PATTERN MATCHING
SUFFIX ARRAY VS DFA

Mario Garrido

Universidad de Chile
Departamento de Ciencias de la Computación
Santiago, Chile

ABSTRACT

En el presente trabajo se estudia el tiempo de construcción y búsqueda de patrones en texto de 2 estrategias: construcción de un Arreglo de sufijos más 2 búsquedas binarias para encontrar la primera y última aparición del patrón y construcción de un Autómata finito determinista más una lectura completa del texto para contar todas las ocurrencias del patrón. La cota teórica de construcción del Arreglo de sufijos, para el algoritmo utilizado, es $\mathcal{O}(n)$, donde n es el tamaño del texto, mientras que para el autómata es de $\mathcal{O}(|\Sigma|m)$, donde $|\Sigma|$ es el tamaño del alfabeto y m el largo del patrón a buscar.

1. INTRODUCCIÓN

Encontrar patrones en texto es una actividad vital en muchas actividades computacionales, bajo ese contexto se estudiarán 2 estrategias que permiten realizar esta tarea:

- Construir un *Suffix array* del texto, utilizando una variante del algoritmo descrito en[1], lo que tiene $\mathcal{O}(n)$ tiempo de construcción, y 2 búsquedas binarias que otorgan la primera y última ocurrencia de un *match* en el arreglo, esto nos permite saber, sin necesidad de realizar una búsqueda lineal $\mathcal{O}(n)$ determinar el número de ocurrencias del patrón. Estas búsquedas tienen $\mathcal{O}(m \log n)$.
- Construir un *DFA* que permite identificar el patrón buscado, lo que implica que por cada patrón distinto se debe confeccionar un autómata distinto, y recorrer todo el texto de entrada para contar el total de ocurrencias del patrón. La construcción del autómata es $\mathcal{O}(|\Sigma|m)$, mientras que la búsqueda en el texto es $\mathcal{O}(n)$.

1.1. Hipótesis

Las hipótesis a estudiar, cuyo origen se encuentra en las cotas teóricas, son las siguientes:

- $|\Sigma|$ debería afectar el tiempo de construcción del autómata y resultar irrelevante para el resto de las operaciones.
- n debería afectar el tiempo de búsqueda en el caso del autómata y el tiempo de construcción y búsqueda en el caso del arreglo.
- m debería afectar el tiempo de búsqueda en el caso del arreglo y el de construcción para el autómata.

Una interrogante no menor es si una estrategia es preferible a la otra de forma definitiva, o si en algun triplete $(|\Sigma|, n, m)$ alguna estrategia es preferible.

2. IMPLEMENTACIÓN

La implementación se desarrolló en el lenguaje *C++*, bajo el standard de *C++11*. La implementación se encuentra contenida en los archivos *experiments.cpp*, *pattern_matching.cpp*, *pattern_matching.h*, *utility.cpp* y *utility.h*.

2.1. Suffix Array y DFA

La presente sección describe aspectos relevantes sobre el contenido del archivo pattern_matching.cpp, donde se encuentran implementados los principal métodos de ambos algoritmos.

radix_sort: Este método permite ordenar los contenidos de un arreglo de entrada (*input_array*) en base a los valores contenidos en el arreglo *keys*, esta distinción es relevante ya que no es el arreglo de entrada quien provee los valores de las llaves directamente.

make_transition_function: Con este método se construye la función de transición del autómata, utilizando la variable *X* que permite reducir su tiempo de construcción considerablemente.

number_of_occurences_automata: Recorre el texto y cambia los estados en base a la función de transición. Cuando se alcanza el último estado se suma una ocurrencia.

number_of_occurences_suffix: Realiza las 2 búsquedas binarias para encontrar la primera y la última aparición del

patrón en el arreglo de sufijos.

suffix_array: Este método crea el arreglo de sufijos del texto suministrado (convertido a un arreglo de enteros) utilizando el algoritmo *DC3*. Para esto, divide los índices de los elementos del texto en 3 conjuntos:

1. α: Corresponde a todos los índices i tales que $i \equiv 0$ (mód 3).
2. β: Corresponde a todos los índices i tales que $i \equiv 1$ (mód 3).
3. γ: Corresponde a todos los índices i tales que $i \equiv 2$ (mód 3).

Cada uno de los índices representa una secuencia de 3 elementos del texto original $T_i := (t[i], t[i+1], t[i+2])$. Se ordenan lexicográficamente los índices de $R = T_\alpha T_\beta$, que corresponde a la concatenación de todos los tripletes de α y luego los de β. Este orden lexicográfico permite nombrar, en base al orden, cada uno de los tripletes T_i de α y β. Estos nombres se guardan en el arreglo R'. Lo que necesitamos, posteriormente, es conseguir un arreglo de sufijos y un arreglo inverso de sufijos, también llamado. El arreglo de sufijos dictará el orden en que se tendrán que intentar mezclar los elementos de R con los T_γ, mientras que el arreglo inverso de sufijos permitirá comparar lexicográficamente los tripletes de R. De aquí en adelante pueden pasar 2 cosas:

- Si R' tiene elementos repetidos (al menos 2 tripletes tienen el mismo nombre), entonces es necesario llamar *suffix_array* recursivamente para obtener el arreglo de sufijos de los nombres $SA_{R'}$ y, en base a ese arreglo de sufijos obtenido, se genera un arreglo inverso de sufijos $SA^{-1}_{R'}$ que nos entregará un orden lexicográfico para los tripletes originales, este arreglo inverso se guarda en R', por lo que se sobreescribe.
- Si R' no tiene elementos repetidos entonces corresponderá al arreglo inverso de sufijos para los tripletes originales, por lo que se construirá el arreglo de sufijos de R', SA'_R.

Al finalizar cualquiera de las 2 ramas contamos con un arreglo de sufijos y un arreglo inverso de sufijos. Ahora es necesario ordenar los tripletes correspondientes a los índices del conjunto γ, para hacer esto se crean los pares $(T[j], R'_{j+1})$, recordando que R' entrega la información que permite comparar lexicográficamente los tripletes de α y β, y se ordenan utilizando *radix_sort*. Los primeros $|\alpha|$ *slots* de R' entregan el *ranking* u orden de los T_α y los siguientes *slots* corresponden a los T_β. Una vez que se cuenta con $SA_{\alpha\beta}$ y SA_γ se mezclan con *merge*.

merge: Combina $SA_{\alpha\beta}$ con SA_γ. Extrae el primer elemento de ambos, que en el caso de γ corresponde al índice de un sufijo del texto original, pero en el caso de $\alpha\beta$ corresponde al índice de un sufijo del arreglo de nombres. Notamos que, por construcción, SA y SA^{-1} cumplen $SA[SA^{-1}[k] - 1] = k$ y $SA^{-1}[SA[K]] = k + 1$, lo cual es casi la definición formal, correspondiendo a una adaptación a los índices de *C++*. Con estas relaciones es posible determinar el índice del sufijo del texto original (*get_index*). Para realizar las comparaciones se utilizan las siguientes equivalencias:

- $T_i \leq T_j \Leftrightarrow (T[i], R'_{i+1}) \leq (T[j], R'_{j+1})$, si $i \in \alpha$, $j \in \gamma$.
- $T_i \leq T_j \Leftrightarrow (T[i], T[i+1], R'_{i+2}) \leq (T[j], T[j+1], R'_{j+2})$, si $i \in \beta$, $j \in \gamma$.

3. RESULTADOS EXPERIMENTALES

Los experimentos fueron realizados en un computador con procesador AMD Phenom II x4 955, corriendo a $\sim$ 3.2Ghz y con 26624 Megabytes de RAM, bajo el sistema operativo *Ubuntu* 14.04.

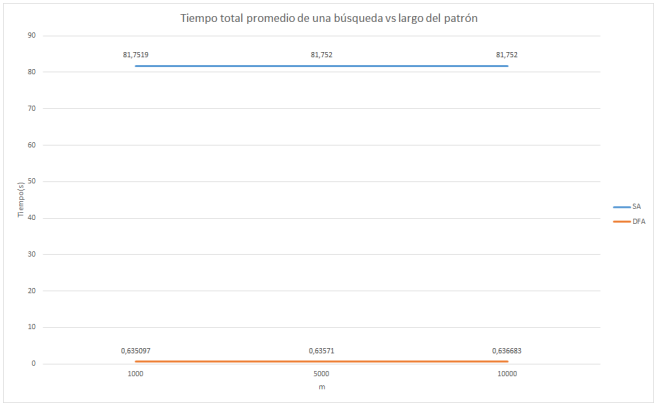

Fig. 1. Tiempo total promedio de una búsqueda.

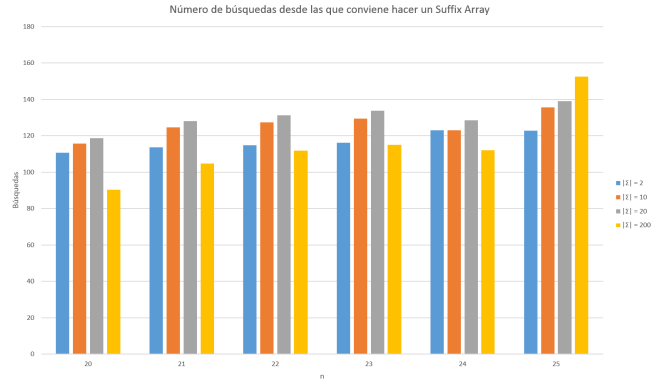

Fig. 2. Número de búsquedas en las que conviene hacer un suffix array.

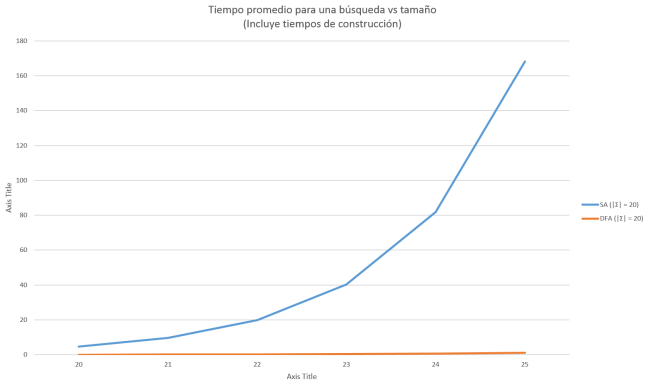

Fig. 3. Tiempo promedio de búsqueda vs tamaño - 45000 búsquedas.

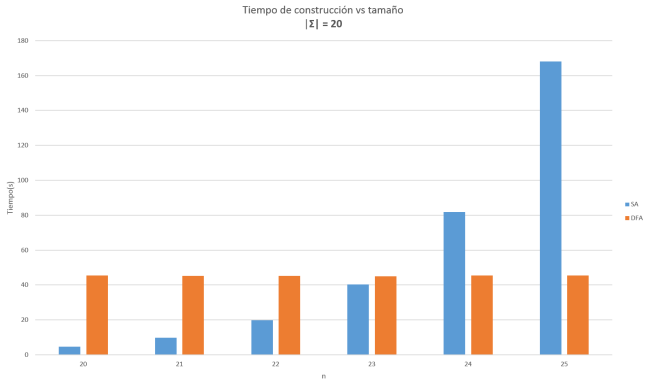

Fig. 4. Tiempo de construcción vs tamaño - 45000 construcciones para DFA.

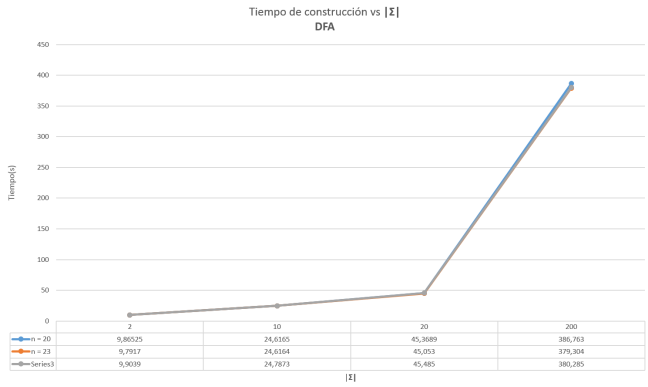

Fig. 5. Tiempo de construcción DFA - 45000 construcciones.

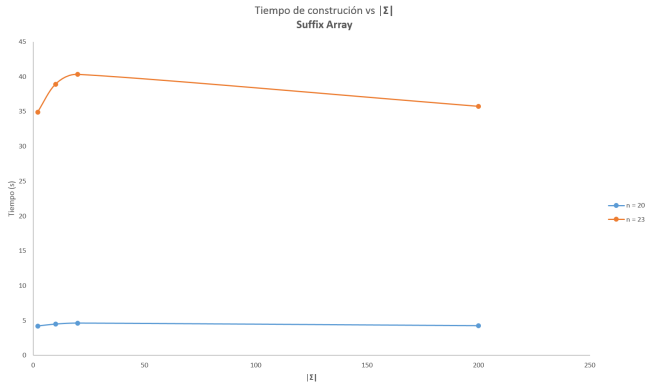

Fig. 6. Tiempo de construcción SA.

Como puede apreciarse en las figuras, el tiempo de construcción de un suffix array es mucho más grande que el tiempo de construcción de un DFA, pero al tener que construir un DFA para cada búsqueda pasa que, para las 45000 búsquedas efectuadas, construir un arreglo de sufijos toma menos tiempo. Sin embargo, notamos que existe un punto bajo el cual es más conveniente construir un DFA (fig 2), ya que aunque el tiempo de búsqueda en un DFA es órdenes de magnitud mayor a una búsqueda en el suffix array, para pocas búsquedas se logra compensar. Notamos como $|\Sigma|$ afecta tanto a DFA como SA, pero la tasa a la que afecta a SA es despreciable, incluso en la búsqueda. El resto de las hipótesis se confirma desde los gráficos.

4. REFERENCES

[1] J. Kärkkäinen, P. Sanders: Simple Linear Work Suffix Array Construction, Proceedings 30th International Conference on Automata, Languages and Programming, volumen 2719 de LNCS, páginas 943-955. Springer, 2003.