Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Wykład 13.

Algorytmy tekstowe

Algorytmy tekstowe – podstawowe pojęcia

Pod pojęciem „tekst”, dla potrzeb algorytmów przetwarzania tekstów, będziemy rozumieli dowolny ciąg znaków (w tym zakodowanych, np. z użyciem kodu ASCII) lub ciąg bitów (np. reprezentujący pamięć ekranu).
Miarą długości tekstu jest liczba „znaków”, rozumianych jako elementy, w których składa się „tekst”.
W przypadku ciągu bitów, długość ciągu będzie wyrażana liczbą bitów. Zatem pojęcie „znak” dotyczy każdego z elementów tworzących „tekst”.

Istnieje szereg zagadnień związanych z przetwarzaniem tekstu. Wykład dotyczy tylko dwóch wybranych zagadnień:

- poszukiwania wzorca w tekście,
- kodowania tekstu.

Pozostałe zagadnienia związane z problematyką należy poznać samodzielnie, w miarę potrzeb i zainteresowań, studiując dostępną literaturę przedmiotu.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Zadanie polega na poszukiwaniu wystąpień wzorca W, o długości m znaków, w tekście T składającym się z n znaków. W praktyce m << n. Każde wystąpienie wzorca w tekście jest określone pozycją i – numerem znaku w tekście T, od którego rozpoczyna się wystąpienie wzorca W.

Przykład tekstu T o długości n=45 znaków i wzorca W o długości 4 znaki:

T:	N	A		U	C	Z	E	L	N	I		U	C	Z	Ę		I		U	C	Z	Ę	,		M	O	Ż	E		K	I	E	D	Y	Ś		N	A	U	C	Z	Ę	.	.	.
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44

W:	U	C	Z	Ę
	0	1	2	3

Inny przykład: ciąg bitów, reprezentowanych przez znaki „0” i „1”:

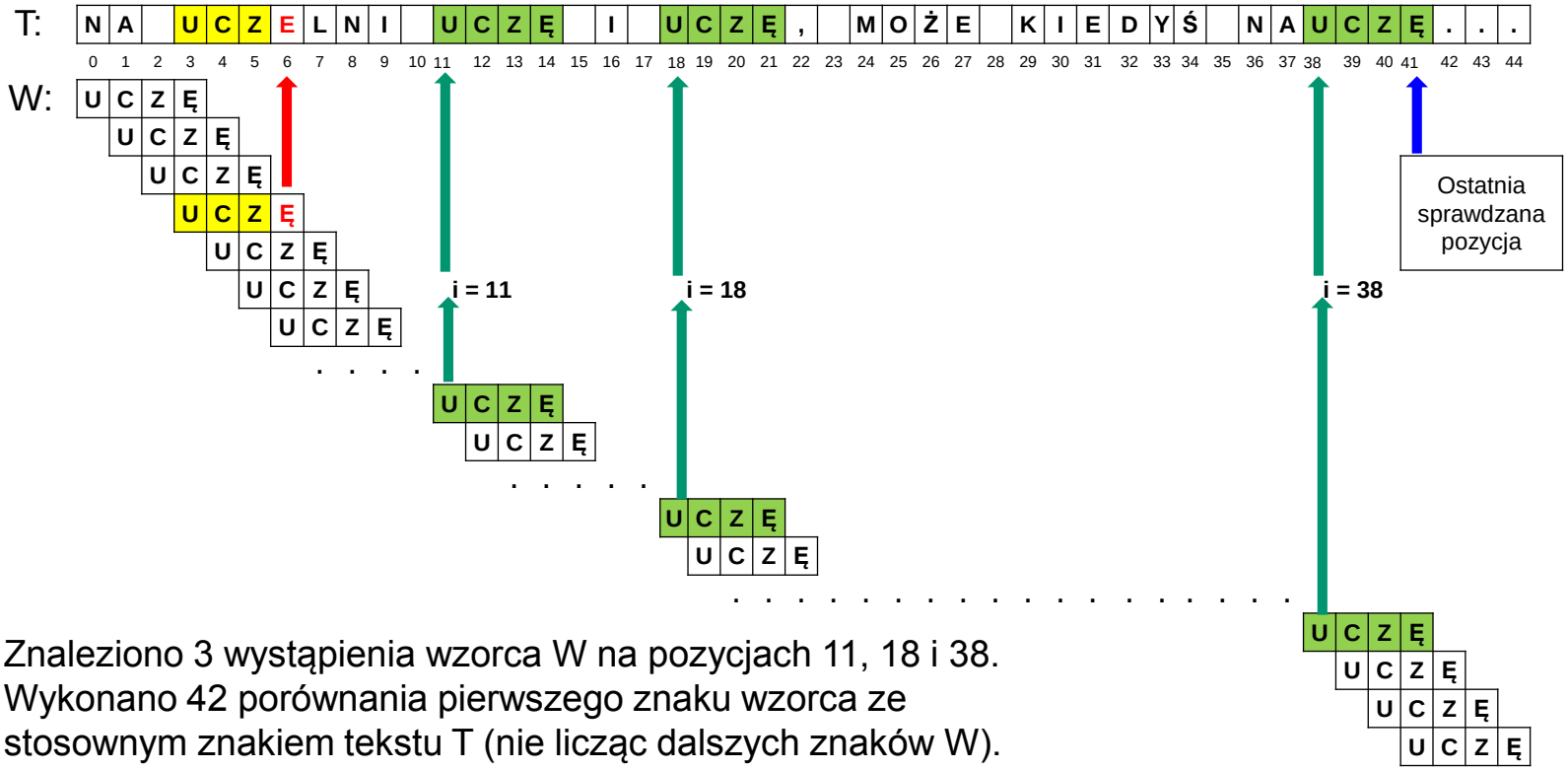
T:	0	0	1	0	1	0	1	0	1	0	1	0	0	1	1	0	0	1	0	0	0	1	0	1	0	1	0	1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

W:	0	1	0	1	0
	0	1	2	3	4

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Algorytm naiwny (siłowy, typu *brute-force*)

Algorytm stosuje naturalne podejście sekwencyjnego przeglądania tekstu, w poszukiwaniu w nim pozycji wystąpienia pierwszego znaku wzorca, oraz, gdy to nastąpi, sprawdzaniu zgodności, parami, pozostałych znaków wzorca na kolejnych pozycjach w tekście.



Znaleziono 3 wystąpienia wzorca W na pozycjach 11, 18 i 38.
Wykonano 42 porównania pierwszego znaku wzorca ze stosownym znakiem tekstu T (nie licząc dalszych znaków W).

Jaka jest złożoność obliczeniowa (optymistyczna, pesymistyczna) tego algorytmu?

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Kod przykładowej klasy (źródło: [Harris, Ross – *Algorytmy od podstaw*]) z metodą *brute-force* realizującą poszukiwanie jednego wystąpienia wzorca „pattern” w tekście „text”, od pozycji „from”.

```
public interface StringSearcher { public int search(CharSequence text, int from); }

public class BruteForceSearcher implements StringSearcher {
    private final CharSequence _pattern;

    public BruteForceSearcher(CharSequence pattern) { _pattern = pattern; }

   public int search(CharSequence text, int from) {
      int s = from;
      while (s <= text.length() - _pattern.length()) {
         int i = 0;
         while (i < _pattern.length() && _pattern.charAt(i) == text.charAt(s + i))
            ++i;
         if (i == _pattern.length())   return s;
         ++s;
      }
      return -1;
   }
}
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Algorytm Boyera – Moore'a (wersja prosta, „poglądowa")

Algorytm wprowadza usprawnienie (w stosunku do algorytmu *brute-force*), wynikające ze spostrzeżenia, że wiele porównań fragmentów teksu można uniknąć, w szczególności dotyczących fragmentów zawierających znaki, których nie ma we wzorcu.

Istota podejścia polega na określeniu dystansu, o jaki należy przesunąć wzorzec w związku z kolejnym porównaniem. Porównanie rozpoczyna się od ostatniego znaku wzorca (choć przy pierwszym porównaniu oczywiście pierwszy znak wzorca pokrywa się z pierwszym znakiem tekstu). Każdorazowo, gdy porównywane znaki są różne, wzorzec „przesuwany jest" na taką pozycję, by ten niezgodny znak tekstu pokrył się z identycznym znakiem wzorca.

Opisuje to następujący (uproszczony względem oryginalnej wersji algorytmu) scenariusz (heurystyka błędnego znaku, *bad character heuristic*):

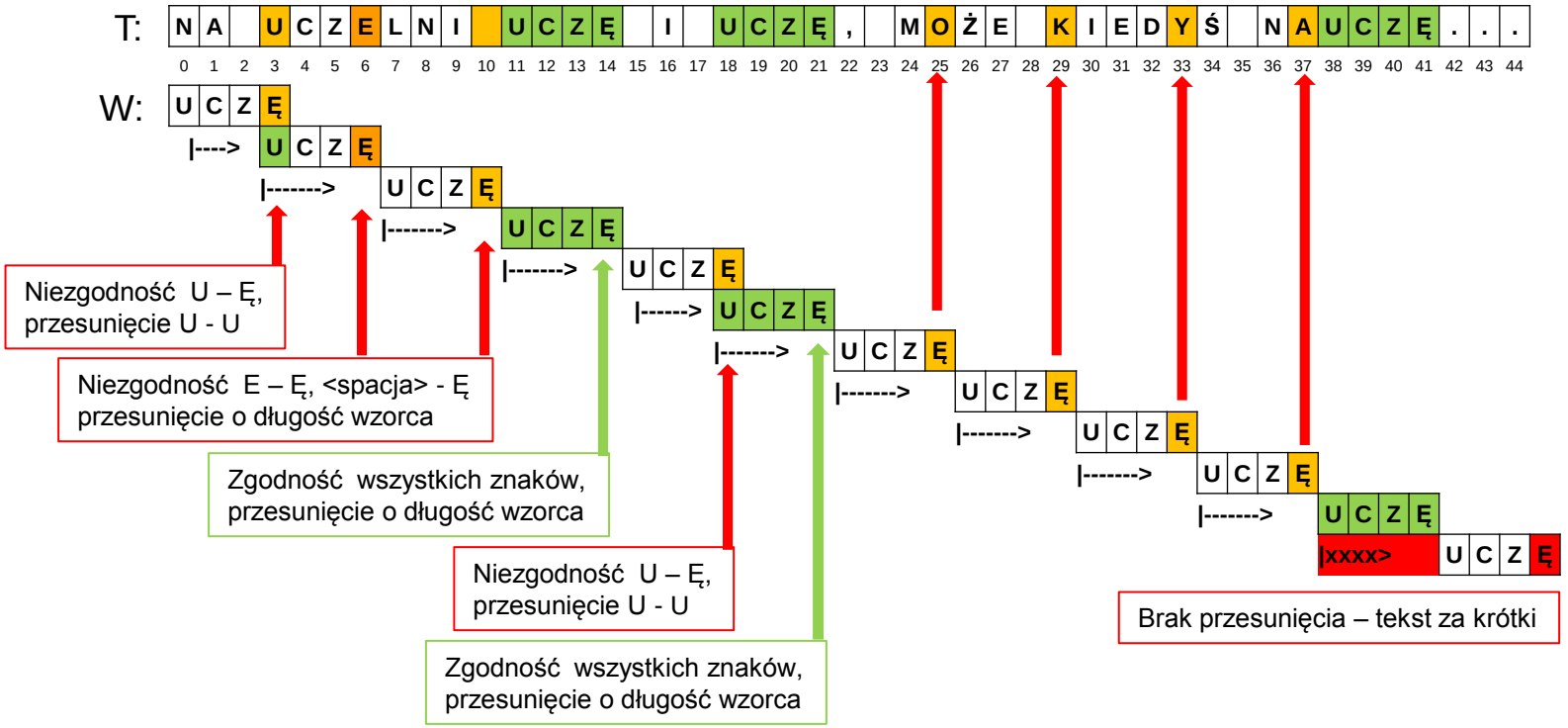
1. Jeśli niepasujący znak tekstu występuje we wzorcu, przesuwamy wzorzec tak, by ten znak pokrył się z identycznym znakiem wzorca.
2. Jeśli niepasujący znak nie występuje we wzorcu, przesuwamy wzorzec o całą jego długość.
3. Jeśli z w/w zasad wynikałaby konieczność przesunięcia wzorca w lewo, ignorujemy to przesunięcie i przesuwamy wzorzec o jedną pozycję w prawo.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Algorytm Boyera – Moore'a

Ilustracja idei działania algorytmu



Znaleziono 3 wystąpienia wzorca W na pozycjach 11, 18 i 38.
Wykonano 11 porównań ostatniego znaku wzorca ze stosownym znakiem tekstu T (nie licząc dalszych znaków W).
Jaka jest złożoność obliczeniowa (optymistyczna, pesymistyczna) tego algorytmu?

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Kod przykładowej klasy ([Harris, Ross – *Algorytmy od podstaw*]) z metodą *Boyera-Moore'a* realizującą poszukiwanie jednego wystąpienia wzorca „pattern" w „text", od pozycji „from".

```
public class BoyerMooreSearcher implements StringSearcher {
   private static final int CHARSET_SIZE = 256;  // The supported character set size (ASCII)
   private final CharSequence _pattern;
   private final int[] _lastOccurrence;   // The position (0, 1, 2...) of the last occurrence of each
                                          // character within the pattern.
   public BoyerMooreSearcher(CharSequence pattern) {
       _pattern = pattern;
      _lastOccurrence = computeLastOccurrence(pattern);
   }

   public int search(CharSequence text, int from) {
      int s = from;
      while (s <= text.length() - _pattern.length()) {
         int i = _pattern.length() - 1;
         char c = 0;
         while (i >= 0 && _pattern.charAt(i) == (c = text.charAt(s + i)))
            --i;
         if (i < 0)
            return s;
         s += Math.max(i - _lastOccurrence[c], 1);
      }
      return -1;
   }
// … c. d.n. …
```

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Algorytm Boyera – Moore'a

```
/// … c.d.

private static int[] computeLastOccurrence(CharSequence pattern) {
        int[] lastOccurrence = new int[CHARSET_SIZE];
        for (int i = 0; i < lastOccurrence.length; ++i)
            lastOccurrence[i] = -1;
        for (int i = 0; i < pattern.length(); ++i)
            lastOccurrence[pattern.charAt(i)] =  i;
        return lastOccurrence;
    }
}  // class BoyerMooreSearcher
```

Metoda computeLastOccurrence służy do wypełnienia tablicy _lastOccurrence pozycjami ostatnich wystąpień, we wzorcu, poszczególnych znaków, które mogą się pojawić w tekście. Jeśli znak nie występuje w tekście, odpowiadający mu element tablicy ma wartość „-1". Tablica służy do wyznaczania kolejnych przesunięć wzorca względem tekstu.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Algorytmy tekstowe – poszukiwanie wzorca w tekście

Algorytm Karpa-Rabina (idea)

W algorytmie Karpa-Rabina wykorzystuje się funkcję haszującą, której konstrukcja charakteryzuje się niskim kosztem uzyskiwania wartości funkcji dla określonego podciągu znaków w tekście.

Znając wartość funkcji haszującej dla wzorca doprowadza się do bezpośredniego porównywania tylko tych podciągów, dla których wartość funkcji haszującej jest równa wartości funkcji haszującej wzorca. Unika się w ten sposób kosztownego porównywania znacznej liczby par znaków w podciągach.

Sposób wydzielania podciągów może być różny (np. taki, jak w algorytmie brute-force).

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Kompresja tekstu

Kody Huffmana

Kompresja tekstu pozwala zapisać go w krótszej postaci za pomocą systemu kodowania, by następnie w działaniu odwrotnym (dekodowania) uzyskać tekst bez utraty jego pierwotnej postaci (kompresja bezstratna) lub z częściową utratą określonych jego elementów (kompresja stratna).

Algorytm Huffmana jest prostym algorytmem kompresji bezstratnej, zazwyczaj używanym w powiązaniu z innymi algorytmami.

Często wykorzystuje się go jako ostatni etap w różnych systemach kompresji, zarówno bezstratnej, jak i stratnej, np. MP3 lub JPEG.

Pomimo że nie jest doskonały, stosuje się go ze względu na prostotę oraz brak ograniczeń patentowych. Jest to przykład wykorzystania algorytmu zachłannego.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Kompresja tekstu

Kody Huffmana

Dany jest alfabet źródłowy S = {s1, s2, …, sn } (zbiór symboli) oraz zbiór stowarzyszonych z nim prawdopodobieństw P = {p1, p2, …, pn } wystąpienia tych symboli w tekście. Prawdopodobieństwa te mogą zostać określone:

- z góry, np. poprzez wyznaczenie częstotliwości występowania znaków w tekstach danego języka,
- indywidualnie dla każdego zestawu danych, w procesie analizy tekstu.

Kodowanie Huffmana polega na utworzeniu słów kodowych (ciągów bitowych), których długość jest odwrotnie proporcjonalna do prawdopodobieństwa.
Im częściej dany symbol występuje (może wystąpić) w ciągu danych, tym mniej zajmie bitów.
Własności kodu Huffmana:
- jest kodem prefiksowym, co oznacza, że żadne słowo kodowe nie jest początkiem innego słowa,
- średnia długość słowa kodowego jest najmniejsza spośród kodów prefiksowych,
- jeśli prawdopodobieństwa są różne, tzn. pj > pi, to długość kodu dla symbolu xj jest nie większa od kodu dla symbolu xi,
- słowa kodu dwóch najmniej prawdopodobnych symboli mają równą długość.
Kompresja polega na zastąpieniu symboli otrzymanymi kodami.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Kompresja tekstu

Algorytm Huffmana

1. Określ prawdopodobieństwo (lub częstość występowania) dla każdego symbolu ze zbioru symboli S.

2. Utwórz listę drzew binarnych, które w węzłach przechowują pary:
<symbol, prawdopodobieństwo>.
Na początku drzewa składają się wyłącznie z korzenia.

3. Dopóki na liście jest więcej niż jedno drzewo, powtarzaj:
   - usuń z listy dwa drzewa o najmniejszym prawdopodobieństwie zapisanym w korzeniu,
   - wstaw nowe drzewo, w którego korzeniu jest suma prawdopodobieństw usuniętych drzew, natomiast one same stają się jego lewym i prawym poddrzewem. Korzeń drzewa nie przechowuje symbolu.

Drzewo, które pozostanie na liście, jest nazywane **drzewem Huffmana.**

Prawdopodobieństwo zapisane w korzeniu jest równe 1, natomiast w liściach drzewa zapisane są symbole.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Kompresja tekstu

Algorytm Huffmana

Algorytm Huffmana jest algorytmem niedeterministycznym, ponieważ nie określa, w jakiej kolejności wybierać drzewa z listy, jeśli mają równe prawdopodobieństwa.

Nie jest również określone, które z usuwanych drzew ma stać się lewym bądź prawym poddrzewem. Jednak bez względu na przyjęte rozwiązanie średnia długość kodu pozostaje taka sama.

Na podstawie drzewa Huffmana tworzone są słowa kodowe.
Algorytm jest następujący:

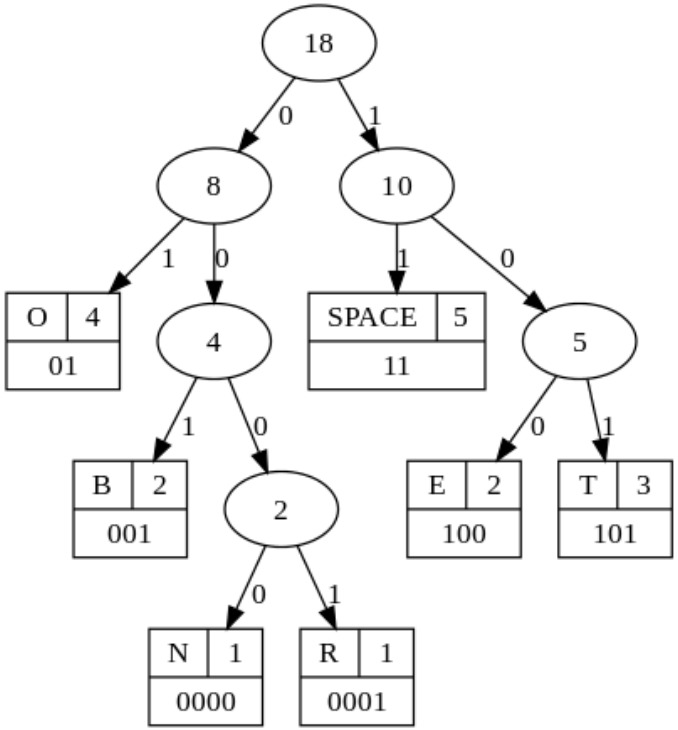
1. Każdej lewej krawędzi drzewa przypisz 0, prawej (można też odwrotnie).
2. Przechodź w głąb drzewa od korzenia do każdego liścia (symbolu):
   - jeśli skręcasz w prawo, dopisz do kodu bit o wartości 1.
   - jeśli skręcasz w lewo, dopisz do kodu bit o wartości 0.

Długość słowa kodowego jest równa głębokości symbolu w drzewie, wartość binarna zależy od jego położenia w drzewie.

Kompresja tekstu

Kody Huffmana

Drzewo Huffmana wygenerowane z frazy: TO BE OR NOT TO BE



Kody symboli są określone przez oznaczenia krawędzi prowadzących do wierzchołków zawierających dane symbole:

O: 01
spacja: 11
B: 001
E: 100
T: 101
N: 0000
R: 0001

Zakodowanie/odkodowanie frazy nie powinno być trudne… ☺