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

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

## 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 [4] 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

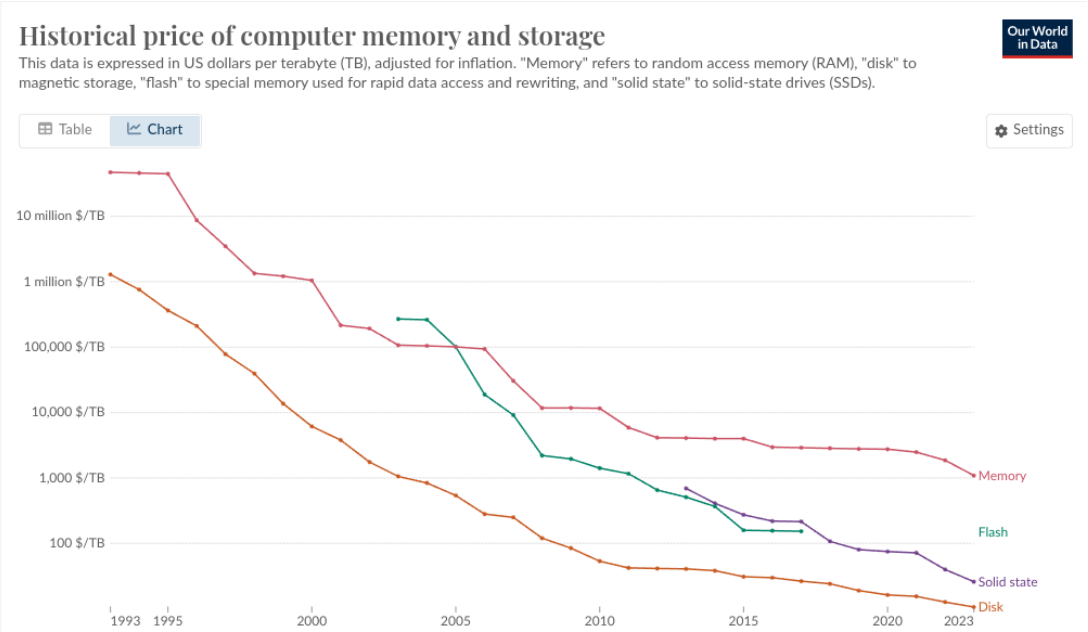
```
1 for i := 0; i<len(pattern); i++{
2   for j := 0; j<len(substring); j++{
3     // compare bytes
4   }
5 }
```

Rysunek 2.1: Przykład algorytmu 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 pamięci, co nie jest problemem w obecnych czasach, gdy pamięć jest stosunkowo tania i szeroko dostępna (rys. 2.2).

Powyższy algorytm można zrównoleglić, dzieląc wzorzec na mniejsze części i wyszukując tylko dane w tym obszarze jak w tabeli 2.1, ale należy dołożyć końce wzorca, aby nie




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

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

Tabela 2.1: Zwykły problem wyszukiwania metodą brute force

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 przedstawione w tabeli 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 (tab. 2.3). 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.

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

Tabela 2.2: Zwykły problem wyszukiwania ciągu równolegle metodą brute force zaimplementowany nie poprawnie

| 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 ciągu równolegle metodą brute force zaimplementowany poprawnie

### 2.4.2 Algorytm Morisa-Pratta

```
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

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.

W drugim etapie można wykorzystać wcześniej przygotowaną tablice przemieszczeń **preproc**, aby obliczyć ilość przesunięcia w przypadku znalezienia niepasującego prefiksu (rys. 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.

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 (rys. 2.5). W Golang, gdy wzorzec jest

```
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 2.4: Przykład preprocesowania podłańcucha w algorytmie MP

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 (rys. 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 (tab. 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 najszybszego rozwiązania wyszukiwania tekstu

## 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. Porównano ilość plików (tab. 3.1), które występują w zbiorze danych i z powodu algorytmów, które zostały wykorzystane należało odrzucić część z nich.

Program nie ma na celu modyfikować informacji zawartych w plikach w żaden sposób. Algorytmy użyte skupiają się głównie na tekście, dlatego w procesie tworzenia odrzucone zostają pliki z takimi rozszerzeniami jak

1. pliki zdjęć:
    - ".jpg",
    - ".gif",
2. pliki pdf:
    - ".pdf",
3. pliki archiwów:
    - ".tar.gz",
    - ".rar",
    - ".zip",
    - ".tgz",
    - ".tar",
    - ".gz",
4. pliki pochodzące od Microsoftu:
    - ".doc",
    - ".docx".

Innym sposobem na rozwiązanie problemu jest wykorzystanie dostępnych narzędzi i dostosowanie ich do problemu, który rozwiązujemy. Takie rozwiązanie może okazać się lepsze, jeżeli zależy nam na uzyskaniu rezultatów szybciej natomiast istnieje prawdopodobieństwo wykorzystania narzędzii, które rozwiązują problem gorzej, niż rozwiązanie stworzone wprost do tego typu problemu.

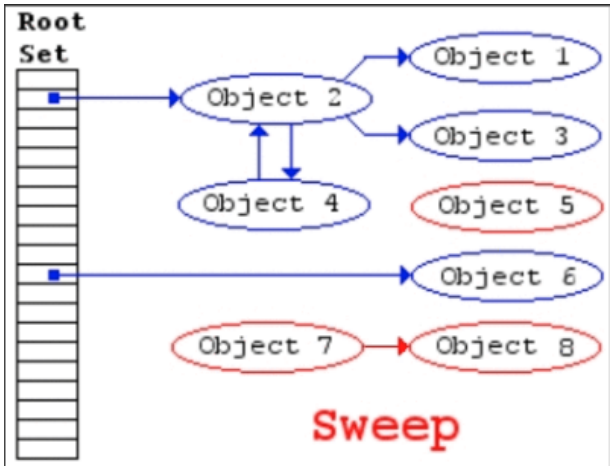
## 3.2 Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi

### 3.2.1 Użycie języka Golang

Do utworzenia programu wykorzystam nowoczesny język programowania Golang [3]. 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).

Zaletą Golang jest to, że jest kompilowany i tworzy się natywny dla danego systemu plik binarny. Daje to możliwość łatwego przenoszenia programu wynikowego. Jest to też

przewaga nad innymi językami programowania takimi jak python czy javascript, gdyż te języki są interpretowane i z natury wolniejsze niż kod binarny. Golang nie wymaga dodatkowego poziomu abstrakcji w postaci maszyny wirtualnej czytającej bytecode jak w przypadku Javy i JVM.




Rysunek 3.1: Przykładowe działanie Garbage Collectora w programie

Wadą tego języka może być fakt, iż język nie daje programiście możliwości pełnej kontroli pamięci. Język wykorzystuje ang. *Garbage Collector* (zbieracz śmieci) przedstawiony na rysunku 3.1, który jest alternatywą do sposobu manualnej alokacji pamięci. Można zauważyć, że objekty wykorzystywane w zbiorze korzenia (ang. *Root Set*) oznaczone na niebiesko, zostaną utrzymane w pamięci z powodu powiązania z innymi elementami łączącymi się do zbioru korzenia. Kolorem czerwonym oznaczone są elementy, które przestały być powiązane z jakimkolwiek elementem, więc zostaną zwolnione z użycia, po zakończeniu operacji sprawdzania.

GC wykorzystuje dwa warianty oznaczania danych. Pierwszy to tradycyjny sposób skalarny, w którym każde wykorzystanie elementu w kodzie jest liczone i zapisywane. W przypadku, gdy żaden fragment kodu nie wykorzystuje pamięci, *Garbage Collector* uwalnia zaalokowaną pamięć. Problem może skutkować wolniejszą egzekucją oraz pauzami w celu oczyszczenia pamięci.

Drugą metodą, która jest preferowana to wariany wektorowy, który wykorzystuje operacje SIMD, i dzięki temu może uwalniać większe ilości pamięci w mniejszej ilości cykli.

Kolejną zaletą jest to, że wewnętrznie Golang wykorzystuje większe obszary zaalokowanej pamięci. W przypadku, gdy program nie potrzebuje pamięci, program wewnętrznie ją uwalnia, jednak nie oddaje jej odrazu do systemu, jeżeli będzie ona wykorzystywana ponownie. Taki mechanizm zmniejsza częstotliwość operacji systemowych, które wymagają potwierdzenia alokacji, zanim praca na pamięci może zostać wykonana.

Program utworzony w ramach pracy porównuje różnicę pomiędzy sytuacją, w której programista sam zaalokował pamięć i wykorzystywał ją ponownie do czytania zawartości plików oraz sytuacje, w której pozwolił kompilatorowi na własną optymalizacje alokowania

bufora do odczytu plików.

# Rozdział 4

# Badania

## 4.1 Metodyka badań

### 4.1.1 Cel badania

Celem badani jest sprawdzenie wydajności algorytmów wyszukujących na zbiorze danych dostarczonego w celu wyszukania treści.

### 4.1.2 Hipoteza badań

Hipotezą badań jest to, że kolejne algorytmy będą wykonywały się szybciej niż ich poprzednicy. Hipotezą pomocniczą jest to, iż im większy zbiór informacji zebrany na podstawie podłańcucha tym większa prędkość algorytmu. Dodatkową hipotezą jest to, iż wykorzystanie *Garbage Collectora* wpływa na stabilność otrzymywanych wyników.

### 4.1.3 Sposób przeprowadzenia badań

```
1  go test -test.bench=. -benchmem . -benchtime=1x -count=10
2
3  func BenchmarkMorisPrattWindowWord(b *testing.B) {
4      var founds = []string{}
5      mp := &MorisPratt{}
6      filepath.Walk(DIR, WalkAndFindByAlgo(mp,
7       &founds, []byte("window")))
8      if len(founds) != 11598 {
9          b.Fatal("test failed", len(founds), 11598)
10     }
11 }
```

Rysunek 4.1: Przykład wykonania testu benchmarkowego

Badanie zostało przeprowadzone na maszynie autora, podczas działania środowiska graficznego na Fedora 40. Procesor wykonujący operacje to Intel Core i7-6700K w architekturze amd64.

Przeprowadzenie badań polegało na uruchomieniu komendy w pierwszej linii (rys. 4.1) i wykonaniu funkcji na algorytmie Morisa Pratta. Wykonano testy na wyszukaniu 3 słów, które mogły występować w zbiorze danych, ze względu na wcześniej przeanalizowaną zawartość. Tymi słowami były *"window"*, *"function"*, *"main"*.

### 4.1.4 Przebieg funkcji benchmarku

*"founds"* jest zmienną przechowującą miejsce, w których znaleziono ciąg wyszukiwany, a mp to struktura przechowująca implementacje algorytmu oraz bufor pre-procesora dla ciągu wyszukiwanego. Pierwsza implementacja nie posiadała tej struktury i bufor pre-procesora był obliczany przy każdym przebiegu algorytmu. To znacznie wpłyneło na prędkość działania algorytmu Boyera Moora.

**Definicja 1.** *WalkFunc jest to typ funkcji przyjmowany przez funkcje Walk w module filePath. Funkcja ta przyjmuje 3 argumenty: ścieżkę, informacje o analizowanym pliku oraz argument przyjmujący błąd i zwraca błąd.*

*type WalkFunc func(path string, info fs.FileInfo, err error) error*

W linijce 6 (rys. 4.1) wykonujemy Przejście (ang. *Walk*) po drzewie plików w *DIR*, który posiada ścieżkę do zbioru danych oraz przyjmuje funkcje zdefiniowaną jako WalkFunc 1. Z powodu potrzeby czytania tylko zawartości plików, wykorzystaliśmy funkcje WalkAndFindByAlgo, która będzie czytała pliki, ale pozwoli algorytmom podanym w argumencie na przeszukanie zawartości w celu znalezienia słowa *"window"*. Po zakończeniu funkcji Walk, posiadamy tablice founds, wypełnioną miejscami, w których wystąpiło znalezienie podanego ciągu. W liniach 8-10 każdy test posiada walidacje, aby wszystkie wyniki otrzymane przez algorytm, były zgodne.

W rezultacie wykonania otrzymujemy dane o czasie przebiegu funkcji, ilości alokacji oraz ile bajtów wykorzystano na operacje (rys. 4.2).

### 4.1.5 Zbiór badań

Zbiór danych posiadał znaczną ilość obrazów, archiwów oraz plików pdf, które nie będziemy skanować (tab. 4.1). Dane przedstawione zostały zredukowane z 15 GB do 3 GB, ponieważ posiadały również pliki audio oraz wideo. Część archiwów została wypakowana przed wykonaniem wyszukiwania, gdyż zdecydowano się na nie rozpakowywanie zawartości podczas wyszukiwania.

| **Typ MIME** | **Rozmiar (w KB)** | **Ilość** |
|---|---|---|
| application/pdf | 1094989.16 | 656 |
| application/zip | 911563.68 | 153 |
| application/x-rar | 751425.98 | 34 |
| application/gzip | 158709.61 | 126 |
| application/postscript | 151226.98 | 488 |
| text/html | 49879.89 | 2473 |
| image/jpeg | 49342.96 | 5303 |
| image/gif | 42260.80 | 2353 |
| text/xml | 16914.17 | 1233 |
| application/x-tar | 10470.00 | 9 |
| application/mac-binhex40 | 4704.44 | 2 |
| text/plain | 2929.91 | 285 |
| application/octet-stream | 2131.61 | 56 |
| application/ms.portable-executable | 1552.50 | 2 |
| application/x-ms-ne-executable | 1404.35 | 1 |
| text/x-c++ | 1139.31 | 121 |
| application/x-ace-compressed | 1004.49 | 1 |
| application/msword | 460.50 | 3 |
| application/x-compress-ttcomp | 362.78 | 2 |
| text/x-c | 332.51 | 134 |
| application/x-java-applet | 293.44 | 270 |
| application/x-msaccess | 154.00 | 1 |
| application/x-dosexec | 117.99 | 3 |
| text/x-java | 94.38 | 49 |
| application/vnd.ms-cab-compressed | 88.34 | 2 |
| text/csv | 54.14 | 64 |
| text/x-diff | 46.94 | 2 |
| application/wordprocessingml.document | 34.65 | 2 |
| text/x-script.python | 1.90 | 1 |
| inode/x-empty | 0.00 | 2 |
| Total | 3253691.41 | 13831 |

Tabela 4.1: Ilość danych na podstawie typu MIME

Pierwszy test wydajnościowy, który został przeprowadzony, sprawdzał wszystkie foldery, w których znajdowały się pliki, które nie miałyby sensu by być odczytane przez algorytm. Wykonano testy na 3 algorytmach, gdzie odczytywano 5191 plików i łącznie 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 algorytmami. Według danych na rysunku 4.3a można zauważyć, KMP w niektórych przypadkach jest wolniejszy, niż algorytm MP, ponieważ posiada większe odchylenie standardowe, co powoduje, że jest mniej stabilny.

Algorytm Boyera-Moore'a wykorzystywany w takich narzędziach jak grep, posiada wolniejszy czas egzekucji co wynika z rys. 4.3a, ale algorytm może zostać napisany w

lepszy sposób. Z powodu implementacji, nie wykorzystywaliśmy ponownie bufora pre-procesora, co wpływało na znaczne spowolnienie algorytmu.

Implementacja, której wyniki widzimy na grafie 4.3a jest znacznie wolniejsza od pozostałych algorytmów. Powodem jest spędzanie znacznej cześć czasu na stworzeniu tablicy pre-procesora. Wiadomym jest, że zawsze sprawdzamy ten sam ciąg w wszystkich plikach w folderze. Istnieje możliwość stworzenia tablicy pre-procasora przy pierwszym użyciu algorytmu, a następnie wykorzystanie tej tablicy w wszystkich odczytach.

Na następnym wykresie 4.3b można zauważyć poprawę, gdy implementacja algorytmu Boyer Moora wykorzystuje ten sam bufor pre-procesora, a pozostałe algorytmy tworzą go od nowa, kiedy otwierany jest kolejny plik. Celem takiej implementacji było uzyskanie informacji o wpływie ponownego wykorzystania bufora pre-procesora na czas wykonania.

Aby sprawdzić faktyczne wyniki, należało zaimplementować ponowne wykorzystanie bufora pre-procesora dla wszystkich algorytmów. Wyniki z drugiego wykresu potwierdziły wartość ponownego użycia bufora pre-procesora w prędkości wykonania algorytmu.

Kolejny wykres 4.3c przedstawia implementacje, w której wszystkie algorytmy tworzą i wykorzystują ponownie ten sam bufor pre-procesora i również wykorzystują bufor, który będzie przechowywał zawartość plików. Taki bufor wcześniej był alokowany automatycznie przez Golang, co powodowało, że mogliśmy w niektórych przypadkach, prosić system operacyjny o dodatkową pamięć podczas wykonywania algorytmu. Gdy na początku programu utworzymy taki bufor sami, algorytm Boyera Moora odnotował 5 % poprawę 4.3c w stosunku do poprzedniej implementacji.

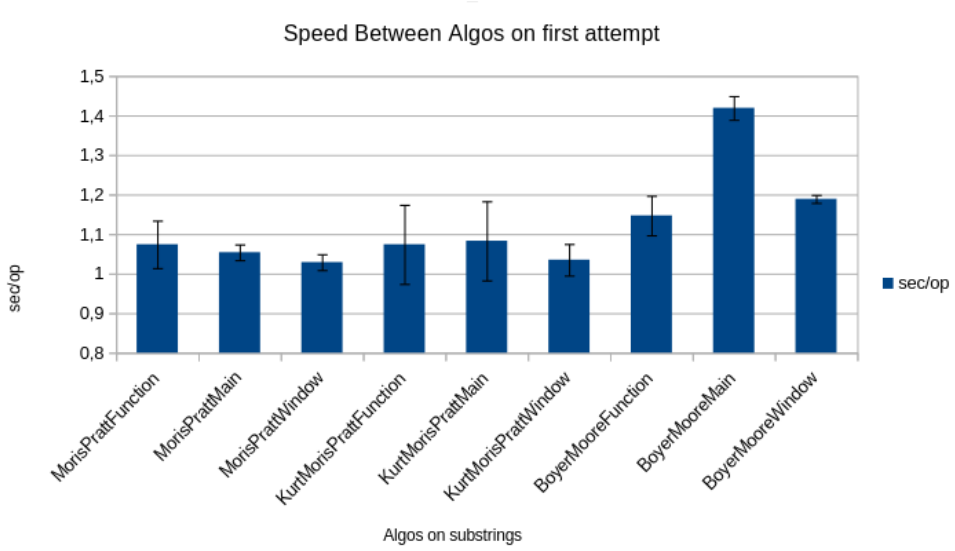
Niestety statyczny bufor przechowujący plik, należy alokować, znając rozmiar największego pliku w folderze, który wynosił 11 MB. Było tak gdyż odrzucaliśmy obrazy. Moglibyśmy przed rozpoczęciem algorytmu sprawdzać rozmiar maksymalny pliku, ale to wydłuży czas działania.

Istnieje też sytuacja w której nie chcielibyśmy tego ograniczać, ponieważ nie znamy największego pliku, a podanie zbyt małej ilość na bufor pliku spowoduje, że nie otrzymamy poprawnych wyników, gdyż nie zmieści się on w całości do pamięci.

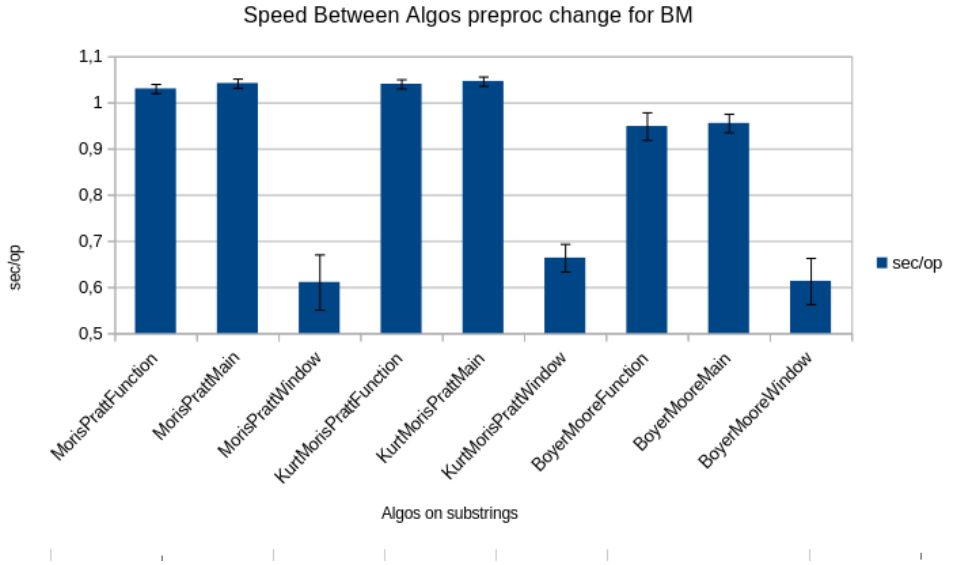
**TODO porównanie wykorzystania pamięci (nie daje dużo info)**

```
goos: linux
goarch: amd64
pkg: github.com/gadzbi123/algorytmy/regular
cpu: Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
BenchmarkMorisPrattFunction-8 1  1041257510 ns/op 264623392 B/op 198976 allocs/op
BenchmarkMorisPrattFunction-8 1  1043653365 ns/op 264645080 B/op 198994 allocs/op
BenchmarkMorisPrattFunction-8 1  1040809273 ns/op 264632096 B/op 198971 allocs/op
BenchmarkMorisPrattFunction-8 1  1043999476 ns/op 264656192 B/op 199008 allocs/op
BenchmarkMorisPrattFunction-8 1  1047795718 ns/op 264641272 B/op 199006 allocs/op
BenchmarkKurtMorisPrattFunction-8 1 1059705156 ns/op 264679568 B/op 199007 allocs/op
BenchmarkKurtMorisPrattFunction-8 1 1044558720 ns/op 264704264 B/op 199016 allocs/op
BenchmarkKurtMorisPrattFunction-8 1 1069466845 ns/op 264722128 B/op 199032 allocs/op
BenchmarkKurtMorisPrattFunction-8 1 1062064344 ns/op 264666880 B/op 199024 allocs/op
BenchmarkKurtMorisPrattFunction-8 1 1062586497 ns/op 264697544 B/op 199018 allocs/op
BenchmarkBoyerMooreFunction-8 1  1163758449 ns/op 264251408 B/op 193795 allocs/op
BenchmarkBoyerMooreFunction-8 1  1142778080 ns/op 264249440 B/op 193831 allocs/op
BenchmarkBoyerMooreFunction-8 1  1127766499 ns/op 264255336 B/op 193817 allocs/op
BenchmarkBoyerMooreFunction-8 1  1169790667 ns/op 264177232 B/op 193775 allocs/op
BenchmarkBoyerMooreFunction-8 1  1128862027 ns/op 264270616 B/op 193811 allocs/op
```

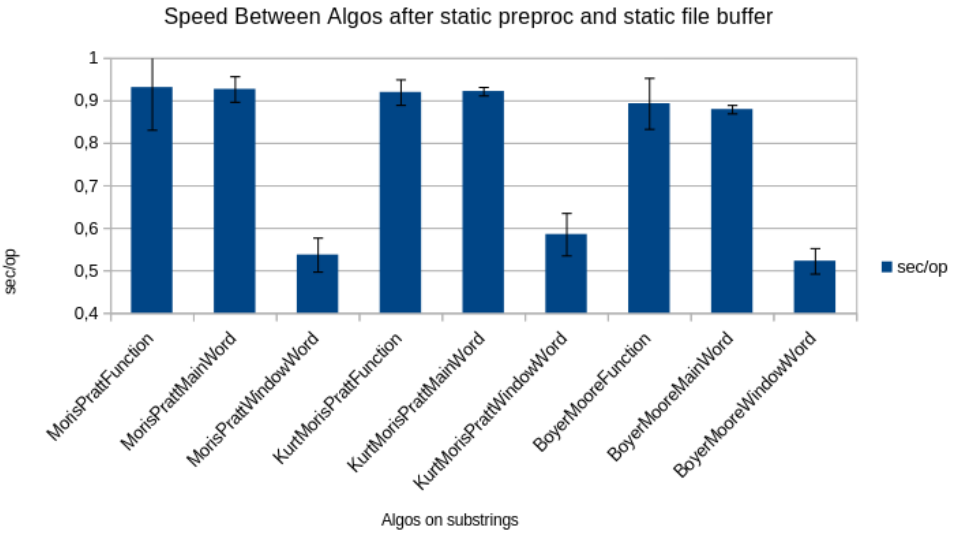
Rysunek 4.2: Przykładowy rezultat performance




(a) Wykres czasów bez statycznego bufora pliku oraz z ponowną rekalkujacją bufora pre-procesora.




(b) Wykres czasów bez statycznego bufora pliku z jednokrotną kalkulacją bufora pre-procesora dla algorytmu Boyer Moore'a.




(c) Wykres czasów z statycznym buforem pliku oraz statycznym buforem pre-procesora dla każdego algorytmu.



Rysunek 4.3: Wykresy kolejnych iteracji na algorytmach

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

[4] 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

GC ang. *Garbage Collector* - Zbieracz śmieci w programie

# 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