

Algoritmos para evaluación de expresiones regulares

Clase 11

IIC 2223

Prof. Cristian Riveros

Sobre expresiones regulares en la práctica

El lenguaje para definir expresiones regulares en la práctica se conoce como **RegEx** (o RegExp).

Las sintaxis más usadas para definir **RegEx** son:

1. POSIX (Portable Operating System Interface for uniX).
2. Perl (PCRE = Perl Compatible Regular Expressions).

Breve explicación de RegEx

	RegEx	Teoría
Carácter	<code>a</code>	a
Escape	<code>\+</code>	$-$
Cualquiera	<code>.</code>	Σ
Clase	<code>[abc]</code>	$(a + b + c)$
Clase consecutivo	<code>[a-zA-Z]</code>	$(a + \dots + z + A + \dots + Z)$
Clase exclusivo	<code>[^0-9]</code>	$(\neg \bigvee_{a \in \Sigma - \{0, \dots, 9\}} a)$
Alternación	<code>cat dog</code>	$cat + dog$
0 o 1	<code>R?</code>	$R^?$
1 o más	<code>R+</code>	R^+
0 o más	<code>R*</code>	R^*
entre n y m	<code>R{n,m}</code>	$R^n(R + \epsilon)^{m-n}$
Backreference	<code>(R)\1</code>	$?$
...

Más información: entrar a <https://regexr.com>.

¿cómo evaluamos una expresión regular?

PROBLEMA: Evaluación de expresiones regulares

INPUT: una expresión regular R

un documento w

OUTPUT: TRUE si, y solo si, $w \in \mathcal{L}(R)$

Tamaño del input

- $|R|$:= número de letras y operadores.
- $|w|$:= largo de documento.

...queremos un algoritmo polinomial en $|R|$ y $|w|$.

¿el tamaño del input esta bien definido?

Diferencia de tamaño entre expresión y documento

$$|R| \ll |w|$$

- $|R|$ puede ser del orden **decenas** operadores ($\sim 1\text{KB}$).
- $|w|$ puede ser del orden de **miles a millones** de símbolos ($\sim 100\text{MB}$).

Análisis de tiempo diferenciado

- **Combined-complexity**: expresión y documento son parte del input.
- **Data-complexity**: solo documento es parte del input.

(tamaño de expresión es considerada una constante)

Buscamos algoritmos que sean **lineales en data-complexity** y
ojalá **polinomiales en combined-complexity**.

¿cómo evaluamos una expresión regular?

PROBLEMA: Evaluación de expresiones regulares

INPUT: una expresión regular R

un documento w

OUTPUT: TRUE si, y solo si, $w \in \mathcal{L}(R)$

1. Convertimos R a un ϵ -NFA \mathcal{A}_R .
2. Verificamos si $w \in \mathcal{L}(\mathcal{A}_R)$.

¿cómo hacemos 1. y cuanto tiempo toma? ¿cómo hacemos 2.?

¿cómo evaluamos un autómatata no-determinista?

PROBLEMA: Evaluación de NFA

INPUT: un autómatata no-determinista $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$

un documento w

OUTPUT: TRUE si, y solo si, $w \in \mathcal{L}(\mathcal{A})$

Tamaño del input

- $|w|$:= largo de documento
- $|\mathcal{A}| := |Q| + |\Delta|$

Algoritmo **lineal** en data-complexity y
ojalá **polinomial** en combined-complexity.

Algoritmos de evaluación de autómatas no-deterministas

Veremos varias soluciones . . .

1. Backtracking
2. DFA
3. NFA determinización
4. NFA on-the-fly
5. ϵ -NFA on-the-fly

Solución 1: Backtracking (naive)

Para un NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ y una palabra $w = a_1 \dots a_n$.

Function eval-backtracking (\mathcal{A}, w)

└ **return** backtracking ($\mathcal{A}, w, q_0, 1$)

Function backtracking (\mathcal{A}, w, p, i)

└ **if** $i \leq n$ **then**

└ **foreach** $q \in \Delta(p, a_i)$ **do**

└ **if** backtracking ($\mathcal{A}, w, q, i + 1$) = TRUE **then**

└ **return** TRUE

else if $q \in F$ **then**

└ **return** TRUE

return FALSE

¿cuál es el tiempo de eval-backtracking en el peor caso?

Solución 1: Backtracking (naive)

Experimento en vivo

1. Abra <https://regexr.com/> para testear expresiones regulares.
2. Pruebe la expresión regular:

$$^a^n a^n \$$$

sobre el input a^n donde R^n significa R repetido n veces.

3. Vaya incrementando el n gradualmente.

Moraleja: *muchos motores de RegEx usan backtracking para la evaluación de expresiones regulares*

¿cuál es la ventaja de usar **backtracking**?

Solución 2: DFA

Para un DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ y una palabra $w = a_1 \dots a_n$.

Function eval-DFA (\mathcal{A}, w)

$q := q_0$

for $i = 1$ **to** n **do**

$q := \delta(q, a_i)$

return **check** ($q \in F$)

Análisis de tiempo

- ¿cómo hacemos $q := \delta(q, a_i)$ de manera **eficiente**?
- ¿cuál es el tiempo de eval-DFA en el **peor caso**?

¿para qué nos sirve **evaluar** un DFA?

Solución 3: NFA determinización

Para un NFA $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ y una palabra $w = a_1 \dots a_n$.

Function eval-NFA (\mathcal{A}, w)

$\mathcal{A}^{\text{det}} := \text{NFAtoDFA}(\mathcal{A})$

return eval-DFA($\mathcal{A}^{\text{det}}, w$)

Análisis de tiempo

- ¿cuál es el tiempo de eval-NFA en el **peor caso**?

¿cuál es el problema con esta solución?

¿es necesario construir la determinización completa?

Recordatorio

Para un autómata no-determinista $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, se define el autómata determinista (**determinización** de \mathcal{A}):

$$\mathcal{A}^{\text{det}} = (2^Q, \Sigma, \delta^{\text{det}}, q_0^{\text{det}}, F^{\text{det}})$$

■ $2^Q = \{S \mid S \subseteq Q\}$ es el conjunto potencia de Q .

■ $q_0^{\text{det}} = I$.

■ $\delta^{\text{det}} : 2^Q \times \Sigma \rightarrow 2^Q$ tal que:

$$\delta^{\text{det}}(S, a) = \{q \in Q \mid \exists p \in S. (p, a, q) \in \Delta\}$$

■ $F^{\text{det}} = \{S \in 2^Q \mid S \cap F \neq \emptyset\}$.

Solución 4: NFA *on-the-fly*

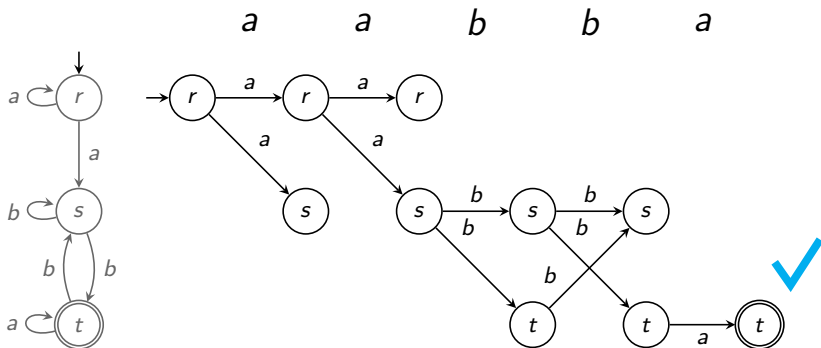
$\delta^{\text{det}} : 2^Q \times \Sigma \rightarrow 2^Q$ tal que:

$$\delta^{\text{det}}(S, a) = \{q \in Q \mid \exists p \in S. (p, a, q) \in \Delta\}$$

Estrategia on-the-fly

1. Mantenemos un conjunto S de estados actuales.
2. Por cada nueva letra a , calculamos el conjunto $\delta^{\text{det}}(S, a)$.

Solución 4: NFA *on-the-fly* (ejemplo)



Solución 4: NFA *on-the-fly*

Para un NFA $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ y una palabra $w = a_1 \dots a_n$.

Function eval-NFAonthe-fly (\mathcal{A}, w)

```
S := I
for i = 1 to n do
    Sold := S
    S := ∅
    foreach p ∈ Sold do
        S := S ∪ {q | (p, ai, q) ∈ Δ}
return check(S ∩ F ≠ ∅)
```

Análisis de tiempo

- ¿cuál es el tiempo de eval-NFAonthe-fly en el **peor caso**?

¿es posible hacer este algoritmo mejor?

Solución 6: ϵ -NFA *on-the-fly*

Para un ϵ -NFA $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ y una palabra $w = a_1 \dots a_n$.

Function eval-eNFAonthefly (\mathcal{A}, w)

```
  S :=  $\epsilon$ -closure( $\Delta, I$ )
  for  $i = 1$  to  $n$  do
    Sold := S
    S :=  $\emptyset$ 
    foreach  $p \in S_{old}$  do
      S := S  $\cup$  { $q \mid (p, a_i, q) \in \Delta$ }
    S :=  $\epsilon$ -closure( $\Delta, S$ )
  return check( $S \cap F \neq \emptyset$ )
```

Análisis de tiempo

- ¿cómo calculamos ϵ -closure(Δ, S) **eficientemente**?
- ¿cuál es el tiempo de eval-eNFAonthefly en el **peor caso**?

Resumen de técnicas de evaluación simple

Para un autómata $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ y una palabra $w = a_1 \dots a_n$:

	Tiempo
Backtracking	$\mathcal{O}(\mathcal{A} ^{ w })$
DFA	$\mathcal{O}(\mathcal{A} + w)$
NFA	$\mathcal{O}(2^{ Q } + w)$
ϵ -NFA on-the-fly	$\mathcal{O}(\mathcal{A} \cdot w)$

¿cuál funciona mejor **en la práctica**?