MINUT 2025 (7), s. 56–67
ISSN 2719-3063

Marta KUCHARCZYK[1], Bożena WIECZOREK[2]

[1]Studentka, Wydział Automatyki, Elektroniki i Informatyki, Politechnika Śląska, ul.Akademicka 16, 44-100 Gliwice

[2]Katedra Algorytmiki i Oprogramowania, Politechnika Śląska, ul. Akademicka 16, 44-100 Gliwice

Efektywne metody wyszukiwania wzorca w tekście: algorytmy Boyera–Moore'a i Horspoola

Streszczenie. W artykule omówiono dwa zaawansowane algorytmy wyszukiwania wzorca w tekście: klasyczny algorytm Boyera-Moore'a (BM) oraz jego uproszczoną wersję zaproponowaną przez Horspoola (BMH). Przedstawiono szczegółowo zasady działania obu algorytmów, ze szczególnym uwzględnieniem heurystyk: złego znaku i dobrego sufiksu. Zaprezentowano implementacje oraz wyniki eksperymentów, które pozwalają na porównanie efektywności obu metod.

Słowa kluczowe: wyszukiwanie wzorca, algorytm Boyera-Moore'a, algorytm Boyera-Moore'a-Horspoola, heurystyka.

1. Wstęp

W dobie cyfryzacji oraz wzrostu zasobów tekstowych coraz większego znaczenia nabierają metody skutecznego wyszukiwania wzorców w ciągach znaków, wykorzystywane zarówno w edytorach tekstu i silnikach wyszukiwarek, jak i w narzędziach do wykrywania plagiatu czy analizie danych biologicznych. W artykule zostaną przedstawione dwa popularne algorytmy wyszukiwania wzorca w tekście: algorytm Boyera-Moore'a wykorzystujący dwie heurystyki - złego znaku (ang. *bad character rule*) i dobrego sufiksu (ang. *good suffix rule*) oraz algorytm Boyera-Moore'a-Horspoola - uproszczona wersja wcześniej wspomnianego rozwiązania, w której stosowana jest tylko heurystyka złego znaku.

Algorytm Boyera-Moore'a to jedna z najszybszych metod wyszukiwania wzorca w tekście. Opublikowany został w 1977 roku przez Roberta S. Boyera oraz J Strother Moore'a [1]. Największą zaletą algorytmu Boyera–Moore'a jest rosnąca wraz z długością wzorca wydajność oraz wysoka efektywność wyszukiwania w długich tekstach.

Uproszczona wersja tej metody, zaproponowana przez Nigela Horspoola w 1980 roku i często nazywana algorytmem Horspoola, rezygnuje z heurystyki dobrego sufiksu i wykorzystuje wyłącznie zasadę złego znaku [2]. Dzięki temu implementacja staje się prostsza i w praktycznych testach algorytm ten osiąga bardzo dobrą wydajność.

Autor korespondencyjny: B. Wieczorek (Bozena.Wieczorek@polsl.pl).
Data wpłynięcia: 20.05.2025.

W dalszej części artykułu szczegółowo omówione zostaną oba algorytmy. Zaprezentowane zostanie ich działanie i przeanalizowana złożoność czasowa. Wykonane zostaną eksperymenty dla przykładowych implementacji algorytmów, co pozwoli na porównanie czasów działania obu metod. Stosowane będą następujące oznaczenia i pojęcia:

- n – długość tekstu, m – długość wzorca, s – przesunięcie wzorca względem początku tekstu, j – indeks znaku; znaki wzorca oraz tekstu indeksowane są od 0,
- **alfabet** $\sum$ – zbiór wszystkich możliwych znaków, które mogą się pojawić zarówno w tekście, jak i we wzorcu,
- **prefiks** – dowolny początkowy podciąg słowa, który może być równy całemu wyrazowi; przykładowo, dla słowa *ALGORYTM* prefiksami są m.in. *A*, *AL*, *ALGO* lub *ALGORYTM*,
- **sufiks** – dowolny końcowy podciąg słowa (również cały wyraz); dla słowa *ALGORYTM* sufiksami są m.in. *M*, *TM*, *RYTM* i *ALGORYTM*,
- **właściwy prefiks/sufiks** – prefiks lub sufiks, który nie jest równy całemu słowu,
- **prefikso-sufiks** – taki fragment słowa, który jest jednocześnie jego właściwym prefiksem i właściwym sufiksem; przykładowo, dla słowa *ABCAB* prefikso-sufiksem jest *AB*,
- **okno tekstu** – fragment tekstu o długości m, który w danej chwili porównywany jest ze wzorcem ($tekst[s..\, s+m-1]$).

2. Algorytm Boyera-Moore'a-Horspoola

2.1. Przedstawienie algorytmu

Algorytm BMH składa się z dwóch faz - przygotowania (ang. *preprocessing*) oraz wyszukiwania [3]. Zastosowano tutaj heurystykę złego znaku, pozwalającą na efektywne przesuwanie wzorca w tekście, gdy napotkane zostanie niedopasowanie. Złym znakiem nazywamy ten znak tekstu, który nie zgadza się z aktualnie analizowanym znakiem wzorca.

2.2. Szczegółowy opis algorytmu

Algorytm Boyer-Moore-Horspool rozpoczyna się od fazy przygotowania, w której tworzona jest tablica złego znaku (ang. *Bad Character Shift Table*) dla każdego znaku w alfabecie. Wartości w tablicy złego znaku określają, o ile pozycji można przesunąć wzorzec w prawo, gdy podczas porównywania wystąpi niezgodność na danej pozycji tekstu. Korzystamy tutaj ze wzoru $zlyZnak[wzorzec[i]] = m - i - 1$, gdzie m jest długością wzorca, a i oznacza pozycję znaku we wzorcu. Dla znaków alfabetu, które nie występują we wzorcu, ustala się domyślne przesunięcie równe m, czyli pełnej długości wzorca. Tablica $zlyZnak$ ma rozmiar równy liczbie wszystkich znaków w alfabecie.

W fazie wyszukiwania algorytm BMH porównuje znaki wzorca z tekstem od końca wzorca. W przypadku niedopasowania stosuje uproszczoną wersję heurystyki złego znaku, w której przesunięcie wzorca wyznacza się na podstawie ostatniego znaku aktualnego okna tekstu. Jeśli ten znak występuje we wzorcu, przesunięcie jest tak dobrane, aby dopasować jego ostatnie wystąpienie w obrębie wzorca. W przeciwnym razie wzorzec przesuwa się o swoją pełną długość.

2.3. Tworzenie tablicy złego znaku

W celu zobrazowania procesu tworzenia tablicy złego znaku przyjmijmy wzorzec *ABACB*, którego długość wynosi $m = 5$, oraz tekst *XXACBABACBBA* o długości $n = 12$. Dla tych danych alfabet definiujemy jako:

$$\Sigma = \{A, B, C, X\}.$$

W tablicy *zlyZnak* dla każdego znaku wzorca (poza ostatnim) zapisywana jest odległość od jego ostatniego wystąpienia (licząc od lewej do prawej) do końca wzorca. Dla znaków, które nie występują we wzorcu (jak X), przyjmuje się wartość domyślną równą długości wzorca, czyli $m = 5$. Poniżej przedstawiono sposób obliczania wartości:

$$\begin{aligned} zlyZnak[A] &= 5 - 0 - 1 = 4 \\ zlyZnak[B] &= 5 - 1 - 1 = 3 \\ zlyZnak[A] &= 5 - 2 - 1 = 2 \\ zlyZnak[C] &= 5 - 3 - 1 = 1 \end{aligned}$$

Końcowa postać tablicy złego znaku znajduje się w tabeli 1.

Tabela 1. Tablica złego znaku

Znak	*Wartość*
A	2
B	3
C	1
X	5

2.4. Przedstawienie działania fazy wyszukiwania

Po zakończeniu fazy przygotowania możliwe jest przejście do właściwego wyszukiwania. Na początku przesunięcie s ustawiane jest na 0. Następnie, dopóki $s \leq n - m$ (czyli dopóki wzorzec mieści się w pozostałej części tekstu), sprawdzana jest zgodność znaków. Proces przebiega zgodnie z poniższymi regułami:

- indeks j ustawiany jest na ostatnią pozycję wzorca; jeżeli znak $tekst[s + j]$ jest zgodny ze znakiem $wzorzec[j]$, indeks j jest dekrementowany, co odpowiada porównaniu kolejnych znaków w kierunku od końca do początku wzorca,
- w przypadku niezgodności znaków, wykorzystywana jest tablica złego znaku do obliczenia nowej wartości przesunięcia s, przy czym przesunięcie obliczane jest na podstawie ostatniego znaku aktualnego okna tekstu,
- jeśli indeks j osiągnie wartość -1, oznacza to pełne dopasowanie wzorca do fragmentu tekstu - zwracana jest wówczas pozycja dopasowania.

Załóżmy, że dany jest wzorzec *ABACB* oraz *XXACBABACBBA*. Kolejne kroki fazy wyszukiwania zostały przedstawione poniżej.

Poszukiwanie rozpoczynamy od $s = 0$ oraz $j = 4$.

0	1	2	3	4	5	6	7	8	9	10	11
X	X	A	C	B	A	B	A	C	B	B	A
A	B	A	C	B							

Znak X nie jest zgodny z B. Obliczamy przesunięcie biorąc pod uwagę ostatni znak okna tekstu, czyli $tekst[s + m - 1] = tekst[4] = B$ i otrzymujemy $s = s + zlyZnak[B] = 3$ oraz $j = 4$.

0	1	2	3	4	5	6	7	8	9	10	11
X	X	A	C	B	A	B	A	C	B	B	A
			A	B	A	C	B				

Znak A nie jest zgodny z B. Nowe przesunięcie wynosi $s = 3 + zlyZnak[A] = 5$, a $j = 4$.

0	1	2	3	4	5	6	7	8	9	10	11
X	X	A	C	B	A	B	A	C	B	B	A
					A	B	A	C	B		

Udało się całkowicie dopasować wzorzec na pozycji $s = 5$. Obliczamy przesunięcie $s = 5 + zlyZnak[B] = 8$. Nowa wartość s jest większa od $n - m$, co kończy wyszukiwanie.

2.5. Przykładowa implementacja

Przykładową implementację algorytmu Boyera-Moore'a-Horspoola przedstawiono na listingu 1.

```
#include <iostream>
#include <vector>
#include <string>

// Funkcja obliczająca tablicę przesunięć dla złego znaku
void heurystykaZlegoZnaku(const std::string& wzorzec, std::vector<int>& przesuniecie) {
  int m = wzorzec.length();
  for (int i = 0; i <= m - 2; i++) {
    przesuniecie[wzorzec[i]] = m - i - 1;
  }
}
void BMH(const std::string& wzorzec, const std::string& tekst) {
  int m = wzorzec.length();
  int n = tekst.length();
  std::vector<int> zlyZnak(256, m); // tablica złego znaku

  heurystykaZlegoZnaku(wzorzec, zlyZnak);

  int s = 0; // indeks przesunięcia wzorca względem tekstu
  int j = 0; // indeks znaku
  while (s <= n - m) {
    j = m - 1; // ustawienie indeksu znaku na koniec wzorca

    while (j >= 0 && wzorzec[j] == tekst[s + j]) {
      j--;
    }
    if (j == -1) {
      std::cout << "Wzorzec na pozycji: " << s << std::endl;
```

```
29      }
30      s = s + zlyZnak[tekst[s + m - 1]];
31    }
32  }
33  int main() {
34    std::string tekst, wzorzec;
35    std::cout << "Podaj wzorzec: " << std::endl;
36    getline(std::cin, wzorzec);
37    std::cout << "Podaj tekst: " << std::endl;
38    getline(std::cin, tekst);
39    BMH(wzorzec, tekst);
40    return 0;
41  }
```

Listing 1. Algorytm BMH

2.6. Złożoność obliczeniowa

Średnia złożoność czasowa algorytmu Boyera-Moore'a-Horspoola jest $O(n)$, czyli liniowa względem długości tekstu. W najgorszym przypadku złożoność czasowa algorytmu wynosi $O(nm)$, co może wystąpić w sytuacjach, gdy tekst i wzorzec zawierają powtarzające się znaki, a zastosowana heurystyka nie pozwala na dokonywanie większych przesunięć wzorca względem tekstu. Przykładowo, dla tekstu składającego się z ciągu znaków a i wzorca postaci $aaaaab$, przesunięcia dokonywane są jedynie o jeden znak. Faza przygotowania ma złożoność czasową $O(m + |\sum|)$, gdzie $|\sum|$ to rozmiar alfabetu.

3. Algorytm Boyera-Moore'a

3.1. Przedstawienie algorytmu

Algorytm Boyera-Moore'a, podobnie jak jego uproszczona wersja, składa się z dwóch faz: przygotowania i wyszukiwania [4]. W fazie przygotowania algorytm analizuje wzorzec, aby utworzyć dwie pomocnicze tablice: tablicę złego znaku (znaną już czytelnikowi) oraz tablicę dobrego sufiksu. W przypadku niedopasowania algorytm wybiera przesunięcie wzorca o odległość określoną przez większą z wartości umieszczonych w tablicach, maksymalizując w ten sposób efektywność wyszukiwania.

Korzystając z heurystyki złego znaku, algorytm BM analizuje niedopasowanie dokładnie w miejscu jego wystąpienia. Oznacza to, że przesunięcie wzorca obliczane jest względem faktycznie niepasującego znaku w tekście, a nie – jak w algorytmie BMH – jedynie względem ostatniego znaku bieżącego okna wyszukiwania. Mechanizm ten realizowany jest poprzez wyrażenie $zlyZnak[tekst[s + j]] - m + j + 1$, gdzie $s + j$ wskazuje pozycję niedopasowanego znaku w tekście. Wartość $zlyZnak[tekst[s + j]]$ określa domyślne przesunięcie dla danego znaku względem końca wzorca. Odejmując długość wzorca m i dodając $j + 1$, otrzymujemy skorygowaną wartość przesunięcia, która umożliwia wyrównanie wzorca z tekstem w miejscu faktycznego niedopasowania. Przykładowo, rozważmy wzorzec $ABCDBB$ o długości $m = 6$ oraz tekst $ABXDBBCDABY$. Dla przesunięcia $s = 0$, sprawdzamy zgodność od końca wzorca:

- $j = 5$: $tekst[5] = B$, $wzorzec[5] = B$ – zgodny
- $j = 4$: $tekst[4] = B$, $wzorzec[4] = B$ – zgodny
- $j = 3$: $tekst[3] = D$, $wzorzec[3] = D$ – zgodny

· $j = 2$: $tekst[2] = X$, $wzorzec[2] = C$ – **niezgodność**

W algorytmie BMH przesunięcie obliczane byłoby na podstawie ostatniego znaku okna, czyli $tekst[s+m-1] = tekst[5] = B$, wynosiłoby więc $zlyZnak[B] = 1$. W algorytmie Boyera–Moore'a heurystyka złego znaku uwzględnia faktyczny znak niezgodny, czyli X, który pojawił się na pozycji $s + j = 2$. Ponieważ znak X nie występuje we wzorcu ($zlyZnak[X] = m = 6$), nowe przesunięcie obliczamy jako:

$$s = s + zlyZnak[tekst[s + j]] - m + j + 1 = 0 + 6 - 6 + 2 + 1 = 3.$$

Wzorzec zostaje przesunięty o 3 pozycje.

3.2. Analiza działania heurystyki dobrego sufiksu

Heurystyka dobrego sufiksu pozwala na maksymalne przesunięcie wzorca w momencie niedopasowania, bazując na fragmencie wzorca, który został już poprawnie dopasowany do tekstu. Oznacza to, że jeśli pewna część wzorca zgadzała się z analizowanym fragmentem tekstu, ale kolejne porównanie zakończyło się niezgodnością, możemy odpowiednio przesunąć wzorzec rozważając przypadki opisane poniżej.

1. Dopasowany fragment wzorca występuje w innym miejscu wzorca – w takim przypadku wzorzec przesuwany jest tak, aby to wcześniejsze wystąpienie pokryło się z dopasowanym fragmentem tekstu. Przykładowo, rozważmy tekst i wzorzec:

   $$tekst : XBXYZABCDABC$$

   $$wzorzec : ZABCDABC$$

   Algorytm rozpoczyna porównywanie od ostatniego znaku wzorca i odpowiadającej mu pozycji w tekście:

X	B	X	Y	Z	A	B	C	D	A	B	C
Z	A	B	C	D	A	B	C				

   Trzy ostatnie znaki wzorca pasują do znaków w tekście, niezgodność pojawia się podczas porównania znaku D ze znakiem Z. Heurystyka dobrego sufiksu odnajduje we wzorcu wcześniejsze wystąpienie dopasowanego fragmentu i wzorzec zostaje przesunięty tak, aby wyrównać ABC z jego wystąpieniem w tekście:

X	B	X	Y	Z	A	B	C	D	A	B	C
				Z	A	B	C	D	A	B	C

2. Sufiks zgodnego fragmentu pasuje do początku wzorca (jeśli cały fragment nie powtarza się we wzorcu, ale sufiks tego fragmentu pokrywa się z prefiksem wzorca, możemy przesunąć wzorzec tak, aby te fragmenty zostały wyrównane). Przykładowo, dla następującego tekstu i wzorca:

   $$tekst : ABDCBABCCBAB$$

   $$wzorzec : ABCCBAB$$

   Algorytm rozpoczyna porównywanie od ostatniego znaku wzorca i odpowiadającej mu pozycji w tekście:

A	B	D	C	B	A	B	C	C	B	A	B
A	B	C	C	B	A	B					

Fragment $CBAB$ został dopasowany, niezgodność występuje na pozycji, gdzie D w tekście nie pasuje do znaku C we wzorcu. Heurystyka dobrego sufiksu rozpoznaje, że sufiks AB dopasowanego fragmentu pokrywa się z prefiksem wzorca, co pozwala na takie przesunięcie wzorca, aby te fragmenty się pokryły:

A	B	D	C	B	A	B	C	C	B	A	B
					A	B	C	C	B	A	B

3. Żaden z powyższych przypadków nie zachodzi i wtedy wzorzec może zostać przesunięty w całości poza aktualnie analizowany fragment tekstu. Dla przykładu, rozważmy następujący tekst i wzorzec:

$$tekst : XYCDABCD$$

$$wzorzec : ABCD$$

X	Y	C	D	A	B	C	D
A	B	C	D				

Po wykryciu niedopasowania, możemy przesunąć wzorzec całkowicie poza bieżące okno tekstu:

X	Y	C	D	A	B	C	D
				A	B	C	D

3.3. Tworzenie tablicy dobrego sufiksu

Uwzględniając trzy wcześniej omówione przypadki, możemy wyznaczyć wartości w tablicy $dobrySufiks$ wykonując następujące kroki:

1. Inicjalizujemy zerami dwie tablice o rozmiarze $m+1$: tablicę $pozycja$, która będzie przechowywała informacje o najdłuższych prefikso-sufiksach podciągów wzorca oraz tablicę $dobrySufiks$, która będzie zawierała wartości przesunięć wzorca w przypadku niedopasowania.

2. Wyznaczamy wartości w tablicy $pozycja$, gdzie $pozycja[j]$ określa pozycję najdłuższego właściwego sufiksu wzorca, który jest jednocześnie prefiksem podciągu $wzorzec[j..m-1]$ (od znaku pod indeksem j do końca wzorca). W celu uproszczenia implementacji (listing 2), pod indeksem m zostaje wstawiona wartość $m+1$. Przykładowo, dla wzorca $ABCAABCABC$ tablica $pozycja$ zawiera następujące wartości:

j	0	1	2	3	4	5	6	7	8	9	10
$wzorzec[j]$	A	B	C	A	A	B	C	A	B	C	
$pozycja[j]$	7	8	9	10	7	8	9	10	10	10	11

Dla $j = 0$ pozycja najdłuższego właściwego sufiksu, który jest jednocześnie prefiksem dla całego wzorca jest równa 7 (ciąg ABC).
Dla $j = 1$ wartość w tablicy wynosi 8, gdyż pod uwagę brany jest podciąg $wzorzec[1..9]$ i tutaj najdłuższy prefikso-sufiks to ciąg BC.
Jeżeli nie istnieje żaden prefikso-sufiks, w tablicy znajduje się wartość m, czyli długość wzorca.

3. W oparciu o tablicę *pozycja*, wyznaczamy wartości w tablicy $dobrySufiks$ tak, by każdy element $dobrySufiks[j+1]$ przechowywał informację o ile należy przesunąć wzorzec w prawo, jeśli niedopasowanie wystąpi na pozycji j. Wartość $dobrySufiks[0]$ jest używana do przesunięcia wzorca w przypadku pełnego dopasowania (znalezienia wzorca w tekście). Dla wzorca $ABCAABCABC$ otrzymujemy:

j	0	1	2	3	4	5	6	7	8	9	10
$wzorzec[j]$	A	B	C	A	A	B	C	A	B	C	
$pozycja[j]$	7	8	9	10	7	8	9	10	10	10	11
$dobrySufiks[j]$	7	7	7	7	7	7	7	3	10	10	1

Dla $j = 0$, $dobrySufiks[0] = 7$ – pełne dopasowanie wzorca, przykładowo:

tekst	A	B	C	A	A	B	C	A	B	C	A	B	B...
wzorzec	A	B	C	A	A	B	C	A	B	C			

Po przesunięciu o 7 pozycji:

tekst	A	B	C	A	A	B	C	A	B	C	A	B	B...
wzorzec								A	B	C	A	A	B...

Dla $j = 1$, $dobrySufiks[1] = 7$ – niedopasowanie na pozycji 0, przykładowo:

tekst	B	B	C	A	A	B	C	A	B	C	A	B	B...
wzorzec	A	B	C	A	A	B	C	A	B	C			

Po przesunięciu o 7 pozycji:

tekst	B	B	C	A	A	B	C	A	B	C	A	B	B...
wzorzec								A	B	C	A	A	B...

Dla wartości $j = 2$ do 6 przesunięcie wzorca realizowane jest w taki sam sposób, jak opisano powyżej – przy pomocy najdłuższego prefikso-sufiksu.

Dla $j = 7$, $dobrySufiks[7] = 3$ – niedopasowanie na pozycji 6, przykładowo:

tekst	C	B	A	C	A	C	B	A	B	C	A	B	B...
wzorzec	A	B	C	A	A	B	C	A	B	C			

Po przesunięciu o 3 pozycje:

tekst	C	B	A	C	A	C	B	A	B	C	A	B	B...
wzorzec				A	B	C	A	A	B	C	A	B	C

Dla $j = 8$, $dobrySufiks[8] = 10$ – niedopasowanie na pozycji 7, np.:

tekst	C	B	A	C	A	C	B	B	B	C	A	B	$B...$
wzorzec	A	B	C	A	A	B	C	A	B	C			

Po przesunięcie o całą długość wzorca:

tekst	C	B	A	C	A	C	B	B	B	C	A	B	$B...$
wzorzec											A	B	$C...$

Podobnie wykonamy przesunięcie dla $j = 9$.

Dla $j = 10$, $dobrySufiks[10] = 1$ – niedopasowanie ostatniego znaku wzorca, przykładowo:

tekst	C	B	A	C	A	C	B	B	B	A	A	B	$B...$
wzorzec	A	B	C	A	A	B	C	A	B	C			

Po przesunięciu o 1 pozycję:

tekst	C	B	A	C	A	C	B	B	B	A	A	B	$B...$
wzorzec		A	B	C	A	A	B	C	A	B	C		

Szczegółowy sposób wyznaczania wartości w tablicy $dobrySufiks$ pomijamy ze względu na jego złożoność. Czytelnik zainteresowany mechanizmem ich obliczania może prześledzić go samodzielnie na podstawie przedstawionej implementacji.

3.4. Przykładowa implementacja

Przykładowa implementacja algorytmu Boyera-Moore'a została przedstawiona na listingu 2.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

// Funkcja do obliczania tablicy przesunięć dla złego znaku
void heurystykaZlegoZnaku(const std::string& wzorzec, std::vector<int>& przesuniecie) {
  int m = wzorzec.length();

  for (int i = 0; i < m - 1; i++) {
    przesuniecie[wzorzec[i]] = m - 1 - i;
  }
}

// Funkcja do obliczania tablicy przesunięć dla dobrego sufiksu
void heurystykaDobregoSufiksu(const std::string& wzorzec, std::vector<int>&
  przesuniecie) {
  int m = wzorzec.length();
  std::vector<int> pozycja(m + 1, 0);
  int i = m, j = m + 1;
  pozycja[i] = j;

  while (i > 0) {
```

```
23        while (j <= m && wzorzec[i - 1] != wzorzec[j - 1]) {
24          if (przesuniecie[j] == 0) {
25            przesuniecie[j] = j - i;
26          }
27          j = pozycja[j];
28        }
29        i--;
30        j--;
31        pozycja[i] = j;
32      }
33
34      for (int i = 0; i <= m; i++) {
35        if (przesuniecie[i] == 0) {
36          przesuniecie[i] = j;
37        }
38        if (i == j) {
39          j = pozycja[j];
40        }
41      }
42    }
43
44    void BoyerMoore(const std::string& tekst, const std::string& wzorzec) {
45      int n = tekst.length();
46      int m = wzorzec.length();
47
48      std::vector<int> zlyZnak(256, m); // tablica przesunięć dla złego znaku
49      heurystykaZlegoZnaku(wzorzec, zlyZnak);
50
51      std::vector<int> dobrySufiks(m + 1, 0); // tablica przesunięć dla dobrego sufiksu
52      heurystykaDobregoSufiksu(wzorzec, dobrySufiks);
53
54      int s = 0; // indeks przesunięcia wzorca względem tekstu
55
56      while (s <= n - m) {
57        int j = m - 1;
58
59        // Porównujemy wzorzec z tekstem od końca wzorca
60        while (j >= 0 && wzorzec[j] == tekst[s + j]) {
61          j--;
62        }
63
64        if (j < 0) {
65          std::cout << "Wzorzec na pozycji: " << s << std::endl;
66          s += dobrySufiks[0];
67        }
68        else {
69          s += std::max(dobrySufiks[j + 1], zlyZnak[tekst[s + j]] - m + j + 1);
70        }
71      }
72    }
73
74    int main() {
75      std::string tekst, wzorzec;
76      std::cout << "Podaj wzorzec: " << std::endl;
77      getline(std::cin, wzorzec);
78      std::cout << "Podaj tekst: " << std::endl;
79      getline(std::cin, tekst);
```

```
80      BoyerMoore(tekst, wzorzec);
81      return 0;
82    }
83
```

Listing 2. Algorytm BM

3.5. Złożoność czasowa algorytmu Boyera–Moore'a

Algorytm Boyera-Moore'a w porównaniu z algorytmem Boyera-Moore'a-Horspoola charakteryzuje się podobną złożonością średnią - liniową względem długości tekstu. Różnicę stanowi zastosowanie dodatkowej heurystyki dobrego sufiksu, która pozwala w wielu przypadkach na dokonanie dłuższych przeskoków wzorca. Dzięki temu algorytm Boyera–Moore'a może działać szybciej niż jego uproszczona wersja, zwłaszcza w przypadku długich wzorców i dużej różnorodności znaków w tekście.

W najgorszym przypadku złożoność czasowa algorytmu Boyera–Moore'a jest również $O(nm)$. Faza przygotowania tablicy dobrego sufiksu wynosi $O(m)$.

3.6. Przeprowadzone eksperymenty

W celu porównania efektywności zaimplementowanych algorytmów przeprowadzono eksperymenty obejmujące trzy metody: Boyera-Moore'a, Boyera-Moore'a-Horspoola oraz algorytm naiwny, którego działanie polega na porównywaniu wzorca ze wszystkimi możliwymi podciągami tekstu, znak po znaku. Każdy eksperyment wykonano dla tekstu o długości $n = 100000$ znaków. Zastosowano cztery typy wzorców: krótki ($m = 40$) i długi ($m = 400$), zarówno bez prefikso-sufiksu, jak i z jego wystąpieniem. Wzorzec umieszczono jednokrotnie na końcu tekstu.

W pierwszym eksperymencie tekst oraz wzorce składały się wyłącznie z dwóch znaków alfabetu: a, b. Czasy wykonania dla poszczególnych metod przedstawiono w tabeli 2. Dane zostały tak dobrane, by uwzględnić przypadek pesymistyczny dla algorytmu naiwnego, czyli sytuację, w której za każdym razem niezgodność występuje dopiero na ostatniej pozycji wzorca, co powoduje konieczność wykonania m porównań dla każdego przesunięcia wzorca w tekście (dwa pierwsze wiersze). Każdą metodę zastosowano pięciokrotnie dla tych samych danych testowych, a w tabeli umieszczono wartości minimalne.

Tabela 2. Czasy działania algorytmów w pierwszym eksperymencie

m	Długość prefikso-sufiksu	Algorytm BMH $[\mu s]$	Algorytm BM $[\mu s]$	Algorytm naiwny $[\mu s]$
40	0	361	418	6410
400	0	393	472	55099
40	7	147	87	398
400	40	181	45	376

W drugim eksperymencie tekst i wzorce wygenerowano losowo z wykorzystaniem małych liter alfabetu łacińskiego (w dwóch przypadkach wzorce zmodyfikowano, by zawierały prefikso-sufiks). W tabeli 3 przedstawiono minimalne wartości czasów działania poszczególnych algorytmów otrzymane w pięciu próbach.

Wyniki przeprowadzonych eksperymentów wskazują na wysoką efektywność algorytmów Boyera–Moore'a oraz Boyera–Moore'a–Horspoola w porównaniu do metody naiwnej. W pierwszym eksperymencie, gdzie tekst i wzorce ograniczono do dwóch znaków, algorytm BM osiągał krótsze czasy działania niż

Tabela 3. Czasy działania algorytmów w drugim eksperymencie

m	Długość prefikso-sufiksu	Algorytm BMH [μs]	Algorytm BM [μs]	Algorytm naiwny [μs]
40	0	37	41	200
400	0	18	39	204
40	7	24	24	172
400	40	15	35	167

algorytm BMH jedynie dla wzorców zawierających prefikso-sufiks. W tych przypadkach BM mógł często korzystać z heurystyki dobrego sufiksu, co znacząco zmniejszało liczbę porównań. Z kolei w drugim eksperymencie, opartym na losowo generowanych danych z zestawu małych liter alfabetu łacińskiego, wzorce rzadziej zawierały odpowiednie układy znaków, umożliwiające skuteczne użycie heurystyki dobrego sufiksu. W efekcie różnice czasowe między BM a BMH okazały się niewielkie, przy czym w dwóch przypadkach zauważalnie lepsze rezultaty osiągnął algorytm BMH.

4. Podsumowanie

W niniejszej pracy szczegółowo przedstawiono dwa efektywne algorytmy wyszukiwania wzorca w tekście: klasyczny algorytm Boyera-Moore'a oraz uproszczony wariant Boyera-Moore'a-Horspoola. Oba algorytmy wykorzystują heurystykę tzw. złego znaku, natomiast algorytm Boyera-Moore'a dodatkowo korzysta z heurystyki dobrego sufiksu. W przeprowadzonych eksperymentach oba algorytmy wykazały wyraźną przewagę nad metodą naiwną. Szczególnie istotna była przewaga algorytmu Boyera-Moore'a w sytuacjach, gdy wzorzec zawierał prefikso-sufiks, co pozwalało na skuteczniejsze wykorzystanie heurystyki dobrego sufiksu i szybsze przeszukiwanie tekstu. Uzyskane wyniki potwierdzają, że zaawansowane algorytmy wyszukiwania wzorca w tekście oferują znaczną poprawę efektywności w praktyce. Wybór konkretnego algorytmu powinien być uzależniony od charakterystyki danych wejściowych oraz rodzaju analizowanego wzorca.

Literatura

1. R.S. Boyer, J.S. Moore, *A fast string searching algorithm*, Commun. ACM 20 (1977), pp. 762–772.
2. R.N. Horspool, *Practical fast searching in strings*, Software: Practice and Experience 10 (1980), pp. 501–506.
3. T. Lecroq, *Horspool Algorithm*, `https://www-igm.univ-mlv.fr/~lecroq/string/node18.html`, [Online; dostęp: maj 2025].
4. T. Lecroq, *Boyer–Moore Algorithm*, `https://www-igm.univ-mlv.fr/~lecroq/string/node14.html#SECTION00140`, [Online; dostęp: maj 2025].