmp*, which employs function *simd-cmpeq* defined in Section A.4.1.1. Function *simd-memcmp* is shown in Algorithm 16 and it works for strings of equal size $|s|$ up to 32 characters[1] using AVX2 instructions in $O(\lceil |s|/b \rceil)$ time, where $b$ is the number of characters that fit in a SIMD register. We call this version BYPSb.

The preprocess and search phases of BYPSb for cases when $l < 8$ are shown in pseudocode in Algorithms 17 and 18 respectively. For cases when $l \geq 8$ the original

[1] Its extension to general sizes is straightforward.

**Algorithm 16**: $simd\text{-}memcmp(s_1, s_2)$ where $|s_1| = |s_2|$ and $|s_1| \leq 32$
$eqmask \leftarrow (1 << |s_1|) - 1$
$x \leftarrow simd\text{-}load(s_1)$
$y \leftarrow simd\text{-}load(s_2)$
$mask \leftarrow simd\text{-}cmpeq(x, y)$
return $(eqmask \ \& \ mask) = eqmask$

MEPSM calculation of $q$ is used. Symbols $\langle\rangle$ are used to denote the beginning and end of a list of elements, and $+\!\!+$ denotes list concatenation.

**Algorithm 17**: BYPSb Preprocess ($l < 8$)
$ANS2bPreprocess(m, k)$
$l \leftarrow \lfloor m/(k+1) \rfloor$
$rem \leftarrow m \bmod l$
$q \leftarrow \min(l, 8)$
$shift \leftarrow l - q + 1$
for $f \leftarrow 0$ to $2^{16} - 1$ do
    $H[f] \leftarrow \langle\rangle$
for $subpat \leftarrow 0$ to $m - l - rem * (l + 1)$ step $l$ do
    for $j \leftarrow 0$ to $shift - 1$ do
        $f \leftarrow simd\text{-}crc(p_{subpat+j} \ldots p_{subpat+j+q-1})$
        $H[f] \leftarrow \langle(subpat, j)\rangle +\!\!+ H[f]$
for $subpat \leftarrow m - rem * (l + 1)$ to $m - (l + 1)$ step $l + 1$ do
    for $j \leftarrow 0$ to $shift - 1$ do
        $f \leftarrow simd\text{-}crc(p_{subpat+j} \ldots p_{subpat+j+q-1})$
        $H[f] \leftarrow \langle(subpat, j)\rangle +\!\!+ H[f]$

We implemented another version of BYPSb which completely skips the step of exactly verifying the occurrence of a subpattern and assumes a match every time there is a fingerprint match in the text. Given that $q = \min(l, 8)$, this step would not be necessary in cases when $q = l$ if the hash function worked perfectly. Note that if a wrong subpattern match assumption is made, the efficacy of the algorithm is not affected because ANS2b analyzes the window afterwards. Let us call it BYPSc.

In the worst case, BYPSb time complexity is equal to executing ANS2b over all the text plus the work needed to check the exact occurrences of all corresponding subpatterns at each of the $O(n/shift)$ text alignments for each fingerprint match. Assuming the worst case for the hash function in which all text fingerprints match every preprocessed fingerprint of the *shift* $q$-grams of each of the $k + 1$ subpatterns, its complexity is in

$$O(\lceil m/b \rceil n + (k+1)shift\lceil l/b \rceil n/shift) = O(\lceil m/b \rceil n + k\lceil l/b \rceil n), \text{ for } k > 0$$

Now let us analyze BYPSb average case. As in the worst case, we will split this calculation into two pieces: the expected work needed to obtain all exact subpattern

**Algorithm 18**: BYPSb Search ($l < 8$)
$lastpos \leftarrow 0$
$occ \leftarrow 0$
for $i \leftarrow 0$ to $n - q$ step *shift* do
  $f \leftarrow simd\text{-}crc(t_i \ ... \ t_{i+q-1})$
  forall $(subpat, j)$ in $H[f]$ do
    if $simd\text{-}memcmp(t_{i-j} \ ... \ t_{i-j+l-1},\ p_{subpat} \ ... \ p_{subpat+l-1})$ then
      $begin \leftarrow \max\{i - j - (m - l),\ lastpos - m + 1,\ 0\}$
      $end \leftarrow \min\{i - j - subpat + m - 1,\ n - 1\}$
      if $end - begin + 1 \geq m$ then
        $lastpos \leftarrow end$
        $occ \leftarrow occ + ANS2bSearch(t_{begin} \ ... \ t_{end}, p, k)$
return *occ*

matches in the processed text alignments $E[exact]$, and the total average amount of time demanded by ANS2b searches $E[approx]$. In this way, the average time complexity of BYPSb can be written as

$$O(E[exact] + E[approx]) \tag{A.5.1}$$

In first place, $E[exact]$ corresponds to traversing the text comparing $l$ characters with at most $k + 1$ subpatterns using *simd-memcmp* at most *shift* times at each of the $O(n/shift)$ alignments. Then

$$E[exact] \in O((k+1)shift\lceil l/b \rceil n/shift) = O(k\lceil l/b \rceil n),\ \text{for } k > 0 \tag{A.5.2}$$

Note that it is a pessimistic bound as we are assuming no hash filtering, otherwise *simd-memcmp* is only called whenever there is a hash occurrence.

If we assume a perfect hash function, we get that the expected number of $q$-gram matches after analyzing a text $q$-gram fingerprint at a given position is $E[qmatches] = (k+1)shift/\sigma^q$. We analyze $O(n/shift)$ text alignments and the expected work needed by *simd-memcmp* to compare $l$ text characters with a subpattern is $\lceil l/b \rceil$, so we get

$$\begin{aligned} E[exact] &\in O\left(E[qmatches]\lceil l/b \rceil \frac{n}{shift}\right) = O\left(\frac{(k+1)shift}{\sigma^q}\lceil l/b \rceil \frac{n}{shift}\right) \\ &= O\left(\frac{k\lceil l/b \rceil n}{\sigma^q}\right),\ \text{for } k > 0 \end{aligned} \tag{A.5.3}$$

For the calculation of $E[approx]$ it is important to firstly note that it cannot be greater than the average complexity of ANS2b $O(\lceil k/b \rceil n)$ because it is applied at most over all the text. Now let us call the expected number of exact subpattern occurrences in the text $E[subpatterns]$, and the expected work needed to check for an approximate occurrence of the pattern each time a subpattern is found $E[pattern]$. Then we can rewrite $E[approx]$ as follows

$$E[approx] \in O\left(\min\left(\lceil k/b \rceil n,\ E[subpatterns] \times E[pattern]\right)\right) \tag{A.5.4}$$

The probability of exactly matching a subpattern in a given alignment is $1/\sigma^l$, which coincides with the expected number of matches for this subpattern in one alignment. As we have at most $k+1$ different subpatterns, then by linearity of expectation we get that the expected number of subpattern matches in a given alignment is at most $(k+1)/\sigma^l$. Also, BYPSb analyzes $O(n/shift)$ alignments of subpatterns in the text. So we can conclude that

$$E[subpatterns] \in O\left(\frac{n}{shift}\frac{k+1}{\sigma^l}\right) \tag{A.5.5}$$

On the other hand, $E[pattern]$ corresponds to the expected work needed by ANS2b to look for an approximate occurrence of the pattern in a window of size at most $2m$ characters. Then, as demonstrated in Section A.4.2.1, we can affirm that

$$E[pattern] \in O\left(\lceil k/b \rceil m\right) \tag{A.5.6}$$

Replacing Equations A.5.5 and A.5.6 in A.5.4, we obtain

$$\begin{aligned} E[approx] &\in O\left(\min\left(\lceil k/b \rceil n,\ \frac{n}{shift}\frac{k+1}{\sigma^l}\lceil k/b \rceil m\right)\right) \\ &= O\left(\min\left(\lceil k/b \rceil n,\ \frac{\lceil k/b \rceil kmn}{\sigma^l shift}\right)\right),\ \text{for } k>0 \end{aligned} \tag{A.5.7}$$

Finally, if we assume no hash filtering, we can replace Equations A.5.2 and A.5.7 in A.5.1 and get an average time complexity of BYPSb in

$$O\left(k\lceil l/b \rceil n + \min\left(\lceil k/b \rceil n,\ \frac{\lceil k/b \rceil kmn}{\sigma^l shift}\right)\right) = O(k\lceil l/b \rceil n),\ \text{for } k>0 \tag{A.5.8}$$

Otherwise, under the assumption of perfect hashing and from Equation A.5.3, we get a total average complexity of BYPSb in

$$O\left(\frac{k\lceil l/b \rceil n}{\sigma^q} + \min\left(\lceil k/b \rceil n,\ \frac{\lceil k/b \rceil kmn}{\sigma^l shift}\right)\right),\ \text{for } k>0 \tag{A.5.9}$$

From this analysis it can be predicted a faster execution of BYPSb for lower difference ratios (because $l = \lfloor m/(k+1) \rfloor$ and $q = \min(l, 8)$) and larger alphabets.

As regards BYPSc, we omit the step of using *simd-cmpeq* after a fingerprint coincidence, so from Equation A.5.2 we get

$$E[exact] \in O(n/shift) \tag{A.5.10}$$

Now for the calculation of $E[approx]$ of BYPSc, we consider the expected number of exact subpattern occurrences $E[subpatterns]$ to be the same as the expected number of fingerprint coincidences. There are at most $(k+1)shift$ different fingerprints, so if

we assume a perfect hashing function and follow an analogous reasoning as that used in Equation A.5.5, we get

$$E[subpatterns] \in O\left(\frac{n}{shift}\frac{(k+1)shift}{\sigma^q}\right) = O\left(\frac{nk}{\sigma^q}\right), \text{ for } k > 0 \tag{A.5.11}$$

Under the assumption of no hash filtering, every hash value would be the same, so we would expect to assume that all the preprocessed $q$-grams of all subpatterns exactly appear at each text position analyzed. Then we get

$$E[subpatterns] \in O\left(\frac{n}{shift}(k+1)shift\right) = O\left(nk\right), \text{ for } k > 0 \tag{A.5.12}$$

Given that $E[patterns]$ does not change in BYPSc, if we assume perfect hashing, we can replace Equations A.5.6 and A.5.11 in Equation A.5.4 and obtain

$$\begin{aligned} E[approx] &\in O\left(\min\left(\lceil k/b\rceil n,\ \frac{n}{shift}\frac{(k+1)shift}{\sigma^q}\lceil k/b\rceil m\right)\right) \\ &= O\left(\min\left(\lceil k/b\rceil n,\ \frac{\lceil k/b\rceil kmn}{\sigma^q}\right)\right), \text{ for } k > 0 \end{aligned} \tag{A.5.13}$$

Otherwise, assuming no hash filtering, from Equation A.5.12 we get

$$\begin{aligned} E[approx] &\in O\left(\min\left(\lceil k/b\rceil n,\ n(k+1)\lceil k/b\rceil m\right)\right) \\ &= O\left(\lceil k/b\rceil n\right) \end{aligned} \tag{A.5.14}$$

Finally, assuming no hash filtering, we replace Equations A.5.10 and A.5.14 in Equation A.5.1 and get a time complexity of BYPSc in

$$O\left(\frac{n}{shift} + \lceil k/b\rceil n\right) = O\left(\lceil k/b\rceil n\right) \tag{A.5.15}$$

On the other hand, if we assume perfect hashing, from Equation A.5.13 we obtain the following average time complexity

$$O\left(\frac{n}{shift} + \min\left(\lceil k/b\rceil n,\ \frac{\lceil k/b\rceil kmn}{\sigma^q}\right)\right), \text{ for } k > 0 \tag{A.5.16}$$

## A.5.2. Multiple Pattern Matching with up to $k$ Mismatches

### Multiple-pattern Baeza-Yates–Perleberg enhanced with SIMD

We have extended the BYPS algorithm to work with multiple patterns. The new algorithm MBYPS works as follows:

1. In the preprocessing, we split every pattern into subpatterns of length $l$ or $l+1$ in the same way as in BYPS (described in Section A.5.1.2), where $l = \lfloor m/(k+1) \rfloor$. Then we compute the CRC fingerprint of *shift* $q$-grams of each subpattern, where $q \leq l$ is a tuned parameter of the MEPSM algorithm, and $shift = l - q + 1$. The fingerprint is used to access a table that stores information about which subpattern of which pattern it was computed from.

2. In the search, we compute the fingerprint of a $q$-gram in the text, with which we fetch the corresponding information from the table. We perform a shift of *shift* characters in the text after analysing each $q$-gram, which is the maximum number of characters we can skip.

3. For every subpattern associated with the fingerprint, we naively check if it *exactly* appears at this point. If it does, a possible approximate occurrence of the corresponding pattern is reported.

4. Every time a match candidate of a pattern is found, we use an *approximate* single pattern matching algorithm to verify it.

For the phase of exact multiple string matching we use our tuned version of MEPSM as described in Section A.5.1.2, except that now we extend our tuning of the $q$ calculation to all subpattern lengths. For the phase of approximate single string matching we use ANS2 for $m \leq 32$. For longer patterns, another algorithm should be used.

To avoid re-verifying an occurrence, we keep track of the position up to which the text has already been analyzed for each pattern. In this way, we do not need to check for occurrences starting in positions to the left of the last one analyzed. This would be trivial if MEPSM guaranteed ordering when reporting exact occurrences of subpatterns, but it does not. This has been solved by executing the approximate single pattern matching algorithm in a larger window. If there is an occurrence at position $x$ in the text of a subpattern that begins at position $sp$ of pattern $p$, we check for an approximate occurrence of pattern $p$ from position $x-(m-l)$ to $x+m-sp$. Thus, once an occurrence of a pattern has been found, a newer occurrence will never precede it positionally.

Following the same reasoning as in BYPSb, we also consider a variation of MBYPS where ANS2b is used instead of ANS2 for searching approximate pattern occurrences and *simd-memcmp* is used for exactly comparing strings, which we call MBYPSb. Algorithms 19 and 20 show pseudocode for MBYPSb preprocess and search phases respectively. Changes respect to BYPSb have been highlighted in red. Note that we are comparing MBYPSb with BYPSb version for $l < 8$ because MBYPSb uses this same tuning for calculating $q$ for all values of $l$.

Furthermore, we implemented another version of MBYPSb analogous to BYPSc which skips the step of exactly comparing a subpattern with a text window of $l$ characters after a fingerprint match. We call it MBYPSc.

As regards the time complexity of MBYPSb, there are two differences with the analysis done in Section A.5.1.2 for BYPSb. One is that MBYPSb looks for exact occurrences of at most $(k+1)r$ subpatterns instead of just $k+1$ as in BYPSb. The

**Algorithm 19**: MBYPSb Preprocess

$ANS2bPreprocess(m, k)$
$l \leftarrow \lfloor m/(k+1) \rfloor$
$rem \leftarrow m \bmod l$
$q \leftarrow \min(l, 8)$
$shift \leftarrow l - q + 1$
for $f \leftarrow 0$ to $2^{16} - 1$ do
  $H[f] \leftarrow \langle\rangle$
for $pat \leftarrow 0$ to $r - 1$ do
  for $subpat \leftarrow 0$ to $m - l - rem * (l + 1)$ step $l$ do
    for $j \leftarrow 0$ to $shift - 1$ do
      $f \leftarrow simd\text{-}crc(p^{pat}_{subpat+j} \ldots p^{pat}_{subpat+j+q-1})$
      $H[f] \leftarrow \langle (pat, subpat, j) \rangle +\!\!+ H[f]$
  for $subpat \leftarrow m - rem * (l + 1)$ to $m - (l + 1)$ step $l + 1$ do
    for $j \leftarrow 0$ to $shift$ do
      $f \leftarrow simd\text{-}crc(p^{pat}_{subpat+j} \ldots p^{pat}_{subpat+j+q-1})$
      $H[f] \leftarrow \langle (pat, subpat, j) \rangle +\!\!+ H[f]$

**Algorithm 20**: MBYPSb Search

$occ \leftarrow 0$
$lastpos_{pat} \leftarrow 0 \ \forall \ pat$
for $i \leftarrow 0$ to $n - q$ step $shift$ do
  $f \leftarrow simd\text{-}crc(t_i \ldots t_{i+q-1})$
  forall $(pat, subpat, j)$ in $H[f]$ do
    if $simd\text{-}memcmp(t_{i-j} \ldots t_{i-j+l-1},\ p^{pat}_{subpat} \ldots p^{pat}_{subpat+l-1})$ then
      $begin \leftarrow \max\{i - j - (m - l),\ lastpos_{pat} - m + 1,\ 0\}$
      $end \leftarrow \min\{i - j + m - subpat,\ n - 1\}$
      if $end - begin \geq m$ then
        $lastpos_{pat} \leftarrow end$
        $occ \leftarrow occ + ANS2bSearch(t_{begin} \ldots t_{end}, p^{pat}, k)$
return $occ$

second one is that a subpattern occurrence can now belong to many patterns, so it needs to perform at most $r$ ANS2b searches in each window of at most $2m$ characters. In this way, we can deduce that MBYPSb time complexity in the worst case is in

$$O(r\lceil m/b \rceil n + rk\lceil l/b \rceil n), \text{ for } k > 0$$

As regards the average time complexity of MBYPSb, if we assume no hash filtering, $E[exact]$ from Equation A.5.2 is increased by a factor of $r$

$$E[exact] \in O(rk\lceil l/b \rceil n), \text{ for } k > 0 \tag{A.5.17}$$

On the other hand, assuming a perfect hash function, we get that $E[qmatches] = r(k+1)shift/\sigma^q$ since there are now $r(k+1)shift$ preprocessed $q$-grams. Then we get

$$\begin{aligned} E[exact] &\in O\left(E[qmatches]\lceil l/b\rceil \frac{n}{shift}\right) = O\left(\frac{r(k+1)shift}{\sigma^q}\lceil l/b\rceil \frac{n}{shift}\right) \\ &= O\left(\frac{rk\lceil l/b\rceil n}{\sigma^q}\right), \text{ for } k > 0 \end{aligned} \tag{A.5.18}$$

The increment in the number of subpatterns also affects $E[subpatterns]$, so Equation A.5.5 becomes

$$E[subpatterns] \in O\left(\frac{n}{shift}\frac{(k+1)r}{\sigma^l}\right) \tag{A.5.19}$$

Since now it needs to perform at most $r$ ANS2b searches for every subpattern match, Equation A.5.6 becomes

$$E[pattern] \in O\left(\lceil k/b\rceil mr\right) \tag{A.5.20}$$

Then, replacing Equations A.5.19 and A.5.20 in A.5.4, and then replacing it and Equation A.5.17 in A.5.1 we obtain that MBYPSb search phase average time complexity if there is no hash filtering is in

$$\begin{aligned} &O\left(rk\lceil l/b\rceil n + \min\left(r\lceil k/b\rceil n,\ \frac{n}{shift}\frac{(k+1)r}{\sigma^l}\lceil k/b\rceil mr\right)\right) \\ &= O\left(rk\lceil l/b\rceil n + \min\left(r\lceil k/b\rceil n,\ \frac{r^2k\lceil k/b\rceil mn}{\sigma^l shift}\right)\right) \\ &= O\left(rk\lceil l/b\rceil n\right), \text{ for } k > 0 \end{aligned} \tag{A.5.21}$$

Or, assuming perfect hashing, from Equation A.5.18, we obtain a total average time complexity of MBYPSb in

$$O\left(\frac{rk\lceil l/b\rceil n}{\sigma^q} + \min\left(r\lceil k/b\rceil n,\ \frac{r^2k\lceil k/b\rceil mn}{\sigma^l shift}\right)\right), \text{ for } k > 0 \tag{A.5.22}$$

Note that we pessimistically assumed in Equation A.5.20 that each subpattern found appears in all patterns. Nonetheless, this analysis implies that the lower the difference ratio (keep in mind that $l = \lfloor m/(k+1)\rfloor$, so it becomes larger), the bigger the alphabet and the smaller the set of patterns, the faster MBYPSb will perform.

As regards MBYPSc, following the same reasoning as in Section A.5.1.2 and the analysis done for MBYPSb, if we assume no hash filtering we obtain an average time complexity of

$$O\left(\frac{n}{shift} + r\lceil k/b\rceil n\right) = O\left(r\lceil k/b\rceil n\right) \tag{A.5.23}$$

And if we assume a perfect hashing function we get the following complexity

$$O\left(\frac{n}{shift} + \min\left(r\lceil k/b\rceil n,\ \frac{r^2\lceil k/b\rceil kmn}{\sigma^q}\right)\right), \text{ for } k > 0 \tag{A.5.24}$$

# Chapter A.6

# Experiments

The tests were run on an Intel Xeon E5-2603 v4 1.70 GHz with 8 GiB DDR3 RAM memory, using 64-bit Ubuntu 17.10. This processor has got a Broadwell microarchitecture and has got SSE4.2 and AVX2, but not AVX-512.

Programs were written in the C programming language and compiled with gcc 7.2.0 using -O3 optimization level. Code for algorithms other than those developed in this document (i.e. those different from ANS2b, HBYN, BYPS/BYPSb/BYPSc and MBYPS/MBYPSb/MBYPSc) was provided by their authors. All the algorithms were implemented and tested in the testing framework of Hume and Sunday [21], which allows to measure preprocessing and search times separately. It also performs all reading operations before starting the search phase of an algorithm. Furthermore, it prints only after measuring times of an algorithm. In this way, there is no CPU time invested on unsynchronized I/O, which could interfere with measurements. All approximate pattern occurrences are only counted and not printed out.

We used two texts from different alphabets for testing: DNA (the genome of Escherichia Coli, 4.6 MiB) and English (the King James version of the Bible, 4.0 MiB). Texts were taken from the Smart corpus[1]. Each string of the sets of patterns was generated by randomly selecting a substring of length $m$ from the text, and then substituting a random number of its characters (between 0 and 8 included) into random characters that appear in the text in order to simulate mismatches.

We report the times of executing the search phase of each algorithm, obtained from the average of 100 runs. We omit the preprocess time because we address the *online* version of the problem (see Section A.2.1), so it becomes insignificant compared to search time for an arbitrarily long text.

We also ran tests on a Protein sequence (from the Human sequence genome, 3.1 MiB) taken from the Smart corpus. Protein files are usually encoded in an alphabet of size 20, each character corresponding to a different amino acid. We obtained similar results to those reported in this chapter for English alphabet for single string matching and DNA alphabet in the case of multiple string matching. We omit these results to avoid redundancy.

[1] http://www.dmi.unict.it/~faro/smart/corpus.php

We executed the same sets of tests in a more modern machine in Appendix A.A, obtaining even more positive results for SIMD-based algorithms than those presented in this chapter.

**A word of warning.** Our experimental results hold on the processors we used in our tests. It is possible that future processors will give different results if the relative speed of instructions changes.

## A.6.1. Single Pattern Matching with up to $k$ Mismatches

The following algorithms have been compared for this problem:

- SA: Shift-Add [3] (Section A.3.1.2).
- TuSA: Tuned Shift-Add [9] (Section A.3.1.2).
- TwSA: Two-way Shift-Add [9] (Section A.3.1.2).
- EF: Enhanced FAAST [31] (Section A.3.1.3).
- EFS: Enhanced FAAST with SIMD [14] (Section A.4.2.2).
- ANS, ANS2 and ANS2b: Approximate Naive enhanced with SIMD [14] (Sections A.4.2.1 and A.5.1.1).
- BYP: the original Baeza-Yates–Perleberg algorithm [5] (Section A.3.1.4).
- BYPS, BYPSb and BYPSc: Baeza-Yates–Perleberg enhanced with SIMD (Section A.5.1.2).

ANS2b, BYPS, BYPSb and BYPSc were developed in this thesis. We used sets of 100 different patterns for testing. For each algorithm we show the sum of times it took to search each pattern of the set.

According to tests by Hirvola [19], TwSA was the best for English data. According to tests by Salmela et al. [31], EF was the best for DNA data. Our tests confirm the tendencies reported in [14], with a dominance of SIMD-based algorithms.

Results are shown in Table A.6.1 with the best times highlighted. There are plots of search times of the fastest variations of tested algorithms as a function of $k$ for $m \in \{16, 32\}$ and both alphabets in Figures A.6.1, A.6.2, A.6.3 and A.6.4. We can observe that ANS2b and BYPSb/BYPSc are almost always the fastest for all parameter combinations on both DNA and English. They are only surpassed by TwSA in English alphabet when $m = 16$ and $k = 2$. BYPSb and BYPSc are the best for cases with low difference ratio. BYPSb performs better in DNA whereas BYPSc does so in English, as predicted by their complexities (see Table A.B.1). On the other hand, ANS2b works better than BYPSb/BYPSc when the difference ratio is rather high.

| | $m = 8$ | | | $m = 16$ | | | $m = 24$ | | | $m = 32$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | $\Sigma$ |
| SA | 1.68 | 1.69 | 1.69 | 1.68 | 1.69 | 1.68 | 1.68 | -[a] | -[a] | 1.68 | -[a] | -[a] | DNA |
| TuSA | 1.31 | 1.31 | 1.31 | 1.31 | 1.31 | 1.31 | 1.31 | -[a] | -[a] | 1.31 | -[a] | -[a] | |
| TwSA | 1.62 | 2.13 | 2.40 | 0.81 | 1.07 | 1.29 | 0.55 | -[a] | -[a] | 0.41 | -[a] | -[a] | |
| ANS | 1.25 | 1.25 | 1.25 | 1.25 | 1.25 | 1.25 | 1.35 | 1.35 | 1.34 | 1.34 | 1.34 | 1.34 | |
| ANS2 | 0.86 | 0.91 | 1.16 | 0.88 | 0.88 | 0.88 | -[b] | -[b] | -[b] | -[b] | -[b] | -[b] | |
| ANS2b | **0.75** | **0.75** | **0.75** | 0.78 | **0.78** | **0.78** | 1.05 | 1.05 | **1.07** | 1.05 | 1.05 | 1.07 | |
| EF | 1.46 | 2.28 | 3.90 | 0.70 | 1.04 | 1.75 | 0.49 | 0.77 | 1.37 | 0.39 | 0.64 | 1.20 | |
| EFS | 1.41 | 2.14 | 3.92 | 0.67 | 0.97 | 1.59 | 0.48 | 0.71 | 1.24 | 0.38 | 0.59 | 1.10 | |
| BYP | 4.18 | 10.13 | 14.82 | 3.56 | 5.32 | 8.23 | 3.62 | 5.16 | 6.36 | 3.67 | 4.99 | 5.83 | |
| BYPS | 1.60 | -[a] | -[a] | 0.35 | 1.56 | 1.93 | 0.25 | 0.42 | 1.60 | 0.19 | 0.35 | 0.48 | |
| BYPSb | 1.43 | -[a] | -[a] | **0.30** | 1.36 | 1.84 | **0.20** | **0.35** | 1.42 | 0.18 | **0.28** | **0.40** | |
| BYPSc | 1.16 | -[a] | -[a] | 0.37 | 1.10 | 1.42 | 0.42 | 0.63 | 1.18 | **0.15** | 0.65 | 0.84 | |
| SA | 1.47 | 1.47 | 1.47 | 1.47 | 1.47 | 1.47 | 1.47 | -[a] | -[a] | 1.47 | -[a] | -[a] | English |
| TuSA | 1.14 | 1.14 | 1.14 | 1.14 | 1.14 | 1.14 | 1.14 | -[a] | -[a] | 1.14 | -[a] | -[a] | |
| TwSA | 0.83 | 1.17 | 1.53 | 0.48 | **0.62** | 0.79 | 0.33 | -[a] | -[a] | 0.26 | -[a] | -[a] | |
| ANS | 1.09 | 1.09 | 1.09 | 1.09 | 1.09 | 1.09 | 1.17 | 1.17 | 1.17 | 1.17 | 1.17 | 1.17 | |
| ANS2 | 0.75 | 0.75 | 0.76 | 0.75 | 0.75 | 0.75 | -[b] | -[b] | -[b] | -[b] | -[b] | -[b] | |
| ANS2b | **0.65** | **0.65** | **0.65** | 0.66 | 0.66 | **0.66** | 0.91 | 0.91 | **0.91** | 0.91 | 0.91 | 0.91 | |
| BYP | 1.19 | 2.29 | 3.24 | 0.77 | 1.33 | 1.87 | 0.54 | 0.92 | 1.31 | 0.49 | 0.80 | 1.08 | |
| BYPS | 1.37 | -[a] | -[a] | 0.27 | 1.43 | 1.42 | 0.16 | 0.28 | 1.44 | 0.17 | 0.20 | 0.28 | |
| BYPSb | 1.19 | -[a] | -[a] | 0.24 | 1.25 | 1.23 | **0.14** | 0.25 | 1.27 | 0.15 | **0.18** | 0.25 | |
| BYPSc | 1.04 | -[a] | -[a] | **0.20** | 1.10 | 1.07 | 0.15 | **0.23** | 1.12 | **0.13** | **0.18** | **0.24** | |

[a] Algorithm not designed to work in this case.
[b] Same as ANS2b.

Table A.6.1: Search times (in seconds) of algorithms for single approximate pattern matching with up to $k$ mismatches ran 100 times with different patterns.

ANS, ANS2 and ANS2b work for all possible values of $k$. ANS does so at an almost constant speed independent of the value of $k$. ANS2 speed decreases as $k$ grows, which is most notorious when $m = 8$ in DNA because of the high number of occurrences. As regards ANS2b, its performance is independent from $k$ for $m \leq 16$ as explained in Section A.5.1.1. It showed a clear improvement compared to ANS2 surpassing it in all tests, even in cases of few occurrences (low difference ratios). Timings for ANS2 when $m > 16$ have been omitted as it works in the same way as ANS2b and it would have been redundant (see Section A.5.1.1 for more details).

We also ran tests of ANS2 for all possible values of $k$ for DNA alphabet and $m = 16$, getting timings plotted in Figure A.6.5. By using `perf`[2] tool, we gathered information about those executions. In particular, the percentage of conditional branches that were wrongly predicted and the number of instructions it took each execution to finish. Each

[2]Performance analysis tool for Linux. `https://perf.wiki.kernel.org/index.php/Main_Page`

characteristic is reported in Figures A.6.7 and A.6.6 respectively as a function of $k$. It is easy to see that the peak in execution time of ANS2 is based on an increment in the number of branch mispredictions. The number of instructions executed by the program also rises as $k$ grows because there are more approximate pattern occurrences so it needs to perform more additions to the occurrences counter. However, it is not as meaningful as the variation in the amount of branch mispredictions. Same testing has been performed on ANS2b, which showed a constant behaviour for all $k$ as predicted in Section A.5.1.1. It kept the same execution time, number of instructions executed and percentage of branch mispredictions independently from the value of $k$. This characteristic is also evident in its timings in Table A.6.1.

SA, TuSA and TwSA are limited to small values of $k$ for long patterns. For example, they only work for $k = 1$ in the case of $m \in \{24, 32\}$. Furthermore, the speed of TwSA degrades when $k$ grows, as shown in its complexity in Table A.B.1. EF, EFS, BYP, BYPS, BYPSb and BYPSc exhibit similar behavior, with $k$ affecting their speed. Despite this, the growth of $k$ can be tolerated given that $m$ is large enough, i.e. when we have small difference ratios (see Table A.B.1 for details about BYP complexity). Some timings of BYPS and its variations have been omitted because they do not work for $l = \lfloor m/(k+1) \rfloor < 4$ (see Section A.5.1.2).

BYPS has also been tested for longer patterns. According to our experiments and following the same line as stated by Baeza-Yates and Perleberg in [5], BYP and BYPS and its variations obtain their best results for low difference ratios. As regards BYPSb, we found it to behave always faster than BYPS. On the other hand, BYPSc sometimes works faster than BYPSb (particularly in English alphabet) while other times it is even slower than BYPS. It is also important to remark the performance improvement of BYPS and its variations respect to original BYP, specially in cases of low difference ratios.

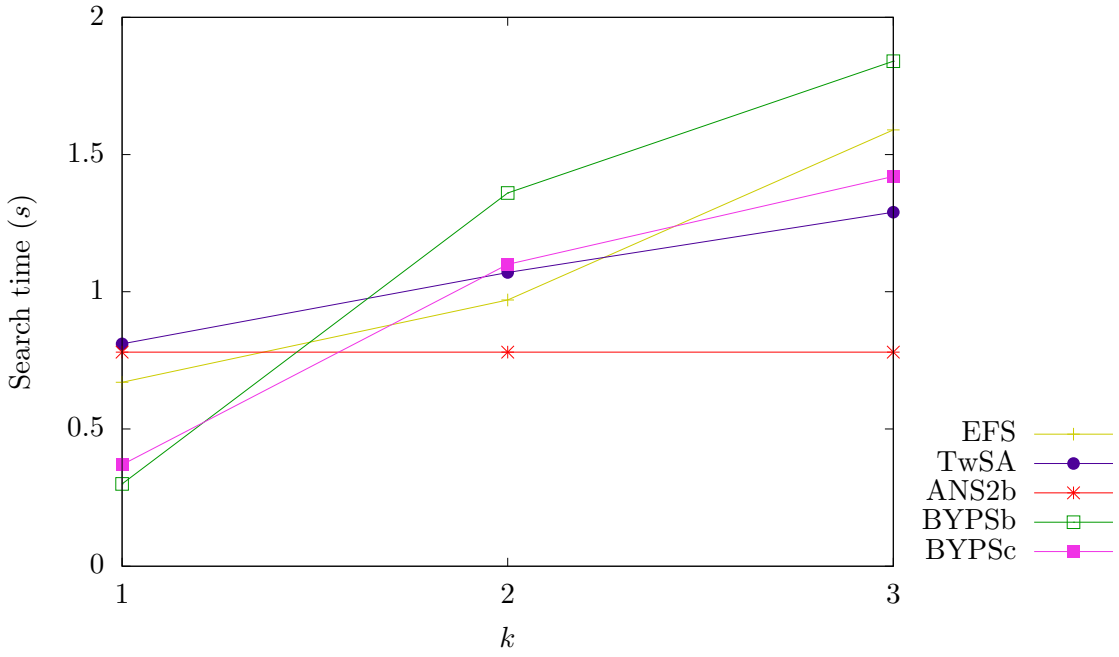



Figure A.6.1: Search times of the fastest algorithms tested as a function of $k$ for $m = 16$ in DNA alphabet.

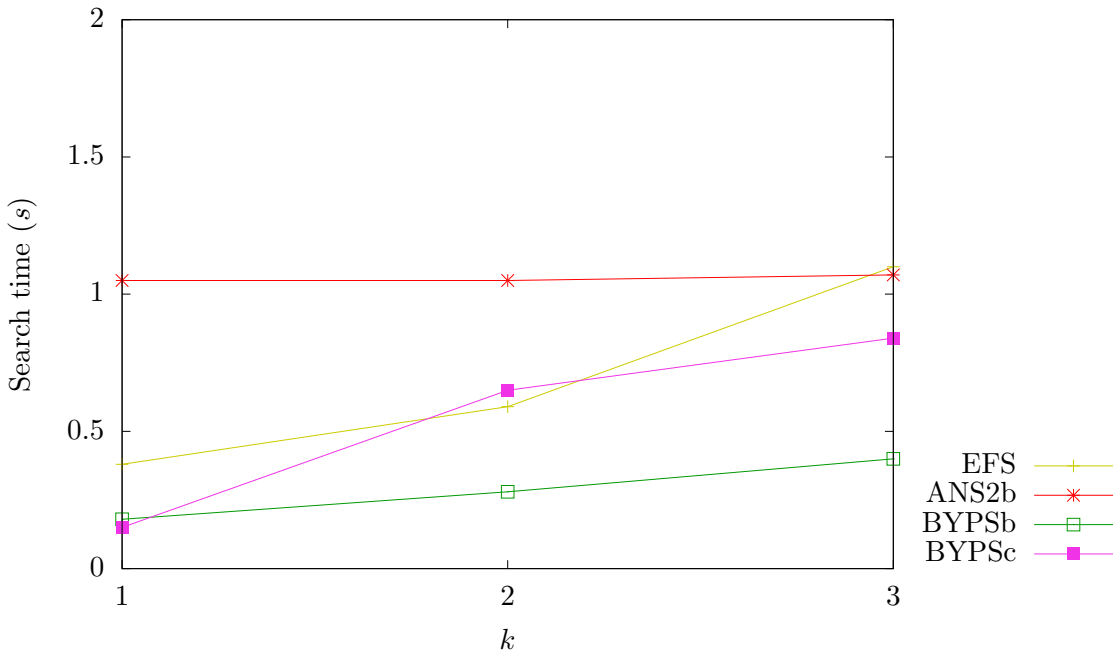



Figure A.6.2: Search times of the fastest algorithms tested as a function of $k$ for $m = 32$ in DNA alphabet.

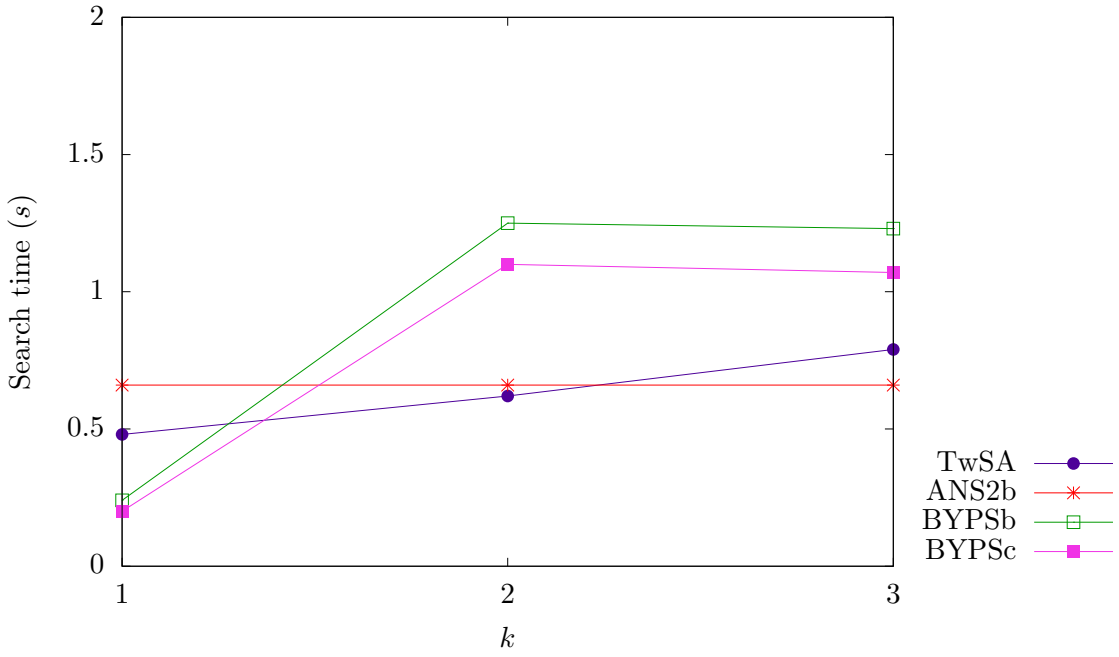



Figure A.6.3: Search times of the fastest algorithms tested as a function of $k$ for $m = 16$ in English alphabet.

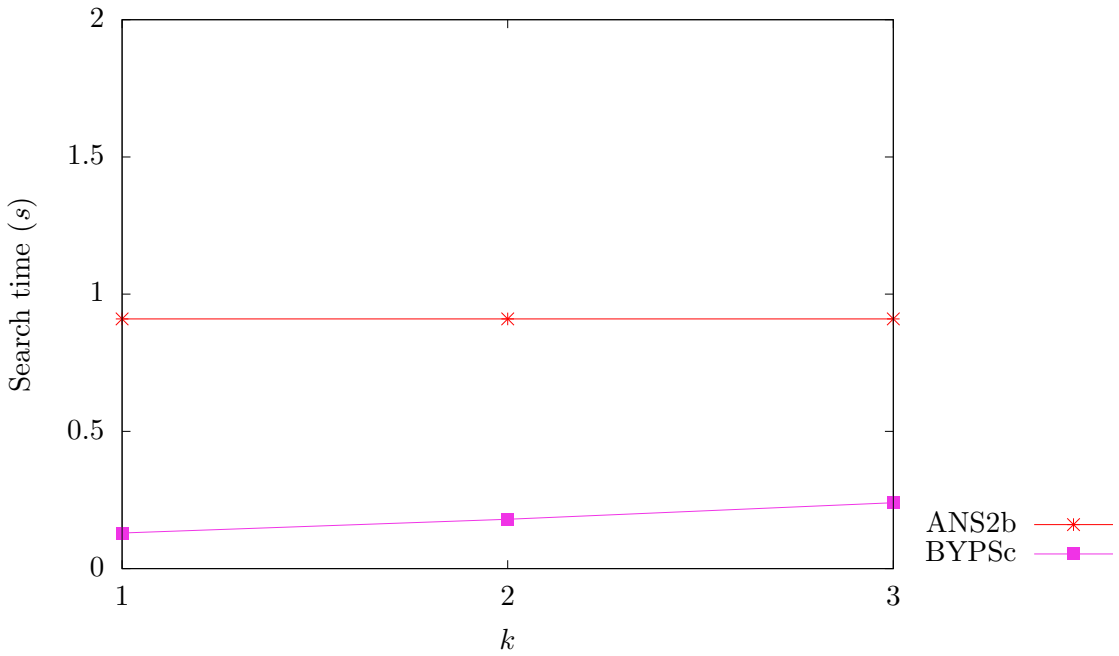



Figure A.6.4: Search times of the fastest algorithms tested as a function of $k$ for $m = 32$ in English alphabet.

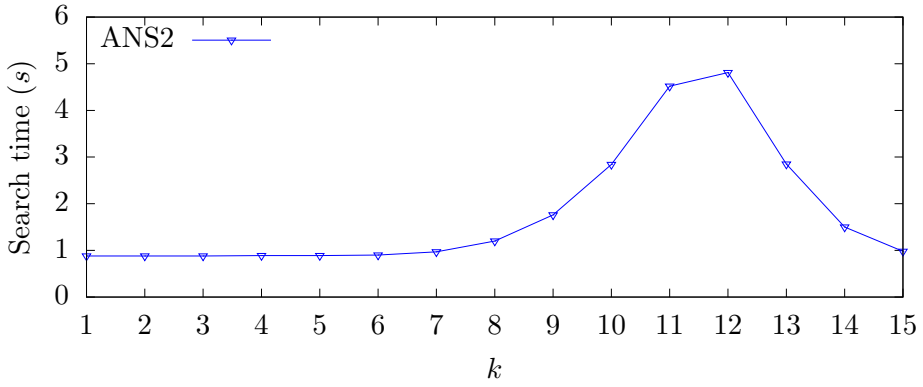



Figure A.6.5: Search times of ANS2 as a function of $k$ for $m = 16$ in DNA alphabet.

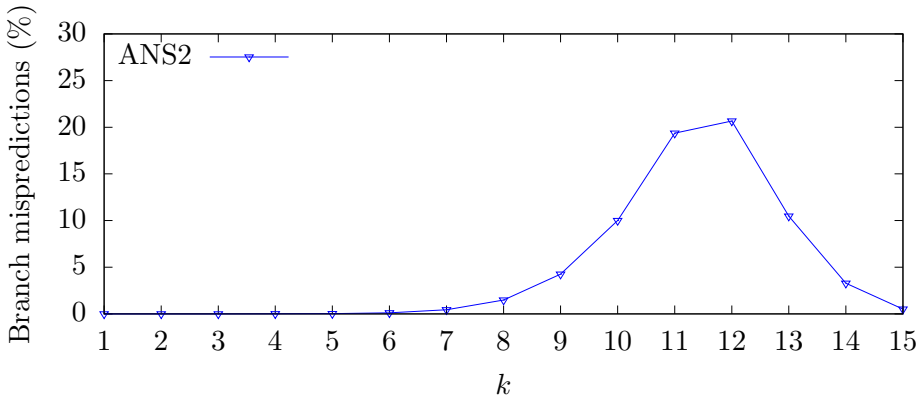



Figure A.6.6: Percentage of branch prediction misses of the total of conditional branches executed in ANS2 as a function of $k$ for $m = 16$ in DNA alphabet.

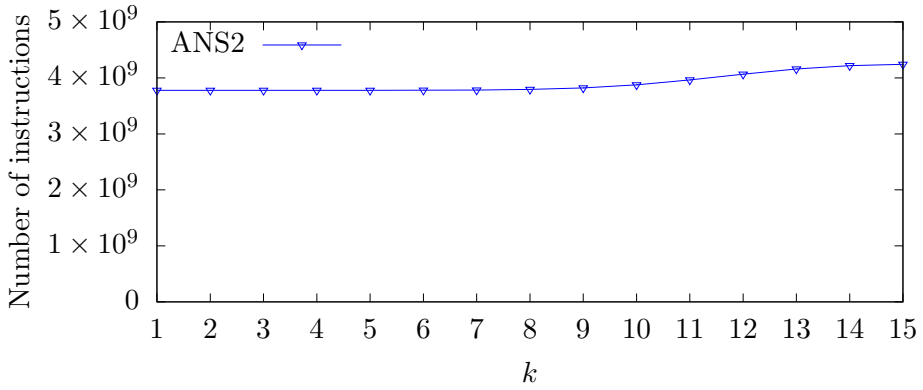



Figure A.6.7: Number of instructions executed in a run of ANS2 as a function of $k$ for $m = 16$ in DNA alphabet.

### A.6.2. Multiple Pattern Matching with up to $k$ Mismatches

We used sets of 10, 100 and 1000 patterns and small $k$ for testing algorithms for the multiple pattern variation of the problem. The following algorithms have been tested:

- MM: the Muth–Manber algorithm [30] (Section A.3.2.2).
- FN: Fredriksson and Navarro's algorithm [15] (Section A.3.2.4).
- BYN: the original Baeza-Yates–Navarro algorithm which detects occurrences under *edit distance* [4] (Section A.3.2.3).
- HBYN: the Baeza-Yates–Navarro algorithm adapted to Hamming distance (Section A.3.2.3).
- MBYPS, MBYPSb and MBYPSc: Multiple-pattern Baeza-Yates–Perleberg enhanced with SIMD (Section A.5.2).

HBYN, MBYPS, MBYPSb and MBYPSc were developed in this thesis.

In first place, it is worth mentioning that HBYN is never significantly slower than original BYN. This means HBYN can be fairly compared to other algorithms since it addresses the same problem as them (unlike BYN), and it was not degraded in performance.

As regards the Fredriksson–Navarro algorithm, thorough testing has been performed in order to choose the best parameters for each case. For DNA we obtained the same tuning mentioned in [15] as the best configuration.

The results are shown in Table A.6.2 with the best times highlighted. There are plots of search times of the tested algorithms as a function of $k$ for $m \in \{16, 32\}$, $r = 100$ and both alphabets in Figures A.6.8, A.6.9, A.6.10 and A.6.11. There are also plots of search times as a function of $r$ for $m = 16$ and $k = 3$, and $m = 32$ and $k = 1$, representing high and low difference ratios respectively, for both alphabets as well, in Figures A.6.12, A.6.13, A.6.14 and A.6.15. For plots of search times versus $r$, logarithmic scales on both axis have been used.

Some timings of the following algorithms have been omitted:

- MM: this algorithm is designed to work only for $k = 1$.
- HBYN: it uses an adaptation of SA (see Section A.3.2.3), which is limited to small values of $k$ for long patterns and only works for $k = 1$ in the case of $m \in \{24, 32\}$.

MBYPSb and MBYPSc almost always outperform all other algorithms, only beaten by MM when $m = 8$, $k = 1$ and $r = 1000$ in English alphabet. In general, there is a larger difference in execution time for lower difference ratios and larger alphabet (i.e. English), as shown in their theoretical complexities in Table A.B.1. Sometimes they even beat other algorithms by an order of magnitude, with MBYPSb best relative

| | m=8 | $m = 16$ | | | $m = 24$ | | | $m = 32$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | 1 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | $r$ | $\Sigma$ |
| MM | 0.105 | 0.108 | -[a] | -[a] | 0.109 | -[a] | -[a] | 0.115 | -[a] | -[a] | 10 | DNA |
| FN | 0.246 | 0.033 | 0.249 | 0.873 | 0.012 | 0.015 | 0.028 | 0.008 | 0.010 | 0.016 | | |
| BYN | 0.138 | 0.115 | 0.132 | 0.196 | 0.114 | 0.131 | 0.135 | 0.117 | 0.126 | 0.140 | | |
| HBYN | 0.136 | 0.115 | 0.133 | 0.189 | 0.113 | -[a] | -[a] | 0.110 | -[a] | -[a] | | |
| MBYPS | 0.043 | 0.017 | 0.029 | 0.087 | 0.004 | 0.017 | 0.022 | **0.002** | 0.006 | 0.018 | | |
| MBYPSb | 0.038 | 0.016 | 0.027 | 0.076 | 0.004 | 0.016 | 0.021 | **0.002** | 0.006 | 0.016 | | |
| MBYPSc | **0.028** | **0.012** | **0.020** | **0.060** | **0.003** | **0.012** | **0.016** | **0.002** | **0.005** | **0.013** | | |
| MM | 0.708 | 0.738 | -[a] | -[a] | 0.730 | -[a] | -[a] | 0.758 | -[a] | -[a] | 100 | |
| FN | 2.215 | 0.306 | 1.977 | 9.952 | 0.023 | 0.062 | 0.164 | 0.015 | 0.027 | 0.065 | | |
| BYN | 0.510 | 0.187 | 0.313 | 0.972 | 0.187 | 0.205 | 0.278 | 0.186 | 0.198 | 0.220 | | |
| HBYN | 0.494 | 0.191 | 0.308 | 0.898 | 0.192 | -[a] | -[a] | 0.188 | -[a] | -[a] | | |
| MBYPS | 0.300 | 0.020 | 0.160 | 0.612 | **0.005** | 0.022 | 0.084 | **0.003** | 0.009 | 0.025 | | |
| MBYPSb | 0.261 | 0.018 | 0.137 | 0.541 | **0.005** | 0.021 | 0.077 | **0.003** | **0.007** | 0.022 | | |
| MBYPSc | **0.221** | **0.014** | **0.121** | **0.488** | **0.005** | **0.017** | **0.066** | 0.005 | 0.009 | **0.019** | | |
| MM | 4.763 | 4.931 | -[a] | -[a] | 4.985 | -[a] | -[a] | 5.151 | -[a] | -[a] | 1000 | |
| FN | 22.283 | 3.094 | 21.133 | 97.438 | 0.183 | 0.678 | 1.999 | 0.111 | 0.301 | 0.668 | | |
| BYN | 3.855 | 0.334 | 1.816 | 8.742 | 0.319 | 0.395 | 1.244 | 0.325 | 0.357 | 0.435 | | |
| HBYN | 3.649 | 0.335 | 1.720 | 7.931 | 0.308 | -[a] | -[a] | 0.324 | -[a] | -[a] | | |
| MBYPS | 2.811 | 0.054 | 1.526 | 6.994 | 0.018 | 0.077 | 0.766 | 0.014 | 0.034 | 0.102 | | |
| MBYPSb | 2.463 | 0.049 | 1.438 | 6.496 | **0.013** | 0.070 | 0.744 | **0.012** | **0.024** | 0.093 | | |
| MBYPSc | **2.216** | **0.041** | **1.313** | **5.808** | 0.033 | **0.065** | **0.681** | 0.035 | 0.059 | **0.090** | | |
| MM | 0.053 | 0.054 | -[a] | -[a] | 0.053 | -[a] | -[a] | 0.052 | -[a] | -[a] | 10 | English |
| FN | 0.048 | 0.016 | 0.027 | 0.052 | 0.010 | 0.015 | 0.023 | 0.006 | 0.012 | 0.017 | | |
| BYN | 0.061 | 0.041 | 0.050 | 0.055 | 0.036 | 0.045 | 0.059 | 0.032 | 0.041 | 0.047 | | |
| HBYN | 0.061 | 0.042 | 0.050 | 0.056 | 0.037 | -[a] | -[a] | 0.031 | -[a] | -[a] | | |
| MBYPS | 0.022 | 0.015 | 0.016 | 0.016 | 0.003 | 0.015 | 0.015 | **0.002** | 0.005 | 0.015 | | |
| MBYPSb | 0.020 | 0.013 | 0.015 | 0.015 | 0.003 | 0.013 | 0.014 | **0.002** | **0.004** | 0.013 | | |
| MBYPSc | **0.015** | **0.010** | **0.012** | **0.011** | **0.002** | **0.010** | **0.011** | **0.002** | **0.004** | **0.010** | | |
| MM | 0.068 | 0.075 | -[a] | -[a] | 0.072 | -[a] | -[a] | 0.066 | -[a] | -[a] | 100 | |
| FN | 0.273 | 0.093 | 0.174 | 0.397 | 0.063 | 0.095 | 0.159 | 0.048 | 0.069 | 0.105 | | |
| BYN | 0.146 | 0.111 | 0.146 | 0.224 | 0.104 | 0.115 | 0.134 | 0.093 | 0.109 | 0.117 | | |
| HBYN | 0.146 | 0.110 | 0.146 | 0.218 | 0.094 | -[a] | -[a] | 0.096 | -[a] | -[a] | | |
| MBYPS | 0.063 | 0.018 | 0.049 | 0.096 | 0.005 | 0.018 | 0.032 | 0.003 | **0.006** | 0.019 | | |
| MBYPSb | 0.054 | 0.016 | 0.043 | 0.089 | **0.004** | 0.016 | 0.029 | **0.002** | **0.006** | 0.017 | | |
| MBYPSc | **0.040** | **0.012** | **0.033** | **0.068** | **0.004** | **0.013** | **0.024** | 0.003 | **0.006** | **0.014** | | |
| MM | **0.242** | 0.272 | -[a] | -[a] | 0.286 | -[a] | -[a] | 0.274 | -[a] | -[a] | 1000 | |
| FN | 3.235 | 0.679 | 2.299 | 7.809 | 0.368 | 0.910 | 1.887 | 0.270 | 0.613 | 1.047 | | |
| BYN | 0.616 | 0.208 | 0.563 | 1.512 | 0.186 | 0.238 | 0.413 | 0.178 | 0.212 | 0.258 | | |
| HBYN | 0.602 | 0.203 | 0.541 | 1.436 | 0.182 | -[a] | -[a] | 0.174 | -[a] | -[a] | | |
| MBYPS | 0.372 | 0.037 | 0.332 | 0.920 | 0.014 | 0.048 | 0.192 | 0.010 | 0.025 | 0.060 | | |
| MBYPSb | 0.327 | 0.032 | 0.299 | 0.823 | **0.011** | **0.041** | 0.162 | **0.008** | **0.019** | **0.051** | | |
| MBYPSc | 0.251 | **0.029** | **0.225** | **0.688** | 0.024 | 0.045 | **0.152** | 0.023 | 0.039 | 0.061 | | |

[a] Algorithm not designed to work in this case.

Table A.6.2: Search times (in seconds) of algorithms for multiple approximate pattern matching with up to $k$ mismatches.

performance being around 24 times (0.048/0.002) faster than the second most efficient algorithm different from MBYPS/MBYPSc (which is FN) for English alphabet, $r = 100$, $m = 32$ and $k = 1$. MBYPSc is more dominant than MBYPSb, as opposed to the case of BYPSc compared to BYPSb, although it sometimes falls behind it in cases of large $r$ and large subpattern length ($l \geq 8$), which corresponds with low difference ratios because $l = \lfloor m/(k+1) \rfloor$ (see Section A.5.2 for the algorithm details).

As regards other algorithms, it is worth mentioning the following:

- Muth–Manber's algorithm remains competitive only for large sets of small patterns in English alphabet, specifically for $r = 1000$ and $m = 8$ in our tests, showing its tolerance to large sets of patterns, as expressed in Section A.3.2.2 and in its complexity in Table A.B.1.
- Fredriksson–Navarro's algorithm sometimes behaves similarly to MBYPS in DNA alphabet for small sets of long patterns, although it never outperforms it.
- Baeza-Yates–Navarro's algorithm adapted to Hamming distance was always surpassed by MBYPS or its variations, but it has been the fastest non-SIMD based algorithm in many cases of high difference ratios or large sets of patterns.

A curious case worth analyzing is presented when an algorithm is run for $r_1$ and $r_2$ where $r_2 > r_1$, and keeping the same $m$, $k$ and alphabet, it yields search times $e_1$ and $e_2$ respectively where it results that $e_2 > \frac{r_2}{r_1} e_1$. For example, in Table A.6.2, we can see that MBYPSc takes 0.488 seconds to search pattern occurrences for $m = 16$, $k = 3$ and $r = 100$ in DNA alphabet, whereas it needs 5.808 seconds to perform the same task for an $r = 1000$, around twelve times its previous time. Two different techniques to address this issue have been used in FN in [15] which are called *pattern grouping* and *pattern clustering*. The former simply consists of dividing the pattern set into smaller sets and search for them separately. The latter is an extension of the first one where the way subsets are divided are beneficial for the searching mechanism, spending more time in the preprocessing phase. Unfortunately, these improvements would not work for MBYPS or its variations, as the reason for its speed decrement relies on its filtering technique. When the number of subpatterns to search starts to grow, the same subpatterns begin to appear in many patterns. So finding them in the text would actually not filter much, since we still need to verify an approximate occurrence of all patterns that contain them. In conclusion, applying pattern grouping in any of these algorithms would only make them go through the text many times, just increasing their search time.

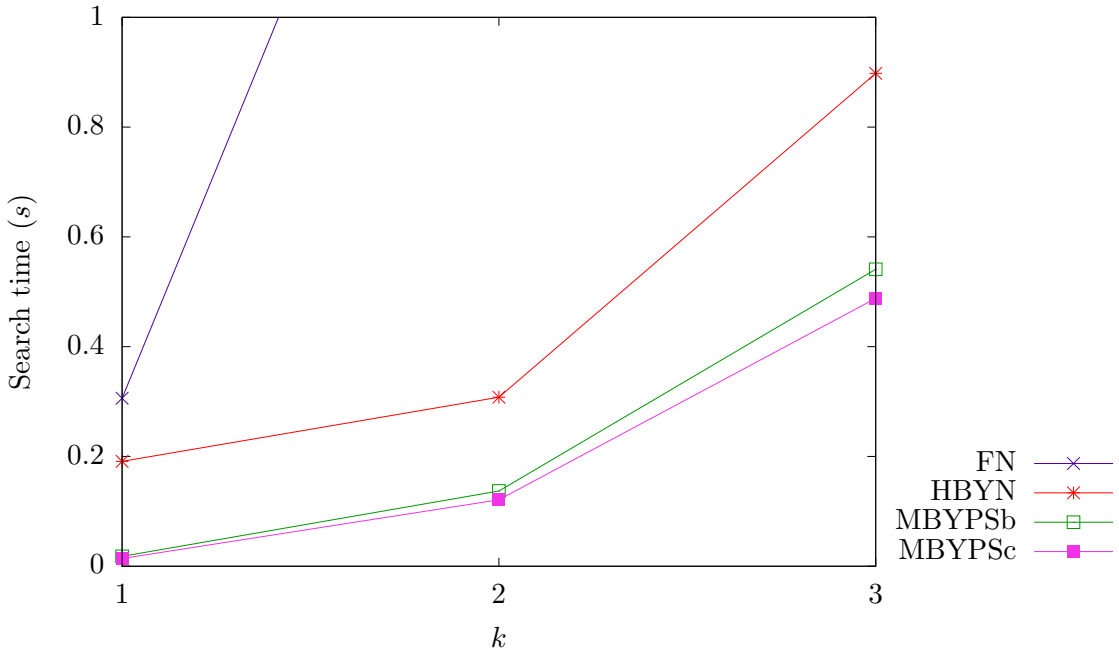



Figure A.6.8: Search times of the algorithms tested as a function of $k$ for $m = 16$ and $r = 100$ in DNA alphabet.

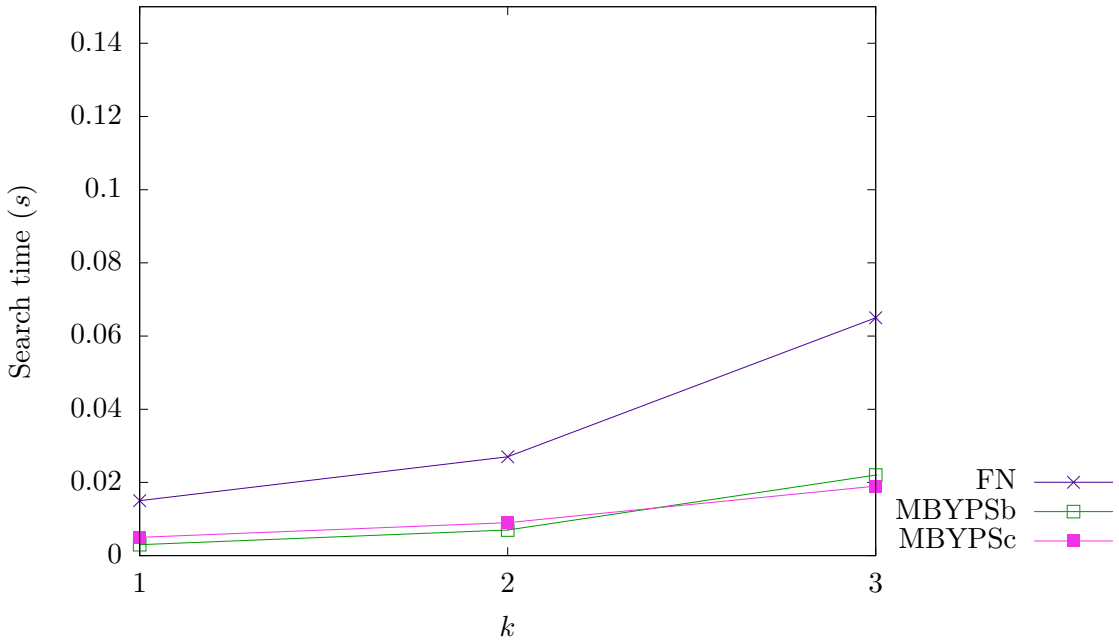



Figure A.6.9: Search times of the algorithms tested as a function of $k$ for $m = 32$ and $r = 100$ in DNA alphabet.

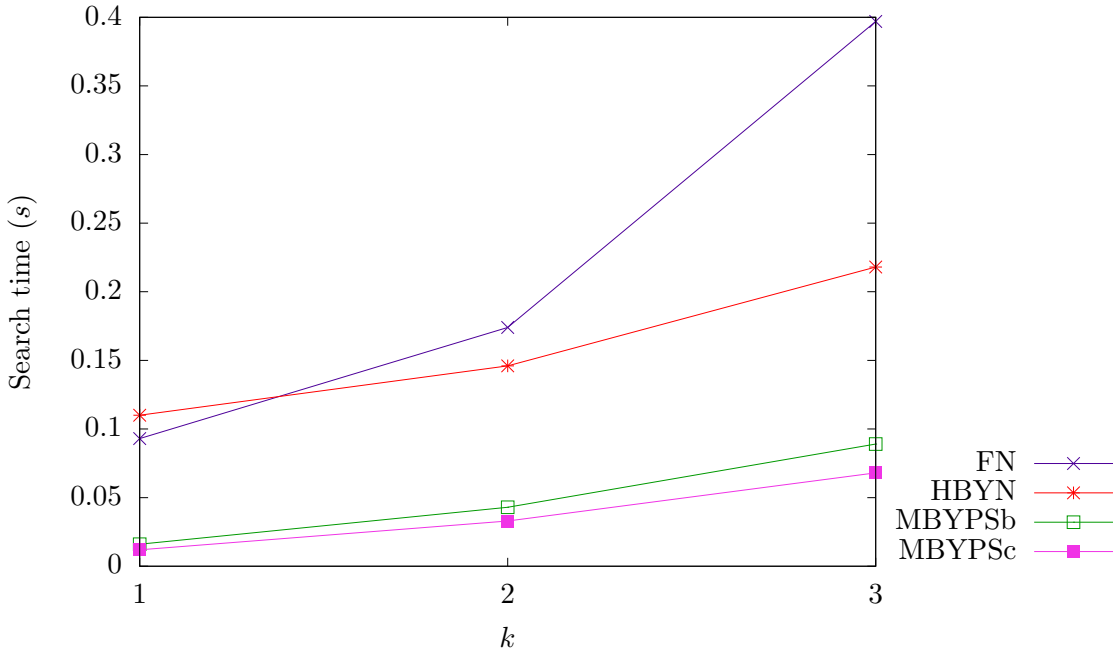



Figure A.6.10: Search times of the algorithms tested as a function of $k$ for $m = 16$ and $r = 100$ in English alphabet.

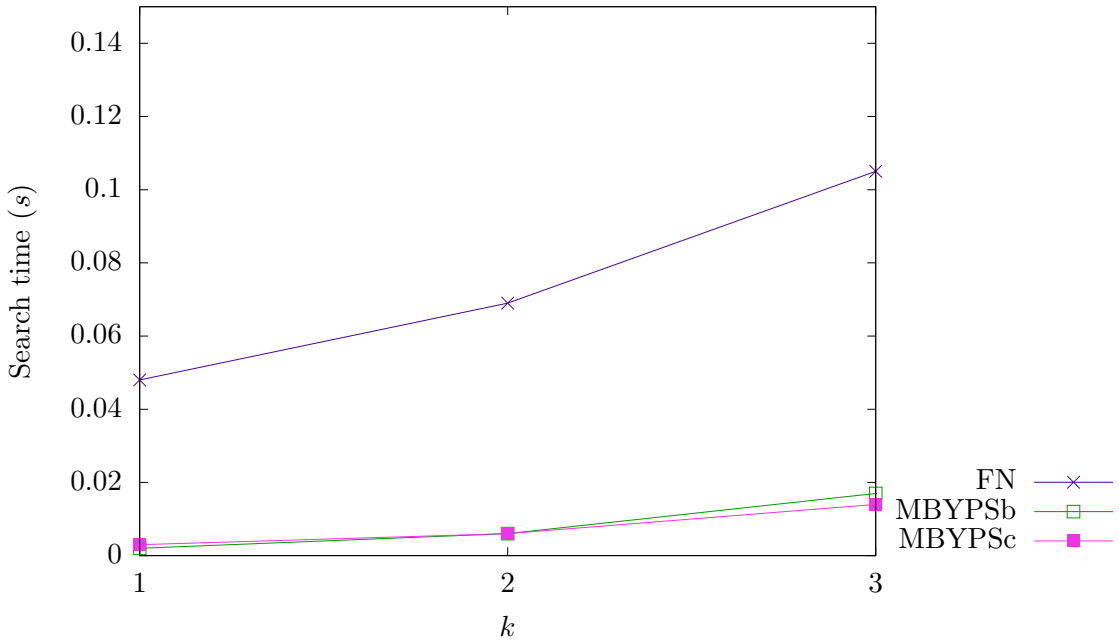



Figure A.6.11: Search times of the algorithms tested as a function of $k$ for $m = 32$ and $r = 100$ in English alphabet.

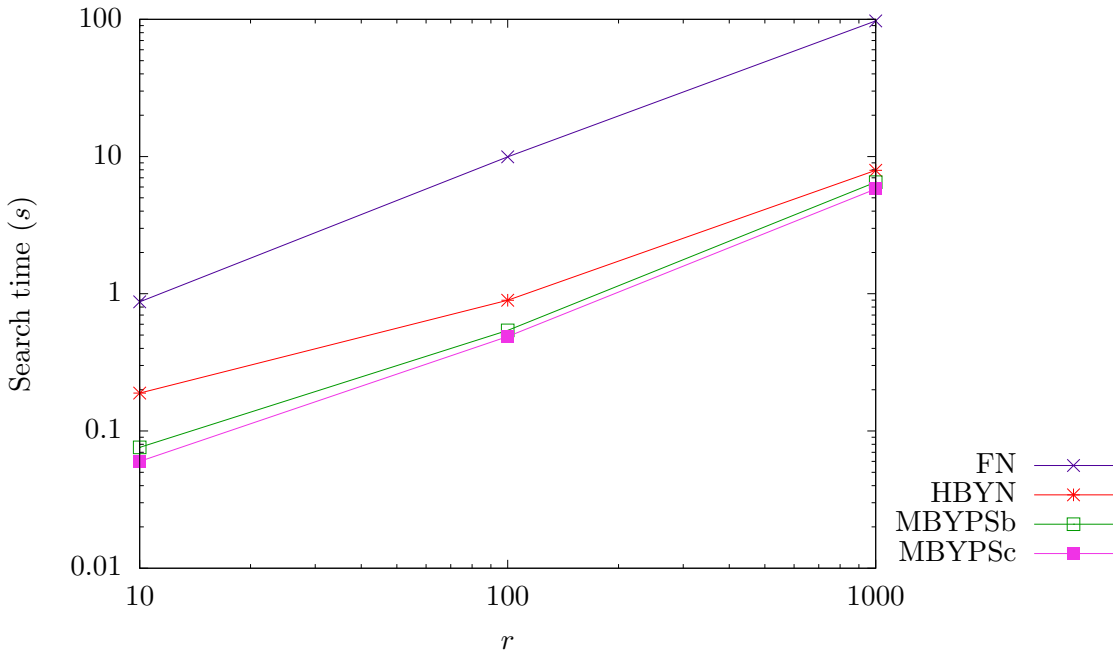



Figure A.6.12: Search times of the multiple pattern algorithms tested as a function of $r$ for $m = 16$ and $k = 3$ (high difference ratio) in DNA alphabet.

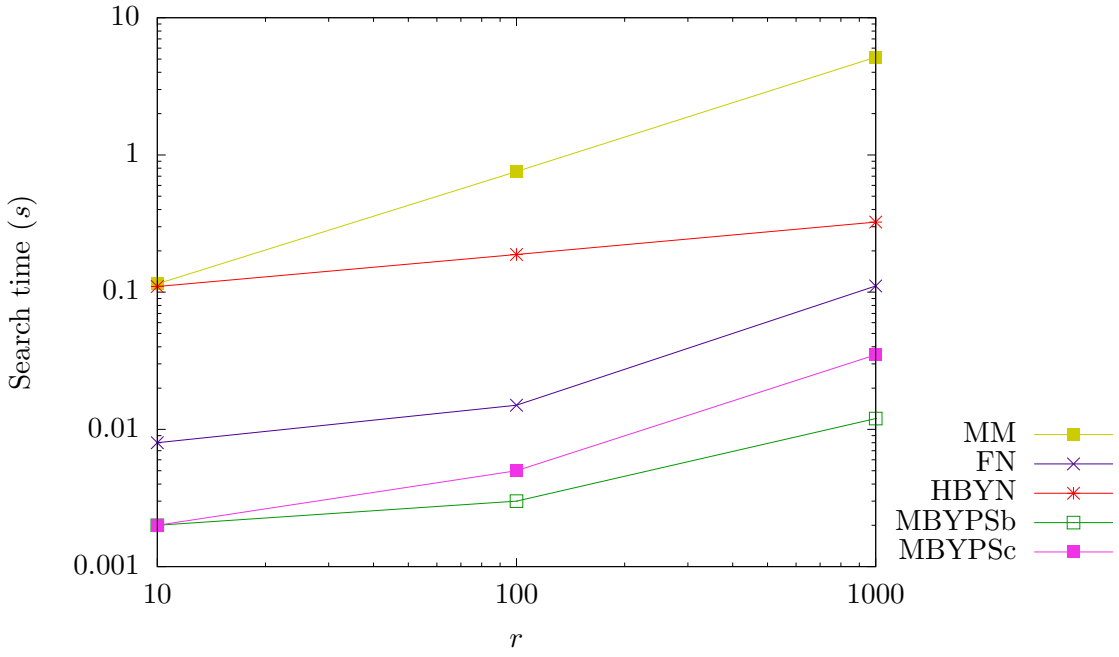



Figure A.6.13: Search times of the multiple pattern algorithms tested as a function of $r$ for $m = 32$ and $k = 1$ (low difference ratio) in DNA alphabet.

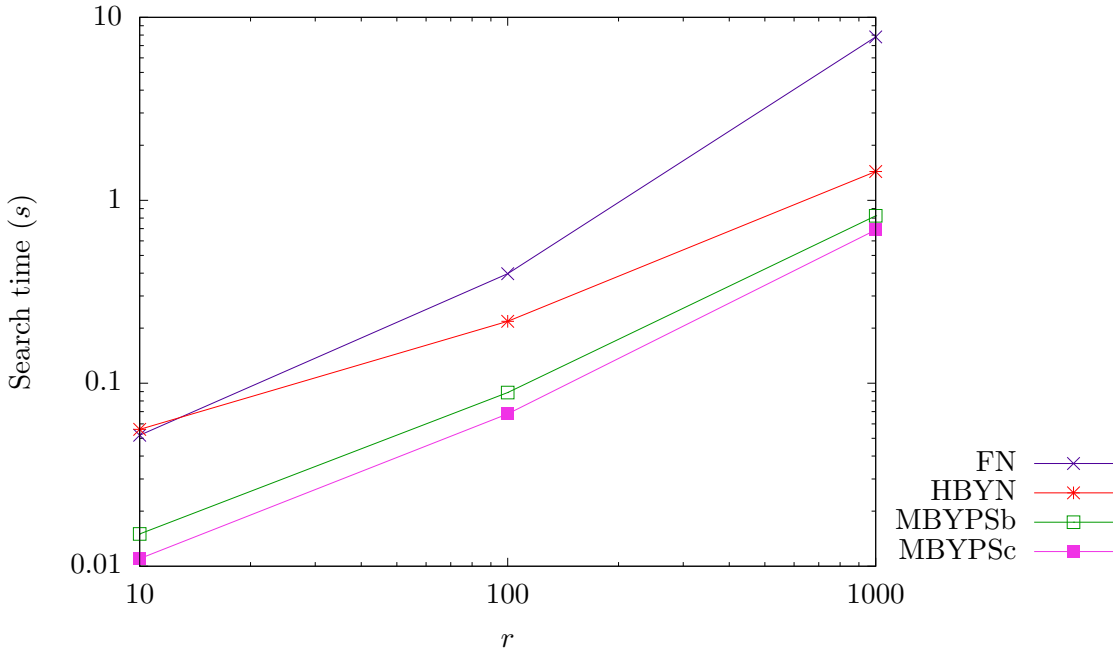



Figure A.6.14: Search times of the multiple pattern algorithms tested as a function of $r$ for $m = 16$ and $k = 3$ (high difference ratio) in English alphabet.

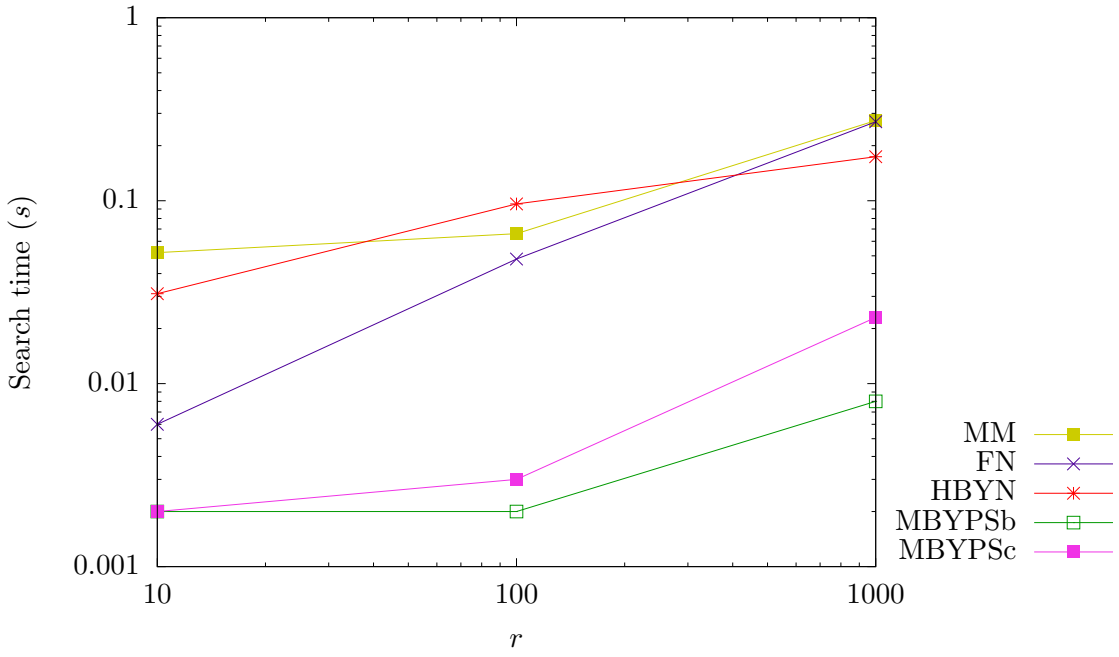



Figure A.6.15: Search times of the multiple pattern algorithms tested as a function of $r$ for $m = 32$ and $k = 1$ (low difference ratio) in English alphabet.

# Chapter A.7

# Conclusions

We have demonstrated that simple SIMD solutions are competitive for searching approximate single and multiple pattern matches within Hamming distance for patterns of at most 32 characters in length. Our experiments in Chapter A.6 and Appendix A.A show that SIMD-based algorithms are the fastest options for small number of mismatches ($k \leq 3$) and sets of at most 1000 patterns in DNA and English alphabets using real life texts. It is important to note that although SIMD-based algorithms presented in this thesis have worse theoretical time complexities than classic algorithms, they perform better in practice.

## A.7.1. Contributions

Part of the results presented in this thesis have already been reported in a conference article coauthored with Waltteri Pakalén and Jorma Tarhio [14]. More precisely, the work done by Waltteri Pakalén in the development of ANS/ANS2 (later improved by me in ANS2b), and my development of BYPS and MBYPS was presented. Jorma Tarhio acted as my advisor during my work.

### ANS2b

We introduced a modification to Approximate Naive enhanced with SIMD [14] (ANS/ANS2, Section A.4.2.1), which we call ANS2b (Section A.5.1.1), that improves performance on small patterns (at most 16 characters long) compared to ANS2. ANS2 has got a performance dependency on $k$ for $m \leq 16$ caused by branch mispredictions, as shown in Figures A.6.5 and A.6.6. ANS2b does not have this issue because it replaces its most evaluated conditional statement by a single instruction, obtaining a constant execution time for all $k$ for $m \leq 16$. This variation resulted in the most efficient algorithm for single pattern for cases of high difference ratio, as shown in Section A.6.1.

### BYPS/BYPSb/BYPSc

We presented a new algorithm for single pattern matching with at most $k$ mismatches called Baeza-Yates–Perleberg enhanced with SIMD (BYPS, Section A.5.1.2). It employs a filtration method based on the SIMD technique described in Section A.4.1.2 for calculating fingerprints.

We also introduced a variation of it which uses ANS2b instead of ANS2 and a SIMD-based string comparison function, which we call BYPSb. Furthermore, we presented a third version of BYPS which skips the filtering step of exact string comparison and assumes a perfect hashing function. We called it BYPSc.

BYPSb and BYPSc demonstrated to be the fastest algorithms for the single pattern variation of the problem in cases of low difference ratio, being BYPSb more useful in DNA alphabet and BYPSc in English alphabet.

### MBYPS/MBYPSb/MBYPSc

We developed a new algorithm for multiple pattern matching under Hamming distance named Multiple-pattern Baeza-Yates–Perleberg enhanced with SIMD (MBYPS, Section A.5.2), which is a natural extension to BYPS. In the same manner as in BYPS, we proposed a variation of it that employs ANS2b instead of ANS2 and a SIMD-based string comparison called MBYPSb, and another one that skips the exact comparison step called MBYPSc. In this case, the latter version proved to be faster than the previous ones more often than observed in the case of BYPSc.

Overall, MBYPSb and MBYPSc resulted in the fastest algorithms for the problem in almost all cases tested. It is worth remarking its importance as an algorithm that runs faster than all algorithms analyzed in this thesis which are the most relevant in the field nowadays.

### HBYN

We modified the Baeza-Yates–Navarro algorithm, which was originally designed to work under edit distance, to work under Hamming distance (see Section A.3.2.3). We obtained a non-SIMD fairly competitive algorithm for this problem, falling behind just SIMD-based solution MBYPS and its variations in many cases.

## A.7.2. Future Work

In the following items we state some interesting lines of work to continue research in the topic.

- Analyze usefulness of algorithms for larger $k$ and higher difference ratio (specially ANS2b), and larger patterns (specially BYPS and its variations and MBYPS and its variations).

- Extend SIMD-based algorithms to work with AVX-512. It may be possible to achieve better speed-ups because compare and mask instructions have been merged into one operation `VPCMPB` in AVX-512BW, as described in Section A.2.3.3. In this way, function *simd-cmpeq*$(a, b)$ from Section A.4.1.1 could be simply defined using just

  ```
  __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b)
  ```

  which compares 64 bytes stored in $a$ and $b$ for equality, and returns the result as a mask in a 64-bit value. Note that this function also allows to extend ANS, ANS2 and ANS2b to efficiently work with patterns up to 64 characters long.

- Investigate more ways of exploiting SIMD instructions. Keep in mind that STTNI instructions latency may change in future architectures, potentially making them suitable for developing high performance algorithms.

- Extend SIMD usage to algorithms for approximate string matching under *edit* distance.

- Study more ways to adapt the automaton of the Baeza-Yates–Navarro algorithm to work under Hamming distance, given that the algorithm obtained in Section A.3.2.3 (HBYN) showed to be a competitive non-SIMD based solution for the multiple pattern case.

# Bibliography


[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. «Basic local alignment search tool». In: *Journal of Molecular Biology* 215.3 (1990), pp. 403–410. ISSN: 0022-2836. DOI: https://doi.org/10.1016/S0022-2836(05)80360-2.

[2] S. Altschul, T. Madden, A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. «Gapped BLAST and PSI-BLAST: a new generation of protein databases search programs». In: *Nucleic acids research* 25 (Oct. 1997), pp. 3389–3402. DOI: 10.1093/nar/25.17.3389.

[3] R. Baeza-Yates and G. Gonnet. «A new approach to text searching». In: *Communications of the ACM* 35.10 (1992), pp. 74–82.

[4] R. Baeza-Yates and G. Navarro. «New and faster filters for multiple approximate string matching». In: *Random Structures & Algorithms* 20.1 (2002), pp. 23–49.

[5] R. Baeza-Yates and C. Perleberg. «Fast and practical approximate string matching». In: *Information Processing Letters* 59.1 (1996), pp. 21–27.

[6] R. Baeza-Yates and B. Ribeiro-Neto. «Modern Information Retrieval». In: *Addison-Wesley* (1999).

[7] T. Chhabra, S. Faro, M. O. Külekci, and J. Tarhio. «Engineering order-preserving pattern matching with SIMD parallelism». In: *Software: Practice and Experience* 47.5 (2017), pp. 731–739.

[8] R. Dixon and T. Martin. «Automatic speech and speaker recognition». In: *IEEE Press* (1979).

[9] B. Ďurian, T. Chhabra, S. Guman, T. Hirvola, H. Peltola, and J. Tarhio. «Improved two-way bit-parallel search». In: *Proceedings of the Prague Stringology Conference* (2014), pp. 71–83.

[10] D. Elliman and I. Lancaster. «A review of segmentation and contextual analysis techniques for text recognition». In: *Pattern Recognition* 23.3/4 (1990), pp. 337–346.

[11] S. Faro and M. O. Külekci. «Fast packed string matching for short patterns». In: *Proc. 15th Meeting on Algorithm Engineering and Experiments, SIAM* (2013), pp. 113–121.

[12] S. Faro and M. O. Külekci. «Towards a Very Fast Multiple String Matching Algorithm for Short Patterns». In: *Proceedings of the Prague Stringology Conference* (2013), pp. 78–91.

[13] S. Faro and E. Pappalardo. «Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem». In: *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science. Lecture Notes in Computer Science* 5901 (2010), pp. 260–281.

[14] F. J. Fiori, W. Pakalén, and J. Tarhio. «Counting Mismatches with SIMD». In: *Proceedings of the Prague Stringology Conference* (2017), pp. 51–61.

[15] K. Fredriksson and G. Navarro. «Average-optimal Single and Multiple Approximate String Matching». In: *J. Exp. Algorithmics* 9 (2004).

[16] S. Gog, K. Karhu, J. Karkkainen, V. Makinen, and N. Valimaki. «Multi-pattern matching with bidirectional indexes». In: *Computing and Combinatorics, J. Gudmundsson, J. Mestre, and T. Viglas. Lecture Notes in Computer Science* 7434 (2012), pp. 384–395.

[17] S. Grabowski and K. Fredriksson. «Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time». In: *Information Processing Letters* 105.5 (2008), pp. 182–187.

[18] D. Higgins and W. R. Taylor. *Bioinformatics: Sequence, Structure, and Databanks: a Practical Approach*. Oxford University Press, 2000. Chap. 3, pp. 173–174. ISBN: 9780199637904.

[19] T. Hirvola. «Bit-parallel approximate string matching under Hamming distance». Master's Thesis. Aalto University, 2016.

[20] R. N. Horspool. «Practical fast searching in strings». In: *Software: Practice and Experience* 10.6 (1980), pp. 501–506.

[21] A. Hume and D. Sunday. «Fast string searching». In: *Software: Practice and Experience* 21.11 (1991), pp. 1221–1248.

[22] Intel. *Intel (R) 64 and IA-32 Architectures Optimization Reference Manual*. December 2017. URL: `https://software.intel.com/en-us/articles/intel-sdm`.

[23] Intel. *Intel (R) 64 and IA-32 Architectures Software Developer's Manual*. December 2017. URL: `https://software.intel.com/en-us/articles/intel-sdm`.

[24] K. Kukich. «Techniques for automatically correcting words in text». In: *ACM Computing Surveys* 24.4 (1992), pp. 377–439.

[25] M. O. Külekci. «Filter based fast matching of long patterns by using SIMD instructions». In: *Proceedings of the Prague Stringology Conference* (2009), pp. 118–128.

[26] S. Kumar and E. Spafford. «A pattern-matching model for intrusion detection». In: *Proc. National Computer Security Conference* (1994), pp. 11–21.

[27] S. Ladra, O. Pedreira, J. Duato, and N. R. Brisaboa. «Exploiting SIMD instructions in current processors to improve classical string algorithms». In: *Proc. 16th East European Conference on Advances in Databases and Information Systems, LNCS* 7503 (2012), pp. 254–267.

[28] Z. Liu, X. Chen, J. Borneman, and T. Jiang. «A fast algorithm for approximate string matching on gene sequences». In: *Proc. 16th Symposium on Combinatorial Pattern Matching, LNCS* 3537 (2005), pp. 79–90.

[29] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001, p. 11. ISBN: 9780879697129.

[30] R. Muth and U. Manber. «Approximate multiple string search». In: *Proc. 7th Symposium on Combinatorial Pattern Matching, LNCS* 1075 (1996), pp. 75–86.

[31] L. Salmela, J. Tarhio, and P. Kalsi. «Approximate Boyer-Moore string matching for small alphabets». In: *Algorithmica* 58.3 (2010), pp. 591–609.

[32] D. Sunday. «A very fast substring search algorithm». In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 132–142.

[33] J. Tarhio, J. Holub, and E. Giaquinta. «Technology beats algorithms (in exact string matching)». In: *Software: Practice and Experience* (2017).

[34] J. Tarhio and E. Ukkonen. «Approximate Boyer-Moore string matching». In: *SIAM Journal on Computing* 22.2 (1993), pp. 243–260.

[35] E. Ukkonen. «Finding approximate patterns in strings». In: *Journal of Algorithmics* 6.1 (1985), pp. 132–137.

# Appendix A.A

# More Experiments

We ran the same experiments to compare performance of algorithms for single and multiple pattern matching with up to $k$ mismatches as in Chapter A.6 but in a more modern machine. This time we used a computer with an Intel i7-8550U CPU with 1.8 GHz with boost up to 4.0 GHz which has a Kaby Lake R microarchitecture. This processor has got SSE4.2 and AVX2, but not AVX-512, in the same way as the CPU used in Chapter A.6. The machine has also got 12 GiB DDR4 2.4 GHz RAM memory and runs 64-bit Ubuntu 18.04. We used gcc version 7.4.0 with the same compiling options as in Chapter A.6.

Results are shown in Tables A.A.1 and A.A.2. We obtained the same outcome as in Chapter A.6 with two exceptions: TwSA and Muth–Manber are not the fastest in cases where they were in previous testing. We can observe even more dominance of SIMD-based algorithms where BYPSb/BYPSc and ANS2b are the fastest ones for single pattern, and MBYPSb/MBYPSc are the fastest solutions for multiple pattern. We conducted these tests to emphasize that our results do not depend on a specific processor, and that the algorithms developed can actually be used in any real-life CPU that supports the required SIMD extensions.

| | $m = 8$ | | | $m = 16$ | | | $m = 24$ | | | $m = 32$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | $\Sigma$ |
| SA | 0.68 | 0.73 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | -[a] | -[a] | 0.70 | -[a] | -[a] | DNA |
| TuSA | 0.60 | 0.62 | 0.59 | 0.58 | 0.56 | 0.56 | 0.56 | -[a] | -[a] | 0.59 | -[a] | -[a] | |
| TwSA | 0.77 | 1.03 | 1.14 | 0.36 | 0.47 | 0.56 | 0.24 | -[a] | -[a] | 0.19 | -[a] | -[a] | |
| ANS | 0.47 | 0.52 | 0.48 | 0.47 | 0.46 | 0.46 | 0.51 | 0.52 | 0.52 | 0.52 | 0.52 | 0.51 | |
| ANS2 | 0.30 | 0.34 | 0.43 | 0.31 | 0.31 | 0.30 | -[b] | -[b] | -[b] | -[b] | -[b] | -[b] | |
| ANS2b | **0.29** | **0.30** | **0.29** | 0.28 | **0.28** | **0.28** | 0.41 | 0.42 | **0.43** | 0.42 | 0.42 | 0.42 | |
| EF | 0.66 | 0.99 | 1.73 | 0.31 | 0.46 | 0.74 | 0.22 | 0.33 | 0.61 | 0.18 | 0.28 | 0.51 | |
| EFS | 0.61 | 0.99 | 1.70 | 0.29 | 0.45 | 0.70 | 0.20 | 0.32 | 0.57 | 0.17 | 0.27 | 0.50 | |
| BYP | 1.82 | 4.51 | 6.46 | 1.56 | 2.34 | 3.60 | 1.59 | 2.33 | 2.87 | 1.66 | 2.19 | 2.56 | |
| BYPS | 0.72 | -[a] | -[a] | 0.15 | 0.64 | 0.80 | 0.11 | 0.19 | 0.81 | 0.09 | 0.15 | 0.21 | |
| BYPSb | 0.63 | -[a] | -[a] | **0.13** | 0.62 | 0.81 | **0.10** | **0.17** | 0.66 | 0.08 | **0.14** | **0.19** | |
| BYPSc | 0.52 | -[a] | -[a] | 0.18 | 0.55 | 0.68 | 0.20 | 0.31 | 0.59 | **0.07** | 0.30 | 0.40 | |
| SA | 0.59 | 0.60 | 0.59 | 0.59 | 0.59 | 0.59 | 0.59 | -[a] | -[a] | 0.59 | -[a] | -[a] | English |
| TuSA | 0.49 | 0.49 | 0.49 | 0.48 | 0.49 | 0.49 | 0.48 | -[a] | -[a] | 0.48 | -[a] | -[a] | |
| TwSA | 0.36 | 0.50 | 0.65 | 0.21 | 0.27 | 0.34 | 0.15 | -[a] | -[a] | 0.11 | -[a] | -[a] | |
| ANS | 0.41 | 0.41 | 0.40 | 0.40 | 0.40 | 0.41 | 0.44 | 0.45 | 0.44 | 0.44 | 0.44 | 0.44 | |
| ANS2 | 0.25 | 0.25 | 0.26 | 0.25 | 0.25 | 0.25 | -[b] | -[b] | -[b] | -[b] | -[b] | -[b] | |
| ANS2b | **0.24** | **0.23** | **0.24** | 0.24 | **0.24** | **0.24** | 0.36 | 0.37 | **0.35** | 0.35 | 0.35 | 0.35 | |
| BYP | 0.52 | 1.00 | 1.39 | 0.33 | 0.58 | 0.82 | 0.23 | 0.40 | 0.57 | 0.21 | 0.35 | 0.47 | |
| BYPS | 0.62 | -[a] | -[a] | 0.12 | 0.65 | 0.63 | 0.07 | 0.12 | 0.67 | 0.07 | 0.09 | 0.19 | |
| BYPSb | 0.56 | -[a] | -[a] | **0.10** | 0.56 | 0.58 | **0.06** | **0.11** | 0.57 | 0.07 | **0.08** | **0.11** | |
| BYPSc | 0.47 | -[a] | -[a] | **0.10** | 0.50 | 0.49 | 0.07 | **0.11** | 0.53 | **0.06** | **0.08** | 0.12 | |

[a] Algorithm not designed to work in this case.
[b] Same as ANS2b.

Table A.A.1: Search times (in seconds) of algorithms for single approximate pattern matching with up to $k$ mismatches ran 100 times with different patterns.

| | m=8 | $m = 16$ | | | $m = 24$ | | | $m = 32$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | 1 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | $r$ | $\Sigma$ |
| MM | 0.045 | 0.047 | -[a] | -[a] | 0.047 | -[a] | -[a] | 0.050 | -[a] | -[a] | 10 | DNA |
| FN | 0.103 | 0.014 | 0.100 | 0.375 | 0.005 | 0.006 | 0.012 | 0.003 | 0.004 | 0.006 | | |
| BYN | 0.061 | 0.048 | 0.059 | 0.087 | 0.047 | 0.057 | 0.059 | 0.049 | 0.053 | 0.060 | | |
| HBYN | 0.059 | 0.048 | 0.057 | 0.081 | 0.047 | -[a] | -[a] | 0.049 | -[a] | -[a] | | |
| MBYPS | 0.017 | 0.007 | 0.012 | 0.036 | 0.002 | 0.008 | 0.010 | 0.001 | 0.003 | 0.008 | | |
| MBYPSb | 0.015 | 0.006 | 0.011 | 0.032 | **0.001** | 0.007 | 0.009 | **0.001** | **0.002** | 0.007 | | |
| MBYPSc | **0.012** | **0.005** | **0.009** | **0.027** | **0.001** | **0.005** | **0.008** | **0.001** | **0.002** | **0.005** | | |
| MM | 0.325 | 0.339 | -[a] | -[a] | 0.335 | -[a] | -[a] | 0.346 | -[a] | -[a] | 100 | |
| FN | 0.968 | 0.131 | 0.843 | 4.291 | 0.009 | 0.026 | 0.068 | 0.006 | 0.012 | 0.026 | | |
| BYN | 0.198 | 0.085 | 0.136 | 0.389 | 0.085 | 0.092 | 0.123 | 0.084 | 0.092 | 0.100 | | |
| HBYN | 0.192 | 0.084 | 0.129 | 0.358 | 0.083 | -[a] | -[a] | 0.084 | -[a] | -[a] | | |
| MBYPS | 0.126 | 0.009 | 0.067 | 0.261 | 0.002 | 0.010 | 0.037 | 0.002 | 0.004 | 0.011 | | |
| MBYPSb | 0.110 | 0.008 | 0.061 | 0.240 | **0.002** | 0.009 | 0.035 | **0.001** | **0.003** | 0.010 | | |
| MBYPSc | **0.092** | **0.006** | **0.053** | **0.205** | **0.002** | **0.007** | **0.031** | 0.002 | 0.004 | **0.008** | | |
| MM | 2.245 | 2.312 | -[a] | -[a] | 2.337 | -[a] | -[a] | 2.409 | -[a] | -[a] | 1000 | |
| FN | 9.765 | 1.328 | 8.756 | 41.400 | 0.079 | 0.290 | 0.828 | 0.048 | 0.125 | 0.273 | | |
| BYN | 1.614 | 0.147 | 0.684 | 3.716 | 0.142 | 0.188 | 0.444 | 0.148 | 0.178 | 0.226 | | |
| HBYN | 1.556 | 0.154 | 0.638 | 3.385 | 0.145 | -[a] | -[a] | 0.148 | -[a] | -[a] | | |
| MBYPS | 1.203 | 0.031 | 0.671 | 3.126 | 0.009 | 0.034 | 0.360 | 0.007 | 0.017 | 0.046 | | |
| MBYPSb | 1.041 | 0.021 | 0.623 | 2.892 | **0.006** | 0.032 | 0.332 | **0.006** | **0.011** | 0.043 | | |
| MYBPSc | **0.876** | **0.018** | **0.565** | **2.477** | 0.015 | **0.029** | **0.301** | 0.016 | 0.027 | **0.041** | | |
| MM | 0.022 | 0.022 | -[a] | -[a] | 0.021 | -[a] | -[a] | 0.021 | -[a] | -[a] | 10 | English |
| FN | 0.020 | 0.006 | 0.011 | 0.022 | 0.004 | 0.007 | 0.010 | 0.003 | 0.004 | 0.007 | | |
| BYN | 0.025 | 0.018 | 0.021 | 0.023 | 0.015 | 0.019 | 0.024 | 0.013 | 0.017 | 0.019 | | |
| HBYN | 0.025 | 0.018 | 0.021 | 0.023 | 0.015 | -[a] | -[a] | 0.013 | -[a] | -[a] | | |
| MBYPS | 0.011 | 0.006 | 0.007 | 0.007 | 0.002 | 0.006 | 0.007 | **0.001** | **0.002** | 0.006 | | |
| MBYPSb | 0.008 | 0.005 | 0.006 | 0.006 | **0.001** | 0.005 | 0.006 | **0.001** | **0.002** | 0.005 | | |
| MBYPSc | **0.007** | **0.004** | **0.005** | **0.005** | **0.001** | **0.004** | **0.005** | **0.001** | **0.002** | **0.004** | | |
| MM | 0.028 | 0.032 | -[a] | -[a] | 0.029 | -[a] | -[a] | 0.028 | -[a] | -[a] | 100 | |
| FN | 0.112 | 0.039 | 0.072 | 0.167 | 0.028 | 0.041 | 0.067 | 0.021 | 0.030 | 0.044 | | |
| BYN | 0.063 | 0.047 | 0.063 | 0.095 | 0.043 | 0.049 | 0.057 | 0.039 | 0.046 | 0.051 | | |
| HBYN | 0.060 | 0.048 | 0.062 | 0.089 | 0.042 | -[a] | -[a] | 0.040 | -[a] | -[a] | | |
| MBYPS | 0.024 | 0.007 | 0.020 | 0.040 | **0.002** | 0.008 | 0.014 | **0.001** | 0.003 | 0.008 | | |
| MBYPSb | 0.021 | 0.007 | 0.017 | 0.034 | **0.002** | 0.007 | 0.012 | **0.001** | **0.002** | 0.008 | | |
| MBYPSc | **0.017** | **0.005** | **0.015** | **0.029** | **0.002** | **0.006** | **0.011** | **0.001** | 0.003 | **0.006** | | |
| MM | 0.111 | 0.124 | -[a] | -[a] | 0.131 | -[a] | -[a] | 0.125 | -[a] | -[a] | 1000 | |
| FN | 1.355 | 0.295 | 0.953 | 3.326 | 0.159 | 0.389 | 0.809 | 0.116 | 0.267 | 0.449 | | |
| BYN | 0.248 | 0.091 | 0.225 | 0.638 | 0.084 | 0.105 | 0.168 | 0.080 | 0.096 | 0.114 | | |
| HBYN | 0.244 | 0.091 | 0.215 | 0.607 | 0.083 | -[a] | -[a] | 0.079 | -[a] | -[a] | | |
| MBYPS | 0.145 | 0.018 | 0.125 | 0.390 | 0.007 | 0.021 | 0.079 | 0.005 | 0.011 | 0.026 | | |
| MBYPSb | 0.122 | 0.014 | 0.109 | 0.343 | **0.005** | **0.019** | 0.073 | **0.004** | **0.009** | **0.024** | | |
| MBYPSc | **0.102** | **0.013** | **0.095** | **0.284** | 0.011 | 0.020 | **0.069** | 0.010 | 0.018 | 0.028 | | |

[a] Algorithm not designed to work in this case.

Table A.A.2: Search times (in seconds) of algorithms for multiple approximate pattern matching with up to $k$ mismatches.

# Appendix A.B

# Summary of Complexities

In Table A.B.1 we summarize the average search time complexities of the most relevant algorithms analyzed in this thesis. It is divided into two parts corresponding to the single pattern and multiple pattern variations of the problem of approximate string matching with up to $k$ mismatches. They have already been presented in previous chapters.

| | Algorithm | Average Complexity |
|---|---|---|
| Single Pattern | SA<br>TuSA | $O(n)$, for $m \leq w/\lceil \log_2(k+1)+1 \rceil$ |
| | TwSA | $O(n(k+log_\sigma m)/m)$, for $m \leq w/\lceil \log_2(k+1)+1 \rceil$ |
| | ANS<br>ANS2<br>ANS2b | $O(\lceil k/b \rceil n)$ [a] |
| | BYP | $O(n)$, for $k \in O(m/\log m)$ |
| | BYPSb | $O\left(\frac{k\lceil l/b \rceil n}{\sigma^q} + \min\left(\lceil k/b \rceil n,\ \frac{\lceil k/b \rceil kmn}{\sigma^l shift}\right)\right)$ [b] |
| | BYPSc | $O\left(\frac{n}{shift} + \min\left(\lceil k/b \rceil n,\ \frac{\lceil k/b \rceil kmn}{\sigma^q}\right)\right)$ [b] |
| Multiple Patterns | MM | $O(ln)$ [c] |
| | FN | $O((k+\log_\sigma(rm))n/m)$, for $k/m < 1/2 - O(1/\sqrt{\sigma})$ |
| | MBYPSb | $O\left(\frac{rk\lceil l/b \rceil n}{\sigma^q} + \min\left(r\lceil k/b \rceil n,\ \frac{r^2\lceil k/b \rceil kmn}{\sigma^l shift}\right)\right)$ [d] |
| | MBYPSc | $O\left(\frac{n}{shift} + \min\left(r\lceil k/b \rceil n,\ \frac{r^2\lceil k/b \rceil kmn}{\sigma^q}\right)\right)$ [d] |

[a] $b$ is the number of characters that fit in a single SIMD register used by the SIMD instructions used in the algorithm.

[b] This complexity assumes a perfect hashing function. See Section A.5.1.2 for details about definitions of $l$, $b$, $q$ and *shift*.

[c] $l$ is the size of the preprocessed prefixes which is chosen empirically. See Section A.3.2.2 for details.

[d] This complexity assumes a perfect hashing function. See Section A.5.2 for details about definitions of $l$, $b$, $q$ and *shift*.

Table A.B.1: Average search time complexities of the most relevant algorithms analyzed in this thesis. All complexities assume $k > 0$.