Proyecto 3: Detección de Patrones en Múltiples Documentos*

Francy Jélvez†, Leonardo Lovera‡, Diego Venegas§

December 14, 2023

It is a very sad thing nowadays
that there is so little useless information.
— Oscar Wilde (1894)

Encontrar patrones en texto es un problema fundamental en computación que ocurre a menudo en la práctica. En el proyecto anterior discutimos la instancia particular de contar el número de ocurrencias de un patrón `s` en *un* texto `t`. Ahora se desea hacer lo mismo sobre *más* de un solo texto. Es evidente que podemos usar las técnicas discutidas en el proyecto 2 para resolver este problema, pero la complejidad incrementará en un factor de `d`, el número de documentos sobre los cuales se busca. Resulta que podemos mejorar esta complejidad utilizando el Suffix Array y el FM-index con una ligera modificación en sus algoritmos de búsqueda para llegar a mejores resultados.

Se deben tener algoritmos y estructuras de datos que resuelvan este problema rápidamente y que usen espacio razonablemente; podríamos estar procesando una página web, todo el texto en cierto idioma en Wikipedia o el Proyecto Gutenberg, o el genoma de una familia. Así que hay que estar consciente del espacio para procesar eficientemente este gran volumen de datos. Para hacer esto, las soluciones propuestas usan la librería SDSL de C++ que se enfoca en usar algoritmos eficientes y estructuras de datos compactas.

En este proyecto, analizaremos el Suffix Array y el FM-index utilizando el corpus *Pizza&Chili*.

*Universidad de Concepción; Estructuras de Datos y Algoritmos Avanzados. Docente: José Fuentes. Ayudante: Oliver Brito.
†Matrícula: 2021452656.
‡Matrícula: 2021459723.
§Matrícula: 2021433473.

Análisis teórico

El problema a resolver es buscar las ocurrencias de cierto *patrón* en d *documentos*, ambos se consideran como cadenas de caracteres sobre un alfabeto Σ de tamaño finito σ (en la práctica solamente consideraremos caracteres ASCII). El patrón siempre se llamará s y los documentos serán t_k con $0 \leqslant k < d$; sus largos son m y n_k respectivamente y asumimos que $m \leqslant n_k$ para todo k, pues si no en tiempo $O(d)$ se puede determinar *a priori* en cuáles documentos no hay ocurrencias del patrón.

El truco para lograr eliminar el factor de d en la complejidad de búsqueda (que ocurre si se aplican técnicas como KMP, Boyer-Moore, Rabin-Karp, búsqueda con Suffix Array o FM-index sobre cada documento *independientemente*) es crear un nuevo texto sobre el cual se puedan hacer búsquedas y rápidamente se identifique a qué documento corresponde. Esto se hace creando un nuevo texto t, que es la *concatenación de todos los documentos* separados por un caracter especial, denotado \$ y que es distinto a todos los caracteres de todos los documentos y lexicográficamente menor a cualquiera de ellos.

El largo de t es N y es igual a:

$$N = d + \sum_{0 \leqslant k < d} n_k.$$

Donde se suma d porque este es el número de \$'s en t.

El análisis que sigue, lo haremos *output-sensitive* en donde introduciremos una nueva variable importante, r, el número de matches de s en t. Algo interesante es que si el patrón ocurre en d' documentos, sigue por el Principio del Palomar que $d' \leqslant r$. Esto nos permite analizar con más detalle nuestras soluciones.

Suffix Array

Recordemos brevemente en qué consiste el Suffix Array.

Consiste en un arreglo de tamaño N que almacena los sufijos de t. Es común que se agregue al final del texto un caracter especial, comúnmente representado por \$ en la literatura, que es lexicográficamente menor al resto de los caracteres del string para poder representar al sufijo vacío. Como t es la concatenación de los d documentos, y los estamos separando por \$, no es necesario agregar uno al final, pues ya está incluido *a fortiori*.

Hay muchas maneras de construir Suffix Arrays, que van desde la *naive* con complejidad $O(N^2 \lg N)$ hasta algoritmos lineales como SA-IS (Nong, Zhang, Chan (2009)) y el de Li, Li, Huo (2016) que es $O(N)$ y con $O(1)$ espacio extra. La implementación que es más rápida en la práctica es la impresionante librería `libdivsufsort` de Yuta Mori, que corre en tiempo $O(N \lg N)$ en el peor caso.

Con el Suffix Array ya construido podemos usarlo para hacer búsqueda de patrones. La idea de tener a los sufijos ordenados es poder hacer búsqueda binaria sobre ellos para ubicar al patrón. Esto se puede hacer, porque si s está presente en t, entonces es un prefijo de algún sufijo del texto. Así que para hacer esto, debemos hacer $O(\lg n)$ comparaciones —como en toda búsqueda binaria—. Ahora, no se puede concluir tan rápido que eso va a tomar tiempo $O(\lg N)$ en total porque cada comparación *no* toma tiempo constante, sino que toma tiempo $O(m)$, pues hay que comparar a s con el sufijo actual. Así que ejecutar toda la búsqueda toma tiempo $O(m \lg N)$.

Ahora, haciendo dos de estas búsquedas, podemos encontrar un intervalo de largo r en el Suffix Array que corresponde a los índices en donde empieza un s en t. Si mantenemos un arreglo de tamaño d en donde guardamos la posición en donde cierto t_k comienza en t en orden, podemos saber en tiempo $O(\lg d)$ a qué texto pertenece cierto índice. Lo que hay que hacer es encontrar su antecesor en este arreglo (o sí mismo, si ya pertenece a este). Esto se hace con una búsqueda binaria. Luego, toma tiempo $O(r \lg d)$ hacer esto para todas las ocurrencias. A medida que se procesan, se puede mantener una tabla hash que vaya marcando los textos en donde hayan habido ocurrencias y en tiempo $O(1)$ detectar duplicados. Luego, toma tiempo $O(m \lg N + r \lg d)$ identificar los documentos en donde hay ocurrencias de s.

Así que efectivamente concatenar todos los documentos logra reducir la complejidad de buscar el mismo patrón en cada uno por separado. La única desventaja de esta solución es el *espacio*, ¡usa mucho espacio!

Almacenar los d textos toma espacio $O(N \lg \sigma)$ pues solamente se necesitan $O(\lg \sigma)$ bits para codificar cada símbolo de Σ, pero el Suffix Array no almacena símbolos, almacena *índices* en el rango $[0..N]$, así que se necesitan $O(\lg N)$ bits para codificarlos lo que resulta en que el arreglo usa $O(N \lg N)$ bits de espacio. Esto involucra una sobrecarga del orden $O(\log_\sigma N)$ en espacio que puede ser restrictiva al tener un N grande.

A continuación, analizaremos una alternativa que es más consciente en el uso de espacio, el FM-index.

FM-index

Esta estructura fue introducida por Paolo Ferragina y Giovanni Manzini en el año 2000, es un *autoíndice* que soporta las operaciones `extract`, `count` y `locate` y hacen lo siguiente: recuperar el texto original, contar cuántas veces un patrón aparece en el texto y ubicar dónde están, respectivamente. Lo que diferencia al FM-index de otros métodos es que resuelve estas operaciones usando asintóticamente el mismo espacio que el texto comprimido y sin necesidad de descomprimir el texto completo, así, si tenemos textos masivos, no necesitamos el espacio del texto original ni el tiempo que demora descomprimirlo.

Al igual que en el Suffix Array agregaremos un caracter de menor valor lexicográfico que todos los símbolos en nuestro alfabeto (denotado por $), al final de nuestra string para diferenciar esta posición.

Para crear un FM-index se comienza con la *Transformada de Burrows-Wheeler* (BWT). La BWT reorganiza una secuencia de caracteres, de manera que los caracteres similares se agrupen, facilitando la identificación de patrones y la compresión de datos.

Primero se generan todas las rotaciones posibles de la secuencia original. Luego, se ordenan rotaciones de manera lexicográfica y se extrae la columna del ultimo caracter de cada rotación. Esta columna es la Transformada de Burrows-Wheeler, a la que llamaremos `L` (*Last*). Solo guardaremos `L` pero además nombraremos a la primera columna `F` (*First*), esta columna contiene a todos los caracteres ordenados lexicográficamente.

Es importante notar que solamente teniendo la BWT es posible recuperar el texto original, pero para no tener que descomprimir el texto completo nos ayudamos de una tabla `C[c]`, y de una función `Occ(c, k)`.

- `C[c]` es una tabla del tamaño del alfabeto que para cada caracter `c` en Σ, contiene el numero de ocurrencias de caracteres lexicográficamente menores en el texto. Es una representación equivalente a `F` pero comprimida.
- `Occ(c, k)` es una fucnión que obtiene el numero de ocurrencias de un caracter `c` hasta la posición `k` de `L`.

Así, el caracter `L[i]` se localiza en `F` en la posicion `LF[i]`, definida como:

```
LF(i) = C[L[i]] + Occ(L[i], i).
```

Esto lo llamaremos *LF-mapping*.

Para implementar `locate` se hacen sampleos sobre el texto y se guardan, algunas respuestas para evitar descomprimir mucho. Se puede hacer que `Occ` tome tiempo constante y para buscar un patrón, se pueden encontar sus ocurrencias en tiempo $O(m)$ empezando desde su último caracter e ir aplicando el LF mapping para ir acotando el intervalo hasta llegar al principio del patrón en donde habremos ubicado todas las ocurrencias. (Ferragina, Manzini (2000)).

Si esto se hace sin compresión, entonces no tiene mucha gracia haber usado un FM-index, así que para reducir el espacio usado en mantener la estructura, se crea un Wavelet Tree de la BWT en donde se puede comprimir cada bitvector para reducir aún más el espacio. Esto se hace con técnicas estándar de estructuras de datos compactas.

En total, usando estas técnicas, se obtiene que obtener las r ocurrencias de s en el texto toma tiempo $O(m + r(\lg n)^\epsilon)$, con $\epsilon \in]0, 1[$ una constante arbitraria que explica el *trade-off* entre tiempo y espacio, esta se escoge en la inicialización de la estructura y dice cuántos bloques hay para calcular los samples en la rutina; además se utilizan a lo más $5nH_k(t) + O\left(\frac{n}{(\lg n)^\epsilon}\right)$ bits para cualquier $k \geqslant 0$. (*Indexing Compressed Text*. Ferragina y Manzini (2005)).

Para identificar a qué documentos corresponde cada ocurrencia se hace lo mismo que se describió en el análisis del Suffix Array, que agrega complejidad $O(r \lg d)$. Así que la complejidad total de `locate` para varios documentos es dada por $O(m + r((\lg n)^\epsilon + \lg d))$.

Implementaciones

Suffix Array

Se implementó la siguiente interfaz para el Suffix Array, tiene tres atributos: `text`, `SA`, `doc_start` que corresponden a los textos concatenados, el Suffix Array como tal y los índices en donde cada texto aparece en los concatenados.

```
class suffix_array
{
    private:
        std::string text;
        sdsl::int_vector<> SA;
        std::vector<uint64_t> doc_start;

    public:
        suffix_array(const std::vector<std::string> &docs);
        std::vector<uint64_t> locate(const std::string &s);
        double size(void);
};
```

Para construir el SA se recibe una lista de documentos y estos se separan por el caracter fin de texto, `ETX`, que tiene el código ASCII 3 y luego se usa la SDSL para calcular el arreglo usando la implementación de Yuta Mori.

```
suffix_array::suffix_array(const std::vector<std::string> &docs)
{
    uint64_t N, D;
    char ETX = 3;
    D = docs.size();
    doc_start.resize(D);
    text = "";
    for (uint64_t i = 0; i < D; i++) {
        doc_start[i] = text.length();
        text += docs[i] + ETX;
    }

    N = text.length();
    SA.width(sdsl::bits::hi(N) + 1);
    SA.resize(N);
    sdsl::algorithm::calculate_sa((unsigned char *) text.c_str(), N, SA);
}
```

Ahora, para implementar la búsqueda se realizó lo siguiente:

```
std::vector<uint64_t> suffix_array::locate(const std::string &s)
{
    uint64_t N, lo, mi, hi, i;
    std::unordered_set<uint64_t> doc_matches;
    std::vector<uint64_t> matches;
    N = text.length();

    lo = 0;
    hi = N;
    while (lo < hi) {
        mi = lo + (hi - lo) / 2;
        if (text.substr(SA[mi]) < s)
            lo = mi + 1;
        else
            hi = mi;
    }
    i = lo;

    hi = N;
    while (lo < hi) {
        mi = lo + (hi - lo) / 2;
        if (text.substr(SA[mi]).starts_with(s))
            lo = mi + 1;
        else
            hi = mi;
    }

    for ( ; i < hi; i++) {
        auto j = std::upper_bound(doc_start.begin(), doc_start.end(), SA[i])
              - doc_start.begin();
        doc_matches.insert(j - 1);
    }

    for (uint64_t d : doc_matches)
        matches.push_back(d);

    return matches;
}
```

Retorna los documentos en donde aparece s no necesariamente ordenados.

FM-index

Esta es la interfaz que se implementó para el FM-index. Se decidió usar un Wavelet Tree `wt_int` en donde cada bitvector sea del tipo `rrr_vector` que corresponden a bitvectors comprimidos a su mínima entropía. Esto hace que se use muy poco espacio aunque tome más tiempo construirse.

```
class FM_index
{
    private:
        sdsl::csa_wt<sdsl::wt_int<sdsl::rrr_vector<> > > fm_index;
        std::vector<uint64_t> doc_start;

    public:
        FM_index(const std::vector<std::string> &docs);
        std::vector<uint64_t> locate(const std::string &s);
        double size(void);
};
```

Para la construcción se crea un archivo `.concatenated.txt` que almacena la concatenación de los documentos y luego se crea el índice usando la SDSL.

```
FM_index::FM_index(const std::vector<std::string> &docs) {
    uint64_t N, D;
    char ETX = 3;
    D = docs.size();
    doc_start.resize(D);
    std::string text;
    for (uint64_t i = 0; i < D; i++) {
        doc_start[i] = text.length();
        text += docs[i] + ETX;
    }
    std::ofstream t(".concatenated.txt");
    if (t.is_open()) {
        t << text;
        t.close();
    } else {
        std::cerr << "Could not open file" << std::endl;
        std::exit(EXIT_FAILURE);
    }
    sdsl::construct(fm_index, ".concatenated.txt", 1);
    std::remove(".concatenated.txt");
}
```

Para identificar a cuáles documentos pertenece el patrón se llama a `sdsl::locate` que implementa el algoritmo descrito en la sección de análisis teórico y luego se aplica la misma técnica usada en el Suffix Array para retornar una lista no necesariamente ordenada de los documentos con `s`.

```
std::vector<uint64_t> FM_index::locate(const std::string &s)
{
    uint64_t i, j;
    std::unordered_set<uint64_t> doc_matches;
    std::vector<uint64_t> matches;
    auto occs = sdsl::locate(fm_index, s.begin(), s.end());

    for (i = 0; i < occs.size(); i++) {
        auto j = std::upper_bound(doc_start.begin(), doc_start.end(), occs[i])
              - doc_start.begin();
        doc_matches.insert(j - 1);
    }

    for (uint64_t d : doc_matches)
        matches.push_back(d);

    return matches;
}
```

Resultados experimentales

Para realizar los experimentos, se utilizaron los textos disponibles en el *Pizza&Chili* corpus, para el mismo texto se buscan siempre los mismos patrones para lograr homogeneidad entre las pruebas para cada tipo de solución probada. Todos los experimentos se hacen en archivos que se llaman `uhr` (reloj, en alemán) estos escriben sus resultados en un `.csv`. Así podemos obtener más datos y apreciar exactamente qué es lo que está ocurriendo.

Todas las pruebas se realizarán en el servidor `chome` que nos provee la universidad, es un entorno de hardware muy poderoso y tiene muchísima memoria (¡más de 256 Gb de RAM!). Además, tiene 80 núcleos Intel Xeon Gold 5320T a 2.3 GHz. Para reducir los efectos de ruido se harán 64 ejecuciones de todo lo probado y se reportará la media y la desviación estándar.

Para compilar los archivos usar la sentencia:

```
g++ -std=c++20 -Wall -O3 uhr.cpp fm_index.cpp suffix_array.cpp -o
            uhr -lsdsl -ldivsufsort -ldivsufsort64
```

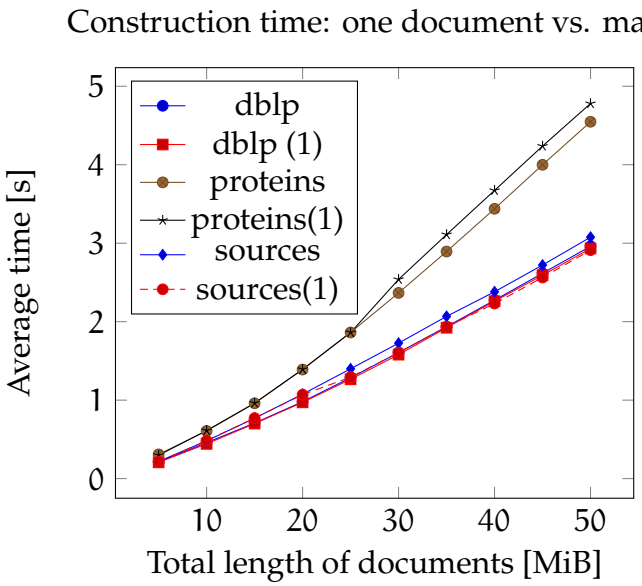
Lo que hacen los archivos en donde se prueba el código es cargar al archivo en un buffer y luego convertirlo en una `std::string`, de esta manera funciona con las interfaces implementadas en este proyecto. Este tiempo no se considera para el tiempo de ejecución.

Se probarán el efecto del número de documentos y largo total para la creación de las estructuras, efecto del número de ocurrencias y efecto del número de documentos en los tiempos de búsqueda. No se probará el efecto del largo del patrón, pues en el proyecto 2 ya se discutió su relación.

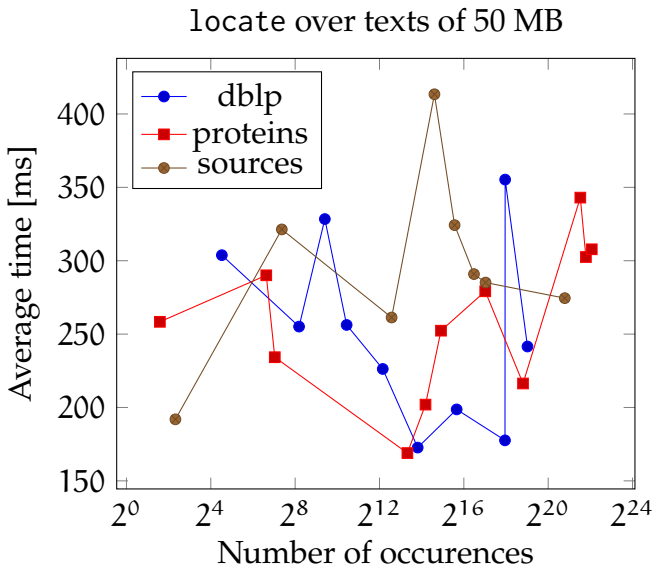
Los datasets, para las pruebas de `locate` se consideran los dataset más significativos, 10 documentos de 5 MB cada uno, para que el largo total del texto sea 50 MB. Para probar el efecto del número de documentos, se busca un patrón que aparezca mucho, de los tres textos usados, el patrón que menos aparece es alrededor de 500000 veces (en dblp) y el que más aparece es más de 4000000 veces (en proteins).

Además, se hace una prueba para medir el espacio en MiB de cada estructura. Para ver detalles de los experimentos, consultar los archivos `uhr` adjuntos. Para ver los datos crudos de los experimentos, junto con las varianzas, ver el directorio `experimental_data` adjunto.

Suffix Array

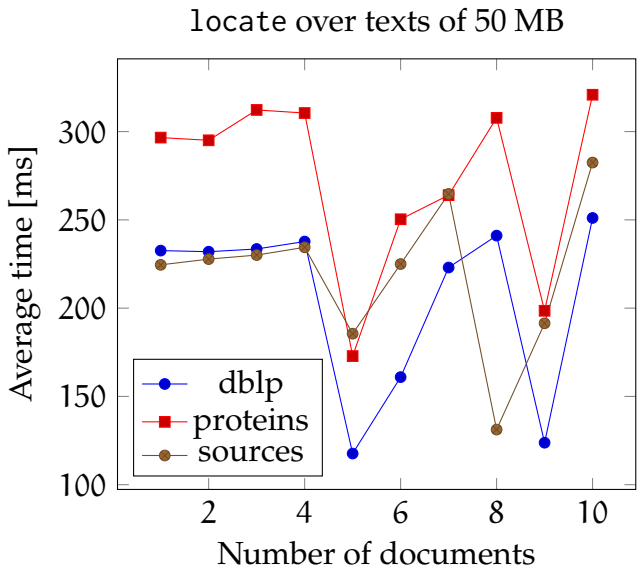


Aquí podemos ver que, en efecto el tiempo de construcción es $O(N \lg N)$ y que no depende de cuántos documentos se usen. Las curvas marcadas con (1) son creadas con los mismos documentos, pero concatenados antes de ser pasados al constructor del Suffix Array, para que efectivamente sea uno solo. Vemos que las curvas son muy parecidas, así que no hay mucha diferencia en este caso.



Aquí observamos el efecto del número de ocurrencias sobre el tiempo de `locate`, se supone que la complejidad de este método es $O(m \lg N + r \lg d)$, así que como se está buscando sobre textos del mismo largo N y todos los patrones que se buscaron son muy pequeños ($m < 16$) y siempre $d = 10$ se supone que debe haber una tendencia *lineal* con respecto a r. Esto porque nuestra implementación es tal como se explicó en el análisis teórico, $O(m \lg N)$ para encontrar los matchs, luego tiempo $O(r \lg d)$ para identificar el documento de cada uno (más $O(r$ trabajo extra para ver si hay duplicados con un `unordered_map`) y luego $O(d') = O(r)$ para crear la lista de los documentos con match. Pero a pesar de coincidir con la teoría los resultados experimentales no muestran la tendencia lineal.

Suponemos que esto se debe a que para $r = o(\frac{m \lg N}{\lg d})$ tenemos que la búsqueda binaria domina el tiempo, y esto ocurre para algunos r pequeños en nuestros experimentos. Pero hay algunos r que no son pequeños, así que esto no explica todo. Suponemos que como N es grande (50 MB) al hacer la búsqueda binaria hay muchos cache misses, pues evidentemente la búsqueda binaria no es muy amigable con el cache. Así que creemos que la sobrecarga de tener que acceder mucho a la memoria es lo que domina el tiempo y así se pierde la relación entre r y el tiempo de `locate` en la práctica.



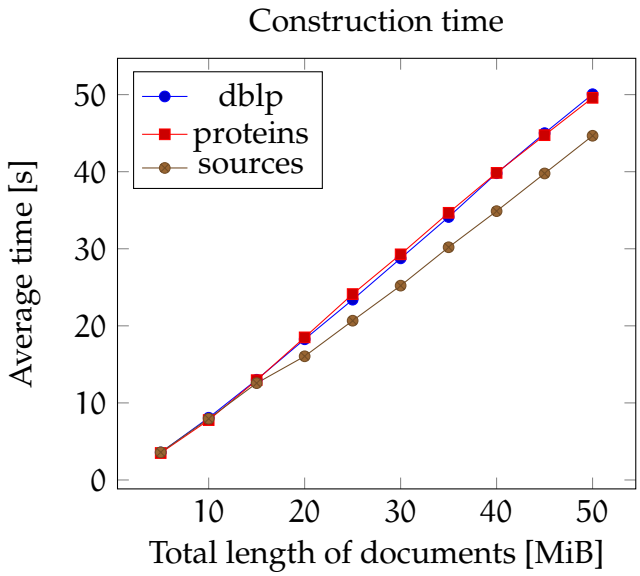
Como se mencionó anteriormente, nuestra función locate toma tiempo $O(m \lg N + r \lg d)$, y en este experimento observamos qué ocurre con los tiempos al variar el tamaño de d, con un N fijo, por lo que, al aumentar, gráficamente deberíamos observar un crecimiento logarítmico en las rectas, lo cual no se esta cumpliendo.

Una razón por la cual puede estar sucediendo esto es que d no crece significativamente, puesto que solo varia en el intervalo $0 < d \leqslant 10$, lo que no repercute tanto en los resultados.

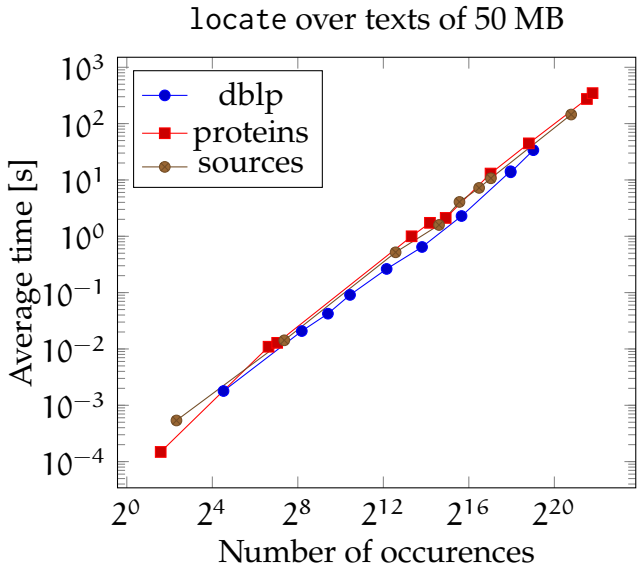
El experimento consistió en tomar el tiempo que tardaron los 10 documentos, para luego hacer una concatenación entre el primero y el segundo, y obtener $d = 9$, se tomó el tiempo nuevamente, seguido, se hizo la concatenación del nuevo segundo con el tercero, logrando tener $d = 8$, y así sucesivamente. De esta forma, siempre tenemos el mismo texto, pero en diferentes documentos, y de la misma forma r, m y N se mantienen constantes en todas las variaciones de d.

A pesar de que los resultados no concuerdan con el análisis teórico, podemos observar que Suffix Array, tuvo un comportamiento similar en todos los datasets usados, esto refiriéndonos, por ejemplo, que en $d = 5$ se obtuvo una reducción de tiempo significativa con respecto a $d = 4$ o también que en $d = 10$ los resultados aumentaron de manera radical con respecto a $d = 9$.

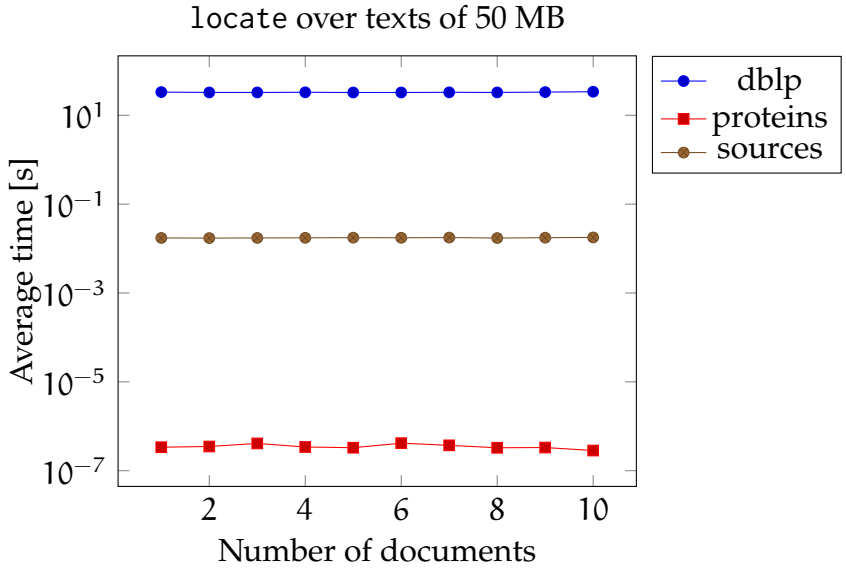
FM-index



Vemos una relación lineal entre el tamaño de los documentos concatenados y el tiempo de construcción. De hecho es muy interesante porque es casi exacta que 1 MiB corresponde a 1 s en el gráfico.

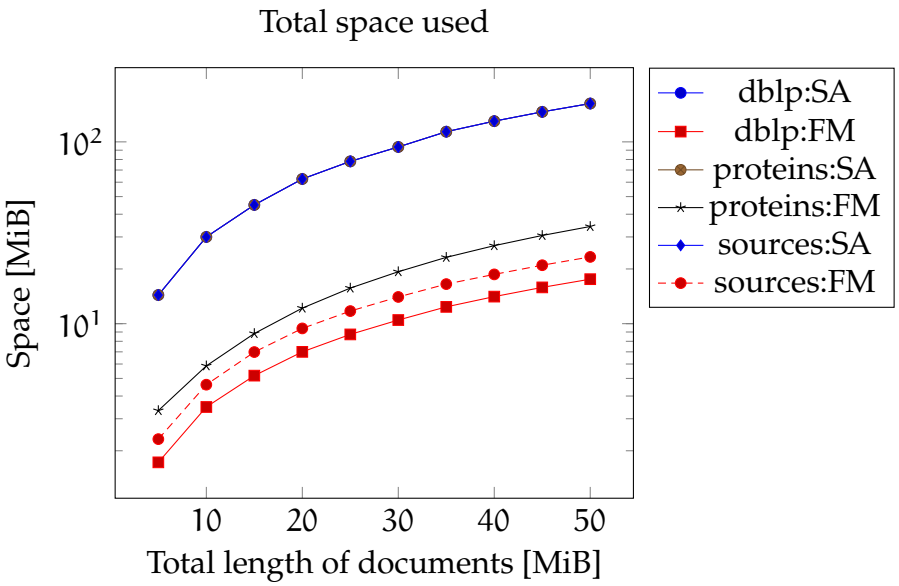


Aquí sí tenemos una coincidencia con la complejidad $O(m + r((\lg N)^\epsilon + \lg d))$ descrita anteriormente. Como el resto de variables es constante, tenemos una relación lineal entre el tiempo de búsqueda y el número de ocurrencias del patrón. Esto sustenta nuestra hipótesis de por qué para esta misma prueba no vimos el mismo resultado con el Suffix Array, habíamos dicho que era una cuestión e jerarquía de memoria, y ahora como estamos siendo conscientes con el espacio con el índice, ya no tenemos esa sobrecarga.



Como en el Suffix Array, no hay variaciones significativas porque $d \in [1..10]$.

Análisis espacial



Aquí podemos ver que el FM-index usa alrededor de *un orden de magnitud* menos espacio que el Suffix Array. Además, se puede verificar que el espacio del FM-index efectivamente depende del texto usado, pues para el texto con mayor entropía, proteins, se usó más espacio con respecto al más estructurado y de menor entropía, dblp. El Suffix Array siempre usa el mismo espacio porque no utiliza compresión y como todos los textos son del mismo tamaño en cada datapoint, tenemos que coinciden en espacio.

Conclusiones

Luego de haber implementado dos soluciones para la detección de patrones en múltiples documentos observamos, como hemos aprendido durante el curso y el semestre, que no hay una sola solución que sea la mejor para *todos* los casos de uso. Hay que ver el contexto y el entorno en donde se van a aplicar nuestras soluciones y discernir cuál es la mejor.

Las diferencias más significativas ocurren en el espacio utilizado y el tiempo de construcción. El Suffix Array siempre gasta más espacio que el texto original, y el FM-index usa menos pues utiliza técnicas de compresión. De hecho la diferencia para un texto de 50 MiB es que el SA usa alrededor de 10 veces más espacio que el índice. Así que si lo que se busca es ser eficiente en espacio el FM-index es la estructura a usar (notar que a veces es necesario priorizar el espacio porque se pueden trabajar con textos de GiB de tamaño, así que un SA no es una opción). Claro, este *trade-off* no es gratis el factor de 10 se vuelve a ver, pero ahora en el tiempo de construcción —toma 10 veces más tiempo crear un FM-index que un Suffix Array—.

En términos de búsquedas, tenemos más dispersión en los resultados. El Suffix Array se mostró indiferente ante variaciones del número de ocurrencias del patrón y del número de documentos y se demoró aproximadamente 250 ms para buscar cualquier patrón (corto). En cambio, el FM-index se demoró casi 10 s en buscar patrones que aparecen más de un millón de veces en textos como dblp, pero como 1 µs para otros como proteins. Esto seguramente se debe al factor ϵ que se dejó a la SDSL que lo escogiera automáticamente según la entropía empírica del texto. Así que si se puede costear el espacio que involucra el Suffix Array y no se conoce nada *a priori* de los documentos, se tiene un buen rendimiento y proponemos que esto se use. Por otra parte, si se sabe que un patrón va a aparecer pocas veces, es bueno buscar con el índice porque `locate` es lineal en r y tuvimos resultados del orden de 1 ms para estas búsquedas.

Lo importante de los resultados anteriores es siempre tener presente los casos de uso de nuestros programas para poder realizar un análisis de frecuencias del tipo de operaciones realizadas y la naturaleza de los datos procesados. Así se pueden realizar experimentos como este y luego decidir qué solución usar.

Referencias

1. Ferragina, Paolo; Manzini, Giovanni (2000). *Opportunistic Data Structures with Applications*. Proceedings of the 41st Annual Symposium on Foundations of Computer Science. p. 390.

2. Ferragina, Paolo; Manzini, Giovanni (2005). *Indexing Compressed Text*. Journal of the ACM. **52** (4): 553.

3. Ferragina, Paolo; Navarro, Gonzalo (2005). *Pizza&Chili Corpus*. Disponible en `https://pizzachili.dcc.uchile.cl/index.html`

4. Li, Zhize; Li, Jian; Huo, Hongwei (2016). *Optimal in-place suffix sorting*. Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. Vol. 11147. Springer. pp. 268–284.

5. Manber, Udi; Myers, Gene (1990). *Suffix Arrays: a new method for on-line string searches*. First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327.

6. Mori, Y. `libdivsufsort`. Ver `https://github.com/y-256/libdivsufsort`

7. Nong, Ge; Zhang, Sen; Chan, Wai Hong (2009). *Linear Suffix Array construction by almost pure Induced-Sorting*. 2009 Data Compression Conference. p. 193.