



**Politechnika
Śląska**

PRACA MAGISTERSKA

Analiza narzędzi wyszukiujących zawartość tekstową w systemie Linux

inż. Kacper NITKIEWICZ

Nr albumu: 290409

Kierunek: Informatyka Przemysłowa

Specjalność: Cyberbezpieczeństwo

Prowadzący pracę

dr inż. Adrian Smagór

Katedra Inżynieria Przemysłowa

Wydział Inżynierii Materiałowej

Katowice 2025

Tytuł pracy

Analiza narzędzi wyszukiujących zawartość tekstową w systemie Linux

Streszczenie

Analiza algorytmów wyszukiujących 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 wyszukikań 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

1	Wstęp	1
1.1	Wprowadzenie do problemu	1
1.2	Cel Pracy	2
1.3	Zakres Pracy	2
2	Analiza tematu wyszukiwania tekstu	5
2.1	Sformułowanie problemu	5
2.2	Dostępne rozwiązania	5
2.2.1	Przykładowe narzędzia dostępne do wykorzystania	5
2.3	Odniesienia do literatury	7
2.4	Opis poznanych rozwiązań	8
2.4.1	Algorytm brute force	8
2.4.2	Algorytm Morisa-Pratta	8
2.4.3	Algorytm Kurta-Morisa-Pratta	11
2.4.4	Algorytm Boyera-Moore'a	12
2.4.5	TODO? Algorytm Karpa-Rabina	13
2.4.6	TODO? Algorytm Aho-Corasick	13
3	Przedmiot pracy - Wybór najszybszego rozwiązania wyszukiwania tekstu	15
3.1	Rozwiązanie zaproponowane przez dyplomanta	15
3.1.1	Rozwiązanie problemu wyszukiwania tekstu w pliku tekstowym	15
3.1.2	Rozwiązanie problemu zarchiwizowanych plików w zbiorze danych	17
3.2	Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi	18
3.2.1	Użycie języka Golang	18
4	Badania	21
4.1	Metodyka badań	21
4.1.1	Cel badania	21
4.1.2	Zakres badania	21
4.1.3	Hipoteza badań	21
4.2	Badanie benchmarku algorytmów	21

4.2.1	Zbiór badań	22
4.3	Badanie ilość otrzymanych wyników z programów	25
4.4	Porównanie prędkości wyszukiwania programów	27
4.4.1	Wykorzystanie profilowania do odczytania charakterystyki programu .	27
5	Podsumowanie	35
5.1	Możliwości rozwoju	35
	Bibliografia	37
	Dokumentacja techniczna	41
	Spis skrótów i symboli	43
	Lista dodatkowych plików, uzupełniających tekst pracy (jeżeli dotyczy)	45
	Spis rysunków	47
	Spis tabel	49

Rozdział 1

Wstęp

1.1 Wprowadzenie do problemu

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

Jednym ze sposobów zachowywania danych i zmniejszenia ich objętości jest archiwizacja plików. Takie rozwiązanie jest bardzo przydatne w przypadku chęci zmniejszenia ilości danych przechowywanych, a także w dystrybucji danych dla innych użytkowników. Otrzymana biblioteka danych była przekazana do analizy w celu wykonania tejże pracy.

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

Problem wyszukiwania danych nastąpił w momencie wyszukiwania dużej ilości zawartości. Oryginalne archiwum wynosi 25 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 wydajność programu.

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.

Dodatkowe odczytywanie archiwów w archiwach wymaga kilkukrotnej dekompresji danych. Aby otrzymać odpowiednią ilość wyników, należy wykonać dekompresję wszystkich zagnieźdzonych archiwów. Odczytanie zawartości odbędzie się poprzez przejście po drzewie plików.

1.2 Cel Pracy

Celem niniejszej pracy jest analiza algorytmów wyszukiujących 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 ilości pamięci. Z biegiem lat zaistniała potrzeba odkrycia algorytmów wyszukiujących zawartość pamięci. Procesor Intel 8008 zaprezentowany w 1972 posiadał jedynie 14-bitową magistralę adresową, co pozwalało na 16 KB pamięci [2]. Obecna ilość pamięci, którą otrzymujemy z chmury Google’a to 15 GB [4] co stanowi milion maksymalnych bloków pamięci w urządzeniu ze wspomnianym procesorem Intela.

Zasadniczym problem naszej pracy jest wyszukiwanie zawartości tekstowej ogromnej ilości 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** [1] pozwalają na wyszukiwanie tekstu. Narzędzia te nie są przystosowane do znalezienia pełnej ścieżki pliku dla archiwów.

2.2.1 Przykładowe narzędzia dostępne do wykorzystania

Narzędzie **find** to znane i popularne narzędzie w systemie z jądrem Linux. Już bardzo często wykorzystywany do znajdowania plików w systemie po nazwie, jednak nie nadaje się 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 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 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 ogólnodostępnym narzędziem GNU. Jego działanie jest dość podobne do find'a, gdyż posiada on możliwość wyszukiwania treści w plikach tekstowych. Grep również nie posiada też możliwości szukania zawartości archiwów oraz nie wspiera wielu formatów. Jest możliwość binarnego przeszukania plików, lecz archiwa nie będą odczytane w przypadku użycia kompresji.

```
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

Na rysunku powyżej możemy zobaczyć przykładowe komendy programu `grep` 2.2. Pierwsza komenda wyszukuje tekst podany jako pierwszy argument w pliku `plik1.txt`. Kolejna komenda pozwala nam przeszukać wszystkie foldery znajdujące się w folderze `/biblioteka` rekursywnie `-r`, gdzie zawartość plików posiada "szukany-tekst". Trzecia komenda wyszuka wszystkie instancje "szukany-tekst" ignorując przy tym wielkość liter. W tym przypadku można wyszukać tekst posiadający treść "szUKaNY-tEKst". Dodając argument `-w` wyświetla się wiersz, w którym znajdowało się dopasowanie.

Przedostatnia komenda w 2.2 pozwala na sprawdzenie kontekstu, w którym znajduje się znalezione ciąg znaków. Jeżeli "szukany-tekst" znajduje się w linii 25, to program przedstawi nam linie 23-27 z wyszczególnioną treścią, którą wyszukiwano. Ostatnia komenda przedstawia, że program można wykorzystać, przyjmując zawartość innego programu przy pomocy standardowego wejścia strumieniowego. Ta komenda zwróci nam wszystkie linie, w których "szukany-tekst" nie występuje.

Istnieje również `ripgrep`, który jest następcą wcześniej wymienionego narzędzia. Jego wydajność przewyższa `grep`a nawet trzydziestokrotnie w niektórych testach wydajnościowych w wymiarze czasu. Nie jest on niestety domyślnie instalowany na większości systemów linuxowych.

2.3 Odniesienia do literatury

Istnieje wiele odniesień do wyszukiwania danych w literaturze. Praca Google [7] odnosi się do problemu wyszukiwania tekstu w dobie internetu i ilości danych, która jest przechowywana w chmurze. Wymagane jest indeksowanie, które przyspiesza wyszukiwanie, ale wykonane nie poprawnie skutkuje słabymi wyszukiwaniami. Ilość danych wydobywana i dostarczana do użytkowników stanowi duże wyzwanie oraz wymaga wykorzystania skomplikowanych algorytmów.

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

Podczas konferencji TREC [5] jeden z zespołów napotkał problem, duplikacji danych. Chęć wydobywania danych w celu utworzenia tekstów wymagało usunięcia duplikacji dokumentów oraz wyborze tej treści, która posiada najwięcej wartości. Mogłoby to być tematem rozszerzenia pracy.

2.4 Opis poznanych rozwiązań

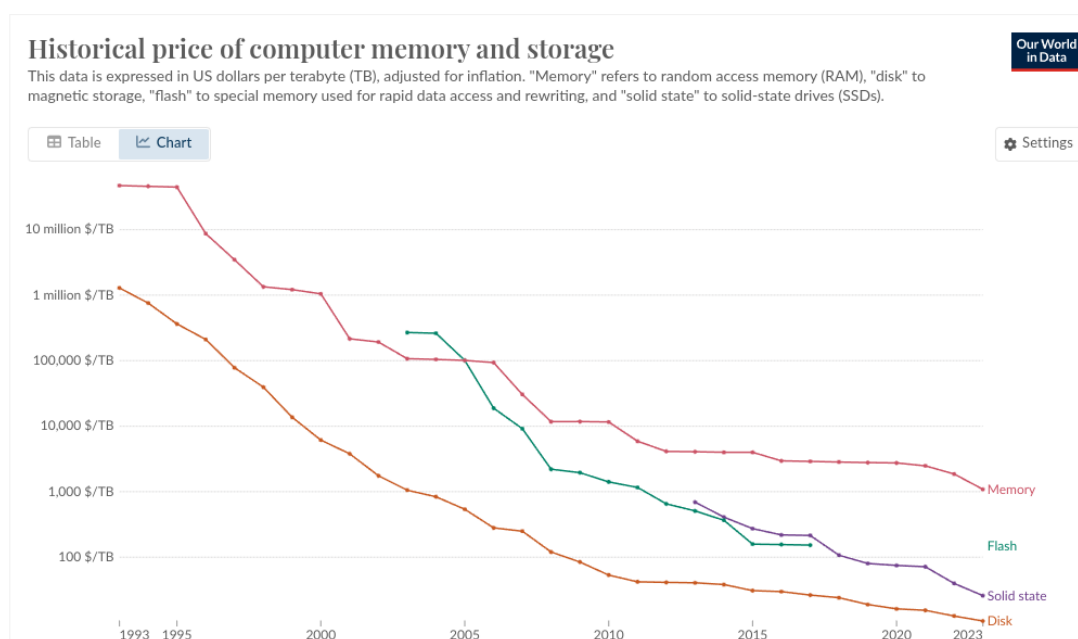
2.4.1 Algorytm brute force

```

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

Rysunek 2.3: Przykład algorytmu brute force

Jest wiele algorytmów, które wyszukują tekst. Jednym z takich algorytmów jest algorytm typu brute-force. Polega sprawdzaniu każdego bajtu, jego implementacja jest bardzo prosta i standardowa, a złożoność czasowa tego rozwiązania wynosi $O(\text{len}(\text{pattern}) * \text{len}(\text{substring}))$, gdzie $\text{len}(\text{pattern})$ to długość bloku przeszukiwanego, a $\text{len}(\text{substring})$ to długość tekstu szukanego. Ten algorytm nie pozwala na wykonanie optymalizacji wykorzystywanej w kolejnych algorytmach, takich jak przesunięcie się do następnej instancji znaku powtórzonego. Przykładową implementację algorytmu można znaleźć na rysunku 2.3.



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

2.4.2 Algorytm Morisa-Pratta

Algorytm Morisa-Pratta jest algorytmem, wykorzystującym możliwość procesowania łańcucha wyszukiwanego w tekście wykorzystując wcześniej pasującą do siebie część (rys.

```
1 preproc := make([], byte, len(substr)+1)
2 curr = -1
3 preproc[0] = -1
4 for i := 1; i <= len(substr); i++ {
5     for (curr > -1) && (substr[curr] != substr[i-1]) {
6         curr = preproc[curr]
7     }
8     curr++
9     preproc[i] = curr
10 }
```

Rysunek 2.5: Przykład uprzedniego procesowania tekstu szukanego

2.5). Polega on na wykorzystaniu faktu istnienia pasującego prefikso-sufiksu. Pozwala to na pominięcie porównania znaków, które się powtarzają w łańcuchu poszukiwanym.

Dzięki wykorzystaniu tej zależności możemy uniknąć cofania się indeksu i . Od teraz jako tablicę przechowującą informacje o przesunięciu w przypadku błędnego znaku, którą zainicjowaliśmy w 2.5 będziemy się odwoływać jako tablica **preproc**. Tablice **preproc** wypełniamy poprzednią wartością tak długo, aż zaistnieje różnica pomiędzy obecnym a następnym znakiem tablicy poszukiwanej (tablicy **substr**). W przypadku różnicy zwiększamy wartość zapisywaną do tablicy **preproc** o odległość różnicy znaków. W ten sposób następnym razem będzie możliwość pominięcia porównania tych znaków.

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

Rysunek 2.6: Przykład procesowania łańcucha poszukiwanego w algorytmie Morisa Pratta

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

(rys. 2.6). Dzięki temu zwykle dłuższy tekst znajdujący się we wzorcu `s` możemy przeanalizować szybciej niż w przypadku algorytmu brute-force. Powoduje to niestety problem w przypadku, gdy wyszukiwany wzorec nie jest wystarczająco długi, gdyż wykonanie operacji wcześniejszego procesowania posiada dodatkowy koszt, którego nie ma w algorytmie brute-force.

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

W podstawowej bibliotece języka Golang, w pakiecie *strings* istnieje implementacja metody *Index()*. Nie jest ona jednak w pełni przedstawiona w kodzie, natomiast w jej implementacji można zauważyć, że algorytm brute-force jest wykorzystywany tylko w przypadku, gdy długość wzorca wynosi więcej niż 64 (rys. ??). W Golang, gdy wzorec jest większy niż 64 znaki, to wykonuje się algorytm podobny do Morisa-Pratta, który jednak posiada dodatkową walidację w przypadku odkrycia wyniku fałszywie dodatniego. Algorytm Morisa-Pratta nie potrzebuje takiej walidacji.

wzorzec S	AAAAAABAAAAAABAAAAAAA
łańcuch W	AAAAAA
liczba cofnięć	20

Tabela 2.1: Przykład wykorzystania algorytmu KMP

2.4.3 Algorytm Kurta-Morisa-Pratta

```

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

```

Listing 2.1: Różnica pomiędzy algorytmami KMP i MP

Kolejny rozpatrywany algorytm jest implementacją rozszerzającą poprzednią implementację przedstawioną w algorytmie Morisa-Pratta. Różnice można zauważyć na podstawie rysunku (rys. 2.1) i występuje w nim następująca różnica. Gdy nie osiągnięto długości łańcucha i obecny znak jest równy temu, który znajduje się w łańcuchu szukanym, to można wykonać skok do znaku znajdującego się w tablicy przygotowanej, a nie sprawdzać kolejny 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 próba dopasowania tekstu będzie kończyć się porażką, 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 przeszukiwany T przedstawia najbardziej niekorzystny scenariusz (tab. 2.1), co można zaobserwować na przykładzie tekstu $S = "AAAAAABAAAAAABAAAAAAA"$.

W tym przypadku algorytm musi sprawdzić każde wystąpienie 'A' przed dotarciem do 'B', co jest bardzo nieefektywne. Sytuacja pogarsza się wraz ze wzrostem liczby powtórzeń fragmentu "AAAAAAB". Mimo że metoda tablicowa działa tu sprawnie (bez potrzeby cofania się), to jej jednokrotne wykonanie dla łańcucha szukanego W może być wolny, gdyż proces wyszukiwania często wymaga wielokrotnych przebiegów. Wielokrotne przeszukiwanie tekstu S w poszukiwaniu wzorca prowadzi do gorszej wydajności. W takich przypadkach, gdzie mamy do czynienia z tego typu charakterystyką tekstu i wzorca, algorytm Boyera-Moore'a może stanowić optymalne rozwiązanie.

Algorytm KMP wykorzystuje w najgorszym przypadku liniowy przebieg, natomiast algorytm Boyera-Moore'a w najlepszym przypadku posiada złożoność $O(len(worzec) + len(łańcuch))$, a w najgorszym przypadku $O(len(worzec) * len(łańcuch))$.

2.4.4 Algorytm Boyera-Moore'a

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

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, które na pewno nie zawierają wzorca. BM wykorzystuje dwie heurystyki: "złego znaku" oraz "dobrego sufiksu", podczas gdy KMP i MP opierają się na pojedynczej tablicy prefiksowej.

W kontekście implementacji algorytm Boyera-Moore'a wymaga utworzenia dwóch tablic pomocniczych dla swoich heurystyk, 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.

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

2.4.5 TODO? Algorytm Karpa-Rabina

2.4.6 TODO? Algorytm Aho-Corasick

Rozdział 3

Przedmiot pracy - Wybór najszybszego rozwiązania wyszukiwania tekstu

3.1 Rozwiązanie zaproponowane przez dyplomanta

3.1.1 Rozwiązanie problemu wyszukiwania tekstu w pliku tekstowym

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

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

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

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

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

Tabela 3.1: Dystrybucja w danych na podstawie typu plików MIME

Wykorzystanie kilku znanych algorytmów do przeszukiwania zawartości tekstu i sprawdzenie, który z nich najlepiej sprawdza się pod względem prędkości i dokładności wyszukiwania. Porównano ilość plików (tab. 3.1), które występują w zbiorze danych i 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 pdf:

- ".pdf",

3. pliki archiwów:

- ".tar.gz",
- ".rar",
- ".zip",
- ".tgz",
- ".tar",
- ".gz",

4. pliki pochodzące od Microsoftu:

- ".doc",
- ".docx".

Innym sposobem na rozwiązanie problemu jest wykorzystanie dostępnych narzędzi i dostosowanie ich do problemu, który rozwiązujemy. Wykorzystane narzędzia znacznie różnią się od siebie implementacją oraz funkcjonalnością. Aby porównać analizowane programy z implementacją gsearch, potrzeba znacznie zawęzić podobieństwo oczekiwanych rezultatów.

3.1.2 Rozwiązanie problemu zarchiwizowanych plików w zbiorze danych

Zbiór danych 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 ciągle 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.

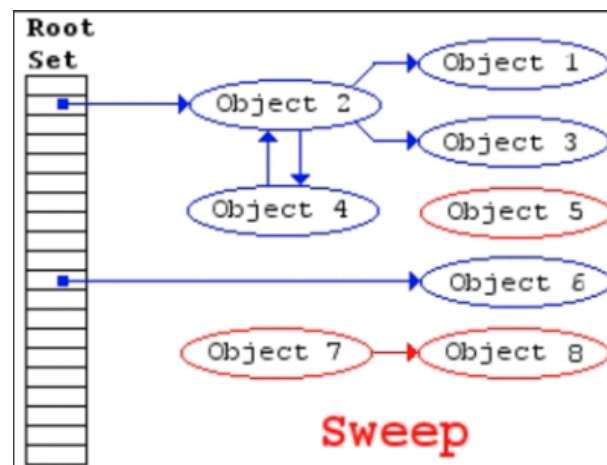
Pliki zawierające archiwum zostaną rozpakowane do osobnego folderu, a następnie wykonane zostanie rekurencyjne wyszukanie zawartość w każdym folderze i pliku. Ta metoda zostanie powtórzona tak długo, aż wszystkie pliki zostaną odczytane i nie będzie już więcej archiwów do rozpakowania.

3.2 Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi

3.2.1 Użycie języka Golang

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

Zaletą Golang jest to, że jest kompilowany i tworzy się natywny dla danego systemu plik binarny. Daje to możliwość łatwego przenoszenia programu wynikowego. Jest to też przewaga nad innymi językami programowania takimi jak python czy javascript, gdyż te języki są interpretowane i z natury wolniejsze niż kod binarny. Golang nie wymaga dodatkowego poziomu abstrakcji w postaci maszyny wirtualnej czytającej bytecode jak w przypadku Javy i JVM.



Rysunek 3.1: Przykładowe działanie Garbage Collectora w programie

Wadą tego języka może być fakt, iż język nie daje programiście możliwości pełnej kontroli pamięci. Język wykorzystuje ang. *Garbage Collector* (zbieracz śmieci) przedstawiony na rysunku 3.1, który jest alternatywą do sposobu manualnej alokacji pamięci. Można zauważyć, że 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 elementy, które przestały być powiązane z jakimkolwiek elementem, więc zostaną zwolnione z użycia, po zakończeniu operacji sprawdzania.

GC wykorzystuje dwa warianty oznaczania danych. Pierwszy to tradycyjny sposób skalarny, w którym każde wykorzystanie elementu w kodzie jest liczone i zapisywane. W przypadku, gdy żaden fragment kodu nie wykorzystuje pamięci, *Garbage Collector* uwalnia 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.

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

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.

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 pełnym zbiorze archiwów zostanie wykonane porównanie narzędzi pod względem liczby wyszukań oraz prędkości wyszukań.

4.1.3 Hipoteza badań

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

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.

Przeprowadzenie badań polegało na uruchomieniu komendy w pierwszej linii (rys. 4.1) i wykonaniu funkcji na algorytmie Morisa Pratta. Wykonano testy na wyszukaniu 3 słów,

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

Rysunek 4.1: Przykład wykonania testu benchmarkowego

które mogły występować w zbiorze danych, ze względu na wcześniej przeanalizowaną zawartość. Tymi słowami były "window", "function", "main".

"founds" jest zmienną przechowującą miejsce znalezionego ciągu wyszukiwanego, a mp to struktura przechowująca implementacje algorytmu oraz bufor wcześniejszego procesowania dla ciągu wyszukiwanego. Pierwsza implementacja nie posiadała tej struktury i bufor wcześniejszego procesowania był obliczany przy każdym przebiegu algorytmu. To znacznie wpłynęło na prędkość działania algorytmu Boyera-Moora.

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

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

W linii 6 (rys. 4.1) wykonujemy Przejście (ang. *Walk*) po drzewie plików w *DIR*, który posiada ścieżkę do zbioru danych oraz przyjmuje funkcję zdefiniowaną jako *WalkFunc* 1. Z powodu potrzeby czytania tylko zawartości plików, wykorzystaliśmy funkcję *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 walidację, aby wszystkie wyniki otrzymane przez algorytm, były zgodne.

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

4.2.1 Zbiór badań

Zbiór danych (tab. 4.1) posiadał znaczną ilość plików pdf, które w niektórych przypadkach można było odczytać. Gdy pdf został stworzony z dokumentu tekstowego takiego jak

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

Tabela 4.1: Ilość danych na podstawie typu MIME

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 odczytania zawartości plików pdf oraz tekstu z obrazów. Takie rozwiązanie nie zostanie użyte w pracy, gdyż nie skupiamy się tu na algorytmach sztucznej inteligencji.

Pliki audio to piosenki oraz podcasty, których nie można łatwo odczytać algorytmem. Do odczytania treści z takich plików można wykorzystać narzędzia dokonujące transkrypcji, czyli konwersujące rozpoznaną mowę na tekst, jednak te rozwiązania bazują na sztucznej inteligencji.

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 podejmuje 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.

Archiwa, które są analizowane też trudno odczytać, gdyż stosowane są różne 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.

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.

Algorytm Morisa-Pratta jest nieznacznie wolniejszy od algorytmu Kurta-Morisa-Pratta. Jest to spowodowane niewielką optymalizacją pomiędzy tymi dwoma algorytmami. Według danych na rysunku 4.3a można zauważyć, KMP w niektórych przypadkach jest wolniejszy, niż algorytm MP, ponieważ posiada większe odchylenie standardowe, co powoduje, że jest mniej stabilny.

Algorytm Boyera-Moore'a wykorzystywany w takich narzędziach jak grep, ma wolniejszy czas egzekucji, co wynika z rys. 4.3a, ale algorytm może zostać napisany w lepszy sposób. Z powodu implementacji, nie wykorzystywaliśmy ponownie bufora wcześniejszego procesowania, co wpływało na znaczne spowolnienie algorytmu.

Implementacja, której wyniki widzimy na grafie 4.3a jest znacznie wolniejsza od pozostałych algorytmów. Powodem jest spędzanie znacznej części czasu na stworzeniu tablicy wcześniejszego procesowania. Wiadome jest, że zawsze sprawdzamy 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 4.3b można zauważyć poprawę, gdy implementacja algorytmu Boyer-Moora wykorzystuje ten sam bufor wcześniejszego procesowania, 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. Wyniki z drugiego wykresu potwierdziły wartość ponownego użycia bufora wcześniejszego procesowania w prędkości wykonania algorytmu.

Ostatni wykres 4.3c 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 zbierane przez *Garbage Collector*. W przypadku użycia stałego bufora zapewniamy, że program nie będzie się pozbywał bufora. Gdy na początku programu utworzymy bufor sami (nie polegając na optymalizacji języka), algorytm Boyera-Moore’a odnotował 5 % poprawę 4.3c w stosunku do poprzedniej implementacji.

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

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

4.3 Badanie ilość otrzymanych wyników z programów

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*).

Wiele z tych narzędzi nie działało poprawnie, gdy próbowaliśmy odczytać zawartość archiwów. Przykładowo narzędzie *ugrep* (rys.4.4) nie rozpoznawało metody kompresji plików zip, co spowodowało, że nie otrzymaliśmy ani jednego rezultatu z programu. To powoduje, że narzędzie zostanie wykluczone z dalszej analizy.

Możliwym rozwiązaniem byłoby rozpakowanie wszystkich plików innym narzędziem, np. 7z. Takie podejście jednak znacznie komplikuje testowanie takiego rozwiązania. Narzędzie powinno być w stanie samo rozpakować i wyszukać wszystkie frazy, które chcemy wyszukać. Dodatkowo narzędzie powinno znajdować frazy zaraz po rozpakowaniu pliku, a nie po rozpakowaniu wszystkich archiwów zacząć przeszukiwać ich zawartość.

Kolejne z wymienionych narzędzi również nie spełnia wymagań. *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. 4.5).

Otrzymany błąd sugeruje, że długość danych w archiwum nie jest zgodna. Dodatkowo mamy 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.

Archiwa jednak są możliwe do otworzenia przez program graficzny Engrampa [3]. Choć wszystkie pliki zostały odczytane, oznacza to, że istnieje możliwość pozyskania części zawartości (rys. 4.6).

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 4.7. Narzędzie niestety nie posiada dokładnej lokalizacji, w której wystąpiło wyszukanie, tylko jest w stanie wskazać miejsce w pliku skompresowanym konkretny bajt. 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.

Gdy odrzucimy pominięte przez gsearch archiwa 4.8, widać liczba wystąpień w gsearch jest większy. Poniżej znajduje się zestawienie ilości znalezionych wystąpień dla wszystkich archiwów przeszukiwanych przez rg, pominiętych archiwów dla rg oraz pominiętych archiwów przeszukanych przez gsearch.

Należy wspomnieć, iż jedno wystąpienie dla ripgrepa 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

Fraza (litera)	rg (wszystkie archiwa)	rg (pominięte)	gsearch (pominięte)
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 dla działających programów

Tabela 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.

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 nie znamy dokładnej wielkości liter frazy, to może to spowodować brak znalezienia frazy (rys. 4.9).

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

4.4 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.

Prędkości wyszukiwania danych za pomocą ripgrepa są znacznie większe, niż te za pomocą gsearch (rys. 4.10). 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.

Dodatkowo ripgrep wykorzystuje operacje SIMD (ang. *Single Instruction, Multiple Data*), czyli wykonanie wielu porównań w ciągu jednego cyklu. Te operacje znacznie przyspieszają wykonanie wyszukiwania.

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.

4.4.1 Wykorzystanie profilowania do odczytania charakterystyki programu

Można również sprawdzić charakterystkę programu gsearch dodając kod na początek wykonania programu (rys. 4.11). 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 przeglądarce otworzy się widok na graf programu (rys. 4.12). Graf jest bardzo skomplikowany do analizy w przypadku, gdy wykorzystujemy rekursję w programie do przeszukiwania folderów i rozpakowywania zawartości.

Można zauważyć, że większość czasu programu wykorzystywana jest na operacje ekstrakcji archiwów i czytania archiwów. Sam algorytm przeszukujący (dolny lewy róg rys. 4.12) wykonuje się jedynie 0.72 sekundy.

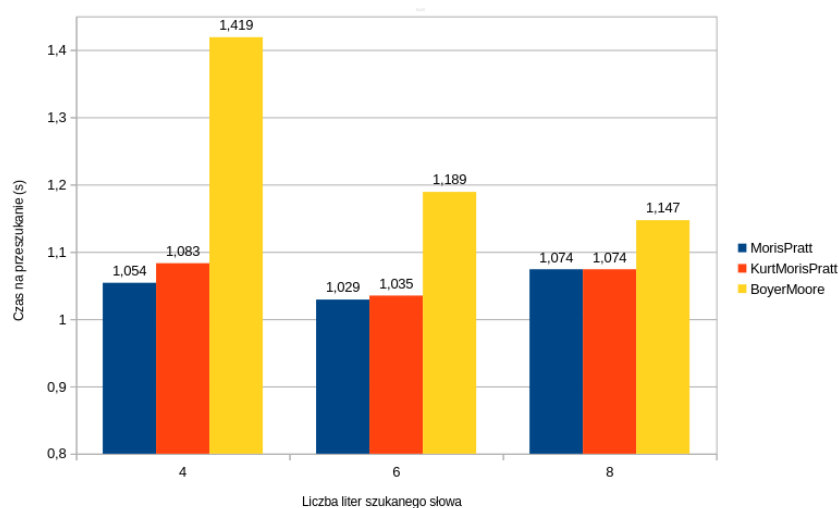
Głównym ograniczeniem prędkości działania jest czas odczytywania zawartości z plików. Funkcje "runtime cgocall" oraz "syscall6" stanowią znaczną część czasu wykonywania programu (rys. 4.13), które wykonują operacje odczytu zawartości treści z dysku.

```

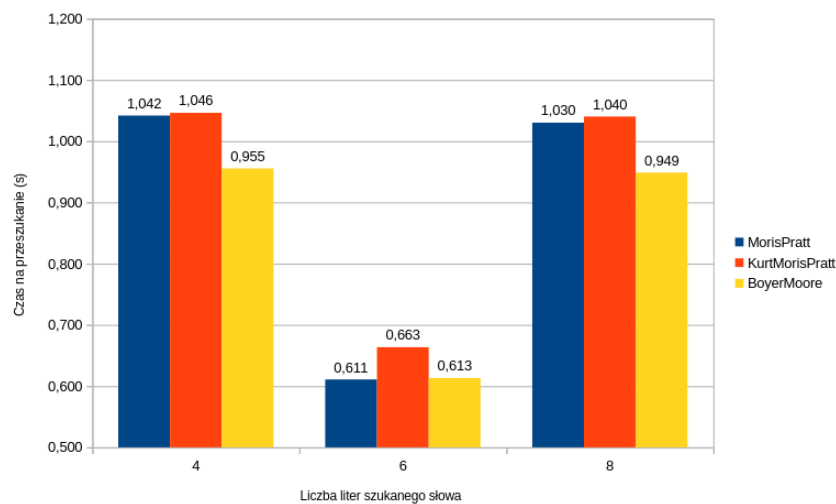
1 goos: linux
2 goarch: amd64
3 pkg: github.com/gadzbi123/algorytmy/regular
4 cpu: Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
5 BenchmarkMorisPrattFunction-8 1 1041257510 ns/op 264623392 B/op
   198976 allocs/op
6 BenchmarkMorisPrattFunction-8 1 1043653365 ns/op 264645080 B/op
   198994 allocs/op
7 BenchmarkMorisPrattFunction-8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

```

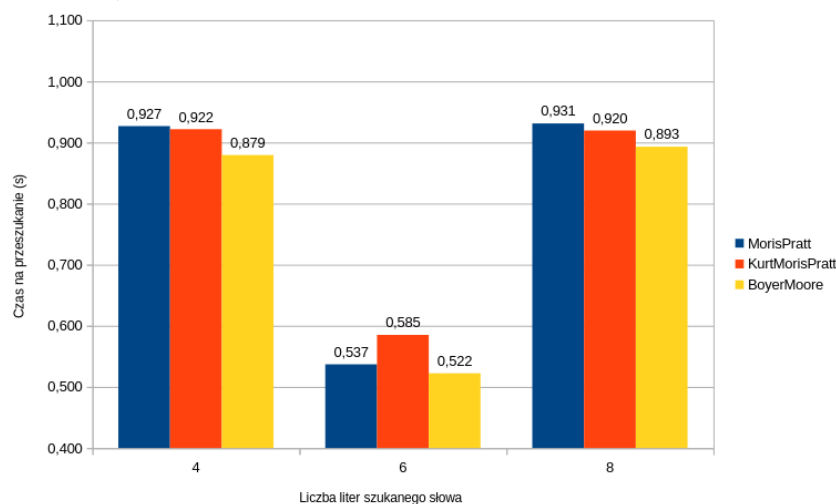
Rysunek 4.2: Przykładowy rezultat performance



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



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



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

Rysunek 4.3: Wykresy kolejnych iteracji na algorytmach

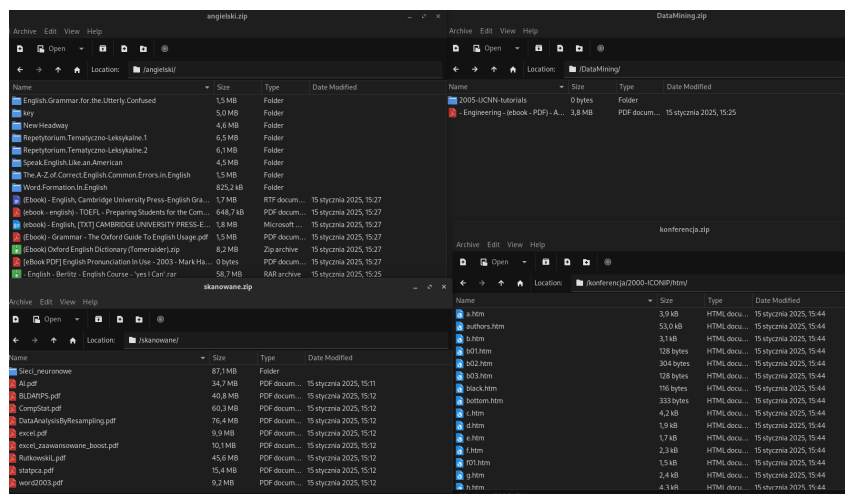
```
[gadzbi@desktop-linux algorytmy]$ time ug -z --max=99 main /run/media/gadzbi/GryIFilmy/baza_mgr
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/ksiazki.zip: unsupported zip compression method 0
ugrep: warning: cannot decompress /run/media/gadzbi/GryIFilmy/baza_mgr/literatura.zip: 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/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    0m0.887s
user    0m0.029s
sys     0m0.009s
```

Rysunek 4.4: Niepoprawne działanie programu ugrep dla archiwów pobranych z chmury

```
✖ [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
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/michal.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/michal.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/papers (1).zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/papers (1).zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/papers.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/papers.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/polskie.zip: invalid compressed data--crc error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/polskie.zip: invalid compressed data--length error
gzip: /run/media/gadzbi/GryIFilmy/baza_mgr/programy_nauczania.zip: invalid compressed data--crc error
```

Rysunek 4.5: Niepoprawne działanie programu zgrep z pomocą finda



Rysunek 4.6: Przykład otwarcia archiwów przez program graficzny Engrampa

```
[gadzbi@desktop-linux algorytmy]$ time rg -lcUz main $DIR*
/run/media/gadzbi/GryIFilmy/baza_mgr/ksiazki.zip:15295
/run/media/gadzbi/GryIFilmy/baza_mgr/literatura.zip:314
/run/media/gadzbi/GryIFilmy/baza_mgr/skanowane.zip:5
/run/media/gadzbi/GryIFilmy/baza_mgr/papers.zip:2778
/run/media/gadzbi/GryIFilmy/baza_mgr/Informatyka.zip:730
/run/media/gadzbi/GryIFilmy/baza_mgr/DataMining.zip:106
/run/media/gadzbi/GryIFilmy/baza_mgr/angielski.zip:65
/run/media/gadzbi/GryIFilmy/baza_mgr/matematyka.zip:1275
/run/media/gadzbi/GryIFilmy/baza_mgr/papers (1).zip:3399
/run/media/gadzbi/GryIFilmy/baza_mgr/konferencja.zip:73203
/run/media/gadzbi/GryIFilmy/baza_mgr/literatura_old.zip:10095

real    5m39,623s
user    0m21,487s
sys     0m33,415s

Suma: 107265
```

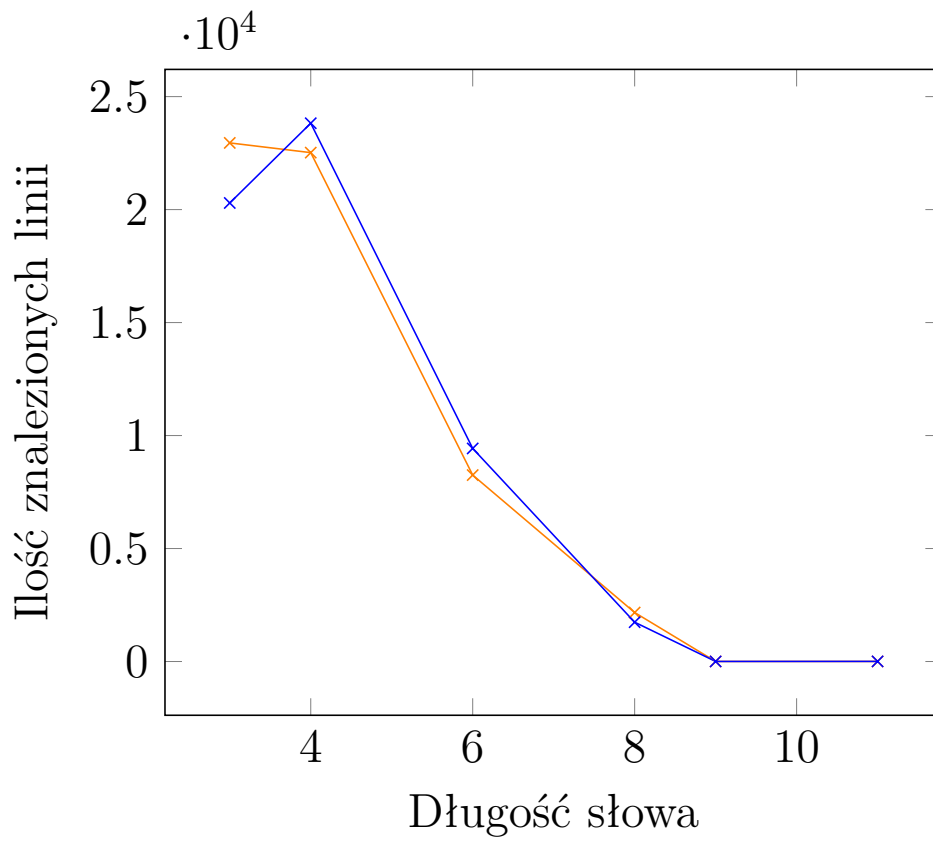
Rysunek 4.7: Przykładowy rezultat wykonania komendy ripgrep ze zmierzonym czasem

```
[gadzbi@desktop-linux algorytmy]$ time rg -lcUz main $DIR*
/run/media/gadzbi/GryIFilmy/baza_mgr/ksiazki.zip:15295
/run/media/gadzbi/GryIFilmy/baza_mgr/literatura.zip:314
/run/media/gadzbi/GryIFilmy/baza_mgr/skanowane.zip:5
/run/media/gadzbi/GryIFilmy/baza_mgr/papers.zip:2778
/run/media/gadzbi/GryIFilmy/baza_mgr/Informatyka.zip:730
/run/media/gadzbi/GryIFilmy/baza_mgr/DataMining.zip:106
/run/media/gadzbi/GryIFilmy/baza_mgr/angielski.zip:65
/run/media/gadzbi/GryIFilmy/baza_mgr/matematyka.zip:1275
/run/media/gadzbi/GryIFilmy/baza_mgr/papers (1).zip:3399
/run/media/gadzbi/GryIFilmy/baza_mgr/konferencja.zip:73203
/run/media/gadzbi/GryIFilmy/baza_mgr/literatura_old.zip:10095

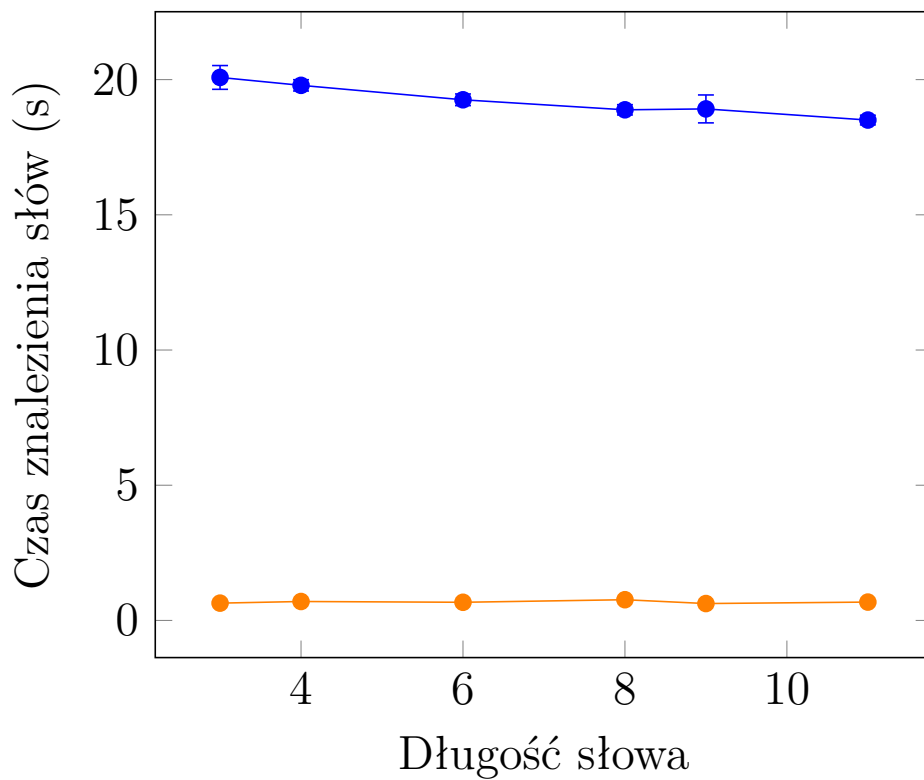
real    5m39,623s
user    0m21,487s
sys     0m33,415s

Suma: 22521
```

Rysunek 4.8: Liczba wystąpień "main" z pominięciem kilku archiwów



Rysunek 4.9: Wykres wystąpień linii z frazami w zależności od długości liter w słowie



Rysunek 4.10: Wykres czasów znalezienia fraz w zależności od długości liter dwóch programach

Rozdział 5

Podsumowanie

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

5.1 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, a istnieje wiele narzędzi, które są darmowe i już na to pozwalają.

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 pdfów, które składają się ze zdjęć i tekstowych skanów treści.

W celu uzyskania lepszych rezultatów można, zamiast wykorzystywać gotową bibliotekę — stworzyć bibliotekę dekompresującą, wszystkich brakujących i nie poprawnie działających formatów archiwów. Jest to jednak dość wymagające zadanie oraz możliwość testowania tego rozwiązania jest ograniczona. Jak bardzo plik może być uszkodzony, żeby można było odczytać z niego dane, które nie są niepoprawne.

Kolejnym usprawnieniem dla programu byłoby wprowadzenie pamięci podręcznej (ang. *caching*). Takie rozwiązanie pozwoliłoby na zapamiętanie plików, które już kiedyś dekompresowano. Pliki skompresowane zwykle nie mogą być zapisane, więc po pierwszej dekompresji można zachować tylko znalezione wystąpienia. Należy jednak przechować informacje o tym, czy suma kontrolna (ang. *hash*) archiwum się nie zmieniła.

Bibliografia

- [1] 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).
- [2] Creative Commons. *Intel 8008*. 2023. URL: https://pl.wikipedia.org/wiki/Intel_8008 (term. wiz. 13.01.2025).
- [3] mate desktop. *Engrampa*. 2025. URL: <https://github.com/mate-desktop/engrampa> (term. wiz. 25.01.2025).
- [4] Google. *Google Drive Storage*. 2024. URL: <https://www.google.com/intx/en-GB/drive/> (term. wiz. 14.01.2025).
- [5] Chenyan Xiong Jeffrey Dalton i Jamie Callan. „The Text REtrieval Conference”. W: *CAsT 2019: The Conversational Assistance Track Overview*. 2020, s. 2–3.
- [6] Ken Thompson Robert Griesemer Rob Pike. *Golang - the programming language*. 2009. URL: <https://go.dev/> (term. wiz. 22.09.2024).
- [7] Larry Page Sergey Brin. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. 1998. URL: <https://www.sciencedirect.com/science/article/abs/pii/S016975529800110X> (term. wiz. 07.12.2024).

Dodatki

Dokumentacja techniczna

Spis skrótów i symboli

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

Lista dodatkowych plików, uzupełniających tekst pracy (jeżeli dotyczy)

W systemie do pracy dołączono dodatkowe pliki zawierające:

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

Spis rysunków

2.1	Przykład użycia programu find	6
2.2	Przykład użycia programu grep	6
2.3	Przykład algorytmu brute force	8
2.4	Historyczne dane cen pamięci w latach 1993-2023	8
2.5	Przykład uprzedniego procesowania tekstu szukanego	9
2.6	Przykład procesowania łańcucha poszukiwanego w algorytmie Morisa Pratta	9
3.1	Przykładowe działanie Garbage Collectora w programie	18
4.1	Przykład wykonania testu benchmarkowego	22
4.2	Przykładowy rezultat performance	29
4.3	Wykresy kolejnych iteracji na algorytmach	30
4.4	Niepoprawne działanie programu ugrep dla archiwów pobranych z chmury	31
4.5	Niepoprawne działanie programu zgrep z pomocą finda	31
4.6	Przykład otwarcia archiwów przez program graficzny Engrampa	32
4.7	Przykładowy rezultat wykonania komendy ripgrep ze zmierzonym czasem	32
4.8	Liczba wystąpień "main"z pominięciem kilku archiwów	32
4.9	Wykres wystąpień linii z frazami w zależności od długości liter w słowie	33
4.10	Wykres czasów znalezienia fraz w zależności od długości liter dwóch programach	33
4.11	Dodanie profilowania do programu gsearch	34
4.12	Przykład grafu wykonań funkcji w gsearch	34
4.13	Zdjęcie grafu funkcji "runtime cgocall"oraz "syscall6"	34

Spis tabel

2.1	Przykład wykorzystania algorytmu KMP	11
3.1	Dystrybucja w danych na podstawie typu plików MIME	16
4.1	Ilość danych na podstawie typu MIME	23
4.2	Tabela zestawienia wyników wyszukiwania dla działających programów . .	26