4. Algorytmy wyszukiwania wzorca

4.1 Wstęp

Alfabet. Każdy niepusty zbiór skończony V nazywamy alfabetem. Elementy alfabetu nazywamy symbolami, znakami lub literami. Natura elementów ze zbioru V czyli symboli jest w gruncie rzeczy sprawą drugorzędną. Symbolami mogą być np.: wyróżnione stany układu fizycznego, ale wygodniej jest wyobrażać sobie symbole jako znaki pisane na papierze.

Przykład. Alfabetami są na przykład następujące zbiory skończone $V = \{0,1\}$, $V = \{a,b,c,d\}$ $V = \{a,b,c,...,x,y,z\}$; $V = \{0,1,2,...,9\}$, $V = \{0,1,...,9,A,B,C,D,E,F\}$

Słowo nad alfabetem V to dowolny ciąg skończony $a_1, a_2, ..., a_n$ o współczynnikach w V tzn. taki, że $a_i \in V$ dla $i = 1,2,...,n$. Z reguły stosujemy notację bez przecinków i piszemy np.: $a_1 a_2$, zamiast a_1, a_2 oraz $a_1 a_2 ... a_n$ zamiast $a_1, a_2, ..., a_n$.

Przykład 1 Niech $V = \{0,1\}$. Słowami nad alfabetem $\{0,1\}$ są: 0, 1, 0111000

Przykład 2 Niech $V = \{a,b,c,\}$. Słowami nad alfabetem V są np.: *aaabc, abccba*.

Uwaga. Czasami słowo nad alfabetem nazywamy tekstem, napisem, łańcuchem lub stringiem

Długość słowa $x = a_1, a_2, ..., a_n$ to długość ciągu skończonego $a_1, a_2, ..., a_n$ czyli n. Długość słowa x oznaczamy symbolem $|x|$.

Zbiór wszystkich słów nad alfabetem V oznaczamy symbolem V^*. Do zbioru V^* zaliczamy również specjalne słowo tzw. **słowo puste** oznaczane symbolem ε. Słowo puste ma długość 0, tzn. $|\varepsilon| = 0$.

W zbiorze V^* wprowadzamy dwuargumentowe działanie konkatenacji czyli działanie składania dwóch słów. Jeśli x, y są słowami $x, y \in V^*$ to konkatenację tych słów oznaczamy symbolem $x \cdot y$ lub xy. Jest to słowo o długości $|x| + |y|$ będące zestawieniem słów x i y, np.:
słowo *a b* i słowo *c d* dają w wyniku konkatenacji *abcd*
kot i *let* dają *kotlet*

Działanie konkatenacji oczywiście nie jest przemienne, ale jest łączne. Poza tym słowo puste ε jest jedynką tego działania tzn. dla każdego $x \in V^*$ $x\varepsilon = \varepsilon x = x$. Zatem para uporządkowana $(V^*, \cdot)$, gdzie „·” jest konkatenacją, jest monoidem (czyli półgrupą z jednością). Każdy podzbiór $L \subset V^*$ nazywamy językiem.

Mówimy, że **słowo *w* jest prefiksem słowa *x*** wtedy i tylko wtedy, gdy istnieje takie słowo $y \in V^*$, że

$$x = wy \qquad (*)$$

Fakt ten zapisujemy jako $w \subset x$. Oczywiście, jeśli $w \subset x$, to $|w| \leq |x|$.

Definicja (*) wprowadza w V^* relację dwuargumentową $\subset$ tzw. relację prefiksową. Podobieństwo symbolu relacji prefiksowej do relacji inkluzji zborów nie będzie w dalszym ciągu rozdziału z uwagi na kontekst prowadziło do nieporozumień. Relacja prefiksowa $\subset$ jest przechodnia tzn. dla każdego $x, y, z \in V^*$ mamy

$$((x \subset y) \wedge (y \subset z)) \Rightarrow x \subset z$$

Krótko: „Prefiks prefiksu jest prefiksem".

Fakt. Relacja prefiksowa w V^* jest relacją częściowego (ale nie liniowego) porządku.

Dowód. Przechodniość, zwrotność i antysymetria relacji prefiksowej $\subset$ są oczywiste.■

Podobnie mówimy, że **słowo *w* jest sufiksem słowa *x*** wtedy i tylko wtedy, gdy istnieje takie słowo $y \in V^*$, że

$$x = yw \qquad (**)$$

Fakt ten zapisujemy jako $w \supset x$. Oczywiście jeśli $w \supset x$ to $|w| \leq |x|$. Definicja (**) wprowadza w V^* relację dwuargumentową $\supset$ tzw. relację sufiksową. Relacja sufiksowa $\supset$ jest przechodnia, tzn. dla każdego $x, y, z \in V^*$ mamy

$$((x \supset y) \wedge (y \supset z)) \Rightarrow x \supset z$$

Krótko: „sufiks sufiksu jest sufiksem".

Fakt: Relacja sufiksowa jest częściowym (ale nie liniowym) porządkiem.

Dowód. Przechodniość, zwrotność i antysymetria relacji sufiksowej $\supset$ są oczywiste. ■

Uwaga. Słowo puste ε jest jednocześnie prefiksem i sufiksem każdego słowa.

Przykład. *pre* $\subset$ *prefix*, *ab* $\subset$ *abccca*
abra $\supset$ *makabra,* *la* $\supset$ *lala*

Kilka prostych własności relacji prefiksu i sufiksu:

Dla każdego $x, y \in V^*$ i $a \in V$

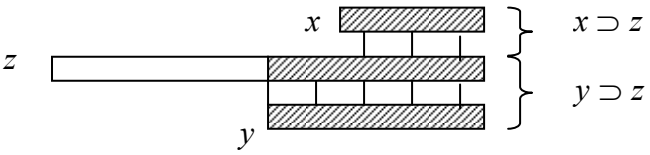
$x \subset y$ wtedy i tylko wtedy, gdy $ax \subset ay$

Dla każdego $x, y \in V^*$ i $a \in V$

$$x \supset y \quad \text{wtedy i tylko wtedy, gdy} \quad xa \supset ya$$

Lemat (o nierozłącznych sufiksach)
Niech $x, y, z \in V^*$ oraz niech $x \supset z$ i $y \supset z$. Jeśli $|x| \leq |y|$ to $x \supset y$. Jeśli $|x| \geq |y|$ to $y \supset x$
Jeśli $|x| = |y|$ to $x = y$.

Dowód. Po kolei trzeba rozwiązać wszystkie przypadki tzn. $|x| \leq |y|$, $|y| \leq |x|$ i $|x| = |y|$. Na przykład w przypadku, gdy $|x| \leq |y|$ oraz $x \supset z$ i $y \supset z$ mamy sytuację pokazaną na rys. 4.1



Rys.4.1 Ilustracja sytuacji dla przypadku, gdy $|x| \leq |y|$ oraz $x \supset z$ i $y \supset z$. Kreski pionowe łączą identyczne znaki w słowach *x*,*y*,*z*.

Ze szkicu sytuacji pokazanej na rys. 4.1 widać, że rzeczywiście $x \supset y$.■

Podsłowo. Niech V będzie ustalonym alfabetem, mówimy, że *x* jest **podsłowem** słowa y (co oznaczamy jako $x \subseteq y$) wtedy i tylko wtedy (z definicji) jeśli istnieją takie dwa słowa $z; w \in V^*$, że $y = zxw$. Słowo x jest **podsłowem właściwym** słowa y jeśli $x \subseteq y$ i $x \neq y$.

Fakt. Wprowadzona w V^* relacja podsłowa $\subseteq$ jest relacją częściowego porządku.

Dowód. Dla dowodu wystarcza proste sprawdzenie czy relacja $\subseteq$ jest zwrotna, antysymetryczna i przechodnia.■

Uwaga Jeśli mamy dane słowo $a_1, a_2, ..., a_n$ to wygodnie je zapisywać jako $a[1..n]$ a każde podsłowo tego słowa jako $a[l..r]$, gdzie $l, r \in N$, $l \leq r \leq n$.

Problem wyszukiwania wzorca

Z problemem wyszukiwania wzorca mamy do czynienia w praktyce bardzo często. Na przykład w programach do redagowania tekstu, wyszukiwarkach służących do wyszukiwania nazwy pliku w systemie plików czy przy szukaniu specjalnych wzorców w sekwencjach DNA w medycynie (ekspresja genów) i genetyce (sprawdzanie pokrewieństwa) a również rozpoznawaniu obrazów. Efektywne algorytmy wyszukiwania wzorca są więc bardzo potrzebne w wielu dziedzinach.

Dane wejściowe w problemie wyszukiwania wzorca to tekst (słowo nad ustalonym alfabetem V) i wzorzec (słowo nad ustalonym tym samym alfabetem V). Dane wyjściowe to wszystkie miejsca wystąpień wzorca P.

Problem wyszukiwania wzorca formalizujemy tak. Tekst T, dokładniej słowo $T[1...n]$ o długości n (nad alfabetem V) utożsamiamy z tablicą znakową $T[1..n]$. Wzorzec P, dokładniej słowo $P[1...m]$ o długości m (nad alfabetem V) utożsamiamy z tablicą znakową $P[1..m]$.

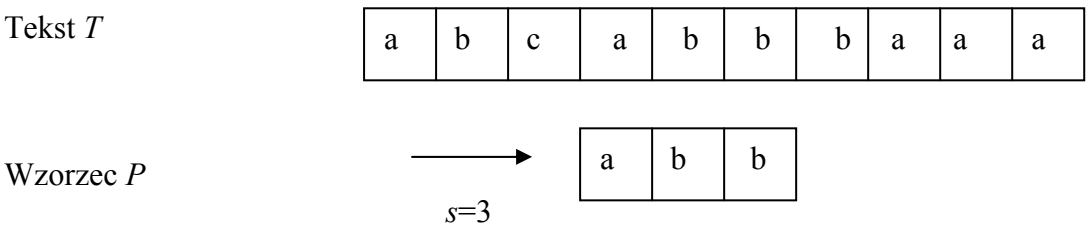
Mówimy, że wzorzec P występuje z przesunięciem s w teście T lub równoważnie wzorzec P występuje począwszy od pozycji $s+1$, gdy:

1) $0 \le s \le n - m$

2) $T[s+1,.s+m] = P[1..m]$ (czyli dla każdego $j \in <1, m>$ $T[s+j] = P[j]$

Jeśli wzorzec P występuje z przesunięcia s w tekście T to s **nazywamy poprawnym przesunięciem**, w przeciwnym razie s **nazywamy niepoprawnym przesunięciem**.

Problem wyszukiwania wzorca polega na znajdowaniu wszystkich poprawnych przesunięć dla danego wzorca P i tekstu T.



Rys. 4.2 Ilustracja problemu wyszukiwania wzorca (problemu WW) dla alfabetu $V = \{a, b, c\}$ tekstu $T[1..10] = abcabbbaac$ i wzorca $P = abb$. Przesunięcie s=3.

Uwaga. Dla uproszczenia notacji przez P_k oznaczamy k-symbolowy prefiks $P[1...k]$ wzorca $P[1...m]$. Zatem: $P_0 = \varepsilon$ (słowo puste), $P_m = P = P[1..m]$. Podobnie k-symbolowy prefiks tekstu T oznaczamy przez T_k.

Używając tych oznaczeń, problem WW można wyrażać jako problem znajdywania wszystkich przesunięć $s \in < 0, n-m >$ takich, że P jest sufiksem T_{s+m} czyli takich, dla których $P \supset T_{s+m}$.

Za elementarną operację można w algorytmach WW uważać porównanie dwóch tekstów o tej samej długości lub porównanie symboli. Jeśli teksty są porównywane od strony lewej do prawej i proces porównania zatrzymuje się, gdy stwierdzimy niezgodność dwóch symboli, to przyjmujemy, że czas porównywania takiego tekstu jest proporcjonalny do liczby niezbędnych porównań symboli.

Dokładniej: test czy "$x = y$" wymaga czasu $\Theta(t+1)$, gdzie t jest długością najdłuższego słowa z będącego jednocześnie prefiksem słowa x i prefiksem słowa y. Słowo z jest więc najdłuższym słowem spełniającym relacje: $z \subset x$ i $z \subset y$.

Krótki przegląd algorytmów wyszukiwania wzorca:

1.Algorytm „naiwny” .Algorytm naiwny ma złożoność czasowa pesymistyczną $O((n-m+1)\cdot m)$.

2.Algorytm Rabina–Karpa (algorytm RK).Złożoność czasowa pesymistyczna tego algorytmu jest taka sama jak w przypadku algorytmu naiwnego czyli $O((n-m+1)\cdot m)$, ale złożoność średnia jest znacznie mniejsza.

3.Algorytm wyszukiwania wzorca wykorzystujący automat skończony. Algorytm ten rozpoczyna działanie od konstrukcji automatu skończonego szukającego wystąpień wzorca P w tekście T. Algorytm ten działa w czasie $O(n+m\cdot|V|)$. Czas działania algorytmu, gdy automat jest już zbudowany wynosi $\Theta(n)$. Natomiast czas konstrukcji automatu jest równy $O(m\cdot|V|)$.

4.Algorytm Knutha-Morrisa-Pratta (algorytm KMP) (działa w czasie $O(n+m)$).

5.Algorytm Boyera – Moore’a (algorytm BM) (działa w czasie $O((n\lg m)/m)$).

Dla porządku należy jeszcze wspomnieć o algorytmie Karpa-Millera-Rosenberga (algorytmie KMR) i algorytmie Fishera-Patersona (algorytmie FP). W dalszym ciągu opiszemy tylko pierwsze 4 algorytmy wyszukiwania wzorca: naiwny, RK, automatowy i KMP.

Warto jeszcze zwrócić uwagę na fakt, że zajmujemy się tu głównie jednowymiarowym problemem wyszukiwania wzorca ale niektóre z algorytmów WW dają się łatwo uogólnić na przypadek 2 i trójwymiarowy.

Omawiane w tym rozdziale algorytmy wyszukiwania wzorca dają się w naturalny sposób zrównoleglić.

Algorytmy wyszukiwania wzorca należą do kategorii tzw. algorytmów tekstowych. Inne algorytmy tekstowe to np.

- algorytmy obliczania najdłuższego wspólnego podsłowa
- algorytmy wyszukiwanie słów symetrycznych (tzw. palindromów)
- algorytmy wyszukiwanie tzw. słów cyklicznie równoważnych

Również algorytmy kompresji bezstratnej tekstów jak np. algorytm Huffmana zaliczane są do algorytmów tekstowych.

4.2 Algorytm naiwny wyszukiwania wzorca

Istota rzeczy. Algorytm „naiwny” znajduje wszystkie poprawne przesunięcia metodą najprostszą z możliwych. Jest to jedna pętla o $n-m+1$ obiegach, w której sprawdza się dla przesunięć $s = 0,1,...,n-m$ warunek równości z wzorcem czyli warunek

$$P[1..m] = T[s+1..s+m]$$

Czas sprawdzania czy równość $P[1..m] = T[s+1..s+m]$ zachodzi zależy od danych wejściowych i ma charakter losowy.

Algorytm: Algorytm naiwny wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$

Dane WYJŚCIOWE: ciąg poprawnych przesunięć, jeśli takie są.

---

procedure *naive_string_matcher*(*T,P*)

begin

for $s := 0$ **to** $n-m$ **do**
  if $P[1..m] = T[s+1..s+m]$ **then** *write*(„Wzorzec występuje z przesunięciem”, *s*)

end *naive_string_matcher*;

Powyższy algorytm w Matlabie można zapisać następująco:

Algorytm: Algorytm naiwny wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$

Dane WYJŚCIOWE: ciąg poprawnych przesunięć, jeśli takie są.

```
function lpp=naive_string_matcher(T,P)
% wartosc lpp podaje liczbe poprawnych przesuniec
k=0; M=size(P); m=M(2); N=size(T); n=N(2);
%przesuwamy wzorzec P i sprawdzamy czy pasuje
%do tekstu T czyli szukamy poprawnych przesuniec
for s=1:n-m+1, if P==T(s:s+m-1); k=k+1;
disp(strcat('poprawne przesuniecie = ',num2str(s))),
end, end
lpp=k
```

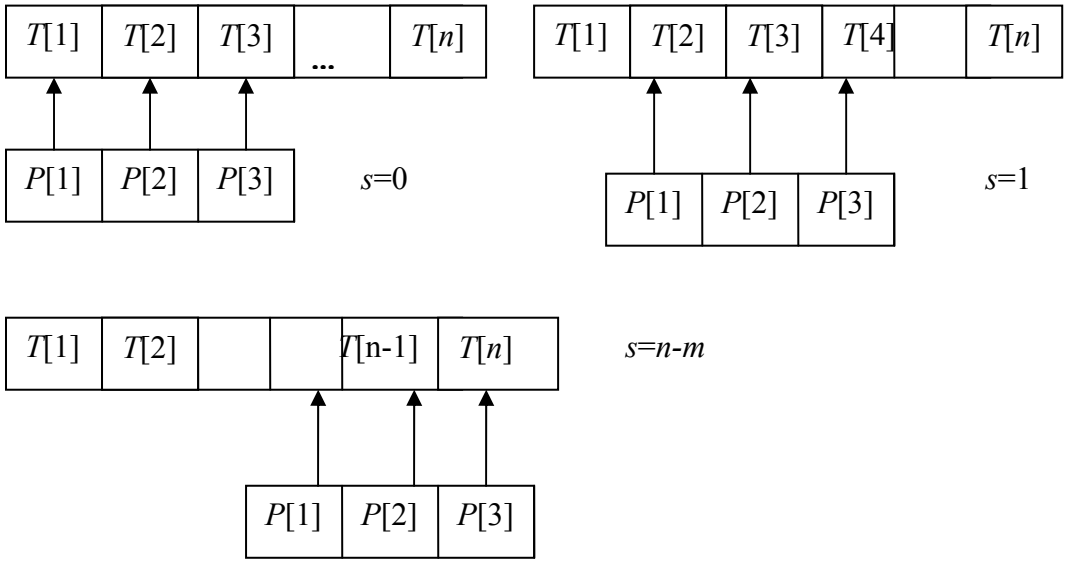
Procedura *naive_string_matcher* działa w czasie pesymistycznym $\Theta\big((n-m+1)\cdot m\big)$. Tak właśnie będzie dla tekstu $\underbrace{aa...a}_{n} = a^n$ i wzorca $\underbrace{aa...a}_{m} = a^m$. Jeśli więc $m = \lfloor n/2 \rfloor$ to pesymistyczny czas wykonania wynosi $\Theta(n^2)$.

Algorytm *naive_string_matcher* nie jest optymalny, istnieją algorytmy o czasie pesymistycznym działania $O(n+m)$.

„Naiwne" szukanie wzorca jest nieefektywne, ponieważ informacja uzyskana dla jednej wartości *s* przesunięcia wzorca jest całkowicie ignorowana dla następnych wartości *s*. Informacja taka jak łatwo zauważyć może mieć pewną wartość.

Przykład. Jeśli $P = aaab$ i stwierdzamy, że przesunięcie $s = 0$ jest poprawne, to żadne z przesunięć 1,2,3 nie jest poprawne.

Istotą lepszych algorytmów jest wykorzystanie informacji z poprzednich sprawdzeń dopasowania wzorca.



Rys. 4.1 Ilustracja działania naiwnego algorytmu wyszukiwania wzorca *naive_string_matcher* Sprawdzamy zgodność z tekstem *T* przesuwającego się ze skokiem 1 okienka o szerokości *m* zawierającego wzorzec *P* . Pierwsza niezgodność symboli kończy sprawdzenie.

Fakt. Niech tekst $T[1..n]$ będzie generowany przez ciąg niezależnych zmiennych losowych $X_1, X_2, ..., X_n$ określonych na przestrzeni probabilistycznej $(\Omega, \mathfrak{M}, P)$. Zakładamy, że zmienne losowe $X_1, X_2, ..., X_n$ mają wartości w $d \geq 2$ elementowym alfabecie *V* i rozkłady równomierne. Załóżmy ponadto, że wzorzec $P[1..m]$ jest generowany przez ciąg

niezaleznych zmiennych losowych $Y_1, Y_2, ..., Y_m$; $n \geq m$ o rozkładach równomiernych i wartościach w alfabecie V. Jeśli σ ciała $\sigma(T_1, T_2, ..., T_n)$ i $\sigma(P_1, P_2, ..., P_m)$ są niezależne to zmienna losowa Z_s (gdzie $s \in < 0, n-m >$) o wartościach w zbiorze $< 1, m >$ określona na przestrzeni probabilistycznej $(\Omega, \mathfrak{M}, P)$ wzorem

$Z_s = 1$ wtedy i tylko wtedy gdy $X_s \neq Y_1$

$Z_s = 2$ wtedy i tylko wtedy gdy $X_s = Y_1$ i $X_{s+1} \neq Y_2$

$Z_s = 3$ wtedy i tylko wtedy gdy $X_s = Y_1$ i $X_{s+1} = Y_2$ i $X_{s+2} \neq Y_3$

...

$Z_s = m-1$ wtedy i tylko wtedy gdy $X_s = Y_1$, $X_{s+1} = Y_2$, ...i $X_{s+m-2} = Y_{m-1}$ i $X_{s+(m-1)} \neq Y_m$

$Z_s = m$ wtedy i tylko wtedy gdy $X_s = Y_1$ i $X_{s+1} = Y_2$ i....i $X_{s+(m-1)} = Y_m$

ma wartość oczekiwaną równą

$$E(Z_s) = \frac{1-d^{-m}}{1-d^{-1}} \qquad (*)$$

Inaczej mówiąc wartość oczekiwana ilości porównań par pojedynczych symboli przy porównaniu $P[1..m]$ i $T[s+1..s+m]$ dla ustalonego $s \in < 0, n-m >$ jest dana wzorem (*).

Dowód. Dowód, że Z_s jest zmienną losową i Z_s ma wartość oczekiwaną pozostawiamy Czytelnikowi. Bezpośrednio z definicji Z_s wynika, że dla $k = 1,2,..., m-1$ mamy

$$P(Z_s = k) = d^{-(k-1)}(1-d^{-1})$$

oraz $P(Z_s = m) = d^{-m}$. Zatem wartość oczekiwana $E(Z_s)$ jest równa

$$E(Z_s) = 1 \cdot (1-d^{-1}) + 2d^{-1}(1-d^{-1}) + ... + md^{-(-1)}(1-d^{-1}) + md^{-m} =$$
$$= (1-d^{-1})(1 + 2d^{-1} + 3d^{-2} + ... + md^{-(m-1)}) + md^{-m}$$

Żeby obliczyć wartość powyższego wyrażenia najpierw obliczymy sumę

$$1 + 2d^{-1} + 3d^{-2} + ... + md^{-(m-1)} \quad (**)$$

Ogólny wzór na sumę $1 + 2x + ... + mx^{(m-1)}$, gdzie $x \in R$, $x \neq 1$ jest taki:

$$1 + 2x + ... + mx^{(m-1)} = (1 + x + x^2 + ... + x^m)' = \left(\frac{x^{m+1} - 1}{x-1} \right)' = \frac{(m+1)x^m(x-1) - x^{m+1} + 1}{(x-1)^2} =$$

$$= \frac{mx^{m+1} - mx^m - x^m + 1}{(x-1)^2}$$

W wyprowadzeniu tego wzoru korzystamy ze wzoru na sumę postępu geometrycznego i wykorzystujemy wzór na różniczkowanie ilorazu funkcji z rachunku różniczkowego

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)\cdot g(x) - f(x)\cdot g'(x)}{(g(x))^2})$$

Podstawiając do powyższego wzoru $x = d^{-1}$ dostajemy następujące wyrażenie na (**)

$$1 + 2d^{-1} + 3d^{-2} + ... + md^{-(m-1)} =$$

$$= \frac{(1-d^{-1})}{(d^{-1}-1)^2}\cdot(md^{-(m+1)} - md^{-m} - d^{-m} + 1) = \frac{1}{1-d^{-1}}(md^{-(m+1)} - md^{-m} - d^{-m} + 1)$$

i ostatecznie poszukiwana wartość oczekiwana jest równa:

$$\frac{1}{1-d^{-1}}(md^{-(m+1)} - md^{-m} - d^{-m} + 1) + m\cdot d^{-m} = \frac{1-d^{-m}}{1-d^{-1}}$$

■

Uwaga. Jest nieco zaskakujące, że wartość oczekiwana $E(Z_s)$ jest mała np.: dla $d = 2$ i dowolnego $m \geq 1$ mamy:

$$\frac{1-d^{-m}}{1-d^{-1}} = 2(1-\frac{1}{2^m}) = 2 - \frac{1}{2^{m-1}} < 2$$

co oznacza, że średnio po mniej niż 2 porównaniach symboli przerywamy sprawdzanie dopasowania wzorca. Ogólnie dla dowolnej liczby symboli w alfabecie $d \geq 2$ i dowolnego $m \geq 1$ mamy

$$\frac{d}{d-1}(1-d^{-m}) < \frac{d}{d-1}$$

4.3 Algorytm Rabina-Karpa

Algorytm WW opisany poniżej został zaproponowany przez Rabina i Karpa i nosi w literaturze nazwę algorytmu Rabina – Karpa lub algorytmu RK. Istotą algorytmu Rabina–Karpa jest przyporządkowanie porównywanym tekstom pewnych liczb i porównywanie tych liczb zamiast porównywania tekstów.

Zalety algorytmu Rabina – Karpa są następujące.

1. Algorytm Rabina – Karpa bardzo dobrze sprawdza się w praktyce i daje się łatwo zmodyfikować tak, by stanowił algorytm rozwiązywania pokrewnych do WW problemów takich jak np.: wyszukiwanie wzorca dwuwymiarowego w dwuwymiarowej tablicy lub trójwymiarowego w trójwymiarowej tablicy.

2. Pesymistyczny czas działania algorytmu wynosi $O((n-m+1)m)$, ale średni czas jest znacznie mniejszy niż w algorytmie "naiwnym".

W algorytmie Rabina-Karpa korzystamy z pojęcia przystawania dwóch liczb całkowitych modulo, czyli z tzw. kongruencji. Mówimy, że dwie liczby całkowite a i b przystają do siebie modulo q (gdzie $q \in N$, $q \geq 2$) jeśli reszty dzielenia a i b przez q są takie same. Zapisujemy ten fakt w postaci $a \equiv b(\mathrm{mod}\, q)$. Przypomnijmy jeszcze, że podobnym symbolem $a \pmod q$ oznaczamy resztę z dzielenia a przez q.

Jak wiadomo kongruencje możemy dodawać, odejmować, mnożyć stronami przez stałą całkowitą i przez siebie tzn. jeśli $a \equiv b(\mathrm{mod}\, q)$ oraz $c \equiv d \pmod q$ to $a + c \equiv b + d \pmod q$, $a - c \equiv b - d \pmod q$, $a \cdot c \equiv b \cdot d \pmod q$ oraz $k \cdot a \equiv k \cdot b \pmod q$ dla każdego $k \in Z$.

Niech d będzie liczbą naturalną taką, że $d \geq 2$ i rozważmy alfabet V o d elementach czyli taki, że card $V = d$. Dla uproszczenia będziemy utożsamiać symbole tego alfabetu z liczbami $0,1,2,...,d-1$. Będziemy te liczby traktować jednocześnie jako cyfry w naturalnym zapisie o podstawie d (lub jak czasem mówimy z wagą d).

Możemy teraz patrzeć na tekst złożony z k symboli jak na k cyfrową liczbę ze zbioru $N \cup \{0\}$ zapisaną w zapisie naturalnym wagowym o podstawie d.

Niech p będzie dla danego wzorca $P[1..m]$ liczbą przyporządkowaną słowu $P[1..m]$. Podobnie niech t_s dla $s = 0,1,...,n-m$ oznacza (w kodzie naturalnym wagowym z wagą d) liczbę ze zbioru $N \cup \{0\}$ odpowiadającą słowu $T[s+1,..s+m]$.

Oczywiście prawdziwe jest stwierdzenie, że

$$t_s = p \text{ wtedy i tylko wtedy, gdy } T[s+1,..s+m] = P[1..m]$$

Inaczej mówiąc

s jest poprawnym przesunięciem wtedy i tylko wtedy, gdy $t_s = p$

Zauważmy, że

I. Liczbę *p* możemy obliczyć w czasie $O(m)$. Istotnie za operacją dominującą można przyjąć tu mnożenie. Do obliczenia wartości p można wykorzystać schemat Hornera obliczania wartości wielomianu $a_n x^n + a_{n-1}x^{n-1} + ...a_1 x + a_0$ dla ustalonego *x*. Stosując schemat Hornera dostajemy:

$$p = P[1]d^{m-1} + P[2]d^{m-2} + ... + P[m-1]d + P[m] =$$

$$= (...((P[1] \cdot d + P[2]) \cdot d + P[3]) \cdot d + ... + P[m-1]) \cdot d + P[m]$$

Zliczając operacje mnożenia dostajemy *m*-1 mnożeń a więc istotnie liczbę *p* możemy obliczyć w czasie $O(m)$.

II. Podobnie znając $T[1..m]$ wartość t_0 odpowiadającą $T[1..m]$ obliczamy w czasie $O(m)$.

$$t_0 = T[1]d^{m-1} + T[2]d^{m-2} + ... + T[m-1]d + T[m] =$$

$$= (...(([\,1] \cdot d + P[2]) \cdot d + P[3]) \cdot d + ... + P[m-1]) \cdot d + P[m]$$

III. Zauważmy, że możemy obliczyć wszystkie pozostałe wartości $t_1, t_2, ..., t_{n-m}$ w sumie w czasie $O(n)$ na zasadzie, że jak przesuniemy „okno porównania o długości *m*" nad tekstem $T[1..n]$ to "trochę przybywa", "trochę ubywa", ale ze „starego" t_s możemy łatwo policzyć t_{s+1}.

$$t_{s+1} = d\left(t_s - d^{m-1}T(s+1)\right) + T[s+m+1]$$

Tak więc liczby $p, t_0, t_1, ..., t_{n-m}$ mogą być obliczone w czasie $O(n+m)$ i jeśli pominie się koszt porównania liczb to czas wykonania algorytmu byłby $O(n+m)$.

Jednak obliczone liczby $p, t_0, t_1, ..., t_{n-m}$ mogą być bardzo duże a operowanie dużymi liczbami to na ogół duży kłopot!

Metoda ratunku jest prosta. Obliczamy p i t_s modulo $q \in N$ dla $q \geq 2$ dla odpowiednio dobranej wartości q. Liczbę q dobieramy tak, aby $d \cdot q$ mieściło się w słowie maszynowym komputera co umożliwia łatwe dzielenie przez q i sprawdzanie równości liczb jednym rozkazem assemblera. Na przykład w przypadku mikroprocesorów rodziny Intel 80xxx możemy porównywać liczby rozkazami : xor ax, bx ; xor eax, ebx ; test ax, bx; czy cmp ax, bx. Mamy też do dyspozycji odpowiednie rozkazy koprocesora operujące na słowach 80 bitowych i 128 bitowe instrukcje SSE.

Zatem obliczamy

$$t_{s+1} = \left(d(t_s - T[s+1]h) + T[s+m+1]\right) (\mathrm{mod}\, q), \text{ gdzie } \; h = d^{m-1} (\mathrm{mod}\, q).$$

Uwaga. Działania w arytmetyce modulo q mogą prowadzić do błędu ponieważ.

$$t_s \equiv p \pmod{q} \text{ nie implikuje, że } t_s = p$$

Inaczej mówiąc równość reszt z dzielenia przez q nie implikuje równości dzielnych.

Z drugiej strony, jeśli $t_s \pmod{q} \neq p \pmod{q}$ to na pewno $t_s \neq p$ i przesunięcie jest niepoprawne!

Nasz test może zawodzić jeśli $t_s \equiv p (\mathrm{mod}\, q)$. Musimy wówczas sprawdzić czy rzeczywiście mamy równość i wówczas testujemy bezpośrednio czy zachodzi równość $P[1..m] = T[s+1..s+m]$.

W zasadzie q może być dowolną liczbą całkowitą taką, że $q \geq 2$ Jeśli q jest dostatecznie duże, to możemy się spodziewać, że dodatkowa weryfikacja będzie raczej rzadka. Dokładniej, liczba „złych trafień" czyli sytuacji pozornej zgodności fragmentu tekstu i wzorca tzn. takich. że $t_s \neq p$ ale

$$t_s \pmod{q} = p \pmod{q}$$

będzie tym mniejsza im q jest większe. Z kolei zbyt duże q utrudnia obliczenia. Przecież dlatego właśnie wprowadziliśmy obliczenia modulo q.

Zakładaliśmy w opisie algorytmu Rabina-Karpa, że $d = \mathrm{card}\, V$ ale algorytm pozostaje poprawny również jeśli $\mathrm{card}\, V \leq d$.

Algorytm: Algorytm Rabina –Karpa wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$, d liczba znaków alfabetu V, q ustalona liczba całkowita $q \geq 2$ oraz $m \leq n$

Dane WYJŚCIOWE: ciąg poprawnych przesunięć, jeśli takie są.

---

procedure *Rabin-Karp_string_matcher*(*T,P,d,q*)

begin
$h := d^{m-1} \pmod q$;
$p := 0$;
$t_0 := 0$;
for $i := 1$ **to** m **do** **begin** $p := (d \cdot p + P[i]) \pmod q$;

$t_0 := (d \cdot t_0 + T[i]) \pmod q$
end

{Stosujemy schemat Hornera do obliczenia t_0 i p}

for $s := 0$ **to** $n - m$ **do**

begin
if $p = t_s$ **then if** $P[1..m] = T[s+1..s+m]$ **then** write("Wzorzec wystepuje przesunięciem"; s)

if $s < n - m$ **then** $t_{s+1} := \left(d(t_s - h \cdot T[s+1]) + T(s+m+1)\right) \pmod q$
end

end *Rabin-Karp_string_matcher*;

W Matlabie powyższy algorytm można zapisać następująco

Algorytm: Algorytm Rabina –Karpa wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$, d liczba znaków alfabetu V, q ustalona liczba całkowita $q \geq 2$ oraz $m \leq n$
Dane WYJŚCIOWE: ciąg poprawnych przesunięć, jeśli takie są.

```
function lpp=Rabin_Karp_string_matcher(T,P,d,q)
% wartosc lpp podaje liczbe poprawnych przesuniec
k=0; h=1; M=size(P); m=M(2);N=size(T); n=N(2);

% obliczamy najpierw wartosc h
if m==1, h=1; else for i=1:m-1, h=mod(h*d,q); end, end

% stosujemy schemat Hornera do obliczenia t0 i p
p=0; t0=0;
for i=1:m, p=mod(d*p+P(i),q); t0=mod(d*t0+T(i),q); end

% przesuwamy wzorzec P i sprawdzamy czy pasuje
%do tekstu T czyli szukamy poprawnych przesuniec
for s=1:n-m+1,
    if P==T(s:s+m-1); k=k+1;
    disp(strcat('poprawne przesuniecie = ',num2str(s))),
    end,
    if s<n-m+1, t0=mod(d*(t0-h*T(s))+T(s+m),q); end
end
lpp=k
```

4.4. Algorytm wyszukiwania wzorca wykorzystujący automat skończony

Naturalnym narzędziem do przetwarzania tekstu są automaty skończone uzasadniona jest więc próba zastosowania teorii automatów skończonych do problemu wyszukiwania wzorca.

Automaty skończone. Automat skończony M to uporządkowana piątka $(Q, \Sigma, A, \delta, q_0)$, gdzie

Q - to skończony zbiór nazywany zbiorem stanów automatu M

Σ - to skończony zbiór nazywany alfabetem wejściowym automatu M

$A \subseteq Q$ - jest tzw. zbiorem stanów akceptujących

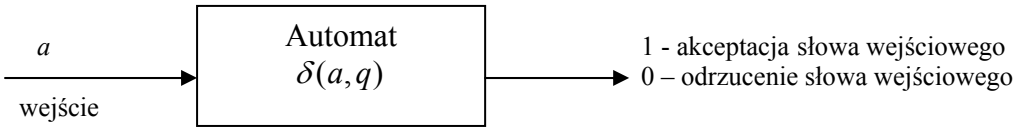
$\delta : Q \times \Sigma \to Q$ - jest tzw. funkcją przejść automatu M

$q_0 \in Q$ - jest wyróżnionym stanem tzw. stanem początkowym automatu M

Tak zdefiniowany automat posłuży nam do zdefiniowania pewnego języka $L(M) \subseteq \Sigma^*$.

Automat skończony $M = (Q, \Sigma, A, \delta, q_0)$ rozpoczyna pracę w stanie q_0 a następnie w kolejnych chwilach 1,2,... wczytuje kolejne symbole słowa wejściowego. Jeśli automat jest w stanie $q \in Q$ (wyszedł z takim stanem z chwili poprzedniej) i czyta symbol wejściowy $a \in \Sigma$, to przechodzi on w danej chwili ze stanu q do stanu $\delta(q, a)$.

Jeśli bieżący stan $q \in A$ to mówimy, że automat M **akceptuje** wczytany dotychczas tekst (czyli słowo wejściowe). Tekst, który nie jest akceptowany jest „odrzucony" przez automat M. Słowa akceptowane zaliczamy do języka $L(M) \subseteq \Sigma^*$. Automat M jest więc narzędziem, mechanizmem do sprawdzenia czy dane słowo należy do języka $L(M)$.



Rys. 4. 1 Automat jako urządzenie do akceptacji lub odrzucenia słowa wejściowego.

Dla danego automatu $M = (Q, \Sigma, A, \delta, q_0)$ wprowadzamy funkcję $\Phi : \Sigma^* \to Q$ tzw. **rozszerzoną funkcję przejść**. Definiujemy ją tak: dla danego $w \in \Sigma^*$, $\Phi(w)$ jest stanem, do którego przechodzi automat M po wczytaniu słowa w.

Zatem automat *M* akceptuje słowo *w* wtedy i tylko wtedy, gdy $\Phi(w) \in A$.

Rozszerzona funkcja przejść jest zdefiniowana rekurencyjnie:

$$\Phi(\varepsilon) = q_0$$

$$\Phi(wa) = \delta(\Phi,(w),a) \text{ dla } w \in \Sigma^*, a \in \Sigma.$$

Przykład. Opiszemy prosty automat skończony $M = (Q, \Sigma, A, \delta, q_0)$. Niech zbiór stanów $Q = \{0,1\}$. Automat jest więc dwustanowy. Przyjmijmy, że stanem początkowym jest 0 tzn. $q_0 = 0$ oraz alfabet wejściowy jest dwuliterowy $\Sigma = \{a, b\}$. Załóżmy ponadto, że funkcja przejścia automatu $\delta : Q \times \Sigma \to Q$ opisana jest tabelką:

	a	**b**	← wejście
0	1	0	
1	0	0	

↑
stan

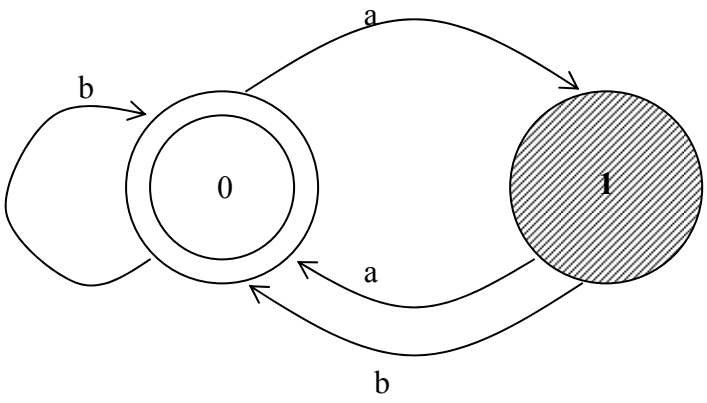
Niech $A = \{1\}$ -będzie zbiorem stanów akceptujących. Zauważmy, że automat ten akceptuje zbiór słów kończących się nieparzystą liczbą symboli a np.: akceptuje słowa: *bbaaa, abaaaaa.*

Dokładniej, słowo $x \in \Sigma^*$ jest akceptowane wtedy i tylko wtedy, gdy jest postaci: $x = yz$ gdzie $y = \varepsilon$ lub y kończy się literą b oraz $z = a^k$, gdzie $k \in N$ jest liczbą nieparzystą.

Na przykład, jeśli na wejście automatu podajemy słowo *abaaa*, to kolejne stany z $q_0 = 0$ włącznie są takie: 0,1,0,1,0,1 a więc słowo *abaaa* jest akceptowane. Jeśli na wejście automatu podajemy słowo *abbaa*, to kolejne stany ze stanem $q_0 = 0$ włącznie są takie: 0,1,00,10 a więc automat odrzuca to słowo.

Wygodnie jest automat skończony opisywać za pomocą grafu skierowanego. Opiszemy nasz automat takim grafem. Stany będą reprezentowane węzłami (w kształcie koła). Skierowane krawędzie grafu będą opisane symbolami wejściowymi i reprezentują przejścia

pomiędzy stanami automatu. Wszystkie informacje o automacie skończonym są wpisane w taki graf. Stan początkowy oznaczony jest podwójnym okręgiem a stan akceptujący jest zakreślony.



Rys. 4.2 Opis automatu skończonego za pomocą grafu skierowanego. Krawędź od stanu 1 do stanu 0 etykietowana symbolem *b* odpowiada przejściu $\delta(1,b) = 0$.

Do wyszukiwania wzorca P w tekście $T[1...n]$ można użyć automatu skończonego, który przegląda tekst $T[1...n]$ w celu znalezienia wszystkich wystąpień wzorca P.

Dla danego wzorca P specjalnie wewnątrz algorytmu konstruuje się odpowiedni automat tzw. „automat wyszukiwania wzorca".

Automat wyszukiwania wzorca działa bardzo efektywnie. Każdy symbol tekstu badany jest dokładnie raz i czas na to poświęcony jest stały. Zatem czas działania algorytmu jest $\Theta(n)$ jeśli automat szukający wzorca P jest już zbudowany.

Czas konstruowania automatu może być jednak długi gdy alfabet Σ jest duży.

Sam algorytm wyszukiwania wzorca składa się więc z dwóch części:

1. konstrukcji automatu wyszukiwania wzorca dla danego wzorca $P[1...m]$
2. przejrzenia testu wejściowego $T[1...n]$ w celu wyłapania wszystkich wystąpień w nim wzorca $P[1...m]$

Automaty wyszukiwania wzorca

1.Dla ustalonego wzorca $P[1..m]$ zbudujemy automat wyszukiwania (dla tego wzorca).

2.Najpierw definiujemy tzw. funkcję sufiksową wzorca P. Jest to odwzorowanie $\sigma : \Sigma^* \to < 0, m >$ takie, że $\sigma(x)$ dla danego słowa $x \in \Sigma^*$ jest długością najdłuższego prefiksu wzorca P, który jest jednocześnie sufiksem x czyli

$$\sigma(x) \stackrel{df}{=} \max\{k \in N;\ P_k \supset x\}$$

Zbiór $\{k \in N;\ P_k \supset x\} \subset < 0, m >$ jest niepusty, ponieważ słowo puste ε $(P_0 = \varepsilon)$ jest sufiksem każdego słowa, zatem funkcja sufiksowa jest dobrze zdefiniowana. Intuicyjnie rzecz biorąc $\sigma(x)$ jest maksymalnym pasującym „nasunięciem od tyłu" wzorca P na słowo x.

Przykład. Dla wzorca $P = ab$ mamy $\sigma(\varepsilon) = 0, \sigma(ccab) = 2$. ■

Dla wzorca P o długości m mamy:

$\sigma(x) = m$ wtedy i tylko wtedy, gdy $P \supset x$

Dla funkcji sufiksowej prawdziwy jest następujący fakt: jeśli słowo x jest sufiksem słowa y czyli $x \supset y$ to $\sigma(x) \le \sigma(y)$.

Automat wyszukiwania wzorca definiujemy tak:

1. Zbiór stanów $Q = \{0,1,...,m\}$, gdzie m jest długością wzorca P

2. Za stan początkowy przyjmujemy $q_0 = 0$

3.Zbiór stanów akceptujących składa się z jednego elementu m

4.Funkcję przejść $\delta : Q \times \Sigma \to Q$ definiujemy za pomocą funkcji sufiksowej tak:

dla każdego stanu $q \in < 0, m >$ i symbolu $a \in \Sigma$ z definicji funkcja przejścia zadana jest wzorem

$$\delta(q, a) = \sigma(P_q a)$$

Okazuje się, że przy takiej definicji funkcji przejść automatu prawdziwa jest dla każdego tekstu wejściowego $T[1..n] \in \Sigma^*$ równość:

$$\Phi(T_i) = \sigma(T_i)$$

co oznacza, że stan automatu (w każdym kroku wczytywania tekstu $T[1..n]$) pokazuje najlepsze dopasowanie sufiksu dotychczas wczytywanego tekstu T_i do prefiksu wzorca P. Powyższy fakt zostanie w dalszym ciągu udowodniony.

Jeśli mamy już skonstruowany dla wzorca P automat tzn. znamy funkcję przejścia $\delta : Q \times \Sigma \to Q$ to algorytm automatowy wyszukiwania wzorca jest następujący.

Algorytm: Algorytm automatowy wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$, $\delta : Q \times \Sigma \to Q$ funkcja przejścia automatu wyszukiwania wzorca, m długość wzorca

Dane WYJŚCIOWE: ciąg poprawnych przesunięć, jeśli takie są.

---

procedure *finite_automaton_matcher* (T, δ, m)

begin $q := 0$; {ustawienie stanu początkowego}

for $i := 1$ **to** n **do** { wczytywanie kolejnych symboli tekstu wejściowego}

begin

$q := \delta(q, T[i])$; {obliczanie „maksymalnego dopasowania”}

if $q = m$ **then begin** $s := i - m$;
write („Wzorzec występuje z przesunięciem”, s)
end ;

end ;
end *finite_automaton_matcher* ;

Ten sam algorytm można zapisać w Matlabie następująco:

Algorytm: Algorytm automatowy wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$, $\delta : Q \times \Sigma \to Q$ funkcja przejścia automatu wyszukiwania wzorca, m długość wzorca

Dane WYJŚCIOWE: ciąg poprawnych przesunięć, jeśli takie są.

---

function lpp=finite_automaton_matcher(T,P,Sig)

```
% Wartosc lpp podaje liczbe poprawnych przesuniec
%  Sig jest wektorem reprezenującym alfabet “sigma duże”

q=0; k=0; N=size(T); n=N(2), M=size(P); m=M(2);

% obliczanie funkcji przejscia automatu dopasowania wzorca
delta=compute_transition_function(P,Sig);

for i=1:n, q=delta(q+1,double(T(i))); if q==m, s=i-m; k=k+1;
disp(strcat('poprawne przesuniecie =',num2str(s))), end, end
lpp=k
```

Lemat (nierówność dla funkcji sufisowej)
Dla każdego słowa $x \in \Sigma^*$ i symbolu $a \in \Sigma$ zachodzi nierówność

$$\sigma(xa) \le \sigma(x) + 1 \qquad (*)$$

Dowód. Powyższy fakt jest oczywisty, gdyby bowiem nierówność(*) nie była prawdziwa czyli $\sigma(xa) > \sigma(x) + 1$, to oznaczałoby to zgodność sufiksu x i wzorca P na większej liczbie pozycji niż $\sigma(x)$, co jest sprzeczne z definicją funkcji sufiksowej σ.■

Twierdzenie: (wzór rekurencyjny dla funkcji sufiksowej)
Niech P będzie ustalonym wzorcem. Dla każdego słowa $x \in \Sigma^*$ i symbolu $a \in \Sigma$ jeśli oznaczymy $q = \sigma(x)$ to:

$$\sigma(xa) = \sigma(P_q a)$$

Dowód. Jeśli $\sigma(x) = q$ to wynika stąd, że $P_q \supset x$ oraz $P_q a \supset xa$. Widać więc jaki jest sufiks słowa xa w takiej sytuacji.

Oznaczmy $r = \sigma(xa)$, z lematu o nierówności dla funkcji sufiksowej mamy $r \le q + 1$. Ponieważ $P_q a \supset xa$ i $P_r \supset xa$ oraz $|P_r| \le |P_q a|$ więc z lematu o nierozłącznych sufiksach z podrozdziału 4.1 mamy $P_r \supset P_q$ a stąd wynika, że $r \le \sigma(P_q a)$ czyli $\sigma(xa) \le (P_q a)$.
Jednocześnie ponieważ $P_q a \supset xa$ to $\sigma(P_q a) \le \sigma(xa)$.
Ostatecznie więc otrzymujemy: $\sigma(xa) = \sigma(P_q a)$.■

Istota rzeczy to sprowadzenie obliczenia wartości funkcji sufiksowej $\sigma(xa)$ do obliczenia $\sigma(x)$ i $\sigma(P_q a)$.

Z powyżej podanych faktów wynika prawdziwość zasadniczego twierdzenia, z którego wynika poprawność opisywanego algorytmu wyszukiwania wzorca.

Twierdzenie:
Niech Φ będzie rozszerzoną funkcją przejścia automatu wyszukiwania wzorca, $P[1..m]$ wzorcem a $T[1..n]$ tekstem wejściowym dla automatu. Wówczas dla każdego $i = 0,1,...,n$ mamy:

$$\Phi(T_i) = \sigma(T_i)$$

Uwaga. Powyższa równość oznacza, że automat wyszukiwania wzorca w każdym kroku przy analizie tekstu wejściowego $T[1..n]$, pamięta długość najdłuższego prefiksu wzorca będącego sufiksem dotychczas wczytanego tekstu.

Dowód. Dowód to zwykły dowód przez indukcję. Dla $i = 0$ twierdzenie oczywiście zachodzi, bo $T_0 = \varepsilon$ i mamy $\Phi(T_0) = 0$ oraz $\sigma(T_0) = 0$.

Zróbmy założenie indukcyjne, że $\Phi(T_i) = \sigma(T_i)$, wykażemy, że $\Phi(T_{i+1}) = \sigma(T_{i+1})$. Oznaczmy dla uproszczenia $\Phi(T_i)$ przez q i $T[i+1]$ przez a. Korzystając z definicji rozszerzonej funkcji przejścia Φ i z definicji funkcji przejścia mamy w dalszym ciągu:

$$\Phi(T_{i+1}) = \Phi(T_i a) = \delta(\Phi(T_i), a) = \delta(q, a) = \sigma(P_q a) =$$

Z poprzedniego twierdzenia o wzorze rekurencyjnym dla funkcji sufiksowej dostajemy teraz.

$$\sigma(P_q a) = \sigma(T_i a) = \sigma(T_{i+1})$$

■

Powyższe twierdzenie jest twierdzeniem o poprawności procedury *finite_automaton_matcher*. Mówi ono bowiem, że jeśli automat dopasowania wzorca przechodzi do stanu q to, to jest największa taka wartość q, że $P_q \supset T_i$. Jeśli więc $q = m$ to wystąpienie wzorca P zostało wykryte.

Obliczanie funkcji przejść $\delta : Q \times \Sigma \to Q$ (konstrukcja odpowiedniej macierzy).

Poniższa procedura oblicza funkcję przejścia $\delta : Q \times \Sigma \to Q$ dla ustalonego wzorca $P[1..m]$.

Algorytm: Algorytm obliczający funkcję przejścia automatu wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, m długość wzorca, alfabet wejściowy $\Sigma = \{a_1, a_2, ..., a_r\}$

Dane WYJŚCIOWE: wartości funkcji przejścia automatu wyszukiwania wzorca $\delta : Q \times \Sigma \to Q$ umieszczone w tablicy δ

---

procedure *compute_transition_function*(P, Σ)

begin **for** $q := 0$ **to** m **do**
for $i := 1$ **to** r **do**
begin
$k := \min(m+1, q+2)$;
repeat $k := k-1$ **until** $P_k \supset P_q a_i$;
$\delta(q, a_i) := k$;
end
end *compute_transition_function*;

W Matlabie powyższy algorytm można zapisać następująco

W Matlabie powyższy algorytm można zapisać następująco

Algorytm: Algorytm obliczający funkcję przejścia automatu wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, m długość wzorca, alfabet wejściowy $\Sigma = \{a_1, a_2, ..., a_r\}$

Dane WYJŚCIOWE: wartości funkcji przejścia automatu wyszukiwania wzorca $\delta : Q \times \Sigma \to Q$ umieszczone w tablicy δ

```
function delta=compute_transition_function(P,Sig)
% Funkcja konstruuje funkcje (macierz) przejsc automatu dopasowania wzorca
% Sig to wektor znakowy, ktory utozsamiamy z alfabetem wejsciowym duze sigma

M=size(P); m=M(2); S=size(Sig); r=S(2);
```

```
for q=1:m+1,
for i=1:r, k=min(m,q)
if q==1, pom=Sig(i); else pom=strcat(P(1:q-1),Sig(i)); end
while (k>=1)&~(P(1:k)==pom(q-k+1:q)), k=k-1; end
aski_number=double(Sig(i))
tr_fun(q,aski_number)=k
```

Istota rzeczy: pracujemy na sufiksie $P_q a$ usiłując zmaksymalizować długość dopasowania $P_q a$ do wzorca P a do konstrukcji funkcji przejść nie jest nam potrzebny tekst wejściowy $T[1..n]$.

Procedura obliczania $\delta(q,a)$ korzysta bezpośrednio z definicji $\delta(q,a)$ i definicji funkcji sufiksowej σ.

$$\sigma(q,a) \stackrel{df}{=} \sigma(P_q a) \stackrel{df}{=} \max\{k \in N;\ P_k \supset P_q a\} \qquad (*)$$

Przebiegamy w pierwszych dwóch pętlach **for** wszystkie możliwe wartości stanów $q \in Q$ i symboli $a \in \Sigma$ i obliczamy wartość $\delta(q,a)$ korzystając z (*). Wewnątrz pętla **repeat** ...**until** rozpoczyna działanie z największą sensowną wartością $k = \min(m, q+1)$ a następnie zmniejsza k do momentu aż $P_k \supset P_q a$.

Widać więc, że czas działania algorytmu jest równy $O(m^3|\Sigma|)$, gdzie $|\Sigma|$ oznacza moc zbioru Σ. ($(m+1)\cdot|\Sigma|$ obiegów zewnętrznych dwóch pętli razy $(m+1)\cdot m$ obiegów pętli wewnętrznych).

Uwaga. Istnieje szybszy algorytm obliczania $\delta(q,a)$ działający w czasie $O(m|\Sigma|)$. Jeśli wykorzystuje się tę szybszą procedurę uzyskujemy czas działania pełnego automatowego algorytmu wyszukiwania wzorca równy $O(n+m|\Sigma|)$.

4.3. Algorytm Knutha-Morrisa-Pratta

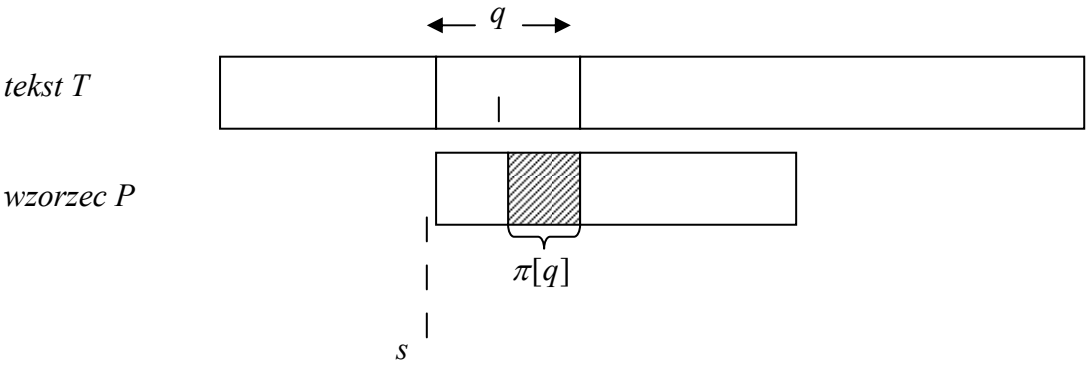
Algorytm Knutha-Morrisa-Pratta dopasowania wzorca w skrócie algorytm KMP wykorzystuje do przyśpieszania sprawdzania dopasowania wzorca P do tekstu T tzw. funkcję prefiksową dla wzorca P.

Definicja funkcji prefiksowej dla wzorca P jest następująca. Niech będzie dany ustalony wzorzec $P = P[1..m]$. Funkcja prefiksowa to odwzorowanie $\pi :< 1, m > \rightarrow < 0, m-1 >$ zdefiniowane wzorem:

$$\pi[q] \stackrel{df}{=} \max\{k \in Z;\ \ 0 \le k < q \ \text{ i } P_k \supset P_q\}$$

Wartość $\pi[q]$ jest więc z definicji maksymalną długością prefiksu wzorca P będącego właściwym sufiksem P_q. Oczywiście mamy $\pi[1] = 0$.

Istota rzeczy. Jeśli wiemy, że q symboli pasuje przy przesunięciu s, to następnym potencjalnie poprawnym przesunięciem jest $s' = s + (q - \pi[q])$. W algorytmie naiwnym wzorzec przesuwany jest pod tekstem zawsze o jeden znak. Można jak widać wykonywać dłuższe przesunięcia dysponując funkcją prefiksową.



Rys. 4.1 Ilustracja sposobu dokonywania dłuższego przesunięcia $s' = s + (q - \pi[q])$ w poszukiwaniu potencjalnie poprawnego przesunięcia. Przez q oznaczona jest liczba zgodnych znaków tekstu T i wzorca P przy przesunięciu s.

Pesymistyczna złożoność czasowa opisanego poniżej algorytmu *KMP* jest równa $\Theta(n)$ a pesymistyczna złożoność czasowa algorytmu obliczania funkcji prefiksowej π jest równa $\Theta(m)$. Analiza złożoności czasowej obu algorytmów zostanie przeprowadzona w dalszym ciągu podrozdziału.

Podany niżej algorytm *KMP* może być uważany za modyfikację algorytmu automatowego z poprzedniego podrozdziału.

Algorytm: Algorytm Knutha-Morrisa-Pratta wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$, m długość wzorca, n długość tekstu, $m \le n$

Dane WYJŚCIOWE: ciąg poprawnych przesunięć $s \in < 0, n-m >$, jeśli takie są.

---

procedure *Knuth_Morris_Pratt_matcher*(*T, P*)

begin *compute_prefix_function* (π, P);

{Obliczamy wstępnie funkcję prefiksową π dla wzorca *P*}

$q := 0$

for $i := 1$ **to** n **do begin**

while $q > 0$ i $P[q+1] \neq T[i]$ **do** $q := \pi[q]$;
if $P[q+1] = T[i]$ **then** $q := q+1$;
if $q = m$ **then** **begin**
write("Poprawne przesunięcie $s =$", $i-m$);
$q := \pi[q]$;
end ;
end ;

end *Knuth_Morris_Pratt_matcher*;

W Matlabie powyższy algorytm można zapisać następująco

Algorytm: Algorytm Knutha-Morrisa-Pratta wyszukiwania wzorca

Dane WEJŚCIOWE: wzorzec $P[1..m]$, tekst $T[1..n]$, m długość wzorca, n długość tekstu, $m \le n$

Dane WYJŚCIOWE: ciąg poprawnych przesunięć $s \in < 0, n-m >$, jeśli takie są.

```
function lpp=Knuth_Morris_Pratt_matcher(T,P)

% wartosc lpp jest liczba poprawnych przesuniec
% wstepnie obliczamy funkcje prefiksowa pi dla wzorca P

M=size(P); m=M(2); N=size(T); n=N(2);
pi=compute_prefix_function(P);
q=0; k=0;
for i=1:n,
   while (q>0)&(P(q+1)~=T(i)), q=pi(q); end;
   if P(q+1)==T(i), q=q+1; end;
   if q==m, disp(strcat('poprawne przesuniecie = ',num2str(i-m))), k=k+1;  q=pi(q); end
end;
lpp=k
```

Własności funkcji prefiksowej

Najprostszym algorytmem obliczania funkcji prefiksowej π dla ustalonego wzorca $P = P[1..m]$ byłby algorytm oparty bezpośrednio na definicji funkcji prefiksowej. Algorytm taki oparty na sprawdzaniu w podwójnej pętli **for** czy spełniona jest relacja $P_k \supset P_q$ miałby jednak pesymistyczną złożoność czasową $\Theta(m^2)$.

Zanim opiszemy znacznie szybszy (pracujący w czasie $\Theta(m)$) algorytm obliczania funkcji π podamy szereg ważnych własności funkcji prefiksowej.

Zauważmy, że bezpośrednio z definicji dla każdego $q \in <1, m>$ mamy $\pi[q] < q$. Iterowanie funkcji prefiksowej π prowadzi więc do utworzenia skończonego, malejącego ciągu liczb całkowitych nieujemnych kończącego się zerem. Oznaczmy *j*-tą iterację funkcji prefiksowej π (o ile taka iteracja istnieje) przez $\pi^{(i)}[q]$. Definiujemy więc $\pi^{(i)}[q]$ jako $\underbrace{\pi[\pi[\pi[...\pi(q)]...]}_{t}$.

$$\pi^{(i)}[q] \stackrel{df}{=} \underbrace{\pi[\pi[\pi[...\pi(q)]...]}_{t}$$

Możemy teraz dla każdego $q \in <1, m>$ zdefiniować zbiór $\pi^*[q]$ wzorem

$$\pi^*[q] \stackrel{df}{=} \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], ..., \pi^{(t)}[q]\}$$

Ciąg $(\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], ..., \pi^{(t)}[q])$ jest oczywiście malejący (bo dla każdego $q \in <1, m>$ mamy $\pi[q] < q$) i kończy się, gdy $\pi^{(t)}[q] = 0$. Liczba t czyli liczba iteracji do momentu osiągnięcia wartości 0 spełnia nierówność $t \leq q$ .Jednocześnie łatwo podać przykład taki, że $t = q$.

Lemat 1 (o iterowaniu funkcji prefiksowej)
Jeśli $P = P[1...m]$ jest wzorcem o długości m a π jest funkcją prefiksową dla tego wzorca, wtedy dla każdego $q \in <1, m>$ mamy:

$$\pi^*[q] = \{k \in <0, q>;\ k < q \quad \text{i} \quad P_k \supset P_q\}$$

Dowód. 1. Wykażemy, że z faktu $i \in \pi^*[q]$ wynika, że $P_i \supset P_q$ (co dowodzi inkluzji $\pi^*[q] \subseteq \{k \in <0, q>;\ k < q \quad \text{i} \quad P_k \supset P_q\}$).

Jeśli $i \in \pi^*[q]$ to (z definicji zbioru $\pi^*[q]$) istnieje takie $u \in <1, q>$, że $i = \pi^{(u)}[q]$.

Zauważmy, że dla każdego $q \in <1, m>$ mamy $P_{\pi(q)} \supset P_q$ zatem jeśli $u = 1$ to $P_i \supset P_q$ więc mamy to chcieliśmy wykazać.

Jeśli $u = 2$ to $P_{\pi(q)} \supset P_q$ oraz $P_{\pi(\pi(q))} \supset P_{\pi(q)}$ i z przechodniości relacji sufiksowej $\supset$ dostajemy $P_{\pi(\pi(q))} \supset P_q$ czyli również $P_i \supset P_q$.

Podobnie możemy rozumować dla dowolnego u takiego, że $i = \pi^{(u)}[q]$ otrzymując z faktu, że $P_{\pi(q)} \supset P_q$ i z przechodniości relacji sufiksowej:

$$P_i = P_{\pi^{(u)}(q)} \supset P_{\pi^{(u-1)}(q)} \supset ... \supset P_{\pi(\pi(q))} \supset P_{\pi(q)}$$

a więc $P_i \supset P_q$ dla dowolnego u takiego, że $i = \pi^{(u)}[q]$. Otrzymaliśmy więc to co chcieliśmy wykazać.

2. Pokazaliśmy, że zachodzi inkluzja $\pi^*[q] \subseteq \{k \in <0, q>;\ k < q \quad \text{i} \quad P_k \supset P_q\}$. Pozostaje pokazać, że mamy $\pi^*[q] \supseteq \{k \in <0, q>;\ k < q \quad \text{i} \quad P_k \supset P_q\}$.

Przeprowadzimy dowód nie wprost. Załóżmy, że istnieje takie $j \in < 0, q >;\ j < q$ i $P_j \supset P_q\}$ i takie, że $j \notin \pi^*[q]$. Możemy przyjąć, że j jest największą taką liczbą.

Oczywiście $j < \pi[q]$ (większe być nie może i równe też nie bo $j \notin \pi^*[q]$).

Weźmy najmniejszą liczbę w zbiorze $\pi^*[q]$ większą od j (no co najmniej jedna taka istnieje bo $j < \pi[q]$) i oznaczmy ją przez j'.

Mamy oczywiście $P_j \supset P_q$ ponieważ $j \in \{k \in < 0, q >;\ k < q$ i $P_k \supset P_q\}$ oraz $P_{j'} \supset P_q$ ponieważ $j' \in \pi^*[q]$. Teraz z lematu o nierozłącznych sufiksach mamy $P_j \supset P_{j'}$ (ponieważ $j' > j$). Jednocześnie *j* jest największą liczbą o tej własności mniejszą od j' więc z definicji funkcji prefiksowej mamy $\pi[j'] = j$.

Ponieważ jednak $j' \in \pi^*[q]$ więc również $j \in \pi^*[q]$ co jest sprzeczne z założeniem, że $j \notin \pi^*[q]$. Stwierdzamy więc, że istotnie zachodzi inkluzja $\pi^*[q] \supseteq \{k \in < 0, q >;\ k < q$ i $P_k \supset P_q\}$

3. Z p.1 i 2 dostajemy ostatecznie , że $\pi^*[q] = \{k \in < 0, q >;\ k < q$ i $P_k \supset P_q\}$ ■

Fakt 1 Jeśli $P = P[1...m]$ jest wzorcem o długości m oraz $\pi :< 1, m > \to < 0, m-1 >$ jest funkcją prefiksową dla P to dla każdego $q \in < 1, m >$ jeśli $\pi(q) > 0$ to mamy

$$\pi[q] - 1 \in \pi^*[q-1] \quad (*)$$

Dowód. Widać od razu, że dla $q = 1$ i $q = 2$ wzór (*) jest prawdziwy. Ogólnie, jeśli $r = \pi[q] > 0$, to oczywiście z definicji funkcji prefiksowej wynika, że $r < q$ i $P_r \supset P_q$. Zatem $r - 1 < q - 1$ i $P_{r-1} \supset P_{q-1}$ (pomijamy ostatni symbol w P_r i P_q). Z udowodnionego lematu 1 dostajemy, że $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$.■

Zdefiniujmy dla $q \in < 2, m >$ podzbiór $E_{q-1} \subset \pi^*[q-1]$ wzorem

$$E_{q-1} = \{k \in \pi^*[q-1]:\ P[k+1] = P[q]\}$$

Korzystając z lematu o iterowaniu funkcji prefiksowej dostajemy

$$E_{q-1} = \{k \in \pi^*[q-1] : P[k+1] = P[q]\} = \{k;\ k < q-1 \ \text{ i } \ P_k \supset P_{q-1} \ \text{ i } \ P[k+1] = P[q]\} =$$

$$= \{k;\quad k < q-1 \ \text{ i } \ P_{k+1} \supset P_q\}$$

Fakt 2 Niech $P = P[1...m]$ będzie m elementowym wzorcem a $\pi :< 1, m > \to < 0, m-1 >$ funkcją prefiksową dla wzorca P. Dla każdego $q \in < 2, m >$ mamy:

$$\pi[q] = \begin{cases} 0 & \text{jesli} \quad E_{q-1} = \emptyset \\ 1 + \max E_{q-1} & \text{jesli} \quad E_{q-1} \neq \emptyset \end{cases} \qquad (*)$$

Dowód. 1.Jeśli. zbiór E_{q-1} jest pusty to oznacza to, że dla żadnego $k \in \pi^*[q-1]$ nie da się rozszerzyć P_k do P_{k+1} tak by było $P_{k+1} \supset P_q$ a więc zgodnie z definicją funkcji prefiksowej mamy wówczas $\pi[q] = 0$.

2. Jeśli. zbiór E_{q-1} nie jest pusty to dla żadnego $k \in E_{k-1}$ mamy (zgodnie z definicją E_{q-1}) $k < q-1$ (czyli $k+1 < q$) i $P_{k+1} \supset P_q$. Zgodnie więc z definicją funkcji sufiksowej mamy

$$\pi[q] \geq 1 + \max E_{q-1}$$

a więc $\pi[q] \geq 1$ przy przyjętym założeniu.

Jeśli $\pi[q] \geq 1$ to oznaczając $r = \pi[q] - 1$ mamy $r+1 = \pi[q] > 0$ oraz $P[r+1] = P[q]$. Stosując teraz fakt 1 dostajemy, że $r = \pi^*[q-1]$ i ponieważ $P[r+1] = P[q]$ stwierdzamy, że $r \in E_{q-1}$. Oczywiście w tej sytuacji $r \leq \max E_{q-1}$ lub równoważnie

$$\pi[q] \leq 1 + \max E_{q-1}$$

Zestawiając nierówności $\pi[q] \geq 1 + \max E_{q-1}$ i $\pi[q] \leq 1 + \max E_{q-1}$ dostajemy, że $\pi[q] = 1 + \max E_{q-1}$ ■

Warto zauważyć, że wzór (*) z powyższego faktu podaje metodę pozwalającą łatwo obliczyć $\pi[q]$ jeśli mamy $\pi^*[q-1]$.

Fakt 3 Niech $T = T[1...m]$ będzie tekstem, $P = P[1...m]$ m elementowym wzorcem, $\pi :< 1, m > \rightarrow < 0, m-1 >$ funkcją prefiksową dla wzorca P a $\delta :< 0, m > \times \Sigma \rightarrow < 0, m >$ funkcją przejścia automatu dopasowania wzorca P wówczas

dla każdego $i = 1,2,...,n$ i każdego $q \in < 1, m-1 >$ mamy $\delta(q, T[i]) = 0$ albo

$$\delta(q, T[i]) - 1 \in \pi^*[q]$$

Dowód. Oznaczmy $k = \delta(q, T[i])$ wówczas z definicji funkcji przejścia δ wynika, że $P_k \supset P_q T[i]$ zatem $k = 0$ albo $k \geq 1$. Dla tego drugiego przypadku mamy po usunięciu ostatnich symboli w P_k i $P_q T[i]$, że $P_{k-1} \supset P_q$.. Jeśli jednak $P_{k-1} \supset P_q$ to z lematu 1 mamy $k - 1 \in \pi^*[q]$ a więc rzeczywiście $\delta(q, T[i]) - 1 \in \pi^*[q]$.■

Algorytm obliczania funkcji prefiksowej o czasowej złożoności liniowej

Algorytm obliczania funkcji prefiksowej o czasowej złożoności $\Theta(m)$ jest następujący

---

Algorytm: Algorytm obliczania funkcji prefiksowej π dla wzorca $P = P[1...m]$ w czasie liniowym.

Dane WEJŚCIOWE: wzorzec $P = P[1...m]$, długość wzorca $m \in N, m \geq 1$

Dane WYJŚCIOWE: tablica $\pi[1,m]$ zawierająca wartości funkcji prefiksowej $\pi :< 1,m >\rightarrow< 0,m-1 >$

---

procedure *compute – prefix – function*(*P*,*m*)

begin

$\pi[1] := 0;$

{stwierdziliśmy, że zawsze mamy $\pi[1] = 0$, pozostaje obliczyć $\pi[k]$ dla $q = 2,3,...,m$ }

$k := 0;$

for $q = 2$ **to** m **do**

begin

while $k > 0$ i $P[k+1] \neq P[q]$ **do** $k := \pi[k];$

if $P[k+1] = P[q]$ **then** $k := k+1;$

$\pi[q] := k;$

end;

end *computer – prefix- function*;

W Matlabie można powyższy algorytm zapisać następująco:

Algorytm: Algorytm obliczania funkcji prefiksowej π dla wzorca $P = P[1...m]$ w czasie liniowym.

Dane WEJŚCIOWE: wzorzec $P = P[1...m]$, długość wzorca $m \in N, m \geq 1$

Dane WYJŚCIOWE: tablica $\pi[1,m]$ zawierająca wartości funkcji prefiksowej $\pi :< 1,m >\rightarrow< 0,m-1 >$

---

function wart=compute_prefix_function(P)
M=size(P); m=M(2); pi(1)=0; k=0;

for q=2:m,

```
while (k>0)&(~(P(k+1)==P(q))), k=pi(k); end,
if P(k+1)==P(q), k=k+1; end,
pi(q)=k;
end,
wart=pi;
disp('funkcja prefiksowa ='); disp(pi)
```

Uzasadnienie poprawności algorytmu KMP-matcher

Wykażemy, że 2 instrukcje procedury KMP-matcher

while $q > 0$ i $P[q+1] \neq T[i]$ **do** $q := \pi[q]$; (*)

oraz

if $P[q+1] = T[i]$ **then** $q := q+1$; (**)

obliczają za pomocą funkcji prefiksowej $\pi :< 1, m > \rightarrow < 0, m-1 >$ wartość funkcji przejścia $\delta(q, T[i])$ (automatu wyszukiwania wzorca P).

Skorzystamy z faktu 3, z którego wynika, że dla każdego $i = 1,2,...,n$ i każdego $q \in < 1, m-1 >$ mamy $\delta(q, T[i]) = 0$ albo

$$\delta(q, T[i]) - 1 \in \pi^*[q]$$

Niech q' oznacza teraz wartość q z jaką wchodzimy do instrukcji **while** (*). Z lematu 1 wynika, że $\pi^*[q] = \{k;\ P_k \supset P_q\}$**.** Zatem iterując funkcję prefiksową π czyli obliczając kolejno $q := \pi[q]$ (tak jak w instrukcji (*)) przeglądamy elementy zbioru $\pi^*[q] = \{k;\ P_k \supset P_q\}$ w kolejności malejącej szukając dopasowania $P_k \supset P_q$ (z możliwie największym k) z jednoczesnym spełnieniem warunku $P[q+1] = T[i]$ (czyli innymi słowy szukamy największego k spełniającego relację $P_{k+1} \supset P_q T[i]$).

Z pętli **while** (*) wychodzimy, gdy $q = 0$ lub $P[q+1] = T[i]$. Jeśli wychodzimy w pętli (*) z warunkiem $q = 0$ i $P[q+1] \neq T[i]$ to oczywiście $\delta(q', T[i]) = 0$ bo żaden sufiks **słowa** $P_q T[i]$ (poza pustym) nie jest prefiksem P**.** Jeśli wychodzimy z pętli (*) z warunkiem $P[q+1] = T[i]$ to zgodnie z (***) mamy $\delta(q', T[i]) = q+1$ i to podstawienie realizowane jest w instrukcji (**).

Instrukcja $q := \pi[q]$ wykonywana jest po wykryciu dopasowania wzorca do tekstu i umożliwia nam odtworzenie sytuacji jaką mamy przy wchodzeniu do pętli (*). Bez tej instrukcji odwołalibyśmy się w następnym wykonaniu pętli **for** do nieistniejącego elementu $P[m+1]$ tablicy $P[1..m]$.

Ostatecznie stwierdzamy, że procedura *KMP-matcher* daje się sprowadzić do procedury *finite–automaton-macher*. Poprawność zaś procedury *finite–automaton-macher* wykazaliśmy w poprzednim podrozdziale.

Uzasadnienie poprawnośći algorytmu obliczania funkcji prefiksowej

Zauważmy, że w każdym przebiegu pętli **for** wartość zmiennej k może być zwiększana o 1 w instrukcji **if** wewnątrz pętli i zmniejszana w instrukcji pętli **while** w konsekwencji zawsze mamy $0 \le k \le m-1$ (takie właśnie wartości może przyjmować funkcja prefiksowa).

Zauważmy, że do pętli **while** wchodzimy zawsze z wartością $k = \pi[q-1]$ dla $q = 2,3,...,m$. Ostatnia instrukcja procedury obliczenia funkcji prefiksowej czyli $\pi[q] := k$ odtwarza ten warunek. Początkowo przy wejściu do pętli **while** mamy zawsze $k = 0$ co odpowiada $\pi[1] = 0$.

W pętli **while** przeglądane są wartośc $k \in \pi^*[q-1]$ (od strony największych wartości) do momentu znalezienia takiego k, że $P[k+1] = P[q]$ lub skończenia się elementów zbioru $\pi^*[q-1]$ (wartość $k = 0$).

Jeśli wychodzimy z pętli **while** z warunkiem $k > 0$ i $P[k+1] = P[q]$ to oznacza to, że $k = \max E_{q-1}$ i z faktu 2 mamy $\pi[q] = 1 + \max E_{q-1}$. i to dodawanie realizujemy wewnątrz kolejnej instrukcji **if**.

Jeśli wychodzimy z pętli **while** z warunkiem $k = 0$ i $P[k+1] = P[q]$ to oczywiście $\pi[q] = 1$ i instrukcja **if** zwiększa k o 1 a następna instrukcja realizuje podstawienie $\pi[q] := 1$**.**

Jeśli wychodzimy z pętli **while** z warunkiem $k = 0$ i $P[k+1] \neq P[q]$ to oczywiście $\pi[q] = 0$ i tak jest, bo instrukcja **if** się nie wykonuje.

Pesymistyczna złożoność czasowa algorytmu obliczania wartości funkcji prefiksowej. Zauważmy, że w instrukcjach **while** i **if** algorytmu obliczania funkcji prefiksowej zmienna k zawsze spełnia nierówność $0 \le k \le m-1$.

Zmienna k (z uwagi na to, że funkcja prefiksowa jest funkcją malejącą tzn. dla każdego $q \in <1, m>$ mamy $\pi(q) < q$) może być w instrukcji **while** tylko zmniejszana przy każdym obiegu pętli a w instrukcji **if** zmienna k jest ewentualnie zwiększana o 1.

Jeśli przyjmiemy pesymistycznie, że zmniejszenie zmiennej k w instrukcji **while** następuje o 1 to liczba wykonań pętli **while** plus liczba wykonań instrukcji **if** jest równa ilości zwiększeń („kroków w górę") i zmniejszeń („kroków w dół") zmiennej k w pętli **for**. Liczba kroków w dół musi być mniejsza równa od ilości kroków w górę a liczba kroków w górę jest mniejsza równa od m. Zatem liczba wszystkich kroków jest mniejsza równa od $2m$. Jednocześnie liczba obiegów pętli **for** jest równa m-1.

Ostatecznie więc stwierdzamy, że pesymistyczna złożoność czasowa algorytmu obliczania wartości funkcji prefiksowej jest równa $\Theta(m)$.

Pesymistyczna złożoność czasowa algorytmu KMP. Analogiczną jak wyżej analizę złożoności czasowej można przeprowadzić dla algorytmu *KMP*. Pesymistyczna złożoność czasowa algorytmu *KMP* jest więc równa $\Theta(n)$ a licząc dodatkowo czas obliczania wartości funkcji prefiksowej dla wzorca $P = P[1..m]$ dostajemy złożoność czasową równą $\Theta(n+m)$.

Literatura

[1] T. H. Cormen, C.E. Leiserson, R.L. Rivest; Wprowadzenie do algorytmów; WNT, Warszawa 2003.

[2] L. Banachowski, K. Diks, W. Rytter; Algorytmy i struktury danych; WNT, Warszawa 1996.

[3] A.V. Aho, J.E. Hopcroft, J.D. Ulman; Projektowanie i analiza algorytmów komputerowych; WNT, Warszawa 2003.

[4] D.E. Knuth; Sztuka programowania; WNT, 2002

Zadania

Zadanie 1
Jakie porównania wykonuje algorytm naiwny jeśli szukamy wzorca $P = 0001$ w tekście $T = 01001000100001$.

Zadanie 2
Załóżmy, że operujemy alfabetem $V = \{0,1,2,...,9\}$ i wykonujemy algorytm Rabina-Karpa dla wzorca $P[1..2] = 36$ i szukamy wzorca w tekście $T[1..16] = 314159365358978$. Niech ponadto $q = 13$. Ile razy wystąpi w algorytmie "fałszywa zgodność".

Rozwiązanie
Widać, że $t_0 = 5$, $t_1 = 1$, $t_2 = 5$, $t_3 = 2$, $t_4 = 7$, $t_5 = 2$, $t_6 = 0$ (prawdziwa zgodność), $t_7 = 0$ (fałszywa zgodność), $t_8 = 1$, $t_9 = 9$, $t_{10} = 7$, $t_{11} = 11$, $t_{12} = 6$, $t_{13} = 1$, $t_{14} = 2$.

Zatem tylko jeden raz wystąpi fałszywa zgodność.■

Zadanie 3
Jak uogólnić algorytm Rabina-Karpa na dwuwymiarowy problem wyszukiwania wzorca $m \times m$ w tablicy $n \times n$, gdzie $m \le n$. Zakładamy, że wzorzec można przesuwać poziomo i pionowo ale wzorca nie można obracać szukając dopasowania.

Zadanie 4
Jak uogólnić algorytm Rabina-Karpa na dwuwymiarowy problem wyszukiwania wzorca $m \times m$ w tablicy $n \times n$, gdzie $m \le n$. Zakładamy, że wzorzec można przesuwać poziomo i pionowo i szukając dopasowania można wzorzec obracać o kąt będący wielokrotnością $\pi / 2$.

Zadanie 5
Mamy alfabet $\Sigma = \{a,b,c\}$. Wyznaczyć funkcję prefiksową π dla wzorca

a) $P[1..10] = ababababab = (ab)^5$

b) $P[1..100] = (a)^{100}$

c) $P[1..100] = (abc)^{100}$

Zadanie 6
Podać górne ograniczenie na liczbę elementów zbioru $\pi^*[q]$ w funkcji $q = 1,2,3,...,m$. Podać przykład, w którym to ograniczenie jest osiągane dla każdego $q = 1,2,3,...,m$.

Zadanie 7

Obliczyć czas wykonania algorytmu KMP jeśli w 100 znakowym tekście $T[1..100] = a^{100}$ tzn. $\underbrace{aaa...a}_{100}$ wzorca *bbb.*

Zadanie 8

A(licja) i B(ob.) pracujący na 2 połączonych siecią odległych komputerach chcą sprawdzić czy dwa słowa binarne (np. 2 pliki) jakimi dysponują są identyczne. Oznaczamy te dwa słowa o długości n przez $\alpha = a_0 a_1 a_2 ... a_{n-1}$ i $\beta = b_0 b_1 ... b_{n-1}$. Jednocześnie Alicja i Bob nie chcą dokonywać porównania tych plików słów na zasadzie przesłania ich w całości przez sieć komputerową z dwóch powodów:

- Słowa α i β są zbyt długie

- Przesłanie ich w całości przez sieć jest ryzykowne z uwagi na możliwy podsłuch.

W celu stwierdzenia identyczności słów α i β Alicja i Bob wykonują następujący test probabilistyczny:

1. Wybierają dowolną liczbę pierwszą $q > m \cdot n$, gdzie $m \in N$, $m \geq 2$ jest dowolnie ustalone a n jest długością słów α i β;

2. Wybierają losowo liczbę $x \in F_q$. Ściślej niech zmienna losowa dyskretna $X : \Omega \to F_q$ określona na pewnej przestrzeni probabilistycznej $(\Omega, \mathfrak{M}, P)$ opisuje wybór liczby $x \in F_q$ tzn.

$$\underset{x \in F_q}{\forall} P(X = x) = \frac{1}{q}$$

3.Alicja oblicza wartość (*) wielomianu W_A o współczynnikach w ciele F_q.

$$W_A(x) = a_{n-1} \otimes_q x^{n-1} \oplus_q a_{n-2} \otimes_q x^{n-2} \oplus_q ... \oplus_q a_1 \otimes_q x \oplus a_0 \qquad (*)$$

gdzie $\oplus_q$ i $\otimes_q$ są odpowiednio sumą modulo q i mnożeniem modulo q.

4.Bob oblicza wartość (**) wielomianu W_B o współczynnikach w ciele F_q.

$$W_B(x) = b_{n-1} \otimes_q x^{n-1} \oplus_q b_{n-2} \otimes_q x^{n-2} \oplus_q ... \oplus_q b_1 \otimes_q b_0. \qquad (*)$$

5. Alicja i Bob porównują wartość $W_A(x)$ i $W_B(x)$. Jeśli $W_A(x) = W_B(x)$ przyjmują, że słowa α i β są identyczne, jeśli $W_A(x) \neq W_B(x)$, to oczywiście słowa α i β są równe.

Wykazać, że prawdopodobieństwo pomyłki, jeśli $\alpha \neq \beta$ jest mniejsze od $\frac{1}{m}$, czyli wykazać, że jeśli $\alpha \neq \beta$, to $P(W_A(X) = W_B(X)) < \frac{1}{m}$.

Rozwiązanie

Oznaczmy $W(x) \overset{ozn}{=} W_A(x) - W_B(x))$. Jest to wielomian co najwyżej *n*-tego stopnia a więc ma co najwyżej *n* pierwiastków. $W(x)$ jest to bowiem wielomianem o współczynnikach w ciele. Gdybyśmy użyli zamiast ciała pierścienia to liczba pierwiastków mogłaby się zwiększyć. Zatem wartości $x \in F_q$ dla których $W(x) = 0$ jest co najwyżej *n* zatem

$$P(W(X) = 0) = P(W_A(X) = W_B(X)) \leq \frac{n}{q} < \frac{n}{n \cdot m} = \frac{1}{m}$$

a więc ostatecznie

$$P(W_A(X) = W_B(X)) < \frac{1}{m}$$

■

Uwaga 1. Porównywane słowa $\alpha = a_0 a_1 a_2 ... a_{n-1}$ i $\beta = b_0 b_1 ... b_{n-1}$ nie muszą być jak w powyższym zadaniu słowami binarnymi. Mogą być natomiast słowami nad pewnym ustalonym alfabetem o *d* symbolach, wówczas dobierając liczbę pierwszą *q* musimy ją dobrać tak by $q \geq m \cdot n$ i $q \geq d$.

Uwaga 2. Widać, że liczby jakie wiążemy ze słowami $\alpha = a_0 a_1 a_2 ... a_{n-1}$ i $\beta = b_0 b_1 ... b_{n-1}$ są uzyskiwane podobnie jak w algorytmie Rabina – Karpa z tym, że w algorytmie Rabina–Karpa nie stosujemy losowego $x \in F_q$ a stosujemy $x = d$ a dokładniej $x = d (\mathrm{mod}\, q)$ (jednak z reguły by zmniejszyć prawdopodobieństwo „fałszywej równości" mamy $q >> d$).

Zadanie 9
Zdefiniować automat skończony (wyszukiwania wzorca) wyszukujący słowo $P[1..5] = begin$ (wzorzec) w tekście T nad alfabetem złożonym z małych liter języka angielskiego.

Zadanie 10
Mamy dany alfabet $\Sigma = \{a, b\}$. Skonstruować automat wyszukiwania wzorca $P[1..5] = aabab$ i przetestować go dla losowo generowanych słów nad alfabetem $\Sigma = \{a, b\}$.

Zadanie 11
Wzorzec P nazywamy samorozłącznym jeśli (ang. nonovelappable) jeśli $P_k \supset P_q$ implikuje ($k = 0$ lub $k = q$). Opisać automat wyszukiwania wzorca tego typu.

Zadanie 12
Opisać precyzyjnie naiwny algorytm wyszukiwania wzorca. Ocenić jego pesymistyczną złożoność czasową. Jakie znasz pomysły "przyśpieszające" stosowane w algorytmach wyszukiwania wzorca.