String matching

Curso Estructuras de datos avanzadas (EDA), período 2021-01

Profs. Erick Gómez Nieto y Eddy Peralta Araníbar

Alum. Saul Andersson Rojas Coila

Correo: saul.rojas@ucsp.edu.pe

Universidad Católica San Pablo

Esc. Profesional de Ciencia de la Computación

Agenda

- Descripción del proyecto
- Justificación de la estructura de datos a implementar
- Procesos a implementar
- Conclusiones
- Referencias

Descripción del proyecto

Objetivo

- Dado un conjunto de metadatos de papers obtenidos de la plataforma Arxiv (aprox. 1.7M), lo que queremos realizar es una búsqueda de palabras claves (nuestras entradas) en esos papers, posteriormente ordenar aquellos papers de acuerdo a una fórmula empleada para obtener los documentos con mejor similitud a la cadena solicitada.

Descripción del proyecto

Resultados esperados

- Obtener un ranking de los papers mejor relacionados a las palabras claves insertadas, el orden se realiza de mayor a menor, según la fórmula empleada TF-IDF[1].

File Name	Total Number of words	Total number of times Pattern occurred	Rank
Document Clustering-Wikipedia.pdf	1194	0	10
Web Mining-Wikipedia.pdf	2642	71	2
Data Mining-Wikipedia.pdf	6118	109	1
Social Media Mining-Wikipedia.pdf	2931	28	4
scikit-learn-org.pdf	1041	1	9
wikid-eu.pdf	956	10	6
Www-tcs-com.pdf	5680	66	3
Www-linguamatics-com.pdf	867	6	8
Searchbusinessanalytics-techtarget-com.pdf	978	12	5

1. InfoQ. 2021. *Understanding Similarity Scoring in Elasticsearch*. [online] Available at: <https://www.infoq.com/articles/similarity-scoring-elasticsearch/> [Accessed 13 May 2021].

Justificación de la estructura de datos a implementar

- ¿Qué estructura de datos implementaré?

1. Decidí escoger el árbol de sufijos (Suffix Tree), por su extensa capacidad en resolver problemas referentes a procesamiento de cadenas de texto.

2. Esta estructura es una modificación de la estructura Suffix Trie, pero tiene la ventaja de no almacenar tantos vértices, por lo que se obtiene una ventaja en la reserva de memoria.

3. La complejidad para realizar la búsqueda de nuestra palabra clave P en un árbol de Sufijos T en el peor de los casos es O(m + occ), donde m=|P| y occ es el número de ocurrencias de P en T, esto nos da una gran ventaja en cuanto a performance[1] (ver la imagen de la siguiente diapositiva) de nuestro algoritmo de búsqueda, esta complejidad aumentará un poco en mi implementación.

4. La estructura es sencilla de comprender, y no requeriría el uso de técnicas como conversión de palabras a puntos multidimensionales para las búsquedas a realizar, esta estructura fue destinada para lo que realizaremos.

1. Halim, S. and Halim, F., 2013. *Competitive programming 3*. 3rd ed. Singapore: Lulu.com, pp.249-253.

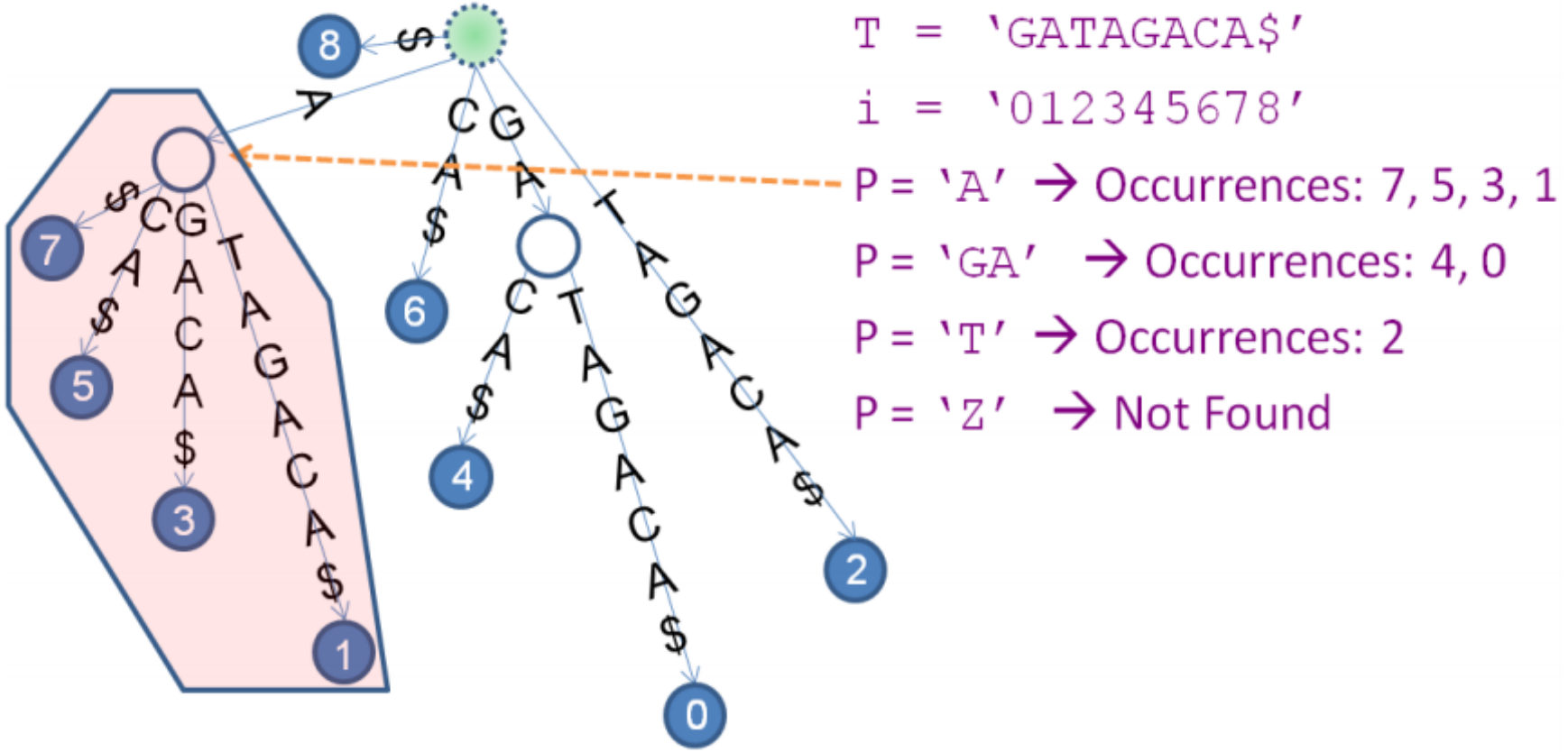


Figure 6.4: String Matching of T = 'GATAGACA$' with Various Pattern Strings

[1]

1. Halim, S. and Halim, F., 2013. *Competitive programming 3*. 3rd ed. Singapore: Lulu.com, pp.249-253.

Justificación de la estructura de datos a implementar

5. La implementación de la estructura contiene bastantes referencias, por lo que bastantes propuestas en su forma de implementarse ya se han hecho, incluso variaciones aplicadas a competencias de programación [2,3,4].

6. Por trabajarse con nodos en esta estructura, estos nodos pueden contener información extra sobre el conjunto de papers.

1. Ukkonen, E., 1995. On-line construction of suffix trees. *Algorithmica*, 14, pp.249--260.
2. Manbers, U. and Myers, G., 1993. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5), pp.935--948.
3. Vladu, A. and Negruseri, C., 2005. Suffix arrays--a programming contest approach. *GInfo*, 15(7).

Procesos a implementar

8

Vista general del proceso

Conjunto de datos

Formateo de datos, y selección de metadata

Construcción de la estructura de datos e indizado de los datos

Consultas y organización de los resultados, aplicación de fórmulas

Muestra de los resultados

{'id': '0704.0001',

'submitter': 'Pavel Nadolsky',

'authors': "C. Bal\\'azs, E. L. Berger, P. M. Nadolsky, C.-P. Yuan",

'title': 'Calculation of prompt diphoton production cross sections at Tevatron and\n LHC energies',

'comments': '37 pages, 15 figures; published version',

'journal-ref': 'Phys.Rev.D76:013009,2007',

'doi': '10.1103/PhysRevD.76.013009',

'report-no': 'ANL-HEP-PR-07-12',

'categories': 'hep-ph',

'license': None,

'abstract': ' A fully differential calculation in perturbative quantum chromodynamics is\npresented for the production of massive photon pairs at hadron colliders. All\nnext-to-leading order perturbative contributions from quark-antiquark,\ngluon-(anti)quark, and gluon-gluon subprocesses are included, as well as\nall-orders resummation of initial-state gluon radiation valid at\nnext-to-next-to-leading logarithmic accuracy. The region of phase space is\nspecified in which the calculation is most reliable. Good agreement is\ndemonstrated with data from the Fermilab Tevatron, and predictions are made for\nmore detailed tests with CDF and DO data. Predictions are shown for\ndistributions of diphoton pairs produced at the energy of the Large Hadron\nCollider (LHC). Distributions of the diphoton pairs from the decay of a Higgs\nboson are contrasted with those produced from QCD processes at the LHC, showing\nthat enhanced sensitivity to the signal can be obtained with judicious\nselection of events.\n',

'versions': [{'version': 'v1', 'created': 'Mon, 2 Apr 2007 19:18:42 GMT'},

 {'version': 'v2', 'created': 'Tue, 24 Jul 2007 20:10:27 GMT'}],

Formateo de los datos

- Los metadatos DOI e id, quedan intactos, estos serán usados únicamente con el objetivo obtener acceso a los documentos al momento de presentar los resultados.
- Los metadatos "Title" y "Abstract" si serán formateados con el lenguaje de programación Python.
- Lo que realizaremos en el formateo se especifica en el algoritmo [Formateo].

Algoritmo Formateo: Formatea cada una de las entradas del conjunto de datos.

F1. Concatenamos los campos title y abstract del documento. Todas las letras serán convertidas a minúsculas.

F2. Solo se conservarán los siguientes caracteres `[a-z],[0-9]` y "-".

F3. Caracteres especiales que empiecen con \ o \\ también serán eliminados.

F4. Se eliminarán los artículos 'a','an' y 'the'; también las conjunciones 'for','and','nor','but','or','yet','so'; también abreviaturas 'e.g.','i.e.','et al.','fig.','ref.','Eq.','Sect.','Ch.'; también los pronombres 'i','you','we','he','she','it','they','me','us','her','his','him','them','my','our','your','her','his','their','myself','yourself','herself','himself','itself','ourselves','yourselves','themselves', etc.

F5. Escribimos las cadenas modificadas en un archivo txt, con el siguiente formato: *id doi cadena_transformada*

En caso que un documento no tenga doi, le igualamos a "nulo"

Algoritmo suffix tree

- Usaremos el algoritmo del científico Esko Ukkonen de complejidad lineal[1] para la construcción del suffix tree.
- Para la construcción del árbol generalizado usaremos el enfoque simple de concatenar las cadenas de un conjunto S = {S1, S2, ..., SN} con un símbolo "$" terminal al final, s1$s2$s3...sn$.

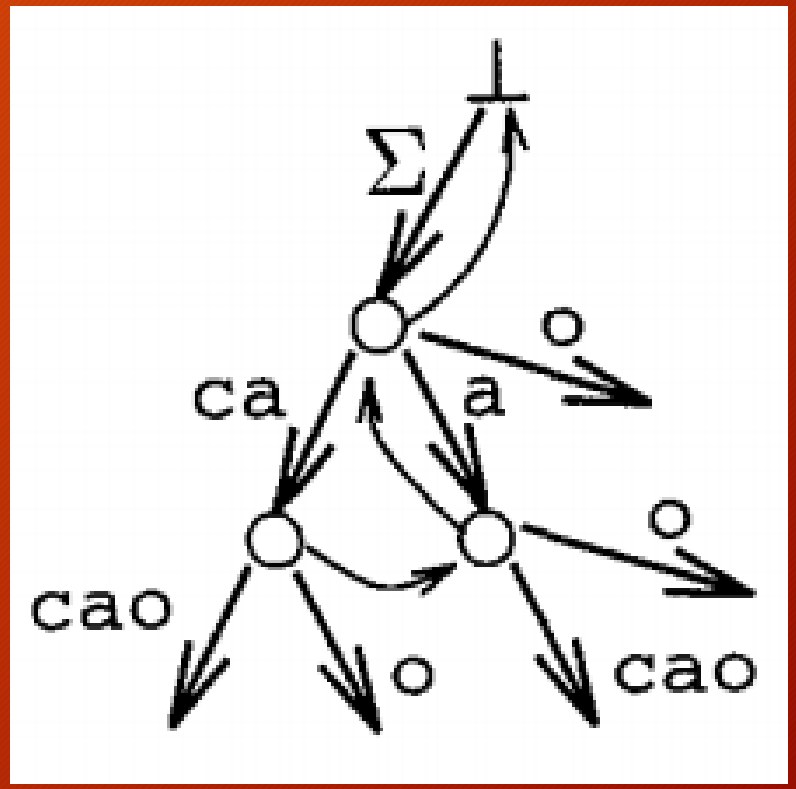
1. Ukkonen, E., 1995. On-line construction of suffix trees. *Algorithmica*, 14, pp.249--260.

Teoría básica del algoritmo lineal

La raíz posee un estado auxiliar, que sirve para diferenciar explícitamente entre sufijos vacíos y no vacíos o entre la raíz y otros estados, este estado auxiliar se representa por el símbolo $\perp$.

El árbol de sufijos posee un conjunto Q' de *sufijos explícitos* que poseen dos tipos de estados (nodos), *branching states* (estados que poseen dos nodos hijos como mínimo, por definición la raíz está dentro de este subconjunto), y todas las hojas (estados en donde no hay transiciones).

También se posee la función sufijo (que ayudará en la construcción del árbol) solo definida para los *branching states* $\underline{x} \neq root$ como $f'(\underline{x})=\underline{y}$ donde y es un *branching state tal que* $x=ay$.



1. Ukkonen, E., 1995. On-line construction of suffix trees. *Algorithmica*, 14, pp.249--260.

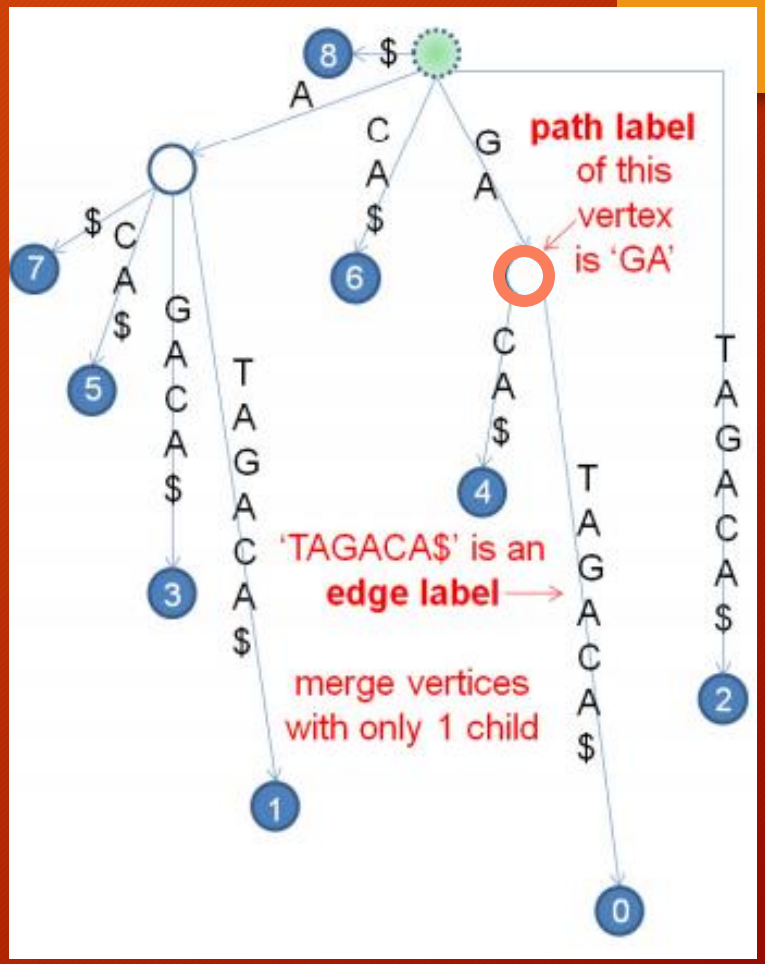
Un nodo es representado por una transición de un estado (nodo) a otro, esta transición contiene una cadena w que es subcadena de la general W, p.ej. la transición (root, "GA") representa el estado (nodo) delineado en rojo.

Para ahorrar espacio la cadena w se expresa con dos punteros (k,p) que son índices en la cadena general W, donde w = t[k]...t[p], p.ej. la transición (root, "GA") es expresada como (root, (1,2)).

W = G A T A G A C A.

1 2 3 4 5 6 7 8

Una hoja es de la forma (s, (p, ∞)), por ejemplo la hoja 2 se expresa como (root, (3, ∞)). Una transición (s, (k,p)) es una *a-transition* si t_k=*a*, donde *a* pertenece al alfabeto Σ .



1. Halim, S. and Halim, F., 2013. *Competitive programming 3*. 3rd ed. Singapore: Lulu.com, pp.249-253.

Bosquejo de funciones

La forma de contruir un árbol de sufijos es con el siguiente algoritmo mostrado en el artículo, este no sufrirá modificaciones, puesto que el problema es string matching, el algoritmo a implementar se hará en la búsqueda de la palabra:

Algorithm 2. Construction of $STree(T)$ for string $T = t_1 t_2 \cdots \#$ in alphabet $\Sigma = \{t_{-1}, \ldots, t_{-m}\}$; $\#$ is the end marker not appearing elsewhere in T.

1. create states *root* and $\perp$;
2. **for** $j \leftarrow 1, \ldots, m$ **do** create transition $g'(\perp, (-j, -j)) = root$;
3. create suffix link $f'(root) = \perp$;
4. $s \leftarrow root$; $k \leftarrow 1$; $i \leftarrow 0$;
5. **while** $t_{i+1} \neq \#$ **do**
6. $\quad i \leftarrow i + 1$;
7. $\quad (s, k) \leftarrow update(s, (k, i))$;
8. $\quad (s, k) \leftarrow canonize(s, (k, i))$.

Proceso UPDATE: transforma un árbol STree(Ti-1) en STree(Ti) insertando el carácter t[i] de la cadena en los estados que correspondan, la entrada es el estado (s,(p,i)) donde se iniciarán los respectivos recorridos e inserciones por el árbol.

procedure *update*$(s, (k, i))$:
    $(s, (k, i-1))$ is the canonical reference pair for the active point;
1. $oldr \leftarrow root$; $(end\text{-}point, r) \leftarrow test\text{-}and\text{-}split(s, (k, i-1), t_i)$;
2. **while not**$(end\text{-}point)$ **do**
3. create new transition $g'(r, (i, \infty)) = r'$ where r' is a new state;
4. **if** $oldr \neq root$ **then** create new suffix link $f'(oldr) = r$;
5. $oldr \leftarrow r$;
6. $(s, k) \leftarrow canonize(f'(s), (k, i-1))$;
7. $(end\text{-}point, r) \leftarrow test\text{-}and\text{-}split(s, (k, i-1), t_i)$;
8. **if** $oldr \neq root$ **then** create new suffix link $f'(oldr) = s$;
9. **return** (s, k).

Proceso TEST-AND-SPLIT: verifica si un estado con una referencia canónica (s, (k,p)) es un punto final(es decir un estado en STree(T^{i-1}) que debería tener una transición t_i), si (s,(k,p)) no fuese un estado explícito, entonces es hecho explícito dividiéndolo en una nueva transición.

procedure *test-and-split*$(s, (k, p), t)$:

1. **if** $k \leq p$ **then**
2. let $g'(s, (k', p')) = s'$ be the t_k-transition from s;
3. **if** $t = t_{k'+p-k+1}$ **then return(true,** s**)**
4. **else**
5. replace the t_k-transition above by transitions
   $g'(s, (k', k' + p - k)) = r$ and $g'(r, (k' + p - k + 1, p')) = s'$
   where r is a new state;
6. **return(false,** r**)**
7. **else**
8. **if** there is no t-transition from s **then return(false,** s**)**
9. **else return(true,** s**).**

Proceso CANONIZE: Dada un par de referencia (s,(k,p)) hacia un estado r. la función canonize devuelve el par de referencia canónico del estado r, es decir el ancestro (que es un estado explícito) mas cercano de r, si r fuese un estado explícito la función retornaría r.

procedure *canonize*$(s, (k, p))$;
1. **if** $p < k$ **then return** (s, k)
2. **else**
3. find the t_k-transition $g'(s, (k', p')) = s'$ from s;
4. **while** $p' - k' \leq p - k$ **do**
5. $k \leftarrow k + p' - k' + 1$;
6. $s \leftarrow s'$;
7. **if** $k \leq p$ **then** find the t_k-transition $g'(s, (k', p')) = s'$ from s;
8. **return** (s, k).

Ejemplo de construcción del árbol de sufijos

c a c a o
1 2 3 4 5

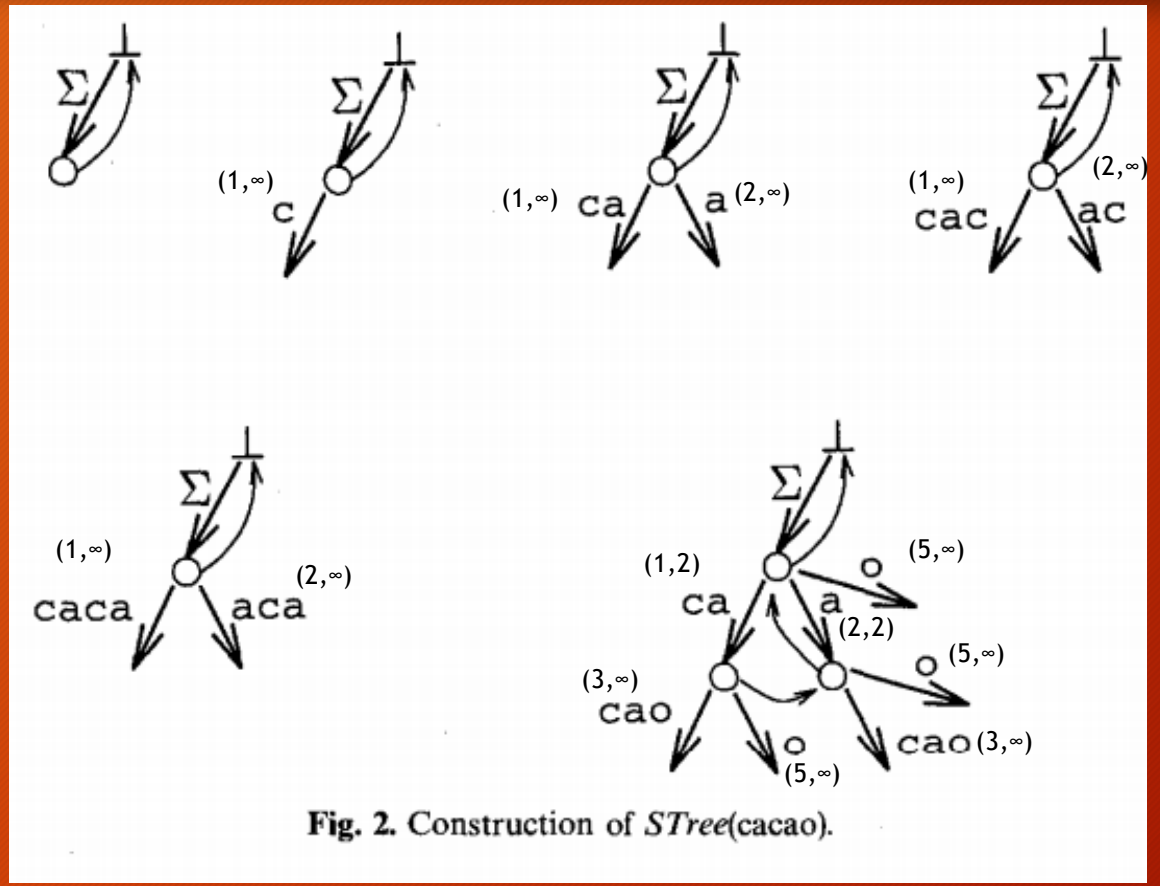


Fig. 2. Construction of *STree*(cacao).

1. Ukkonen, E., 1995. On-line construction of suffix trees. *Algorithmica*, 14, pp.249--260.

Proceso STRING-MATCH: dada una cadena **ss** como solicitud, la función buscará en el árbol de sufijos todos aquellos documentos que contengan ss, para ello antes la palabra pasará por un proceso de formateo similar al del algoritmo FORMATEO.


```
1:  procedure STRING-MATCH(ss)
2:      ss ← format_string(ss)
3:      r ← root; i ← 0;
4:      while i ≤ ss.length() do
5:          c ← ss_i
6:          let g'(r,(k,p)) = r' be the t_c-transition from r;
7:          if r' does not exist then
8:              return
9:          else
10:             j ← k
11:             while j ≤ p and i ≤ ss.length() and t_j = ss_i do
12:                 j ← j + 1
13:                 i ← i + 1
14:             if i > ss.length() then
15:                 results ← get_results(r, r')
16:                 show_results(results)
17:                 return
18:             if t_j ≠ ss_i then
19:                 return
20:         r ← r'
21:     return
```

Proceso GET-RESULTS: dado un estado y los índices que lo apuntan, recorre todos los estados hojas que encuentre, por cada hoja encontrada obtiene el id y doi del documento al cual le pertenece dicha hoja, por cada documento contabiliza la cantidad de veces que aparece la palabra dada.

Algorithm 2 get-results

```
    procedure GET-RESULTS(r, (k, p))
 2:     results ← {}; // un mapa de la forma ['string_id'] — #ocurrencias
        if r is a leaf then
 4:         create results[get-id(k-1)]—1 or add 1 to results[get-id(k-1)]
           return results
 6:     dfs ← r // dfs es una pila conteniendo r
        while dfs is not empty do
 8:         top ← dfs.top()
           dfs.pop()
10:        let (top, (k'_i, p'_i)) — r'_i be the t_{k'_i}-transition of top for i — 1, 2, ...N
           where N is the number of its children states
12:        if r'_i is a leaf then
               create results[get-id(k'_i − 1)]—1 or add 1 to results[get-id(k'_i − 1)]
14:        else
               dfs.push(r'_i)
16:    return results
```

Proceso GET-ID: dado un índice k, retornaremos el id de la cadena a la cual pertenece aquel índice, también su doi, para ello realizaremos una búsqueda binaria sobre un arreglo general que se construye al inicio, cuando empezamos a concatenar todas las entradas dadas.

p.ej. dados 5 cadenas s1:"holaatodos", s2:"holaami", s3:"holaati", s4:"adiosati", s5:"adiosami", tras concatenar todas las entradas terminamos con el string:

"holaatodosholaamiholaatiadiosatiadiosami"

Cada una de estas cadenas contiene un par de índices que delimita su inicio y final: s1:(1,10), s2:(11,17), s3:(18,24), s4:(25,32), s5:(33,40), lo que realizaremos será una búsqueda binaria sobre los índices iniciales de cada cadena:

[1,11,18,25,33], luego de esto, simplemente verificamos a qué par pertenece y sabremos el identificador de cadena, una posible forma de esta estructura es [(idx_l, idx_r, string_id, string_doi)].

Proceso SHOW-RESULTS: dado un mapa de resultados con las respectivas ocurrencias de alguna palabra dada (['id_string']:#ocurrencias), mostraremos en la terminal un ranking ordenado descendentemente por el puntaje obtenido de aplicar la fórmula TF-IDF [6] a cada documento dado.

Algorithm 3 show-results

```
    procedure SHOW-RESULTS(results)
 2:     if results is empty then
            return
 4:
        df ← results.size()
 6:     N ← total_number_of_processed_docs
        for doc in results do
 8:         tf_idf ← doc.occ × log(N/df)
            add field tf_idf field to doc
10:     sort results by tf_idf field
        print("Results of search")
12:     print("Score document_prev document_doi document_pdf")
        for doc in sorted results do
14:         print("doc.tf_idf","https://arxiv.org/abs/doc.id",
            "https://www.doi.org/doc.doi"or"nulo",
16:         "https://arxiv.org/pdf/doc.id")
        return
```

Conclusiones

- El algoritmo de búsqueda propuesto (STRING-MATCH) posee una complejidad de O(m + occ*log(occ)), donde m es el tamaño de la consulta y occ el número de ocurrencias de esa consulta.
- El algoritmo seleccionado tiene complejidad O(n), donde n será la suma de todas las cadenas (títulos y abstract) formateadas.
- La gran ventaja que nos ofrece la construcción de este árbol sufijo en tiempo lineal, es el ahorro de espacio que se da usando un par de índices de referencia en la cadena general.
- La complejidad de la búsqueda es óptima, sin embargo la masiva cantidad de documentos a insertar hace a la construcción del árbol de sufijos ineficiente, esto tendrá que hacerse más rápido, la paralelización es una opción.

Referencias

1. Halim, S. and Halim, F., 2013. *Competitive programming 3*. 3rd ed. Singapore: Lulu.com, pp.249-253.
2. Ukkonen, E., 1995. On-line construction of suffix trees. *Algorithmica*, 14, pp.249--260.
3. Manbers, U. and Myers, G., 1993. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5), pp.935--948.
4. Vladu, A. and Negruseri, C., 2005. Suffix arrays--a programming contest approach. *GInfo*, 15(7).

5. Kaggle.com. 2021. *arXiv Dataset*. [online] Available at: <https://www.kaggle.com/Cornell-University/arxiv> [Accessed 3 May 2021].

6. InfoQ. 2021. *Understanding Similarity Scoring in Elasticsearch*. [online] Available at: <https://www.infoq.com/articles/similarity-scoring-elasticsearch/> [Accessed 13 May 2021].