Politechnika Śląska

PRACA MAGISTERSKA

Analiza narzędzi wyszukujących zawartość tekstową w systemie Linux

inż. Kacper NITKIEWICZ
Nr albumu: 290409

Kierunek: Informatyka Przemysłowa
Specjalność: Cyberbezpieczeństwo

Prowadzący pracę
dr inż. Adrian Smagór
Katedra Informatyki Przemysłowej
Wydział Inżynierii Materiałowej

Katowice 2025

Tytuł pracy

Analiza narzędzi wyszukujących zawartość tekstową w systemie Linux

Streszczenie

Analiza algorytmów wyszukujących zawartość tekstową w ASCII, sprawdzenie szybkości działania oraz ich wydajności w systemie Linux. Algorytmy zostały porównane w języku Golang, który jest zaopatrzony w wiele narzędzi testujących. Porównano poprawność wyszukań oraz wydajność podobnych programów w konsolowym interfejsie użytkownika na zbiorze archiwów.

Słowa kluczowe

Algorytmy, Linux, Wyszukiwanie, Wydajność, Optymalizacja

Thesis title

Analysis of text search tools in Linux system

Abstract

Analysis of algorithms for searching textual content in ASCII, verification of performance and efficiency in the Linux system. The algorithms were compared in the Golang language, which is equipped with many testing tools. The correctness of searches and the performance of similar programs in the console user interface were compared on a set of archives.

Key words

Algorithms, Linux, Searching, Performance, Optimization

Spis treści

Rozdział 1

Wstęp

1.1 Wprowadzenie do problemu

Analiza przeszukiwania struktur danych o dużych rozmiarach z wieloma typami danych stanowi istotne wyzwanie w dziedzinie inżynierii oprogramowania i zarządzania danymi.

Jednym ze sposobów zachowywania danych jest archiwizacja plików. Innym rozwiązaniem redukującym rozmiar danych jest kompresja. Takie rozwiązanie jest bardzo przydatne w przypadku chęci zmniejszenia ilości danych przechowywanych, a także w dystrybucji danych dla innych użytkowników.

Otrzymana biblioteka jako zbiór książek i dokumentów była przekazana do analizy. Celem przekazania było wykonanie odszukań tekstu w zbiorze i zlokalizowania ścieżki pliku, znalezionego tekstu.

W kwestii technicznej należało rozważyć sposób efektywnego zarządzania pamięcią w przypadku czytania dużej ilości danych z dysku, jak i opóźnienia związane z wydajnością operacji I/O [8].

Problem wyszukiwania danych nastąpił w momencie wyszukiwania dużej ilości zawartości danych. Oryginalne archiwum zajmuje 15 GB danych, gdzie pliki są zarchiwizowane, co utrudnia odczytanie z nich danych.

Wykonanie operacji odczytu archiwów będzie miało kluczowe znaczenie w otrzymaniu informacji o miejscu znajdowania się treści, choć może znacznie wpłynąć na kompleksowość rozwiązania [14].

Implementacja poznanych algorytmów pozwoliła na określenie, który algorytm optymalnie wyszukuje zawartość w wykorzystywanym zbiorze danych. A niewielkie różnice sposobu odczytu danych wpływały na prędkość wydajność wyszukania.

Dodatkowo odczytywanie archiwów w archiwach wymaga kilkukrotnego wyodrębnienia danych. Aby otrzymać odpowiednią ilość wyników, należy wyodrębnić wszystkie zagnieżdżone archiwa. Zasadniczo archiwa po wydobyciu, tworzą drzewo plików, w którym mogą znajdować się kolejne archiwa, z kolejnym drzewem plików itd. Odczytanie zawartości

odbędzie się poprzez przejście po drzewie każdego z wyodrębnionego archiwum.

1.2 Cel Pracy

Celem niniejszej pracy jest analiza algorytmów wyszukujących zawartość tekstową w standardzie ASCII oraz ocena ich efektywności w systemie operacyjnym Linux. Kluczowym zagadnieniem badawczym jest porównanie różnych metod przeszukiwania tekstu pod względem szybkości działania oraz dokładności. Skupiono się na implementacjach tych algorytmów w języku programowania Golang, który oferuje rozbudowane narzędzia testujące i profilujące kod.

Praca ma na celu nie tylko teoretyczne zestawienie wybranych algorytmów, ale także ich praktyczną implementację i porównanie w rzeczywistym środowisku obliczeniowym. W tym kontekście istotnym aspektem jest zbadanie wydajności różnych podejść do przeszukiwania tekstu w dużych zbiorach danych, w tym w archiwach z zagnieżdżonymi folderami.

Kolejnym celem jest ocena poprawności działania zaimplementowanych algorytmów, co oznacza sprawdzenie, czy znajdują one wszystkie wystąpienia szukanych fraz w sposób zgodny z oczekiwaniami. W szczególności badane będą przypadki graniczne, takie jak wyszukiwanie w dużych plikach, przeszukiwanie czy analiza wpływu długości wzorca na czas wyszukiwania.

Ostatecznym celem pracy jest dostarczenie rekomendacji dotyczących wyboru odpowiedniego algorytmu wyszukiwania tekstowego w zależności od specyfiki zadania. Analiza porównawcza umożliwi wskazanie rozwiązań optymalnych pod względem wydajności oraz zastosowania w różnych warunkach systemowych.

1.3 Zakres Pracy

Zakres pracy obejmuje szczegółową analizę algorytmów wyszukiwania tekstu w Golang. W pierwszej części pracy przedstawione zostaną teoretyczne podstawy algorytmów wyszukiwania, w tym klasyczne podejścia, takie jak algorytm Knutha-Morrisa-Pratta, Boyera-Moore'a oraz inne techniki wyszukiwania w tekście ASCII.

Kolejnym etapem będzie implementacja wybranych algorytmów w języku Golang oraz ich optymalizacja pod kątem wydajności. Przeprowadzone zostaną testy porównawcze, w których oceniana będzie szybkość wyszukiwania oraz skuteczność w znajdowaniu wzorców tekstowych. Dodatkowo uwzględniona zostanie analiza wpływu wielkości pliku oraz długości wzorca na efektywność działania poszczególnych metod.

Praca obejmuje również testowanie wydajności algorytmów w rzeczywistym środowisku systemu Linux. Badania będą prowadzone w konsolowym interfejsie użytkownika, gdzie zaimplementowane algorytmy zostaną przetestowane na rzeczywistych zbiorach archiwalnych. Zostaną również porównane czasy wykonania wyszukiwania w zależności od długości

frazy.

Na zakończenie pracy zostaną zaprezentowane wyniki przeprowadzonych badań wraz z wnioskami dotyczącymi efektywności analizowanych algorytmów. Na podstawie uzyskanych wyników zostaną sformułowane rekomendacje dotyczące stosowania poszczególnych metod wyszukiwania tekstowego w zależności od rodzaju danych oraz wymagań wydajnościowych.

Rozdział 2

Analiza tematu wyszukiwania tekstu

2.1 Sformułowanie problemu

Wyszukiwanie tekstu w systemach towarzyszy ludziom od początków istnienia maszyn. Jednak pierwsze komputery nie posiadały ogromnych ilość pamięci. Z biegiem lat zaistniała potrzeba odkrycia algorytmów wyszukujących zawartość pamięci. Procesor Intel 8008 zaprezentowany w 1972 posiadał jedynie 14-bitową magistralę adresową, co pozwalało na 16 KB pamięci [3]. Obecna ilość pamięci, którą można otrzymać po założeniu konta z chmury Google'a to 15 GB [6], co stanowi milion maksymalnych bloków pamięci w urządzeniu ze wspomnianym procesorem Intela. Należy zauważyć, że lokalne zasoby pamięci znacznie przewyższają te, które możemy otrzymać za darmo w chmurze.

Zasadniczym problem naszej pracy jest wyszukiwanie zawartości tekstowej ogromnej ilość plików w różnych formatach. Takie podejście może okazać się problematyczne w przypadku plików dźwiękowych, filmowych czy zdjęć wszelkiego rodzaju. Dodatkowym problemem może okazać się rozwiązanie problemu przeszukiwania zagnieżdżonych w sobie archiwów.

2.2 Dostępne rozwiązania

Podjęcie problemu wyszukiwania plików po nazwach oraz zawartości jest bardzo złożonym i trudnym problemem w sferze programistycznej. Istnieje kilka rozwiązań tego problemu. Narzędzia takie jak **find**, **grep** czy **ripgrep** [2] pozwalają na wyszukiwanie tekstu. Narzędzia te nie są przystosowane do znalezienia pełnej ścieżki pliku dla archiwów, a są jedynie w stanie określić kontekst znalezienia, bez dokładniej ścieżki pliku.

2.2.1 Przykładowe narzędzia dostępne do wykorzystania

Narzędzie **find** to znane i popularne narzędzie w systemie z jądrem Linux. Jest ono bardzo często wykorzystywane do znajdowania plików w systemie po nazwie, jednak nie

jest przystosowane do znajdowania zawartości plików.

```
find /biblioteka -name '*.pdf'
find /biblioteka -path 'książki/*.docx'
find /biblioteka -name '*.py' -not -path '*/site-packages/*'
find /biblioteka -maxdepth 2 -size +500k -size -10M
find /biblioteka -type f -empty -delete
```

Rysunek 2.1: Przykład użycia programu find

Jak pokazano na rysunku (rys. 2.1) te przykładowe komendy find pozwalają na wyszukanie zawartości folderu biblioteka, a kolejne argumenty pozwalają na doprecyzowanie określonych parametrów pliku. Argument **-name** pozwala na wyszukanie wszystkich słów odpowiadających nazwie pliku, jednak nie uwzględniają ścieżki. Argument **-path** pozwala na wyszukanie plików, które odpowiadają podanej ścieżce, a nie jedynie nazwie pliku.

Połączenie argumentów **-name** oraz argumentu **-not** z **-path** umożliwi wyszukanie plików z rozszerzeniem .py. Dodatkowo odrzucone zostaną pliki z paczek pobocznych (ang. *site-packages*).

Kolejna komenda wyszuka wszystkie pliki znajdujące się tylko w folderze /biblioteka (**-maxdepth 2**) oraz jednym zagnieżdżonym folderze poniżej. Dodatkowo pokazane zostaną pliki większe niż 500 KB i mniejsze niż 10 MB.

Ostatnia komenda z rysunku (rys. 2.1) daje możliwość usunięcia (**-delete**) plików (nie folderów) **-type f**, które są puste (**-empty**).

Narzędzie to niestety nie ma możliwości przeszukania folderów z pliku archiwum. Aby dokonać takiego przeszukania, należy wykonać dekompresję danych przy użyciu tar lub innego podobnego narzędzia. Taki proces nie jest jednak prosty w przypadku wielu typów archiwów.

Do przeszukiwania zawartości plików dobrze nadaje się narzędzie grep, które jest dostępny narzędziem GNU's not unix. Jego działanie jest dość podobne do **find'a**, gdyż posiada on możliwość wyszukiwania treść w plikach tekstowych. Narzędzie grep również nie posiada też możliwość szukania zawartości archiwów oraz nie wspiera wielu formatów. Jest możliwość binarnego przeszukania plików, lecz archiwa nie będą odczytane w przypadku użycia kompresji.

Na rysunku powyżej przedstawione są przykładowe komendy programu grep (rys. 2.2). Pierwsza komenda wyszukuje tekst 'szukany-tekst' w pliku /biblioteka/plik1.txt. Kolejna komenda pozwala nam przeszukać rekurencyjnie wszystkie pliki w folderach znajdujące się w /biblioteka (**-r**). Trzecia komenda wyszuka wszystkie wystąpienia 'szukany-tekst' ignorując przy tym wielkość liter. To pozwoli wyszukać tekst posiadający treść 'szukany-tekst' w pliku. Dodając argument **-w** wyświetla się liczba wiersza, w którym znajdowało się dopasowanie w pliku.

```
grep "szukany-tekst" /biblioteka/plik1.txt
grep -r "szukany-tekst" /biblioteka
grep -i "szukany-tekst" /biblioteka/plik1.txt
grep -w "szukany-tekst" /biblioteka/plik1.txt
grep -C 2 "szukany-tekst" plik1.txt
cat /biblioteka/plik1.txt | grep -v "szukany-tekst"
```

Rysunek 2.2: Przykład użycia programu grep

Przedostatnia komenda pozwala na pokazanie kontekstu znalezionego dopasowania (rys. 2.2). Jeżeli 'szukany-tekst' znajduje się w linii 25, to program przedstawi nam linie od 23 do 27 z wyszczególnioną treścią, którą wyszukiwano. Ostatnia komenda przyjmuje treść innego programu przy pomocy standardowego wejścia strumieniowego. Ta komenda zwróci nam wszystkie linie, w których 'szukany-tekst' nie występuje.

Kolejnym narzędziem jest **ripgrep**. Nie jest on domyślnie instalowany na większości systemów Linux, ale pozwala na szybsze wyszukiwanie z powodu wykorzystania równoległości wątków.

```
rg "szukany-tekst" /biblioteka
rg -i "szukany-tekst" /biblioteka
rg -l "szukany-tekst" /biblioteka
rg -c "szukany-tekst" /biblioteka
rg -U "szukany\ntekst" /biblioteka
rg -z "szukany-tekst" /biblioteka/archiwum.zip
```

Rysunek 2.3: Przykład użycia programu ripgrep

Przykładowa komenda pozwalające na wyszukanie frazy 'szukany-tekst' we wszystkich folderach i podfolderach /biblioteka (rys. 2.3). Kolejna komenda pozwala na wyszukanie wszystkich fraz z pominięciem wielkości znaków. Podając parametr **-l** otrzymano nazwy plików bez linii, w których wystąpiła fraza. Parametr **-c** zwróci nam ilość wystąpień tego wyrazu w pliku. Parametr **-U** pozwala na znalezienie wzorca występującego przez kilka linii. Parametr **-z** pozwala na przejrzenie nierozpakowanego archiwum. Ostatnia opcja jest bardzo dobra dla plików, które znajdują się w zbiorze danych.

2.3 Odniesienia do literatury

Istnieje wiele odniesień do wyszukiwania danych w literaturze. Problem wyszukiwania tekstu w dobie internetu, gdzie wiele zasobów przechowywana jest w chmurze, wymaga stworzenia rozwiązania pozwalającego na szybkie znajdowanie treści. Indeksowanie stron

przyspiesza wyszukiwanie, ale wymaga dodatkowej przestrzeni pamięci na przechowywanie zaindeksowanych danych [17].

Ilość wydobywanych i dostarczanych danych poprzez operacje wejścia i wyjścia stanowi duże wyzwanie oraz wymaga wykorzystania skomplikowanych algorytmów i jest skomplikowanym problemem.

Operacje wejścia-wyjścia (I/O) stanowią kluczowy element w architekturze komputerowej, wpływając na ogólną wydajność systemu. W artykule 'W skrócie o wydajności pamięci' porównano zarządzanie danymi w komputerze do organizacji przedmiotów w domu, podkreślając znaczenie efektywnego rozmieszczenia i dostępu do zasobów. W kontekście operacji I/O, analogie do fizycznych obiektów pomagają zrozumieć, jak istotne jest minimalizowanie opóźnień. Kolejnym istotnym aspektem jest zarządzanie pamięcią podręczną (cache) w kontekście operacji wejścia / wyjścia. Efektywne wykorzystanie pamięci cache może znacząco zmniejszyć opóźnienia dostępu do danych, jednak wymaga to zaawansowanych mechanizmów koherencji, zwłaszcza w systemach wieloprocesorowych [1].

Jednym z głównych wyzwań w operacjach I/O jest różnorodność i zmienność urządzeń peryferyjnych. Urządzenia te różnią się pod względem szybkości działania, protokołów komunikacyjnych oraz sposobu obsługi przez system operacyjny. W notatkach dotyczących systemów operacyjnych zwraca się uwagę na problem identyfikacji źródła przerwania. Konieczność obsługi współbieżnej wielu urządzeń komplikuje proces zarządzania operacjami I/O co opisane zostało na wykładzie [8].

Na ten temat w swojej pracy doktorskiej odniósł się też dr Artur Malinowski [12]. W tej pracy do złożoności systemów odzyskiwania danych (str. 24). Dane przetrzymywane w określonym obszarze pamięci L1, L2, L3 znajdują się bliżej niż na dysku SSD lub HDD i przez to zostaną znacznie szybciej odszukane.

Artykuł 'Love your cache' [18] przedstawia kompleksowe podejście do zarządzania pamięcią podręczną w kontekście nowoczesnych aplikacji internetowych. Przede wszystkim, artykuł podkreśla znaczenie strategicznego podejścia do buforowania treści w pamięci podręcznej przeglądarki. Autor wskazuje, że standardy dotyczące pamięci podręcznej pochodzące z 1999 roku nie są w pełni dostosowane do współczesnych wymagań aplikacji internetowych. Proponowane jest wprowadzenie bardziej zaawansowanych mechanizmów kontroli, które pozwalają na optymalizację wydajności przy jednoczesnym zachowaniu aktualności danych.

Istotnym elementem omawianym w artykule jest koncepcja odcisków cyfrowych w nazwach plików. Polega ona na dodawaniu unikalnych identyfikatorów do nazw zasobów, co umożliwia efektywne zarządzanie przechowywanym danymi w pamięci podręcznej. Jest to szczególnie ważne w kontekście optymalizacji wydajności.

Podczas konferencji TREC [7] jeden z zespołów napotkał problem zduplikowanych danych. Chęć wydobycia danych w celu utworzenia tekstów wymagało usunięcia duplikacji dokumentów oraz wyborze tej treści, która posiada najwięcej wartości, co mogłoby być

dodatkowym aspektem do wzbogacenia pracy.

Archiwizacja danych stanowi istotne wyzwanie w różnych dziedzinach, od medycyny po zarządzanie dokumentami historycznymi. Analiza systemów archiwizacji obrazów medycznych oraz procesów ekstrakcji piśmiennictwa wskazuje na kluczowe problemy i rozwiązania stosowane w tych obszarach. W dokumencie o technicznej formie proponowanych rozwiązań problemów można w 'Przegląd otwartych rozwiązań systemów archiwizacji i komunikacji obrazów medycznych' [11].

Archiwa przechowywane mogą też być w różnym stanie. Niektóre dane mogą być uszkodzone z powodu złego przechowywania, sposobu pozyskanych danych lub niepoprawnej implementacji odczytu danych z archiwów. W artykule o digitalizacji piśmiennictwa można znaleźć analogię do rozwiązywanego problemu [13].

W książce W. Curtis Prestona można przeczytać o problemie odczytania tych danych w przypadku uszkodzonych zawartości [15]. Sugeruje on wiele narzędzi pozwalających na odzyskanie zawartości z archiwów i problem z radzeniem sobie w przypadkach uszkodzenia danych.

Problemem odczytywania danych mocno skompresowanych, zajmowali się również doktorzy z Uniwersytetu w Dublinie [9]. Odnosili się oni do skomplikowania odczytu i redundancji w przypadku transportu danych przez sieć.

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 na sprawdzaniu każdego bajtu, a jego implementacja jest traktowana jako naiwna. Złożoność czasowa tego rozwiązania wynosi $O(len(pattern) * len(substring))$, gdzie len(pattern) to długość bloku przeszukiwanego, a len(substring) to długość tekstu szukanego. Nie jest wykonana żadna optymalizacja w tym algorytmie, jak np. przesunięcie się do następnej instancji znaku powtórzonego. Przykładową implementację algorytmu można znaleźć na rysunku (rys. 2.4).

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

Rysunek 2.4: Przykład algorytmu brute force

2.4.2 Interpretacja prefixo-sufiksu

W algorytmie Morrisa-Pratta (MP) oraz KMP prefikso-sufiks jest kluczowym elementem, który pozwala na efektywne wyszukiwanie wzorca w tekście. Prefikso-sufiks to ciąg znaków, który jest jednocześnie prefiksem (początkiem) i sufiksem (końcem) danego wzorca, ale nie jest całym wzorcem. Innymi słowy, jest to wspólna część występująca zarówno na początku, jak i na końcu wzorca.

W algorytmie Morrisa-Pratta (MP) prefikso-sufiks stanowi fundamentalny element sposobu wyszukiwania wzorca w tekście. Jest to specyficzna sekwencja znaków, która występuje jednocześnie na początku (jako prefiks) i na końcu (jako sufiks) analizowanego wzorca. Ta właściwość pozwala algorytmowi na znacznie efektywniejsze działanie w porównaniu do prostego, naiwnego przeszukiwania tekstu.

Działanie prefikso-sufiksu można najlepiej zrozumieć na konkretnym przykładzie. Weźmy wzorzec 'ABABCA'. W tym przypadku możemy znaleźć różne prefiksy (takie jak 'A', 'AB', 'ABA', 'ABAB', 'ABABC') oraz sufiksy ('A', 'CA', 'BCA', 'ABCA', 'BABCA'). Analizując te zbiory, znajdujemy części wspólne, które spełniają definicję prefikso-sufiksu. Dla tego konkretnego wzorca, 'AB' jest przykładem prefikso-sufiksu, ponieważ występuje zarówno na początku, jak i końcu pewnego prefiksu wzorca ('ABAB').

Kluczowym elementem algorytmu MP jest wykorzystanie prefikso-sufiksów do konstrukcji tablicy prefiksowej (**preproc**). Tablica ta przechowuje informacje o długościach najdłuższych prefikso-sufiksów dla każdej pozycji we wzorcu. To właśnie ta struktura danych pozwala algorytmowi na inteligentne przesuwanie wzorca w przypadku wystąpienia niedopasowania, eliminując potrzebę cofania się w tekście. Wykorzystanie prefikso-sufiksów w algorytmach MP i KMP prowadzi do znaczącej optymalizacji czasowej.

W praktycznych zastosowaniach, zrozumienie koncepcji prefikso-sufiksu jest kluczowe w celu zrozumienia działania algorytmów omówionych w tej pracy. Właściwości prefikso-sufiksów znajdują również zastosowanie w innych algorytmach tekstowych, jak na przykład w algorytmie Aho-Corasick, używanym do jednoczesnego wyszukiwania wielu wzorców. Ta uniwersalność koncepcji prefikso-sufiksu sprawia, że jest ona jednym z fundamentalnych pojęć w dziedzinie algorytmów tekstowych. Algorytm wyszukiwania wielu wzorców nie będzie omawiany w tej pracy.

2.4.3 Algorytm Morisa-Pratta

Algorytm Morisa-Pratta, jest algorytmem wykorzystującym możliwość procesowania łańcucha w tekście, aby dopasować wcześniej odpowiadającą część zbioru (rys. 2.5). Polega on na wykorzystaniu faktu istnienia pasującego prefikso-sufiksu. Pozwala on na pominięcie porównania znaków, które się powtarzają w łańcuchu poszukiwanym.

Dzięki wykorzystaniu tej zależności można uniknąć cofania się indeksu i. Od teraz jako tablicę przechowującą informacje o przesunięciu w przypadku błędnego znaku, którą

```
preproc := make([]byte, len(substr)+1)
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 uprzedniego procesowania tekstu szukanego

zainicjowano na rysunku (rys. 2.5) jako tablica **preproc**.

```
  res := []int{}
  curr := 0
  found := 0
  for i := 0; i < len(pattern); i++ {
    for (curr > -1) && (substr[curr] != pattern[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 procesowania łańcucha poszukiwanego w algorytmie Morisa Pratta

Tablice **preproc** należy wypełnić poprzednią wartości tak długo, aż zaistnieje różnica pomiędzy obecnym, a następnym znakiem tablicy szukanej (tablicy **substr**). W przypadku różnicy obu znaków — zwiększa się wartość zapisywaną do tablicy **preproc** o odległość różnicy znaków. W ten sposób następnym razem będzie możliwość pominięcia porównania tych znaków.

W drugim etapie można wykorzystać wcześniej przygotowaną tablicę przemieszczeń **preproc** (rys. 2.6), aby obliczyć przesunięcia w przypadku znalezienia niepasującego prefiksu. Dzięki temu zwykle dłuższy tekst znajdujący się we wzorcu **pattern** można przeanalizować szybciej niż w przypadku algorytmu brute-force.

W podstawowej bibliotece języka Golang, w pakiecie *strings* istnieje implementacja

```
1  func Index(pattern, substr string) int {
2    n := len(substr)
3    switch {
4    case n == 0:
5      return 0
6    [...]
7    case n > len(pattern):
8      return -1
9    case n <= bytealg.MaxLen: // Zwykle ten przypadek uzyty
10     // brute-force kiedy pattern jest krotki
11     if len(pattern) <= bytealg.MaxBruteForce /* max == 64 */{
12       return bytealg.IndexString(pattern, substr)
13     } else {
14     // implementacja BM
15     [...]
16     }
17   }
18 }
```

Rysunek 2.7: Szukanie łańcucha w standardowej bibliotece Golang

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 **pattern** wynosi mniej lub równo 64 bajty (rys. 2.7). W przypadku większego tekstu przeszukiwanego wykorzystuje się podobny algorytm do implementacji BM. Wartość stała **bytealg.MaxBruteForce** jest zadeklarowana w innym miejscu, ale jej wartość wynosi 64, o czym informuje komentarz. Do pełnej implementacji zamieszczono odniesienie [5].

W implementacji wyszukiwania w Golang, gdy bufor szukany jest mniejszy niż 64 bajty, obliczenie bufora wcześniejszego procesowania **BWP** nie jest wykonywane, wiec stosuje się algorytm brute-force. Jeżeli wzorzec jest większy niż 64 bajty to wykonuje się wyszukanie z pomocą **BWP**.

W Golang, gdy wzorzec jest większy niż 64 znaki, to wykonuje się inny algorytm, który posiada dodatkową walidację w przypadku odkrycia wyniku fałszywie dodatniego. Algorytm Morisa-Pratta nie potrzebuje takiej walidacji.

2.4.4 Algorytm Kurta-Morisa-Pratta

Kolejny rozpatrywany algorytm jest implementacja, rozszerzająca poprzednią implementację przedstawioną w algorytmie Morisa-Pratta. Różnice można zauważyć na podstawie rysunku (rys. 2.8).

Gdy nie osiągnięto długości łańcucha i obecny znak jest równy temu, który znajduje się w łańcuchu szukanym (rys. 2.8 linijka 10), to można wykonać skok do znaku znajdującego

```
1  preproc := make([]int, lensubstr+1)
2  preproc[0] = -1
3  curr := -1
4  for i := 1; i <= lensubstr; i++ {
5    for (curr > -1) && (substr[curr] != substr[i-1]) {
6      curr = preproc[curr]
7    }
8    curr++
9  - preproc[i] = curr
10 + if (i == lensubstr) || (substr[i] != substr[curr]) {
11 +   preproc[i] = curr
12 + } else {
13 +   preproc[i] = preproc[curr]
14 + }
15 }
16 mp.preproc = preproc
```

Rysunek 2.8: Różnica pomiędzy algorytmami KMP i MP

się w tablicy przygotowanej (rys. 2.8 linijka 13). Nie należy sprawdzać kolejnego znak w pętli. Ta różnica powoduje, że algorytm wykonuje się szybciej.

Złożoność tego algorytmu wynosi $O(2 * len(pattern))$, gdzie pattern to długość tekstu przeszukiwanego. W najbardziej pesymistycznym przypadku, gdy nie znaleziono dopasowania, będzie to wymagało ilości operacji powrotu równej długości łańcucha szukanego. Powoduje to, że złożoność obliczeniowa sprowadza się do $O(len(substr) * len(pattern))$, co posiada tę samą złożoność co algorytm naiwny.

wzorzec S	AAAAAABAAAAAABAAAAAA
łańcuch W	AAAAAA
liczba cofnięć	20

Tabela 2.1: Przykład wykorzystania algorytmu KMP

W scenariuszu wykorzystania algorytmu Kurta-Morrisa-Pratt (tab. 2.1) można zaobserwować działanie na przykładzie tekstu przeszukiwanego S = 'AAAAAABAAAAAABAAAAAA' oraz frazy szukanej W = 'AAAAAA'. W tym przypadku algorytm musi sprawdzić każde wystąpienie 'A' przed dotarciem do 'B', co jest bardzo nieefektywne, ponieważ pisany tekst miałby większą różnorodność liter i proces przeszukiwania mógłby zacząć się od symbolu B. Sytuacja pogarsza się wraz ze wzrostem liczby powtórzeń fragmentu 'AAAAAAB'. Mimo że metoda tablicowa **preproc** działa tu sprawnie (bez potrzeby cofania się), to jej jednokrotne wykonanie dla łańcucha szukanego W wykona się podobnie jak w algorytmie brute-force. Proces ten wyszukiwania często wymaga wielokrotnych powtórzonych przebiegów. Wielokrotne przeszukiwanie tekstu S w poszukiwaniu wzorca prowadzi do gorszej wydajności, gdy zbiór danych zawiera tego typu charakterystykę tekstu

i wzorca (powtarzające się litery). Naturalny język posiada rzadziej powtarzające się litery, dlatego 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 $O(len(pattern) + len(substr))$, a w najgorszym przypadku $O(len(pattern) * len(substr))$.

2.4.5 Algorytm Boyera-Moore'a

```
1  i := 0
2  len_str := len(str)
3  len_substr := len(substr)
4  for i+len_substr <= len_str {
5    for j := len_substr - 1; j >= 0; j-- {
6     if str[i+j] != substr[j] {
7       if loc := bm.preproc[str[i+j]]; loc == 0 {
8         if j == len_substr-1 {
9           i += len_substr
10        } else {
11          i += len_substr - j - 1
12        }
13      } else {
14        n := loc - len_substr + j + 1
15        if n <= 0 {
16          i++
17        } else {
18          i += n
19        }
20      }
21      goto loop
22    }
23   }
24   res = append(res, i)
25   if v := bm.preproc[str[i+len_substr-1]]; v != 0 {
26    i += v
27   } else {
28    i += len_substr
29   }
30 }
```

Rysunek 2.9: Główna część wyszukiwania w algorytmie Boyera-Moore'a

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 algorytmowi BM na znacznie efektywniejsze przeskakiwanie fragmentów tekstu (rys. 2.9), 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.

Zaletą algorytmu jest to, że ilość skoków pomiędzy porównaniami jest zwykle większa od 1, a gdy istnieje sytuacja, w której litery w łańcuchu szukanym nie powtarzają się, to można przeskoczyć o długość całego łańcucha szukanego.

```
1  if bm.preproc == nil {
2    l := len(substr)
3    table := make([]int, 256)
4
5    for i := 0; i < l-1; i++ {
6      j := substr[i]
7      table[j] = l - i - 1
8    }
9    bm.preproc = table
10 }
```

Rysunek 2.10: Kalkulacja BWP w algorytmie Boyera-Moore'a

W kontekście implementacji algorytm Boyera-Moore'a wymaga utworzenia tokenów dla każdego znaku w ASCII (rys. 2.10) w buforze wcześniejszego procesowania (**BWP**), co zwiększa złożoność pamięciową w porównaniu do pojedynczej tablicy 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.

W algorytmie BM heurystyka 'złego znaku' w algorytmie Boyer-Moore jest jedną z dwóch głównych heurystyk wykorzystywanych do przyspieszenia procesu wyszukiwania wzorca w tekście. Jej podstawową zasadą jest przesunięcie wzorca w prawo, gdy napotkany zostanie znak w tekście, który nie pasuje do odpowiadającego mu znaku we wzorcu. Wielkość tego przesunięcia jest określana na podstawie ostatniego wystąpienia "złego znaku"we wzorcu, lub jeśli znak nie występuje we wzorcu, wzorzec można przesunąć całkowicie poza ten znak.

Działanie tej heurystyki polega na utworzeniu tablicy przesunięć dla wszystkich możliwych znaków w alfabecie. Dla każdego znaku występującego we wzorcu zapisywana jest pozycja jego ostatniego wystąpienia (licząc od prawej strony wzorca). Gdy podczas porównywania tekstu ze wzorcem napotkany zostanie niedopasowany znak, algorytm

sprawdza jego pozycję w tablicy przesunięć i przesuwa wzorzec w prawo o odpowiednią liczbę pozycji, tak aby ten znak mógł zostać dopasowany do jego ostatniego wystąpienia we wzorcu.

Heurystyka 'złego znaku' jest szczególnie efektywna w przypadkach, gdy wzorzec zawiera znaki, które rzadko występują w przeszukiwanym tekście, ponieważ pozwala na wykonanie dużych przesunięć wzorca. Jest ona również bardzo skuteczna w połączeniu z drugą heurystyką algorytmu Boyer-Moore - heurystyką 'dobrego sufiksu'.

Heurystyka 'dobrego sufiksu' w algorytmie Boyer-Moore jest drugą z głównych heurystyk tego algorytmu, która współpracuje z heurystyką 'złego znaku'. Działa ona w sytuacji, gdy podczas porównywania wzorca z tekstem znajdziemy częściowe dopasowanie — czyli gdy pewien sufiks wzorca został dopasowany do tekstu, ale wystąpiło niedopasowanie na wcześniejszej pozycji. W takim przypadku, algorytm próbuje wykorzystać informację o tym dopasowanym sufiksie, aby wykonać jak największe bezpieczne przesunięcie wzorca.

Implementacja tej heurystyki wymaga wstępnego przetworzenia wzorca i utworzenia tablic pomocniczych. Tablica przechowuje informacje o najdalszym wystąpieniu każdego sufiksu wzorca w jego wcześniejszej części, pod warunkiem, że przed tym wystąpieniem znajduje się inny znak niż we wzorcu. Następnie tablica zawiera długości najdłuższych sufiksów, które są jednocześnie prefiksami wzorca. Te tablice są wykorzystywane do określenia, o ile pozycji można bezpiecznie przesunąć wzorzec w prawo, gdy nastąpi częściowe dopasowanie.

Siła heurystyki 'dobrego sufiksu' ujawnia się szczególnie w przypadkach, gdy wzorzec zawiera powtarzające się podciągi znaków. Jest ona niezwykle skuteczna w sytuacjach, gdy długi sufiks został dopasowany, ale wystąpiło niedopasowanie na wcześniejszej pozycji. W takich przypadkach możliwe jest wykonanie dużych przesunięć wzorca, co znacząco przyspiesza proces wyszukiwania.

Z tego też powodu należy wykonać heurystykę danych, które są analizowane. Można to zrobić na kilka sposobów i zostanie to przedstawione w następnych rozdziałach.

Rozdział 3

Przedmiot pracy

3.1 Rozwiązanie problemu wyszukiwania ciągu znaków w pliku tekstowym

Przed wyborem metody sprawdzającej należy wykonać heurystykę danych. Jest to wymagane, ponieważ wydajność algorytmu jest ściśle powiązana z danymi, które będą przeszukiwane. Jeżeli większość analizowanych danych mają charakter tekstowy, to lepszym rozwiązaniem będzie skorzystanie z ostatniego algorytmu BM (rozdział 2.4.5), jednak w innym przypadku warto wykorzystać jeden z pozostałych.

Przedstawiony program został stworzony do ustalenia typów plików w zbiorze rozpakowanych archiwów (rys. 3.2). Na tym zbiorze testowano działanie poszczególnych algorytmów. Ten zbiór jest mniejszy, aby ograniczyć wpływ czytania z dysku na wyniki badań. Porównano typ plików na podstawie identyfikatora formatu danych MIME. Pozwala on na określenie, z jakim typem danych pracujemy, aby przeszukać dane.

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 odrzucono część z nich z powodu sposobu, w jaki analizowano dane.

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 skompresowane:
   - ".tar.gz",

- ".tgz",
- ".gz",

3. pliki pochodzące od Microsoftu:

- ".doc",

Typ MIME	**Ilość**
image/jpeg	5303
text/html	2473
image/gif	2353
text/xml	1233
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

Innym sposobem na rozwiązanie problemu, jest wykorzystanie dostępnych narzędzi i dostosowanie ich do problemu, który jest rozwiązywany. Wykorzystanie wielu narzędzi zaimplementowanych w jednym celu (**tar**, **gzip**, **grep**) i połączenie ich do skomplikowanej operacji przeszukiwania archiwów zajełoby więcej czasu i wymagałoby obsłużenia błędów każdego z narzędzi. Dodatkowo nie każda treść jest odpowiednia do użycia narzędzia rozpakowującego tar, np. w przypadku skompresowanej treści, gdzie należy wykorzystać narzędzie **gzip**.

3.2 Rozwiązanie problemu zarchiwizowanych plików w zbiorze danych

Zbiór biblioteki, składa się danych zawierających archiwa oraz pliki skompresowane, które zawierają drzewo innych plików skompresowanych i archiwów. Aby odczytać wystąpienia, należy je rozpakować, jednak to powoduje ciągłe wydłużanie się ścieżki, a w Windows jest ona ograniczona do 256 znaków. Na systemie Linux ten limit nie powinien być problemem, gdyż system pozwala na ścieżki długości 4096 znaków, jednak ta różnica może wpływać na jakość działania programu na różnych systemach.

Pliki zawierające archiwum zostaną rozpakowane do osobnego folderu, a następnie wykonane zostanie rekurencyjne wyszukanie zawartość w każdym folderze i pliku. Ta metoda zostanie powtórzona tak długo, aż wszystkie pliki zostaną odczytane i nie będzie już więcej archiwów do rozpakowania.

Aby porównać analizowane programy z implementacją gsearch, potrzeba porównać ilość oczekiwanych rezultatów dla długości słów oraz czas wykonania całego przeszukania.

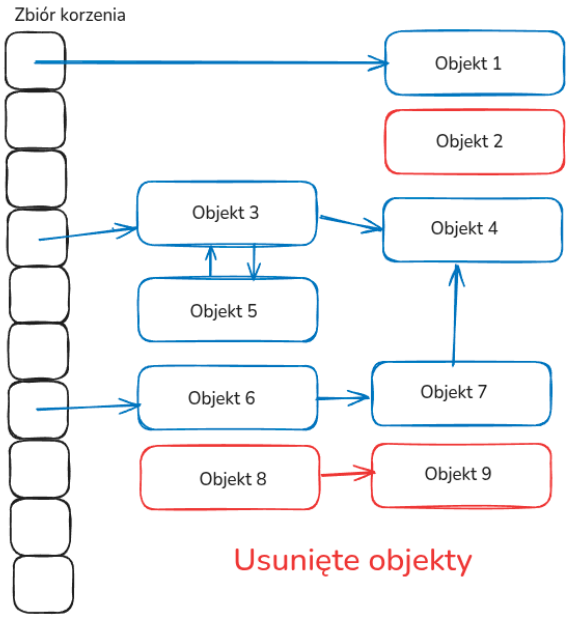
3.3 Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi

3.3.1 Użycie języka Golang

Do utworzenia programu wykorzystam nowoczesny język programowania Golang [16]. 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.

Wadą tego języka może być fakt, iż język nie daje programiście możliwości pełnej kontroli pamięci. Język wykorzystuje ang. *Garbage Collector* (zbieracz śmieci), który jest alternatywą do sposobu manualnej alokacji pamięci. Można zaobserwować działanie Garbage Collectora (GC) (rys. 3.1), jak obiekty 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



Rysunek 3.1: Przykładowe działanie Garbage Collectora w programie

elementy, które przestały być powiązane z jakimkolwiek elementem, więc zostaną zwolnienie z użycia, po zakończeniu operacji sprawdzania [10].

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 przydzieloną pamięć. Problem może skutkować wolniejszą egzekucją oraz pauzami w celu oczyszczenia pamięci.

Drugą metodą, która jest preferowana to warianty 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 przydzielonej pamięci. W przypadku, gdy program nie potrzebuje pamięci, program wewnętrznie ją uwalnia, jednak nie oddaje jej od razu do systemu, jeżeli będzie ona wykorzystywana ponownie. Taki mechanizm zmniejsza częstotliwość operacji systemowych, które wymagają potwierdzenia alokacji, zanim praca na pamięci zostanie wykonana [19].

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

3.3.2 Użycie osobnej biblioteki do dearchiwizacji

Do rozpakowania archiwów danych zastosowano bibliotekę unarr, które posiadała dobrą dokumentację odpowiadającą zadaniu, które należało wykonać. Okazała się problematyczna w stosunku do niektórych typów archiwów.

3.3.3 Rozpakowywanie plików na dysk zamiast w pamięci

Zdecydowano się na rozpakowane danych w folderze tymczasowym. Próba rozpakowywania zawartości w pamięci powodowała uszkodzenie stosu z powodu zbyt dużej ilości danych przechowywanych w pamięci przez Garbage Collector w języku Golang.

```
#!/usr/bin/env python3
import os
import subprocess
from collections import defaultdict
def get_mime_type(file_path):
    result = subprocess.run(['file', '--mime-type', file_path],
    capture_output=True, text=True)
    return result.stdout.split(': ')[1].strip()
def get_file_size(file_path):
    try:
        return os.path.getsize(file_path)
    except Exception:
        return 0
def collect_mime_stats(start_path='.'):
    mime_stats = defaultdict(lambda: {'size': 0, 'count': 0})
    total_size = 0
    total_count = 0
    for root, _, files in os.walk(start_path):
        for file in files:
        file_path = os.path.join(root, file)
        mime_type = get_mime_type(file_path)
        size = get_file_size(file_path)

        mime_stats[mime_type]['size'] += size
        mime_stats[mime_type]['count'] += 1
        total_size += size
        total_count += 1
    return mime_stats, total_size, total_count
def format_size_kb(size_bytes):
    return f"{size_bytes / 1024:.2f}"
def main():
    mime_stats, total_size, total_count = collect_mime_stats()
    sorted_stats = sorted(mime_stats.items(), key=lambda x: x
       [1]['size'], reverse=True)
    format_str = "{:<45} {:>10} KB {:>8}"
    for mime_type, stats in sorted_stats:
        print(format_str.format(
            mime_type,
            format_size_kb(stats['size']),
            stats['count']
        ))
    print(format_str.format(
        "Total",
        format_size_kb(total_size),
        total_count
    ))
```

Rysunek 3.2: Program pozwalający na wyświetlenie ilości danych w formacie MIME

Rozdział 4

Badania

4.1 Metodyka badań

4.1.1 Cel badania

Celem badania jest sprawdzenie wydajności algorytmów wyszukujących na zbiorze danych dostarczonego w celu wyszukania treści. Dodatkowym celem jest porównanie działań zaimplementowanego programu z innymi, podobnymi rozwiązaniami.

4.1.2 Zakres badania

Zakres badań obejmuje porównanie prędkości algorytmów w celu ustalenia, który z nich jest najszybszy pod względem czasu wykonania. Zostanie to ustalone na mniejszym zbiorze rozpakowanych archiwów. Następnie na nierozpakowanym zbiorze archiwów porównane zostaną narzędzia pod względem liczby wyszukań oraz prędkości wyszukań.

4.1.3 Hipoteza badań

Hipotezą badań jest to, że kolejne algorytmy posiadające większe zużycie zasobów na przygotowanie wyszukania, będą wykonywały się szybciej, niż te z mniejszym zużyciem. Hipotezą pomocniczą jest to, iż im większy zbiór informacji zebrany na podstawie łańcucha szukanego, tym większa prędkość algorytmu. Dodatkową hipotezą jest to, iż wykorzystanie *Garbage Collectora* wpływa na stabilny czas działania programu pomiędzy wykonaniami, oraz że czas wykonywania odczytów danych z dysku zajmuje zdecydowaną ilość czasu.

4.2 Badanie benchmarku algorytmów

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

```
1 go test -test.bench=. -benchmem . -benchtime=1x -count=20
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 testu dla algorytmu MP

Przeprowadzenie badań polegało na uruchomieniu komendy w pierwszej linii (rys. 4.1) i wykonaniu funkcji na algorytmie Morisa Pratta. Wykonano 20 testów na wyszukaniu trzech słów, które mogły występować w zbiorze danych, ze względu na wcześniej przeanalizowaną zawartość. Tymi słowami były 'main', 'window', 'function'.

Zmienna **founds** przechowuje miejsce znalezionego ciągu wyszukiwanego, a zmienna **mp** to struktura przechowująca implementacje algorytmu MP oraz bufor wcześniejszego procesowania (**BWP**) dla ciągu wyszukiwanego. Pierwsza implementacja nie posiadała tej struktury i **BWP** był obliczany przy każdym przebiegu algorytmu. To znacznie wpłynęło na prędkość działania algorytmu Boyera-Moora, który posiadał największy rozmiar **BWP**.

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) wykona się przejście (ang. *Walk*) po drzewie plików w **DIR**, który posiada ścieżkę do zbioru danych oraz przyjmuje funkcje zdefiniowaną jako **WalkFunc** jak w definicji (def. 1). Z powodu potrzeby czytania tylko zawartości plików, wykorzystano funkcje **WalkAndFindByAlgo**, która będzie czytała pliki, ale pozwoli algorytmom podanym w argumencie na przeszukanie zawartości w celu znalezienia słowa 'window'.
Po zakończeniu funkcji Walk otrzymano tablice znalezionych **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 z wcześniej ustaloną sumą.

W rezultacie wykonania otrzymano dane o czasie przebiegu funkcji, ilości alokacji oraz ile bajtów wykorzystano na operacje (rys. 4.2). Zebrane dane zostały zaprezentowane oraz omówione w sekcjach 4.3.1 oraz 4.3.2.

```
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

4.3 Zbiór badań

Zbiór danych (tab. 3.1) posiadał znaczną ilość plików pdf, które w niektórych przypadkach można było odczytać. Gdy pdf został stworzony z dokumentu tekstowego takiego jak docx czy odt, to zawartość tekstowa została zawarta w dokumencie pdf. Jeżeli natomiast pdf został stworzony ze skanu książki, nie zapisała się treść tekstowa. Wtedy cała treść jest przechowana w postaci zdjęcia, które nie można odczytać użytym rozwiązaniem.

Możliwość odczytania zawartości daje narzędzie OCR (ang. *Optical Character Recognition*). Narzędzie może służyć do oczytania zawartości plików pdf oraz tekstu z obrazów. Takie rozwiązanie nie zostanie użyte w pracy, gdyż praca nie skupia się na algorytmach sztucznej inteligencji.

Pliki audio to piosenki oraz podcasty, których nie można łatwo odczytać algorytmem. Do oczytania treści z takich plików można wykorzystać narzędzia dokonujące trankrypcji, czyli konwertujące rozpoznaną mowę na tekst, jednak te rozwiązania bazują na sztucznej inteligencji, które nie są przedmiotem pracy.

Kolejnym problemem okazało się odczytanie plików formatu doc, które są starym formatem dokumentów wykorzystywanych przez Microsoft i ich zakodowana treść nie jest łatwa do odczytu. W odróżnieniu od formatu docx, który jest archiwum, narzędzie nie podejmie się odczytania plików doc. Z uwagi na to, że archiwum danych jest dość stare, nie wystąpiły tam dokumenty formatu docx. Gdyby wystąpił można by odczytać zawartość archiwów docx w ten sam sposób jak w przypadku innych archiwów.

Niestety w przypadku starych plików doc, nie ma możliwości rozpakowania treści pliku i otrzymania zawartości. Dodatkowo pliki posiadają nietypowe kodowania, co powoduje dalsze skomplikowanie z odczytania ich treści.

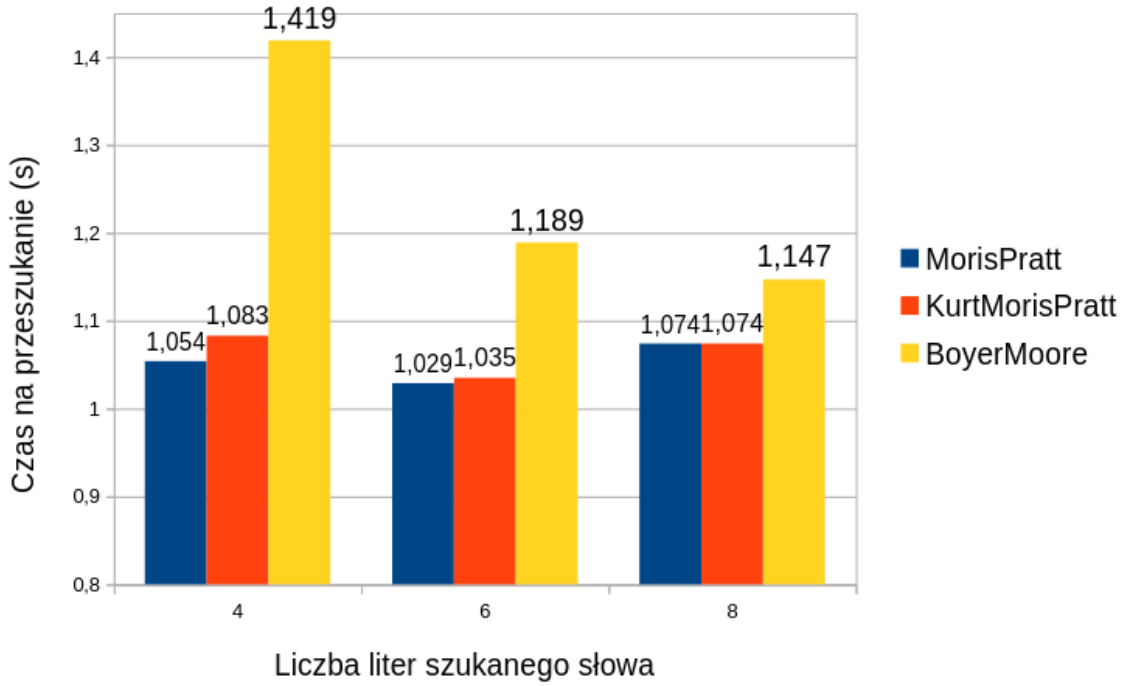
Archiwa posiadają również różne algorytmy kompresji oraz różne sposoby zapisu w zależności od rozszerzenia np. zip, rar, 7z, tar. To powoduje, że należy wykorzystać wiele metod konwersji tych plików do faktycznej struktury możliwej do odczytania. Biblioteka, którą wykorzystano nie obsługuje plików skompresowanych przez gzip. To powoduje, że pliki z rozszerzeniem .gz nie będą otwierane.

Pobrane archiwa posiadały brakujące dane i to powodowało, że biblioteka konwertująca archiwa miała problem z ich otwarciem. Z tego powodu część danych musiała być pominięta, aby program nie został przerwany przez SEGV. Wykorzystana biblioteka musiała zostać niepoprawnie zaimplementowana, gdyż istnieją programy umiejące otworzyć te archiwa, choć nie wszystkie z tych narzędzi działały poprawnie. To wymagało zmniejszenia zbioru przeszukiwanych danych z archiwum i ograniczenia się do 15 GB danych (tab. 4.1).

Pierwszy test wydajnościowy, który został przeprowadzony, sprawdzał wszystkie foldery, w których znajdowały się pliki. Za drugim razem ograniczono się tylko do plików, które mogą posiadać oczekiwaną zawartość, odrzucając zatem część plików ze zbioru. Wykonano testy na 3 algorytmach, gdzie odczytywano 5191 plików i łącznie 240 MB danych. Oto

rezultaty określonych algorytmów.

4.3.1 Porównanie czasów algorytmów



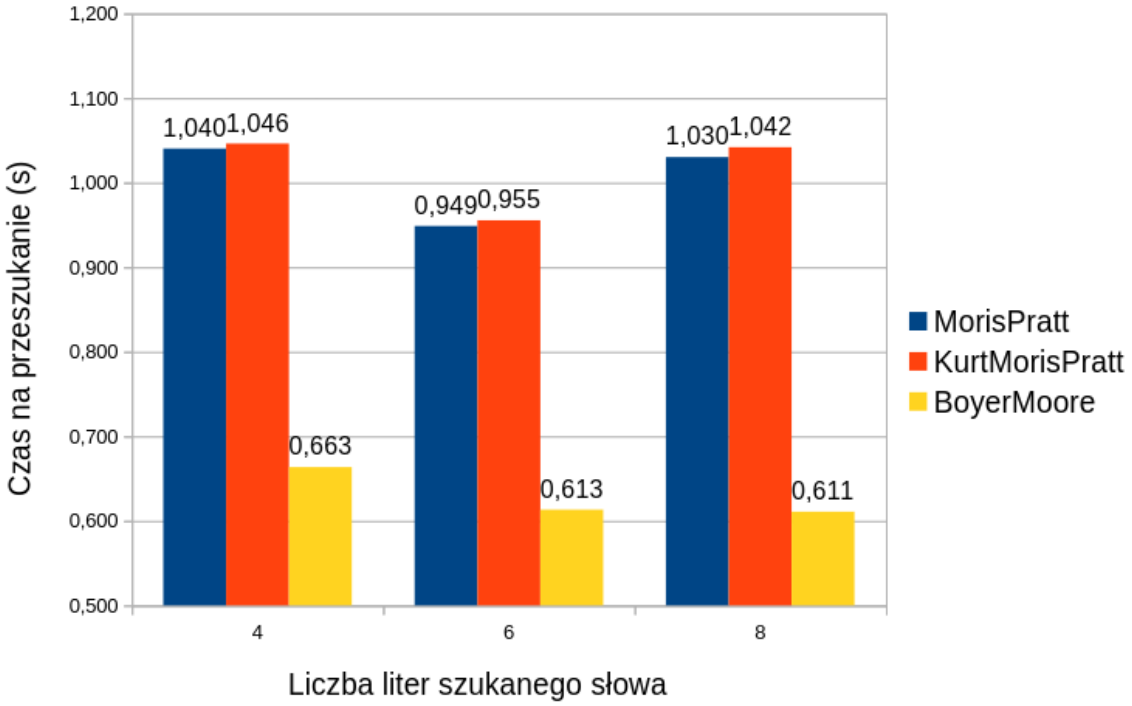
Rysunek 4.3: Wykres czasów bez gotowego bufora pliku oraz z ponowną liczeniem (BWP).

Algorytm Morisa-Pratta jest nieznacznie wolniejszy od algorytmu Kurta-Morisa-Pratta. Jest to spowodowane niewielką optymalizacją pomiędzy tymi dwoma algorytmami. Według danych na rysunku (rys. 4.3) można zauważyć, KMP w niektórych przypadkach jest wolniejszy, niż algorytm MP.

Algorytm Boyera-Moore'a wykorzystywany w takich narzędziach jak grep, ma wolniejszy czas egzekucji, co wynika z rysunku (rys. 4.3), ale algorytm może zostać poprawiony.
Z powodu błędnej implementacji, wykonywano ponowną kalkulację BWP podczas otwarcia nowego pliku, choć szukana fraza pozostawała taka sama. Wykorzystanie wcześniejszego przeliczonego bufora ponownie wpłynęło na znaczne przyspieszenie algorytmu.

Implementacja BM, której wyniki można zobaczyć na wykresie (rys. 4.3) jest znacznie wolniejsza od pozostałych algorytmów. Powodem jest spędzanie znacznej cześć czasu na stworzeniu tablicy wcześniejszego procesowania. Wiadome jest, że zawsze sprawdzany jest ten sam ciąg we wszystkich plikach w folderze. Istnieje możliwość stworzenia tablicy wcześniejszego procesowania przy pierwszym użyciu algorytmu, a następnie wykorzystanie tej tablicy we wszystkich odczytach.

Na następnym wykresie (rys. 4.4) można zauważyć poprawę, gdy implementacja algorytmu Boyer-Moora wykorzystuje ten sam bufor wcześniejszego procesowania,



Rysunek 4.4: Wykres czasów bez statycznego bufora pliku z jednokrotną kalkulacją BWP w implementacji algorytmu Boyera-Moore'a.

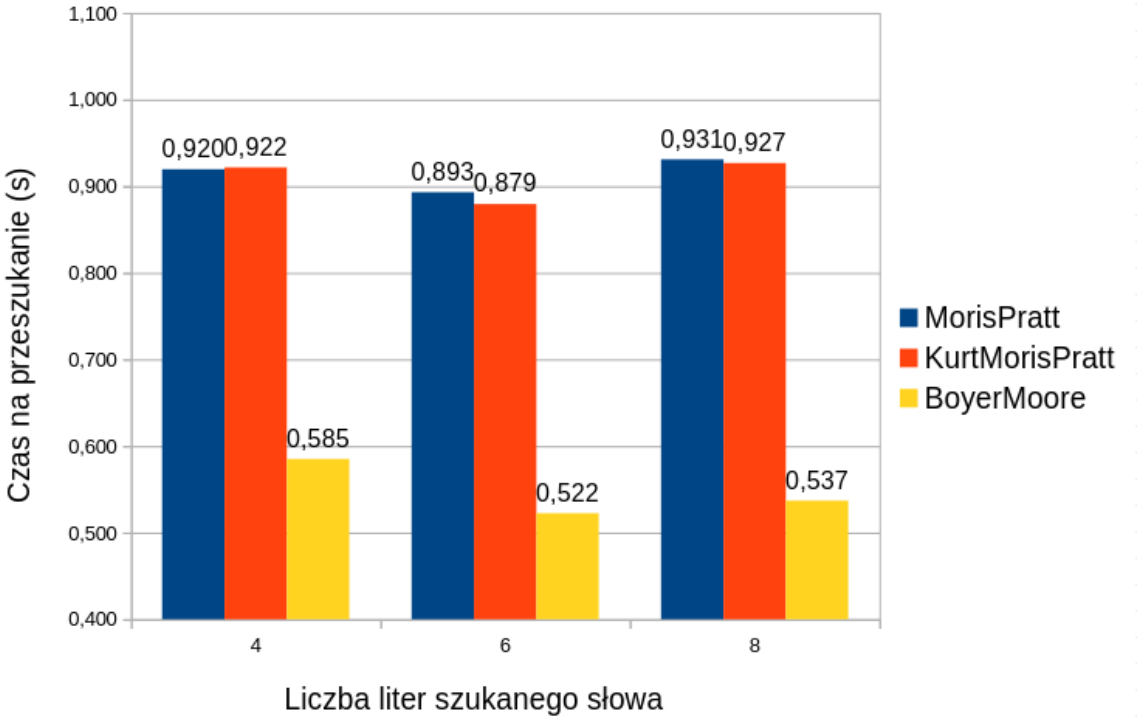
a pozostałe algorytmy tworzą go od nowa, kiedy otwierany jest kolejny plik. Celem takiej implementacji było uzyskanie informacji o wpływie ponownego wykorzystania bufora wcześniejszego procesowania na czas wykonania.

Aby sprawdzić faktyczne wyniki, należało zaimplementować ponowne wykorzystanie bufora wcześniejszego procesowania dla wszystkich algorytmów, a nie tylko dla BM. Wyniki z drugiego wykresu potwierdziły wartość ponownego użycia bufora wcześniejszego procesowania w prędkości wykonania algorytmu.

Ostatni wykres (rys. 4.5) przedstawia implementację wykorzystującą ponownie bufor wcześniejszego procesowania, jak i bufor przechowujący plik. Bufor wcześniejszego procesowania był przydzielany przy każdym otwarciu nowego pliku, co powodowało, że ten bufor mógł być zbierany przez *Garbage Collector*. W przypadku użycia stałego bufora zapewniamy, że program nie będzie się pozbywał bufora, co uniknie proszenia o pamięć systemu. Gdy na początku programu utworzymy bufor przechowywania pliku sami (nie polegając na optymalizacji języka), algorytm Boyera-Moore'a odnotował 5 % poprawę jak na rysunku (rys. 4.5) 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. Można przed rozpoczęciem algorytmu sprawdzać rozmiar maksymalny pliku, ale to wydłuży czas działania.

Istnieje też sytuacja, w której nie powinno się ustawiać największego rozmiaru, ponieważ nie jest on znany w zbiorze. Podanie zbyt małej ilość na bufor pliku spowoduje, że nie



Rysunek 4.5: Wykres czasów ze statycznym buforem pliku oraz jednokrotną kalkulacją BWP dla każdego algorytmu.

zostaną uzyskane wszystkie wyniki z pliku, gdyż nie zmieści się on w buforze pamięci statycznej.

Można wykorzystać pamięć dynamicznie przydzielaną i w przypadku zbyt małego bufora dla danych z pliku, ustawić nowy rozmiar równy zawartości pliku. Zmniejszy to ilość alokowania nowego bufora, ale nie będzie to aż tak efektywne w szybkim wyszukiwaniu danych, ponieważ znacząco zwiększy to czas działania Garbage Collectora.

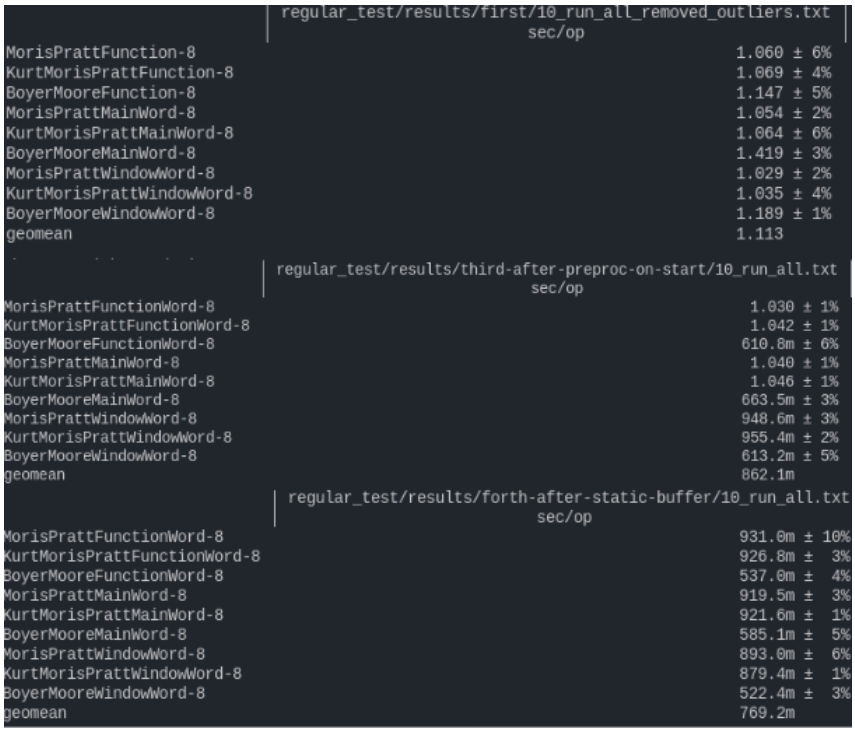
W badaniu porównawczym dla programów w następnej sekcji zdecydowano się na czytanie pojedynczej linii z pliku, który skanujemy. Niestety to podejście powodowało taki sam problem jak ten w przypadku bufora pamięci statycznej.

4.3.2 Omówienie otrzymanych wyników

Otrzymane wyniki z testów wydajności algorytmów zestawiono z przedstawionych danych otrzymanych z programu **benchstat** 4.6. Program pozwolił na zorganizowanie danych przedstawionych na rysunku 4.2.

Z wyjścia programu **benchstat** wynika, że średnia odchylenia standardowego wynosi około 3 %, wiec nie została przedstawiona na wykresach 4.3, 4.4, 4.5.

W tabeli przedstawiono wyniki dla każdego z algorytmów, a wyniki podzielono na liczbę znaków testowanego słowa 4.7. Kolejne wiersze przedstawiają wyniki, gdy nie zastosowano **BWP**, zastosowano **BWP** jedynie dla algorytmu Boyera-Moore'a oraz kiedy wykorzystano **BWP** dla wszystkich algorytmów. Dodatkowo w ostatnim wierszu dodano



```
                                  | regular_test/results/first/10_run_all_removed_outliers.txt
                                  |                                                  sec/op
MorisPrattFunction-8                                                          1.060 ± 6%
KurtMorisPrattFunction-8                                                      1.069 ± 4%
BoyerMooreFunction-8                                                          1.147 ± 5%
MorisPrattMainWord-8                                                          1.054 ± 2%
KurtMorisPrattMainWord-8                                                      1.064 ± 6%
BoyerMooreMainWord-8                                                          1.419 ± 3%
MorisPrattWindowWord-8                                                        1.029 ± 2%
KurtMorisPrattWindowWord-8                                                    1.035 ± 4%
BoyerMooreWindowWord-8                                                        1.189 ± 1%
geomean                                                                       1.113
                                  | regular_test/results/third-after-preproc-on-start/10_run_all.txt
                                  |                                                  sec/op
MorisPrattFunctionWord-8                                                      1.030 ± 1%
KurtMorisPrattFunctionWord-8                                                  1.042 ± 1%
BoyerMooreFunctionWord-8                                                     610.8m ± 6%
MorisPrattMainWord-8                                                          1.040 ± 1%
KurtMorisPrattMainWord-8                                                      1.046 ± 1%
BoyerMooreMainWord-8                                                         663.5m ± 3%
MorisPrattWindowWord-8                                                       948.6m ± 3%
KurtMorisPrattWindowWord-8                                                   955.4m ± 2%
BoyerMooreWindowWord-8                                                       613.2m ± 5%
geomean                                                                      862.1m
                                  | regular_test/results/forth-after-static-buffer/10_run_all.txt
                                  |                                                  sec/op
MorisPrattFunctionWord-8                                                     931.0m ± 10%
KurtMorisPrattFunctionWord-8                                                 926.8m ±  3%
BoyerMooreFunctionWord-8                                                     537.0m ±  4%
MorisPrattMainWord-8                                                         919.5m ±  3%
KurtMorisPrattMainWord-8                                                     921.6m ±  1%
BoyerMooreMainWord-8                                                         585.1m ±  5%
MorisPrattWindowWord-8                                                       893.0m ±  6%
KurtMorisPrattWindowWord-8                                                   879.4m ±  1%
BoyerMooreWindowWord-8                                                       522.4m ±  3%
geomean                                                                      769.2m
```

Rysunek 4.6: Obraz przedstawiający wynik działania programu 'benchstat'

	Moris Pratt			Kurt Moris Pratt			Boyer Moore		
	4	6	8	4	6	8	4	6	8
bez bufora	1,054	1,029	1,074	1,083	1,035	1,074	1,419	1,189	1,147
BWP dla BM	1,040	0,949	1,030	1,046	0,955	1,042	0,663	0,613	0,611
BWP oraz bufor pliku	0,920	0,893	0,931	0,922	0,879	0,927	0,585	0,522	0,537

Rysunek 4.7: Obraz przedstawiający wynik dla wszystkich algorytmów i ich iteracji.

bufor dla przechowywanego pliku.

	4 znaki		
	Moris Pratt	Kurt Moris Pratt	Boyer Moore
bez bufora	1,054	1,083	1,419
BWP dla BM	1,040	1,046	0,663
BWP oraz bufor pliku	0,920	0,922	0,585

Rysunek 4.8: Obraz przedstawiający wynik dla 4 znaków wszystkich algorytmów.

Implementacja algorytmu Morisa-Pratta dla konfiguracji bez bufora, okazała się najszybsza spośród wszystkich algorytmów. Najkrótszy czas wykonania został oznaczony na ciemno zielono w tej iteracji algorytmów 4.8.

Implementacja z buforem wcześniejszego procesowania spowodowała znaczne przyspieszenie algorytmu BM w drugiej iteracji (kolor jasny czerwony). Nie zmieniono implementacji pomiędzy testami w pierwszej i drugiej iteracją dla algorytmów MP i KMP. Otrzymane wyniki pomiędzy dwoma iteracjami mieszczą się w średniej odchylenia standardowego słupka błędu.

Z powodu znacznego przyspieszenia wykonania algorytmu BM, zdecydowano się na dodanie **BWP** w 3 iteracji dla wszystkich algorytmów. Pomimo dodania jedynie statycznego

bufora pliku dla BM, algorytm ten wciąż wykonywał się znacznie szybciej od pozostałych rozwiązań (kolor czerwony). W trzeciej iteracji dodano implementację **BWP** dla MP i KMP. Dodatkowo wprowadzono bufor statyczny plików dla wszystkich algorytmów, aby zmniejszyć prawdopodobieństwo dealokacji bufora pliku.

	6 znaków		
	Moris Pratt	Kurt Moris Pratt	Boyer Moore
bez bufora	1,029	1,035	1,189
BWP dla BM	0,949	0,955	0,613
BWP oraz bufor pliku	0,893	0,879	0,522

Rysunek 4.9: Obraz przedstawiający wynik dla 6 znaków wszystkich algorytmów.

Dla 6 znaków wyniki prezentują się podobnie jak dla czterech 4.9. Ponownie algorytm MP wygrywa nieznacznie z KMP (zielony kolor), jednak po wprowadzeniu **BWP** i bufora pliku algorytm BM wciąż ma przewagę szybkości wyszukiwania (kolor czerwony).

	8 znaków		
	Moris Pratt	Kurt Moris Pratt	Boyer Moore
bez bufora	1,074	1,074	1,147
BWP dla BM	1,030	1,042	0,611
BWP oraz bufor pliku	0,931	0,927	0,537

Rysunek 4.10: Obraz przedstawiający wynik dla 8 znaków wszystkich algorytmów.

W przypadku 8 znaków czas wykonania algorytmów MP i KMP jest identyczny dla pierwszej iteracji (kolor fioletowy 4.10). Algorytm BM wykonuje się szybciej w drugiej i trzeciej iteracji, dokładnie tak samo, jak w poprzednich przypadkach.

	Średnia dla każdej ilości znaków		
	Moris Pratt	Kurt Moris Pratt	Boyer Moore
bez bufora	1,052	1,064	1,252
BWP dla BM	1,006	1,014	0,629
BWP oraz bufor pliku	0,915	0,909	0,548

Rysunek 4.11: Obraz przedstawiający wynik średnie znaków wszystkich algorytmów.

Zestawiając wyniki w postaci średniej ze wszystkich ilości znaków, można porównać prędkości algorytmów w całości w zależności od iteracji 4.11. Z tabeli można wywnioskować, że dla wariantu bez bufora, algorytm MP posiada najszybszą implementację. W przypadku zastosowania **BWP**, który nie posiada znacznych wad, otrzymujemy dwukrotnie szybszy algorytm w odniesieniu do BM oraz o 40% szybszą implementację od wariantu bez **BWP** dla MP.

Pomimo przyspieszenia algorytmów MP i KMP w trzeciej iteracji, nie udało się dorównać prędkości BM z drugiej iteracji, a implementacja BM w trzeciej iteracji uzyskała jeszcze lepszy wynik (czerwony i jasny czerwony 4.11). Niestety zastosowanie bufora statycznego pliku, nie zwiększa znacznie prędkości wykonywania obu algorytmów, a wymaga

od użytkownika programu informacji o największym pliku ze zbioru przeszukiwanego. Z tego powodu w przypadku badania archiwów, zdecydowano się na wykorzystanie algorytmu z **BWP** bez bufora pliku.

4.4 Badanie ilość otrzymanych wyników z programów dla zbioru archiwów

W zbiorze archiwów znajdowało się 15 GB danych różnego rodzaju. Tabela przestawia hierarchiczną listę typów MIME (tab. 4.1).

Wybrano kilka narzędzi, które będą porównywane pod względem liczby wyszukiwań oraz ich prędkości. Narzędzia, które zachowują się podobnie do implementacji autora do **ugrep** (ug), **zgrep** oraz **ripgrep** (rg).

```
[gadzbi@desktop-linux algorytmy]$ time ug -z --zmax=99 main /run/media/gadzbi/GryIFilmy/baza_mgr
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/ksiazki.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/literatura.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/Informatyka.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/konferencja.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/DataMining.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/literatura_old.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/angielski.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/papers.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/polskie.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/michal.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/programy_nauczania.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/matematyka.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/papers (1).zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/skanowane.zip: unsupported zip compression method 0

real    0m0,807s
user    0m0,029s
sys     0m0,099s
```

Rysunek 4.12: Niepoprawne działanie programu ugrep dla archiwów pobranych z chmury

Wiele z tych narzędzi nie działało poprawnie, gdy próbowano odczytać zawartość archiwów. Przykładowo narzędzie ugrep (rys. 4.12) nie rozpoznawało metody kompresji plików zip, co spowodowało, że nie otrzymano ani jednego rezultatu z programu. To powoduje, że narzędzie zostanie wykluczone z dalszej analizy.

Możliwym rozwiązaniem byłoby rozpakowanie wszystkich plików innym narzędziem, np. 7z. Następnie wymaga to przeszukanie rozpakowanego pliku oraz rozpakowanie zawartego w nim kolejnego archiwum. Po wykonaniu takiego działania można odczytać zawartość, co znacznie wydłużyłoby czas działania. Takie podejście również komplikuje testowanie takiego rozwiązania, bo wykonywane są różne programy, które wzajemnie na siebie oczekują.

Narzędzie powinno być w stanie samo rozpakować i wyszukać wszystkie frazy poszukiwane. Dodatkowo narzędzie powinno znajdować frazy zaraz po rozpakowaniu pliku i nie powinno czekać na rozpakowanie wszystkich archiwów, aby przeszukiwać ich zawartość.

Kolejne z wymienionych narzędzi również nie spełnia wymagań. Narzędzie **zgrep** nie pozwala na rekurencyjne wyszukiwanie danych w folderach, w których znajdują się archiwa. Nawet zastosowanie pomocniczego narzędzia **find** w celu wykonania zadania, powoduje, że program nie jest w stanie przeskanować archiwów (rys. 4.13).

Typ MIME	**Rozmiar (w KB)**	**Ilość plików**
application/zip	9312014,35	236
audio/mpeg	2771693,64	541
application/x-rar	1358304,96	31
application/vnd,ms-htmlhelp	337207,65	100
application/octet-stream	302676,00	71
image/jpeg	228285,10	1347
video/x-ms-asf	223755,41	15
image/vnd,djvu	174702,98	24
application/postscript	84778,60	442
application/x-ace-compressed	83686,03	3
application/gzip	72011,68	58
image/gif	38853,47	1622
text/html	38485,94	1788
application/msword	29155,00	75
audio/ogg	24242,73	25
application/x-tar	20940,00	18
application/x-dosexec	9318,01	2
audio/x-wav	9180,17	1
application/x-bzip2	6383,85	5
text/rtf	4710,93	2
application/mac-binhex40	4704,44	2
text/x-c++	4021,66	347
video/x-msvideo	3731,00	1
application/vnd,ms-powerpoint	3420,50	1
text/x-c	987,73	345
text/plain	858,96	136
text/x-tex	748,25	64
application/winhelp	216,03	1
application/pdf	112,13	1
image/x-portable-greymap	54,24	5
text/x-diff	45,52	2
application/x-matlab-data	40,19	1
message/rfc822	31,62	1
text/xml	23,51	1
application/x-shockwave-flash	21,97	1
application/x-ole-storage	20,00	1
application/x-winhelp	16,43	1
text/x-makefile	10,37	7
image/x-xfig	7,15	3
application/javascript	6,91	1
application/x-wine-extension-ini	2,53	4
text/x-perl	1,92	1
inode/x-empty	0,00	20
Total	15149469,59	7353

Tabela 4.1: Ilość danych na podstawie typu MIME

```
[gadzbi@desktop-linux pracaMagisterska]$ find "$DIR" -type f -name "*.zip" -exec zgrep "main" {} +
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/angielski.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/angielski.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/DataMining.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/DataMining.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/Informatyka.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/Informatyka.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/konferencja.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/konferencja.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/ksiazki.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/ksiazki.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/literatura.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/literatura.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/literatura_old.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/literatura_old.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/matematyka.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/matematyka.zip: invalid compressed data--length error
```

Rysunek 4.13: Niepoprawne działanie programu zgrep z pomocą finda

Otrzymany błąd sugeruje, że długość danych w archiwum nie jest zgodna. Dodatkowo wyświetlono informacje o błędzie w wartości cyklicznej kontroli nadmiarowej CRC (ang. *Cyclic Redundancy Check*). Ta wartość to system sum kontrolnych pozwalający na wykrycie błędów zmagazynowanych danych. Narzędzie **zgrep** nie pozwala rozpakować i przeszukać zawartości szukanych archiwów.

Archiwa jednak są możliwe do otworzenia przez program graficzny Engrampa [4]. Choć wszystkie pliki zostały odczytane, oznacza to, że istnieje możliwość pozyskania części zawartości (rys. 4.14).

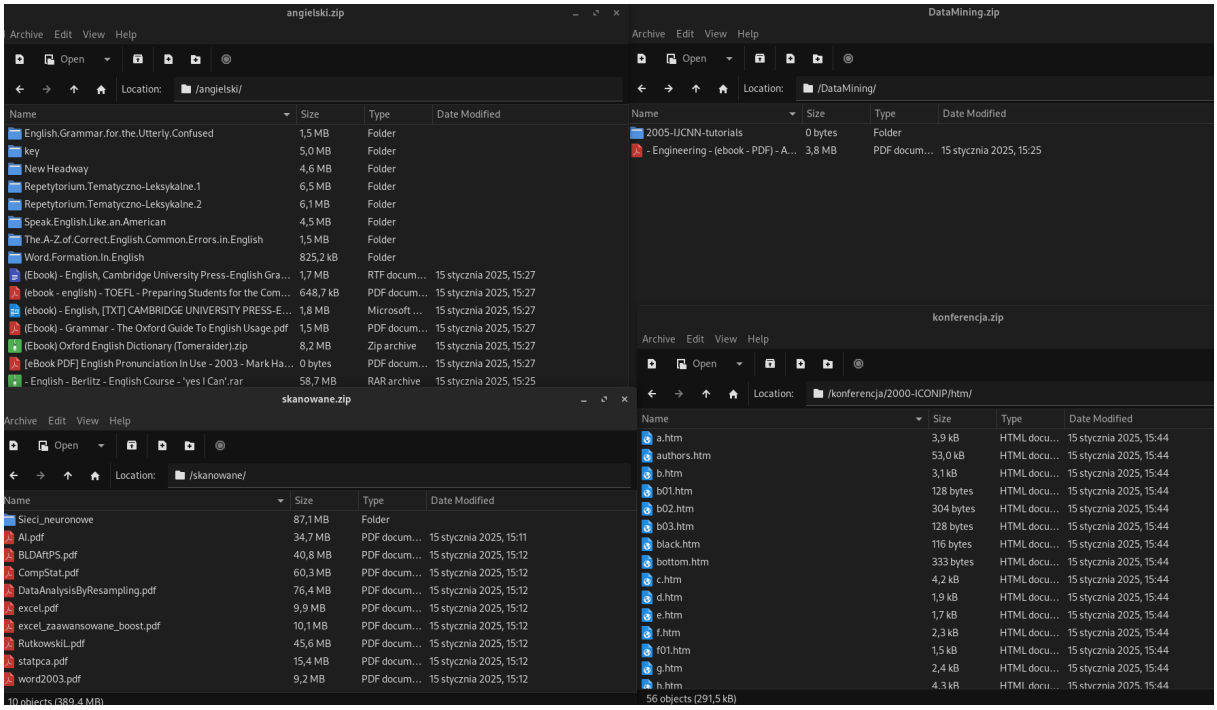
Narzędzie graficzne nie będzie brane pod uwagę do badania, zostało jedynie podane jako przykład poprawności archiwów.

Narzędzie **ripgrep** pozwala na wyszukanie zawartości w archiwach i działa bardzo dobrze. Narzędzie było w stanie wyszukać ogromną liczbę fraz 'main' we wszystkich plikach jak na rysunku (rys. 4.15). Narzędzie niestety nie posiada dokładnej lokalizacji, w której wystąpiło wyszukanie, tylko jest w stanie wskazać miejsce w pliku skompresowanym. Taki rezultat nie mówi, w którym pliku znajdują się frazy, a jedynie może znaleźć kontekst w pliku lub podać ilość wystąpień.

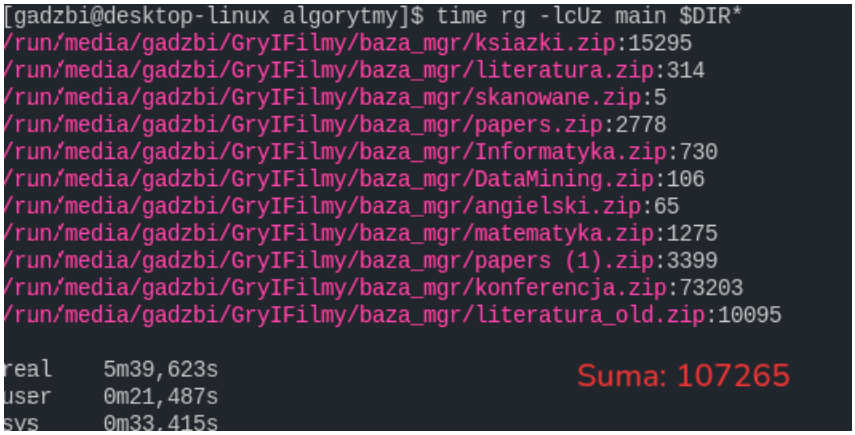
Różnica ta jest znacząca w celu znalezienia frazy w konkretnym pliku. Implementacja **gsearch** pozwala na wyszukanie linii pliku, w którym fraza się znajduje.

Autorskim programem do przeszukiwania treści w archiwach jest **gsearch**. Od teraz wszystkie odniesienia do implementacji będą odnosiły się do nazwy programu.

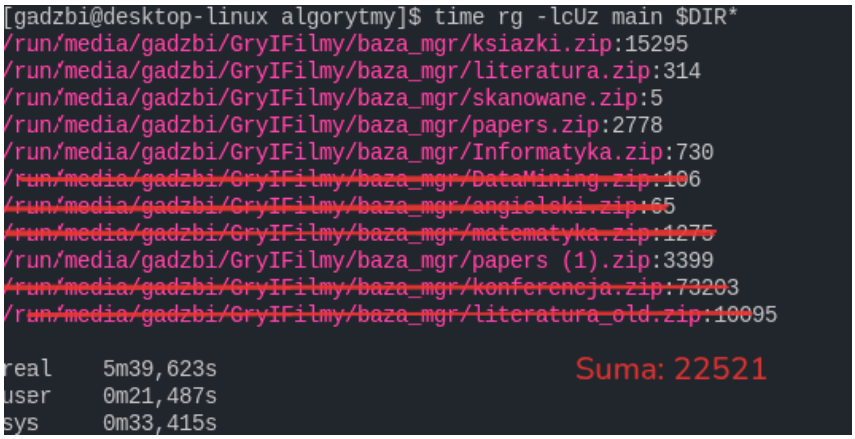
Gdy **gsearch** próbuje przeszukać archiwum z niepoprawną sumą kontrolną, program



Rysunek 4.14: Przykład otwarcia archiwów przez program graficzny Engrampa



Rysunek 4.15: Przykładowy rezultat wykonania komendy ripgrep ze zmierzonym czasem



Rysunek 4.16: Liczba wystąpień 'main' z pominięciem kilku archiwów

zostaje wyłączony z powodu błędu. Program **gsearch** może wykonać pełne szukanie w przypadku, gdy suma kontrola plików archiwów jest poprawna.

Gdy **gsearch** przeszukuje archiwa nieuszkodzone, przedstawione na rysunku (rys. 4.16), to widać, że liczba wystąpień w **gsearch** jest większa. Poniżej znajduje się zestawienie ilości znalezionych wystąpień dla wszystkich archiwów przeszukiwanych przez **rg**, pominiętych archiwów dla **rg** oraz pominiętych archiwów przeszukanych przez **gsearch**.

Fraza (litery)	**rg (25 GB archiw)**	**rg (15 GB)**	**gsearch (15 GB)**
wan (3)	34821	22951	20293
main (4)	107265	22521	23822
window (6)	16998	8251	9430
analysis (8)	3511	2168	1740
book desc (9)	6	3	3
informatyka (11)	1	1	5
wInDoW (6)	56760	20939	0

Tabela 4.2: Tabela zestawienia wyników wyszukiwania konkretnych fraz dla programów

Należy wspomnieć, iż jedno wystąpienie dla narzędzia **ripgrep** to pojawienie się frazy w danej linijce. W ten sam sposób liczone jest znalezienie wystąpienia w **gsearch**. Implementacja **gsearch** nie posiada wrażliwości na wielkość liter, dlatego nie jest testowana.

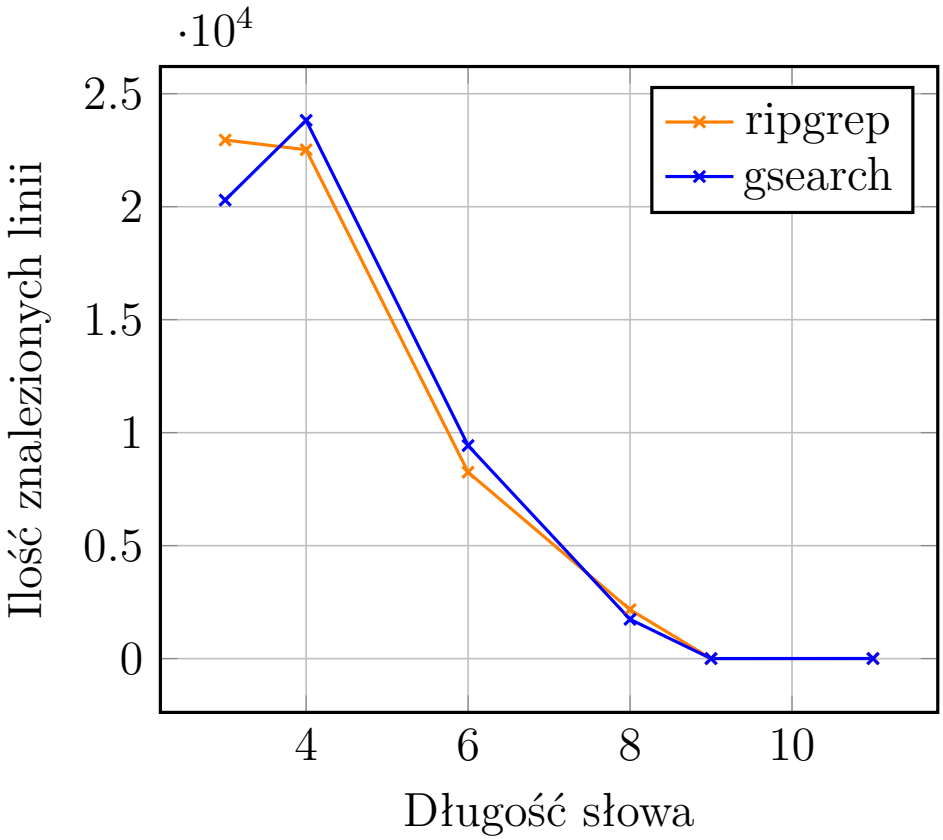
Tabela (tab. 4.2) przedstawia wyniki ilości wystąpień fraz w zbiorze archiwów. Ponieważ implementacja **gsearch** nie potrafi przeszukać określonych archiwów, zdecydowano się na zestawienie rezultatów **ripgrepa** ze wszystkich archiwów, oraz z tego samego zbioru archiwów pominiętych.

Należy zauważyć, że pozostałe programy (**ugrep**, **zgrep**) nie zostały zamieszczone w tabeli, z powodu braku uzyskania wyników. Każdy z tych programów miałby wartość 0 dla każdej frazy.

Ostatni wiersz pokazuje jakie znaczenie ma wyszukiwanie po wielkości liter (tab. 4.2). Program **ripgrep** posiada możliwość ignorowania wielkości liter czego nie robi program gsearch. Jeżeli nieznana jest dokładna wielkości liter frazy, to może to spowodować brak znalezienia frazy (rys. 4.17).

Na obrazie (rys. 4.18) można zauważyć przykładowy rezultat programu **gsearch**. Zapisany zostaje strumień wyjścia jako zawartość do pliku przy użyciu programu **tee**. Następnie obliczana jest ilość znalezionych linii, na podstawie zawartości pliku używając **wc -l**.

Program **ripgrep** nie jest w stanie znaleźć zawartości archiwów zagnieżdżonych w innych archiwach. Rozwiązuje ten problem gsearch i pozwala na wyszukanie znalezienie konkretnego pliku w którym fraza się znajduje.



Rysunek 4.17: Wykres wystąpień linii z frazami w zależności od długości liter w słowie

```
[gadzbi@desktop-linux algorytmy]$ _build/gsearch informatyka $DIR | tee temp.txt
/tmp/baza_mgr_unarr/michal.unzipped/michal/bookware/BOOKWARE.unzipped/2003-04/Uklady.Mikroprocesorowe.Przyklady.Rozwiazan
.SPECIAL.Tech-PROPER-eBook-Uuk/uuk.nfo:l64
/tmp/baza_mgr_unarr/michal.unzipped/michal/bookware/BOOKWARE.unzipped/2003-04/Uklady.Mikroprocesorowe.Przyklady.Rozwiazan
.SPECIAL.Tech-PROPER-eBook-Uuk/uukum.unzipped/uuk.nfo:l64
/tmp/baza_mgr_unarr/polskie.unzipped/polskie/_inne/Bill_Gates_i_jego_imperium_Microsoft.unzipped/James Wallance - Bill Ga
tes i jego imperium Microsoft.rtf:l1080
/tmp/baza_mgr_unarr/polskie.unzipped/polskie/_inne/Zagadnienia maturalne z informatyki.unzipped/Zagadnienia maturalne z i
nformatyki/doc/R-00.doc:l81
/tmp/baza_mgr_unarr/polskie.unzipped/polskie/hardware/Mikroprocesory.unzipped/Mikroprocesory/doc/r0-1.doc:l150
[gadzbi@desktop-linux algorytmy]$ cat temp.txt | wc -l
5
```

Rysunek 4.18: Zdjęcie przedstawia przykładowe wykonanie programu gsearch na słowie informatyka.

4.5 Porównanie prędkości wyszukiwania programów

Porównane zostaną programy, które pozwoliły uzyskać oczekiwany rezultat i są to **ripgrep** oraz implementowany **gsearch**. Porównanie szybkości wyszukiwania zostanie wykonane na pomniejszonym zbiorze, aby porównanie czasów było sprawdzane na tym samym zbiorze, gdzie oba programy są w stanie odczytać z nich dane.

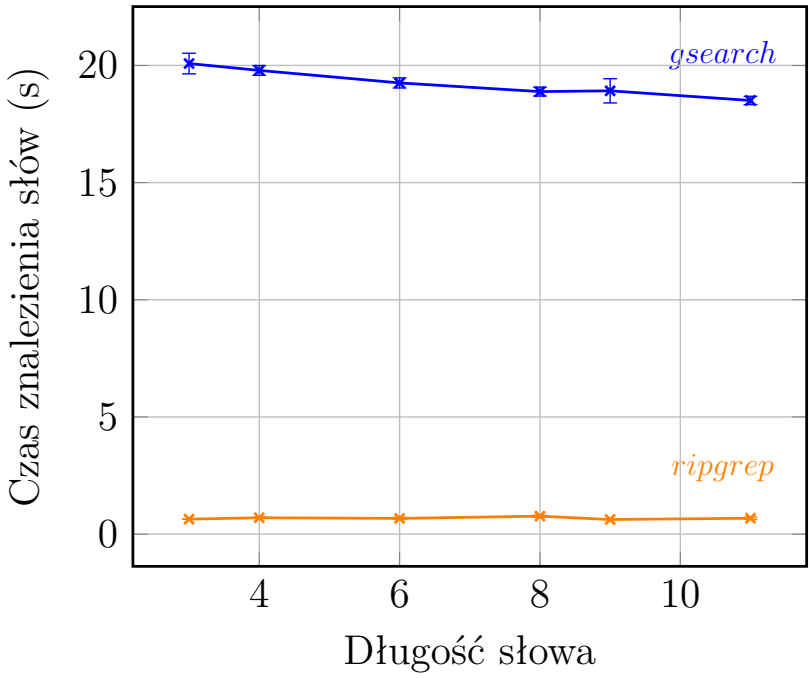
```
[gadzbi@desktop-linux algorytmy]$ hyperfine --runs 1 "_build/gsearch informatyka $DIR" "_build/rg -lcUz informatyka $DIR/
*" --export-csv binary_tests/informatyka.csv
Benchmark 1: _build/gsearch informatyka /run/media/gadzbi/GryIFilmy/baza_mgr_reduced
  Time (abs ≡):        18.751 s               [User: 12.880 s, System: 6.322 s]

Benchmark 2: _build/rg -lcUz informatyka /run/media/gadzbi/GryIFilmy/baza_mgr_reduced/*
  Time (abs ≡):        31.531 s               [User: 2.026 s, System: 6.020 s]

Summary
  _build/gsearch informatyka /run/media/gadzbi/GryIFilmy/baza_mgr_reduced ran
    1.68 times faster than _build/rg -lcUz informatyka /run/media/gadzbi/GryIFilmy/baza_mgr_reduced/*
```

Rysunek 4.19: Zdjęcie wyników testów hyperfine.

Porównanie zostanie przeprowadzone wykorzystując narzędzie **hyperfine**, które jest bardzo dobre do testowania różnic pomiędzy szybkościami podobnych programów. Przykładowy rezultat wykonania jednego testu daje dużo informacji i zostaje zapisany do pliku csv (rys. 4.19). Dwa programy **gseach** oraz **ripgrep** wykonują się jednokrotnie. Następnie otrzymano czas wykonania obu programów i krótkie podsumowanie otrzymanych wyników.



Rysunek 4.20: Wykres czasów wyszukania fraz dwóch programów w zależności od ilości liter

Prędkości wyszukiwania danych za pomocą funkcji **ripgrep** są znacznie większe niż te za pomocą **gsearch** (rys. 4.20). Wynika to z tego, że **ripgrep** wykonuje wyszukiwania z wykorzystaniem kilku wątków. Dodatkowo program **rg** ładuje pliki prosto do pamięci,

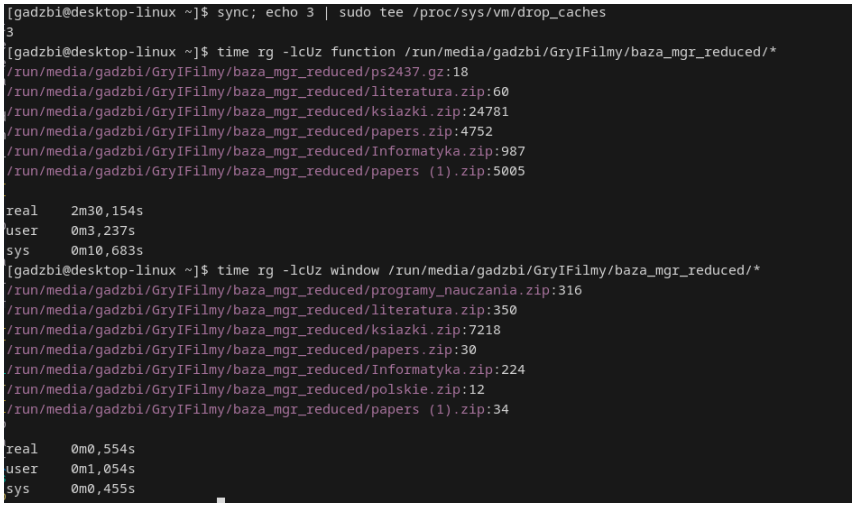
natomiast program **gsearch** rozpakowuje zawartość pliku archiwum do folderu, a następnie czyta zawartość każdego pliku.

Takie podejście było konieczne, gdyż Golang ma wbudowane większe ograniczenia w proces alokowania danych i tego ile danych może przechowywać w pamięci w danym momencie.

Wykorzystanie innego języka dającego większą swobodę w manipulowaniu pamięcią dałoby lepsze czasy wykonania, jednak wykorzystanie języka niskopoziomowego, wiąże się z większym czasem implementacji algorytmów oraz trudniejszym procesem analizowania błędów.

Program **ripgrep** dla fraz o długości 3 lub mniejszych używa algorytmu Teddy. Ten algorytm pozwala na załadowanie całej frazy do rejestru XMM i wykonania bitowego porównania małej frazy z zawartością danych przeszukiwanych.

4.6 Narzędzie ripgrep, a pamięć tymczasowa

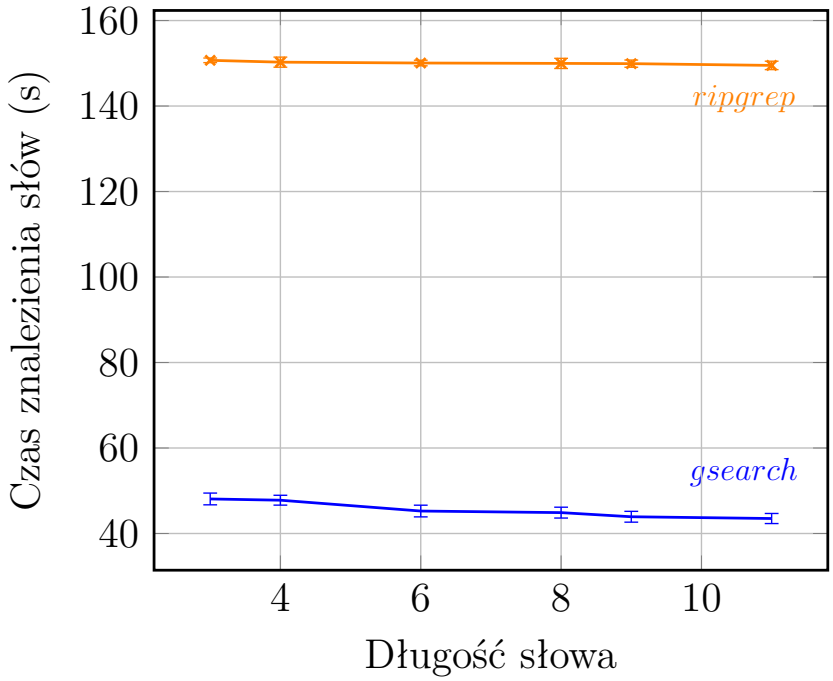


Rysunek 4.21: Przykład czasu wykonania narzędzia ripgrep z wyczyszczoną pamięcią podręczną i bez wyczyszczenia

Komenda **ripgrep** posiada bardzo nietypowe działanie w przypadku kolejnych wykonań programu. Po pierwszym wykonaniu programu dane z dysku są znacznie dłużej wyszukiwane. Pomimo że program nie wykorzystuje operacji zachowania danych w pamięci tymczasowej (cache), to system operacyjny przechowuje zawartość plików i ścieżki w RAM. To powoduje, że zanim **ripgrep** zacznie czytać treść z dysku, to sprawdza zawartość pamięci RAM. Gdy wyczyszczono cache jak na obrazie (rys. 4.21), zauważyć można znaczną pogorszenie rezultatów **ripgrep**.

Pierwsza komenda zapewnia brak utraty danych, a następnie druga komenda powoduje zapisanie pliku drop caches z wartością 3. Ta komenda czyści wszystkie zapisane zawartości pliku oraz dane o odczytywanych folderach i plikach.

Ta komenda powoduje spowolnienie działania obu programów, jednak w znacznie mniejszym stopniu w przypadku **gsearch** niż **ripgrep** jak na wykresie (rys. 4.22). Można również zauważyć większe odchylenie standardowe czasów wyszukiwań dla **gsearch**, natomiast ogólny czas działania programu jest o połowę krótszy niż w przypadku polecenia **ripgrep**.



Rysunek 4.22: Wykres czasów znalezienia fraz po wyczyszczeniu pamięci tymczasowej

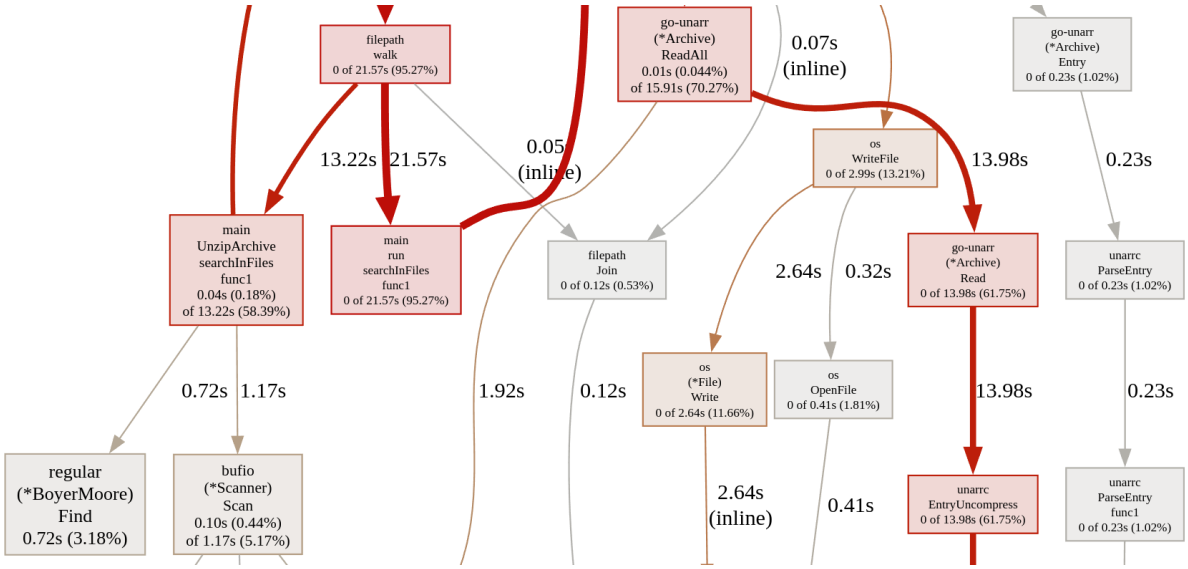
4.7 Wykorzystanie profilowania do oczytania charakterystyki programu

```
1 f, _ := os.Create("profile.pprof")
2 pprof.StartCPUProfile(f)
3 defer pprof.StopCPUProfile()
```

Rysunek 4.23: Dodanie profilowania do programu gsearch

Można również sprawdzić charakterystykę programu gsearch dodając kod na początek wykonania programu (rys. 4.23). Wykonanie programu na słowie 'informatyka' utworzy plik profile.pprof. Ten plik można przejrzeć w przeglądarce wcześniej wykorzystując komendę **go tool pprof -http=localhost:8090 profile.pprof**.

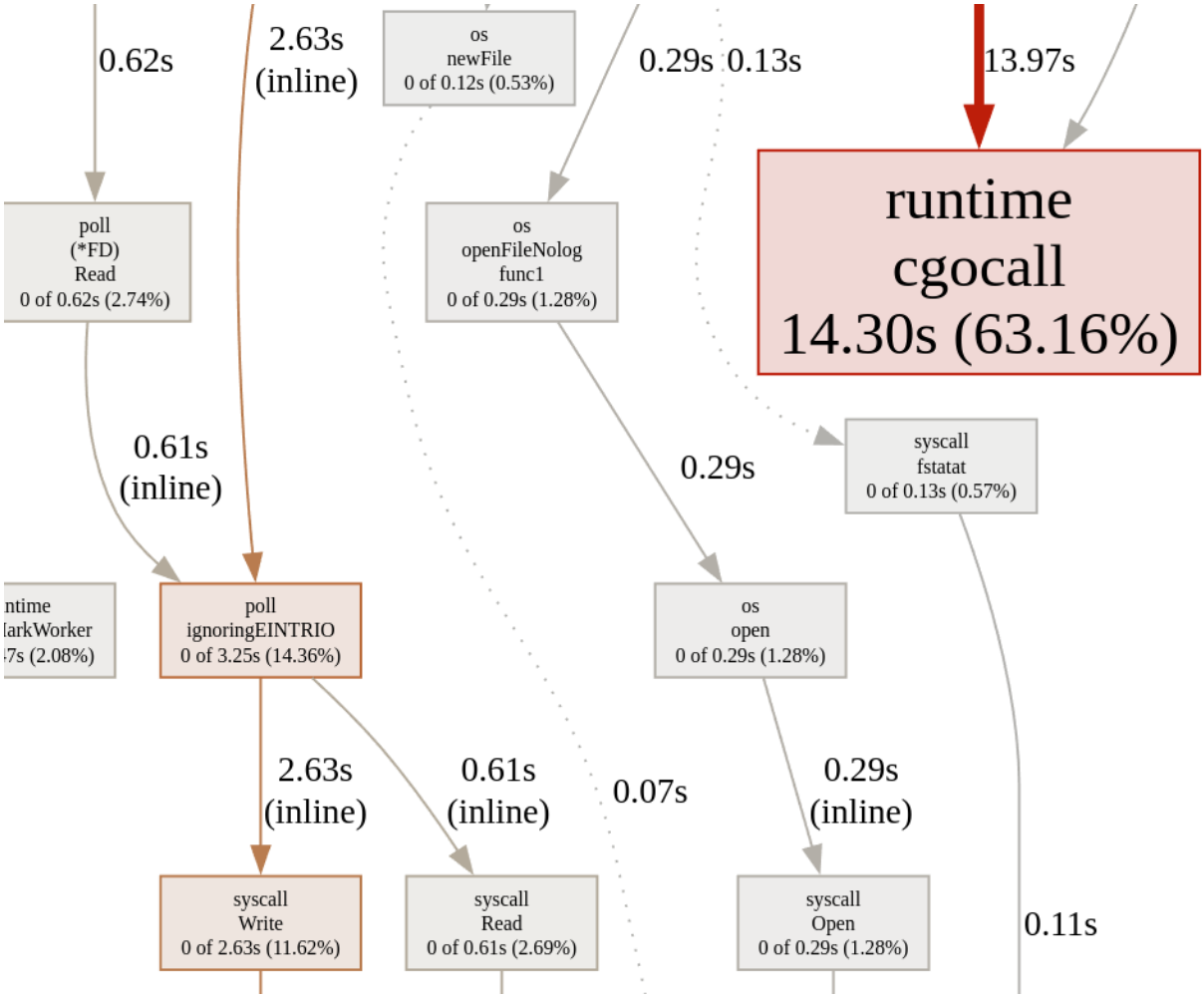
Po wykonaniu komendy w przeglądarce otworzy się widok na graf programu (rys. 4.24). Graf jest bardzo skomplikowany do analizy w przypadku, gdy program wykorzystuje rekursje do przeszukiwania folderów i rozpakowywania zawartości.



Rysunek 4.24: Zdjęcie profilu programu gsearch z użyciem narzedzia 'pprof'

Można zauważyć, że większość czasu programu wykorzystywana jest na operacje ekstrakcji archiwów i czytania archiwów. Sam algorytm przeszukujący (dolny lewy róg rys. 4.24) wykonuje się jedynie 0,72 sekundy.

Głównym ograniczeniem prędkości działania jest czas odczytywania zawartości z plików. Funkcje 'runtime cgocall' stanowi znaczną część czasu wykonywania programu (rys. 4.25), ponieważ wykonuje operacje odczytu zawartości treści z dysku.



Rysunek 4.25: Zdjęcie pokazujące procent czasu wykonania 'runtime cgocall' w gsearch

Rozdział 5

Podsumowanie

5.1 Opis wykonanych prac

W pierwszej części przeprowadzono badania na algorytmach, porównując szybkość ich wykonania na zbiorze plików. Dowiedziono wtedy, że na czasy wykonania znacznie wpływa czas alokowania nowych zasobów przez program. Udało się dowieść, że po wykorzystaniu tego samego bufora (**BWP**) do kolejnych przeszukań, przyspieszono algorytm BM o około 20%. Wynika to z różnicy pomiędzy wykresami 4.3 oraz 4.4.

Zaletą takie rozwiązania jest to, że bufor pozwala na zmniejszenie czasów na alokowanie pamięci pomiędzy wykonaniami. **BWP** zasadniczo nie posiada wad w tej implementacji.

Ponad udało się pokazać, że stworzenie jednego bufora na przechowywanie treści pliku redukuje dodatkowo czas pomiędzy wykonaniami o około 8%, co stanowi zaletę tego rozwiązania. Można to zaobserwować porównując różnice czasów wykonań w 4.4 oraz 4.5. Wadą tego rozwiązania jest to, że rozmiar buforu na plik musi być zanany przed wykonaniem programu. Ostatecznie dowiedziono, że algorytm Boyera-Moore'a będzie najlepszym wyborem dla danego zbioru danych (tab. 3.1).

Następnie przeprowadzono badania na zbiorze archiwów, wykorzystując najszybszy algorytm Boyera-Moore'a. Porównano działanie programów **ugrep**, **zgrep**, **ripgrep** oraz autorskiego **gsearch**. Z badania wynikło, że jedynie narzędzie **ripgrep** oraz **gsearch** mogą być mierzone pod względem ilości wyszukiwań i prędkości. Narzędzia **ugrep** oraz **zgrep** nie znalazły żadnych wyników.

Zaletą programu **ripgrep** okazała się szybkość wykonania w przypadku, gdy pliki znajdują się w pamięci tymczasowej, co wynika z rysunku 4.20. Zaletą programu **gsearch** jest jego szybkość wykonania, kiedy pliki zbioru są czytane po raz pierwszy. Wynika to z rysunku 4.22.

Wadą narzędzia **ripgrep** okazało się wolniejsze wykonywanie programu po raz pierwszy. Wadą **gsearch** było wolniejsze wykonywanie kolejnego identycznego wyszukiwania.

W ostatniej części badań odkryto, że czas wykonania algorytmu zajmuje jedynie $0,3\%$

całego wykonania programu **gsearch** (rys. 4.24). Z obrazka wynika, że funkcja main wykonywała się 21,57 s, a sam algorytm Boyera-Moore'a jedynie 0,72 s. Zdecydowaną większość czasu oczekiwano na system operacyjny, aby dostarczył zawartości plików w funkcji 'cgocall'. (rys. 4.24). Wadą wykonania ekstrakcji archiwów w funkcji 'cgocall' jest dłuższy czas wykonania. Zaletą jest fakt, że otrzymuje się dokładne miejsce wystąpienia frazy w zbiorze danych.

5.2 Wnioski

Przeprowadzone badania koncentrowały się na analizie wydajności algorytmów wyszukujących w kontekście przeszukiwania zawartości tekstowej w różnych formatach plików, ze szczególnym uwzględnieniem archiwów. Metodyka badawcza obejmowała porównanie prędkości wykonania algorytmów na mniejszym zbiorze rozpakowanych archiwów oraz analizę porównawczą narzędzi pod względem liczby i prędkości wyszukiwań na nierozpakowanym zbiorze. Testy przeprowadzono na środowisku Linux.

Zbiór danych testowych składał się z różnorodnych formatów plików o łącznym rozmiarze około 15 GB. W trakcie badań uwzględniono problematykę różnych formatów plików, co pozwoliło na kompleksową ocenę wydajności algorytmów w rzeczywistych warunkach. W ramach analizy algorytmów porównano implementacje Morrisa-Pratta, Knutha-Morrisa-Pratta i Boyera-Moore'a. Szczególną uwagę poświęcono wpływowi optymalizacji bufora wstępnego przetwarzania na wydajność oraz znaczeniu ponownego wykorzystania buforów w kolejnych wyszukiwaniach. Badania wykazały istotny wpływ tych optymalizacji na ogólną wydajność wyszukiwania. Algorytm Boyera-Moore'a otrzymywał rezultaty najszybciej, dlatego że zachowywał więcej informacji o łańcuchu szukanym. To pokrywa się z hipotezą badań.

Istotnym elementem badań było porównanie autorskiego narzędzia **gsearch** z popularnymi rozwiązaniami takimi jak **ugrep**, **zgrep** i **ripgrep**. Analiza objęła różnice działania programów, w kontekście różnych archiwów oraz wpływ pamięci podręcznej na wydajność wyszukiwania.

Wykazano, że operacje I/O, w tym odczyt z dysku i ekstrakcja archiwów, stanowią największe obciążenie, podczas gdy sam algorytm wyszukiwania zajmuje relatywnie małą część całkowitego czasu wykonania. Autorskie rozwiązanie **gsearch**, choć ustępuje wydajnością **ripgrep** w przypadku kolejnych wyszukiwań, oferuje unikalną funkcjonalność znalezienia wyników w złożonej strukturze zagnieżdżonych archiwów. Funkcja **ripgrep** pozwala znaleźć wystąpienia w archiwum bez dokładnej lokalizacji ścieżki w archiwum.

5.3 Możliwości rozwoju

Program można usprawnić w celu wyszukiwania większej ilości zawartości przy pomocy dodatkowej implementacji dla transkrypcji. Pozwoliłaby ona na wyszukanie treści w plikach audio, ponieważ istnieją darmowe narzędzia pozwalające na taką konwersję.

Kolejnym elementem, który można rozważyć w celu kontynuacji pracy, byłoby wykorzystanie OCR (ang. *Optical Character Recognition*). Zaimplementowanie takiego rozwiązania pozwoli pozyskać treść ze zdjęć oraz plików pdf, które składają się ze zdjęć i tekstowych skanów treści.

W celu uzyskania lepszych rezultatów można, zamiast wykorzystania gotowej biblioteki, stworzyć autorską bibliotekę dekompresującą. Obejmowałaby ona brakujące i nie poprawnie działania niektórych formatów archiwów. To zadanie wymaga ustalenia granicy, jak bardzo archiwum może być uszkodzone, żeby można było odczytać z niego dane.

Kolejnym usprawnieniem dla programu byłoby wprowadzenie pamięci podręcznej (ang. *caching*). Takie rozwiązanie pozwoliłoby na zapamiętanie plików, które już kiedyś dekompresowano. Należy jednak przechować informacje o tym, czy suma kontrolna (ang. *hash*) archiwum nie zmieniła się, pomiędzy wykonaniami programu.

Dodatkowo można pozwolić programowi działać na większej ilości wątków. To wymaga dodatkowej synchronizacji pomiędzy odczytem archiwów, natomiast Golang jest bardzo łatwym językiem do wprowadzania takich zmian.

Kolejnym kierunkiem do rozwoju jest wykorzystanie wyrażeń regularnych podczas wyszukiwania zawartości. Taka implementacja na pewno spowolni algorytm wyszukujący, natomiast pozwoli na uzyskanie większej ilości rezultatów.

Bibliografia

[1] Bevin Brett. *Memory Performance in a Nutshell*. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html (term. wiz. 29. 01. 2025).

[2] BurntSushi. *ripgrep recursively searches directories for a regex pattern while respecting your gitignore*. 2024. URL: https://github.com/BurntSushi/ripgrep (term. wiz. 22. 09. 2024).

[3] Creative Commons. *Intel 8008*. 1972. URL: http://8008.classiccmp.org//8008UM.pdf (term. wiz. 07. 02. 2025).

[4] mate desktop. *Engrampa*. 2025. URL: https://github.com/mate-desktop/engrampa (term. wiz. 25. 01. 2025).

[5] dneil@google.com. *Algorytm wyszukiwania w standardowej bibliotece Golang*. 2024. URL: https://cs.opensource.google/go/go/+/refs/tags/go1.23.5:src/internal/stringslite/strings.go (term. wiz. 05. 02. 2025).

[6] Google. *Google Drive Storage*. 2024. URL: https://www.google.com/intx/en-GB/drive/ (term. wiz. 14. 01. 2025).

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

[8] Dariusz Wawrzyniak Jerzy Brzeziński. *Urządzenia Wejscia-Wyjscia*. 2017. URL: https://www.cs.put.poznan.pl/dwawrzyniak/SysOp2017/io_notatki.pdf (term. wiz. 30. 01. 2025).

[9] A. Kokaram L.M. Manus. *An improved error resilience scheme for highly compressed data*. 2020. URL: https://ieeexplore.ieee.org/document/993497 (term. wiz. 29. 01. 2025).

[10] Phil Winterbottom Lorenz Huelsbergen. *Very Concurrent Mark-And-Sweep Garbage Collection without Fine-Grain Synchronization*. 1998. URL: https://doc.cat-v.org/inferno/concurrent_gc/concurrent_gc.pdf (term. wiz. 03. 02. 2025).

[11] Adam Piórkowski Magdalena Ładniak. *Przegląd otwartych rozwiązań systemów archiwizacji i komunikacji obrazów medycznych*. 2006. URL: https://home.agh.edu.pl/~pioro/dyd/BDwBiM/open_pacs.pdf (term. wiz. 29. 01. 2025).

[12] Artur Malinowski. *Zastosowanie bajtowo adresowanej pamieci nvram do zwiekszenia wydajnosci wybranych aplikacji*. 2020. URL: https://mostwiedzy.pl/pl/publication/download/0/zastosowanie-bajtowo-adresowanej-pamieci-nvram-do-zwiekszenia-wydajnosci-wybranych-aplikacji-rownole_20200403095603973267.pdf (term. wiz. 29. 01. 2025).

[13] Dariusz Paradowski. *Digitalizacja piśmiennictwa*. 2010. URL: https://www.gov.pl/attachment/8f6586b0-3a9e-4af2-9cc5-6eb994bfdbd7 (term. wiz. 29. 01. 2025).

[14] Paweł Perzyna. *Komputeryzacja i Digitalizacja w Archiwach*. Warszawa: PASAŻ, 2016. ISBN: 978-83-7629-956-3.

[15] W. Curtis Preston. *Archiwizacja i Odzyskiwanie danych*. Helion: O'Reilly, 2008. ISBN: 978-83-246-1182-9.

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

[17] 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).

[18] Sam Thorogood. *Twoja pamięć podręczna*. 2020. URL: https://web.dev/articles/love-your-cache?hl=pl (term. wiz. 29. 01. 2025).

[19] Vincent. *Go: Memory Management and Allocation*. 2019. URL: https://medium.com/a-journey-with-go/go-memory-management-and-allocation-a7396d430f44 (term. wiz. 03. 02. 2025).

Dodatki

Spis skrótów i symboli

BM - Algorytm Boyera-Moore'a.

BWP - Bufor Wcześniejszego Procesowania — Przed rozpoczęciem wyszukiwania, algorytm sprawdza frazę szukaną i zapisuje do tego bufora optymalizację skoków (w zależności od algorytmu).

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

GNU - Uniksopodobny system operacyjny z wolnego oprogramowania. Jego częścią są takie narzędzia jak gzip, bash czy wget.

KMP - Algorytm Kurta Moris'a Pratta.

MP - Algorytm Moris'a Pratta.

Spis rysunków

Spis tabel