Evaluación de RegEx

Clase 14

IIC2223 / IIC2224

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	`a`	a
Escape	`\+`	-
Cualquiera	`.`	Σ
Clase	`[abc]`	$(a+b+c)$
Clase consecutivo	`[a-zA-Z]`	$(a+\cdots+z+A+\cdots+Z)$
Clase exclusivo	`[^0-9]`	$\left(+_{a\in\Sigma-\{0,\ldots,9\}}\, a\right)$
Alternación	`cat | dog`	$cat + dog$
0 o más	`R*`	R^*
1 o más	`R+`	R^+
0 o 1	`R?`	$R^?$
entre n y m	`R{n,m}`	$R^n(R+\epsilon)^{m-n}$
Backreference	`(R)\1`	?
...	...	...

Más información: **https://regexr.com**.

¿cómo evaluamos una expresión regular?

`PROBLEMA:`	Evaluación de expresiones regulares
`INPUT:`	una expresión regular R y
	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 $|w|$.

¿son ambos inputs **igual de importantes**?

Diferencia de tamaño entre expresión y documento

$|R| << |w|$

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

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 y
	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.?

...dedicaremos el resto
de la clase para ver 2.

¿cómo evaluamos un autómata no-determinista?

`PROBLEMA:`	Evaluación de NFA
`INPUT:`	un autómata no-determinista $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ y
	un documento w
`OUTPUT:`	`TRUE` si, y solo si, $w \in \mathcal{L}(R)$

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

Solución 1: Backtracking

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** $(p, a_i, q) \in \Delta$ **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

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 n es un parámetro.
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**

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

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**? $\mathcal{O}(|\mathcal{A}| + |w|)$

¿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` $(\mathcal{A}, w)$

  $\mathcal{A}^{\text{det}} :=$ `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**? $\mathcal{O}(2^{|Q|} + |w|)$

¿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 $\mathcal{A}$):

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

- $2^Q = \{S \mid S \subseteq Q\}$ es el conjunto potencia de Q.
- $q_0^{\text{det}} = I$.
- $\delta^{\text{det}} : 2^Q \times \Sigma \to 2^Q$ tal que:

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

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

Solución 4: NFA *on-the-fly*

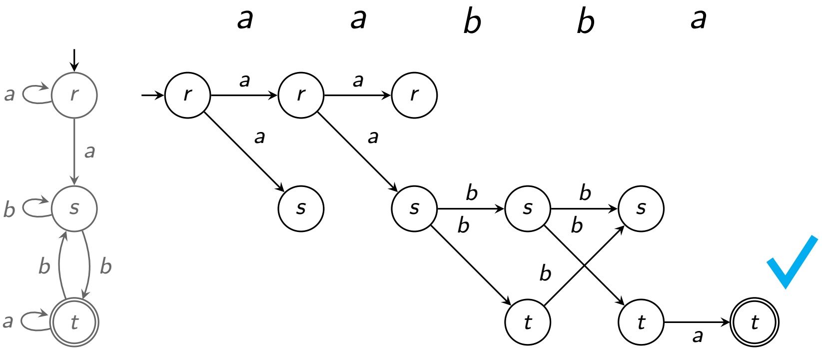
$\delta^{\text{det}} : 2^Q \times \Sigma \to 2^Q$ tal que:

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

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-NFAonthefly` $(\mathcal{A}, w)$

$S := I$

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

$\quad S_{\text{old}} := S$

$\quad S := \varnothing$

$\quad$ **foreach** $p \in S_{old}$ **do**

$\quad\quad S \ := \ S \cup \{q \mid (p, a_i, q) \in \Delta\}$

return check $(S \cap F \neq \varnothing)$

Análisis de tiempo

- ¿cuál es el tiempo de `eval-NFAonthefly` en el **peor caso**? $\mathcal{O}(|\mathcal{A}| \cdot |w|)$

¿es posible mejorar este algoritmo?

…es posible demostrar que no.

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

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

Bonus: NFA *on-the-fly* en la práctica

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

Function `eval-NFAonthefly` $(\mathcal{A}, w)$

- **let** H := HashTable en $2^Q \times \Sigma$
- $S := I$
- **for** $i = 1$ **to** n **do**
  - **if** $H(S, a)$ is not null **then**
    - $S := H(S, a_i)$
  - **else**
    - $S_{\text{old}} := S$
    - $S := \varnothing$
    - **foreach** $p \in S_{old}$ **do**
      - $S \ := \ S \cup \{q \mid (p, a, q) \in \Delta\}$
    - $H(S_{\text{old}}, a_i) := S$
- **return check** $(S \cap F \neq \varnothing)$

¿cúal es la ventaja de este algoritmo en **la práctica**?

Cierre de clase

En esta clase vimos:

1. Evaluación de regex.
2. Análisis de algoritmos de evaluación de autómatas.
3. Soluciones para evaluar un NFA en la práctica.

Próxima clase: Transductores.