Fulltextové systémy - rešerše a report

Algoritmy pro vyhledávání textu pomocí automatu

Lukáš Čejka

9. března 2021

Obsah

0 Úvod

V této rešerši je popsán výběr algoritmů, které při práci s textem využívají automaty. V první části budou tyto algoritmy popsány a v druhé části bude diskuze o změřených časech běhů těchto algoritmů v porovnání s následujícími algoritmy: *Elementární* (naivní), *Knuth–Morris–Pratt* (KMP) a *Boyer–Moore* (BM).

1 Algoritmy pro vyhledávání textu pomocí automatu

V informačních technologiích jsou občas základní funkcionality považovány za triviální a ne vždy je na povrchu vidět důležitost a rozsáhlost jejich algoritmů. Exemplární příklad základní funkcionality jsou algoritmy pro vyhledávání textu v nestrukturovaných datech (data, která nejsou mezi sebou rozlišená, tzv. „tok bytů“ (prostý text) a lze v se v nich orientovat pouze pomocí plnotextového vyhledávání[1]), které fungují na následujícím principu: najít místa ve vyhledávaném textu, kde se vzorek (vyhledávaný řetězec znaků) shoduje s nějakou jeho částí.

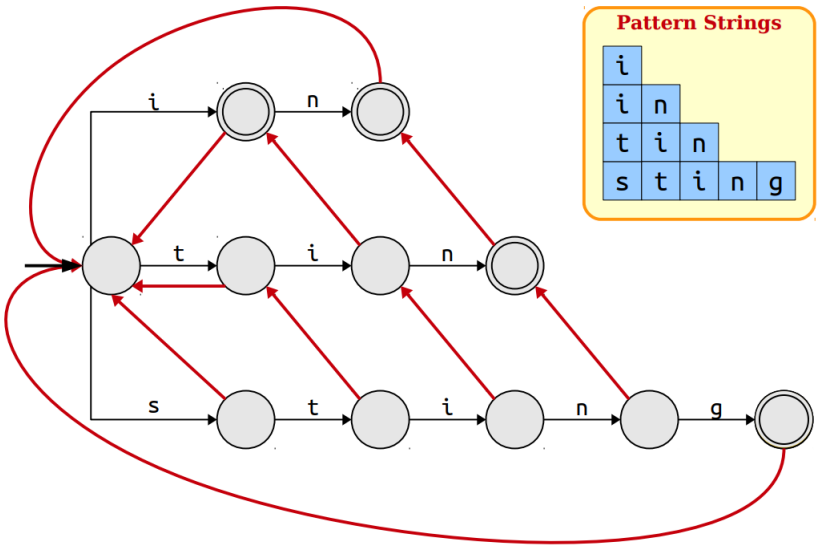
Pro formální učely bude Σ označovat množinu nazývanou *abeceda*. Tato *abeceda* může být například regulérní abeceda ($\Sigma = \{A, B, ..., \check{Z}, a, b, ..., \check{z}\}$), číselná abeceda ($\Sigma = \{0, 1, ..., 9\}$), binární abeceda ($\Sigma = \{0, 1\}$), apod., nebo jejich různé kombinace. Pak vzorek a prohledávaný text jsou posloupnosti (řetězce) znaků z abecedy Σ.
V této rešerši budou popsány následující algoritmy pro vyhledávání textu pomocí automatů:

- Aho-Corasick algoritmus (AC) (zdroje: [2])
- Commentz-Walter algoritmus (CW) (zdroje: [5][4][6])

1.1 Aho-Corasick algoritmus (AC)

AC algoritmus se používá zejména v případě potřeby vyhledání k vzorků $(P_1, P_2, ..., P_k)$ o celkové délce m v prohledávaném textu o délce n. Při použití Elementárního algoritmu nebo KMP algoritmu by se porovnával každý vzorek zvlášť, ale při použití AC algoritmu toto není potřeba.

Konstrukce Trie AC algoritmus tento problém předchází vytvořením stromové datové struktury tzv. *Trie*. Tato datová struktura se používá k reprezentaci několik různých řetězců znaků - v případě vyhledávání textu se jedná o reprezentaci vzorků.
Nechť je potřeba vyhledat následující vzorky v libovolném textu: `i`, `in`, `tin`, `sting`. Vezme se první znak z každého vzorku a udělá se pro něj hrana do svého uzlu z hlavního kořene. Tyto hrany se ohodnotí tím daným znakem. Pokud je tento znak zároveň posledním znakem v tom vzorku, tak se uzel označí jako konečný. Pokud není konečným znakem, tak se vezme znak následující a udělá se pro něj hrana vedoucí z jeho nově vytvořené uzlu do nového uzlu - hrana se opět ohodnotí tímto aktuálním znakem. Takto se postupuje iterativně, dokud se nedojde na konec každého vzorku. Výsledná Trie se nachází v obrázku 1 (v obrázku jsou navíc odkazy selhání - označené červenými šipkami). Každý uzel má hodnotu posloupnosti znaků, které se zapíší po navštívení hrany při cestě k tomu uzlu (například, hodnota kořenového uzlu je prázdný řetězec, hodnota uzlu do něhož ústí hrana `g` je `sting`, hodnota uzlu vlevo od něj je `stin`, apod.).
S tímto postupem, pokud selže porovnání uprostřed větve tak se vrací do kořene, což je chybné, například kdyby se prohledával text `sti`, tak by se začalo v kořenovém uzlu a znak po znaku by se šlo podle hodnocení hran. Tedy prvně by se šlo po hraně `s`, poté po hraně `t`, až by se skončilo v uzlu do něhož ústí hrana `i`. Jelikož je konec vzorku, tak by se prohlásilo, že prohledávaný text neobsahuje ani jeden vzorek - protože se ani jednou nenavštívil jeden z koncových uzlů.
Tento problém se řeší vytvořením odkazů selhání. Pokud dojde v nějakém uzlu k neshodě, tak se udělá hrana do uzlu, který sdílí nejdelší sufix s aktuálním uzlem. Například, pokud jsme v uzlu do kterého ústí hrana ohodnocena `n` na prostřední větvi, tak je nejdelší možný sufix `in` (nepočítá se sufix shodný s celým řetězcem cesty z kořene). Nyní je potřeba najít cestu z kořene ve tvaru `in` a na poslední uzel této cesty napojit odkaz selhání pro původní uzel. Pokud žádná cesta z kořene neexistuje, tak napojíme odkaz selhání na kořen. Tento postup se provede pro každý uzel v Trii. Výsledná Trie se nachází v obrázku 1.

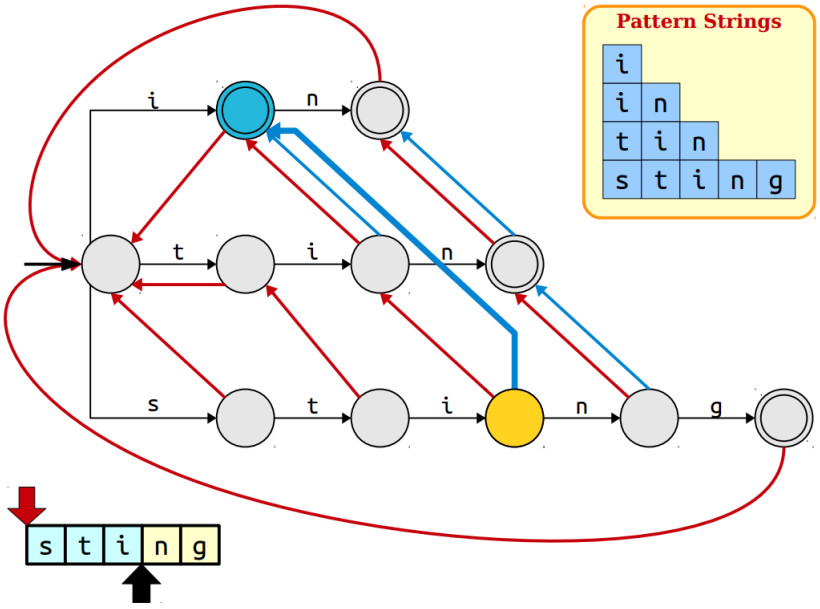


Obrázek 1: Trie s odkazy selhání (červené hrany) pro AC algoritmus[2]. Výsledné uzly jsou označeny dvojím okrajem.

Algoritmus pro vyhledávání Pseudokód pro AC algoritmus je následující:

1. Sestaví se Trie.
2. Vytvoří se koncové uzle a odkazy selhání.
3. Použije se prohledávaný text k pohybu po Trii.
4. Pokud se aktuální znak na vstupu z textu shoduje s nějakou hranou, tak se postupuje tou hranou. V jiném případě se jde po odkazu selhání.
5. Pokud se ocitne výpočet v koncovém uzlu, tak se zaznamená kterého vzorku je to koncový uzel a uloží se index.
6. Pokračuje se dokud nebude konec vstupu (už není žádný další znak v prohledávaném textu).

Příklad Příklady pro vyhledávání vzorky `i`, `in`, `tin`, `sting` v textu `sti` se nachází v obrázku 2.



Obrázek 2: Příklad použití AC algoritmu[2] pro vyhledávání vzorků v textu `sti`. Začíná se v kořenu, poté se postupuje podle hran: `s` → `t` → `i`, zde je ale konec vstupu (selhání shody), použije se odkaz selhání do uzlu `ti` na prostřední větvi. Poté opět je konec vstupu (selhání shody) a tak se naposledy použije odkaz selhání do koncového uzlu. Tento koncový uzel náleží vzorku `i` a tedy je konec algoritmu s tvrzením, že se v prohledávaném textu objevuje pouze vzorek `i`.

Časová náročnost konstrukce Trie je $\mathcal{O}(m)$, kde m je celková délka vzorků. Časová náročnost konstrukce koncových uzlů a odkazů selhání je $\mathcal{O}(m)$ a časová náročnost vyhledávání je $\mathcal{O}(n+z)$, kde n je délka prohledávaného textu a z je počet výskytů. Celková časová náročnost AC algoritmu tedy je $\mathcal{O}(n+m+z)$.

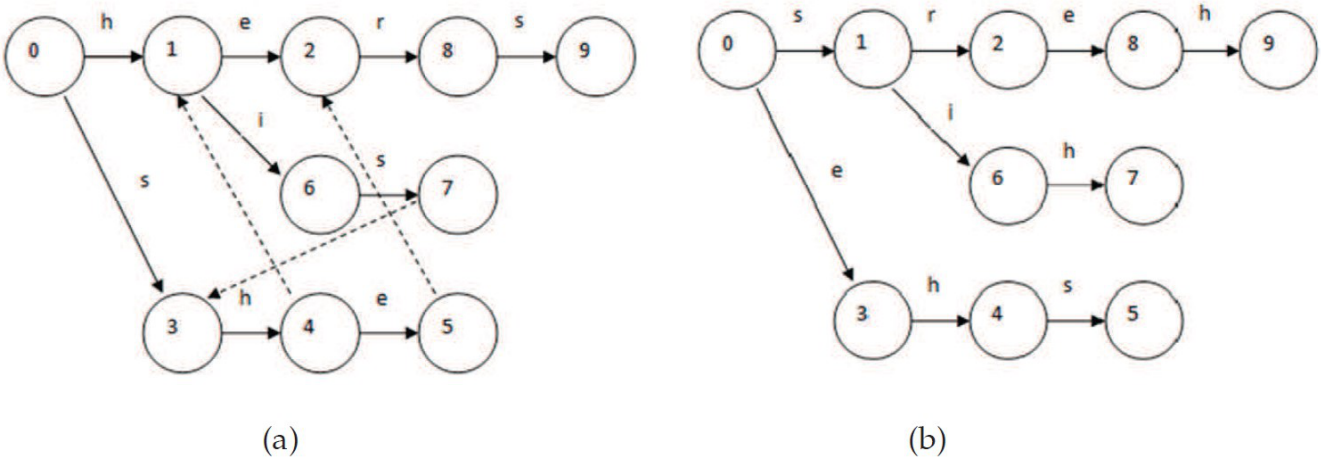
1.2 Commentz-Walter algoritmus (CW)

Commentz-Walter (CW) algoritmus spojuje myšlenky algoritmů Boyer-Moore (BM) a Aho-Corasick (AC). Z AC algoritmu přebírá CW v podstatě celý postup, s tím, že konstrukce Trie a směr prohledávání textu se provádí podle BM algoritmu - pozpátku.

Obdobně k AC algoritmu CW algoritmus předzpracovává vzorek do Trie. Postup sestavování Trie se u CW algoritmu liší od AC následně: AC sestavuje Trii podle prefixů vzorků, mezitím co CW Trii sestavuje podle suffixů vzorků, jež mají své znaky seřazené pozpátku. Další rozdíl je, že AC algoritmus sestavuje tzv. odkazy selhání mezi jednotlivými prvky Trie, pro případ, že dojde k neshodě mezi prohledávaným textem a vzorkem, CW algoritmus tyto odkazy nevytváří. Místo odkazů selhání používá CW algoritmus tzv. posuvné hodnoty, které umožňují za běhu určit o kolik pozic se vzorek má podél textu posunout. Posuvné hodnoty jsou následující:

- `shift1` - posuvná hodnota beroucí v potaz řádné suffixy slov (proper suffixes) vzorků
- `shift2` - posuvná hodnota beroucí v potaz suffixy slov se výstupními vzory
- `char (.)` - posuvná hodnota beroucí v potaz další výskyty znaků vzorků

Jinými slovy CW algoritmus vytvoří Trii ze zpětných vzorků. Poté vyhledávání začíná podobně jako u BM algoritmu: Počínaje odzadu vzorku, pokud dojde k neshodě tak se použijí pozice předchozích shodných znaků k posunutí (posuvné pozice), poté se opět pokračuje ve vyhledávání. Příklad Trií sestavených podle AC a CW algoritmu se nachází v obrázku 3. Pseudo-kód CW algoritmu se nachází v obrázku 4.
Časová náročnost konstrukce Trie pro CW algoritmus je obdobná AC algoritmu, neboli $\mathcal{O}(m)$, kde m je celková délka vzorků. Časová náročnost konstrukce koncových uzlů a posuvných hodnot je $\mathcal{O}(m)$ a časová náročnost vyhledávání je $\mathcal{O}(nm)$, kde n je délka prohledávaného textu. Celková časová náročnost CW algoritmu tedy je $\mathcal{O}(nm)$.



Obrázek 3: (a) Příklad použití AC algoritmu[4] k sestavení Trie ze vzorků slov `he`, `she`, `hers` a `his`. (b) Příklad použití CW algoritmu k sestavení Trie ze stejných vzorků.

```
1: procedure CW(y, n, m, p, root)
       ▷ Input:
       ▷ y ← array of n bytes representing the text input
       ▷ n ← integer representing the text length
       ▷ m ← array of keyword lengths
       ▷ p ← number of keywords
       ▷ root ← root node of the trie

2:     v ← root                                          ▷ The current node
3:     i ←min{m[0], m[1], ..., m[p − 1]}        ▷ i points to the current position in y
4:     j ← 0                                  ▷ j indicates depth of the current node v

5:     while i ≤ n do                                                ▷ Matching

6:         while v has child v' labeled y[i − j] do
7:             v ← v'
8:             j ← j + 1
9:             if out(v) ≠ ∅ then
10:                output i − j          ▷ Path from v to root matches y[i − j] to y[i]
11:            end if
12:        end while
                                                                     ▷ Shifting
13:        i ← i+ min { shift2(v), max { shift1(v), char(y[i − j]) − j − 1 } }
14:        j ← 0

15:    end while
16: end procedure
```

Obrázek 4: Pseudo-kód CW algoritmu převzatý z *Performance of Multiple String Matching Algorithms in Text Mining*[5].

2 Měření časů běhů algoritmů

Algoritmy pro vyhledávání v textu jsou navrženy s účelem rychlého a na paměť nenáročného vyhledávání vzorku v textu. Pro účel porovnání různých vyhledávacích algoritmů je nutné spustit vyhledávání vzorku (nikoliv vzorků) podle daných algoritmů na stejném textu. V této rešerši a reportu budou algoritmy porovnávány pouze v rámci vyhledávání jednoho vzorku v souborech o různých velikostech. Pro tento účel byla implementována tzv. Benchmarkovací procedura v programovacím jazyce Python, která má následující vstupní parametry:

- `data_path` - relativní cesta k souboru s prohledávaným textem
- `pattern` - hledaný vzorek
- `read_by_row` - povolené čtení podle řádků
- `algorithms` - pole měřených algoritmů
- `iterations` - počet iterací benchmarku, výsledné časy jsou pak průměry všech iterací

Benchmarkovací procedura je následující: prvně se určí hledaný vzorek, poté se do paměti načte vstupní soubor, buď jako jeden řetězec nebo po řádcích. Následně se definují třídy algoritmů, jež mají všechny metodu `search(text, pattern)`. Vyhledávání pro všechny algoritmy se opakuje tolikrát kolik je požadováno iterací, přičemž se při každé iteraci zaznamenává čas běhu předzpracování a vyhledávání pro každý algoritmus (*Elementární* algoritmus nepředzpracovává vzorek a tedy nemá měření v této oblasti). Jako poslední krok se vytvoří souhrnný CSV soubor s velikostí souboru v `MB`, názvy algoritmů, složitostmi algoritmů, jednotlivými iteracemi benchmarků (lze vynechat) a průměrnými časy pro dané algoritmy přes všechny iterace. Kód pro tuto proceduru se nachází v kódovém bloku 1.

```
# Benchmark given search algorithms on a given pattern and text
# Save results to each class's respective time arrays
def benchmark_algorithms(algorithms, text, pattern):
    for alg in algorithms:
        preprocess_t_ms, search_t_ms = alg.search(text, pattern)
        alg.preprocess_t_ms.append(preprocess_t_ms)
        alg.search_t_ms.append(search_t_ms)

# Benchmark given search algorithms on a given pattern and text "iteration" times
def benchmark_algorithms_n_times(algorithms, iterations, text, pattern):
    print("=================================================")
    print("Benchmark:")
    print("=================================================")
    # Run the benchmark n times
    for _ in range(0, iterations):
        benchmark_algorithms(algorithms, text, pattern)

def main():
    if not DATA_PATH.is_file():
        print("The supplied data file does not exist.\nExiting...")
        exit()

    file_size = round(DATA_PATH.stat().st_size/1000000, 2)

    # Search for this pattern
    pattern = "sit"

    # In the following file:
    # -> Reading files by row is by default set to csv files.
    if READ_FILE_BY_ROW == True:
        with open(DATA_PATH, 'r') as csv_file:
```

```
32          text = csv_file.readlines()
33      else:
34          with open(DATA_PATH, 'r') as file:
35              text = file.read()
36
37      # Declare which algorithms are to be benchmarked
38      algorithms = [ElementarySearch(), KMPSearch(), BMSearch(), ACSearch(), CWSearch()]
39
40      # Benchmark the algorithms n times
41      benchmark_algorithms_n_times(algorithms, ITERATIONS, text, pattern)
42
43      # Create a summarizing csv of preprocess, search times, etc.
44      csv_summary.create_summary_csv("benchmark_summary_table.csv", file_size, algorithms,
            EXPORT_INDIVIDUAL_RUNS)
```

Kódový blok 1: Implementace benchmarkovací procedury v programovacím jazyce Python.

Datové soubory s textem Měření bylo prováděno na dvou CSV souborech, jenž obsahují 4 sloupce:

- `id` - číslo řádku, nepočítaje záhlaví
- `label` - řetězec náhodných slov o délce 256 znaků, generovaný pomocí Lorem ipsum knihovny v Pythonu
- `lat` - náhodná zeměpisná šířka omezená v rámci krabicového ohraničení zeměpisné šířky a délky České republiky podle *OpenStreetMap*[1]
- `lon` - náhodná zeměpisná délka omezená podobně jako `lat`

První soubor obsahuje 50.000 takovýchto řádků (celková velikost souboru: ∼15 MB) a druhý soubor obsahuje 400.000 řádků (celková velikost souboru: ∼117 MB).

Nastavení benchmarku Hledaný vzorek v souboru je latinské slovo "*sit*" (hrubý překlad do češtiny: "nech to být"). Parametr `read_by_row` je nastaven jako `True`, neboli se bude soubor číst po řádcích. Benchmarkované algoritmy jsou následující: *Elementární* (naivní), *Knuth–Morris–Pratt* (KMP), *Boyer-Moore* (BM), *Aho-Corasick* (AC) a *Commentz-Walter* (CW). Počet iterací, neboli kolikrát se provede prohledávání, byl nastaven na 3.

Výsledky benchmarku Výesledky benchmarku pro menší, resp. větší, soubor se nacházejí v tabulce 1, resp. v tabulce 2.
Pokud se porovnávají algoritmy které nepoužívají automaty tak je z výsledků vidět, že Elementární algoritmus byl nejpomalejší, zatím co KMP algoritmus byl nejrychlejší. Silná stránka BM algoritmu - přeskakování až o celou délku slova - se v tomto případě při vyhledávaní vzorku "*sit*" neuplatnila, jelikož vzorek obsahuje pouze tři znaky.
Při porovnání všech algoritmů je znatelný zdánlivý nedostatek v rychlosti běhu pro algoritmy, které používají automaty k předzpracování textu (AC, CW). Tento nedostatek je očekávaný, jelikož v benchmarku se vyhledával pouze jeden vzorek, zatímco procedura těchto algoritmů je navržená specificky pro vyhledávání více vzorků v textu najednou. Tento nedostatek by se naopak stal výhodou při prohledávání více vzorků v textu, jelikož algoritmy: Elementární, KMP a BM by museli vyhledávat jednotlivé vzorky postupně - několikrát by se spouštělo individuální vyhledávání.

[1] OpenStreetMap České republiky: https://wiki.openstreetmap.org/wiki/Czech_Republic

File size [MB]	Algorithm	Complexity	Preprocess time [ms]	Search time [s]	Total time [s]
14.53	Elementary	O(mn)	-	3.77	3.77
14.53	KMP	O(m+n)	0.0	1.65	1.65
14.53	BM	O(m)	0.996	3.13	3.13
14.53	AC	O(n + m + z)	0.0	9.03	9.03
14.53	CW	O(nm)	0.0	8.51	8.51

Tabulka 1: Výsledky benchmarku algoritmů na souboru o velikosti 15 MB. Časy uvedené v tabulce jsou průměrné časy tří iterací běhu předzpracování a vyhledávání.

File size [MB]	Algorithm	Complexity	Preprocess time [ms]	Search time [s]	Total time [s]
116.58	Elementary	O(mn)	-	29.70	29.70
116.58	KMP	O(m+n)	0.0	13.51	13.51
116.58	BM	O(m)	0.333	25.51	25.51
116.58	AC	O(n + m + z)	0.0	72.09	72.09
116.58	CW	O(nm)	0.327	65.31	65.31

Tabulka 2: Výsledky benchmarku algoritmů na souboru o velikosti 117 MB. Časy uvedené v tabulce jsou průměrné časy tří iterací běhu předzpracování a vyhledávání.

3 Shrnutí

Pro práci s textem se využívá mnoho různých algoritmů od velice základních, až po složitější a sofistikovanější postupy používané v komerčních prostředí.
Pro prohledávání textu podle více vzorků je vhodné používat algoritmy, které dovolují porovnávání více vzorků najednou bez toho aby se vyhledávání muselo spouštět vícekrát - jak by tomu bylo například při použití jednoho z následujících algoritmů: *Elementární* (naivní), *Knuth-Morris-Pratt* (KMP) a *Boyer-Moore* (BM). Pro tento účel byli v této práci popsány algoritmy *Aho-Corasick* (AC) a *Commentz-Walter* (CW), které pro vyhledávání více vzorků najednou používají automat, neboli tzv. Trii, pomocí níž pak mohou dané vzorky vyhledávat v textu simultánně. AC algoritmus pomocí Trie prohledává text pro všechny vzorky najednou tak, že v případě neshody použije odkazy selhání, jenž v Trii sestavuje. CW algoritmus též s použitím Trie prohledává text pro všechny vzorky najednou, avšak porovnává vzorky odzadu, a případě neshody používá místo odkazů selhání posuvné hodnoty k určení pozic posunutí vzorku.
V rámci porovnání rychlostí vyhledávání základních algoritmů, kde se hledá pouze jeden vzorek v textu je vhodné použít algoritmy KMP nebo BM. Naopak není vhodné použít algoritmy AC nebo CW. Pro skutečné porovnání efektivity těchto dvou algoritmů je potřeba vyhledávat více vzorků v textu.

Reference

[1] wikisofia. (2013). *Data*. URL: https://wikisofia.cz/wiki/Data

[2] Agarwal, A. (2021). *Aho Corasick Algorithm*. OpenGenus IQ. URL: https://iq.opengenus.org/aho-corasick-algorithm/

[3] Hilbert, M., López, P. (2011). *The World's Technological Capacity to Store, Communicate, and Compute Information*. Science, 332(6025), 60 – 65. URL: http://www.martinhilbert.net/WorldInfoCapacity.html/

[4] Dewasurendra, S. D., Vidanagamachchi, S. M. (2018). *Average time complexity analysis of Commentz-Walter algorithm*. Journal of the National Science Foundation of Sri Lanka, 46(4), 547-557. https://doi.org/10.4038/jnsfsr.v46i4.8630

[5] Sheshasaayee, A., Thailambal, G. (2017). *Performance of Multiple String Matching Algorithms in Text Mining*. In S. C. Satapathy, V. Bhateja, S. K. Udgata, P. K. Pattnaik (Eds.), Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications (pp. 671-681). Springer Singapore. https://doi.org/10.1007/978-981-10-3156-4_71

[6] Watson, B. W. (1995). *Taxonomies and Toolkits of Regular Language Algorithms*. Ph.D. thesis, Faculty of Mathematics and Computer Science, Eindhoven University of Technology, https://doi.org/10.6100/IR444299