Regex Matching

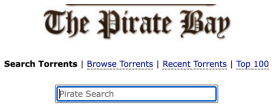
od regulárních výrazů přes automaty a algoritmy ke složitosti a bezpečnosti

Ondřej Lengál, Tomáš Vojnar

TIN'24, FIT VUT v Brně

11. listopadu 2024

Regex matching

- regex — regular expression (regulární výraz)
- používán každodenně:
  - vyhledávače  ,  , 
  - systémové nástroje: `grep`, `sed`, `awk` (UNIX), `re2` (Google), `hyperscan` (Intel)
  - zpracování/sanitizace uživatelského vstupu (CLI, web, . . .)
  - detekce/prevence síťových útoků  ,  , `L7-filter`
  - detekce úniku privátních dat: `git add pass.txt ; git commit ; git push`

Regexy v praxi

- teorie: $r = (a + b)^* a (a + b)^*$
  - prázdný řetězec (ϵ), symbol ($a \in \Sigma$), konkatenace (.), alternativa ($+$), Kleeneho hvězdička (*)
- praxe:

```
<\/?[\w.:-]+\s*(?:\s+(?:[\w\.:-]+(?:=(?:("|')(?:\\[\s\S]|(?!\1)[^\\])*\1|
[^\s'">=]+|(?:\{[^}]*\}))))?|\{\.{3}\w+\}))*\s*\/?>
```

  - třídy znaků: `.`, `[a-zA-Z0-9]`, `[^a-z]`, `[:print:]`, ...
  - rozšířené alternativy/opakování: `+`, `?`, `{n}`, `{min,max}`, ...
  - kotvy: `^`, `$`, `\b`, `\B`, `\A`, `\Z`, ...
  - podvýrazy (capture groups): `(...)`, `(?:...)`
  - zpětné odkazy: `\1`, `\2`, ... (neregulární)
  - look-arounds: `(?<=...)`, `(?=...)`, `(?<!...)`, `(?!...)` (neregulární)
  - non-greedy matching: `*?`, `+?`, ...

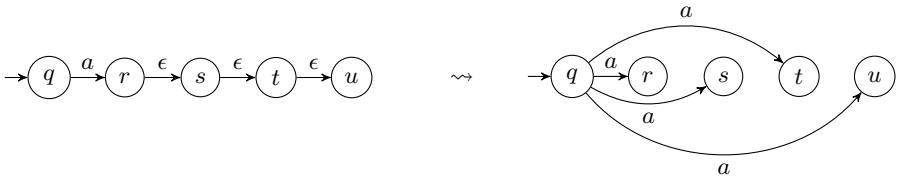
Jak efektivně zjistit, zda $w \in \mathcal{L}(r)$?

1. krok: převedeme r na konečný automat (KA):

- jedna možnost: sestavíme pro r rozšířený KA (RKA) — viz přednášky/opora/IFJ
  - Jak velký RKA vznikne z r, je-li $|r|$ dána počtem znaků v r, vnímáme-li ho jako řetězec (např. $|a.b| = 3$)?
  - Velikost RKA pro atomické regexy je konstantní.
  - Každá operace konstrukce RKA přidá konstantní počet stavů/přechodů k RKA pro podvýrazy.
  - Vznikne tedy RKA, který má $\mathcal{O}(|r|)$ stavů a $\mathcal{O}(|r|)$ přechodů.
- regex matching lze provádět přímo nad RKA s využitím backtrackingu a „přeskakováním“ ϵ-přechodů
  - nedělá se, příliš neefektivní

Jak efektivně zjistit, zda $w \in \mathcal{L}(r)$?

- obvykle se odstraní alespoň ϵ-přechody $\rightsquigarrow$ lze bez determinizace
  - rozkopírování přechodu $q \xrightarrow{a} r$ do všech stavů, do kterých se lze dostat z r přes ϵ-přechody (ϵ-uzávěr):



  - navíc je zapotřebí analogicky vyřešit ϵ-přechody z q_0 (iniciálního stavu)
  - dopad:
    - stavy: beze změny: $\mathcal{O}(|r|)$
    - přechody: počet může dramaticky narůst: lze omezit $\mathcal{O}(|r|^3)$

$$\underbrace{\text{výchozí stavy}}_{\mathcal{O}(|r|)} \times \underbrace{\text{symboly}}_{\mathcal{O}(|r|)} \times \underbrace{\text{cílové stavy}}_{\mathcal{O}(|r|)}$$

- $\rightsquigarrow$ NKA $\mathcal{A}_r$ t.ž. $\mathcal{L}(\mathcal{A}_r) = \mathcal{L}(r)$

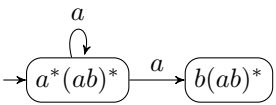
Jiné způsoby konstrukce KA

Existují i jiné možnosti převodu RV na KA: např. Antimirovovy a Brzozowského derivace

- komplikovanější, ale často efektivnější (mohou konstruovat rovnou DKA!)
- „hezčí“ implementace ve funkc. prog. jazycích
- dále např. Glushkovův automat, Follow automat, ...

Konstrukce pomocí Antimirovových derivací:

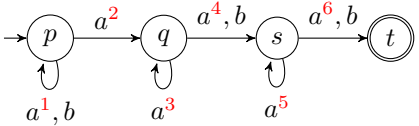
- používá koncept parciálních derivací RV dle symbolů z abecedy
- parciální derivace RV r vůči symbolu a, značeno $\partial_a(r)$, je množina RV, které reprezentují „zbytek“ řetězců popsaných r po odtržení počátečního a
- např. $\partial_a(a^*(ab)^*) = \{a^*(ab)^*, b(ab)^*\}$
- prvky vznikajících množin se stanou stavy automatu
- výsledný NKA bude mít:
  - $\mathcal{O}(n)$ stavů, kde n je počet výskytů znaků ze Σ v daném RV r (tj. $n \leq r$)
    - pro $r = a^*(ab)^*$ je $n = 3$
  - $\mathcal{O}(n^2)$ přechodů (omezený fan-out)



Regex matching pomocí backtrackingu

zkoušení různých způsobů přijetí vstupu $w \in \Sigma^*$ v $\mathcal{A}_r = (Q, \Sigma, \delta, q_0, F)$:

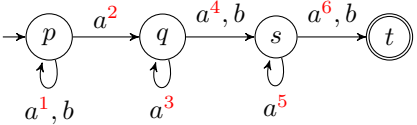
- Jaká bude složitost?
- příklad:
  - $r = (a + b)^* aa^*(a + b)a^*(a + b)$
    - v grep formátu: `^[ab]*aa*[ab]a*[ab]$`
  - $w = aaa$
  - NKA (horní indexy značí pořadí v reprezentaci $\delta(q, a)$):




```
def membership_test(w):
  return check(w, q_0)

def check(w, p):
  if w = ""
    return (q in F)
  else:
    let w = a.u    # 'a' is in Sigma
    for q in delta(p, a):
      if check(u, q): return True
  return False
```

Regex matching pomocí backtrackingu ($w = aaa$)



Regex matching pomocí backtrackingu

- Algoritmus zavolal 15krát ($\in \mathcal{O}(2^{|w|})$) funkci `check` aby zjistil, že $aaa \in \mathcal{L}(\mathcal{A}_r)$.
- Obecně počet výpočetních kroků:
  - počet uzlů ve stromě výšky $|w|+1$ s faktorem větvení daným počtem možných následníků — $\mathcal{O}(|r|)$
  - $\mathcal{O}(|r|^{|w|+1})$
- Celková doba je tedy $\mathcal{O}(|r|^2) + \mathcal{O}(|r|^{|w|+1}) \rightsquigarrow$ velmi neefektivní.
- Drahé, ale přesto se používá v knihovnách mnoha jazyků, protože se snadno kombinuje s mnoha rozšířeními RV a operací s nimi (extrakce podvýrazů, zpětné odkazy, ...).
- **Pozor!** nevhodné použít v situacích, kde vstupní text může vytvořit útočník — hrozí ReDoS
  - Regular expression Denial of Service

ReDoS

Příklad — Stack Overflow

- 20. července 2016: výpadek služby po dobu 34 min
- regex: `^[\s\u200c]+|[\s\u200c]+$`
  - `\s` — bílý znak
  - `\u200c` — *zero width non-joiner* (Unicode)
  - mazání mezer před/po textu

- text (dostal se na hlavní stránku Stack Overflow):

  The malformed post contained roughly 20,000 consecutive characters of whitespace on a comment line that started with "`-- play happy sound for player to enjoy`". For us, the sound was not happy.

- způsobilo složitost $\mathcal{O}(|w|^2)$
- detaily: https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016

Determinizace

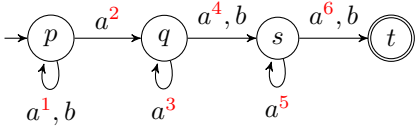
Determinizace:

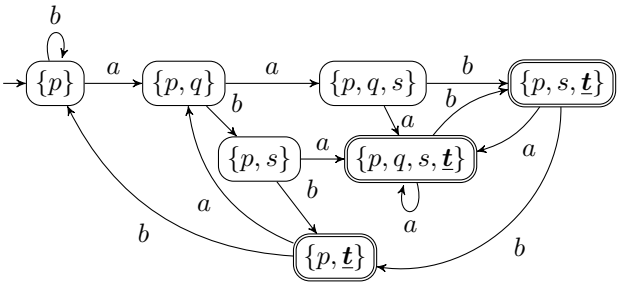
- eliminace backtrackingu — časová složitost regex matchingu $\mathcal{O}(|w|)$
  - za předpokladu použití rozumně efektivní datové struktury pro uložení DKA
  - tj. ne seznam přechodů, ale např. matice přechodů
- problém — **stavová exploze!**
  - počet stavů DKA: $2^{\mathcal{O}(|r|)}$
  - počet přechodů DKA:

$$\underbrace{\mathcal{O}(|r|)}_{\text{počet znaků v abecedě}} \cdot \overbrace{2^{\mathcal{O}(|r|)}}^{\text{počet zdrojových stavů}} = 2^{\mathcal{O}(|r|)}$$

- celková časová složitost regex matchingu $w \in \mathcal{L}(r)$ pomocí DKA:
  - $2^{\mathcal{O}(|r|)} + \mathcal{O}(|w|)$
- používá se reálně tak, že regex matcher se pokusí o převod na DKA s limitem na počet stavů
  - pokud převod selže, zůstane u NKA

DKA pro příklad

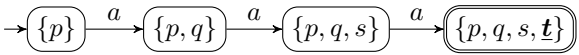




Determinizace za běhu (on the fly)

On-the-fly determinizace

- použijeme klasickou subset konstrukci řízenou w
  - tj. nezkonstruujeme nenavštívené stavy
- pro náš příklad:




```
def membership_test_otf(w):
  ms = {q0}
  for a in w:
    ms = {q | q in delta(p, a),
              p in ms}
  return (ms.inter(F) != {})
```

- složitost:

$$\underbrace{\mathcal{O}(|r|^2)}_{\substack{\text{konstrukce NKA přes} \\ \text{Antimirovovy derivace}}} + \mathcal{O}(\overbrace{|w|}^{\text{pro každý znak}} \cdot \underbrace{|r|}_{\substack{\text{pro každý} \\ \text{výchozí stav}}} \cdot \overbrace{|r|}^{\substack{\text{počet stavů} \\ \text{které můžeme} \\ \text{mít za cíl}}}) = \mathcal{O}(|w| \cdot |r|^2)$$

- lepší, ale v kritických aplikacích často pomalé
- chtěli bychom, aby časová složitost byla co nejblíž $\mathcal{O}(|w|) \rightsquigarrow$ zavedeme cache!

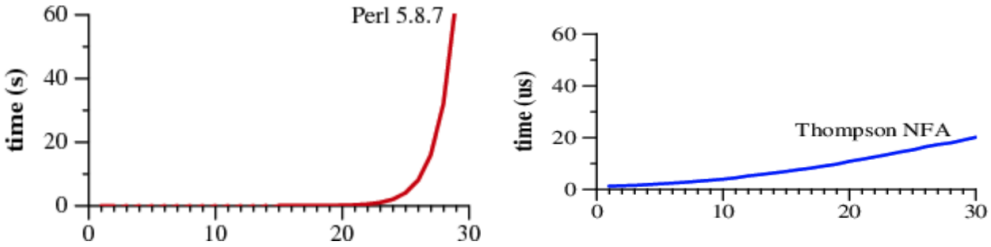
On-the-fly determinizace s cachováním — Ken Thompson

```
def thompson(w):
  state_cache = {}  # mapuje množiny stavů na id
  trans_cache = {}  # mapuje (id množiny stavů, symbol) na id cílové množiny stavů
  state = state_cache.store({q0})                # získá id
  for a in w:
    if (q = trans_cache[(state, a)]) != NULL:    # pokud je delta(state, a) v cache
      state = q
    else:
      ms = state_cache.get(state)                # id -> množina
      next_ms = {q | q in delta(p, a), p in ms}
      next_state = state_cache.store(next_ms)
      trans_cache[(state,a)] = next_state
      state = next_state
  return (F.inter(state_cache.get(state)) != {})# test akceptace
```

On-the-fly determinizace s cachováním — Ken Thompson

On-the-fly determinizace s cachováním:

- složitost zpracování 1 znaku:
  - $\mathcal{O}(1)$ — cache hit
  - $\mathcal{O}(|r|^2)$ — cache miss
- pro dostatečně dlouhý řetězec při úspěšném použití cache dostaneme složitost blízkou $\mathcal{O}(|w|)$
  - plus nutnost sestavit NKA: $\mathcal{O}(|r|^2)$
- pro běžné regexy obvykle není problém
  - ale existují vylepšení/alternativy, např. Intel HyperScan



Time to match $a?^n a^n$ against a^n

On-the-fly determinizace s cachováním — Ken Thompson

- **ale** problémy nastanou s rozšířenými regexy, např.
  - omezené opakování: `r{min,max}`
  - zpětné odkazy: `(.*)\1` (neregulární)
  - ...
- **omezené opakování** (bounded repetition): `r{min,max}`
  - např. `.*a{1,100}`
  - klasický přístup: rozvinutí pomocí základních operátorů: `.*a(ϵ|a(ϵ|a(ϵ|a(...))))`
  - nezanořená opakování:
    - velikost rozvinutí: $\mathcal{O}(|r| \cdot \texttt{max})$
    - pozor! `max` je zapsáno dekadicky a platí, že $\texttt{max} \in 2^{\mathcal{O}(|r|)}$ (např. $|100| = 3$)
    - $\rightsquigarrow$ počet stavů NKA: $2^{\mathcal{O}(|r|)}$
  - zanořená opakování: `((r{min1,max1}){min2,max2}){min3,max3}`
    - nutnost pamatovat si kombinaci hodnot zanořených čítačů (vektor hodnot)
    - velikost rozvinutí: $\mathcal{O}(|r| \cdot \texttt{max}^{\mathcal{O}(|r|)})$ (`max` je největší omezení)
    - $\rightsquigarrow$ sestavení NKA: $2^{\mathcal{O}(|r|^2)}$
  - potenciál pro ReDoS

Regex matching s omezenými opakováními

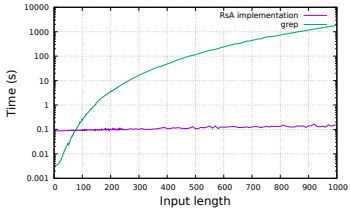
Jak efektivně řešit omezená opakování?

- základní myšlenka: nerozvíjet hodnoty čítačů předem do řídicích stavů, ale rozšířit KA o dodatečné čítače počítající za běhu počet opakování
- velikost takového NKA: $\mathcal{O}(|r|^2)$
- nutno zobecnit všechny operace, které byly uvedeny výše:
  - derivace,
  - výpočet následného stavu,
  - determinizace
    - pro některé podtřídy jde udělat efektivně
  - ...
- mnoho otevřených problémů $\rightsquigarrow$ projektová praxe, diplomová práce?

Regex matching se zpětnými odkazy

zpětné odkazy:

- neregulární: `^(.*)\1$`
- většinou řešeny backtrackingem $\rightsquigarrow$ pomalé!
  - opět potenciál pro ReDoS
- někdy rovnou zakázány (`re2`, `hyperscan`, ...)
- pro efektivní matching nutnost použít jiné formální modely, např.
  - memory automaty,
  - registrové automaty:
    - NKA + registry pro uložení symbolů (z potenciálně ∞ abecedy)
    - obecně nejdou determinizovat
    - prázdnost: **PSPACE**-complete
    - inkluze/ekvivalence: nerozhodnutelné
  - registr-množinové automaty:
    - NKA + registry pro uložení množiny symbolů
    - umožňují determinizovat podtřídu registrových automatů
    - prázdnost: $\mathbf{F}_\omega$-complete
    - příklad: `^.*(.).*\1.*;.*;.*\1$`
- mnoho otevřených problémů $\rightsquigarrow$ projektová praxe, diplomová práce?



Průmyslové matchery: Intel HyperScan

Intel HyperScan:

- aktuálně zřejmě nejvýkonnější matcher (pro průměrný případ)
- on-the-fly determinizace bez cache
- řada dalších technik
  - maximální využití Intel HW — vektorové instrukce z instrukční sady `AVX512VBMI`
  - přednostní hledání výskytů konstantních podvýrazů a až pak použití výše uvedených technik mezi nalezenými výskyty
  - např. pro regex `.*foo.*a{1,100}bar` bude hledat výskyty `foo` a `bar` jako první
- více na https://branchfree.org/ (+ články)

Průmyslové matchery: .NET7 regex matcher

.NET7 regex matcher:

- nekonstruuje dopředu automat: používá on-the-fly derivace
  - včetně podpory konstrukcí jako kotvy nebo čítače
- více na https://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7/

Odkazy

- Ken Thompson. Programming Techniques: Regular expression search algorithm. Commun. ACM 11, 6 (June 1968), 419–422. link
- Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). link
- Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, Tomáš Vojnar. Succinct Determinisation of Counting Automata via Sphere Construction. APLAS 2019: 468-489. link
- Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, Tomáš Vojnar. Regex matching with counting-set automata. Proc. ACM Prog. Lang. 4(OOPSLA): 218:1-218:30 (2020). link
- Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, Tomáš Vojnar. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. USENIX Security Symposium 2022: 4165-4182. link