Politechnika
Śląska

# PRACA MAGISTERSKA

Analiza narzędzi dostępnych w systemie linux służących do przeszukiwania zawartości tekstowych w zbiorze plików i archiwów

**Kacper NITKIEWICZ**
**Nr albumu: 290409**

**Kierunek:** Informatyka Przemysłowa
**Specjalność:** Cyberbezpieczeństwo

**PROWADZĄCY PRACĘ**
**Dr inż. Adrian Smagór**
**KATEDRA Informatyka Przemysłowa**
**Wydział Informatyki Przemysłowej**

**Gliwice 2024**

**Tytuł pracy**

Analiza narzędzi dostępnych w systemie linux służących do przeszukiwania zawartości tekstowych w zbiorze plików i archiwów

**Streszczenie**

Analiza algorytmów wyszukujących zawartość tekstową w ascii, sprawdzenie szybkości działania, zużycia zasobów oraz ich wydajności. Algorytmy zostały porównane w języku Golang, który jest zaopatrzony w wiele narzędzi testujących. Porównano wydajność programu wykorzystującego najszybszą implementacje z innymi programami służącymi do wyszukiwania tekstu.

**Słowa kluczowe**

Algorytmy, Linux, Wyszukiwanie, Wydajność, Optymalizacja

**Thesis title**

Analysis of tools available on Linux system used for text search of files and archives

**Abstract**

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

**Key words**

Algorithms, Linux, Searching, Performance, Optimization

# Spis treści

# Rozdział 1

# Wstęp

## 1.1 Wprowadzenie do problemu

Analiza struktur danych o dużych rozmiarach, szczególnie gdy mamy do czynienia z rozproszoną strukturą katalogów, co stanowi istotne wyzwanie w dziedzinie inżynierii oprogramowania i zarządzania danymi.

Jednym ze sposobów zachowywania danych i zmniejszenia ich objętości jest archiwizacja plików. Takie rozwiązanie jest bardzo przydatne w przypadku chęci zmniejszenia ilości danych przechowywanych, a także w przypadku chęci dystrybucji danych dla innych użytkowników jak zostało to zrobione, gdy repozytorium danych zostało przekazane do analizy w celu wykonania tejże pracy.

W kwestii technicznej należało rozważyć sposób efektywnego zarządzania pamięcią w przypadku czytania dużej ilości danych z dysku, opóźnienia związane z wydajnością operacji I/O, które należało ograniczyć do minimum.

Problem wyszukiwania danych nastąpił w momencie wyszukiwania dużej ilość zawartości. Posiadane archiwum wynosi 14.7 GB danych, niektóre z plików są zarchiwizowane co utrudnia odczytanie z nich danych.

Nie mniej jednak, posiadane narzędzia w systemach dają dużą dowolność w wyszukania zawartości. Istnieje również możliwość napisania własnych implementacji, które mogą zostać zoptymalizowane do danych, które odczytujemy.

Praca będzie obejmowała analizę algorytmów jak również analizę porównawczą narzędzi stosowanych do wyszukiwania tekstu w podobny sposób. **TODO opis ogólny rozdziałów**

# Rozdział 2

# Analiza tematu wyszukiwania tekstu

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

**Definicja 1.** *Definicja to zdanie (lub układ zdań) odpowiadające na pytanie o strukturze „co to jest a?". Definicja normalna jest zdaniem złożonym z 2 członów: definiowanego (łac. definiendum) i definiującego (łac. definiens), połączonych spójnikiem definicyjnym („jest to", „to tyle, co" itp.).*

**Twierdzenie 1** (Pitagorasa)**.** *W dowolnym trójkącie prostokątnym suma kwadratów długości przyprostokątnych jest równa kwadratowi długości przeciwprostokątnej tego trójkąta.*

**Przykład 1** (generalizacja)**.** *Przykładem generalizacji jest para: zwierzę i pies. Pies jest zwierzęciem. Pies jest uszczegółowieniem pojęcia zwierzę. Zwierzę jest uogólnieniem pojęcia pies.*

## 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).

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.

### 2.2.1 Przykładowe narzędzia dostępne do wykorzystania

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.

Przykłady użycia programu find:

- find rootPath -name '*.ext'
- find rootPath -path 'path/*.ext'
- find rootPath -name '*.py' -not -path '*/site-packages/*'
- find rootPath -maxdepth 1 -size +500k -size -10M
- find rootPath -type f -empty -delete

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, lecz nie w archiwach. Grep nie posiada też możliwość szukania zawartości plików .pdf oraz nie wspiera formatów książkowych takich jak .djvu.

Przykłady użycia programu find:

- grep "searchPattern"path/to/file
- grep -F|–fixed-strings "exactString"path/to/file
- cat path/to/file | grep -v|–invert-match "searchPattern"

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 Odniesienia do literatury

Istnieje wiele odniesień do wyszukiwania danych w literaturze. Praca Google [5] odnosi się do problemu wyszukiwania tekstu w dobie internetu i ilości danych, która jest przechowywana w chmurze. Wymagane jest indeksowanie, które przyspiesza wyszukiwanie, ale wykonane nie poprawnie skutkuje słabymi wyszukiwaniami. Ilość danych wydobywana i dostarczana do użytkowników stanowi duże wyzwanie oraz wymaga wykorzystania skomplikowanych algorytmów.

Należy również mieć świadomość, że wyszukiwanie tekstu html posiada dodatkową złożoność związaną z linkami (ang. *anchor*). Linki mogą prowadzić do kolejnych stron lub plików, które należy przeszukać w celu wydobycia informacji. Powoduje to, że proste wyszukiwanie wymaga adaptacji kodu, aby odnieść się do przypadków o których wcześniej nie wiedziano.

W plikach, które znajdują się na stronach mogą być na przykład archiwa, które mają różne wielkości oraz różne typy kompresji. Dodatkowo kompresja może być bardziej agresywna lub nawet stratna.

Podczas konferencji TREC [2] jeden z zespołów napotkał problem, duplikacji danych. Chęć wydobycia danych w celu utworzenia tekstów wymagało odpowiedniej deduplikacji dokumentów oraz wyborze tej treści, która posiada najwięcej wartości. Mogłoby to być tematem rozszerzenie pracy.

**TODO MORE PAPERS ADDED**

## 2.4 Opis poznanych rozwiązań

### 2.4.1 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) wyszukiwanego. Ten sposób odnajdywania sprawdza się jednak tylko w przypadku, gdy pattern jest niezbyt długi. Przykładową implementacje algorytmu można znaleźć na rysunku 2.1.

Zaletą tego algorytmu jest to, że nie posiada potrzeby przechowywać żadnych danych w pamięci. Ten algorytm dobrze sprawdza się, gdy posiadamy ograniczoną ilość zasobów

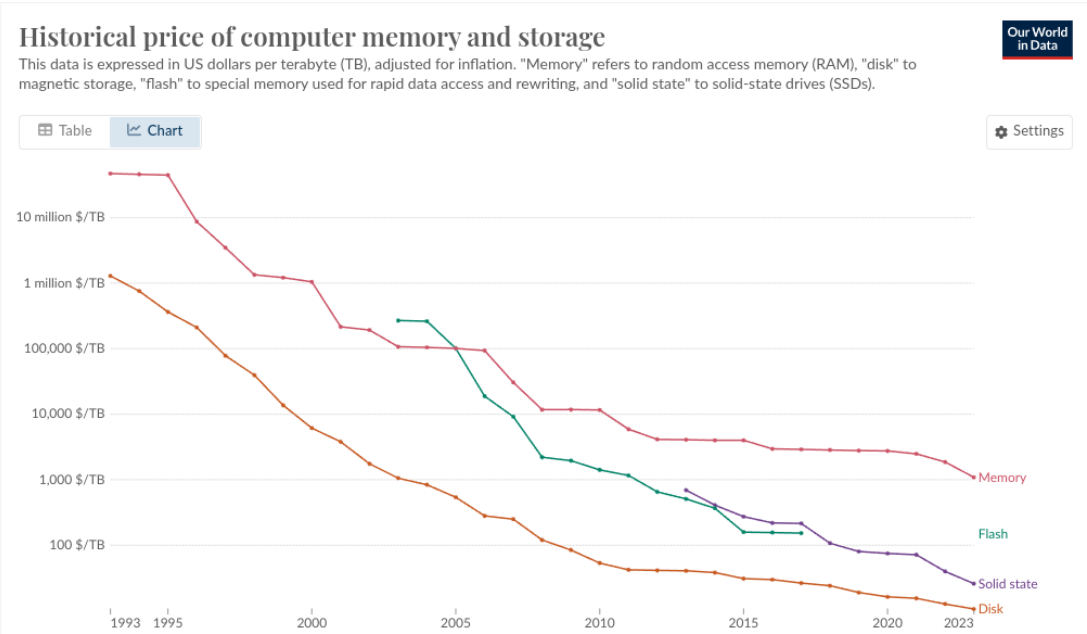
```
for i := 0; i<len(pattern); i++{
  for j := 0; j<len(substring); j++{
    // compare bytes
  }
}
```

Rysunek 2.1: Przykłady algorytmu brute force

| Metoda Brute Force | |
|---|---|
| wzorzec | ABCABCABDABD |
| podłańcuch | BCA |
| rezultat | 4 |

Tabela 2.1: Zwykły problem wyszukiwania metodą brute force

pamięci, co nie jest problemem w obecnych czasach, gdy pamięć jest stosunkowo tania i szeroko dostępna 2.2.




Rysunek 2.2: Historyczne dane cen pamięci w latach 1993-2023

Powyższy algorytm można zrównoleglić, dzieląc wzorzec na mniejsze części i wyszukując tylko dane w tym obszarze 2.1, ale należy dołożyć końce wzorca, aby nie wynikła sytuacja, w której wzorzec by wystąpił, ale nie wzięto pod uwagę końca zdania.

Jeżeli podzielimy wzorzec na dwa procesy wyszukujące algorytmem brute-force, otrzymamy dwa zadania 2.2. 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 2.3. W takim przypadku sprawdzamy

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

Tabela 2.2: Zwykły problem wyszukiwania metodą brute force

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

Tabela 2.3: Zwykły problem wyszukiwania metodą brute force

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

### 2.4.2 Algorytm Morisa-Pratta

Algorytm Morisa-Pratta jest dość prostym algorytmem wykorzystującym możliwość wcześniejszego sprocesowania podłańcucha wyszukiwanego w tekście co przyspiesza sposób procesowania tekstu jak na rysunku 2.3. 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.

```
1 curr = -1
2 preproc[0] = -1
3 for i := 1; i <= len(substr); i++ {
4   for (curr > -1) && (substr[curr] != substr[i-1]) {
5     curr = preproc[curr]
6   }
7   curr++
8   preproc[i] = curr
9 }
```

Rysunek 2.3: Przykład preprocesowania podłańcucha

W drugim etapie można wykorzystać wcześniej przygotowaną tablice przemieszczeń

**preproc**, aby obliczyć ilość przesunięcia w przypadku znalezienia niepasującego prefiksu 2.4. Dzięki temu zwykle dłuższy tekst znajdujący się w **wzorcu s** możemy przeanalizować szybciej, niż w przypadku algorytmu bruteforce. Powoduje to niestety problem w przypadku, gdy wyszukiwany wzorzec nie jest wystarczająco długi, gdyż wykonanie operacji preprocesu posiada dodatkowy koszt, którego nie ma w algorytmie brute force.

```
1  res := []int{}
2  curr := 0
3  found := 0
4  for i := 0; i < len(s); i++ {
5    for (curr > -1) && (substr[curr] != s[i]) {
6      curr = preproc[curr]
7    }
8    curr++
9    if curr == len(substr) {
10     for found < i-curr+1 {
11       found++
12     }
13     res = append(res, found)
14     found++
15     curr = preproc[curr]
16   }
17 }
```

Rysunek 2.4: 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 2.5. W Golang, gdy wzorzec jest większy niż 64 znaki, 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.

```
func Index(s, substr string) int {
  n := len(substr)
   switch {
   case n == 0:
      return 0
  [...]
   case n > len(s):
      return -1
   case n <= bytealg.MaxLen: // Usually this case is used
      // Use brute force when s and substr both are small
      if len(s) <= bytealg.MaxBruteForce /* max == 64 */{
         return bytealg.IndexString(s, substr)
      }
  [...]
  }
}
```

Rysunek 2.5: Szukanie łańcucha w standardowej bibliotece Golang

### 2.4.3 Algorytm Kurta-Morisa-Pratta

```
preproc := make([]int, lensubstr+1)
preproc[0] = -1
curr := -1
for i := 1; i <= lensubstr; i++ {
  for (curr > -1) && (substr[curr] != substr[i-1]) {
    curr = preproc[curr]
  }
  curr++
- preproc[i] = curr
+ if (i == lensubstr) || (substr[i] != substr[curr]) {
+   preproc[i] = curr
+ } else {
+   preproc[i] = preproc[curr]
+ }
}
mp.preproc = preproc
```

Listing 1: Różnica pomiędzy algorytmami KMP i MP

Kolejnym algorytmem, który rozpatrujemy jest implementacja rozszerzająca poprzednią implementacje. Różnice można zauważyć na podstawie rysunku 1 i polega ona na dodatkowym sprawdzeniu gdy nie osiągnieliśmy długości łańcucha i obecny znak jest

| wzorzec S | AAAAAABAAAAAABAAAAAAA |
|---|---|
| podłańcuch W | AAAAAA |
| liczba cofnięć | 20 |

Tabela 2.4: Przykład wykorzystania algorytmu KMP

równy temu, który znajduje się w podłańcuchu to możemy wykonać skok do elementu znajdującego sie w tablicy przygotowanej, a nie zostawać w obecnym puncie pętli. Ta różnica powoduje, że algorytm wykonuje się szybciej.

Złożoność tego algorytmu wynosi $O(2 * k)$ co mieści się w złożoności $O(k)$, gdzie k jest długością wzorca, w którym podłańcuch jest wyszukiwany. W najbardziej pesymistycznym przypadku, gdy próba dopasowania tekstu będzie kończyć się porażką, będzie wymagało to $O(n)$ operacji co nie jest szybsze, niż algorytm naiwny.

Wzorzec T przedstawia najbardziej niekorzystny scenariusz 2.4, co można zaobserwować na przykładzie tekstu S = "AAAAAABAAAAAABAAAAAAA". W tym przypadku algorytm musi sprawdzić każde wystąpienie 'A' przed dotarciem do 'B', co jest bardzo nieefektywne. Sytuacja pogarsza się wraz ze wzrostem liczby powtórzeń fragmentu "AAAAAAB". Mimo że metoda tablicowa działa tu sprawnie (bez potrzeby cofania się), to jej jednokrotne wykonanie dla podłańcucha W może być wolny, gdyż proces wyszukiwania często wymaga wielokrotnych przebiegów. Wielokrotne przeszukiwanie tekstu S w poszukiwaniu wzorca prowadzi do gorszej wydajności. W takich przypadkach, gdzie mamy do czynienia z tego typu charakterystyką tekstu i wzorca, algorytm Boyera-Moore'a może stanowić optymalne rozwiązanie.

Algorytm KMP wykorzystuje w najgorszym przypadku liniowy przebieg, natomiast algorytm Boyera-Moore'a w najlepszym przypadku posiada złożoność $O(n + k)$, a w najgorszym przypadku $O(n * k)$, gdzie n jest długością podłańcucha.

### 2.4.4 Algorytm Boyera-Moore'a

Zaletą algorytmu jest to, że ilość skoków pomiędzy porównaniami jest zwykle większa od 1, a gdy istnieje sytuacja, w której litery w podłańcuchu nie potwarzają się, to możemy przeskoczyć o długość całego łańcucha.

Algorytm Boyera-Moore'a wprowadza rewolucyjne podejście poprzez skanowanie wzorca od prawej do lewej strony, w przeciwieństwie do MP i KMP, które analizują tekst od lewej do prawej. Ta fundamentalna różnica pozwala BM na znacznie efektywniejsze przeskakiwanie fragmentów tekstu, które na pewno nie zawierają wzorca. BM wykorzystuje dwie heurystyki: "złego znaku"oraz "dobrego sufiksu", podczas gdy KMP i MP opierają się na pojedynczej tablicy prefiksowej.

W kontekście implementacji, BM wymaga utworzenia dwóch tablic pomocniczych dla swoich heurystyk, co zwiększa złożoność pamięciową w porównaniu do pojedynczej ta-

blicy prefiksowej w KMP i MP. Jednak ta dodatkowa pamięć przekłada się na możliwość wykonywania większych skoków w tekście. KMP i MP różnią się między sobą głównie w sposobie konstrukcji tablicy prefiksowej - KMP wykorzystuje bardziej zaawansowaną technikę, która pozwala na uniknięcie niektórych niepotrzebnych porównań występujących w MP.

Praktyczny wybór między tymi algorytmami zależy od charakterystyki danych wejściowych. BM sprawdza się najlepiej w przypadku długich wzorców i tekstów napisanych w językach naturalnych, gdzie występuje duża różnorodność znaków. KMP i MP są bardziej przewidywalne w działaniu i mogą być lepszym wyborem dla krótkich wzorców lub tekstów o ograniczonym alfabecie, jak na przykład sekwencje DNA.

Z tego też powodu należy wykonać heurystykę danych, które są analizowane. Można to zrobić na kilka sposobów, jednak z powodu iż nie jest to główny temat pracy zostaną użyte narzędzia potrzebne do takiej analizy.

### 2.4.5 TODO? Algorytm Karpa-Rabina

### 2.4.6 TODO? Algorytm Aho-Corasick

# Rozdział 3

# Przedmiot pracy - Wybór najlepszego rozwiązania wyszukiwania tekstu pod względem wydajności

## 3.1 Rozwiązanie zaproponowane przez dyplomanta

Przed wyborem metody sprawdzającej należy wykonać heurystyke danych. Jest to wymagane, ponieważ wydajność algorytmu jest ściśle powiązana z danymi, które będziemy przeszukiwać. Jeżeli większość danych, które analizujemy mają charakter tekstowy, to lepszym rozwiązaniem będzie skorzystanie z ostatniego algorytmu 2.4.4, jednak w innym przypadku warto wykorzystać jeden z pozostałych.

Do tego zadania należałoby przeznaczyć narzędzia, które dobrze sobie radzą z taką analizą.

- duc
- rclone
    1. ls
    2. ncdu
- tree
    1. tree -h –du | wc -l -> 14307
    2. tree -h –du -> 13827 files, 481 dirs
- skomplikowane połączenie komend

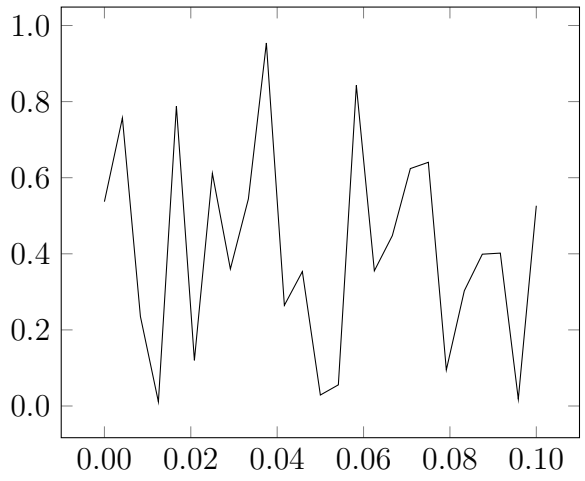
```
find . -type f -exec file --mime-type {} \; |
awk -F': ' '{print $2}' |
sort |
uniq -c |
sort -nr
```

| **Typ MIME** | **Ilość** |
|---|---|
| image/jpeg | 5,303 |
| text/html | 2,473 |
| image/gif | 2,353 |
| text/xml | 1,233 |
| application/pdf | 656 |
| application/postscript | 488 |
| text/plain | 285 |
| application/x-java-applet | 270 |
| application/zip | 153 |
| text/x-c | 134 |
| application/gzip | 126 |
| text/x-c++ | 121 |
| text/csv | 64 |
| application/octet-stream | 56 |
| text/x-java | 49 |
| application/x-rar | 34 |
| application/x-tar | 9 |
| application/x-dosexec | 3 |
| application/msword | 3 |
| text/x-diff | 2 |
| inode/x-empty | 2 |
| application/x-compress-ttcomp | 2 |
| application/vnd.openxmlformats-officedocument.wordprocessingml.document | 2 |
| application/vnd.ms-cab-compressed | 2 |
| application/vnd.microsoft.portable-executable | 2 |
| application/mac-binhex40 | 2 |
| application/x-ms-ne-executable | 1 |
| application/x-msaccess | 1 |
| application/x-ace-compressed | 1 |

Tabela 3.1: Dystrybucja w danych na podstawie typu plików MIME

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.




Rysunek 3.1: Wykres przebiegu funkcji.

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 [4]. 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).

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

Tekst dokumentu powinien również zawierać odniesienia do tabel (tab. 3.2).

Tabela 3.2: Opis tabeli nad nią.

| | metoda | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | alg. 3 | | | alg. 4, $\gamma = 2$ | |
| $\zeta$ | 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 |

# Rozdział 4

# Badania

Pierwszy test wydajnościowy, który został przeprowadzony, sprawdzał wszystkie foldery w których znajdowały się pliki bez rozszerzeń, które nie miałyby sensu, do odczytania przez algorytm. Są to takie rozszerzenia jak .pdf, .zip, .tar czy inne rozszerzenia archiwów jak i pliki zawierające obrazy takie jak .jpg czy gif. Wykonano testy na 3 algorytmach, gdzie odczytywano 5191 plików i łącznie około 240 MB danych. Oto rezultaty określonych algorytmów.

Algorytm Morisa-Pratta jest nieznacznie wolniejszy od algorytmu Kurta-Morisa-Pratta. Jest to spowodowane niewielką optymalizacją pomiędzy tymi dwoma alogrytmami.

TODO DIFF IMPLEMENTACJI

Algorytm Boyera-Moore'a wykorzystywany w takich narzędziach jak Grep, posiada wolniejszy czas egzekucji, ale algorytm może zostać napisany w lepszy sposób, aby otrzymać lepszy rezultat.

TODO SLOW BOYER PORÓWNANIE STATYSTYK ALGOS

Następna implementacja korzysta z algorytmu ze strony na githubie [3]. Jest ona znacznie wolniejsza od pozostałych algorytmów, dlatego że znaczną cześć czasu spędzamy na stworzeniu tablicy pre-procesora. Jako że zawsze sprawdzamy ten sam ciąg w wszystkich plikach w folderze, możemy stworzyć tablice pre-procasora przy pierwszym użyciu algorytmu, a następnie wykorzystywanie tej tablicy w wszystkich odczytach. To spowodowało, że czasy wykonania algorytmu boyer moora są znacznie szybsze od pozostałych.

PORÓWNANIE STATYSTYK ALGOS

Dodatkową poprawą egzekucji algorytmów było ponowne wykorzystanie bufora, do którego odczytywane zawartości plików. Powoduje to niestety, że należy wiedzieć jaki będzie rozmiar największego pliku, który odczytamy (11 MB bez plików archiwów). Moglibyśmy przed rozpoczęciem algorytmu sprawdzać rozmiar maksymalny pliku ale to wydłuży czas działania. Istnieje też sytuacja w której nie chcielibyśmy tego ograniczać.

PORÓWNANIE PAMIECI ALGOS

# 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] Chenyan Xiong Jeffrey Dalton i Jamie Callan. „The Text REtrieval Conference". W: *CAsT 2019: The Conversational Assistance Track Overview*. 2020, s. 2–3.

[3] Tatsuhiko Kubo. *Algorithm Implementation for Boyer Moore*. 2013. URL: `https://github.com/cubicdaiya/bms/blob/master/bms.go` (term. wiz. 07. 11. 2024).

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

[5] Larry Page Sergey Brin. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. 1998. URL: `https://www.sciencedirect.com/science/article/abs/pii/S016975529800110X` (term. wiz. 07. 12. 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

$\mu$ 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