Politechnika
Śląska

PRACA MAGISTERSKA

Analiza narzędzi wyszukujących zawartość tekstową w systemie Linux

Kacper NITKIEWICZ
Nr albumu: 290409

Kierunek: ⟨wpisać właściwy⟩
Specjalność: ⟨wpisać właściwą⟩

PROWADZĄCY PRACĘ
Dr inż. Adrian Smagór
KATEDRA ⟨wpisać właściwą⟩
Wydział Automatyki, Elektroniki i Informatyki

OPIEKUN, PROMOTOR POMOCNICZY
⟨stopień naukowy imię i nazwisko⟩

Gliwice 2024

Tytuł pracy

Analiza narzędzi wyszukujących zawartość tekstową w systemie Linux

Streszczenie

(Streszczenie pracy – odpowiednie pole w systemie APD powinno zawierać kopię tego streszczenia.)

Słowa kluczowe

(2-5 slow (fraz) kluczowych, oddzielonych przecinkami)

Thesis title

Thesis title in English

Abstract

(Thesis abstract – to be copied into an appropriate field during an electronic submission – in English.)

Key words

(2-5 keywords, separated by commas)

Spis treści

Rozdział 1

Wstęp

Rozdział 2

Analiza tematu wyszukiwania tekstu

$$y = \frac{\partial x}{\partial t} \tag{2.1}$$

- analiza tematu
- wprowadzenie do dziedziny (*state of the art*) – sformułowanie problemu,
- poszerzone studia literaturowe, przegląd literatury tematu (należy wskazać źródła wszystkich informacji zawartych w pracy)
- opis znanych rozwiązań, algorytmów, osadzenie pracy w kontekście
- Tytuł rozdziału jest często zbliżony do tematu pracy.
- Rozdział jest wysycony cytowaniami do literatury [4, 5, 3]. Cytowanie książki [5], artykułu w czasopiśmie [4], artykułu konferencyjnego [3] lub strony internetowej [2].

2.1 Sformułowanie problemu

Wyszukiwanie tekstu w systemach towarzszy ludziom od początków istnienia maszyn, choć pierwsze komputery nie posiadały ogromnych ilość pamięci co nie powodowało potrzeby istnienia algorytmów wyszukujących tekst. Procesor Intel 8008 zaprezentowany w 1972 posiadał jedynie 14 bitową magistrale adresową co pozwalało na 16 Kbi pamięci. Model Motoroli 68000 posiada 5 MB dysku twardego, co nie może się równać z opecnym standardem darmowej pamięci udostępnianej w chmurze przez Google (15 GB).

Problem wyszukiwania danych nastąpił w momencie tworzenia dużej ilość zawartości. Posiadane archiwum wynosi 14.7 GB danych, niektóre z zawartości są zarchiwizowane co znacznie utrudnia odczytania z nich danych. Nie mniej jednak, posiadane narzędzia w systemach dają dużą dowolność w wyszukania zawartości, która nas interesuje.

Zasadniczym problem naszej pracy jest wyszukiwanie zawartości tekstowej ogramnej ilość plików w różnych formatach. Takie podejście może okazać się problematyczne w przypadku plików dzwiękowych, filomowych czy zdjęć wszelkiego rodzaju.

2.2 State of art

Podjęcie problemu wyszukiwania plików po nazwach oraz zawartości jest bardzo złożonym i trudnym problemem w sferze programistycznej. Istnieje wiele rozwiązań tego problemu, które istnieją od początku pracy z komputerem. Narzędzia takie jak **find**, grep czy fzf [1] pozwalają na wyszukiwanie zawartości która nas interesuje, ale kompleksowość tych narzędzi nie jest przystosowana do tak trudnego problemu, jakim jest wyszukiwanie treści w plikach, które są zarchiwizowane. Z taką samą niedogodnością spotykamy się w przypadku plików pochodzących z pakietu Microsoft Office 365, jednak jeśli rozwiążemy zadanie otrzymywania zawartości z archiwów, będziemy w stanie otrzymać również zawartość z plików z rozszerzeniami .doc, .docx czy .pptx.

Narzędzie **find** to znane i popularne narzędzie wśród osób zaznajomionych z technologiami linuxowymi. Już bardzo często wykorzystywany do znajdowania plików w systemie, jednak nie nadaje sie do znajdowania zawartości plików.

Do przeszukiwania zawartości plików dobrze nadaje się narzędzie grep, który jest dostępny w każdej dystrybucji Linuxa. Jego działanie jest dość podobne do finda, lecz posiada on możliwość wyszukiwania treść w plikach tekstowych jak również archiwach. Nie posiada on niestety możliwość szukania zawartości plików .pdf oraz nie wspiera formatów książkowych takich jak .djvu.

Istnieje również ripgrep, który jest sukcesorem wcześniej wymienionego narzędzia. Jego wydajność przewyższa grepa nawet trzydziestokrotnie w niektórych testach sprawnościowych, jednak zazwyczaj jest to niewielki wzrost. Nie jest on niestety domyślnie instalowany na większości systemów linuxowych. Nie posiada on również wsparcia dla formatów pdf i djvu.

Można wyszukiwać również po treści piosenek, ale wymagałoby to utworzenie modelu sztucznej inteligencji, która wydobywałaby tekst z piosenek do postaci tekstowej.

2.3 Opis poznanych rozwiązań

2.3.1 Algorytm brute force

Jest wiele algorytmów, które wyszukują tekst. Jednym z takich algorytmów jest algorytm typu brute-force. Polega sprawdzaniu każdego bajtu, jego implementacja jest bardzo prosta i standardowa, a złożoność czasowe tego rozwiązania wynosi O(m * n), gdzie m to długość bloku (pattern), a n to długość tekstu (substring), którego szukamy.

```
# Find files by extension:
    find root_path -name '*.ext'

# Find files matching multiple path/name patterns:
    find root_path -path '**/path/**/*.ext' -or -name '*pattern*'

# Find directories matching a given name, in case-insensitive mode:
    find root_path -type d -iname '*lib*'

# Find files matching a given pattern, excluding specific paths:
    find root_path -name '*.py' -not -path '*/site-packages/*'

# Find files matching a given size range, limiting the recursive depth to "1":
    find root_path -maxdepth 1 -size +500k -size -10M

# Run a command for each file (use `{}` within the command to access the filename):
    find root_path -name '*.ext' -exec wc -l {} \;

# Find all files modified today and pass the results to a single command as arguments:
    find root_path -daystart -mtime -1 -exec tar -cvf archive.tar {} \+

# Find empty (0 byte) files and delete them:
    find root_path -type f -empty -delete
```

Rysunek 2.1: Przykłady użycia programu find

Zaletą tego algorytmu jest to, że nie posiada potrzeby przechowywać żadnych danych w pamięci. Ten algorytm dobrze sprawdza się gdy posiadamy ograniczoną ilość zasobów pamięci, co nie jest problemem w obecnych czasach, gdy pamięć jest stosunkowo tania i szeroko dostępna.

Powyższy algorytm można zrównoleglić, dzieląc wzorzec na mniejsze części i wyszukując tylko dane w tym obszarze, ale należy dołożyć końce wzorca, aby nie wynikła sytuacja, w której wzorzec by wystąpił, ale nie wzięto pod uwagę końca zdania.

TYTUL	
wzorzec	ABCABCABDABD
podłańcuch	BCA
rezultat	4

Jeżeli podzielimy wzorzec na dwa procesy wyszukujące algorytmem brute-force, otrzy-

```
# Search for a pattern within a file:
    grep "search_pattern" path/to/file

# Search for an exact string (disables regular expressions):
    grep -F|--fixed-strings "exact_string" path/to/file
sec
# Search for a pattern in all files recursively in a directory,
    showing line numbers of matches, ignoring binary files:
    grep -r|--recursive -n|--line-number --binary-files without-
        match "search_pattern" path/to/directory

# Use extended regular expressions (supports `?`, `+`, `{}`, `()
    ` and `|`), in case-insensitive mode:
    grep -E|--extended-regexp -i|--ignore-case "search_pattern"
        path/to/file

# Print 3 lines of context around, before, or after each match:
    grep --context|before-context|after-context 3 "
        search_pattern" path/to/file

# Print file name and line number for each match with color
    output:
    grep -H|--with-filename -n|--line-number --color=always "
        search_pattern" path/to/file

# Search for lines matching a pattern, printing only the matched
    text:
    grep -o|--only-matching "search_pattern" path/to/file

# Search `stdin` for lines that do not match a pattern:
    cat path/to/file | grep -v|--invert-match "search_pattern"
```

Rysunek 2.2: Przykłady użycia programu grep

mamy dwa zadania:

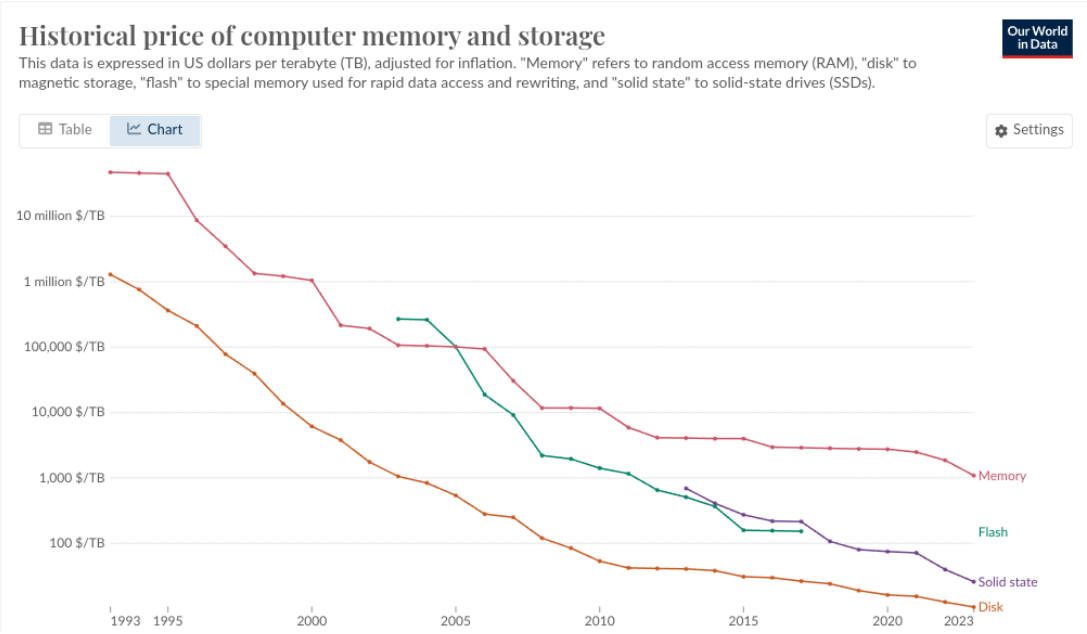
Zadania			
Zadanie 1		Zadanie 2	
wzorzec	ABCABC	wzorzec	ABDABD
podłańcuch	BCA	podłańcuch	BCA
rezultat	-1	rezultat	-1

Zrównoleglenie procesu powoduje, że otrzymaliśmy nie poprawny wynik, gdyż w żadnym z wzorców nie występuje podłańcuch "BCA", choć łańcuch występuje w miejscu 4, to algorytm nie posiada wiedzy o dalszej części wzorca.

Aby poprawić dany algorytm należy dołożyć znaki, które należy sprawdzać w przypadku poprawnego rozpatrzenia ostatniego znaku.

```
for i := 0; i<len(pattern); i++{
  for j := 0; j<len(substring); j++{
    // compare bytes
  }
}
```

Rysunek 2.3: Przykłady algorytmu brute force



Rysunek 2.4: Historyczne dane cen pamięci w latach 1993-2023

Zadania			
Zadanie 1		Zadanie 2	
wzorzec	ABCABC(AB)	wzorzec	ABDABD(nil)
podłańcuch	BCA	podłańcuch	BCA
rezultat	4	rezultat	-1

W takim przypadku sprawdzamy tylko do sytuacji, w której BC jest częścią podłańcucha, ale podłańcuch nie został w pełni znaleziony. Długość ponownego wyszukania byłaby równa len(podłańcuch) - 1.

2.3.2 Algorytm Morisa-Pratta

Algorytm Morisa-Pratta jest dość prostym algorytmem wykorzystującym możliwość wcześniejszego sprocesowania podłańcucha wyszukiwanego w tekście co przyspiesza sposób procesowania tekstu. Polega on na wykorzystaniu faktu istnienia pasującego prefikso sufiksu. Pozwala to na pominięcie pewnych porównania niektórych znaków, bez szkody w wyniku wyszukiwania.

Dzięki wykorzystaniu tej zależności możemy uniknąć cofania się indeksu i. Tablice preproc wypełniamy poprzednią wartości tak długo aż zaistnieje różnica pomiędzy obecnym a następnym znakiem tablicy substr. W przypadku różnicy zwiększamy wartość zapisywaną do tablicy preprocesora o odległość różnicy znaków. W ten sposób następnym razem będzie możliwość pominięcia porównania tych znaków.

```
curr = -1
preproc[0] = -1
for i := 1; i <= len(substr); i++ {
  for (curr > -1) && (substr[curr] != substr[i-1]) {
    curr = preproc[curr]
  }
  curr++
  preproc[i] = curr
}
```

Rysunek 2.5: Przykład preprocesowania podłańcucha

W drugim etapie można wykorzystać wcześniej przygotowaną tablice przemieszczeń **preproc**, aby obliczyć ilość przesunięcia w przypadku znalezienia niepasującego prefiksu. Dzięki temu zwykle dłuższy tekst znajdujący się w **s** możemy przeanalizować szybciej niż w przypadku algorytmu bruteforce. Powoduje to niestety problem w przypadku, gdy napis w którym wyszukujemy nie jest wystarczająco długi.

```
res := []int{}
curr := 0
found := 0
for i := 0; i < len(s); i++ {
  for (curr > -1) && (substr[curr] != s[i]) {
    curr = preproc[curr]
  }
  curr++
  if curr == len(substr) {
    for found < i-curr+1 {
      found++
    }
    res = append(res, found)
    found++
    curr = preproc[curr]
  }
}
```

Rysunek 2.6: Przykład preprocesowania podłańcucha

W podstawowej bibliotece języka Golang, w pakiecie *strings* istnieje implementacja metody *Index()*. Nie jest ona jednak w pełni przedstawiona w kodzie, natomiast w jej

implementacji można zauważyć, że algorytm brute force jest wykorzystywany tylko w przypadku gdy długość wzorca wynosi więcej niż 64.

```
1  func Index(s, substr string) int {
2    n := len(substr)
3     switch {
4     case n == 0:
5        return 0
6     [...]
7     case n > len(s):
8        return -1
9     case n <= bytealg.MaxLen: // Zwykle ten case
10       // Use brute force when s and substr both are small
11       if len(s) <= bytealg.MaxBruteForce /* 64 */{
12          return bytealg.IndexString(s, substr)
13       }
14    [...]
15   }
16 }
```

Rysunek 2.7: Szukanie łańcucha w standardowej bibliotece Golang

W przypadku w go gdy wzorzec jest większy niż 64 to wykonuje się algorytm podobny do Morisa-Pratta, który jednak posiada dodatkową walidacje w przypadku odkrycia false positives. Algorytm Morisa-Pratta nie potrzebuje takiej walidacji.

Rozdział 3

[Przedmiot pracy]

- Jak ja rozwiązuję problem?
  - rozwiązanie zaproponowane przez dyplomanta
  - analiza teoretyczna rozwiązania
  - uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi

3.1 Rozwiązanie zaproponowane przez dyplomanta

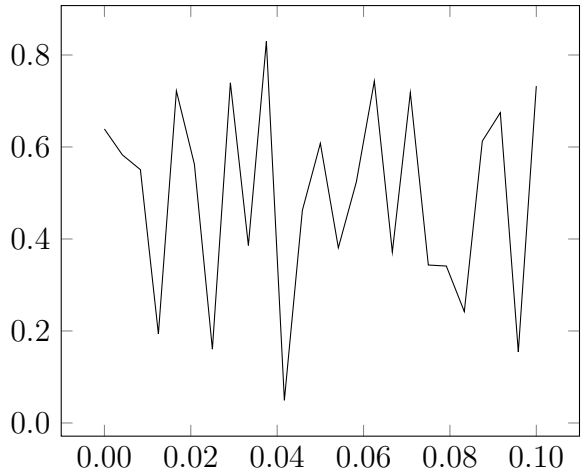
Wykorzystanie kilku znanych algorytmów do przeszukiwania zawartości tekstu i sprawdzenie, który z nich najlepiej sprawdza się pod względem prędkości i dokładności wyszukiwania.

Program nie ma na celu modyfikować informacji zawartych w plikach w żaden sposób. Powoduje to, że zostaną sprawdzane w statyczny sposób z co spowoduje zasadniczą regularność i spójność w wynikach danego algorytmu na tych samych danych.

Innym sposobem na rozwiązanie problemu jest wykorzystanie dostępnych narzędzi i dostosowanie ich do problemu, który rozwiązujemy. Takie rozwiązanie może okazać się szybsze jeżeli zależy nam na uzyskaniu rezultatu natomiast istnieje prawdopodobieństwo wykorzystania narzędzia nieodpowiedniego do danego problemu.

3.2 Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi

Do utworzenia programu wykorzystam nowoczesny język programowania Golang [6]. Posiada on bardzo wygodny model współbierzności programu co może okazać się kluczowe w przypadku tego rodzaju problemu. Dodatkowym plusem tego języka jest to, że jego składnia jest bardzo czytelna i wzorująca na prostocie początkowych kompilowanych języków programowania (C).



Rysunek 3.1: Wykres przebiegu funkcji.

Tabela 3.1: Opis tabeli nad nią.

	metoda						
			alg. 3			alg. 4, $\gamma = 2$	
ζ	alg. 1	alg. 2	$\alpha = 1.5$	$\alpha = 2$	$\alpha = 3$	$\beta = 0.1$	$\beta = -0.1$
0	8.3250	1.45305	7.5791	14.8517	20.0028	1.16396	1.1365
5	0.6111	2.27126	6.9952	13.8560	18.6064	1.18659	1.1630
10	11.6126	2.69218	6.2520	12.5202	16.8278	1.23180	1.2045
15	0.5665	2.95046	5.7753	11.4588	15.4837	1.25131	1.2614
20	15.8728	3.07225	5.3071	10.3935	13.8738	1.25307	1.2217
25	0.9791	3.19034	5.4575	9.9533	13.0721	1.27104	1.2640
30	2.0228	3.27474	5.7461	9.7164	12.2637	1.33404	1.3209
35	13.4210	3.36086	6.6735	10.0442	12.0270	1.35385	1.3059
40	13.2226	3.36420	7.7248	10.4495	12.0379	1.34919	1.2768
45	12.8445	3.47436	8.5539	10.8552	12.2773	1.42303	1.4362
50	12.9245	3.58228	9.2702	11.2183	12.3990	1.40922	1.3724

W całym dokumencie powinny znajdować się odniesienia do zawartych w nim ilustracji (rys. 3.1).

Tekst dokumentu powinien również zawierać odniesienia do tabel (tab. 3.1).

Rozdział 4

Badania

Rozdział 5

Podsumowanie

- syntetyczny opis wykonanych prac
- wnioski
- możliwość rozwoju, kontynuacji prac, potencjalne nowe kierunki
- Czy cel pracy zrealizowany?

Bibliografia

[1] Junegunn Choi. *A command-line fuzzy finder - Fzf*. 2024. URL: `https://github.com/junegunn/fzf` (term. wiz. 22. 09. 2024).

[2] Imię Nazwisko i Imię Nazwisko. *Tytuł strony internetowej*. 2021. URL: `http://gdzies/w/internecie/internet.html` (term. wiz. 30. 09. 2021).

[3] Imię Nazwisko, Imię Nazwisko i Imię Nazwisko. „Tytuł artykułu konferencyjnego". W: *Nazwa konferecji*. 2006, s. 5346–5349.

[4] Imię Nazwisko, Imię Nazwisko i Imię Nazwisko. „Tytuł artykułu w czasopiśmie". W: *Tytuł czasopisma* 157.8 (2016), s. 1092–1113.

[5] Imię Nazwisko, Imię Nazwisko i Imię Nazwisko. *Tytuł książki*. Warszawa: Wydawnictwo, 2017. ISBN: 83-204-3229-9-434.

[6] Ken Thompson Robert Griesemer Rob Pike. *Golang - the programming language*. 2009. URL: `https://go.dev/` (term. wiz. 22. 09. 2024).

Dodatki

Dokumentacja techniczna

Spis skrótów i symboli

DNA kwas deoksyrybonukleinowy (ang. *deoxyribonucleic acid*)

MVC model – widok – kontroler (ang. *model–view–controller*)

N liczebność zbioru danych

μ stopnień przyleżności do zbioru

$\mathbb{E}$ zbiór krawędzi grafu

$\mathcal{L}$ transformata Laplace'a

Lista dodatkowych plików, uzupełniających tekst pracy (jeżeli dotyczy)

W systemie do pracy dołączono dodatkowe pliki zawierające:

- źródła programu,
- zbiory danych użyte w eksperymentach,
- film pokazujący działanie opracowanego oprogramowania lub zaprojektowanego i wykonanego urządzenia,
- itp.

Spis rysunków

Spis tabel