Kompresja tekstu

May 11, 2021

1 Kompresja tekstu

1.1 dr inż. Aleksander Smywiński-Pohl

1.2 konsultacje: pt. 16:30 - 17:30

2 Klasyfikacja metod kompresji

długość / słownik	stała-do-zmiennej	zmienna-do-zmiennej
	kodowanie różnicowe	kompresja powtórzeń
stały słownik	kodowanie statystyczne	kompresja składowych
zmienny słownik	sekwencyjne kodowanie statystyczne	algorytm Ziva i Lempela

- dla kompresji można wyróżnić dwa morfizmy g oraz h:
  - g przeprowadza $a \in \Sigma$ w $b \in B^* = \{0, 1\}$
  - h przeprowadza $b \in B^*$ w $c \in B^*$
- długość fragmentu tekstu która jest zamieniana dla metod o stałej długości odpowiada najczęściej długości pojedynczej litery b w kodzie binarnym, wtedy $b \in B^n$
- słowa c najczęściej mają zmienną długość, dzięki czemu możemy uzyskać kompresję
- morfizm h posiada morfizm odwrotny h^{-1}, dzięki czemu możemy skompresowany tekst odtworzyć bez strat
- dla metod o stałym słowniku, h jest stałe w trakcie całego procesu kompresji
- dla metod o zmiennym słowniku h zmienia się w procesie kompresji, a odpowiadająca jej funkcja h^{-1} jest wyliczana w trakcie dekompresji

2.1 Kompresja powtórzeń

- zastępowanie powtarzających się wystąpień litery, np. aaaaaaa
- prosta metoda kompresji, np. wykorzystywana do zastępowania białych znaków
- sekwencja a^n zastępowana jest napisem $\&an$, gdzie $\& \notin \Sigma$, a n reprezentowane jest w formie binarnej. Jeśli $n > 255$, to zapis dzieli się na fragmenty takie, żeby $n < 256$

3 Kodowanie różnicowe

- 1980, 1982, 1981, 1985, 1981, …
- 1980, +2, -1, +4, -4
- Linie faksu:
  - `0101001 0101010 1001001 1101001 1011101 1000000` - linia n

– `0101000 0101011 0111001 1100101 1011101 0000000` - linia n + 1
– `0000001 0000001 1110000 0001100 0000000 1000000` - różnica
– `(7,1), (7,4), (8, 2), (10, 1)` - zakodowana różnica
– dobre, jeśli strona zawiera np. pismo odręczne oraz wiele pustej przestrzeni

4 Statyczne kodowanie Huffmana

- opiera się na obserwacji, że rozkład liter w tekstach nie jest równomierny
- literom pojawiającym się częściej można przyporządkować krótsze sekwencje bitów
- literom pojawiającym się rzadziej można przyporządkować dłuższe sekwencje bitów
- można też przyjąć rozkład *a priori* (np. wyliczony dla dużego korpusu tekstów), dzięki czemu unikamy konieczności wyliczania prawdopodobieństw oraz umieszczania słownika w pliku wynikowym

5 Kod prefiksowy

Żadne słowo $c_i \in B^*$ nie jest prefiksem innego słowa $c_j \in B^*$, dla $i \neq j$.

Uwaga: każdy kod nie posiadający tej własności, może zostać zastąpiony kodem prefiksowym, bez utraty stopnia kompresji.

6 Nierówność Krafta - McMillana (3.1)

$\sum_{c_i \in \Gamma} r^{-|c_i|} \leq 1$

1. $\Sigma = \{s_1, s_2, s_3, \cdots\}$
2. $B^* \supset \Gamma = \{h(s_1), h(s_2), h(s_3), \cdots\} = \{c_1, c_2, c_3, \cdots\}$
3. $r = |B| = 2$ - rozmiar alfabetu kodów
4. Kraft: isnieje kod prefiksowy $\Leftrightarrow$ spełniona jest nierówność 3.1.
5. McMillan: istnieje jednoznacznie odwracalny kod ze słowami o długościach $l_1, l_2, l_3, \cdots \Leftrightarrow$ spełniona jest nierówność 3.1.

7 Rozmiar spakowanego tekstu

$|h(text)| = \sum_{a \in \Sigma} n_a \times |h(a)|$

$|h(text)| = \sum_{a \in \Sigma} n_a \times |depth(f_a)|$

gdzie:

- n_a - liczba wystąpień w całym tekście litery a
- $depth(f_a)$ - głębokość liścia odpowiadającego literze a

8 Drzewo Huffmana dla słowa *abracadabra*

9 Algorytm budowy drzewa Huffmana

[ ]:
```
def huffman(letter_counts):
    nodes = []
    for a, weight in letter_counts.items():
        nodes.append(Node(a, weight))
    internal_nodes = []
    leafs = sorted(nodes, key=lambda n: n.weight)
    while(len(leafs) + len(internal_nodes) > 1):
        element_1, element_2 = # elementy nodes i internal nodes o najniższym␣
↪koszcie, usunięte z list
        # warto wziąć pod uwagę, że można wziąć 2 razy najmniejszy węzeł, co␣
↪istotnie upraszcza kod
        internal_nodes.
            append(Node(element_1, element_2, element_1.weight + element_2.
↪weight)
    return internal_nodes[0]
```

[3]:
```
def huffman(letter_counts):
    nodes = []
    for a, weight in letter_counts.items():
        nodes.append(Node(a, weight))
    internal_nodes = []
    leafs = sorted(nodes, key=lambda n: n.weight)
    while(len(leafs) + len(internal_nodes) > 1):
        head = []
        if(len(leafs) >= 2):
            head += leafs[:2]
        elif(len(leafs) == 1):
            head += leafs[:1]
        if(len(internal_nodes) >= 2):
            head += internal_nodes[:2]
        elif(len(internal_nodes) == 1):
            head += internal_nodes[:1]
        element_1, element_2 = sorted(head, key=lambda n: n.weight)[:2]
        internal_nodes.append(Node(element_1,
                        element_1.weight + element_2.weight,
                        element_2))
        if(len(leafs) > 0 and element_1 == leafs[0]):
            leafs = leafs[1:]
        else:
            internal_nodes = internal_nodes[1:]
        if(len(leafs) > 0 and element_2 == leafs[0]):
            leafs = leafs[1:]
        else:
```

```
            internal_nodes = internal_nodes[1:]
    return internal_nodes[0]
```

[4]:
```
from queue import Queue as queue

class Node:
    def __init__(self, child_1, weight, child_2=None):
        self.child_1 = child_1
        self.weight = weight
        self.child_2 = child_2

    def __str__(self, depth=0):
        result = f"#{self.weight} "
        if(self.child_2 is None):
            result += self.child_1 + "\n"
        else:
            result += "\n"
            result += " " * (depth + 1) + "0 -> " + self.child_1.
 ↪__str__(depth+1)
            result += " " * (depth + 1) + "1 -> " + self.child_2.
 ↪__str__(depth+1)
        return result

    def __lt__(self, other):
        return self.weight < other.weight
```

[5]:
```
print(Node(Node("a", 1), 2, Node("b", 1)))
```

```
#2
 0 -> #1 a
 1 -> #1 b
```

[8]:
```
print(huffman({"a": 5, "b": 2, "c": 1, "d": 1, "r": 2}))
```

```
#11
 0 -> #5 a
 1 -> #6
  0 -> #2
   0 -> #1 c
   1 -> #1 d
  1 -> #4
   0 -> #2 b
   1 -> #2 r
```

10 Twierdzenie (3.2)

Algorytm Huffmana tworzy drzewo prefiksów w czasie $O(|\Sigma|log|\Sigma|)$

Dowód: dominującą operacją jest posortowanie liter względem częstości występowania. Pozostałe oparacje mają złożoność liniową ze względu na liczbę liter.

11 Dynamiczny algorytm budowy drzewa Huffmana

- Wadą zwykłego algorytmu jest fakt, że tekst jest czytany 2 razy
- dynamiczny algorytm czyta tekst tylko raz
- idea algorytmu polega na budowaniu kolejnych drzew T_{za} na podstawie T_z, gdzie z to łańuch znaków, natomiast a, to pojedynczy znak
- drzewo budowane jest nad alfabetem $\Sigma \cup \{\#\}$, gdzie # reprezentuje wszystkie znaki, które nie pojawiły się w fragmencie z, wtedy drzewo T_z jest drzewem Huffmana, posiadającym następujący koszty:
  - n_a - liczba wystąpień znaku a w z
  - 0 - dla spejcanlego znaku #

https://pypi.org/project/bitarray/

[35]:
```
# ! pip install bitarray treelib

from bitarray import bitarray

# tworzenie tablicy bitów na podstawie reprezentacji w formie łańcucha znaków
array = bitarray('1010')
print(array)

# odczytywanie pojedynczych bitów
print(array[0])
print(array[1])
```

```
bitarray('1010')
True
False
```

[36]:
```
# dodawanie dwóch tablic
array = bitarray('0100')
array += bitarray('0001')
print(array)

#zamiana tablicy na łańcuch znaków, który można zapisać do pliku
print(array.tobytes())
```

```
bitarray('01000001')
b'A'
```

[37]:
```
# zamiana łańcucha znaków (np. odczytanego z pliku) na tablicę bitów
array = bitarray()
array.frombytes(b'A')
print(array)

array = bitarray()
array.frombytes(b'AB')
print(array)
```

```
bitarray('01000001')
bitarray('0100000101000010')
```

[39]:
```
# usuwanie pierwszych 8 elementów tablicy
array = bitarray()
array.frombytes(b'AB')
array[:8] = bitarray()
print(array)
```

```
bitarray('01000010')
```

[ ]:
```
from collections import defaultdict

def adaptive_huffman(text):
    Node.nodes = []
    count = defaultdict(int)
    nodes = {"#": Node("#", weight=0)}
    root = nodes["#"]
    for letter in list(text):
        if letter in nodes:
            node = nodes[letter]
            print(node.code() + ' ' + node.letter)
            node.increment()
        else:
            updated_node = nodes["#"]
            print(updated_node.code()  + ' ' + updated_node.letter)
            print("{0:b}".format(ord(letter)) + ' ' + letter)
            node = Node(letter, parent=updated_node)
            nodes[letter] = node
            del nodes["#"]
            zero_node = Node("#", parent=updated_node, weight=0)
            updated_node.add_child(0, zero_node)
            updated_node.add_child(1, node)
            nodes["#"] = zero_node
            updated_node.increment()
```

[63]:
```
from collections import defaultdict
```

```
def adaptive_huffman(text):
    Node.nodes = []
    count = defaultdict(int)
    nodes = {"#": Node("#", weight=0)}
    root = nodes["#"]
    print(root)
    for letter in list(text):
        if letter in nodes:
            node = nodes[letter]
            print(node.code() + ' ' + node.letter)
            node.increment()
        else:
            updated_node = nodes["#"]
            print(updated_node.code()  + ' ' + updated_node.letter)
            print("{0:b}".format(ord(letter)) + ' ' + letter)
            node = Node(letter, parent=updated_node)
            nodes[letter] = node
            del nodes["#"]
            zero_node = Node("#", parent=updated_node, weight=0)
            updated_node.add_child(0, zero_node)
            updated_node.add_child(1, node)
            nodes["#"] = zero_node
            updated_node.increment()
        #print(root)
```

[53]:
```
class Node():
    nodes = []

    def __init__(self, letter, weight=1, parent=None):
        self.letter = letter
        self.weight = weight
        self.parent = parent
        self.children = [None, None]
        Node.nodes[0:0] = [self]
        self.index = -len(Node.nodes)
        self.label = None
        self.id = len(Node.nodes)

    def add_child(self, index, node):
        self.children[index] = node
        node.label = index

    def increment(self):
        self.weight += 1
        if self.parent:
            for i in range(len(Node.nodes) + self.index + 1, len(Node.nodes)):
```

```
                #print([str(f"#{e.weight}({e.index},{e.label},{e.id}) ") for e␣
↪in Node.nodes])
                if(self.weight <= Node.nodes[i].weight):
                    other_node = Node.nodes[i-1]
                    if(self != other_node and self.parent != other_node):
                        #print(f"swapping\n{self}\nand\n{other_node}")
                        Node.nodes[self.index] = other_node
                        Node.nodes[other_node.index] = self
                        self.index, other_node.index = other_node.index, self.
↪index
                        self.parent.children[self.label] = other_node
                        other_node.parent.children[other_node.label] = self
                        self.parent, other_node.parent = other_node.parent,␣
↪self.parent
                        self.label, other_node.label = other_node.label, self.
↪label
                    break
            self.parent.increment()

    def code(self):
        if(self.parent):
            return self.parent.code() + str(self.label)
        else:
            return ''

    def __str__(self, depth=0):
        result = f"#{self.weight}({self.index},{self.label}) "
        #result = ""
        if(self.children[0] is None):
            result += self.letter + "\n"
        else:
            result += "\n"
            result += " " * (depth + 1) + "0 -> " + self.children[0].
↪__str__(depth+1)
            result += " " * (depth + 1) + "1 -> " + self.children[1].
↪__str__(depth+1)
        return result
```

[64]:
```
adaptive_huffman('abracadabra')
```

```
#0(-1,None) #

 #
1100001 a
0 #
1100010 b
00 #
```

```
1110010 r
0 a
100 #
1100011 c
0 a
1100 #
1100100 d
0 a
110 b
110 r
0 a
```

[62]:
```
adaptive_huffman('bookkeeper')
```

```
#0(-1,None) #

 #
1100010 b
#1(-1,None)
 0 -> #0(-3,0) #
 1 -> #1(-2,1) b

0 #
1101111 o
#2(-1,None)
 0 -> #1(-3,0)
  0 -> #0(-5,0) #
  1 -> #1(-4,1) o
 1 -> #1(-2,1) b

01 o
#3(-1,None)
 0 -> #1(-3,0)
  0 -> #0(-5,0) #
  1 -> #1(-4,1) b
 1 -> #2(-2,1) o

00 #
1101011 k
#4(-1,None)
 0 -> #2(-3,0)
  0 -> #1(-5,0)
   0 -> #0(-7,0) #
   1 -> #1(-6,1) k
  1 -> #1(-4,1) b
 1 -> #2(-2,1) o

001 k
```

```
#5(-1,None)
 0 -> #2(-3,0) o
 1 -> #3(-2,1)
  0 -> #1(-5,0)
   0 -> #0(-7,0) #
   1 -> #1(-6,1) b
  1 -> #2(-4,1) k

100 #
1100101 e
#6(-1,None)
 0 -> #2(-3,0) o
 1 -> #4(-2,1)
  0 -> #2(-5,0)
   0 -> #1(-7,0)
    0 -> #0(-9,0) #
    1 -> #1(-8,1) e
   1 -> #1(-6,1) b
  1 -> #2(-4,1) k

1001 e
#7(-1,None)
 0 -> #3(-3,0)
  0 -> #1(-7,0)
   0 -> #0(-9,0) #
   1 -> #1(-8,1) b
  1 -> #2(-6,1) e
 1 -> #4(-2,1)
  0 -> #2(-5,0) o
  1 -> #2(-4,1) k

000 #
1110000 p
#8(-1,None)
 0 -> #4(-3,0)
  0 -> #2(-7,0)
   0 -> #1(-9,0)
    0 -> #0(-11,0) #
    1 -> #1(-10,1) p
   1 -> #1(-8,1) b
  1 -> #2(-6,1) e
 1 -> #4(-2,1)
  0 -> #2(-5,0) o
  1 -> #2(-4,1) k

01 e
#9(-1,None)
 0 -> #4(-3,0)
```

```
  0 -> #2(-7,0)
   0 -> #1(-9,0)
    0 -> #0(-11,0) #
    1 -> #1(-10,1) p
   1 -> #1(-8,1) b
  1 -> #2(-6,1) k
 1 -> #5(-2,1)
  0 -> #2(-5,0) o
  1 -> #3(-4,1) e

0000 #
1110010 r
#10(-1,None)
 0 -> #4(-3,0)
  0 -> #2(-7,0) o
  1 -> #2(-6,1) k
 1 -> #6(-2,1)
  0 -> #3(-5,0)
   0 -> #1(-9,0) b
   1 -> #2(-8,1)
    0 -> #1(-11,0)
     0 -> #0(-13,0) #
     1 -> #1(-12,1) r
    1 -> #1(-10,1) p
  1 -> #3(-4,1) e
```

12 Twierdzenie 3.3 (własność rodzeństwa)

Niech T oznacza kompletne, binarne, wagowane drzewo (z n liśćmi), w których liście mają dodatni koszt, a koszt dowolnego węzła wewnętrznego stanowi koszt jego dzieci. T jest drzewem Huffmana wtedy i tylko wtedy, gdy jego liście mogą zostać uporządkowane w następujący sposób: 1. ciąg kosztów $(c(x_1), c(x_2), c(c_3), \ldots)$ jest niemalejący, 2. $\forall_{1 \leq i < n} : P(x_{2i-1}) = P(x_{2i})$, gdzie $P(x)$ to rodzic x.

Problem stanowi dowiedzenie $\leftarrow$, ponieważ $\rightarrow$ wynika wprost z algorytmu Huffmana. Czyli chcemy udowodnić, że mając te dwie własności, dostaniemy drzewo Huffmana.

Dowód jest indukcyjny. Zaczynamy od jakiegoś drzewa T^i, w którym mamy węzły x_1 oraz x_2. Wiadomo, że są to liście, ponieważ koszt jest niemalejący. Liście te mogą być wybrane w kroku algorytmu (choć nie muszą), ponieważ ich koszt jest najniższy. Przyjmijmy, że x to ich rodzic. Rozważmy drzewo T^{i-1}, w którym nie ma liści x_1 oraz x_2. Przyjmując założenie indukcyjne, wywodzimy, że ze spełnienia warunków 1 i 2 drzewo jest drzewem Huffmana. Zatem drzewo T^i również jest drzewem Huffmana. Warto zwrócić uwagę, że z własności T^{i-1} wywodzimy własność T^i, a nie na odwrót.

13 Kompresja ze zmienną długością bloku

Optymalna kompresja ze zmienną długościa bloku musi uwzględnić 3 zagadnienia: * znaleźć zbiór podsłów F, na który należy podzielić kompresowany tekst T, * znaleźć optymalny sposób podziału tekstu T na podsłowa ze zbioru F, * znaleźć kod C, który będzie odpowiadał podsłowom, ze zbioru F.

Optymalna kompresja ze zmienną długością bloku jest problemem NP-zupełnym.

14 Podział tekstu na podsłowa - strategia zachłanna

Dopasuj zawsze najdłuższe słowo występujące w słowniku.

- Jeśli F ogranicza się do liter i podsłów 2-elementowych, to strategia zachłanna jest optymalna.
- Jeśli F zawiera wszystkie swoje pod-słowa, to strategia zachłanna jest optymalna.

15 Strategia semi-zachłanna

- niech $m = max\{|uv| : u, v \in F \wedge uv \sqsubset s\}$;
- niech $f_1 \in F$ taki, że $f_1 v \sqsubset s \wedge |f_1 v| = m$, dla pewnego v;
- wybierz f_1 jako kolejny element do zakodowania;
- kontynuuj proces dla s skróconego o f_1.

Strategia semi-zachłanna jest optymalna, jeśli F zawiera wszystkie swoje prefiksy.

16 Algorytm Lempel-Ziv-Welch

[25]:
```
from math import ceil
def lempel_ziv_welch(text):
    prefixes = []
    while(len(text) > 0):
        prefix = max_prefix(text, prefixes)
        letter = text[len(prefix)]
        # wypisz indeks prefix-u na ceil(log2(len(prefixes))) bitach
        # wypisz początkowy kod letter na ceil(log(len(alphabet))) bitach
        prefixes.append(prefix + letter)
        text= text[len(prefix)+1:]
```

Przykład algorytmu, w którym kodowany fragment tekstu ma zmienną długość. W praktyce rozmiar tablicy prefiksów ogranicza się z góry oraz dodaje się do niej wszystkie litery alfabetu.