DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: *tlengrep*

(tomas—martin—ignacio){3}

Teoría de Lenguajes
2do cuatrimestre 2023

Integrante	LU	Correo electrónico
Tomás Miguez	94/19	tomasmiguez99@gmail.com
Martin Yoel Belgorodsky	183/19	martin.belgorodsky@gmail.com
Ignacio Niesz	722/10	ignacio.niesz@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Índice

1. Introducción

Este informe se enfoca en la optimización de la búsqueda de expresiones regulares, una herramienta esencial en el procesamiento de cadenas de texto. Analizaremos nuestra solución y la compararemos en términos de tiempo de ejecución con la implementación de referencia proporcionada por la cátedra.

Nuestro objetivo principal es evaluar la eficiencia de nuestras soluciones, considerando factores como la longitud de las cadenas y la complejidad de las expresiones regulares utilizadas. A lo largo del informe, describiremos nuestra estrategia y las decisiones clave tomadas durante el proceso, además de presentar los resultados obtenidos.

Estos resultados nos ayudarán a obtener conclusiones importantes sobre cómo optimizar las expresiones regulares y mejorar la eficiencia en el procesamiento de texto.

2. Algoritmos

Para nuestra solución buscamos mejorar la solución naive provista por la catedra que utiliza recursión y barridos lineales. Para esto realizamos la implementación del algoritmo de Thompson que transforma expresiones regulares definidas de forma recursiva en AFND-λ y combinamos esta solución con las soluciones vistas en la cursada para transformar AFND-λ en AFD y para obtener AFD minimos a partir de AFD.

Para la implementación de nuestra solución partimos de la definición recursiva de una expresión regular E, con E_1 y E_2 expresiones regulares:

- $E = Empty$ es una expresión regular que admite el lenguaje $\emptyset$.
- $E = Lambda$ es una expresión regular que admite el lenguaje $\Lambda = \{\emptyset\}$.
- $E = Char(a)$ es una expresión regular que admite el lenguaje $\{a\}$.
- $E = Concat(E_1, E_2)$ es una expresión regular que admite el lenguaje $\mathscr{L}(E_1)\mathscr{L}(E_2)$.
- $E = Union(E_1, E_2)$ es una expresión regular que admite el lenguaje $\mathscr{L}(E_1) \bigcup \mathscr{L}(E_2)$.
- $E = Plus(E_1)$ es una expresión regular que admite el lenguaje $\mathscr{L}(E_1)^+$.
- $E = Star(E_1)$ es una expresión regular que admite el lenguaje $\mathscr{L}(E_1)^*$.

Finalmente la solución consiste en

1. Tomar una expresión regular E construida de forma recursiva.
2. Aplicar el algoritmo de Thompson transformando E en un AFND-λ que admite el mismo lenguaje que E.
3. Utilizar el algoritmo de determinización que permite transformar un AFND-λ en un AFD equivalente.
4. Utilizar el algoritmo de minización que permite transformar el AFD en un automata equivalente de estados mínimos.

2.1. Algoritmo de Thompson

Algorithm 1 Algoritmo recursivo de Thompson.

Input: Una ER que puede ser de tipo Empty, Lambda, Char, Union, Concat, Star, Plus.

Output: Un AFND-λ que acepta el mismo lenguaje que la ER resultado de la construcción recursiva de Thompson.

1: **function** to_afnd($Empty$)
2: $Q \leftarrow \{q_0\}$
3: $\Sigma \leftarrow \{q_0\}$
4: $\delta \leftarrow \{\}$
5: $F \leftarrow \{\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$
6: **function** to_afnd($Lambda$)
7: $Q \leftarrow \{q_0\}$
8: $\Sigma \leftarrow \{q_0\}$
9: $\delta \leftarrow \{\}$
10: $F \leftarrow \{q_0\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$
11: **function** to_afnd($Char(c)$)
12: $Q \leftarrow \{q_0\}$
13: $\Sigma\{q_0\}$
14: $\delta(q_0, c) = q_1$
15: $F \leftarrow \{q_1\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$
16: **function** to_afnd($Union(re1, re2)$)
17: $afnd1 \leftarrow to\_afnd(re1)$
18: $afnd2 \leftarrow to\_afnd(re2)$
19: $Q \leftarrow Q(afnd1) \cup Q(afnd2) \cup \{q_0, q_f\}$
20: $\Sigma \leftarrow \Sigma(afnd1) \cup \Sigma(afnd2)$
21: $\delta(q_0, \lambda) = q_0(afnd1)$
22: $\delta(q_0, \lambda) = q_0(afnd2)$
23: $\delta(q_f(afnd1), \lambda) = q_f$
24: $\delta(q_f(afnd2), \lambda) = q_f$
25: $F \leftarrow \{q_f\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$
26: **function** to_afnd($Concatenation(re1, re2)$)
27: $afnd1 \leftarrow to\_afnd(re1)$
28: $afnd2 \leftarrow to\_afnd(re2)$
29: $Q \leftarrow Q(afnd1) \cup Q(afnd2) \cup \{q_0, q_f\}$
30: $\Sigma \leftarrow \Sigma(afnd1) \cup \Sigma(afnd2)$
31: $\delta(q_0, \lambda) = q_0(afnd1)$
32: $\delta(q_f(afnd1), \lambda) = q_0(afnd2)$
33: $\delta(q_f(afnd2), \lambda) = q_f$
34: $F \leftarrow \{q_f\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$

Algorithm 2 Algoritmo recursivo de Thompson.

Input: Una ER que puede ser de tipo Star, Plus.

Output: Un AFND-λ que acepta el mismo lenguaje que la ER resultado de la construcción recursiva de Thompson.

1: **function** to_afnd($Plus(re)$)
2: $afnd \leftarrow to\_afnd(re1)$
3: $Q \leftarrow Q(afnd) \cup \{q_0, q_f\}$
4: $\Sigma \leftarrow \Sigma(afnd)$
5: $\delta(q_0, \lambda) = q_0(afnd1)$
6: $\delta(q_f(afnd), \lambda) = q_f$
7: $\delta(q_f(afnd), \lambda) = q_0(afnd)$
8: $F \leftarrow \{q_f\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$
9: **function** to_afnd($Star(re)$)
10: $afnd \leftarrow to\_afnd(re1)$
11: $Q \leftarrow Q(afnd) \cup \{q_0, q_f\}$
12: $\Sigma \leftarrow \Sigma(afnd)$
13: $\delta(q_0, \lambda) = q_0(afnd)$
14: $\delta(q_f(afnd), \lambda) = q_f$
15: $\delta(q_f(afnd), \lambda) = q_0(afnd)$
16: $F \leftarrow \{q_0, q_f\}$
return $AFND < Q, \Sigma, \delta, q_0, F >$

Notemos que en los pseudocodigos 1 y 2 usamos abuso de notación donde Q, Σ, q_0 y q_f por un lado representan los estados de un automata, el alfabeto, el estado inicial y el estado final, y por otro lado representan funciones que permiten obtener dichso valores dado un automata.
Tambien notamos que al construir nuevos estados en base a los estados de un automata dado siempre hay un renombre de etiquetas de por medio agregando siempre estados nuevos.

El algoritmo recursivo de thompson tiene complejodad $O(d)$ donde d es la cantidad de subexpresioens del árbol de la construcción recursiva de la expresión regular.

3. Experimentación y metricas

Para la experimentación de este trabajo realizamos un análisis de los tiempos de computo de nuestra solución comparandolos contra los de la solución naive.

Los experimentos realizados fueron

- `exp1` donde se realizaron metricas de tiempo sobre la expresión regular $re\_1 = (ab)*$ para la solución implementada y la solución naive sobre cadenas validas de tamaño incremental.
- `exp2` donde buscamos un peor caso tanto para la solución como para la solución naive. Para esto construimos expresiones regulares que fuercen un barrido de toda la cadena por cada subexpresión.
- `exp3` donde construimos un peor caso para la solución naive y verificamos que no afecta a nuestra solución.

En el experimento `exp1` primero comparamos nuestra implementación de la solución de thompson y minimización de automatas con la solución naive provista por la catedra para la expresión regular $re\_1 = (ab)*$. Realizamos $n = 100$ ejecuciones quedandonos con el menor tiempo de ejecución para eliminar ruído y precalentamiento de memoria cache sobre la familia de instancias $I_{exp1} = \{(ab)^{i*100} : 1 \leq i \leq 10\}$. El motivo por el cuál utilizamos cadenas validas es para fozar a la expresion regular a analizar toda la cadena de texto. Por lo visto en el análisis previo esperamos que la solución naive para la expresión regular se comporte con un peor caso del orden $O(n^m)$ donde m es la profundad del árbol recursivo generado por la cantidad de

aplicaciones de la clausura o la clausura de kleene. En este caso esperamos que tenga un comportamiento $O(n^2)$. Por otro lado nuestra solución debería tener un comportamient lineal.

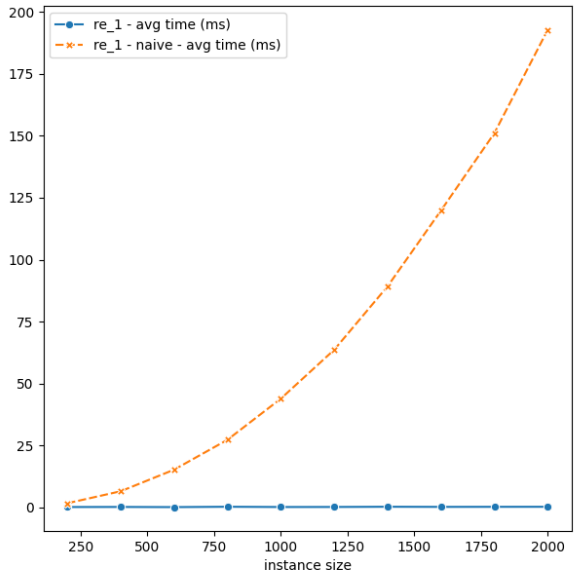

Figura 1: Comparativa de los tiempos de ejecución para la familia de instancias I_{exp1}. Se observa el comportamiento lineal de la solución y el comportamiento cuadratico de la solución naive.

En la figura 1 se observa una diferencia considerable entre la solución naive y nuestra implementación. Para verificar experimentalmente las complejidades cocientamos por las complejidades teóricas $O(n)$ y $O(n^2)$ como se observa en la figura 2.

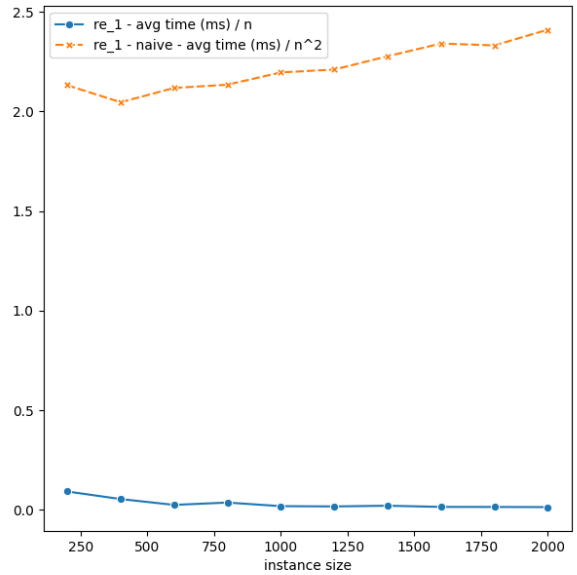

Figura 2: Comparativa de las complejidades experimentales para la familia de instancias I_{exp1}. Se observa el comportamiento lineal de la solución y el comportamiento cuadratico de la solución naive.

En el experimento `exp2` buscamos construir un "peor" caso sobre las cadenas que forzara el mayor tiempo de ejecución posible sobre una cadena de largo fija con el objetivo de verificar que nuestra solución siempre es lineal aun en el peor caso. Para ello construimos una familia de 5 expresiones regulares cada una con su

respectiva familia de instancias donde cada familia de instancias tiene los mismos tamaños de cadena pero varían en la forma de construcción.

- Expresión regular exp_2_k
- Sea k el parametro que determina la cantidad de subexpresiones con $2k + 1 \leq 52$,
- Sea un alfabeto $\Sigma = \{c_1, \ldots, c_k\} \bigcup \{d_1, \ldots, d_k\}$
- Tomamos la expresión regular formada por la concatenación de las clausuras de kleene de los elementos $c_1, \ldots, c_k$ de la forma $R_1 = c_1^* c_2^* \ldots c_k^*$.
- tomamos la expresión regular $R = (R_1 d_1 | R_2 d_2 | \ldots | R_k d_k)$

Ahora para forzar a la expresión regular a ejecutar k veces sobre la cadena de texto construimos la siguiente familia de instancias.

- Familia de instancias $I_{exp\_2\_k}$
- Cada instancia $x_{l+1} \in I_{exp2}$ se contruye como
- Una cantidad l de caracteres al azar pertenecientes a el alfabeto de R_1 es decir $\{c_1, \ldots, c_k\}$
- Agregar el caracter final d_k al final de cada instancia
- $I_{exp\_2\_k} = \{x_{l+1} : |x_{l+1}| = (i * 100), 1 \leq i \leq 10\}$

De esta forma la expresión regular exp_2_k ejecutara k veces sobre cada instancia de largo n ya que solo la expresion final $R1d_k$ admite la cadena.

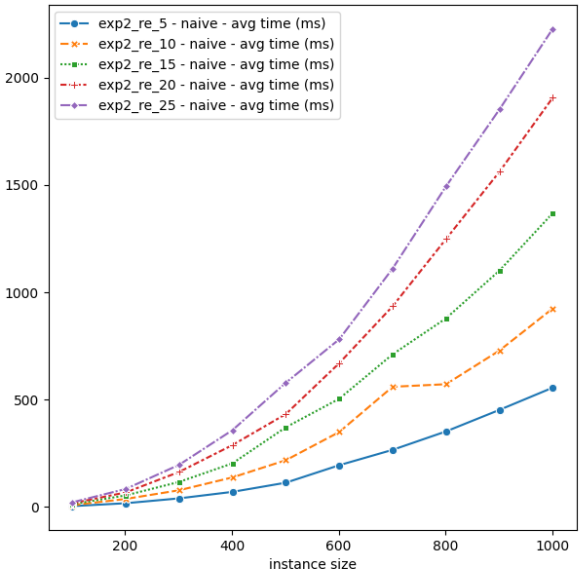

Figura 3: Comparativa de los tiempos de ejecución para cada exp_2_k naive sobre la familia de instancias $I_{exp\_2\_k}$. Se observa el comportamiento cuadratico afectado por una constante.

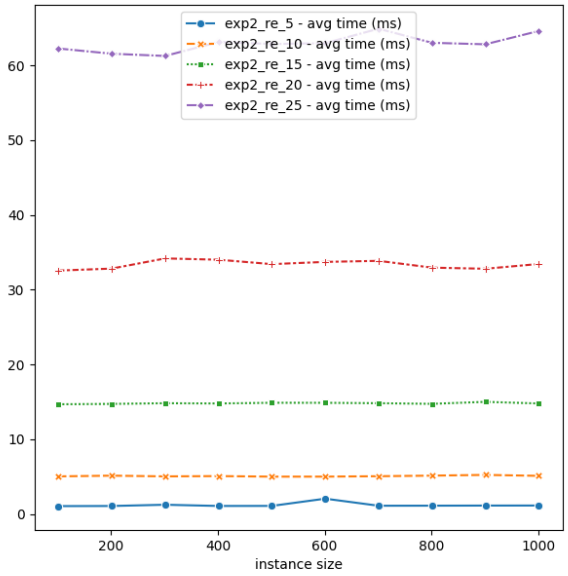

Figura 4: Comparativa de los tiempos de ejecución para cada er exp_2_k sobre la familia de instancias $I_{exp\_2\_k}$. Se observa el comportamiento lineal afectado por una constante.

En el peor caso por union nuestra solución se ve afectada por la cantidad de subexpresiones de la expresion regular siendo la última la que admite la cadena, en el peor caso sigue siendo lineal en el tamaño del largo de la cadena de entrada.

Por último en `exp3` verificamos el caso patologico para multiples clausuras de kleene. El calculo de tiempos y la familia de instancias del experimento es igual al del experimento `exp1` con $I_{exp3} = I_{exp1}$. Definimos las expresiones regulares:

- $exp3\_re1 = (ab)^*$
- $exp3\_re2 = ((ab)^*)^*$
- $exp3\_re3 = (((ab)^*)^*)^*$

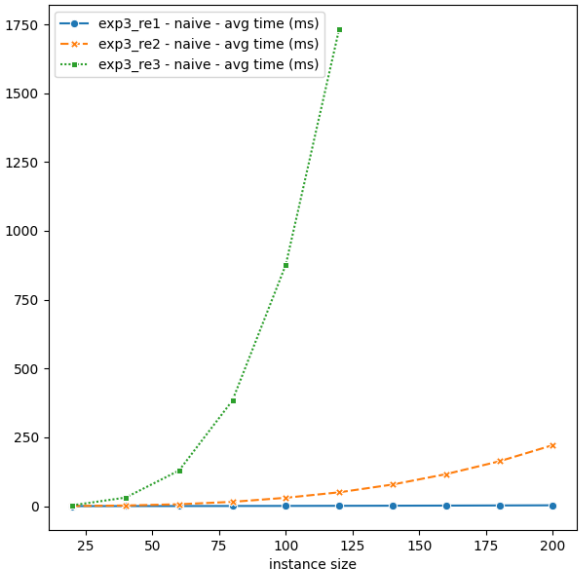

Figura 5: Tiempos de ejecución para las soluciones naive de las expresiones regulares $exp3\_re1$, $exp3\_re2$ y $exp3\_re3$ sobre la familia de instancias I_{exp_3}.

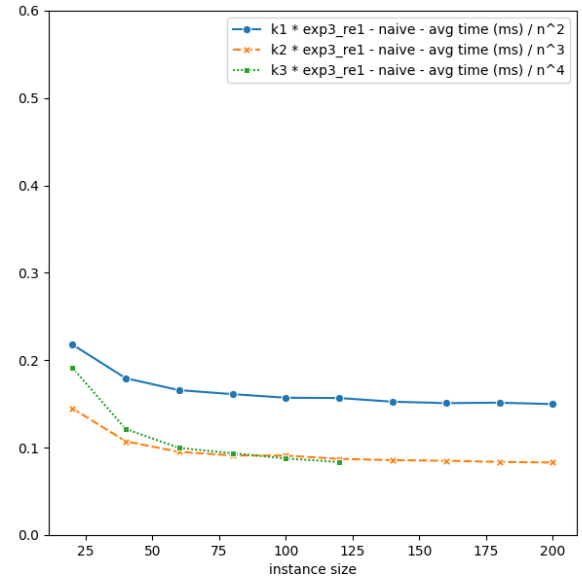

Figura 6: Complejidad experimental para las expresiones regulares $exp3\_re1$, $exp3\_re2$ y $exp3\_re3$ sobre la familia de instancias I_{exp_3}.

En la figura 5 se observa que para los casos donde hay clausuras de kleene anidadas el tiempo de computo crece rápidamente alcanzando ordenes de complejidad $O(n^2)$, $O(n^3)$ y $O(n^4)$ respectivamente. Esto es verificado experimentalmente como se observa en 6. Por otro lado en la figura 7 se observa que nuestra solución mantiene linealidad aun en casos patologicos.

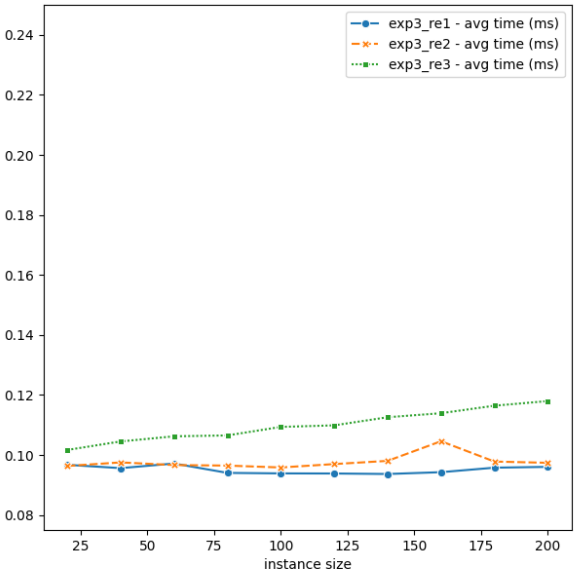

Figura 7: Tiempos de ejecución para las soluciones de las expresiones regulares $exp3\_re1$, $exp3\_re2$ y $exp3\_re3$ sobre la familia de instancias I_{exp_3} .

Los experimentos `exp1`, `exp2` y `exp3` verifican experimentalmente que nuestra solución conserva la linealidad aun en los peores casos de uniones y clausuras anidadas.

4. Conclusion

Estudiamos el algoritmo para construcción y verificación de expresiones regulares primero en su version más elemental implementado a partir de definiciones recursivas y verificación por medio de recursión más barridos lineales y lo comparamos contra la solución del algoritmo de Thompson más minimización transformando las expresiones regulares en AFND-λ, luego transformandolos en AFD y finalmente minimizando el automata.

Experimentalmente verificamos la linealidad de los peores casos de nuestra solución al momento de verificar cadenas y la comparamos contra la solución naive provista por la catedra. Lo rápido que puede crecer la complejidad dependiendo el tipo de expresión regular que se necesite (uniones, clausuras anidadas) nos confirma la necesidad de utilizar la solución de thompson u otras soluciones más sofisticadas para que los problemas sean resolubles en tiempo razonable.

Futuras lineas de investigación y posibles mejoras a este trabajo

- De la bibliografía se desprende que existen otras soluciones implementadas a partir de backtracking que no son inmunes a caer en peores casos patologicos perdiendo la linealidad dependiendo de la expresión regular. Sin embargo estas implementaciones son utilizadas en la práctica. Estudiar estos casos y como se diferencian de los casos analizandos en este informe quedan como linea de investigación para futuros trabajos.
- Quedo fuera del scope de este trabajo estudiar el comportamiento y la performance temporal del algoritmo de construcción cuya cota computacional es lineal $O(2^d)$ con d la cantidad de estados. Sería interesante analizar para que tipos de expresiones regulares la solución de "determinismo + minimización"se vuelve prohibitiva, cuáles son estos peores casos y posibles alternativas.