VERİ SIKIŞTIRMA

Sözlük Tabanlı Yöntemler
Bölüm 6

Prof.Dr. Banu DİRİ

- İstatistiksel sıkıştırma yöntemleri, verinin istatistiksel modelini kullanır ve sıkıştırmanın kalitesi seçilen modele bağlıdır
- Sözlük tabanlı sıkıştırma yöntemleri ise, ne istatistiksel modelleri ne de değişken uzunluktaki kodları kullanmazlar
- Sadece sözlük ve sembollerden oluşan string'ler «token» kullanılır
- Sözlükler statik veya dinamik olarak tutulurlar. Statik sözlüklerde, sözlük sabit olup, bazen sözlüğe yeni bir string'in eklenmesine izin verilir ancak silinmez
- Dinamik sözlükte ise input stream'de daha önce bulunmuş olan string'ler tutulur, yani giriş değerleri okunurken ekleme ve silme işlemi yapılabilir

- Statik sözlüğün en basit örneği, İngilizce dokümanları sıkıştırmak için kullanılan İngilizce diline ait sözlüktür (kelimeler ek aldığı için Türkçe dili için uygun olmaz)
- Tanımları olmaksızın yarım milyon kelimeden oluşan bir sözlük düşünün
- Bir kelime input stream'den okunur ve sözlükte aranır. Eğer kelime sözlükte bulunuyorsa, sözlükteki sırası ile output stream'e yazılır
- Eğer kelime sözlükte bulunmaz ise kelime sıkıştırılmadan output stream'e aynen yazılır
- Sözlükte yarım milyon kelime varsa, 2^{19}=524.288 kelime için, index alanı olarak 19 bit'e ihtiyaç vardır
- Output stream, kelimenin sözlükteki sıra değeri (index) ve ham kelimelerden oluşur
- Bu iki değeri birbirinden ayırt etmemiz gerekir. Bunun için extra bir bit kullanılır

Örnek

Karakter uzunluğu 5 olan bir kelime sözlükte var ise toplamda 20 bit gerekir

1 bit (0)	19 bit (index)
flag	kelime

Kelime sözlükte yok ise toplamda 48 bit gerekir

1 bit (1)	7 bit	1 byte	1 byte	1 byte	1 byte	1 byte
flag	#karakter	Kelime (1.harfi)	Kelime (2.harfi)	Kelime (3.harfi)	Kelime (4.harfi)	Kelime (5.harfi)

Soru

5 karakter uzunluğundaki kelimeden kaç tanesi sözlükte bulunmalıdır ki sıkıştırmadan bahsedilebilsin

P → çakışma olasılığı

$$N[\ 20*P + 48(1-P)] = N40$$
$$N[-28P + 48] = N40$$
$$28P = 8$$

$P = 0{,}29 \quad \rightarrow P > 0{,}29$

* Özel durumlarda, birkaç yüz kelimeden oluşan statik sözlüklerin kullanılması daha iyi sonuç verecektir

String Sıkıştırma

Sembollerden oluşan string'lere dayalı olan sıkıştırma yöntemleri, tek bir sembole dayalı olan yöntemlerden daha etkilidir.

İki sembollü bir alfabe olsun (a1→ P1=0,8 a2 → P2 = 0,2)

Ortalama olasılık değeri = 0,5

Varyansı |0,8 – 0,5| + |0,2 – 0,5| = 0,6

Bu iki sembol **1** bit ile kodlanır

Kodun ortalama uzunluğu 1 bit/sembol

String	Probability	Code
a_1a_1	$0.8 \times 0.8 = 0.64$	0
a_1a_2	$0.8 \times 0.2 = 0.16$	11
a_2a_1	$0.2 \times 0.8 = 0.16$	100
a_2a_2	$0.2 \times 0.2 = 0.04$	101

a1→ P1=0,8 a2 → P2 = 0,2

Aynı şekilde iki sembollü string'ler oluşturalım

Ortalama olasılık değeri $\frac{1}{4}$ = 0,25

Varyans |0,64 – 0,25| + |0,16 – 0,25| + |0,16 – 0,25| + |0,04 – 0,25| = 0,78

Huffman Ort. kod uzunluğu 1*0,64 + 2*0,16 + 3*0,16 + 3*0,04 = 1,56 bits/string

1,56/2 = 0,78

String	Probability	Code
$a_1a_1a_1$	$0.8 \times 0.8 \times 0.8 = 0.512$	0
$a_1a_1a_2$	$0.8 \times 0.8 \times 0.2 = 0.128$	100
$a_1a_2a_1$	$0.8 \times 0.2 \times 0.8 = 0.128$	101
$a_1a_2a_2$	$0.8 \times 0.2 \times 0.2 = 0.032$	11100
$a_2a_1a_1$	$0.2 \times 0.8 \times 0.8 = 0.128$	110
$a_2a_1a_2$	$0.2 \times 0.8 \times 0.2 = 0.032$	11101
$a_2a_2a_1$	$0.2 \times 0.2 \times 0.8 = 0.032$	11110
$a_2a_2a_2$	$0.2 \times 0.2 \times 0.2 = 0.008$	11111

Str. size	Variance of prob.	Avg. size of code
1	0.6	1
2	0.78	0.78
3	0.792	0.728

Üç sembolden oluşan string, olasılık değerleri ve Huffman kodları yukarıdaki gibi verilmiş olsun.

Ortalama olasılık değeri = 1/8 = 0,125

Varyans = |0,512-0,125| + 3*|0,128-0,125| + 3*|0.032-0,125| + |0.008-0,125| = 0.792

Ort. Huffman kod uzunluğu 1*0,512+3*3*0,128+3*5*0,032+5*0,008 = 2.184 bits/string

2.1846/3 = 0,728 bits/symbol

LZ ve Türev Algoritmalar

- Jacob Ziv ve Abraham Lempel isimli iki araştırmacı 1970'li yıllarda LZ77 ve LZ78 adını verdikleri ilk Dictionary_based tabanlı sıkıştırma algoritmasını geliştirdi
- Bu çalışma referans alınarak arkasından benzer yöntemler geliştirilmiştir

LZSS → Storee-Szymanski

LZFG → Fiala-Greane

LZRW1 → Ross-Williams

LZW → Welch

LZMW → Miller-Wegman

LZAP → (All prefix)

LZY → (Yabba)

LZP → (Predictin)

LZ77 (Kayan Pencere - Sliding Window)

- Bu yöntemdeki ana fikir, önceden input stream'den okunmuş olan bilgiyi sözlük gibi kullanmaktır
- Kodlayıcı, input stream için bir pencere (window) oluşturur ve kodlanan input değerlerini sağdan sola doğru bu pencere içerisinde öteler
- Bu yöntem kayan pencere (sliding window) mantığı üzerine kurulmuştur

- Pencere iki parçadan oluşur. Sol taraftaki bölüm Arama Alanı (Search Buffer) olup, mevcut sözlüktür. En son okunan ve kodlanan sembollerden oluşur
- Pencerenin sağındaki bölüm ise Kaynak Kod Alanı (Look Ahead Buffer) olup, henüz kodlanmamış sembollerden oluşur
- Pratikte Arama Alanı binlerce byte uzunluğunda iken Kaynak Kod Alanı sadece onlarca byte uzunluğundadır

←------ coded text

sir sid eastman easily t	eases sea sick seals

------←---- text to be read

- Kodlayıcı, Kaynak Kod Alanının ilk sembolü olan «**e**» yi Arama Alanın da sağdan sola doğru aramaya başlar
- İlk «**e**» harfini «**e**asily» kelimesinde yakalar
- «**e**» nin offset değeri sağdan sayıldığında 8'dir (arama alanında, 0 dan başlar)
- Kodlayıcı «**e**» sembolünü ikinci kez «**e**astman» kelimesinde yakalar
- Kaynak Kod Alanındaki «**eas**» üç sembol olarak Arama Alanında yakalanır
- Bu işleme en uzun eşleşen string bulununcaya kadar devam ettirilir
- Kodlayıcı en uzun çakışmayı bulduktan sonra, en son bulunan çakışmayı referans alarak **(16, 3, «e»)** token'ı hazırlar

- En uzaktaki çakışmanın token olarak alınması, algoritmayı basitleştirir ancak, offset değeri büyür
- İlk çakışma alınırsa offset değeri küçüktür, ancak algoritma karmaşıklaşır (geri dönüş için ekstra işlem yapmak gerekir)
- Oluşturulan token üç bölümden oluşur

(offset, length, next symbol)

(başlangıç adresi, çakışma uzunluğu, gelecek sembol)

- Hazırlanan token output stream'e yazılır ve pencere sola doğru dört pozisyon ötelenir (3 tanesi çakışan ve dördüncüsü de gelecek semboldür)
- Eğer geriye doğru aramada hiç çakışma oluşmamış ise LZ77 0 offset değeri, uzunluk ve çakışmayan sembol ile bir token hazırlar ve output stream'e yazar
- Bu durum sıkıştırma işleminin başlarında meydana gelir (başlangıçta arama alanı boş olduğundan)

	sir␣sid␣eastman␣	⇒	(0,0,"s")
s	ir␣sid␣eastman␣e	⇒	(0,0,"i")
si	r␣sid␣eastman␣ea	⇒	(0,0,"r")
sir	␣sid␣eastman␣eas	⇒	(0,0,"␣")
sir␣	sid␣eastman␣easi	⇒	(4,2,"d")

(0, 0, «s») ile sadece bir tek sembol kodlanır ve bu durum iyi bir sıkıştırma sağlamaz

- Search Buffer'ın uzunluğu «S» ise offsetin uzunluğu $\lceil \log_2 S \rceil$ olur
- Pratikte Search Buffer, birkaç byte uzunluğunda ise, offsetin uzunluğu 10-12 bits olur
- Look Ahead Buffer'ın uzunluğu «L» ise lenght alanının uzunluğu $\lceil \log_2 (L-1) \rceil$ olur
- Pratikte Look Ahead Buffer birkaç on byte dan oluşuyorsa, lenght field'ının uzunluğu birkaç bit olur
- Sembol field'nın uzunluğu da 8 bittir. Yaklaşık olarak bir token uzunluğu 11+5+8 = 24 bittir.

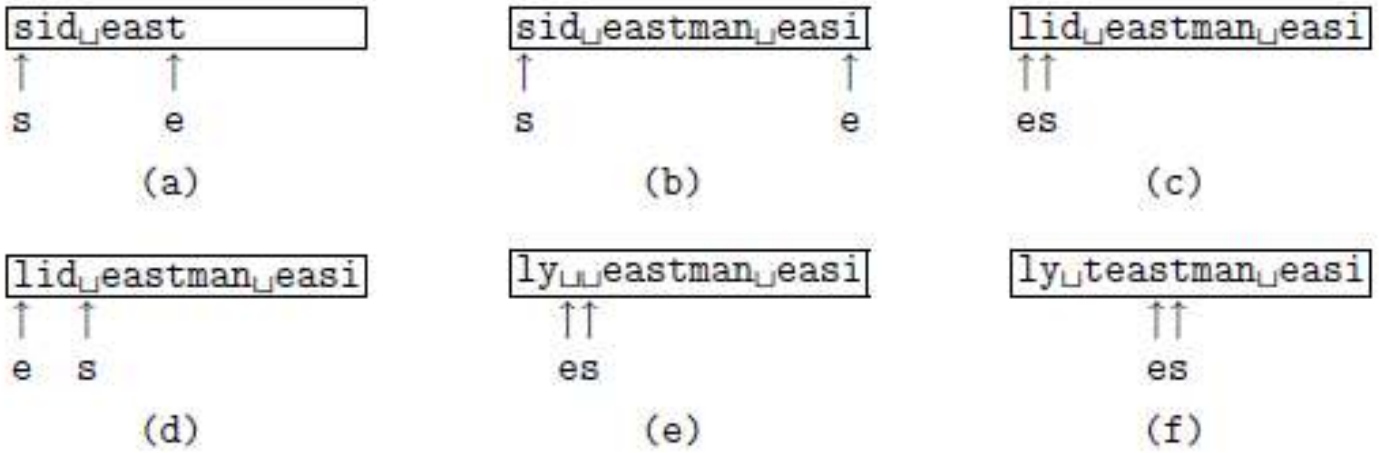
- Kod çözücü kodlayıcıdan daha basittir
- Kodlayıcı pencere boyutuna eşit bir buffer alanı oluşturmalıdır
- Kod çözücü token'ı okur, buffer'ında bulduğunda çakışan sembolleri ve daha sonra da üçüncü alanı output stream'e yazar ve buffer alanını öteler
- Bu yöntemler, bir kez sıkıştırılan çoğu kez açılarak kullanılan uygulamalar için elverişlidir (arşiv)
- Bu yöntemde, okunan veri pattern'ları birbirlerine ne kadar yakınsa sıkıştırma performansı da o kadar iyidir

LZ77 birçok araştırmacı tarafından geliştirilmiştir

- Offset ve Length alanları için değişken uzunluklar denenmiş
- Her iki buffer'ın boyutları değiştirilmiş
- Search Buffer'ın boyu artırıldığında, daha iyi çakışmalar elde edilmiş ancak arama zamanı artmıştır
- Kayan pencere mantığında **Lineer Window** yerine **Circular Queue m**antığı kullanılmıştır

Dairesel Kuyruk (Circular Queue)

- Bu yöntemde «s (start) ve «e» (end) olmak üzere iki pointer mevcuttur
- Karakterler «end» pointer'dan eklenir, «start» pointer'dan çıkarılır
- Dairesel kueruğa yeni semboller eklendikçe pointer'lar yer değiştirir



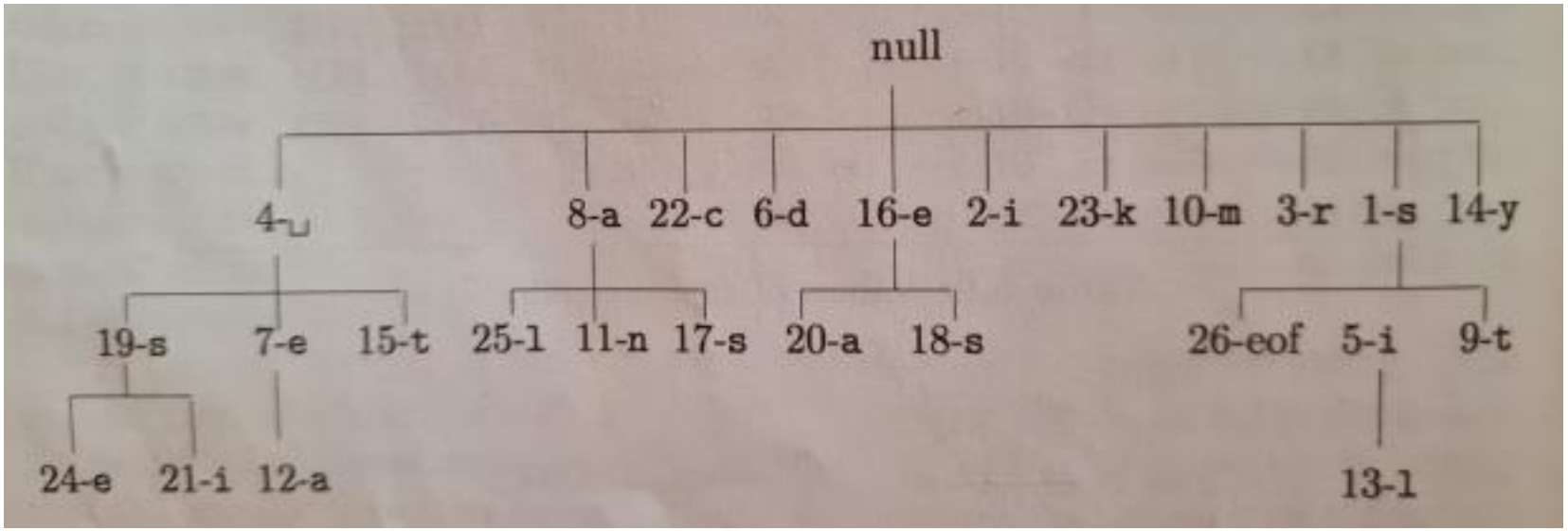
LZ78

- LZ77 alg. olduğu gibi kayan pencere, arama ve kaynak kod alanı kullanılmaz
- Önceden karşılaşılmış olan string'lerden oluşan bir sözlük kullanılır
- Boş olarak başlayan sözlüğün boyu kullanılabilir hafıza ile sınırlıdır
- Kodlayıcı, output stream'e iki alanlı bir token yazar
- Token'nın ilk alanı sözlüğün «**sıra belirleyicisi**», ikinci alanı da «**sembol**» dür
- Input stream'den gelen bir «**w**» sembolü, sözlükte aranır
- Çakışma yoksa, sözlükteki en son sıraya sıra belirleyicisi «0» değerini alarak yerleşir
- Eğer bir çakışma olmuş ise, bir sonraki sembol «**k**» bir önceki sembole eklenerek «**wk**» oluşturulur ve sözlükte arama işlemine devam edilir
- Bu işleme en uzun string, sözlükte bulununcaya kadar devam ettirilir
- «**wk**» sözlükte bulunmaz ise, sözlüğün en son sırasına yerleştirilir
- Çıkışa yazılacak olan token'nın ilk alanına **en son çakışmanın olduğu sıra**, ikinci alanına da «**k**» yerleştirilir

"sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals".

Dictionary		Token	Dictionary		Token
0	null				
1	"s"	(0, "s")	8	"a"	(0, "a")
2	"i"	(0, "i")	9	"st"	(1, "t")
3	"r"	(0, "r")	10	"m"	(0, "m")
4	"␣"	(0, "␣")	11	"an"	(8, "n")
5	"si"	(1, "i")	12	"␣ea"	(7, "a")
6	"d"	(0, "d")	13	"sil"	(5, "l")
7	"␣e"	(4, "e")	14	"y"	(0, "y")

Sözlük için en iyi veri yapısı «trie» ağaçlarıdır.



- LZ78 algoritmasında sözlüğe sembol ekleme işlemine, sözlük dolana kadar devam edilir

Sözlük dolduğunda

- Gelen semboller sözlüğe eklenmeden kodlanır (FROZEN)
- Bütün ağaç silinir ve yeni boş bir ağaç ile işleme başlanır
- En az kullanılan sembol sözlükten çıkarılıp, yerine yenisi eklenir

LZW – Lempel Ziv Welch- Kodlama

- Geniş bir kodlama alanına sahiptir
- Codebook dinamik olarak oluşturulur
- Kodların boyutu, codebook uzunluğuna bağlı olarak değişir (codebook boyu 8 ise sadece 8 adet codeword oluşturulur)
- Codebook'un başlangıcını kullanılan alfabe oluşturur
- Input stream'den her okunan sembol ile sözlük oluşturulmaya başlanır

Örnek **Input stream** abbbaacbbabbaacacaabb

0	0000	a
1	0001	b
2	0010	c
3	0011	ab
4	0100	bb
5	0101	bba
6	0110	aa
7	0111	ac
8	1000	cb
9	1001	bbab
10	1010	bbaa
11	1011	aca
12	1100	acaa
13	1101	abb

abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
bbbaacbbabbaacacaabb
bbbaacbbabbaacacaabb
bbaacbbabbaacacaabb
bbaacbbabbaacacaabb
bbaacbbabbaacacaabb
aacbbabbaacacaabb
aacbbabbaacacaabb
acbbabbaacacaabb
acbbabbaacacaabb
cbbabbaacacaabb
cbbabbaacacaabb
bbabbaacacaabb
bbabbaacacaabb
bbabbaacacaabb
bbabbaacacaabb
bbaacacaabb
bbaacacaabb
bbaacacaabb
bbaacacaabb
acacaabb
acacaabb
acacaabb

acaabb
acaabb
acaabb
acaabb
abb
abb
abb

0	0000	a
1	0001	b
2	0010	c
3	0011	ab
4	0100	bb
5	0101	bba
6	0110	aa
7	0111	ac
8	1000	cb
9	1001	bbab
10	1010	bbaa
11	1011	aca
12	1100	acaa
13	1101	abb

Input stream

abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb
abbbaacbbabbaacacaabb

.........acaabb
.........acaabb
.........acaabb
.........acaabb

Output stream 0 1 4 0 0 2 5 5 7 11 3 1