Asymptotyczna i praktyczna efektywność różnych algorytmów dopasowania ciągów

Julian Leśniak

Praca roczna
opiekun naukowy: dr hab. Krzysztof Turowski

Uniwersytet Jagielloński
Instytut Informatyki Analitycznej
Kraków 2023

Spis treści

Rozdział 1

Wprowadzenie

Problem dopasowania ciągów (*string searching* / *string matching*) jest jednym z najbardziej podstawowych problemów związanych z przetwarzaniem tekstu. Problem ten polega na znalezieniu wystąpienia (zwykle pierwszego lub wszystkich) zadanego słowa (nazywanego wzorcem) jako podsłowo w innym słowie (nazywanym tekstem) i zwróceniu pozycji tegoż dopasowania.

Praca z tekstem praktycznie implikuje napotkanie tego problemu, a tekst jest wciąż jednym z najpowszechniejszych sposobów przechowywania i przekazywania informacji. Wynika z tego, że dopasowanie ciągów może być bezpośrednio stosowane w wielu dziedzinach – od mocno związanych z przetwarzaniem tekstu takich jak lingwistyka, przez informatykę, w tym w szczególności w problemach związanych z sieciami komputerowymi [11], czy bezpieczeństwem [12], aż do biologii [13].

Powstało wiele algorytmów rozwiązujących problem dopasowania ciągów. Jednym z pierwszych ulepszeń algorytmu naiwnego, wykonującego w pesymistycznym przypadku $O(len(text) \cdot len(pat))$ porównań, był algorytm przedstawiony przez D. E. Knutha, J. H. Morrisa Jr. i V. R. Pratta, wykonujący w pesymistycznym przypadku $O(len(text))$ porównań i wykorzystujący $O(len(pat))$ dodatkowej pamięci [2]. Inne podejście zaproponowali R. S. Boyer i J. S. Moore w [1]. Ich algorytm (także wykorzystujący $O(len(pat))$ dodatkowej pamięci) w pesymistycznym przypadku wykonywał, podobnie do podejścia naiwnego, $O(len(text) \cdot len(pat))$ porównań, ale optymistycznie jedynie $O(len(text)/len(pat))$. Wersje tego algorytmu ograniczające pesymistyczną liczbę porównań do $O(len(text))$ zaproponowali m.in.: Z. Galil [3] oraz A. Apostolico i R. Giancarlo [5]. Warto zwrócić uwagę jeszcze na istnienie algorytmów wykorzystujących jedynie $O(1)$ dodatkowej pamięci i $O(len(text))$ porównań ([7], [4], [6]).

Poniższa praca składa się z dwóch części. W rozdziale 2. zostaną przedstawione wybrane algorytmy dopasowania ciągów - szkice ich działania, pseudokody i ograniczenia wykonywanych porównań znaków. W rozdziale 3. zostaną przedstawione wyniki pomiarów liczby porównań znaków wykonywanych przez dane algorytmy dla różnych klas wzorców i tekstów. Liczba porównań znaków jest dobrym miernikiem wydajności, gdyż wszystkie wybrane algorytmy wykonują liniową liczbę operacji elementarnych względem wykonanej liczby porównań znaków.

1.1 Oznaczenia i definicje

Poniżej przedstawione zostały oznaczenia i definicje powtarzających się pojęć:

- **pat** – wzorzec
- **text** – tekst
- $\mathcal{A}$ – alfabet
- $w[i]$ – i-ty znak słowa w
- $w[i : j]$ – podsłowo słowa w zaczynające się na pozycji i (włącznie), a kończące na pozycji j (włącznie)
- wv – słowo złożone ze wszystkich znaków słowa w po których występują wszystkie znaki słowa v
- w^e – słowo złożone z e-krotnego powtórzenia słowa w
- **len**(w) – długość słowa w
- **podsłowo** słowa w – takie słowo u, że istnieją słowa v' i v'' dla których $w = v'uv''$
- **prefiks** słowa w – podsłowo słowa w postaci $w[1 : i]$ (tj. podsłowo z v' pustym)
- **prefiks właściwy** słowa w – prefiks słowa w różny od całego słowa w (tj. podsłowo z v' pustym i v'' niepustym)
- **sufiks** słowa w – podsłowo słowa w postaci $w[i : len(w)]$ (tj. podsłowo z v'' pustym)
- **sufiks właściwy** słowa w – sufiks słowa w różny od całego słowa w (tj. podsłowo z v'' pustym i v' niepustym)
- **okres** słowa w – takie słowo u, że $w = u^e u'$, $e \geq 1$ i u' jest właściwym prefiksem słowa u
- **per**(w) – długość najkrótszego okresu słowa w
- słowo w jest **podstawowe** – nie istnieje słowo u, takie że $w = u^e$ oraz e jest liczbą naturalną większą od 1

Rozdział 2

Opis wybranych algorytmów

2.1 Algorytm naiwny (BF)

Zacznijmy od przedstawienia naiwnego podejścia do wyszukania wzorca w tekście. Zapoznanie się z nim pozwala na łatwiejsze zrozumienie zasad działania opisywanych później algorytmów. Ponadto, porównanie jego wydajności działania z bardziej wysublimowanymi programami pozwala na ocenę, czy podejścia te warto stosować w praktyce.

2.1.1 Opis algorytmu

Algorytm ten działa w następujący sposób. Dla każdej pozycji $1 \leq i \leq len(text) - len(pat)$ w tekście sprawdza równość $text[i : i + len(pat) - 1]$ z pat porównując je znak po znaku. Sprawdzanie zgodności dla danego i jest przerywane w momencie wystąpienia nierówności znaków lub stwierdzenia równości z całym wzorcem – w tym przypadku i jest zwrócone jako pozycja wystąpienia wzorca w tekście.

Algorytm ten przedstawiony w pseudokodzie wygląda następująco:

Algorytm 1 Algorytm naiwny

```
1:  i ← 1
2:  while i ≤ len(text) − len(pat) + 1 do
3:      j ← 0
4:      while j < len(pat) and text[i + j] = pat[j + 1] do
5:          j ← j + 1
6:      end while
7:      if j = len(pat) then
8:          return i
9:      end if
10:     i ← i + 1
11: end while
```

2.1.2 Poprawność

Łatwo zauważyć, że algorytm ten poprawnie zwraca wystąpienia wzorca w tekście. Weryfikuje on równość wzorca z każdym podsłowem tekstu długości $len(pat)$ (więc sprawdza wszystkie potencjalne sposoby dopasowania) i zwraca informację od dopasowaniu jedynie po stwierdzeniu równości danego podsłowa tekstu z wzorcem.

2.1.3 Złożoność

Algorytm ten ma złożoność pesymistyczną $O(len(pat) \cdot len(text))$ porównań znaków, gdzie $len(pat)$ jest długością wzorca, a $len(text)$ długością teksu. Górne ograniczenie wynika bezpośrednio z liczby możliwych wykonań pętli *while* z 2. linii (rzędu $O(len(text))$) przemnożonych przez wykonania pętli *while* z 4. linii (rzędu $O(len(pat))$). Na przykład liczba porównań tego rzędu wystąpi dla wzorca równego $a^n b$ i tekstu $a^{2n} b$ (algorytm wykona wtedy dokładnie $(n+1)^2$ porównań znaków).

2.2 Algorytm Knutha, Morrisa i Pratta (KMP)

D. E. Knuth, J. H. Morris i V. R. Pratt zaproponowali algorytm, który ulepsza podejście naiwne, korzystając z obserwacji, że w przypadku wystąpienia niezgodności podczas porównywania wzorca z tekstem możliwe jest pominięcie części porównań [2]. Możliwy przeskok jest określany na podstawie obliczonej początkowo dla danego wzorca tablicy, która uzależnia go od pozycji we wzorcu, na której wystąpiła nierówność znaków.

2.2.1 Opis algorytmu

Ogólna zasada działania tego algorytmu jest podobna do podejścia naiwnego, jednak w przypadku niedopasowania zaczyna on sprawdzać nie kolejny znak, ale pierwszy od którego jeszcze jest możliwe dopasowanie wzorca. W tym celu wykorzystywana jest tablica (nazwijmy ją *next*) informująca, od którego znaku należy kontynuować porównywanie.
Powyższy algorytm w pseudokodzie, przy założeniu że mamy już dostęp do tablicy *next*, wygląda następująco:

Algorytm 2 Algorytm KMP

```
1:  j ← 1
2:  k ← 1
3:  while j ≤ len(pat) and k ≤ len(text) do
4:      while j > 0 and text[k] ≠ pat[j] do
5:          j ← next[j]
6:      end while
7:      k ← k + 1
8:      j ← j + 1
9:      if j > len(pat) then
10:         return k − len(pat)
11:     end if
12: end while
```

Teraz przedstawmy dwie możliwe wersje tablicy *next*: tablicę słabych prefikso-sufiksów (*wps*) i tablicę mocnych prefikso-sufiksów (*sps*). Obie można wykorzystać bezpośrednio w powyższym algorytmie, ale ich zastosowanie potencjalnie prowadzi do innej liczby wykonywanych porównań. Algorytm korzystający z tablicy *wps* będziemy nazywać MP, a korzystający z tablicy *sps* – KMP.

Tablica słabych prefikso-sufiksów

Zdefiniujmy tablicę słabych prefikso-sufiksów (nazywaną dalej *wps*) dla zadanego wzorca (nazywanego dalej *pat*). Niech $wps[j]$ będzie największym i takim, że $i < j$, $pat[1 : i-1] = pat[j-i+1 : j-1]$ oraz przyjmijmy $wps[1] \leftarrow 0$.

Zauważmy, że jeśli $pat[j] = pat[wps[j]]$ to $wps[j+1] = wps[j] + 1$, a jeśli nie, to użyjemy sposobu analogicznego do przedstawionego w algorytmie 2, gdyż obliczanie *wps* jest równoważne wyszukiwaniu wzorca w samym sobie.

Powyższa metoda przedstawiona w pseudokodzie wygląda następująco:

Algorytm 3 Algorytm liczenia słabych prefikso-sufiksów

```
1: j ← 1
2: while j < len(pat) do
3:     t ← wps[j]
4:     while t > 0 and pat[j] ≠ pat[t] do
5:         t ← wps[t]
6:     end while
7:     wps[j + 1] ← t + 1
8:     j ← j + 1
9: end while
```

Tablica silnych prefikso-sufiksów

Zdefiniujmy tablicę silnych prefikso-sufiksów (nazywaną dalej *sps*) dla zadanego wzorca. Niech $sps[j]$ będzie największym i takim, że $i < j$, $pat[1 : i-1] = pat[j-i+1 : j-1]$, $pat[i] \neq pat[j]$ oraz przyjmijmy $sps[1] \leftarrow 0$.

Zauważmy, że jedyną różnicą między zadanymi definicjami jest dodatkowy wymóg $pat[i] \neq pat[j]$, więc:

$$sps[j] = \begin{cases} wps[j], & \text{dla } pat[j] \neq pat[wps[j]] \\ sps[wps[j]], & \text{dla } pat[j] = pat[wps[j]] \end{cases}$$

Powyższy sposób przedstawiony w pseudokodzie wygląda następująco:

Algorytm 4 Algorytm liczenia silnych prefikso-sufiksów

```
1:  j ← 1
2:  t ← 0
3:  sps[1] ← 0
4:  while j < len(pat) do
5:      while t > 0 and pat[j] ≠ pat[t] do
6:          t ← sps[t]
7:      end while
8:      t ← t + 1
9:      j ← j + 1
10:     if pat[j] = pat[t] then
11:         sps[j] ← sps[t]
12:     else
13:         sps[j] ← t
14:     end if
15: end while
```

2.2.2 Poprawność

Aby udowodnić poprawność algorytmu 2 można użyć niezmiennika: „niech $p = k - j$ (więc p to pozycja w tekście poprzedzająca pierwszy znak wzorca w danym ustawieniu). Wtedy $text[p+i] = pat[i]$ dla $1 \leq i < j$ (więc jest dopasowanych pierwszych $j-1$ znaków wzorca, jeśli $j > 0$), ale dla $0 \leq t < p$ istnieje $text[t+i] \neq pat[i]$ dla pewnego i, gdzie $1 \leq i \leq m$ (więc nie ma możliwego dopasowania dla całego wzorca na lewo od p)". Zauważmy, że program będzie poprawny, jeśli zadany niezmiennik będzie utrzymywany dla operacji $j \leftarrow next[j]$.

Program ustawia $j \leftarrow next[j]$ tylko wtedy, gdy $j > 0$ i ostatnie j znaków z wejścia (wliczając $text[k]$) wynosiło: $pat[1 : j-1]x$, gdzie $x \neq pat[j]$. Teraz chcemy znaleźć minimalne przesunięcie wzorca, takie że jego prefiks będzie pasował do odpowiednich znaków w tekście, czyli chcemy żeby $next[j]$ było największym i takim, że $i < j$ i że ostatnie i znaków z tekstu to: $pat[1 : i-1]x$, gdzie $pat[i] \neq pat[j]$, a jeśli takie i nie istnieje to $next[j] \leftarrow 0$. Z przyjętej definicji $next[j]$ bezpośrednio wynika poprawność przyjętego niezmiennika. Zauważmy, że przyjęta definicja $next$ jest zgodna z definicją tablicy silnych prefikso-sufiksów, więc jeśli przyjmiemy ją za tablicę $next$, to algorytm będzie poprawny.

Skomentujmy jeszcze krótko poprawność przyjęcia tablicy słabych prefikso-sufiksów jako $next$. Skupmy się na pętli while znajdującej się w liniach 4. i 5. algorytmu 2, gdyż tylko tam wystąpi różnica w działaniu programu. Jeśli $wps[j] = sps[j]$, to nie wystąpi różnica w działaniu algorytmu. W przeciwnym wypadku wiemy, że $pat[wps[j]] = pat[j]$, a ponieważ $text[k] \neq pat[j]$ (zgodnie z linią 4.), więc $text[k] \neq pat[wps[j]]$. Spowoduje to kolejne wykonanie pętli. Wynika z tego, że algorytm wykorzystujący tablicę słabych prefikso-sufiksów ma szansę opuścić tą pętlę jedynie w wykonaniach, w których $wps[j] = sps[j]$, więc pod względem zwróconego wyniku jest równoważny wersji wykorzystującej silne prefikso-sufiksy.

2.2.3 Złożoność

Rozważmy teraz złożoność algorytmu 2 (mierzoną liczbą porównań znaków) z pominięciem obliczania tablicy $next$. Zauważmy, że podczas całego działania programu j jest inkrementowane tyle razy co k (linie 7. i 8.), a k jest inkrementowane co najwyżej $len(text)$ razy (linia 3.), więc j też jest inkrementowane co najwyżej $len(text)$ razy. Jedyne porównanie znaków występuje w linii 4. i zawsze następuje po nim zmniejszenie j (gdyż $next[j] < j$) w przypadku niezgodności znaków lub inkrementacja - w przypadku zgodności. Ponieważ j jest zawsze nieujemne, a liczba inkrementacji jest ograniczona przez $len(text)$, więc liczba zmniejszeń jest także ograniczona przez $len(text)$. Wynika z tego, że liczba wszystkich wykonanych porównań jest ograniczona przez $2 \cdot len(text)$. Ograniczenie to jest prawdziwe zarówno dla przyjęcia słabych, jak i mocnych prefikso-sufiksów za tablicę $next$. Warto jednak zauważyć, że ponieważ dla każdego j, $sps[j] \leq wps[j]$, więc dla danego wzorca i tekstu liczba porównań wykonanych przez algorytm 2 korzystający z sps będzie mniejsza lub równa liczbie porównań wykonanej przez algorytm korzystający z wps (nie biorąc pod uwagę porównań wykonanych podczas obliczania samej tablicy $next$).

Złożoność algorytmu 3 jest całkowicie analogiczna – w tym przypadku liczba porównań jest ograniczona przez $2 \cdot len(pat)$.

W przypadku algorytmu 4 ograniczenie wynosi $3 \cdot len(pat)$, z powodu dodatkowego porównania w linii 10., których liczba także jest ograniczona przez $len(pat)$.

2.3 Algorytm Boyera i Moore'a (BM)

R. S. Boyer i J. S. Moore zaproponowali algorytm, który korzysta z obserwacji, że sprawdzanie dopasowania wzorca od prawej strony zamiast od lewej pozwala na zdobycie większej ilości informacji [1]. Pozwala to na wydajniejsze pomijanie porównań w przypadku znalezienia niezgodności między wzorcem i tekstem.

2.3.1 Opis algorytmu

Algorytm sprawdza dopasowanie wzorca na danej pozycji, zaczynając od jego prawego znaku i kontynuując porównania kolejnych znaków na lewo aż do momentu wystąpienia niezgodności (lub dopasowania całego wzorca). W przypadku wystąpienia niezgodności korzysta on z dwóch obliczonych wcześniej tablic pomocniczych – nazwijmy je przesunięciem nieodpowiedniego znaku (*bad character shift – bcs*) i przesunięciem dobrego sufiksu (*good suffix shift – gss*).

Niech tablica *bcs* dla każdego znaku z rozważanego alfabetu utrzymuje odległość najbardziej wysuniętego na prawo wystąpienia tego znaku we wzorcu od prawego końca wzorca lub długość wzorca, jeśli znak ten we wzorcu nie występuje.

Niech tablica gss dla każdego indeksu i od 1 do $len(pat)$ będzie odległością między i oraz początkiem najbardziej wysuniętego na prawo dopasowania $x(pat[i+1 : len(pat)])$, gdzie $x \neq pat[i]$, do pat powiększoną o $len(pat) - i$.
Algorytm w pseudokodzie, przy założeniu że mamy dostęp do tablic bcs i gss, wygląda następująco:

Algorytm 5 Algorytm BM

```
1:  i ← len(pat)
2:  while i ≤ len(text) do
3:      j ← len(pat)
4:      while j > 0 and text[i] = pat[j] do
5:          j ← j − 1
6:          i ← i − 1
7:      end while
8:      if j = 0 then
9:          return i + 1
10:     end if
11:     i ← i + max(bcs[text[i]], gss[j])
12: end while
```

W części praktycznej zostanie sprawdzona także prostsza wersja algorytmu, nazwijmy ją BMB (Boyer Moore Basic), która nie korzysta z tablicy bcs, więc od powyższego pseudokodu różni się tym, że 11. linia przyjmuje postać: $i \leftarrow i + gss[j]$.

2.3.2 Poprawność

Zauważmy, że algorytm przesuwa wzorzec względem tekstu jedynie o odpowiednie wartości tablic *bcs* i gss (linia 11.).

Uzasadnijmy najpierw, że przesunięcie wzorca zgodnie z przesunięciem nieodpowiedniego znaku nie pominie żadnego dopasowania. Załóżmy, że podczas porównywania wzorca z tekstem znajdziemy w tekście dany znak *char* na pozycji w tekście k, który jest różny od odpowiadającego mu znakowi we wzorcu. Wtedy najwcześniejsze możliwe dopasowanie wzorca to takie, w którym na pozycji k znajdzie się najbardziej wysunięty na prawo znak *char* we wzorcu lub ustawienie początku wzorca na pozycji $k + 1$, gdy *char* w ogóle nie występuje we wzorcu, co jest zgodne z definicją tablicy *bcs*.

Teraz uzasadnijmy, że przesunięcie wzorca zgodnie z przesunięciem dobrego sufiksu nie pominie żadnego dopasowania. Załóżmy, że podczas porównywania wzorca z tekstem znajdziemy na pozycji k w tekście znak różniący się od znaku na pozycji j we wzorcu (dla danego ułożenia wzorca względem tekstu). Ponieważ algorytm BM porównuje wzorzec z tekstem, zaczynając od prawej strony, więc oznacza to, że znaki tworzące sufiks wzorca od pozycji $j + 1$ były zgodne z odpowiadającymi im znakami z tekstu. Na podstawie tej informacji wiemy, że prawidłowe dopasowanie nie może wystąpić wcześniej niż dla ustawienia wzorca w taki sposób, że na pozycjach w tekście od $k+1$ znajdzie się fragment wzorca odpowiadający sufiksowi od $j+1$, a dopasowany na pozycji k znak będzie inny niż j-ty znak wzorca i będzie to takie dopasowanie wysunięte najbardziej na prawo. Na tej obserwacji jest z kolei oparta definicja tablicy gss.

Ponieważ żadne z powyższych przesunięć nie może pominąć odpowiedniego dopasowania, więc algorytm działa poprawnie.

2.3.3 Złożoność

Analiza ilości porównań wykonywanych przez ten algorytm jest trudna, gdyż jest ona silnie uzależniona od wielkości alfabetu, długości wzorca i statystycznego rozkładu występowania poszczególnych znaków we wzorcu i tekście. Z tego powodu, praktyczne pomiary wydają się lepszym wyznacznikiem wydajności. Przytoczmy jednak kilka teoretycznych ograniczeń.

W oryginalnej pracy zauważono, że algorytm ten w optymistycznym przypadku wykonuje jedynie rzędu $len(text)/len(pat)$ porównań i wynik taki jest osiągany średnio dla alfabetu dużego względem długości wzorca.

D. E. Knuth udowodnił, że nawet wersja BMB wykonuje co najwyżej $6 \cdot len(text)$ porównań w przypadku braku wystąpienia wzorca w tekście [2]. Ograniczenie to poprawił R. Cole [8], który dowiódł, że przy dodatkowym założeniu braku okresowości wzorca (tj. nie jest on postaci wv^k, gdzie w jest sufiksem właściwym v), w przypadku niewystąpienia wzorca w tekście algorytm ten wykonuje co najwyżej $3 \cdot len(text)$ porównań.

W przypadku wyszukiwania jedynie pierwszego wystąpienia wzorca i jego obecności w tekście ograniczenia te zwiększają się o $len(pat)$ porównań. Jednak w przypadku wyszukiwania wszystkich wystąpień liczba porównań jest już rzędu $O(len(text) \cdot len(pat))$, czego najlepszym przykładem jest wyszukiwanie wzorca postaci a^m w tekście postaci a^n.

2.4 Algorytm Apostolico i Giancarlo (AG)

A. Apostolico i R. Giancarlo przedstawili algorytm, oparty na algorytmie Boyera i Moore'a, który usprawnia go poprzez przechowywanie informacji o dopasowanych wcześniej sufiksach wzorca do tekstu [5]. Pozwala to na zmniejszenie pesymistycznej liczby porównań do liniowej względem długości tekstu.

2.4.1 Opis algorytmu

Algorytm ten analogicznie do algorytmu BM sprawdza dopasowanie wzorca na danej pozycji, zaczynając od jego prawego znaku i kontynuując porównania kolejnych znaków na lewo aż do momentu wystąpienia niezgodności (lub dopasowania całego wzorca), a w przypadku wystąpienia niezgodności korzysta on z tablic przesunięcia nieodpowiedniego znaku (bcs) i przesunięcia dobrego sufiksu (gss) opisanych w części poświęconej algorytmowi BM. Korzysta on jednak dodatkowo z dwóch kolejnych struktur – aktualizowanej podczas działania algorytmu tablicy $skip$ i możliwej do szybkiego obliczenia na podstawie odpowiedniego preprocessingu wzorca funkcji $Q(i,k)$.

Opiszmy teraz, czym jest tablica $skip$. Przyjmijmy, że w danym momencie działania algorytm sprawdza dopasowanie wzorca dla ułożenia, w którym jego prawy koniec odpowiada l-temu znakowi tekstu. Zauważmy, że gdy algorytm przerywa dalsze porównywanie (z powodu wystąpienia niezgodności lub pełnego dopasowania) po porównaniu k zgodnych znaków, to posiada on informację o tym, że $l-k+1$ do l znaki tekstu są równe sufiksowi wzorca o długości k. Z pomocą tej informacji jest możliwe późniejsze pomijanie porównań ze znakami $l-k+1$ do l tekstu. Niech więc tablica $skip$ o długości $len(text)$ będzie inicjalizowana zerami, a w czasie działania algorytmu odpowiednio uzupełniana wartościami takimi, że $skip[l]=k$ oznacza, że $text[l-k+1:l]=pat[len(pat)-k+1:len(pat)]$.

Zdefiniujmy teraz funkcję boolowską $Q(i,k)$. Intuicyjnie, ma ona zwracać $false$, jeśli dopasowanie sufiksu wzorca o długości k w taki sposób, że jego prawy koniec odpowiada i-temu znakowi wzorca, nie zawiera niezgodności. Formalnie funkcja ta jest zdefiniowana w następujący sposób:

$$Q(i,k)=\begin{cases} true, & \text{jeśli } (k\le i \text{ and } pat[i-k+1:i]\neq pat[len(pat)-k+1:len(pat)]) \\ & \text{lub } (k>i \text{ and } pat[1:i]\neq pat[len(pat)-i+1:len(pat)]) \\ false, & \text{w p. p.} \end{cases}$$

Tablicę pomocniczą zawierającą długości najdłuższych sufiksów możliwych do dopasowania na danym indeksie wzorca umożliwiającą szybkie zwracanie wartości funkcji $Q(i,k)$ można obliczyć w czasie liniowym od długości wzorca, korzystając ze zmodyfikowanego algorytmu KMP.
Algorytm AG w pseudokodzie, przy założeniu że mamy już dostęp do tablic bcs i gss oraz funkcji Q, jest przedstawiony poniżej:

Algorytm 6 Algorytm AG

```
1:  i ← len(pat)
2:  while i ≤ len(text) do
3:      j ← len(pat)
4:      while j > 0 do
5:          if Q(j, skip[i − len(pat) + j]) then
6:              break
7:          else if ((skip[i − len(pat) + j] = 0) and (pat[j] ≠ text[i − len(pat) + j])) then
8:              break
9:          else
10:             j ← j − max(1, skip[i − len(pat) + j])
11:         end if
12:     end while
13:     if j ≤ 0 then
14:         return i − len(pat) + 1
15:     end if
16:     skip[i] ← len(pat) − j
17:     i ← i + max(bcs[text[i − len(pat) + j]], gss[j])
18: end while
```

W części praktycznej zostanie sprawdzona także prostsza wersja algorytmu, nazwijmy ją AGB (Apostolico Giancarlo Basic), która nie korzysta z tablicy *bcs*, więc od powyższego pseudokodu różni się tym, że 15. linia przyjmuje postać: $i \leftarrow i + gss[j]$.

2.4.2 Poprawność

Poprawność tego algorytmu wynika z poprawności algorytmu BM, gdyż algorytm ten sprawdza wszystkie pozycje potencjalnych dopasowań, które sprawdziłby algorytm BM. Wykorzystana dodatkowo funkcja Q oraz tablica *skip* pozwalają na sprawdzanie w czasie stałym (co w algorytmie BM mogło zająć czas rzędu $O(len(pat))$ równości pewnych podsłów tekstu i wzorca, nie powodują więc one różnic w zwracanych wartościach. Poprawność działania Q i *skip* wynika bezpośrednio z ich definicji.

2.4.3 Złożoność

Rozważmy teraz złożoność algorytmu 6 (mierzoną liczbą porównań znaków) z pominięciem obliczania tablic pomocniczych. Zacznijmy od obserwacji, że algorytm ten podobnie do algorytmu BM, którego jest usprawnieniem, w optymistycznym przypadku wykonuje rzędu $len(text)/len(pat)$ porównań. Rozważmy teraz jednak ograniczenie górne liczby porównań, które jest lepsze niż w przypadku algorytmu 5.

Przedstawmy najpierw wraz z dowodem ograniczenie wskazane w oryginalnej pracy [5]. Przyjmijmy, że *and* w 5. linii nie sprawdza drugiego warunku, jeśli pierwszy nie jest spełniony. Zauważmy, że każde porównanie znaków tekstu i wzorca prowadzi do dopasowania lub niedopasowania. Jeśli dojdzie do dopasowania, to dany znak zostanie uwzględniony w aktualizacji tablicy *skip* w 16. linii i nie będzie później bezpośrednio porównywany.

Wynika z tego, że każdy znak tekstu może być co najwyżej raz poddany porównaniu ze znakiem z wzorca kończącym się dopasowaniem, więc takich porównań jest łącznie co najwyżej $len(text)$. Teraz zauważmy, że każde niedopasowanie znaku prowadzi do zwiększenia zmiennej i, więc takich porównań może być co najwyżej $len(text) - len(pat) + 1$. Łącznie więc, wszystkich porównań jest nie więcej niż $2 \cdot len(text) - len(pat) + 1$.

Ograniczenie to zostało poprawione przez M. Crochemore'a i T. Lecroq'a, którzy dowiedli, że algorytm AG nie wykonuje więcej niż $\frac{3}{2} \cdot len(text)$ porównań i że ograniczenie to jest ścisłe, gdyż przykładowo dla wyszukiwania wzorca postaci $a^{m-1}ba^mb$ w tekście postaci $(a^{m-1}ba^mb)^e$, gdzie $m, e > 0$ i $len(text) = (2m+1)e$, algorytm wykona $2m+1+(3m+1)e$, czyli $\frac{3m+1}{2m+1}len(pat) - m$ porównań znaków [10].

2.5 Algorytm Crochemore'a (Cr)

M. Crochemore przedstawił algorytm oparty na pewnych własnościach kombinatorycznych słów nad uporządkowanym alfabetem, w szczególności związanych z długością cykli [7]. Algorytm ten wykonuje w pesymistycznym przypadku liniową liczbę porównań względem długości tekstu, a jednocześnie wyróżnia się brakiem wstępnego przetwarzania wzorca i wymaganiem jedynie stałej dodatkowej pamięci.

2.5.1 Opis algorytmu

Algorytm ten sprawdza dopasowanie wzorca w danym miejscu tekstu przez porównywanie znaków począwszy od lewej strony (podobnie jak w algorytmie KMP). W przypadku wystąpienia niezgodności i-tego znaku tekstu z j-tym znakiem wzorca, algorytm liczy długość najkrótszego okresu słowa $pat[1:j-1]text[i]$ lub jego pewne ograniczenie dolne. W celu obliczenia tych wartości algorytm korzysta z rozkładu MS (Maximal Suffix).

Definicja 2.5.1 (Rozkład MS)**.** Niech $x = uv$ oraz v będzie największym leksykograficznie sufiksem słowa x. Niech $v = w^e w'$, gdzie w jest najkrótszym okresem słowa v, a w' jest właściwym prefiksem w. Wtedy rozkładem MS niepustego słowa x nazywamy ciąg (u, w, e, w').

Z własności rozkładu MS wynikają dwa twierdzenia, z których bezpośrednio korzysta rozważany algorytm:

1. Niech $uw^e w'$ będzie rozkładem MS niepustego słowa x i niech $v = w^e w'$ będzie największym leksykograficznie sufiksem x. Wtedy:
   Jeśli u jest sufiksem w to $per(x) = per(v) = len(w)$.
   W przeciwnym przypadku $per(x) > \max(len(u), min(len(v), len(uw^e))) \geq \frac{len(x)}{2}$.
2. Niech (u, w, e, w') będzie rozkładem MS niepustego słowa x. Załóżmy, że $per(x) = len(w)$ oraz $e > 1$. Wtedy:
   $(u, w, e-1, w')$ jest rozkładem MS słowa $x' = uw^{e-1}w'$. W szczególności, słowo $w^{e-1}w'$ jest maksymalnym sufiksem x' i $per(w^{e-1}) = len(w)$.

Algorytm w pseudokodzie wygląda następująco:

Algorytm 7 Algorytm Cr

```
1: function nextMaximalSuffix(x[1 : n], (i, j, k, p))
2:     while j + k ≤ n do
3:         if (x[i + k] = x[j + k]) then
4:             if k = p then
5:                 j ← j + p; k ← 1
6:             else
7:                 k ← k + 1
8:             end if
9:         else if x[i + k] > x[j + k] then
10:            j ← j + k; k ← 1; p ← j − i
11:        else
12:            i ← j; j ← i + 1; k ← 1; p ← 1
13:        end if
14:    end while
15:    return (i, j, k, p)
16: end function
17: pos ← 0; m ← 1; (i, j, k, p) ← (0, 1, 1, 1)
18: while pos + m ≤ len(text) do
19:    while pos + m ≤ len(text) and m ≤ len(pat) and text[pos + m] = pat[m] do
20:        m ← m + 1
21:    end while
22:    if m = len(pat) + 1 then
23:        return pos
24:    end if
25:    (i, j, k, p) ← nextMaximalSuffix(pat[1 : m − 1]text[pos + m], (i, j, k, p))
26:    y ← pat[i + 1 : m − 1]text[pos + m]
27:    if pat[1 : i] = y[p − i + 1 : p] then
28:        pos ← pos + p; m ← m − p + 1;
29:        if j − i > p then
30:            j ← j − p
31:        else
32:            (i, j, k, p) ← (0, 1, 1, 1)
33:        end if
34:    else
35:        pos ← pos + max(i, min(m − i, j)) + 1; m ← 1; (i, j, k, p) ← (0, 1, 1, 1)
36:    end if
37: end while
```

W ramach komentarza do powyższego pseudokodu warto przytoczyć interpretację zmiennych i, j, k i p. Zachowując notację rozkładu MS jako (u, w, e, w'), wartości danych zmiennych wynoszą odpowiednio: $i = len(u)$; $j = len(uw^e)$; $k = len(w') + 1$; $p = len(w) = per(v)$. Przy interpretacji pseudokodu warto zwrócić uwagę, że warunek z linii 27. odpowiada wyżej przytoczonemu twierdzeniu 1., a warunek z linii 29. twierdzeniu 2.

2.5.2 Poprawność

Zacznijmy od obserwacji, że w przypadku niedopasowania i-tego znaku wzorca do j-tego znaku tekstu nie może istnieć dopasowanie zaczynające się wcześniej niż na $j-i+per(pat[1 : i-1]text[j])$ pozycji tekstu. Wynika to z tego, że w przypadku istnienia dopasowania na pozycji $j - i + s$, gdzie $s < per(pat[1 : i-1]text[j])$, prefiks długości s byłby krótszym okresem, co prowadzi do sprzeczności.

Zauważmy, że jeśli wzorzec występuje na pozycji pos w tekście, to pętla z linii 19. zatrzyma się po dopasowaniu $len(pat)$ znaków, a następnie odpowiednia pozycja zostanie zwrócona w linii 23.

Należy jeszcze zwrócić uwagę na to, że nie wykona się zbyt dużego przesunięcia (czyli nie większego niż wskazana powyżej długość odpowiedniego okresu). Wynika to z tego, że przeskoki wykonywane w liniach 28. i 35. są bezpośrednimi odpowiednikami wniosków z 1. twierdzenia.

2.5.3 Złożoność

Autor oryginalnej pracy wskazał ograniczenie górne na liczbę porównań znaków wynoszące $6 \cdot len(text) + 5$, którego dowód jest oparty na dwóch obserwacjach [7]. Po pierwsze, porównania w linii 27. (których jest i) zawsze prowadzą do zwiększenia pos o więcej niż i (a pos nie jest nigdy zmniejszany), więc łączna ich liczba jest ograniczona przez $len(text)$. Po drugie, pozostałe porównania znaków zawsze prowadzą do zwiększenia wartości wyrażenia $5 \cdot pos + m + i + j + k$, którego początkowa wartość wynosi 3, a końcowa jest ograniczona przez $5 \cdot len(text) + 8$. Łącznie więc liczba porównań jest ograniczona przez $6 \cdot len(text) + 5$.

2.6 Algorytm Galila i Seiferasa (GS)

Z. Galil i J. Seiferas zaproponowali algorytm korzystający z pewnego podziału wzorca na dwa podsłowa wykonywanego podczas preprocessingu [4]. Podczas właściwego wyszukiwania, najpierw porównuje się z tekstem jedno z nich, a dopiero w przypadku całkowitej zgodności – drugie. Algorytm ten wykonuje w pesymistycznym przypadku liniową liczbę porównań względem długości tekstu oraz w głównej fazie działania wymaga jedynie stałej dodatkowej pamięci.

2.6.1 Opis algorytmu

Zdefiniujmy funkcję $reach(w,p) = max\{q \leq len(w)\colon w[1:p] \text{ jest okresem } w[1:q]\}$. Dla pewnej ustalonej liczby naturalnej $k > 1$, którą w oryginalnej pracy zaproponowano jako 4 (udowodniono, że w tym algorytmie możliwe jest użycie $k \geq 3$ [9]).

Definicja 2.6.1 (Prefiks okresowy)**.** Niech prefiks okresowy słowa w to taki $w[1:p]$, że jest on słowem podstawowym i $reach(w,p) \geq k \cdot p$.

Wykorzystywany podział wzorca na dwa podsłowa pat_l i pat_r spełnia: $pat = pat_l pat_r$, pat_r ma co najwyżej jeden prefiks okresowy oraz $len(pat_l) \leq 2 \cdot per(pat_r)$. Podział taki zawsze istnieje i można go obliczyć w $O(len(pat))$. W dalszej części zakładamy, że mamy dostęp do obliczonych wcześniej wartości $s = len(pat_l) + 1$ oraz jeśli pat_r miało prefiks okresowy, to do p_1 będącego długością tego okresu i $q_1 = reach(pat_r, p_1) - p_1$.

Wzorzec jest więc postaci $pat_l pat_r$ oraz pat_r jest postaci $x^r x' a x''$, gdzie x jest słowem podstawowym, $len(x) = p_1$, x' jest prefiksem x, $x'a$ nie jest prefiksem x oraz $len(x^r x') = p_1 + q_1$. Wtedy podczas wyszukiwania pat_r w $text$, jeśli dopasowano $pat[s+1 : s+p_1+q_1]$, to można wykonać przesunięcie długości p_1 i kontynuować porównania odpowiednio ze znakami wzorca od $pat[s+1+q_1]$. W przeciwnym przypadku, jeśli niedopasowanie wystąpi z $pat[s+q]$, gdzie $q \neq p_1 + q_1 + 1$, to można wykonać przesunięcie długości $q/k+1$ i zacząć porównywanie z powrotem od $pat[1]$.

Algorytm w pseudokodzie (z założeniem dostępu do k, s, p_1 i q_1) wygląda następująco:

Algorytm 8 Algorytm GS

```
1:  pos ← 1; i ← 1
2:  while pos ≤ len(text) − len(pat) do
3:      while s + i ≤ len(pat) and pat[s + i] = text[pos + s + i] do
4:          i ← i + 1
5:      end while
6:      if i = len(pat) − s and pat[1 : s] = text[pos : pos + s − 1] then
7:          return pos
8:      end if
9:      if i = p1 + q1 then
10:         pos ← pos + p1
11:         i ← i − p1
12:     else
13:         pos ← pos + i/k + 1
14:         i ← 1
15:     end if
16: end while
```

2.6.2 Poprawność

Nietrywialną częścią algorytmu jest wyszukiwanie wystąpień pat_r, gdyż wystąpienie pat_l przed znalezionym pat_r algorytm weryfikuje w sposób naiwny (linia 6.). Pokażmy teraz, że algorytm nie pominie żadnego wystąpienia pat_r. Zacznijmy od wprowadzenia funkcji $shift(w, q)$ dla słowa w i jego indeksu q. Niech $shift(w, q) = min\{h > 0 \colon w[h + 1 : q - 1] = w[1 : q - h - 1]\}$. Łatwo zauważyć, że w przypadku próby dopasowania wzorca do $text[i : i + len(pat)]$ i wystąpienia nierówności j-tego znaku wzorca z $i + j$-tym znakiem tekstu, poprawne dopasowanie nie może wystąpić wcześniej niż dla $text[i + shift(pat, j) : i + shift(pat, j) + len(pat)]$, więc przesunięcie wzorca względem tekstu o mniej niż odpowiednia wartość funkcji $shift$ nie może prowadzić do pominięcia poprawnego dopasowania. Wprowadźmy teraz dwa lematy:

1. Jeśli $shift(w, q) \leq q/k$, to $w[1 : shift(w, q)]$ jest prefiksem okresowym w.
2. Jeśli $1 : h$ jest prefiksem okresowym w, to
$h = shift(w, q) \leq q/k \leftrightarrow k \cdot h \leq q \leq reach(w, h)$.

Zauważmy, że jeśli pat_r nie ma prefiksu okresowego, to z lematu 1. wynika, że $shift(pat_r, i) > i/k$, więc przesunięcie wzorca o $i/k + 1$ nie prowadzi do pominięcia dopasowania (jest to realizowane w linii 13.). Jeśli pat_r ma jednak jeden prefiks okresowy (o długości p_1) to z 1. i 2. lematu wynika, że $shift(pat_r, i)$ może być mniejsze od i/k tylko dla $k \cdot p_1 \leq i \leq reach(pat, p_1)$ i można wtedy przesunąć wzorzec o p_1 (linia 10.). Dla innych i można przesunąć wzorzec o $i/k + 1$ (linia 13.).

2.6.3 Złożoność

Zauważmy, że porównania wykonuje się jedynie w liniach 3. i 6., gdzie w linii 3. porównuje się znaki tekstu z pat_r, a w 6. z pat_l. Liczba porównań wykonywana w linii 3. jest ograniczona przez $k \cdot len(text)$ (z tego powodu wybranie mniejszego k ma znaczenie praktyczne), czego dowód opiera się na obserwacji, że każde wykonane w tej linii porównanie prowadzi do zwiększenia wyrażenia $k \cdot pos + i$. Liczba porównań wykonywanych w linii 6. jest z kolei ograniczona przez $2 \cdot len(text)$. Wynika to z tego, iż odległość pomiędzy kolejnymi wystąpieniami pat_r wynosi co najmniej $per(pat_r)$, a długość $pat_l \leq 2 \cdot per(pat_r)$, więc każdy znak tekstu zostanie porównany ze znakiem z pat_l co najwyżej 2 razy. Łącznie więc liczba porównań znaków jest ograniczona przez $(k + 2) \cdot len(text)$, czyli dla minimalnego możliwego k równego 3 jest to $5 \cdot len(text)$ [9].

2.7 Algorytm Two-Way (TW)

M. Crochemore i D. Perrin zaproponowali algorytm, będący kompromisem pomiędzy podejściem algorytmu KMP (dopasowywaniem wzorca od lewej strony) i algorytmu BM (dopasowywaniem wzorca od prawej strony) [6]. Na początku dzieli on wzorzec na pewne dwie części, a następnie sprawdza zgodność z tekstem, porównując jedną z nich od lewej strony, a drugą od prawej. Algorytm ten wykonuje w pesymistycznym przypadku liniową liczbę porównań względem długości tekstu oraz w głównej fazie działania wymaga jedynie stałej dodatkowej pamięci.

2.7.1 Opis algorytmu

Definicja 2.7.1 (Lokalny okres słowa)**.** Lokalnym okresem słowa w na pozycji l określamy najmniejszą taką liczbę r, że $w[i] = w[i + r]$ dla w takich, że $l - r + 1 \leq i \leq l$. Będziemy oznaczać go przez przez $lper(w, l)$.

Algorytm korzysta z podziału wzorca na dwa podsłowa pat_l i pat_r takie, że $pat = pat_l pat_r$, $len(pat_l) < per(pat)$ oraz długość lokalnego okresu wzorca na pozycji $l = len(pat_l)$ jest równa długości okresu wzorca, czyli $lper(pat, l) = per(pat)$. Podział taki zawsze istnieje i można go obliczyć w $O(len(pat))$. W dalszej części zakładamy, że mamy dostęp do obliczonych wcześniej wartości l i $per(pat)$.

Algorytm ten dopasowuje znaki wzorca do odpowiednich pozycji tekstu, porównując je w obu kierunkach. Początkowa pozycja porównań jest wyznaczona przez $l = len(pat_l)$. W celu weryfikacji, czy na danej pozycji w tekście występuje wzorzec, algorytm wykonuje dwie fazy dopasowania.

W pierwszej porównuje, iterując się od lewej do prawej, pat_r do odpowiednich pozycji w tekście. Jeśli wystąpiło niedopasowanie podczas tej fazy, to algorytm nie wykonuje drugiej fazy, a wzorzec jest przesunięty w prawo w taki sposób, żeby l-ty element wzorca odpowiadał kolejnemu znakowi tekstu, niż ten, na którym wystąpiła niezgodność.

Jeśli w pierwszej fazie wszystkie porównania były zgodne, to algorytm przechodzi do drugiej fazy. W tej fazie porównywane jest pat_l z tekstem. W tej fazie algorytm iteruje się od prawej do lewej. Jeśli całość została dopasowana, to algorytm odpowiednio zwraca informację o dopasowaniu wzorca na danej pozycji tekstu. Z kolei w przypadku wystąpienia niedopasowania algorytm przesuwa wzorzec w prawo o $per(pat)$ pozycji. Po takim przesunięciu wiemy, że pewien prefiks wzorca odpowiada danym znakom tekstu. Długość tego prefiksu jest zapamiętywana, żeby podczas kolejnego dopasowywania ograniczyć liczbę porównań.
Algorytm w pseudokodzie (z założeniem dostępu do l i $per(pat)$) wygląda następująco:

Algorytm 9 Algorytm Two-Way

```
1:  pos ← 0; s ← 0
2:  while pos + len(pat) ≤ len(text) do
3:      i ← max(l, s) + 1
4:      while i ≤ len(pat) and pat[i] = text[pos + i] do
5:          i ← i + 1
6:      end while
7:      if i ≤ len(pat) then
8:          pos ← pos + max(i − l, s − per(pat) + 1)
9:          s ← 0
10:     else
11:         j ← l
12:         while j > s and pat[j] = text[pos + j] do
13:             j ← j − 1
14:         end while
15:         if j ≤ s then
16:             return pos
17:         end if
18:         pos ← pos + per(pat)
19:         s ← len(pat) − per(pat)
20:     end if
21: end while
```

2.7.2 Poprawność

Zauważmy, że w przypadku dopasowania wzorca, wartość *pos* zwiększa się o długość okresu wzorca (linia 18.), więc algorytm nie pominie wtedy poprawnego dopasowania. W przypadku wystąpienia niezgodności przedstawiony w oryginalnej pracy dowód poprawności algorytmu jest oparty na wykazaniu nie wprost, że w przypadku istnienia poprawnego dopasowania zaczynającego się pomiędzy kolejnymi wartościami pos – pos_i i pos_{i+1}, dopasowanie to byłoby przesunięte o wielokrotność okresu wzorca względem pos_i i powtórzyłoby tę samą niezgodność [6].

2.7.3 Złożoność

Autorzy oryginalnej pracy wskazali ograniczenie górne na liczbę porównań znaków w głównej części algorytmu wynoszące $2 \cdot len(text)$ [6]. Zauważmy, że porównania znaków wykonuje się jedynie w 4. i 12. linii algorytmu. Wskażmy więc osobno ograniczenia liczby porównań wykonywanych w każdej z tych linii podczas całego działania algorytmu.

Pokażmy, że każde porównanie wykonane w linii 3. silnie zwiększa wartość wyrażenia $pos+i$ przed wystąpieniem kolejnego porównania w tej linii. Jest to trywialne, gdy $pat[i] = text[pos + i]$ i $i < len(pat)$, gdyż wtedy i jest zwiększona w 5. linii, a pos pozostaje bez zmian. Jeśli $pat[i] = text[pos + i]$ i $i = len(pat)$, czyli nie wystąpiła niezgodność przy porównywaniu pat_r, to analogicznie i jest zwiększona o 1 w 5. linii. Następnie pos jest zwiększona o $per(pat)$ w linii 18. i zmniejszona o co najwyżej $per(pat)$ w linii 3. Łącznie

więc rozważane wyrażenie się zwiększyło. Jeśli $pat[i] \neq text[pos + i]$ dla pewnego $i = i'$, to pos jest zwiększona o co najmniej $i' - l$ w 8. linii, a następnie i jest zmniejszona o co najwyżej $i' - (l + 1)$. W tym wypadku także $pos + i$ się zwiększyło. Ponieważ $pos + i$ ma na początku wartość $l + 1$, a na końcu co najwyżej $len(text)$, więc łącznie w 3. linii zostanie wykonanych co najwyżej $len(text)$ porównań.

Zliczmy teraz porównania w linii 12. Zauważmy, że pomiędzy kolejnymi wejściami do pętli *while* w 12. linii zmienna pos zostaje zwiększona o $per(pat)$ w 18. linii. Ponieważ z założeń $l < per(pat)$, $s \geq 0$, a wartość zmiennej pos nie jest zmniejszana podczas działania algorytmu, więc porównania w 12. linii będą wykonywane dla parami różnych wartości $pos + j$. Wynika z tego, że każde porównanie będzie dotyczyło innego znaku tekstu, więc łącznie takich porównań będzie co najwyżej $len(text)$.

Łącznie więc podczas całego działania algorytmu liczba porównań jest ograniczona przez $2 \cdot len(text)$.

2.8 Podsumowanie

Poniżej przedstawiono tabelę podsumowującą najważniejsze cechy opisanych algorytmów.

Algorytm	Kierunek porównań	Pesymistyczna liczba porównań w głównej części	Pesymistyczna liczba porównań w preprocessingu	Ilość wymaganej dodatkowej pamięci
BF	w prawo	$len(text) \cdot len(pat)$	—	$O(1)$
KMP	w prawo	$2 \cdot len(text)$	$3 \cdot len(pat)$	$O(len(pat))$
MP	w prawo	$2 \cdot len(text)$	$2 \cdot len(pat)$	$O(len(pat))$
BM	w lewo	$len(text) \cdot len(pat)$	$O(len(pat))$	$O(len(pat))$
BMB	w lewo	$len(text) \cdot len(pat)$	$O(len(pat))$	$O(len(pat))$
AG	w lewo	$\frac{3}{2} \cdot len(text)$	$O(len(pat))$	$O(len(pat))$
AGB	w lewo	$\frac{3}{2} \cdot len(text)$	$O(len(pat))$	$O(len(pat))$
Cr	w prawo	$6 \cdot len(text)$	—	$O(1)$
GS	w prawo	$5 \cdot len(text)$	$O(len(pat))$	$O(1)$
TW	inny	$2 \cdot len(text)$	$O(len(pat))$	$O(1)$

Rozdział 3

Pomiary liczby porównań

Implementacje algorytmów wykorzystane do pomiarów zaprezentowanych w poniższym rozdziale pochodzą z repozytorium:
`https://github.com/krzysztof-turowski/string-algorithms`.
W repozytorium tym (w folderze *benchar* - pull request: `https://github.com/krzysztof-turowski/string-algorithms/pull/46`) znajduje się także kod źródłowy autorskiej biblioteczki służącej do pomiaru liczby porównań.

3.1 Metodologia

Zacznijmy od opisu sposobu pomiarów liczby porównań wykonywanej przez poszczególne algorytmy. W celu przeprowadzenia pomiarów stworzono klasę *benchar*. Poniżej przedstawiono związane z nią obiekty:

- Obiekt klasy *benchar*:
  - działa jako fabryka obiektów *count_str*
  - posiada pole *cmp_count*, w którym znajduje się liczba porównań wykonanych przez obiekty *count_str* będące jego potomkami (tj. wytworzone przez dany obiekt klasy *benchar* lub przez jego innych potomków)
- Obiekt klasy *count_str*:
  - dziedziczy metody klasy *str* (tj. standardowego łańcucha znaków w języku Python)
  - posiada referencję do rodzica (tj. obiektu *benchar*, który go stworzył lub był rodzicem obiektu *count_str* który go stworzył)
  - każda operacja porównująca znaki (m.in.: ==, ! =, >, <) odpowiednio zwiększa wartość pola *cmp_count* w rodzicu
  - wszystkie operacje tworzące nowe łańcuchy znaków (m.in.: +, [:]) tworzą nowy obiekt *count_str*, który dziedziczy rodzica.

Przedstawione powyżej działanie klasy *benchar* pozwala na skuteczne zliczanie porównań dla wykorzystanych implementacji algorytmów. Wynika to z tego, że wszystkie użyte implementacje działają, przyjmując tekst i wzorzec będące łańcuchami znaków (podczas pomiarów są to obiekty *count_str* wytworzone przez ten sam obiekt *benchar*) i podczas działania nie tworzą nowych łańcuchów znaków w inny sposób niż operacjami na samych łańcuchach (m.in.: +, [:]). Dzięki temu jest zagwarantowane, że wszystkie łańcuchy znaków wykorzystywane w danym uruchomieniu algorytmu będą odpowiednimi obiektami *count_str* i wszystkie wykonane porównania zostaną zliczone.

3.1.1 Porównywane algorytmy

W pomiarach liczby porównań znaków zostały sprawdzone następujące algorytmy:

- BF – algorytm naiwny
- MP – algorytm Morissa i Pratta, czyli KMP wykorzystujący tablicę słabych prefikso-sufiksów
- KMP – algorytm Knutha, Morissa i Pratta, czyli KMP wykorzystujący tablicę silnych prefikso-sufiksów
- BMB – algorytm Boyera i Moore'a korzystający jedynie z tablicy *gss* (bez *bcs*)
- BM – algorytm Boyera i Moore'a korzystający z tablic *gss* i *bcs*
- AGB – algorytm Apostolico i Giancarlo korzystający jedynie z tablicy *gss* (bez *bcs*)
- AG – algorytm Apostolico i Giancarlo korzystający z tablic *gss* i *bcs*
- Cr – algorytm Crochemore'a dla uporządkowanych alfabetów
- GS – algorytm Galila i Seiferasa
- TW – algorytm Two-Way.

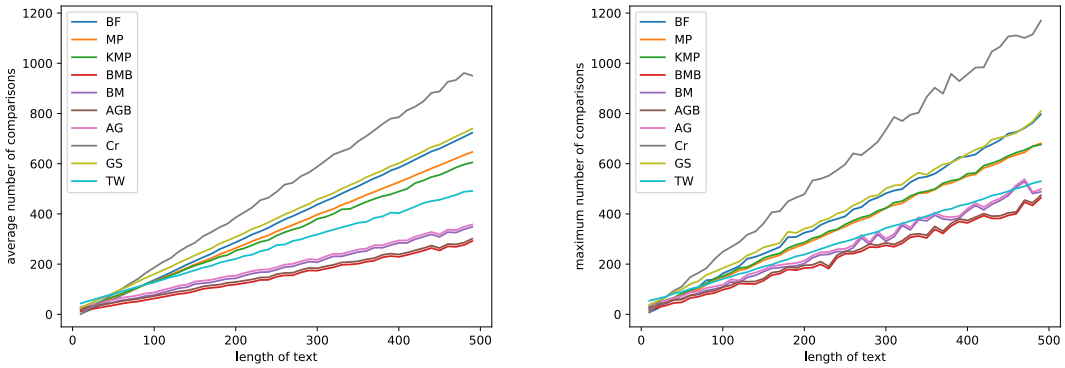
3.2 Przeprowadzone testy

W poniższym podrozdziale przedstawiono w formie wykresów wyniki pomiarów liczby porównań znaków wykonywanych przez dane algorytmy. Wykorzystane podczas pomiarów sposoby wyboru tekstu i wzorca można podzielić na dwie grupy. Pierwszą stanowią metody randomizowane (sekcje: rozkład jednostajny, rozkład geometryczny, język naturalny), dla których wspólną cechą w metodyce pomiarów było generowanie dla danych parametrów 100 par tekst-wzorzec i przedstawianie na wykresach średniej oraz maksymalnej liczby wykonanych porównań w tej próbce. Drugą grupę stanowią trudne przypadki świadczące o pesymistycznej liczbie porównań wykonywanej przez część algorytmów (sekcje: pesymistyczny przypadek dla algorytmu naiwnego, pesymistyczny przypadek dla algorytmu BM,

pesymistyczny przypadek dla algorytmu AG). W ich przypadkach dla danych parametrów wykonano (i zaprezentowano na wykresach) jeden pomiar, gdyż są to jednoznacznie ustalone wzorce i słowa.

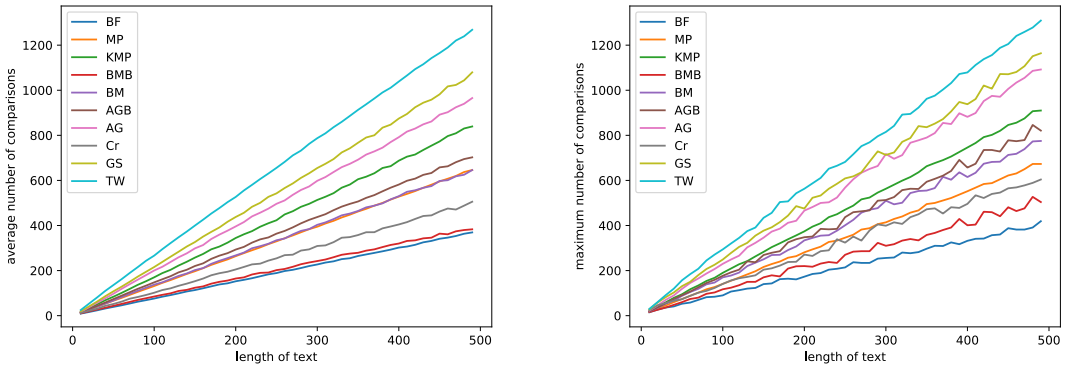
3.2.1 Rozkład jednostajny

W tej sekcji przedstawiono wyniki pomiarów dla tekstów i wzorców generowanych zgodnie z rozkładem jednostajnym nad ustalonym alfabetem (każda litera ma prawdopodobieństwo równe $\frac{1}{|\mathcal{A}|}$ wystąpienia na danym indeksie tekstu oraz wzorca).



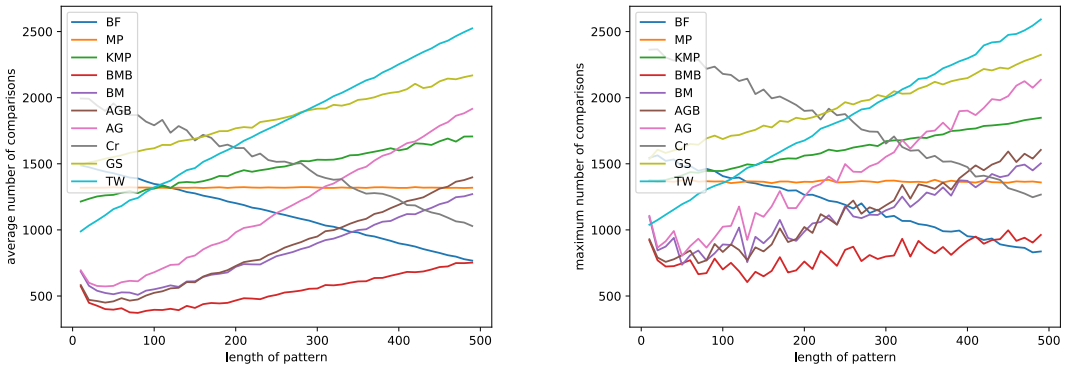
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.1: Liczba porównań w zależności od $len(text)$ dla $|\mathcal{A}| = 3$ i $len(pat) = 10$



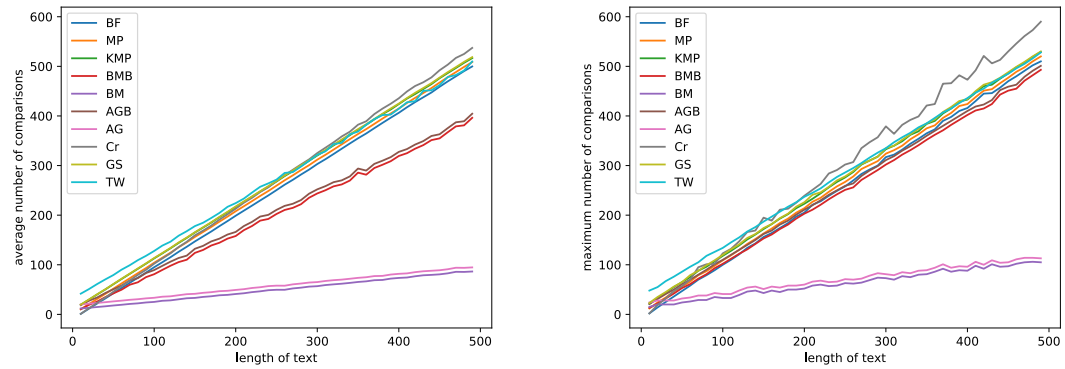
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.2: Liczba porównań w zależności od $len(text)$ dla $|\mathcal{A}| = 3$ i $len(pat) = \frac{len(text)}{2}$



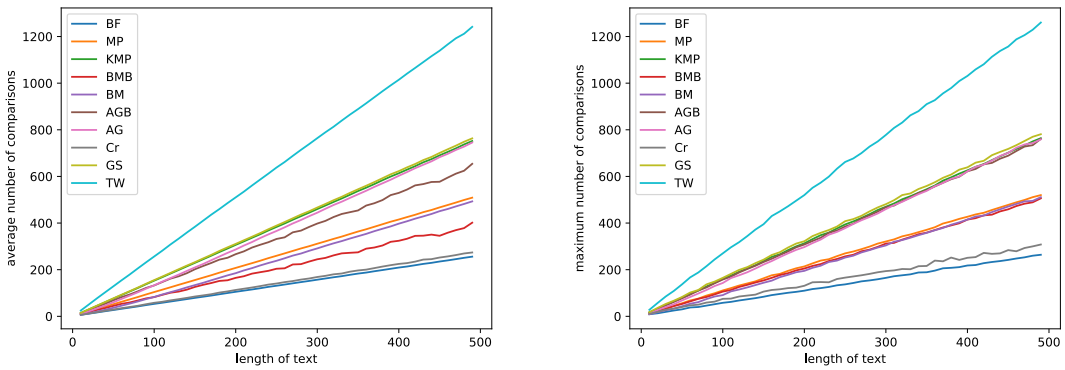
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.3: Liczba porównań w zależności od $len(pat)$ dla $|\mathcal{A}| = 3$ i $len(text) = 1000$



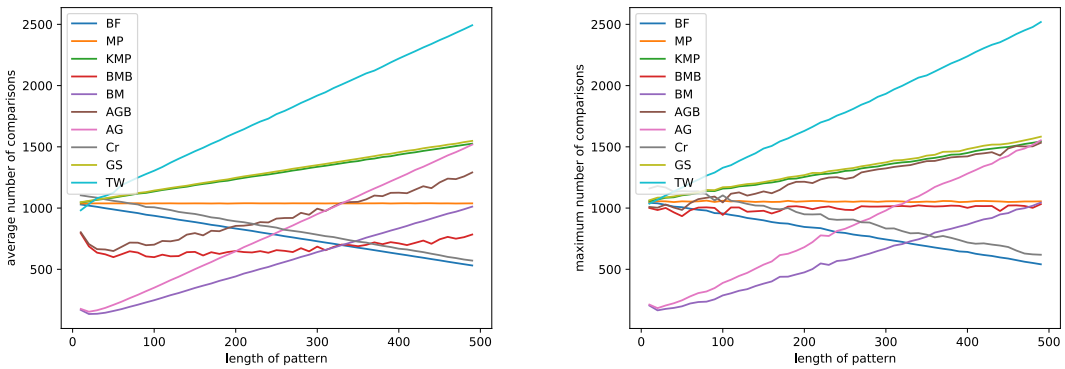
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.4: Liczba porównań w zależności od $len(text)$ dla $|\mathcal{A}| = 26$ i $len(pat) = 10$



(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.5: Liczba porównań w zależności od $len(text)$ dla $|\mathcal{A}| = 26$ i $len(pat) = \frac{len(text)}{2}$

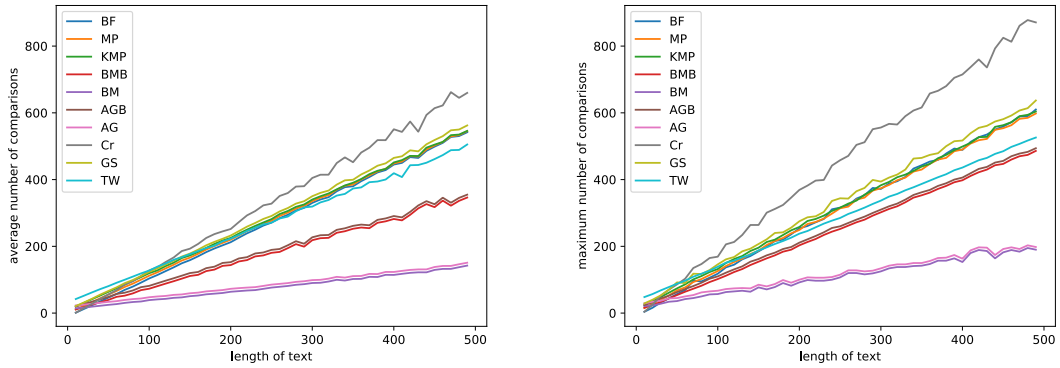


(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.6: Liczba porównań w zależności od $len(pat)$ dla $|\mathcal{A}| = 26$ i $len(text) = 1000$

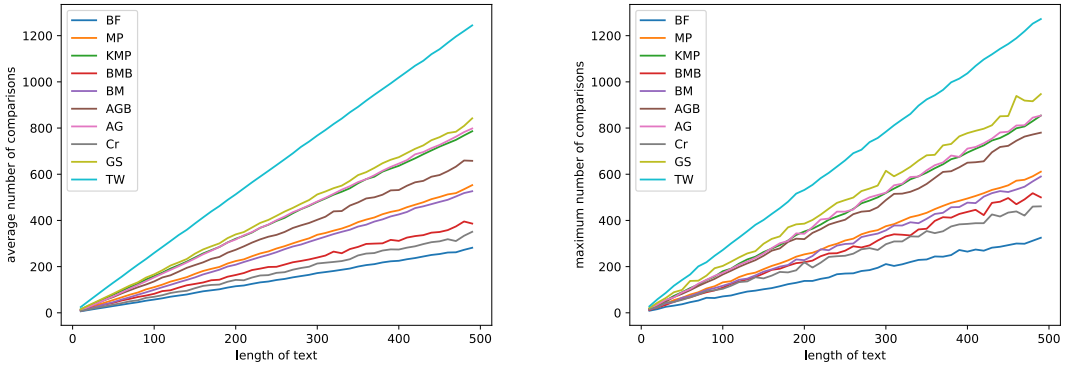
3.2.2 Rozkład geometryczny

W tej sekcji przedstawiono wyniki pomiarów dla tekstów i wzorców generowanych zgodnie z rozkładem geometrycznym z ustalonym parametrem p (i-ta litera alfabetu ma prawdopodobieństwo równe $(1-p)^{i-1}p$ wystąpienia na danym indeksie tekstu oraz wzorca).



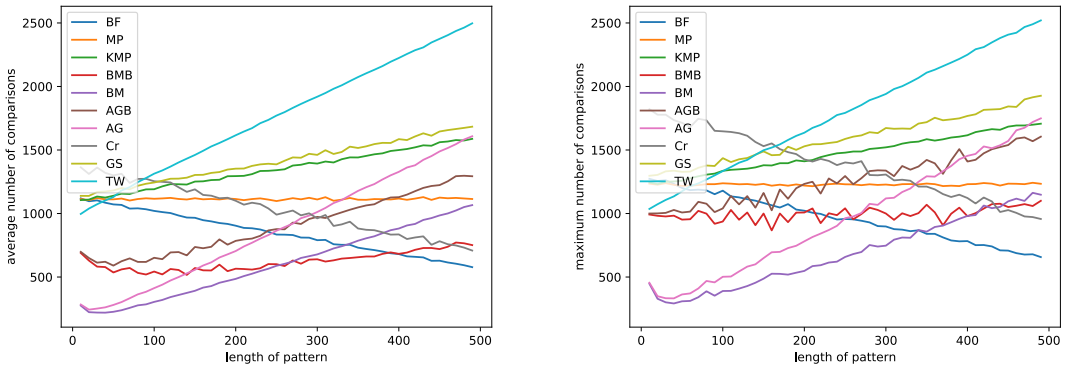
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.7: Liczba porównań w zależności od $len(text)$ dla $p = 0.2$ i $len(pat) = 10$



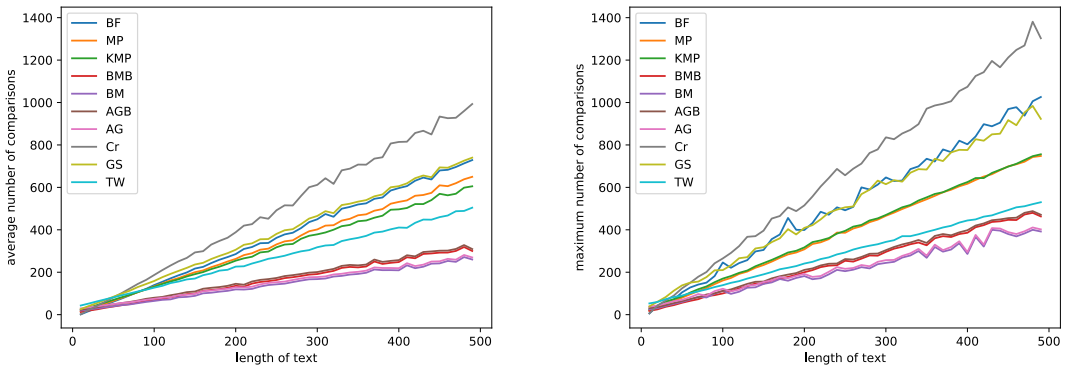
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.8: Liczba porównań w zależności od $len(text)$ dla $p = 0.2$ i $len(pat) = \frac{len(text)}{2}$



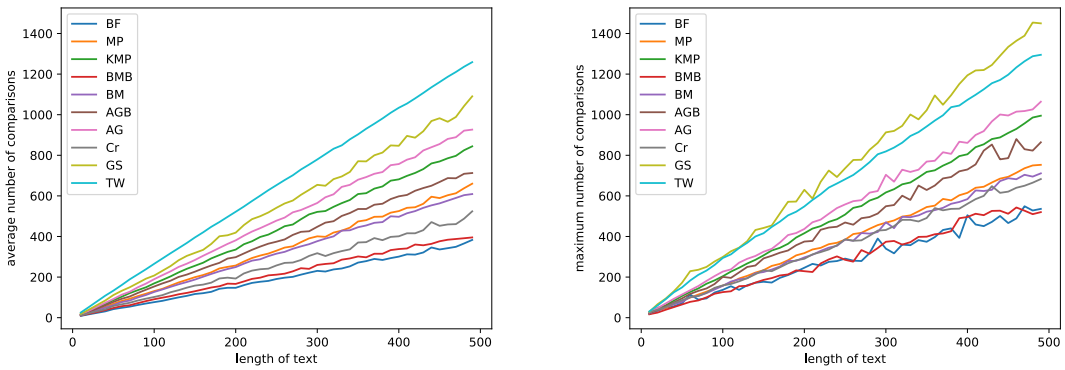
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.9: Liczba porównań w zależności od $len(pat)$ dla $p = 0.2$ i $len(text) = 1000$



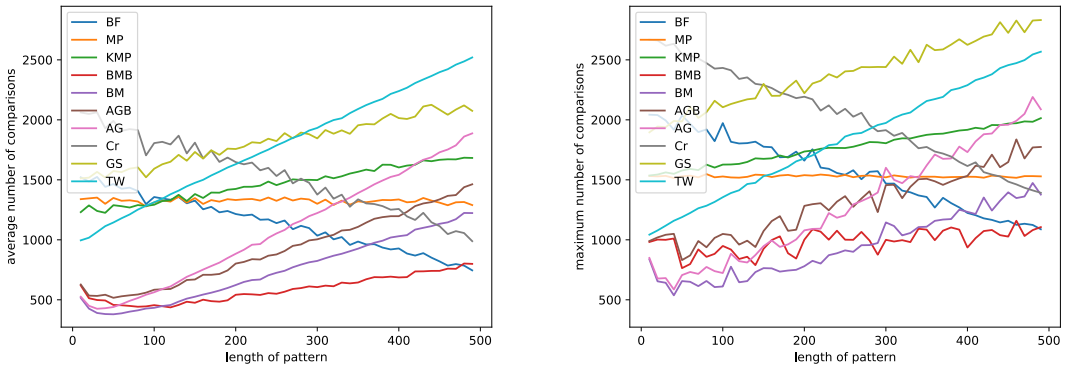
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.10: Liczba porównań w zależności od $len(text)$ dla $p = 0.5$ i $len(pat) = 10$



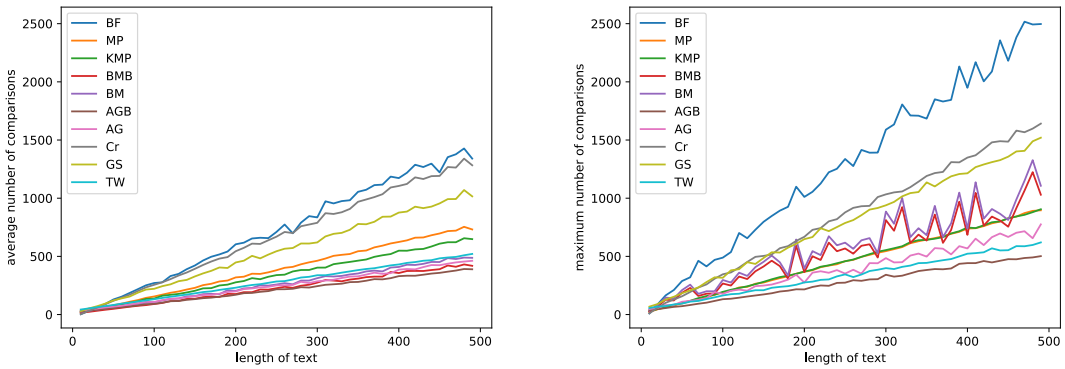
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.11: Liczba porównań w zależności od $len(text)$ dla $p = 0.5$ i $len(pat) = \frac{len(text)}{2}$



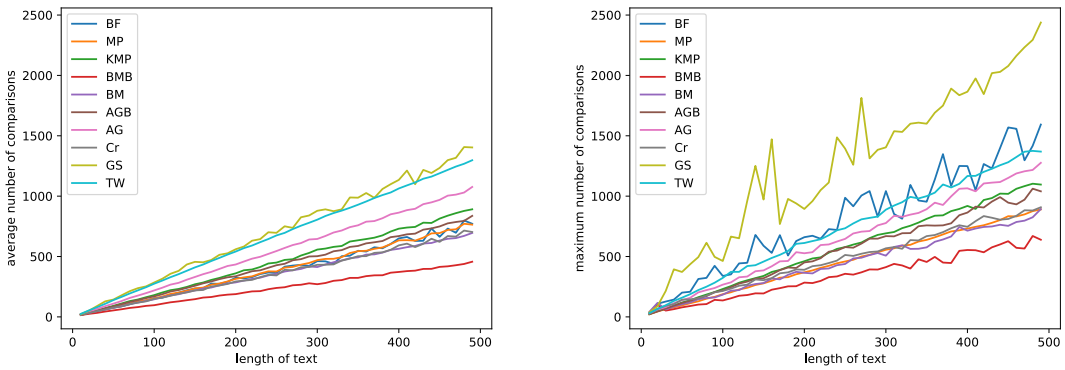
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.12: Liczba porównań w zależności od $len(pat)$ dla $p = 0.5$ i $len(text) = 1000$



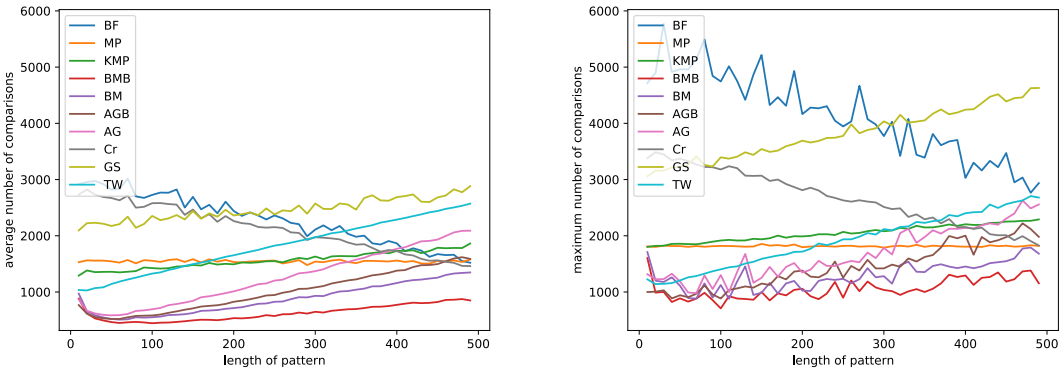
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.13: Liczba porównań w zależności od $len(text)$ dla $p = 0.8$ i $len(pat) = 10$



(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.14: Liczba porównań w zależności od $len(text)$ dla $p = 0.8$ i $len(pat) = \frac{len(text)}{2}$

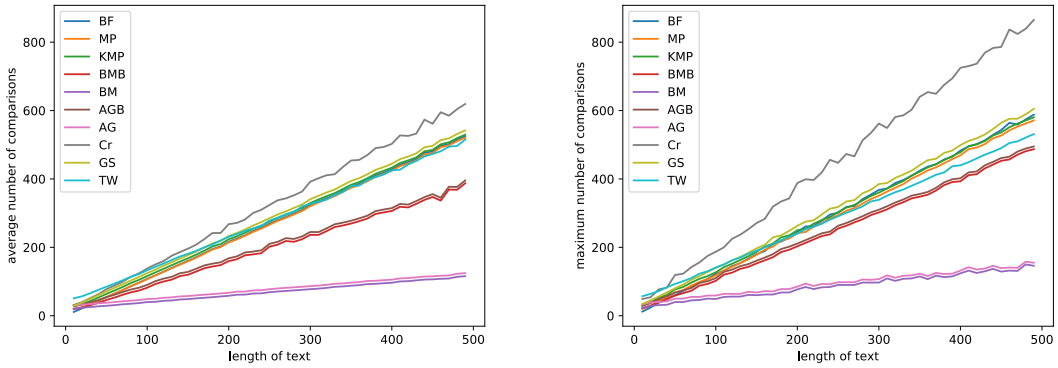


(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.15: Liczba porównań w zależności od $len(pat)$ dla $p = 0.8$ i $len(text) = 1000$

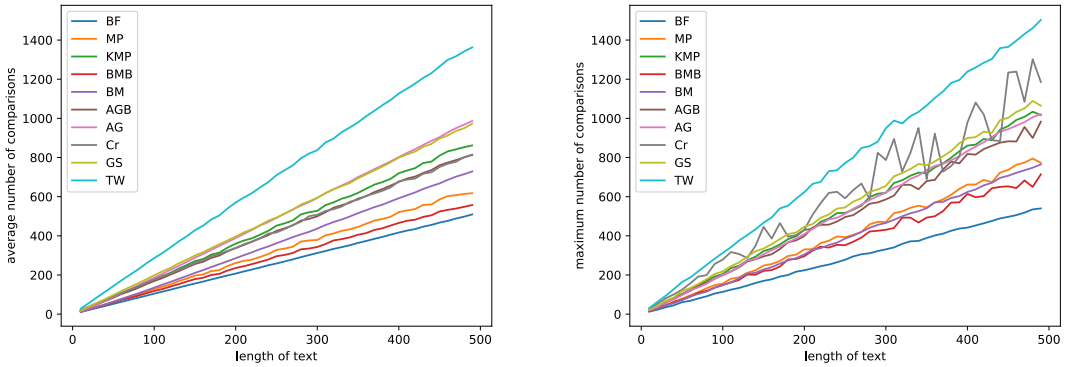
3.2.3 Język naturalny

W tej sekcji przedstawiono wyniki pomiarów dla tekstów i wzorców wybieranych losowo z pewnego tekstu języka naturalnego (Adam Mickiewicz: "Pan Tadeusz", źródło: `https://gist.github.com/tomekziel/47b7c3dffb17adbc70c42b9be8dc93a7`), który został znormalizowany poprzez zamianę wszystkich liter na małe, zamianę wszystkich znaków białych na spacje i usunięcie znaków niebędących znakami alfanumerycznymi lub spacjami.



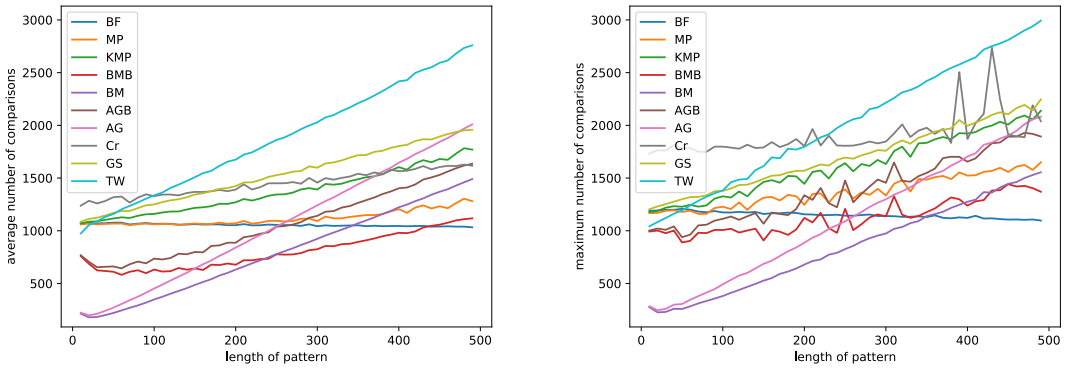
(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.16: Liczba porównań w zależności od $len(text)$ dla $len(pat) = 10$



(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.17: Liczba porównań w zależności od $len(text)$ dla $len(pat) = \frac{len(text)}{2}$

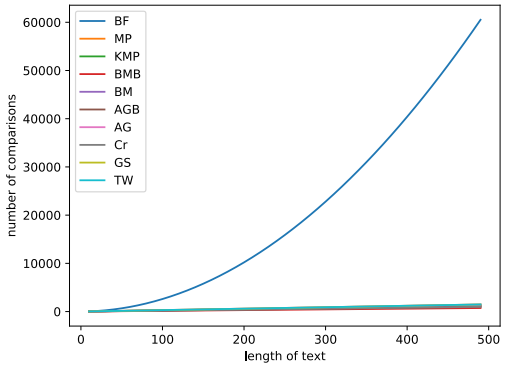


(a) średnia liczba porównań (b) maksymalna liczba porównań

Rysunek 3.18: Liczba porównań w zależności od $len(pat)$ dla $len(text) = 1000$

3.2.4 Pesymistyczny przypadek dla algorytmu naiwnego

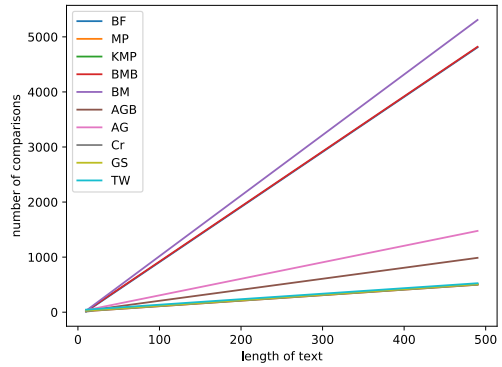
W poniższej sekcji przedstawiono wyniki dla tekstów i wzorców będących przypadkiem pesymistycznym dla algorytmu naiwnego, czyli $text = a^e b$ i $pat = a^{\frac{e}{2}} b$, gdzie a i b są różnymi literami.



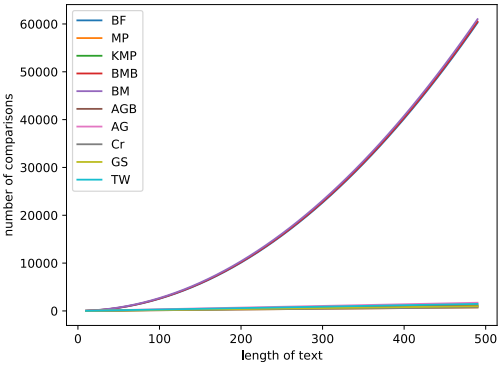
Rysunek 3.19: Liczba porównań w zależności od $len(text)$

3.2.5 Pesymistyczny przypadek dla algorytmu BM

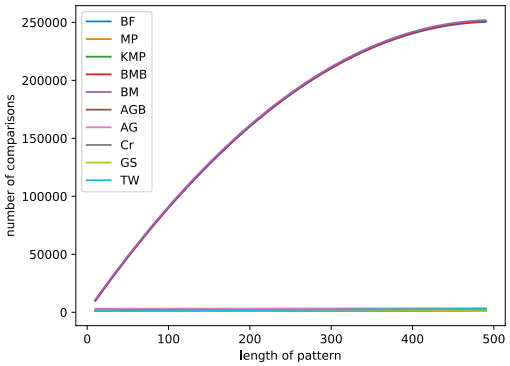
W poniższej sekcji przedstawiono wyniki dla tekstów i wzorców będących przypadkiem pesymistycznym dla algorytmu BM, czyli $text = a^n$ i $pat = a^m$, gdzie a jest literą.



Rysunek 3.20: Liczba porównań w zależności od $len(text)$ dla $len(pat) = 10$



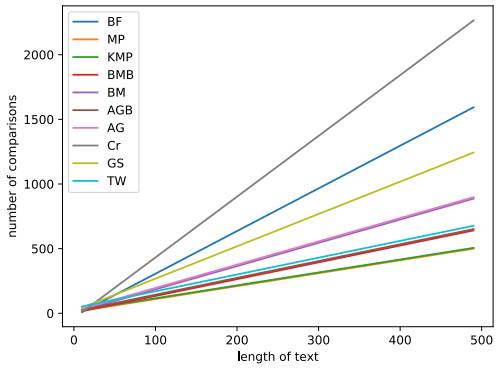
Rysunek 3.21: Liczba porównań w zależności od $len(text)$ dla $len(pat) = \frac{len(text)}{2}$



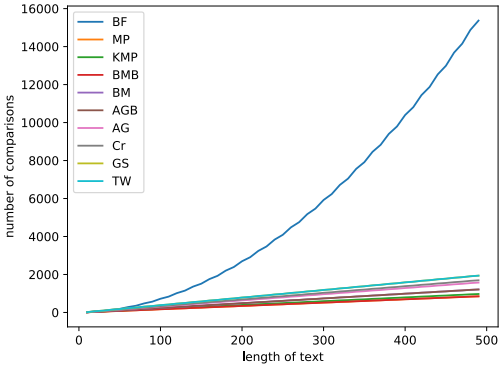
Rysunek 3.22: Liczba porównań w zależności od $len(pat)$ dla $len(text) = 1000$

3.2.6 Pesymistyczny przypadek dla algorytmu AG

W poniższej sekcji przedstawiono wyniki dla tekstów i wzorców będących przypadkiem pesymistycznym dla algorytmu AG, czyli $text = pat^e$ i $pat = a^{m-1}ba^mb$, gdzie a i b są różnymi literami.



Rysunek 3.23: Liczba porównań w zależności od $len(text)$ dla $len(pat) = 10$



Rysunek 3.24: Liczba porównań w zależności od $len(text)$ dla $len(pat) = \frac{len(text)}{2}$

3.3 Wnioski

Na podstawie powyższych pomiarów można stwierdzić, że **w celu wyboru najwydajniejszego algorytmu dla danego zastosowania powinno się wziąć pod uwagę przede wszystkim rozmiar alfabetu i długość wzorca względem długości tekstu**.

Można zauważyć jednak, że **w większości przypadków najmniejszą liczbę porównań (do uzyskania której dążymy) wykona jedna z wersji algorytmu BM** – często liczba porównań będzie nawet mniejsza niż długość tekstu. **W przypadku małego rozmiaru alfabetu lepiej sprawdza się wersja BMB** – pomijająca przesunięcie nieodpowiedniego znaku (rys. 3.1–3.3 i rys. 3.13–3.15). W takich przypadkach korzyść z użycia tej tablicy jest znikoma, a obliczenie jej jest pewnym kosztem.

Sytuacja zmienia się znacznie w przypadku dużych alfabetów – w szczególności złożonego z wszystkich liter alfabetu łacińskiego. W takich przypadkach przesunięcia dobrego sufiksu są dominowane przez przesunięcia nieodpowiedniego znaku. Różnica w liczbie porównań jest tym większa, im krótszy jest wzorzec, gdyż koszt dodatkowego preprocessingu jest wtedy odpowiednio mniejszy (rys. 3.4–3.7 i rys. 3.10). **Dla dużych alfabetów wydajniejsza jest więc pełna wersja algorytmu BM**.

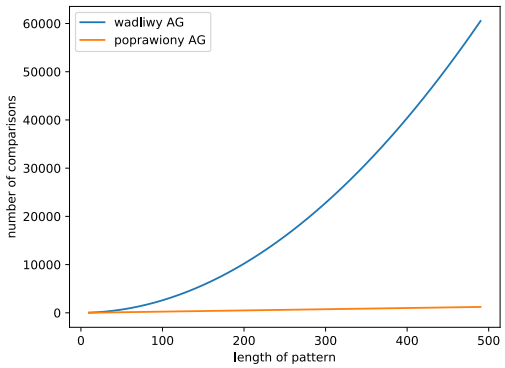
Warto zauważyć, że spośród przeprowadzonych testów **tylko w jednym przypadku algorytm AG wykonał mniej porównań niż algorytm BM (i odpowiednio AGB niż BMB) - w przypadku pesymistycznym dla algorytmu BM** (rys. 3.20–3.22). Warto jednak zauważyć, że różnice w wydajności dla pozostałych przypadków nie były zwykle znaczne, więc warto rozważyć, czy dla danego zastosowania ważniejszy jest średni zysk wydajności, czy gwarancja liniowego ograniczenia liczby wykonanych porównań.

Na koniec zwróćmy uwagę na jeszcze jeden ciekawy wynik. **Dla odpowiednio długich wzorców, w przypadku wyszukiwania w tekście naturalnym, najlepiej sprawdzał się algorytm naiwny** (rys. 3.17). Jest to zapewne spowodowane tym, że pesymistyczne przypadki dla algorytmu naiwnego (czyli długie słowa mające krótki najkrótszy cykl) rzadko występują w języku naturalnym. Algorytm naiwny zyskuje przewagę, gdyż nie wykonuje preprocessingu, podczas którego liczba wykonanych porównań jest uzależniona od długości wzorca.

Dodatek A

Wadliwa implementacja algorytmu AG

Podczas analizy algorytmów znaleziono błąd w implementacji algorytmu AG zawartej w użytym repozytorium. Implementacja ta w swojej części będącej odpowiednikiem funkcji Q sprawdzała równość pewnego podsłowa wzorca z pewnym jego sufiksem w sposób naiwny – a więc w pesymistycznym przypadku w czasie liniowym od długości wzorca. Powodowało to utratę zalety algorytmu AG względem algorytmu BM – czyli gwarancji liczby porównań rzędu $O(len(text)+len(pat))$. Implementację poprawiono kosztem dodatkowego preprocessingu w czasie $O(len(pat))$. Zyskano natomiast to, że powyższe zapytania są wykonywane w czasie stałym. Poniżej przedstawiono różnicę w wydajności dla $text = a^e$ i $pat = a^{\frac{e}{2}}$.



Rysunek A.1: Liczba porównań w zależności od $len(text)$

Bibliografia

[1] R. S. Boyer i J. S. Moore, „A Fast String Searching Algorithm," *Commun. ACM*, t. 20, nr. 10, s. 762–772, 1977, ISSN: 0001-0782. DOI: 10.1145/359842.359859. adr.: https://doi.org/10.1145/359842.359859.

[2] D. E. Knuth, J. H. Morris Jr. i V. R. Pratt, „Fast Pattern Matching in Strings," *SIAM Journal on Computing*, t. 6, nr. 2, s. 323–350, 1977. DOI: 10.1137/0206024. adr.: https://doi.org/10.1137/0206024.

[3] Z. Galil, „On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm," *Commun. ACM*, t. 22, nr. 9, s. 505–508, 1979, ISSN: 0001-0782. DOI: 10.1145/359146.359148. adr.: https://doi.org/10.1145/359146.359148.

[4] Z. Galil i J. Seiferas, „Time-space-optimal string matching," *Journal of Computer and System Sciences*, t. 26, nr. 3, s. 280–294, 1983, ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(83)90002-8. adr.: https://www.sciencedirect.com/science/article/pii/0022000083900028.

[5] A. Apostolico i R. Giancarlo, „The Boyer–Moore–Galil String Searching Strategies Revisited," *SIAM Journal on Computing*, t. 15, nr. 1, s. 98–105, 1986. DOI: 10.1137/0215007. adr.: https://doi.org/10.1137/0215007.

[6] M. Crochemore i D. Perrin, „Two-way string-matching," *Journal of the ACM (JACM)*, t. 38, s. 650–674, lip. 1991. DOI: 10.1145/116825.116845.

[7] M. Crochemore, „String-matching on ordered alphabets," *Theoretical Computer Science*, t. 92, nr. 1, s. 33–47, 1992, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(92)90134-2. adr.: https://www.sciencedirect.com/science/article/pii/0304397592901342.

[8] R. Cole, „Tight Bounds on the Complexity of the Boyer–Moore String Matching Algorithm," *SIAM Journal on Computing*, t. 23, nr. 5, s. 1075–1091, 1994. DOI: 10.1137/S0097539791195543. adr.: https://doi.org/10.1137/S0097539791195543.

[9] M. Crochemore i W. Rytter, „Squares, cubes, and time-space efficient string searching," *Algorithmica*, t. 13, s. 405–425, maj 1995. DOI: 10.1007/BF01190846.

[10] M. Crochemore i T. Lecroq, „Tight bounds on the complexity of the Apostolico-Giancarlo algorithm," *Information Processing Letters*, t. 63, nr. 4, s. 195–203, 1997, ISSN: 0020-0190. DOI: https://doi.org/10.1016/S0020-0190(97)00107-5. adr.: https://www.sciencedirect.com/science/article/pii/S0020019097001075.

[11] A. Donnelly i T. Deegan, „IP Route Lookups as String Matching," w *Proceedings of the 25th Annual IEEE Conference on Local Computer Networks*, USA: IEEE Computer Society, 2000, s. 589, ISBN: 0769509126.

[12] R.-T. Liu, N.-F. Huang, C.-H. Chen i C.-N. Kao, „A Fast String-Matching Algorithm for Network Processor-Based Intrusion Detection System," t. 3, nr. 3, s. 614–633, 2004, ISSN: 1539-9087. DOI: 10.1145/1015047.1015055. adr.: https://doi.org/10.1145/1015047.1015055.

[13] C. Ryu, T. Lecroq i K. Park, „Fast string matching for DNA sequences," *Theoretical Computer Science*, t. 812, s. 137–148, 2020, In memoriam Danny Breslauer (1968-2017), ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2019.09.031. adr.: https://www.sciencedirect.com/science/article/pii/S0304397519305821.