MINUT 2025 (1), s. 87–98
ISSN 2719-3063

Bartosz KUCHAREWICZ[1], Marta KUCHARCZYK[2], Bożena WIECZOREK[3]

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

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

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

Zaawansowane algorytmy wyszukiwania wzorca w tekście

Streszczenie. Algorytmy wyszukiwania wzorca w tekście stanowią istotne narzędzie we współczesnym świecie. Znajdują zastosowanie w wielu dziedzinach, takich jak wyszukiwarki internetowe, filtry antyspamowe, biotechnologia czy sztuczna inteligencja. W niniejszym artykule zaprezentowane zostaną dwa mniej znane, zaawansowane algorytmy wyszukiwania wzorca – algorytm DFA oparty na deterministycznym automacie skończonym (ang. *Deterministic Finite Automaton*) oraz algorytm Shift-Or, wykorzystujący operacje bitowe.

Słowa kluczowe: wyszukiwanie wzorca, algorytm DFA, algorytm Shift-Or, automat skończony, przetwarzanie tekstu.

1. Wstęp

Wyszukiwanie wzorca w tekście stanowi jedno z kluczowych narzędzi w erze cyfryzacji. Algorytmy takie jak Shift-Or oraz DFA znajdują zastosowanie w bardziej zaawansowanych obszarach informatyki, w przeciwieństwie do szeroko znanych metod, takich jak algorytmy Knutha-Morrisa-Pratta (KMP), Boyera-Moore'a czy Rabina-Karpa. Niniejszy artykuł można potraktować jako rozwinięcie problemu przedstawionego w poprzednich publikacjach: „Algorytmy wyszukiwania wzorca w tekście" [3] oraz „Efektywne metody wyszukiwania wzorca w tekście: algorytmy Boyera-Moore'a i Horspoola" [4].

Algorytm oparty na deterministycznym automacie skończonym (DFA) stosuje się m.in. w przetwarzaniu wyrażeń regularnych (ang. *regular expression*, regex). Wykorzystywany jest także w analizatorach leksykalnych języków programowania - do rozpoznawania słów kluczowych, identyfikatorów czy liczb - jak również w systemach filtrowania ruchu sieciowego (np. do wykrywania wzorców ataków) oraz w wyszukiwarkach pełnotekstowych. Szczególnie dobrze sprawdza się w sytuacjach, gdy ten sam wzorzec należy wielokrotnie wyszukiwać w różnych tekstach, ponieważ raz skonstruowany automat może być używany wielokrotnie.

Autor korespondencyjny: B. Wieczorek (Bozena.Wieczorek@polsl.pl).
Data wpłynięcia: 25.08.2025.

Z kolei algorytm Shift-Or opiera się na dwóch operacjach bitowych: przesunięciu (ang. *shift*) oraz sumie (operator *OR*), służących do aktualizacji stanu dopasowania podczas przeszukiwania tekstu. Znajduje on zastosowanie w bioinformatyce, szczególnie w wyszukiwaniu fragmentów DNA, oraz w dopasowywaniu krótkich wzorców (do 64 znaków). Dla dłuższych wzorców konieczny jest podział na segmenty lub zastosowanie struktur wielobitowych, np. *std::bitset<N>* w języku C++.

W dalszej części artykułu przedstawione zostaną oba algorytmy. Zobrazowane będzie ich działanie oraz omówiona zostanie złożoność czasowa. Przeprowadzone eksperymenty dla przykładowych implementacji algorytmów pozwolą na porównanie czasów działania obu rozwiązań. Stosowane będą następujące oznaczenia i pojęcia:

- n – długość tekstu, m – długość wzorca, j – indeks znaku; znaki wzorca oraz tekstu są indeksowane 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 obejmować cały wyraz; przykładowo, dla słowa *AUTOMAT* prefiksami są m.in. *A*, *AU*, *AUTO* lub *AUTOMAT*,
- **sufiks** – dowolny końcowy podciąg słowa, w tym także całe słowo; dla słowa *AUTOMAT* sufiksami są m.in. *T*, *AT*, *OMAT* i *AUTOMAT*,
- **właściwy prefiks/sufiks** – prefiks lub sufiks, który nie jest równy całemu słowu,
- **prefikso-sufiks** – fragment słowa będący jednocześnie jego właściwym prefiksem i właściwym sufiksem.

2. Algorytm DFA

2.1. Przedstawienie algorytmu

Algorytm składa się z dwóch faz: konstrukcji automatu (budowy funkcji przejścia) oraz właściwego przeszukiwania tekstu. Obie te fazy są od siebie wyraźnie oddzielone — pierwsza jest niezależna od tekstu, a druga wykorzystuje wcześniej zbudowany automat do szybkiego dopasowywania wzorca. Charakterystyczną cechą automatu deterministycznego jest to, że dla każdego stanu i każdego symbolu alfabetu wejściowego określone jest dokładnie jedno przejście do innego (lub tego samego) stanu [2, 5].

2.2. Szczegółowy opis algorytmu

Główną ideą algorytmu DFA jest zbudowanie deterministycznego automatu skończonego, który skanuje tekst w celu znalezienia wszystkich wystąpień wzorca. Automat przetwarza każdy znak dokładnie jeden raz, dlatego złożoność obliczeniowa algorytmu jest liniowa względem długości tekstu. Czas fazy przygotowawczej (ang. *preprocessing*) może być jednak znaczny, ponieważ zależy od długości wzorca oraz

rozmiaru alfabetu. Skończony automat definiowany jest jako pięcioelementowa krotka:

$$DFA = (Q, q_0, \Sigma, \delta, A)$$

gdzie:

Q – to skończony zbiór stanów,

$q_0 \in Q$ – to stan początkowy,

Σ – oznacza alfabet,

$\delta\colon Q \times \Sigma \to Q$ – to funkcja przejścia,

$A \subseteq Q$ – oznacza zbiór stanów akceptowalnych.

Automat rozpoczyna poszukiwanie wzorca w stanie początkowym q_0 i odczytuje kolejno po jednym znaku z przeszukiwanego tekstu. Jeśli znajduje się w stanie q i odczytuje znak oznaczony jako c, następuje przejście do nowego stanu wyznaczonego przez funkcję przejścia: $\delta(q, c)$. Za każdym razem, gdy aktualny stan q należy do zbioru stanów A, automat akceptuje dotychczas odczytaną część tekstu. Znaki nienależące do alfabetu są odrzucane. Liczba stanów automatu wynosi $m + 1$, gdzie m oznacza długość wzorca. Najważniejszym założeniem przy konstruowaniu automatu jest zapewnienie, że dla każdego znaku alfabetu możliwe jest wyznaczenie kolejnego stanu na podstawie bieżącego.

2.3. Tworzenie deterministycznego automatu skończonego

Do zbudowania funkcji przejścia dla deterministycznego automatu skończonego posłuży pseudokod przedstawiony w listingu 1. Tablica P zawiera wzorzec o długości m.

```
1  procedure DFA(P[0..m-1])
2    for all c ∈ ∑ do
3      δ[0][c] := 0
4    end_for
5    δ[0][P[0]] := 1
6    x := 0
7    for j := 1 to m - 1 do
8      for all c ∈ ∑ do
9        δ[j][c] := δ[x][c]
10     end_for
11     δ[j][P[j]] := j + 1
12     x := δ[x][P[j]]
13   end_for
14 end_procedure
```

Listing 1. Konstrukcja funkcji przejścia dla DFA

Konstrukcja funkcji przejścia opiera się na iteracyjnym przetwarzaniu kolejnych znaków wzorca. W fazie inicjalizacji przypisuje się wartość 0 dla wszystkich znaków alfabetu Σ w stanie początkowym $q = 0$, który odpowiada teoretycznemu stanowi q_0. Oznacza to, że automat trwa w stanie początkowym do momentu rozpoznania pierwszego znaku wzorca $P[0]$. Dla tego znaku definiuje się przejście do stanu 1, co zapisuje się jako $\delta[0][P[0]] = 1$.

W kolejnych iteracjach ($j = 1..m - 1$) funkcja przejścia budowana jest w następujący sposób. Dla każdego stanu j kopiowane są wszystkie przejścia ze stanu pomocniczego x, który wskazuje najdłuższy właściwy prefiks wzorca $P[0..j - 1]$ będący jednocześnie jego sufiksem. Dzięki temu automat w stanie j zachowuje możliwość obsługi częściowych dopasowań. Następnie definiowane jest przejście dla znaku

wzorca $P[j]$ poprzez ustawienie:

$$\delta[j][P[j]] := j + 1.$$

Oznacza to, że automat może przejść do następnego stanu po poprawnym dopasowaniu znaku w tekście. Na koniec zmienna pomocnicza x jest aktualizowana zgodnie z wynikiem przejścia:

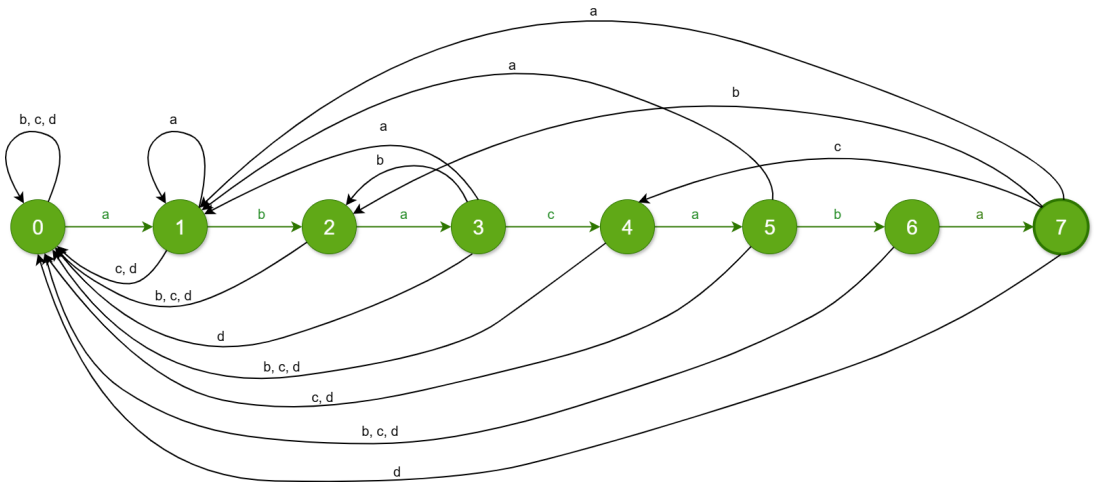
$$x := \delta[x][P[j]].$$

W efekcie wyznaczona zostaje tablica funkcji przejścia δ, która umożliwia przetwarzanie tekstu znak po znaku i wykrywanie pełnych dopasowań wzorca w czasie liniowym względem długości tekstu. W tabeli 1 przedstawiono przykładowe przejścia dla wzorca $P = abacaba$ i alfabetu $\Sigma = \{a, b, c, d\}$.

$\delta(q, c)$	0	1	2	3	4	5	6	7
a	1	1	3	1	5	1	7	1
b	0	2	0	2	0	6	0	2
c	0	0	0	4	0	0	0	4
d	0	0	0	0	0	0	0	0

Tabela 1. Tablica funkcji przejścia

Podana tabela została zobrazowana za pomocą grafu przejść na rysunku 1.



Rysunek 1. Graf przejść

2.4. Przedstawienie działania fazy wyszukiwania

Przeszukiwanie rozpoczyna się od stanu początkowego $q = 0$. Przejścia do kolejnych stanów automatu odbywają się zgodnie z odczytywanymi znakami tekstu. Osiągnięcie stanu końcowego przez automat oznacza, że cały wzorzec został dopasowany do fragmentu tekstu.

Na podstawie tabeli 1 skonstruowanej dla wzorca $P = abacaba$, przedstawione zostaną kolejne kroki fazy wyszukiwania dla tekstu $T = ababacababadabacaba$.

- Algorytm rozpoczyna działanie od pozycji $q = 0$.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0																			

- Na podstawie funkcji przejścia określa się zgodność, a nowy stan wynosi $\delta(0, a) = 1$. Proces kontynuowany jest aż do napotkania niezgodności.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1																		

- W momencie wykrycia niezgodności znaku b ze wzorcem, funkcja przejścia wskazuje stan $\delta(3, b) = 2$, co umożliwia kontynuację wyszukiwania.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1	2	3	2															

- Znak a jest zgodny z wzorcem, następuje przejście do stanu $\delta(2, a) = 3$.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1	2	3	2	3														

- Znak c jest zgodny z wzorcem, w związku z czym następuje przejście do stanu $\delta(3, c) = 4$, a proces wyszukiwania jest kontynuowany.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1	2	3	2	3	4													

- Udało się osiągnąć stan 7 równy długości wzorca. Wzorzec został dopasowany.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1	2	3	2	3	4	5	6	7										

- W tekście pojawia się litera d, która nie występuje we wzorcu.Następuje powrót do stanu $\delta(3, d) = 0$.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1	2	3	2	3	4	5	6	7	2	3	0							

- Na końcu tekstu wzorzec również został całkowicie dopasowany.

T		a	b	a	b	a	c	a	b	a	b	a	d	a	b	a	c	a	b	a
q	0	1	2	3	2	3	4	5	6	7	2	3	0	1	2	3	4	5	6	7

2.5. Przykładowa implementacja

Przykładową implementację algorytmu wyszukiwania wzorca w tekście z pomocą automatu skończonego przedstawiono w listingu 2. Rozwiązanie składa się z trzech zasadniczych funkcji. Główna funkcja DFA odpowiada za przeszukiwanie tekstu w celu odnalezienia wszystkich wystąpień wzorca. W pierwszym kroku wywołuje ona funkcję obliczTF, która na podstawie wzorca buduje tablicę funkcji przejścia automatu. W tym celu korzysta z funkcji pomocniczej oblNastepnyStan, która wyznacza, do jakiego stanu należy przejść po odczytaniu danego znaku w określonym stanie.

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

//Funkcja pomocnicza, oblicza następny stan dla każdego stanu automatu
int oblNastepnyStan(const std::string& wzorzec, int stan, int x)
{
  //sprawdzana jest zgodność aktualnego znaku x z kolejnym znakiem we wzorcu
  if (stan < wzorzec.size() && x == wzorzec[stan])
  return stan + 1;

  //jeżeli znaki nie są zgodne, szukany jest najdłuższy prefiks będący również sufiksem
  for (int nastepnyStan = stan; nastepnyStan > 0; nastepnyStan--)
  {
    //sprawdzana jest zgodność znaku x z końcowym znakiem możliwego prefiksu
    if (wzorzec[nastepnyStan - 1] == x)
    {
      bool zgodny = true;
      for (int i = 0; i < nastepnyStan - 1; i++)
      {
        if (wzorzec[i] != wzorzec[stan - nastepnyStan + 1 + i])
        {
          zgodny = false;
          break;
        }
      }
      if (zgodny)
      return nastepnyStan;
    }
  }
  return 0;
}
//Funkcja inicjuje tablicę funkcji przejścia na podstawie wzorca
void obliczTF(const std::string& wzorzec, std::vector<std::vector<int>>& TF)
{
  for (int stan = 0; stan <= wzorzec.size(); ++stan)
  {
    for (int x = 0; x < 256; ++x)
    {
      //wyznaczenie kolejnego stanu po odczytaniu znaku x
      TF[stan][x] = oblNastepnyStan(wzorzec, stan, x);
    }
  }
}
//Główna funkcja służąca do wyszukiwania wzorca w tekście
void DFA(const std::string& wzorzec, const std::string& tekst)
{
  int m = wzorzec.size();
  int n = tekst.size();
  std::vector<std::vector<int>> TF(m + 1, std::vector<int>(256));

  obliczTF(wzorzec, TF);

  int stan = 0;
  for (int i = 0; i < n; i++)
  {
    stan = TF[stan][tekst[i]];
```

```
58      if (stan == m)
59      std::cout << "Wzorzec znaleziony na pozycji: " << i - m + 1 << std::endl;
60    }
61  }
62  int main() {
63    std::string wzorzec, tekst;
64    std::cout << "Podaj wzorzec:" << std::endl;
65    getline(std::cin, wzorzec);
66    std::cout << "Podaj tekst:" << std::endl;
67    getline(std::cin, tekst);
68    DFA(wzorzec, tekst);
69    return 0;
70  }
```

Listing 2. Algorytm DFA

2.6. Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu wyszukiwania wzorca w tekście przy użyciu deterministycznego automatu skończonego obejmuje dwa etapy: fazę konstrukcji automatu (tj. funkcji przejścia) oraz fazę przeszukiwania tekstu. Dla etapu budowy automatu złożoność czasowa wynosi $O(m^2 \cdot |\Sigma|)$ i jest zależna zarówno od rozmiaru alfabetu $|\Sigma|$, jak i od długości wzorca m. W przypadku standardowego alfabetu ASCII ($|\Sigma| = 256$) złożoność ta upraszcza się do $O(m^2)$. Natomiast faza przeszukiwania tekstu o długości n odbywa się w czasie liniowym względem długości tekstu, co daje złożoność $O(n)$.

3. Algorytm Shift-Or

3.1. Przedstawienie algorytmu

Algorytm Shift-Or wykorzystuje operacje bitowe w celu dopasowywania wzorca do tekstu [1]. Jego działanie można podzielić na dwie fazy: przygotowanie masek bitowych oraz właściwe przeszukiwanie tekstu. Podejście bitowe pozwala na równoległe porównywanie wielu pozycji wzorca w ramach pojedynczych operacji maszynowych, co znacząco zwiększa efektywność wyszukiwania.

3.2. Szczegółowy opis algorytmu

Podstawową ideą algorytmu Shift-Or jest realizacja dopasowywania wzorca w tekście przy użyciu reprezentacji bitowej. Każda pozycja we wzorcu jest reprezentowana przez jeden bit w słowie maszynowym. Jeżeli długość wzorca m przekracza rozmiar słowa maszynowego w, wzorzec dzielony jest na $\lceil \frac{m}{w} \rceil$ segmentów, a operacje bitowe wykonywane są kolejno dla każdego segmentu.

W pierwszej fazie dla każdego znaku c alfabetu Σ tworzona jest tzw. maska bitowa $B[c]$, która wskazuje pozycje wystąpień znaku c we wzorcu P. $B[c]$ to słowo bitowe długości m, w którym bit na pozycji i ma wartość 0 w przypadku wystąpienia znaku c na pozycji i we wzorcu. W przeciwnym przypadku bit ten przyjmuje wartość 1. Formalnie można to zapisać następująco:

$$B[c] = (b_{m-1}, b_{m-2}, \ldots, b_0),$$

$$\text{gdzie } b_i = \begin{cases} 0, & \text{jeśli } P[i] = c, \\ 1, & \text{w przeciwnym przypadku.} \end{cases}$$

Po przygotowaniu masek bitowych następuje faza przeszukiwania, polegająca na sekwencyjnym przetwarzaniu znaków tekstu. Wykorzystywany jest tutaj operator przesunięcia bitowego w lewo $\ll$, który przesuwa wszystkie bity w rejestrze o określoną liczbę pozycji, wstawiając od prawej strony zera. Dodatkowo stosowana jest suma logiczna bitów, realizowana za pomocą operatora |.

Aktualny stan dopasowania wzorca reprezentowany jest przez rejestr bitowy R o rozmiarze m, który aktualizowany jest dla każdego znaku c w tekście według następującego wzoru:

$$R = ((R \ll 1) \mid B[c])$$

Dla tekstu T i aktualnego przesunięcia s, i-ty bit rejestru $R[i] = 0$ wtedy i tylko wtedy, gdy

$$T[s, \ldots, s+i] = P[0, \ldots, i].$$

Sygnałem wskazującym na pełne dopasowanie wzorca jest wyzerowanie najbardziej znaczącego bitu rejestru R znajdującego się na pozycji $m-1$. Zachodzi wtedy następująca zależność:

$$R \;\&\; (1 \ll (m-1)) = 0$$

3.3. Przedstawienie działania algorytmu

Załóżmy, że dany jest wzorzec $BACB$ oraz tekst $BABCBACB$ o długości $n = 8$. Tworzenie masek dla każdej z liter zostało przedstawione poniżej:

Maska dla litery A: 1101; $b_1 = 0$, ponieważ $P[1] = A$

Maska dla litery B: 0110; $b_0 = 0$ i $b_3 = 0$, ponieważ $P[0] = B$ i $P[3] = B$

Maska dla litery C: 1011; $b_2 = 0$, ponieważ $P[2] = C$

Na początku rejestr R przyjmuje wartość 1111. Kolejne kroki fazy przeszukiwania tekstu i sprawdzania dopasowania są następujące:

- Przetwarzanie znaku B znajdującego się na pozycji 0:

$$\begin{aligned} R &= ((R << 1) | B['B']) \\ R &= ((1111 << 1) | 0110) \\ R &= (1110 | 0110) \\ R &= 1110 \end{aligned}$$

- Przetwarzanie znaku A znajdującego się na pozycji 1:

$$\begin{aligned} R &= ((R << 1) | B['A']) \\ R &= ((1110 << 1) | 1101) \\ R &= 1100 | 1101 \\ R &= 1101 \end{aligned}$$

- Przetwarzanie znaku B znajdującego się na pozycji 2:

$$R = ((R << 1)|B['B'])$$
$$R = ((1101 << 1)|0110)$$
$$R = 1010|0110$$
$$R = 1110$$

- Przetwarzanie znaku C znajdującego się na pozycji 3:

$$R = ((R << 1)|B['C'])$$
$$R = ((1110 << 1)|1011)$$
$$R = 1100|1011$$
$$R = 1111$$

- Przetwarzanie znaku B znajdującego się na pozycji 4:

$$R = ((R << 1)|B['B'])$$
$$R = ((1111 << 1)|0110)$$
$$R = 1110|0110$$
$$R = 1110$$

- Przetwarzanie znaku A znajdującego się na pozycji 5:

$$R = ((R << 1)|B['A'])$$
$$R = ((1110 << 1)|1101)$$
$$R = 1100|1101$$
$$R = 1101$$

- Przetwarzanie znaku C znajdującego się na pozycji 6:

$$R = ((R << 1)|B['C'])$$
$$R = ((1101 << 1)|1011)$$
$$R = 1010|1011$$
$$R = 1011$$

- Przetwarzanie znaku B znajdującego się na pozycji 7:

$$R = ((R << 1)|B['B'])$$
$$R = ((1011 << 1)|0110)$$
$$R = 0110|0110$$
$$R = 0110$$

Wartość najbardziej znaczącego bitu rejestru R została ustawiona na 0, co oznacza, że wzorzec został całkowicie dopasowany.

3.4. Przykładowa implementacja

Przykładową implementację algorytmu Shift-Or przedstawiono w listingu 3. Założono, że długość słowa maszynowego wynosi 64 bity, a wzorzec mieści się w tym limicie.

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

void shiftOr(const std::string& wzorzec, const std::string& tekst)
{
  int m = wzorzec.size();
  int n = tekst.size();
  const int rozmiar_słowa_maszynowego = 64;

  // wektor masek dla znaków ASCII, zainicjowanych jedynkami
  std::vector<unsigned long long> maska(256, ~0ULL);
  unsigned long long R = ~0ULL; // początkowy stan rejestru R

  // sprawdzenie czy długość wzorca nie przekracza długości słowa maszynowego
  if (m == 0 || m > rozmiar_słowa_maszynowego)
      return;

  for (int i = 0; i < m; ++i)
  {
    maska[wzorzec[i]] &= ~(1ULL << i); // wyznaczanie masek bitowych
  }

  for (int i = 0; i < n; ++i)
  {
    R <<= 1; // przesunięcie bitowe rejestru
    R |= maska[tekst[i]]; // operacja OR na rejestrze i masce znaku
    if ((R & (1ULL << m-1)) == 0)
    {
      std::cout << "Wzorzec znalezniony na pozycji: " << i - m + 1 << std::endl;
    }
  }
}

int main()
{
  std::string wzorzec, tekst;
  std::cout << "Podaj wzorzec: " << std::endl;
  getline(std::cin, wzorzec);
  std::cout << "Podaj tekst: " << std::endl;
  getline(std::cin, tekst);
  shiftOr(wzorzec, tekst);
  return 0;
}

```

Listing 3. Algorytm Shift-Or

3.5. Złożoność obliczeniowa algorytmu

Algorytm Shift-Or charakteryzuje się złożonością czasową $O\left(m + |\Sigma| + n \cdot \left\lceil \frac{m}{w} \right\rceil\right)$, gdzie $|\Sigma|$ oznacza rozmiar alfabetu, a w długość słowa maszynowego. Obliczenie masek bitowych wymaga czasu $O(m+|\Sigma|)$, natomiast wyszukiwanie wzorca, gdy $m > w$, wymaga podziału wzorca na $\lceil \frac{m}{w} \rceil$ segmentów, co daje złożoność $O\left(n \cdot \left\lceil \frac{m}{w} \right\rceil\right)$. Sumarycznie złożoność czasowa algorytmu wynika z połączenia obu faz.

3.6. Przeprowadzone eksperymenty

W celu porównania czasów wykonania algorytmów wyszukiwania wzorca w tekście zaimplementowano klasyczny algorytm naiwny, który realizuje zadanie poprzez sukcesywne porównywanie wzorca z każdym możliwym podciągiem tekstu – znak po znaku. Eksperymenty przeprowadzono dla tekstu o długości $n = 100000$ znaków, wykorzystując dwa wzorce: krótki o długości $m = 3$ oraz długi, liczący $m = 40$ znaków.

Na wstępie zbadano przypadek pesymistyczny dla algorytmu naiwnego. W tym celu wygenerowano tekst oraz wzorce złożone wyłącznie z liter a, przy czym ostatni znak stanowiła litera b. Wyniki przeprowadzonych eksperymentów przedstawiono w tabeli 2. Zaprezentowane czasy są najmniejszymi wartościami spośród pięciu niezależnych pomiarów.

Tabela 2. Czasy działania algorytmów dla przypadku pesymistycznego

m	Algorytm DFA $[\mu s]$	Algorytm Shift-Or $[\mu s]$	Algorytm naiwny $[\mu s]$
3	189	71	427
40	264	74	5953

W kolejnym eksperymencie zarówno tekst, jak i wzorce zawierały zróżnicowane ciągi znaków, typowe dla naturalnego języka pisanego, przy czym wzorzec występował w tekście wielokrotnie. Minimalne czasy z 5 prób zestawiono w tabeli 3.

Tabela 3. Czasy działania algorytmów dla przypadku średniego

m	Algorytm DFA $[\mu s]$	Algorytm Shift-Or $[\mu s]$	Algorytm naiwny $[\mu s]$
3	203	97	316
40	193	75	257

Wyniki przedstawione w tabeli 2 potwierdzają, że algorytm naiwny w przypadku pesymistycznym wykazuje znacznie gorszą wydajność niż pozostałe algorytmy. Dla krótkiego wzorca ($m = 3$) jego czas działania był ponad dwukrotnie dłuższy niż czas algorytmu DFA i sześciokrotnie dłuższy niż czas algorytmu Shift-Or. Różnice te stają się jeszcze bardziej wyraźne w przypadku długiego wzorca ($m = 40$).

Z kolei tabela 3 ukazuje, że w przypadku tekstu naturalnego algorytmy DFA i Shift-Or utrzymują wysoką i stabilną wydajność, niezależnie od długości wzorca. Szczególnie korzystnie wypada algorytm Shift-Or, który konsekwentnie osiąga najkrótsze czasy wykonania.

4. Podsumowanie

W niniejszym artykule omówiono dwa zaawansowane algorytmy wyszukiwania wzorca w tekście: deterministyczny automat skończony (DFA) oraz algorytm Shift-Or, oparty na operacjach bitowych. Oba

algorytmy zostały porównane z metodą naiwną zarówno w przypadku pesymistycznym, jak i przy wykorzystaniu tekstu naturalnego. Wyniki eksperymentów potwierdziły znaczną przewagę obu badanych algorytmów nad podejściem naiwnym w scenariuszu pesymistycznym, gdzie wzorzec pojawiał się jednokrotnie i był trudny do dopasowania. Dla tekstu naturalnego różnice w czasach wykonania uległy zmniejszeniu, a szczególnie zauważalna była przewaga algorytmu Shift-Or, który utrzymał najkrótsze czasy niezależnie od długości wzorca. Algorytm DFA, choć wymaga wcześniejszej konstrukcji automatu, wykazuje dużą stabilność czasową i większą elastyczność pod względem długości wzorca. Co istotne, raz zbudowany automat może być wykorzystywany wielokrotnie do przeszukiwania różnych tekstów, co czyni tę metodę szczególnie efektywną w sytuacjach, w których poszukiwany wzorzec jest stały, a zmienia się jedynie przeszukiwany tekst. Uzyskane wyniki wskazują, że dobór algorytmu powinien uwzględniać zarówno charakter danych wejściowych, jak i długość oraz strukturę wzorca — przy czym algorytm Shift-Or sprawdzi się najlepiej w przypadku krótkich wzorców, natomiast DFA może stanowić korzystne rozwiązanie w przetwarzaniu wielu tekstów dla tego samego wzorca.

Literatura

1. R. Baeza-Yates, G. H. Gonnet, *A new approach to text searching*, October 1992/Vol.35, No.10/COMMUNICATION OF THE ACM `https://dl.acm.org/doi/pdf/10.1145/135239.135243`
2. B. Barilee, P. Ejendibia *String Searching with DFA-based Algorithm*, [w:] *International Journal of Applied Information Systems* (2015) 9. 1-6. 10.5120/ijais2015451415.
3. M. Kucharczyk, B. Wieczorek *Algorytmy wyszukiwania wzorca w tekście*, MINUT 2024(6), s. 70 - 80. `https://minut.polsl.pl/articles/C-24-001.pdf`
4. M. Kucharczyk, B. Wieczorek *Efektywne metody wyszukiwania wzorca w tekście: algorytmy Boyera-Moore'a i Horspoola*, MINUT 2025(7), s. 56 - 67. `https://minut.polsl.pl/articles/C-25-001.pdf`
5. B. Melichar, *Approximate string matching by finite automata*, [w:] Hlaváč, V., Šára, R. (eds) Computer Analysis of Images and Patterns. CAIP 1995. Lecture Notes in Computer Science, vol 970. Springer, Berlin, Heidelberg. `https://doi.org/10.1007/3-540-60268-2_315`