

Comparación de Algoritmos de coincidencia exacta de cadenas para secuencias biológicas

Petri Kalsi, Hannu Peltola y Jorma Tarhio

Departamento de Ciencias Informáticas e Ingeniería
Universidad Tecnológica de Helsinki
P.O. Box 5400, FI-02015 HUT, Finlandia
petri.kalsi@iki.fi, {hpeltola, tarhio}@cs.hut.fi

Resumen. Se estudia la correspondencia exacta de patrones individuales en secuencias de ADN y aminoácidos. Realizamos una extensa comparación experimental de algoritmos presentados en la literatura. Además, presentamos nuevas variaciones de algoritmos anteriores. Los resultados de la comparación muestran que los nuevos algoritmos son eficientes en la práctica.

1 Introducción

La comparación de cadenas se utiliza para localizar patrones de secuencias de nucleótidos o aminoácidos en las bases de datos de secuencias biológicas. Las secuencias de nucleótidos o ADN tienen un alfabeto de cuatro caracteres y las secuencias de aminoácidos tienen un alfabeto de 20 caracteres, respectivamente. Debido a que el número total de secuencias aumenta rápidamente, se necesitan métodos eficientes. Recientemente se propusieron tres algoritmos novedosos, a saber, "New" de Lecroq [13], SSABS [16] y TVSBS [19], para buscar un patrón único exacto en secuencias de ADN y aminoácidos. Esto nos motivó a comparar estos algoritmos con métodos más antiguos que se sabe que son eficientes. Otra razón son los avances en la tecnología de procesadores y memorias y en compiladores. Los algoritmos antiguos no necesariamente se comportan con los nuevos procesadores y compiladores de la misma manera que lo hicieron hace diez años en nuestra extensa comparación [18]. Además de la comparación en los alfabetos de ADN y aminoácidos, desarrollamos nuevas variaciones de nuestro algoritmo anterior [18].

Resultó que SSABS y TVSBS no eran buenos para las secuencias de ADN. Las nuevas variantes de nuestro algoritmo fueron ligeramente más rápidas que el algoritmo de Lecroq en la mayoría de los casos en ambos alfabetos. El claro ganador para los patrones de aminoácidos de longitud moderada fue SBNDM2 [6], que no ha sido diseñado específicamente para alfabetos biológicos.

En términos de informática, una secuencia biológica es una cadena y, por lo tanto, utilizamos ambos términos. El objetivo de la comparación de cadenas es informar todas las ocurrencias exactas de una cadena de patrones en una cadena de texto más larga. Utilizamos las siguientes notaciones:

Con el apoyo de la Academia de Finlandia.

Autor correspondiente: hpeltola@cs.hut.fi

A lo largo del artículo, la longitud del patrón y del texto son m y n , respectivamente. El tamaño del alfabeto es σ .

El resto del artículo está organizado de la siguiente manera: en la Sección 2, analizamos soluciones anteriores y en la Sección 3, presentamos las nuevas variantes de nuestro algoritmo. En la Sección 4, informamos sobre los resultados de nuestros experimentos antes de llegar a la conclusión en la Sección 5.

2 soluciones conocidas para secuencias de ADN

La mayoría de los algoritmos de comparación de cadenas eficientes en el alfabeto de ADN son [modificaciones del algoritmo de Boyer-Moore \[3\]](#). La mayoría de las veces, solo se aplica la heurística de ocurrencia (también llamada heurística de caracteres incorrectos) para el desplazamiento. Los algoritmos de este tipo son voraces en el sentido de que el patrón se desplaza hacia adelante después de que se observa la primera falta de coincidencia de caracteres de una alineación. Los desplazamientos pueden entonces ser innecesariamente cortos para el alfabeto de nucleótidos si el desplazamiento se basa en un solo carácter. Por lo tanto, resulta ventajoso aplicar q-gramas, cadenas de q caracteres, en lugar de caracteres individuales. Esta técnica ya se mencionó en el artículo original de Boyer y Moore [3, p. 772], y Knuth [12, p. 341] analizó teóricamente su beneficio. Zhu y Takaoka [21] presentaron el primer algoritmo que utilizó la idea.

Su algoritmo utiliza dos caracteres para indexar una matriz bidimensional. También presentaron otra versión basada en hash. El tamaño de la tabla hash era el doble de la longitud del patrón.

Baeza-Yates [1] introdujo una extensión al algoritmo [Boyer-Moore-Horspool \[7\]](#) donde la matriz de desplazamiento se indexa con un entero formado a partir de un q-grama con instrucciones de desplazamiento y adición. Para este tipo de enfoque, el límite superior práctico con caracteres de 8 bits es de dos caracteres.

Para el alfabeto de ADN, [Kim y Shawe-Taylor \[10\]](#) introdujeron una compresión alfabética conveniente enmascarando los tres bits más bajos de los caracteres ASCII. Además de a, c, g y t, se obtienen códigos distinguibles también para n y u.

Incluso el código de control importante $\backslash n = \text{LF}$ tiene un valor distinto, pero $\backslash r = \text{CR}$ obtiene el mismo código que u. Con este método pudieron utilizar q-gramas de hasta seis caracteres. La indexación de la matriz de desplazamiento fue similar al algoritmo de Baeza-Yates.

Con un alfabeto pequeño, la probabilidad de que aparezca un q-grama arbitrario en un patrón largo es alta. Esto restringe la longitud promedio del desplazamiento. Kim y Shawe-Taylor [10] también introdujeron una variación interesante para los casos en los que el q-grama del texto aparece en el patrón. Luego se verifican dos caracteres adicionales uno por uno para lograr un desplazamiento más largo.

En la mayoría de los casos, el q-grama que se toma del texto no coincide con el sufijo del patrón, y el patrón se puede desplazar hacia adelante. Para lograr eficiencia, se debe utilizar un [bucle de salto rápido](#) [8], donde el patrón se desplaza hacia adelante hasta que el último q-grama del patrón coincida con un q-grama en el texto. La forma más fácil de implementar esta idea es definir artificialmente el desplazamiento del último q-grama del patrón como cero. También debemos agregar al final del texto una copia del patrón como tope. Después del bucle de salto, el patrón se compara con las posiciones de texto correspondientes. Una ventaja del bucle de salto es que usamos caracteres de texto tanto para

Comparación preliminar y cálculo del desplazamiento. Por lo tanto, recuperamos los caracteres solo una vez.

Nuestro algoritmo anterior [18] aplica un bucle de salto. La característica clave de este algoritmo es el manejo eficiente de los q-gramas. Los detalles se presentan en la Sección 3.1. Recientemente, **Lecroq [13]** presentó un algoritmo relacionado. Su implementación se basa en el algoritmo **Wu-Manber [20]** para la coincidencia de múltiples cadenas, pero, como se sugirió anteriormente, la idea es más antigua [3,21].

SSABS [16] y **TVSBS [19]** fueron desarrollados para secuencias biológicas. **SSABS** es un algoritmo de tipo **Boyer-Moore**. En la fase de búsqueda, el algoritmo verifica que el primer y el último carácter del patrón coincidan con la alineación actual antes de verificar el resto de la alineación (también conocidas como pruebas de protección). **TVSBS** utiliza un 2-grama para calcular el desplazamiento, adoptado del algoritmo **Berry-Ravindran [2]**, que es un cruce del algoritmo **Zhu-Takaoka** y el algoritmo **QS de Sunday [17]**.

En lugar de la tabla de desplazamiento bidimensional de Berry-Ravindran, **TVSBS** utiliza una función hash para calcular un índice de una tabla unidimensional.

En la Sección 4, presentaremos una comparación experimental de varios algoritmos. Los resultados experimentales de algunos algoritmos anteriores para secuencias de ADN se pueden encontrar en [18]. Hemos incluido tres algoritmos eficientes, a saber, **FAOSO [4]**, **SBNDM [14,15]** y **SBNDM2 [6]** en la comparación, aunque no han sido diseñados para secuencias biológicas.

Recientemente, Kim et al. [11] presentaron un algoritmo para el ADN empaquetado. Lo excluimos de la comparación porque no existe una forma justa de compararlo con otros algoritmos.

3 variaciones de nuestro algoritmo

3.1 Nuestro algoritmo anterior

Nuestro algoritmo [18] es una modificación del algoritmo **Boyer-Moore-Horspool [7]** para el alfabeto de ADN. En lugar de inspeccionar un solo carácter en cada alineación del patrón, el algoritmo lee un q-grama y calcula un entero llamado huella digital a partir de él. Esta idea de aplicar q-gramas ya estaba presente en los algoritmos anteriores [1,10,21], pero nosotros introdujimos una forma eficiente de manejar q-gramas.

Los códigos ASCII de a, c, g y t son 97, 99, 103 y 116, respectivamente.

Los códigos ASCII se asignan al rango de 4: $0 \leq r[x] \leq 3$, donde $r[x]$ es el nuevo código de x, de modo que los caracteres a, c, g y t obtienen códigos diferentes y otros caracteres posibles obtienen, por ejemplo, el código 0. De esta manera, el cálculo se limita al alfabeto efectivo de cuatro caracteres. La huella digital es simplemente un número invertido de base 4. Se utiliza una tabla de transformación separada h_i para cada posición i de un q-grama y se incorporan multiplicaciones durante el preprocesamiento en las tablas: $h_i[x] = r[x] \cdot 4^i$ que luego se calcula como

ⁱ Para $q = 4$, la huella digital de $x_0 \cdot \dots \cdot x_3$ es $\sum_{y=0}^3 r[x_i] \cdot 4^i$

$$h_0[x_0] + h_1[x_1] + h_2[x_2] + h_3[x_3].$$

El algoritmo se presenta a continuación como **BMHq**. Corresponde al algoritmo 4 en [18] sin desenrollar. En el algoritmo, $T = t_1 \cdot \dots \cdot t_n$ denota el texto y $P =$

$p_1 \cdot \dots \cdot p_m$ el patrón. En cada alineación del patrón el último q -grama del patrón se compara con el q -grama correspondiente en el texto probando la igualdad de sus huellas dactilares. Si las huellas dactilares coinciden, una posible ocurrencia se ha encontrado que es necesario comprobarlo. Necesitamos comprobar solo los primeros $m - q$ caracteres del patrón porque nuestro método de huella digital no causa hash colisiones, suponiendo que el texto buscado contiene sólo caracteres de ADN. El algoritmo aplica una variación de un bucle de salto llamado "rápido desenrollado" (=ufast) por Hume y Domingo [8].

Algoritmo 1 BMHq($P = p_1 p_2 \dots p_m$, $T = t_1 t_2 \dots t_n$)

```

1: Inicializar  $D[\ ]$ 
2:  $tn+1 \dots tn+m \leftarrow P$  /* añadiendo tapón */
3:  $p \leftarrow f(P, m, q)$ 
4:  $r \leftarrow D[p]$ ;  $D[p] \leftarrow 0$ ;  $k \leftarrow m$ 
5:  $s \leftarrow D[f(T, k, q)]$ 
6: bucle
7: mientras  $s > 0$  hacer
8:      $k \leftarrow k + s$ 
9:      $s \leftarrow D[f(T, k, q)]$ 
10:    si  $k > n$  entonces salir /* LINEA CORREGIDA */
11:    Verificar la posible ocurrencia
12:     $es \leftarrow r$ 
```

En el algoritmo, $f(T, k, q)$ ¹ denota la huella digital de $t_{k-q+1} \dots t_k$. Para el cálculo de la tabla de desplazamiento D , consulte [18]. El patrón debe copiarse al final del texto, de modo que el bucle de salto ufast finalice cuando se complete la búsqueda.

El "nuevo" algoritmo de **Lecroq** [13] está estrechamente relacionado con **BMHq**. Sin embargo, Lec-roq aplica un método diferente basado en hash para calcular huellas digitales de q -gramas. Además, el desplazamiento máximo de su algoritmo es $m - q + 1$, mientras que de **BMHq** es m , porque **BMHq** puede manejar todos los prefijos del primer q -grama del patrón.

3.2 Variaciones del ADN

Probamos varias formas de calcular la huella digital para que nuestro algoritmo **BMH4** fuera más rápido. Al considerar 4 gramos, caben en una palabra de 32 bits. en caracteres de 8 bits. Algunas arquitecturas de CPU, en particular la x86, permiten caracteres no alineados. lecturas de memoria de varios bytes. Esto nos inspiró a intentar leer varios bytes en **Una instrucción, en lugar de cuatro lecturas de caracteres independientes. Lectura de varios bytes** No es una técnica nueva, muchos investigadores la han utilizado [5,9]. Pero no hemos visto ninguna comparación con la lectura estándar byte a byte. Se puede argumentar que no es justo aplicar lectura múltiple, porque todas las arquitecturas de CPU

¹ Esta es f_2 en [18].

No lo soportan, pero la arquitectura x86 es hoy tan dominante que es razonable adaptar algoritmos para ella.

Presentaremos dos variaciones de **BMH4**. **BMH4b** lee una palabra de 32 bits y **BMH4c** lee dos medias palabras consecutivas. Debido a que en **BMH4b** tenemos acceso a un entero que consta de cuatro caracteres, sería ineficiente utilizar el antiguo método de huellas dactilares basado en caracteres. Las matrices de cálculo de huellas dactilares de **BMH4** se reemplazan con una expresión de hash, donde la entrada es un 4-grama completo como un entero de 4 caracteres. Los códigos ASCII para a, c, g y t se distinguen por los últimos tres bits. La siguiente expresión empaqueta los bits únicos de los cuatro caracteres juntos en unas pocas instrucciones, para formar un entero en el rango de 2313...16191 = 00100100001001...11111100111111.

$$FP(x) = ((x \gg 13) \vee 0x3838) | (x \vee 0x0707)$$

El preprocesamiento de **BMH4b** es similar al algoritmo anterior, solo calculamos las huellas dactilares de los 4-gramas en el patrón con la nueva función hash. Por lo tanto, la principal diferencia entre **BMH4b** y **BMH4** está en el cálculo de f. En **BMH4b**, esto se hace con enmascaramiento, desplazamiento y OR bit a bit. El hash recibe el puntero de ubicación de texto actual como argumento y lee el entero de 32 bits de esa dirección con $FP(*(k-3))$. $D[x]$ contiene los valores de desplazamiento preprocesados para cada q-grama hash del patrón.

Según nuestras pruebas, las lecturas de memoria no alineadas en procesadores x86 incurrir en una penalización de velocidad de hasta el 70% en comparación con las lecturas alineadas. Desafortunadamente, esto reduce la velocidad de **BMH4b**, porque el 75% de las lecturas no están alineadas en promedio. Por lo tanto, hicimos otra variación de **BMH4c**, que lee dos medias palabras consecutivas. En el caso de **BMH4c**, solo el 25% de las lecturas no están alineadas y reciben la penalización de velocidad (llegan al límite de los 4 bytes). En **BMH4c**, el valor de una huella digital se obtiene como $a1[x1] + a2[x2]$ donde $x1$ es una media palabra y $a1$ una tabla de transformación preprocesada, para $i = 1, 2$.

3.3 Variaciones de los aminoácidos

El enfoque del q-grama es válido también para alfabetos más grandes, aunque con un alfabeto más grande el valor óptimo de q es menor. Un tamaño de alfabeto más grande requirió solamente cambios menores en el algoritmo. Se creó una nueva variación **BMH2** basada en **BMHq**. El rango del código ASCII que mapea $r[x]$ se incrementó de 4 a 20, para cubrir el alfabeto de aminoácidos "ACDEF GHIKL MNPQR STVWY" en lugar del alfabeto de ADN. Por lo demás, el algoritmo es el mismo que **BMHq**.

También creamos **BMH2c**, que lee un gramo de 2 como una media palabra de 16 bits. La matriz de desplazamiento está indexada directamente con medias palabras.

Estos algoritmos se pueden utilizar con cualquier texto, por ejemplo, texto en inglés. Para este tipo de datos, asignamos cada carácter a un rango más pequeño con una función de módulo en el preprocesamiento. Los mejores resultados para el texto en inglés se obtuvieron con módulo 25. Debido a que las tablas de mapeo se crean en la fase de preprocesamiento, la operación de módulo no afecta directamente el tiempo de búsqueda.

4 Resultados experimentales

Algoritmos. Entre los algoritmos probados se encuentran **SSABS [16]**, **TVSBS [19]**, **BMH4**, **BMH4b**, **BMH4c**, **BMH2** y **BMH2c**. **KS de Kim y Shawe-Taylor [10]** utiliza un trie de q-gramas invertidos del patrón. El Fast Average Optimal **Shift-Or (FAOSO) [4]** se probó con diferentes valores del tamaño del paso y del factor de desenrollado. Los tiempos de ejecución dados de FAOSO se basan en la mejor combinación de parámetros posible para cada longitud de patrón.

SBNDM [15] se basa en el algoritmo **BNDM [14]**, con cálculos de desplazamiento simplificados. **SBNDM2 [6]** es una modificación de **SBNDM**. Uno de los puntos clave de **SBNDM2** es el desenrollado del bucle, es decir, el manejo de un 2-grama antes de entrar en el bucle interno.

También probamos **LEC**, que es el "nuevo" algoritmo de **Lecroq [13]**, y que utiliza q-gramas y hash. Los tiempos de ejecución de **LEC** se dan en función del mejor q posible para cada longitud de patrón.

SSABS y TVSBS se implementaron como se describe en los artículos [16] y [19]. Debido a que **TVSBS se basa en el algoritmo Berry–Ravindran [2]**, también implementamos dos versiones de este último, **BR y BRX**. En **BRX** hemos modificado el algoritmo para calcular los cambios en función del último carácter de la alineación de texto actual ts y el siguiente carácter $ts+1$, en lugar de $ts+1$ y $ts+2$ como en **BR**, que corresponde al **Berry–Ravindran** original. La probabilidad de un cambio del patrón en una posición es aproximadamente $1/\sigma$ para **BR** y $1/\sigma^2$ para **BRX**.

Por lo tanto, **BRX produce con alfabetos pequeños turnos más largos que BR en promedio**.

Esto se debe a la debilidad inherente del algoritmo **QS** del domingo [17]: si el último carácter del patrón es común, entonces la probabilidad de un desplazamiento de uno es alta.

En tal caso, $ts+2$ suele ser inútil para calcular el desplazamiento de **BR**. Los resultados de nuestras pruebas muestran que esta debilidad de **BR** es perceptible incluso con aminoácidos.

Todos los algoritmos se implementaron en C. Los códigos de KS, SBNDM2, FAOSO y LEC se obtuvieron de los autores originales. Otros códigos fueron implementados por

a. mazzoni

Configuración de prueba. Las pruebas se realizaron en una CPU Pentium D (doble núcleo) de 2,8 GHz con 1 GB de memoria. Ambos núcleos tienen una caché de datos L1 de 16 KB y una caché L2 de 1024 KB. La computadora ejecutaba Fedora 8 Linux.

Todos los algoritmos fueron probados en un marco de pruebas de Hume y Sun-day [8]. Todos los programas fueron compilados con el compilador C **icc 10.0** de Intel, produciendo código x86 de "32 bits" y utilizando el nivel de optimización **-O3**.

Los patrones de prueba se generaron en base a los archivos de texto de modo que aproximadamente la mitad de los patrones tienen coincidencias en el texto. Estos patrones más largos se cortaron luego en patrones más cortos. Cada conjunto de patrones contiene **200 patrones para ADN y 100 para aminoácidos** de la misma longitud. **Para el ADN, los datos de prueba se tomaron del genoma de la mosca de la fruta. El texto de aminoácidos son las secuencias de péptidos de Arabidopsis thaliana. Todos los archivos de texto consistían en aproximadamente $2 \cdot 10^6$ caracteres.** Las ejecuciones de prueba se ejecutaron en una computadora que no tenía carga. Para lograr tiempos precisos, la búsqueda de cada patrón se repitió 20 veces (**con ADN y 50 veces con aminoácidos**). Los resultados informados son medianas para **cinco ejecuciones de prueba** sucesivas utilizando

conjunto de patrones correspondiente. El cambio del proceso de un núcleo de procesador a otro a otro vaciaba las memorias caché en distintos grados. Esto ralentizaría
 Lee desde la memoria e induce variaciones molestas en el tiempo de ejecución de las pruebas.
 Para evitarlo, **hemos utilizado la función de Linux sched_setaffinity para vincular el proceso a sólo un procesador o núcleo.**

Secuencias de ADN. Los tiempos de búsqueda de secuencias de ADN se enumeran en la Tabla 1.
 Los tiempos informados no incluyen el preprocesamiento. Para la mayoría de los algoritmos probados
 Los tiempos de preprocesamiento fueron inferiores al 1% de los tiempos de búsqueda. El preprocesamiento
 El tiempo fue un poco más largo para **BMH2c, BMH4b, BMH4c, TVSBS, KS, BR,**
 y algoritmos BRX, porque tienen grandes estructuras de memoria, que son
 inicializado durante el preprocesamiento.

Tabla 1. Tiempos de búsqueda en milisegundos para secuencias de ADN ($\sigma = 4$)

m	bmh4	bmh4c	ssabs	tvbs	ks	br	brx	sbndm	sbndm2	faoso	lec3-5
4	1004	806	1991	1559	-	1545	1294	1754	1152	601	1390
6	678	540	750	1260	1126	1250	984	1246		885	558
8	534	429	1612	1073	592	1063	811	984		743	357
10	447	355	1570	945	411	936	696	806		657	306
12	391		305	1586	866	324	860	624	692	582	276
14	351	270	556	102	281	795	573	603		515	278
16	317	250	1551	755	241	747	538	538		462	228
18	286		236	1526	716	212			710	506	482
20	257	224	1516	683	196				672	477	440
22	240		210	1536	663	179			654	462	406
24	232		199	1519	645	170			633	442	375
26	220		191	1491	620	162			612	431	348
28	209		182	1526	602	154			593	417	327
30	201		178	1529	589	149			580	407	308
40	173		156	1492	545	126			536	377	-
50	156		144	1481	521	113			511	361	-
60	146		135	1432	501	106			491	346	-
70	135		126	1495	498	105			489	343	-
80	125		112	1551	498	102			489	334	-
90	117	110	1588	494	102				483	334	-
100	111		108	1569	502	107			491	335	-

FAOSO, KS y BMH4c fueron los ganadores de la prueba de ADN. FAOSO fue el ganador más rápido para $m \leq 16$ excluyendo $m = 6$ y $m = 14$ donde BMH4c fue el mejor. KS fue el más rápido para $18 \leq m \leq 100$.

La lectura múltiple es ventajosa, porque BMH4c fue aproximadamente del 20 % más rápido que BMH4 para patrones cortos. Los tiempos de BMH4b (no se muestran) fueron

Tabla 2. Tiempos de búsqueda en milisegundos para secuencias de aminoácidos ($\sigma = 20$)

m	bmh2	bmh2c	ssabs	tvbs	br	brx	sbndm	sbndm2	faoso	lec3-4			
4	320	277	326	336	316	288	346			241	164		622
6	225	195	252	260	246	214	294			138	123		340
8	280	156	207	213	206	172	250			104	101		245
10	152	133	186	187	177	149	219			84	91		201
12	131	118	173	170	161	133	196			74	79		166
14	119	106	155	148	142	119	175			67	79		142
16	112	98	145	137	132	109	157			62	75		128
18	107	91	137	128	121	101			141	59	69		118
20	101	85	132	120	115			96	127	55	70		112
22	94	81	127	116	109			90	116	53	70		105
24	90	77	122	107	103			85	106	51	68		99
26	86	74	118	102			98	82	97	50	69		93
28	83	71	115		98	93	79	90		49	69		89
30	80	69	114		96	93	77	84		48	68		86
40	72	62		103	81	79	67	-	-	-	-		74
50	67	58		98	74	71	61	-	-	-	-		67
60	63	55		95	69	67	58	-	-	-	-		62
70	58	51		92	63	61	54	-	-	-	-		59
80	53	48		92	58	57	51	-	-	-	-		55
90	51	48		91	56	53	50	-	-	-	-		52
100	51	49		91	54	52	49	-	-	-	-		50

entre los de **BMH4** y **BMH4c** para patrones de más de 25 nucleótidos;

Los patrones más cortos eran un poco más lentos que los de **BMH4**. Incluso **BMH2c** (no se muestra)
Fue uno de los mejores para patrones más cortos de 10.

TVSBS demuestra ser una mejora de **SSABS** para patrones largos, pero su

El rendimiento sigue estando en línea con los otros algoritmos de la comparación. Sin embargo, los
resultados de la prueba muestran que **SSABS** y **TVSBS** no son adecuados para

Secuencias de ADN. El algoritmo original de **Berry-Ravindran** era ligeramente más rápido
que **TVSBS**. Los resultados muestran que la modificación del cambio en **BRX** es una clara

Mejora en **BR** para secuencias de ADN.

Secuencias de aminoácidos. Los tiempos de búsqueda en el alfabeto de aminoácidos de la Tabla 2 son
más uniformes. **BMH2c funciona de manera eficiente para todas las longitudes de patrones probadas, pero
SBNDM2 es el ganador absoluto de esta prueba**, ya que los vectores de bits eran de 32 bits.

largo, **BMH2c** y **BRX** dominan cuando $m > 32$. El rendimiento de **BMH2** está cerca
a **BR** y **BRX** para todas las longitudes de patrón probadas.

Notas adicionales. Optimizaciones adicionales podrían mejorar los tiempos de búsqueda de
TVSBS, especialmente el mismo que usamos en **BRX**. Con vectores de bits de 64 bits **FAOSO**,

SBNDM y SBNDM2 pueden manejar patrones más largos que en estas pruebas. En una prueba separada con "código de 64 bits", SBDNM2 fue el más rápido también para longitudes de patrón de hasta 64. Actualmente, solo tenemos la versión de 32 bits de FAOSO, por lo que su rendimiento con vectores de bits más largos sigue siendo desconocido.

Fue una sorpresa que KS fuera el ganador para patrones largos, porque fue más lento en nuestra prueba anterior [18] y también en la prueba preliminar con un compilador más antiguo para este artículo. La mejora de KS se debe a los nuevos compiladores. Con gcc 4.1 el rendimiento es casi el mismo que en la Tabla 1, pero cuando se compila con gcc 4.0.1, 3.4.4 o 3.3.5 los tiempos de búsqueda son al menos un 12% más largos en varias computadoras probadas, e incluso un 100% en el conjunto de los patrones más cortos.

Repetimos las mismas pruebas en otra computadora que tenía dos procesadores AMD Opteron DP 246 de 2,0 GHz, cada uno con 6 KB de caché L1 y 1 MB de caché L2. El tamaño de la memoria principal es de 6 GB (400 MHz DDR PC3200). Lamentablemente, en esa computadora hay una variación inusualmente grande en los tiempos, por lo que fue difícil obtener resultados confiables. En las pruebas con los datos de ADN, BMH4 fue bueno para patrones cortos. KS fue bueno para patrones más largos de 15. BMH4c fue el más rápido para $m \leq 22$. En aminoácidos, BMH2c fue superior para todos los conjuntos de patrones probados. Para la mayoría de los algoritmos, la mejora relativa de la velocidad fue claramente menor que en las pruebas informadas anteriormente, cuando los patrones se hicieron más largos.

Aunque FAOSO era rápido para patrones cortos, es poco práctico. Es decir, tiene dos parámetros constantes y es un proceso tedioso encontrar la mejor combinación de ellos para cada tipo de entrada. Sin variar ambos, es probable que se obtengan tiempos de ejecución más lentos.

5 Conclusión

Hemos demostrado que es posible acelerar el cálculo de q-gramas leyendo varios caracteres al mismo tiempo. Según la comparación, las nuevas variantes BMH4c y BMH2c se encuentran entre las más rápidas para los datos de ADN y aminoácidos, respectivamente. Nuestra intención es seguir trabajando en la lectura múltiple.

Esperamos que la lectura múltiple también acelere otros algoritmos que manejan q-gramas continuos, como BR, LEC y SBNDM2. KS es prometedor para la búsqueda de patrones de ADN largos en procesadores modernos mientras se utiliza un compilador actualizado.

Mostramos cómo solucionar la ineficiencia del algoritmo Berry-Ravindran en el caso de datos de ADN. Esta modificación también es ventajosa para los datos de aminoácidos.

Observamos que los resultados dependen más del procesador y del compilador de lo que esperábamos. El algoritmo A puede ser más rápido que el algoritmo B en la computadora C y viceversa en la computadora D. Para una evaluación práctica más profunda de los algoritmos, se deben probar varios entornos.

Referencias

1. R. Baeza-Yates: Búsqueda de cadenas mejorada. Software: Práctica y experiencia, 19 (3):257–271, 1989.

2. T. Berry y S. Ravindran: Un algoritmo rápido de comparación de cadenas y resultados experimentales. Actas del Taller del Club de Stringología de Praga '99, Universidad Técnica Checa, Praga, República Checa, Informe colaborativo DC-99-05, págs. 16-28, 1999.
3. RS Boyer y J S. Moore: Un algoritmo rápido de búsqueda de cadenas. Comunicaciones de la ACM, 20(10):762–772, 1977.
4. K. Fredriksson y Sz. Grabowski: Coincidencia de cadenas práctica y óptima. En Proc SPIRE'05, Lecture Notes in Computer Science 3772:376–387, 2005.
5. K. Fredriksson: comunicación personal.
6. J. Holub y B. Durian: Variantes rápidas del enfoque de bits paralelos para autómatas de sufijo. (Conferencia inédita) Universidad de Haifa. 2005-04-05.
7. RN Horspool: Búsqueda rápida práctica en cadenas. Software: Práctica y experiencia. Enc, 10(6):501–506, 1980.
8. A. Hume y D. Sunday: Búsqueda rápida de cadenas. Software: Práctica y experiencia, 21(11):1221–1248, 1991.
9. H. Hyrö: Comunicación personal.
10. JY Kim y J. Shawe-Taylor: Coincidencia rápida de cadenas utilizando un algoritmo de n-gramas. Software: Práctica y Experiencia, 24(1):79–88, 1994.
11. JW Kim, E. Kim y K. Park: Método de comparación rápida de secuencias de ADN. En Proc ESCAPE 2007, Lecture Notes in Computer Science 4614:271–281, 2007.
12. DE Knuth, JH Morris y VR Pratt: Coincidencia rápida de patrones en cadenas. SIAM Revista de Computación, 6(1): 323–350, 1977.
13. T. Lecroq: Algoritmos rápidos y exactos de coincidencia de cadenas. Information Processing Letters, 102(6): 229–235, 2007.
14. G. Navarro y M. Raffinot: Coincidencia de cadenas rápida y flexible combinando paralelismo de bits y autómatas de sufijo. ACM Journal of Experimental Algorithms, 5(4):1– 36, 2000.
15. H. Peltola y J. Tarhio: Algoritmos alternativos para la correspondencia de cadenas de bits paralelos. En Proc SPIRE'03, Notas de clase en Ciencias de la Computación 2857:80–93, 2003.
16. SS Sheik, SK Aggarwal, A. Poddar, N. Balakrishnan y K. Sekar: Un algoritmo de coincidencia de patrones FAST. J. Chem. Inf. Comput. Sci., 44(4):1251–1256, 2004.
17. DM Sunday: Un algoritmo de búsqueda de subcadenas muy rápido. Comunicaciones de la Revista de Medicina Interna, 33(8):132–142, 1990.
18. J. Tarhio y H. Peltola: Coincidencia de cadenas en el alfabeto de ADN. Software: Práctica y experiencia, 27(7):851–861, 1997.
19. R. Thathoo, A. Virmani, S. Sai Lakshmi, N. Balakrishnan y K. Sekar: TVSBS: Un algoritmo rápido y exacto de coincidencia de patrones para secuencias biológicas. Current Science 91(1):47–53, 2006.
20. S. Wu y U. Manber: Un algoritmo rápido para la búsqueda de patrones múltiples, Informe TR-94-17, Departamento de Ciencias de la Computación, Universidad de Arizona, Tucson, AZ, 1994.
21. RF Zhu y T. Takaoka: Sobre la mejora del caso promedio del algoritmo de coincidencia de cadenas de Boyer-Moore. Journal of Information Processing, 10(3):173–177, 1987.