



**Politechnika  
Śląska**

## **PRACA MAGISTERSKA**

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

**Kacper NITKIEWICZ**

**Nr albumu: 290409**

**Kierunek:** Informatyka Przemysłowa

**Specjalność:** Cyberbezpieczeństwo

**PROWADZĄCY PRACĘ**

**Dr inż. Adrian Smagór**

**KATEDRA Informatyka Przemysłowa**

**Wydział Informatyki Przemysłowej**

**Gliwice 2024**



## **Tytuł pracy**

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

## **Streszczenie**

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

## **Słowa kluczowe**

Algorytmy, Linux, Wyszukiwanie, Wydajność, Optymalizacja

## **Thesis title**

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

## **Abstract**

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

## **Key words**

Algorithms, Linux, Searching, Performance, Optimization



# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>1</b>
1.1	Wprowadzenie do problemu . . . . .	1
<b>2</b>	<b>Analiza tematu wyszukiwania tekstu</b>	<b>3</b>
2.1	Sformułowanie problemu . . . . .	3
2.2	State of art . . . . .	4
2.2.1	Przykładowe narzędzia dostępne do wykorzystania . . . . .	4
2.3	Odniesienia do literatury . . . . .	5
2.4	Opis poznanych rozwiązań . . . . .	5
2.4.1	Algorytm brute force . . . . .	5
2.4.2	Algorytm Morisa-Pratta . . . . .	7
<b>3</b>	<b>[Przedmiot pracy]</b>	<b>11</b>
3.1	Rozwiązanie zaproponowane przez dyplomanta . . . . .	11
3.2	Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi . . . . .	11
<b>4</b>	<b>Badania</b>	<b>13</b>
<b>5</b>	<b>Podsumowanie</b>	<b>15</b>
	<b>Bibliografia</b>	<b>17</b>
	<b>Dokumentacja techniczna</b>	<b>21</b>
	<b>Spis skrótów i symboli</b>	<b>23</b>
	<b>Lista dodatkowych plików, uzupełniających tekst pracy (jeżeli dotyczy)</b>	<b>25</b>
	<b>Spis rysunków</b>	<b>27</b>
	<b>Spis tabel</b>	<b>29</b>



# Rozdział 1

## Wstęp

### 1.1 Wprowadzenie do problemu

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

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

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

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

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

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





# Rozdział 2

## Analiza tematu wyszukiwania tekstu

$$y = \frac{\partial x}{\partial t} \quad (2.1)$$

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

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

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

### 2.1 Sformułowanie problemu

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

Zasadniczym problem naszej pracy jest wyszukiwanie zawartości tekstowej ogromnej ilości plików w różnych formatach. Takie podejście może okazać się problematyczne w przypadku plików dźwiękowych, filmowych czy zdjęć wszelkiego rodzaju.

## 2.2 State of art

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

### 2.2.1 Przykładowe narzędzia dostępne do wykorzystania

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

Przykłady użycia programu find:

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

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

Przykłady użycia programu find:

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

Istnieje również **ripgrep**, który jest sukcesorem wcześniej wymienionego narzędzia. Jego wydajność przewyższa grepa nawet trzydziestokrotnie w niektórych testach sprawnościowych, jednak zazwyczaj jest to niewielki wzrost. Nie jest on niestety domyślnie instalowany

na większości systemów linuxowych. Nie posiada on również wsparcia dla formatów pdf i djvu.

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

## 2.3 Odniesienia do literatury

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

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

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

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

**TODO MORE PAPERS ADDED**

## 2.4 Opis poznanych rozwiązań

### 2.4.1 Algorytm brute force

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

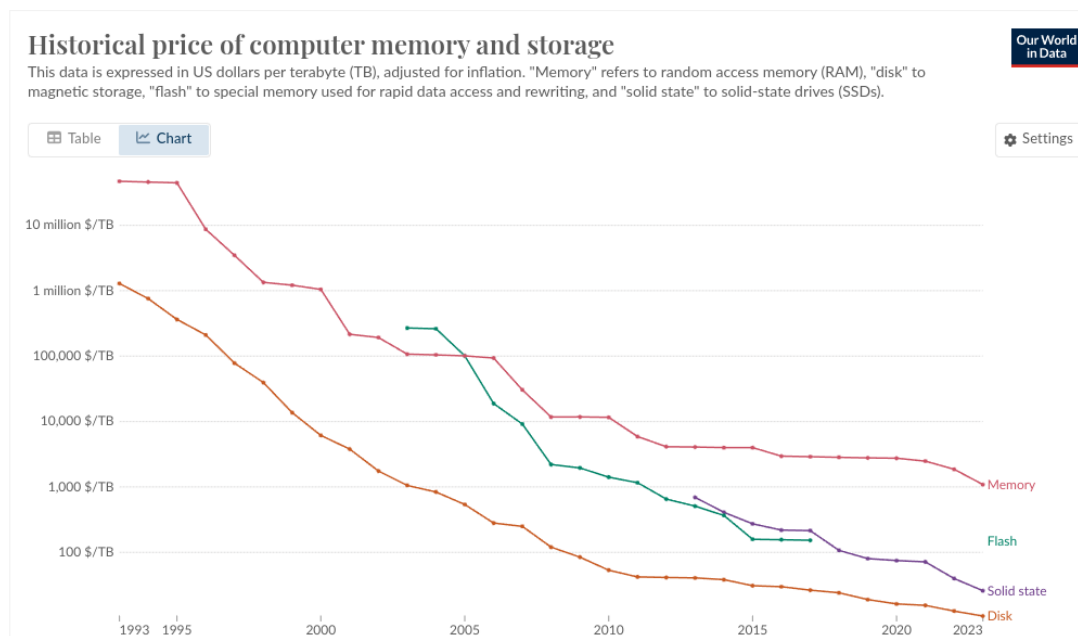
Zaletą tego algorytmu jest to, że nie posiada potrzeby przechowywać żadnych danych w pamięci. Ten algorytm dobrze sprawdza się gdy posiadamy ograniczoną ilość zasobów pamięci, co nie jest problemem w obecnych czasach, gdy pamięć jest stosunkowo tania i szeroko dostępna.

```

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

```

Rysunek 2.1: Przykłady algorytmu brute force



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

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

TYTUL	
wzorzec	ABCABCABDABD
podłańcuch	BCA
rezultat	4

Jeżeli podzielimy wzorzec na dwa procesy wyszukujące algorytmem brute-force, otrzymamy dwa zadania:

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

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

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

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

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

### 2.4.2 Algorytm Morisa-Pratta

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

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

---

```

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

---

Rysunek 2.3: Przykład preprocesowania podłańcucha

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

Dzięki temu zwykle dłuższy tekst znajdujący się w `s` możemy przeanalizować szybciej niż w przypadku algorytmu brute force. Powoduje to niestety problem w przypadku, gdy napis w którym wyszukujemy nie jest wystarczająco długi.

---

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

---

Rysunek 2.4: Przykład preprocesowania podłańcucha

W podstawowej bibliotece języka Golang, w pakiecie *strings* istnieje implementacja metody *Index()*. Nie jest ona jednak w pełni przedstawiona w kodzie, natomiast w jej implementacji można zauważyć, że algorytm brute force jest wykorzystywany tylko w przypadku gdy długość wzorca wynosi więcej niż 64.

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

---

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

---

Rysunek 2.5: Szukanie łańcucha w standardowej bibliotece Golang





# Rozdział 3

## [Przedmiot pracy]

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

### 3.1 Rozwiązanie zaproponowane przez dyplomanta

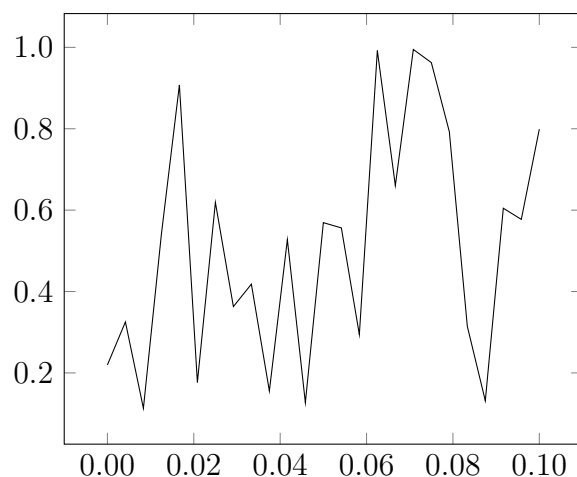
Wykorzystanie kilku znanych algorytmów do przeszukiwania zawartości tekstu i sprawdzenie, który z nich najlepiej sprawdza się pod względem prędkości i dokładności wyszukiwania.

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

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

### 3.2 Uzasadnienie wyboru zastosowanych metod, algorytmów, narzędzi

Do utworzenia programu wykorzystam nowoczesny język programowania Golang [4]. 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).



Rysunek 3.1: Wykres przebiegu funkcji.

Tabela 3.1: Opis tabeli nad nią.

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

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

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

# Rozdział 4

## Badania

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

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

### TODO DIFF IMPLEMENTACJI

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

### TODO SLOW BOYER PORÓWNANIE STATYSTYK ALGOS

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

### PORÓWNANIE STATYSTYK ALGOS

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

### PORÓWNANIE PAMIECI ALGOS



# Rozdział 5

## Podsumowanie

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



# Bibliografia

- [1] Junegunn Choi. *A command-line fuzzy finder - Fzf*. 2024. URL: <https://github.com/junegunn/fzf> (term. wiz. 22.09.2024).
- [2] Chenyan Xiong Jeffrey Dalton i Jamie Callan. „The Text REtrieval Conference”. W: *CAsT 2019: The Conversational Assistance Track Overview*. 2020, s. 2–3.
- [3] Tatsuhiko Kubo. *Algorithm Implementation for Boyer Moore*. 2013. URL: <https://github.com/cubicdaiya/bms/blob/master/bms.go> (term. wiz. 07.11.2024).
- [4] Ken Thompson Robert Griesemer Rob Pike. *Golang - the programming language*. 2009. URL: <https://go.dev/> (term. wiz. 22.09.2024).
- [5] Larry Page Sergey Brin. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. 1998. URL: <https://www.sciencedirect.com/science/article/abs/pii/S016975529800110X> (term. wiz. 07.12.2024).





# Dodatki



# Dokumentacja techniczna



# Spis skrótów i symboli

DNA kwas deoksyrybonukleinowy (ang. *deoxyribonucleic acid*)

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

$N$  liczebność zbioru danych

$\mu$  stopnień przyleżności do zbioru

$\mathbb{E}$  zbiór krawędzi grafu

$\mathcal{L}$  transformata Laplace’a



# Lista dodatkowych plików, uzupełniających tekst pracy (jeżeli dotyczy)

W systemie do pracy dołączono dodatkowe pliki zawierające:

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





# Spis rysunków

2.1	Przykłady algorytmu brute force . . . . .	6
2.2	Historyczne dane cen pamięci w latach 1993-2023 . . . . .	6
2.3	Przykład preprocesowania podłańcucha . . . . .	7
2.4	Przykład preprocesowania podłańcucha . . . . .	8
2.5	Szukanie łańcucha w standardowej bibliotece Golang . . . . .	9
3.1	Wykres przebiegu funkcji. . . . .	12



# Spis tabel

3.1	Opis tabeli nad nią. . . . .	12
-----	------------------------------	----