Data Compression

Basics + Huffman coding

How much can we compress?

Assuming all input messages are valid, if even one string is (lossless) compressed, some other must expand.

Take all messages of length *n.*

Is it possible to compress ALL OF THEM in less bits ?

NO, they are 2^n but we have less compressed msg…

$$\sum_{i=1}^{n-1} 2^i = 2^n - 2$$

We need to talk about stochastic sources

Entropy (Shannon, 1948)

For a set of symbols S with probability $p(s)$, the **self information** of s is:

$$i(s) = \log_2 \frac{1}{p(s)} = -\log_2 p(s) \quad \text{bits}$$

Lower probability → higher information

Entropy is the weighted average of $i(s)$

$$H(S) = \sum_{s \in S} p(s) * \log_2 \frac{1}{p(s)}$$

Statistical Coding

How do we use probability p(s) to encode s?

- **Prefix codes and relationship to Entropy**
- **Huffman codes**
- **Arithmetic codes**

Uniquely Decodable Codes

A **variable length code** assigns a bit string (codeword) of variable length to every symbol

e.g. a = 1, b = 01, c = 101, d = 011

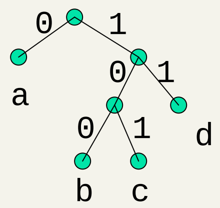
What if you get the sequence 1011 ?

A **uniquely decodable code** can always be uniquely decomposed into their codewords.

Prefix Codes

A **prefix code** is a variable length code in which no codeword is a prefix of another one

e.g a = 0, b = 100, c = 101, d = 11
Can be viewed as a binary trie



Average Length

For a code C with codeword length $L[s]$, the **average length** is defined as

$$L_a(C) = \sum_{s \in S} p(s) * L[s]$$

We say that a prefix code C is **optimal** if for all prefix codes C', $L_a(C) \leq L_a(C')$

A property of optimal codes

***Theorem* (Kraft-McMillan).** For any optimal uniquely decodable code, it does exist a prefix code with the same symbol lengths and thus same average optimal length. And vice versa…

***Theorem* (golden rule).** If C is an optimal prefix code for a source with probabilities $\{p_1, \ldots, p_n\}$ then $p_i < p_j \Rightarrow L[s_i] \geq L[s_j]$

Relationship to Entropy

Theorem (lower bound, Shannon). *For any probability distribution and any uniquely decodable code C, we have*

$$H(S) \leq L_a(C)$$

Theorem (upper bound, Shannon). *For any probability distribution, there exists a prefix code C such that*

$$L_a(C) < H(S) + 1$$

Shannon code takes $\lceil \log 1/p \rceil$ bits

Huffman Codes

Invented by Huffman as a class assignment in '50.

Used in most compression algorithms

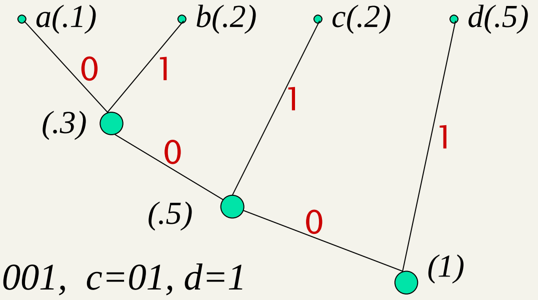
- gzip, bzip, jpeg (as option), fax compression,…

Properties:

- Generates optimal **prefix** codes
- Cheap to encode and decode
- L_a(Huff) = H if probabilities are powers of 2
  - Otherwise, at most 1 bit more per symbol!!!

Running Example

p(a) = .1, p(b) = .2, p(c) = .2, p(d) = .5



a=000, b=001, c=01, d=1

There are 2^{n-1} "equivalent" Huffman trees

What about ties (and thus, tree depth) ?

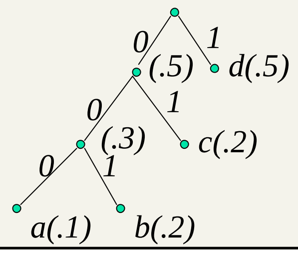
Encoding and Decoding

Encoding: Emit the root-to-leaf path leading to the symbol to be encoded.

Decoding: Start at root and take branch for each bit received. When at leaf, output its symbol and return to root.

abc... → 00000101

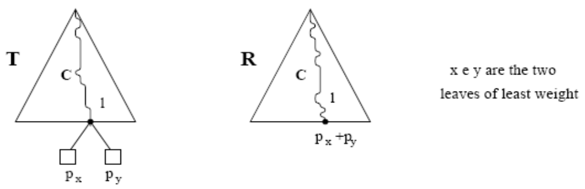
101001... → dcb



A property on tree contraction

Lemma 2.16 *The relation between a tree T and his reduced tree R, according to the average length, is* $L_T = L_R + (p_x + p_y)$.

Proof:



Something like substituting symbols **x,y** with one new symbol **x+y**

...by induction, optimality follows...

$$L_H = L_{R_H} + (p_n + p_{n-1}) \overset{hyp}{=} L_{opt}[p_n + p_{n-1}, p_1, \ldots, p_{n-2}] + (p_n + p_{n-1})$$

$$\leq L_R + (p_n + p_{n-1}) = L_T \Rightarrow L_H \leq L_T$$

Optimum vs. Huffman

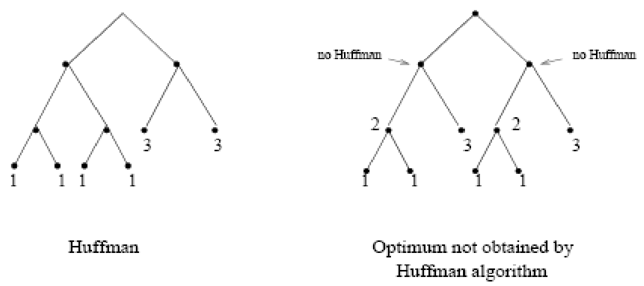
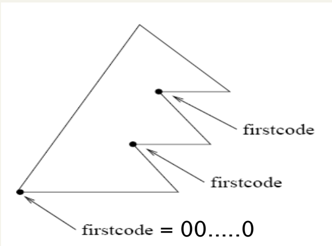


Figure 2.17: Example of a optimum code not obtained by Huffman coding

Model size may be large

Huffman codes can be made *succinct* in the representation of the codeword tree, and *fast* in (de)coding.

Canonical Huffman tree



We store for any level L:

- firstcode[L]
- Symbol[L,i], for each i in level L

This is ≤ $h^2 + |\Sigma| \log |\Sigma|$ bits

Canonical Huffman

Encoding

$$firstcode[l_{max}] = 0;$$

$$\begin{array}{l} for\ \ l = l_{max} - 1\ \ downto\ \ 1 \\ \quad firstcode[l] = (firstcode[l+1] + num[l+1])/2; \end{array}$$

Note that the longest codeword will be $\overbrace{00\ldots0}^{l_{max}}$.

Symbol i	Code length l_i
1	2
2	5
3	5
4	3
5	2
6	5
7	5
8	2

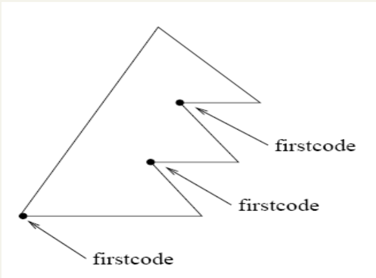
	1	2	3	4	5
num[*l*]	0	3	1	0	4
firstcode[*l*]					0

Canonical Huffman

Decoding

$v = 0; \quad l = 1;$

$\underline{while} \quad v < firstcode[l] \quad \underline{do} \quad \begin{cases} v = 2v + next\_bit(); \\ l++; \end{cases}$



firstcode[1]=2
firstcode[2]=1
firstcode[3]=1
firstcode[4]=2
firstcode[5]=0

T=...00010...

Symbol i	Code length l_i	*codeword*[i]	Bit pattern
1	2	1	01
2	5	0	00000
3	5	1	00001
4	3	1	001
5	2	2	10
6	5	2	00010
7	5	3	00011
8	2	3	11

num[l]	0	3	1	0	4
firstcode[l]	2	1	1	2	0

Problem with Huffman Coding

Consider a symbol with probability .999. The self information is

$$-\log(.999) = .00144$$

If we were to send 1000 such symbols we might hope to use 1000*.0014 = 1.44 bits.

Using Huffman, we take at least 1 bit per symbol, so we would require 1000 bits.

What can we do?

Macro-symbol = **block** of k symbols

☺ 1 extra bit per macro-symbol = 1/k extra-bits per symbol

☹ Larger model to be transmitted

Shannon took infinite sequences, and k → ∞ !!

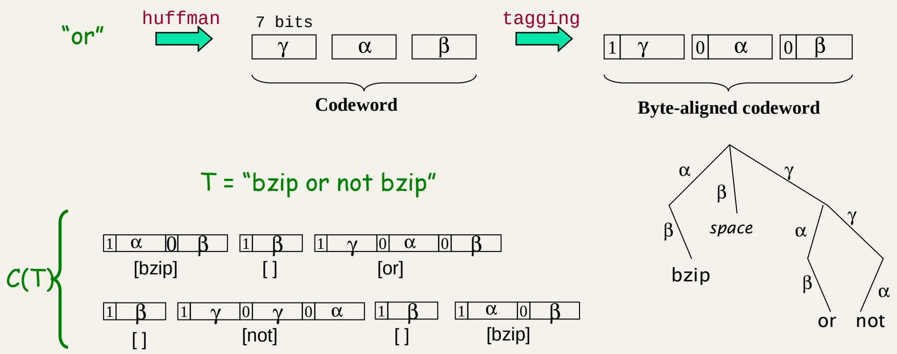
In practice, we have:

- Model takes $|\Sigma|^k$ (k * log $|\Sigma|$) + h^2 (where h might be $|\Sigma|$)
- It is $H_0(S_L) \leq L * H_k(S) + O(k * \log |\Sigma|)$, for each $k \leq L$

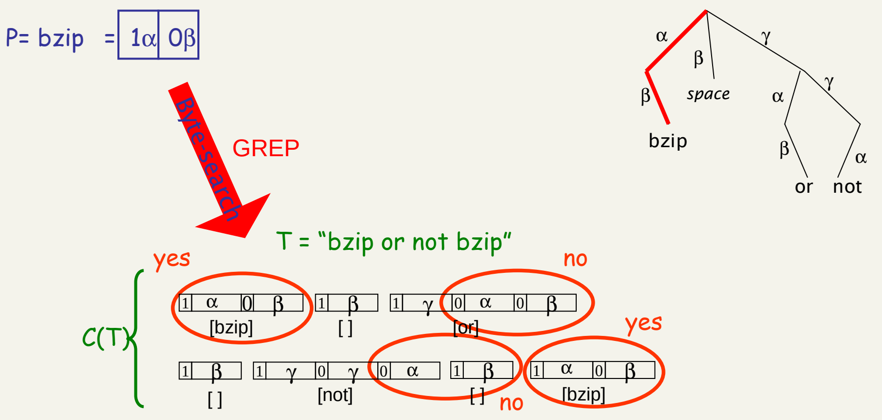
Compress + Search ? [Moura *et al*, 98]

Compressed text derived from a word-based Huffman:

- ✓ Symbols of the huffman tree are the *words* of T
- ✓ The Huffman tree has fan-out *128*
- ✓ Codewords are *byte-aligned* and *tagged*

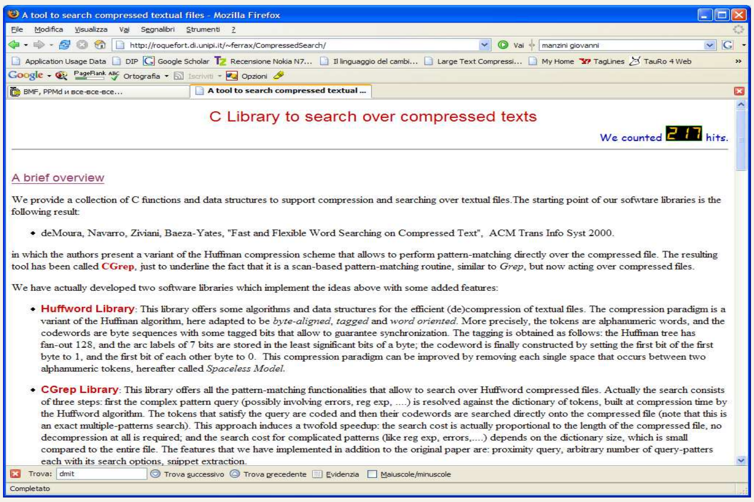


CGrep and other ideas...



Speed ≈ Compression ratio

You find this at



A tool to search compressed textual files - Mozilla Firefox

http://roquefort.di.unipi.it/~ferrax/CompressedSearch/

C Library to search over compressed texts

We counted 217 hits.

A brief overview

We provide a collection of C functions and data structures to support compression and searching over textual files.The starting point of our sofwtare libraries is the following result:

- deMoura, Navarro, Ziviani, Baeza-Yates, "Fast and Flexible Word Searching on Compressed Text", ACM Trans Info Syst 2000.

in which the authors present a variant of the Huffman compression scheme that allows to perform pattern-matching directly over the compressed file. The resulting tool has been called **CGrep**, just to underline the fact that it is a scan-based pattern-matching routine, similar to *Grep*, but now acting over compressed files.

We have actually developed two software libraries which implement the ideas above with some added features:

- **Huffword Library**: This library offers some algorithms and data structures for the efficient (de)compression of textual files. The compression paradigm is a variant of the Huffman algorithm, here adapted to be *byte-aligned*, *tagged* and *word oriented*. More precisely, the tokens are alphanumeric words, and the codewords are byte sequences with some tagged bits that allow to guarantee synchronization. The tagging is obtained as follows: the Huffman tree has fan-out 128, and the arc labels of 7 bits are stored in the least significant bits of a byte; the codeword is finally constructed by setting the first bit of the first byte to 1, and the first bit of each other byte to 0. This compression paradigm can be improved by removing each single space that occurs between two alphanumeric tokens, hereafter called *Spaceless Model*.
- **CGrep Library**: This library offers all the pattern-matching functionalities that allow to search over Huffword compressed files. Actually the search consists of three steps: first the complex pattern query (possibly involving errors, reg exp, ...) is resolved against the dictionary of tokens, built at compression time by the Huffword algorithm. The tokens that satisfy the query are coded and then their codewords are searched directly onto the compressed file (note that this is an exact multiple-patterns search). This approach induces a twofold speedup: the search cost is actually proportional to the length of the compressed file, no decompression at all is required; and the search cost for complicated patterns (like reg exp, errors,....) depends on the dictionary size, which is small compared to the entire file. The features that we have implemented in addition to the original paper are: proximity query, arbitrary number of query-patters each with its search options, snippet extraction.

Trova: dmit Trova successivo Trova precedente Evidenzia Maiuscole/minuscole

Completato

You find it under my Software projects

Data Compression

Basic search algorithms:

Single and Multiple-patterns

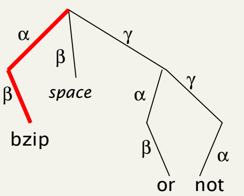
Mismatches and Edits

Problem 1

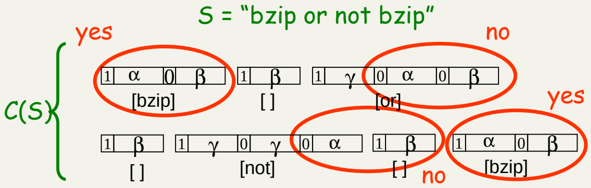
P = bzip = | 1α | 0β |

Dictionary

bzip
not
or
space



S = "bzip or not bzip"



Speed ≈ Compression ratio

Pattern Matching Problem

Exact match problem: we want to find all the occurrences of the pattern P[1,m] in the text T[1,n].

T	A	B	C	A	B	D	A	B
P	A	B						

✓ Naïve solution
- For any position i of T, check if T[i,i+m-1]=P[1,m]

☹ Complexity: O(nm) time

✓ (Classical) Optimal solutions based on comparisons
- Knuth-Morris-Pratt
- Boyer-Moore

☺ Complexity: O(n + m) time

Semi-numerical pattern matching

- We show methods in which Arithmetic and Bit-operations replace comparisons
- We will survey two examples of such methods
  - The Random Fingerprint method due to Karp and Rabin
  - The Shift-And method due to Baeza-Yates and Gonnet

Rabin-Karp Fingerprint

- We will use a class of functions from strings to integers in order to obtain:
  - An efficient randomized algorithm that makes an error with small probability.
  - A randomized algorithm that never errors whose running time is efficient with high probability.
- We will consider a binary alphabet. (i.e., $T=\{0,1\}^n$)

Arithmetic replaces Comparisons

- Strings are also numbers, H: strings → numbers.
- Let s be a string of length m

$$H(s) = \sum_{i=1}^{m} 2^{m-i} s[i]$$

  - P = 0 1 0 1 H(P) = $2^3 0+2^2 1+2^1 0+2^0 1= 5$
  - s=s' **if and only if** H(s)=H(s')
- Definition:
  let T_r denote the m length substring of T starting at position r (i.e., T_r = T[r,r+m-1]).

Arithmetic replaces Comparisons

- Strings are also numbers, H: strings → numbers
  - Exact match = Scan T and compare $H(T_i)$ and $H(P)$
  - There is an occurrence of P starting at position r of T **if and only if** $H(P) = H(T_r)$

T = 1 0 1 1 0 1 0 1
P = 0 1 0 1 H(P) = 5

T = 1 0 1 1 0 1 0 1 $H(T_2) = 6 \neq H(P)$
P = 0 1 0 1

T = 1 0 1 1 0 1 0 1 $H(T_5) = 5 = H(P)$ Match!
P = 0 1 0 1

Arithmetic replaces Comparisons

- We can compute $H(T_r)$ from $H(T_{r-1})$

$$H(T_r) = 2H(T_{r-1}) - 2^m T(r-1) + T(r+n-1)$$

T = 1 0 1 1 0 1 0 1
T_1 = 1 0 1 1
T_2 = 0 1 1 0

$$H(T_1) = H(1011) = 11$$

$$H(T_2) = H(0110) = 2 \cdot 11 - 2^4 \cdot 1 + 0 = 22 - 16 = 6 = H(0110)$$

Arithmetic replaces Comparisons

- A simple efficient algorithm:
- Compute H(P) and $H(T_1)$
- Run over T
  - Compute $H(T_r)$ from $H(T_{r-1})$ in constant time, and make the comparisons (i.e., $H(P)=H(T_r)$).
- Total running time O(n+m)?
  - **NO! why?**
  - The problem is that when m is large, it is unreasonable to assume that each arithmetic operation can be done in O(1) time.
    - Values of H() are **m**-bits long numbers. In general, they are too **BIG** to fit in a machine's word.
- IDEA! Let's use modular arithmetic:

  For some prime q, the *Karp-Rabin fingerprint* of a string s is defined by $H_q(s) = H(s) \pmod q$

An example

P = 1 0 1 1 1 1
q = 7

H(P) = 47
$H_q(P) = 47 \pmod 7 = 5$

$H_q(P)$ can be computed incrementally!

$$1 \cdot 2 \pmod 7 + 0 = 2$$
$$2 \cdot 2 \pmod 7 + 1 = 5$$
$$5 \cdot 2 \pmod 7 + 1 = 4$$
$$4 \cdot 2 \pmod 7 + 1 = 2$$
$$2 \cdot 2 \pmod 7 + 1 = 5$$
$$5 \pmod 7 = 5 = H_q(P)$$

We can still compute $H_q(T_r)$ from $H_q(T_{r-1})$.

$2^m \pmod q = 2(2^{m-1} \pmod q) \pmod q$

Intermediate values are also small! (< 2q)

Karp-Rabin Fingerprint

- How about the comparisons?

  Arithmetic:
  There is an occurrence of P starting at position r of T **if and only if** $H(P) = H(T_r)$

  Modular arithmetic:
  If there is an occurrence of P starting at position r of T **then** $H_q(P) = H_q(T_r)$

  False match! There are values of q for which the converse is not true (i.e., $P \neq T_r$ AND $H_q(P) = H_q(T_r)$)!

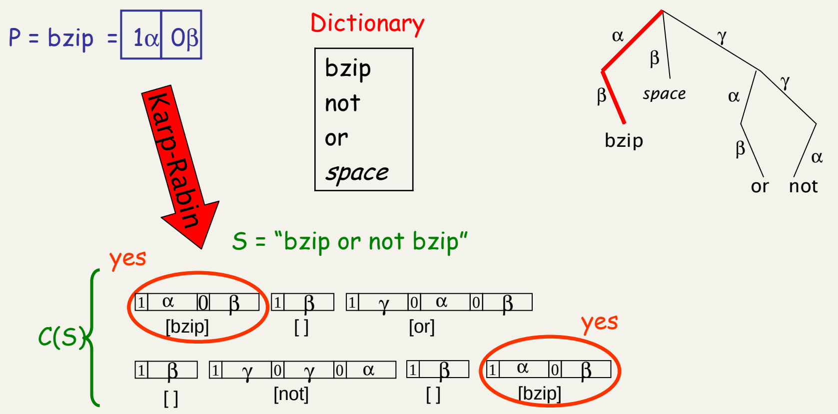
- Our goal will be to choose a modulus q such that
  - q is small enough to keep computations efficient. (i.e., $H_q()$s fit in a machine word)
  - q is large enough so that the probability of a false match is kept small

Karp-Rabin fingerprint algorithm

- Choose a positive integer I
- Pick a random prime q less than or equal to I, and compute P's fingerprint – $H_q(P)$.
- For each position r in T, compute $H_q(T_r)$ and test to see if it equals $H_q(P)$. If the numbers are equal either
  - declare a probable match (randomized algorithm).
  - or check and declare a definite match (deterministic algorithm)
- Running time: excluding verification O(n+m).
- Randomized algorithm is correct w.h.p
- Deterministic algorithm whose expected running time is O(n+m)

Proof on the board

Problem 1: Solution



Speed ≈ Compression ratio

The Shift-And method

- Define M to be a binary m by n matrix such that:

  M(*i*,*j*) = *1* iff the first *i* characters of P exactly match the *i* characters of T ending at character *j*.
  - i.e., M(*i*,*j*) = *1* iff P[*1* … *i*] = T[*j*-*i*+*1*…*j*]
- Example: T = california and P = for

M		c	a	l	i	f	o	r	n	i	a
T	n m	1	2	3	4	5	6	7	8	9	10
P	f 1	0	0	0	0	1	0	0	0	0	0
	o 2	0	0	0	0	0	1	0	0	0	0
	r 3	0	0	0	0	0	0	1	0	0	0

- How does **M** solve the exact match problem?

How to construct M

- We want to exploit the bit-parallelism to compute the j-th column of M from the j-1-th one
  - Machines can perform bit and arithmetic operations between two words in constant time.
  - Examples:
    - And(A,B) is bit-wise *and* between A and B.
    - *BitShift(A)* is the value derived by *shifting* the A's bits down by one and setting the first bit to *1*.

$$BitShift\left(\begin{pmatrix}0\\1\\1\\0\\1\end{pmatrix}\right) = \begin{pmatrix}1\\0\\1\\1\\0\end{pmatrix}$$

  - Let w be the word size. (e.g., 32 or 64 bits). We'll assume m=w. NOTICE: any column of M fits in a memory word.

How to construct M

- We want to exploit the bit-parallelism to compute the j-th column of M from the j-th one
- We define the m-length binary vector U(*x*) for each character *x* in the alphabet. U(*x*) is set to 1for the positions in P where character *x* appears.
- Example:

  P = abaac

$$U(a) = \begin{pmatrix}1\\0\\1\\1\\0\end{pmatrix} \quad U(b) = \begin{pmatrix}0\\1\\0\\0\\0\end{pmatrix} \quad U(c) = \begin{pmatrix}0\\0\\0\\0\\1\end{pmatrix}$$

How to construct M

- Initialize column 0 of M to all zeros
- For $j > 0$, j-th column is obtained by

$$M(j) = BitShift(M(j-1)) \,\&\, U(T[j])$$

- For $i > 1$, Entry M(i,j) = 1 iff
  (1) The first i-1 characters of P match the i-1characters of T ending at character j-1 ⇔ M(i-1,j-1) = 1
  (2) P[i] = T[j] ⇔ the i-th bit of U(T[j]) = 1
- BitShift moves bit M(i-1,j-1) in i-th position
- AND this with i-th bit of U(T[j]) to estabilish if both are true

An example j=1

     1 2 3 4 5 6 7 8 9 10
T = x a b x a b a a c a
     1 2 3 4 5
P = a b a a c

n m	1	2	3	4	5	6	7	8	9	10
1	0									
2	0									
3	0									
4	0									
5	0									

$$U(x) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$BitShift(M(0)) \,\&\, U(T[1]) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example j=2

1 2 3 4 5 6 7 8 9 10
T = x a b x a b a a c a
1 2 3 4 5
P = a b a a c

n m	1	2	3	4	5	6	7	8	9	10
1	0	1								
2	0	0								
3	0	0								
4	0	0								
5	0	0								

$$\boldsymbol{U}(\boldsymbol{a}) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

$$BitShift(M(1))\,\&\,U(T[2]) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example j=3

1 2 3 4 5 6 7 8 9 10
T = x a b x a b a a c a
1 2 3 4 5
P = a b a a c

n m	1	2	3	4	5	6	7	8	9	10
1	0	1	0							
2	0	0	1							
3	0	0	0							
4	0	0	0							
5	0	0	0							

$$\boldsymbol{U}(\boldsymbol{b}) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$BitShift(M(2))\,\&\,U(T[3]) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

An example j=9

1 2 3 4 5 6 7 8 9 10
T = x a b x a b a a c a
1 2 3 4 5
P = a b a a c

n m	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	1	0	1	1	0	
2	0	0	1	0	0	1	0	0	0	
3	0	0	0	0	0	0	1	0	0	
4	0	0	0	0	0	0	0	1	0	
5	0	0	0	0	0	0	0	0	1	

$$U(c) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

$$BitShift(M(8)) \,\&\, U(T[9]) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \& \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Shift-And method: Complexity

- If m<=w, any column and vector U() fit in a memory word.
  - Any step requires O(1) time.
- If m>w, any column and vector U() can be divided in m/w memory words.
  - Any step requires O(m/w) time.
- Overall O(n(1+m/w)+m) time.

- Thus, it is very fast when pattern length is close to the word size.
  - Very often in practice. Recall that w=64 bits in modern architectures.

Some simple extensions

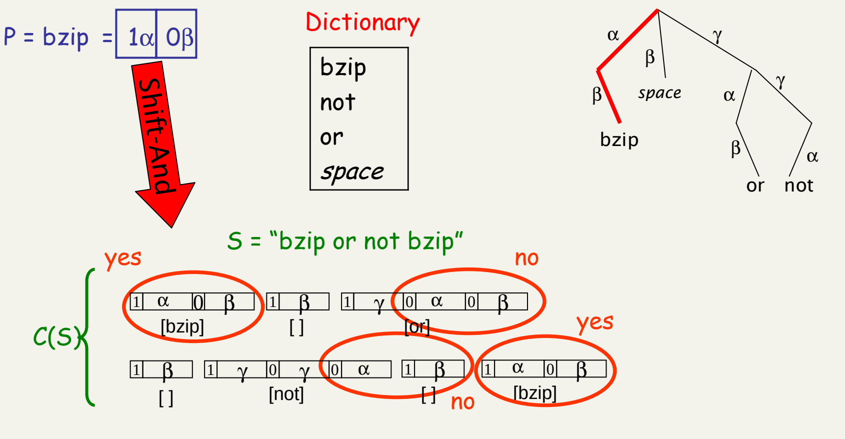
- We want to allow the pattern to contain special symbols, like [a-f] classes of chars

P = [a-b]baac

$$U(a) = \begin{pmatrix}1\\0\\1\\1\\0\end{pmatrix} \quad U(b) = \begin{pmatrix}1\\1\\0\\0\\0\end{pmatrix} \quad U(c) = \begin{pmatrix}0\\0\\0\\0\\1\end{pmatrix}$$

- What about '?', '[^…]' (not).

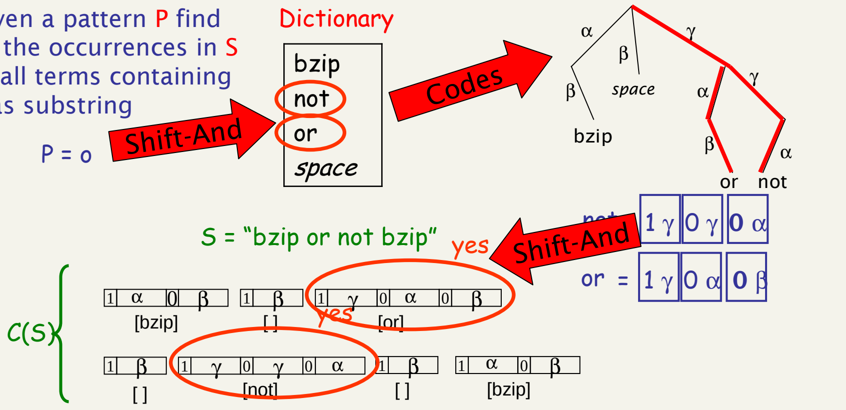
Problem 1: An other solution



Speed ≈ Compression ratio

Problem 2

Given a pattern P find all the occurrences in S of all terms containing P as substring

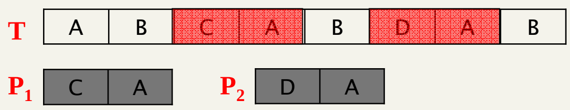


Speed ≈ Compression ratio? No! Why?

A scan of C(s) for each term that contains P

Multi-Pattern Matching Problem

Given a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_l\}$ of total length m, we want to find all the occurrences of those patterns in the text T[1,n].



✓ Naïve solution

- Use an (optimal) Exact Matching Algorithm searching each pattern in $\mathcal{P}$

☹ Complexity: O(nl+m) time, not good with many patterns

✓ Optimal solution due to Aho and Corasick

- Complexity: O(n + l + m) time

A simple extention of Shift-And

- S is the concatenation of the patterns in $\mathscr{P}$
- R is a bitmap of lenght m.
  - R[i] = 1 iff S[i] is the first symbol of a pattern
- Use a variant of Shift-And method searching for S
  - For any symbol c, U'(c) = U(c) *and* R
    - U'(c)[i] = 1iff S[i]=c **and** is the first symbol of a pattern
  - For any step j,
    - compute M(j)
    - **then** M(j) OR U'(T[j]). Why?
      - Set to 1 the first bit of each pattern that start with T[j]
    - Check if there are occurrences ending in j. How?

Problem 3

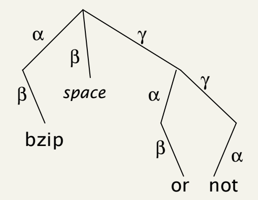
Given a pattern P find all the occurrences in S of all terms containing P as substring allowing at most k mismatches

P = bot k=2

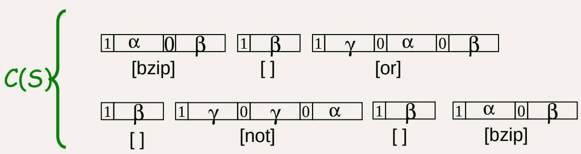
???

Dictionary

bzip
not
or
space



S = "bzip or not bzip"



Agrep: Shift-And method with errors

- We extend the *Shift-And* method for finding inexact occurrences of a pattern in a text.
- Example:
  T = aatatccacaa
  P = atcgaa

  P appears in T with 2 mismatches starting at position 4, it also occurs with 4 mismatches starting at position 2.

  a a t a t c c a c a a   a a t a t c c a c a a
  a t c g a a   a t c g a a

Agrep

- Our current goal given *k* find all the occurrences of P in T with up to *k* mismatches
- We define the matrix M^l to be an m by n binary matrix, such that:

  $M^l(i,j) = 1$ iff
  There are no more than l mismatches between the first *i* characters of P match the *i* characters up through character *j* of T.

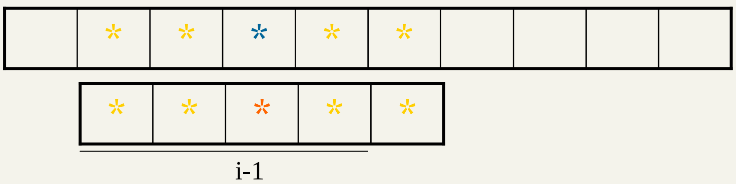
- What is M^0?
- How does M^k solve the k-mismatch problem?

Computing M^k

- We compute M^l for all $l=0, \ldots, k$.
- For each j compute $M(j)$, $M^1(j)$, … , $M^k(j)$
- For all l initialize $M^l(0)$ to the zero vector.
- In order to compute $M^l(j)$, we observe that there is a match iff

Computing M^l: case 1

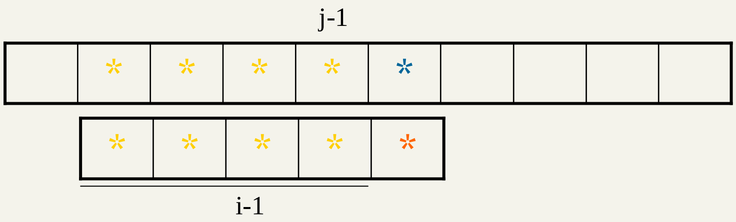
- The first i-1 characters of P match a substring of T ending at j-1, with at most l mismatches, and the next pair of characters in P and T are equal.



$$BitShift(M^l(j-1)) \wedge U(T[j])$$

Computing M^l: case 2

- The first *i-1* characters of P match a substring of T ending at *j-1*, with at most *l-1* mismatches.



$$BitShift(M^{l-1}(j-1))$$

Computing M^l

- We compute M^l for all $l=0, \ldots, k$.
- For each j compute $M(j), M^1(j), \ldots, M^k(j)$
- For all l initialize $M^l(0)$ to the zero vector.
- In order to compute $M^l(j)$, we observe that there is a match iff

$$M^l(j) =$$
$$[BitShift(M^l(j-1)) \wedge U(T(j))] \vee$$
$$BitShift(M^{l-1}(j-1))$$

Example M^1

1 2 3 4 5 6 7 8 9 10
T = x a b x a b a a c a
P = a b a a d

M^1=

	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	0	0	1	0	1	1	0
3	0	0	0	1	0	0	1	0	0	1
4	0	0	0	0	1	0	0	1	0	0
5	0	0	0	0	0	0	0	0	1	0

M^0=

	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	1	0	1	1	0	1
2	0	0	1	0	0	1	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	0	0	0

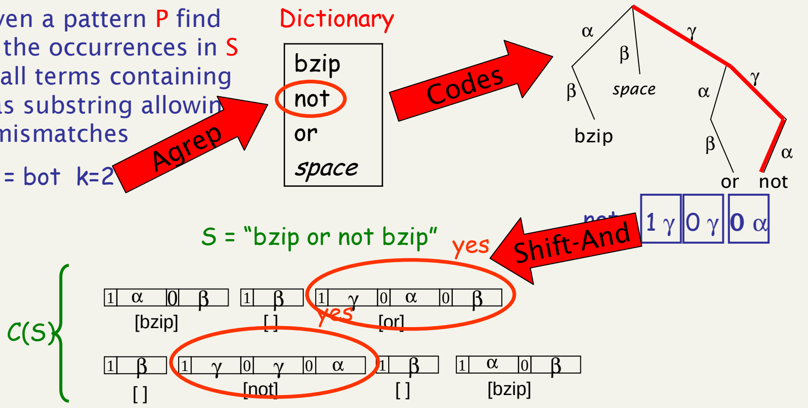
How much do we pay?

- The running time is O(kn(1+m/w)
- Again, the method is practically efficient for small m.
- Still only a O(k) columns of M are needed at any given time. Hence, the space used by the algorithm is O(k) memory words.

Problem 3: Solution

Given a pattern P find all the occurrences in S of all terms containing P as substring allowin[g] k mismatches

P = bot k=2



Agrep: more sophisticated operations

- The Shift-And method can solve other ops
  - The edit distance between two strings p and s is d(p,s) = minimum numbers of operations needed to transform p into s via three ops:
    - Insertion: insert a symbol in p
    - Delection: delete a symbol from p
    - Substitution: change a symbol in p with a different one
  - Example: d(ananas,banane) = 3
  - Search by regular expressions
    - Example: (a|b)?(abc|a)

Data Compression

Some thoughts

Variations…

Canonical Huffman still needs to know the codeword lengths, and thus build the tree…

This may be extremely time/space costly when you deal with Gbs of textual data

A simple algorithm

Sort p_i in decreasing order, and encode s_i via the variable-length code for the integer i.

γ–code for integer encoding

0000...........0	x in binary

Length-1

- x > 0 and Length = $\lfloor \log_2 x \rfloor + 1$

e.g., 9 represented as <000,1001>.

- γ–code for x takes $2 \lfloor \log_2 x \rfloor + 1$ bits
  (ie. factor of 2 from optimal)

- Optimal for Pr(x) = $1/2x^2$, and i.i.d integers

It is a prefix-free encoding…

- Given the following sequence of γ–coded integers, reconstruct the original sequence:

0001000001100110000011101100111

8 6 3 59 7

Analysis

Sort p_i in decreasing order, and encode s_i via the variable-length code $\gamma(i)$.

Recall that: $|\gamma(i)| \leq 2 * \log i + 1$

How much good is this approach wrt Huffman?

Compression ratio $\leq 2 * H_0(s) + 1$

Key fact:
$1 \geq \Sigma_{i=1,\dots,x} p_i \geq x * p_x \rightarrow x \leq 1/p_x$

How good is it ?

Encode the integers via δ-coding:

$|\gamma(i)| \leq 2 * \log i + 1$

The cost of the encoding is (recall $i \leq 1/p_i$):

$$\sum_{i=1,\dots,\Sigma} p_i * |\gamma(i)| \leq \sum_{i=1,\dots,\Sigma} p_i * [2 * \log \frac{1}{p_i} + 1]$$

This is: $\leq 2 * H_0(X) + 1$

No much worse than Huffman,
and improvable to $H_0(X) + 2 + \dots$

A better encoding

- **Byte-aligned and tagged Huffman**
  - 128-ary Huffman tree
  - First bit of the first byte is tagged
  - Configurations on 7-bits: just those of Huffman

- **End-tagged dense code**
  - The *rank* r is mapped to r-th binary sequence on 7*k bits
  - First bit of the last byte is tagged

A better encoding

Surprising changes

- It is a prefix-code
- Better compression: it uses all 7-bits configurations

Rank	ETDC	THC
1	100	100
2	101	101
3	110	110
4	111	111 000
5	000 100	111 001
6	000 101	111 010
7	000 110	111 011 000
8	000 111	111 011 001
9	001 100	111 011 010
10	001 101	111 011 011

Table 1: Comparative example among ETDC and THC, for b=3.

(s,c)-dense codes

Distribution of words is skewed: $1/i^{\theta}$, where $1 < \theta < 2$

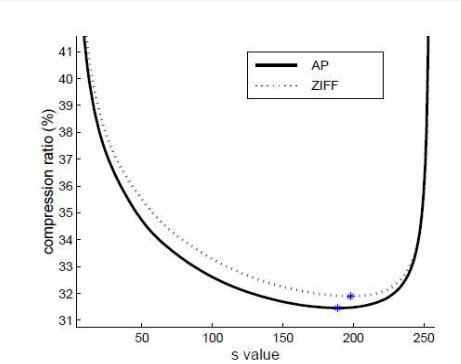
- **A new concept: Continuers vs Stoppers**
  - Previously we used: s = c = 128
- **The main idea is:**
  - s + c = 256 (we are playing with 8 bits)
  - Thus s items are encoded with 1 byte
  - And s*c with 2 bytes, s * c^2 on 3 bytes, ...
- **An example**
  - 5000 distinct words
  - ETDC encodes 128 + 128^2 = 16512 words on 2 bytes
  - (230,26)-dense code encodes 230 + 230*26 = 6210 on 2 bytes, hence more on 1 byte and thus if skewed...

It is a prefix-code

Optimal (s,c)-dense codes

Find the optimal s, by assuming c = 128-s.

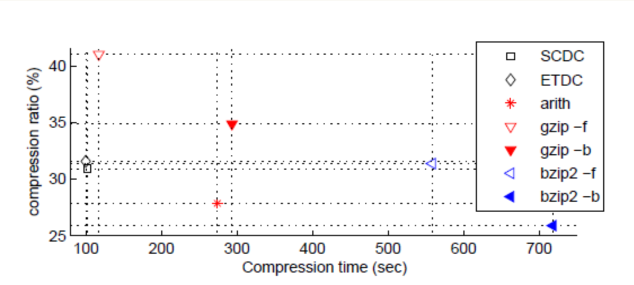
- **Brute-force approach**
- **Binary search:**
  - On real distributions, it seems that one unique minimum



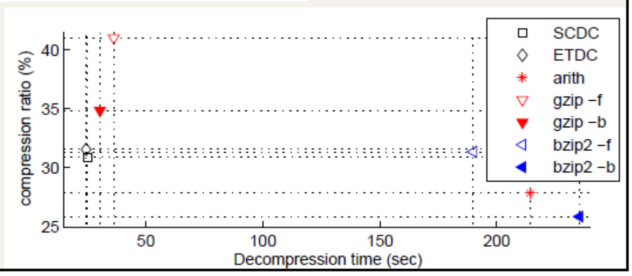
$$L(s,c) = \sum_{k=0}^{K^s-1} (1 - F_k^s)$$

K^s = max codeword length
F^s_k = cum. prob. symb. whose |cw| <= k

Experiments: (s,c)-DC much interesting…



Search is 6% faster than byte-aligned Huffword



Streaming compression

Still you need to determine and sort all terms….

Can we do everything in one pass ?

- **Move-to-Front (MTF)**:
  - As a freq-sorting approximator
  - As a caching strategy
  - As a compressor
- **Run-Length-Encoding (RLE)**:
  - FAX compression

Move to Front Coding

Transforms a **char** sequence into an **integer** sequence, that can then be *var-length coded*

- Start with the list of symbols L=[a,b,c,d,…]
- For each input symbol *s*
  1) output the position of *s* in L
  2) move *s* to the front of L

There is a memory

Properties:

- Exploit *temporal locality*, and it is *dynamic*
- X = $1^n 2^n 3^n \dots n^n$ → Huff = $O(n^2 \log n)$, MTF = $O(n \log n) + n^2$

No much worse than Huffman
...but it may be far better

MTF: how good is it ?

Encode the integers via δ-coding:

$$|\gamma(i)| \leq 2 * \log i + 1$$

Put Σ in the front and consider the cost of encoding:

$$O(\Sigma \log \Sigma) + \sum_{x=1}^{\Sigma} \sum_{i=2}^{n_x} \gamma(p_i^x - p_{i-1}^x)$$

By Jensen's:

$$\leq O(\Sigma \log \Sigma) + \sum_{x=1}^{\Sigma} n_x [2 * \log \frac{N}{n_x} + 1]$$

$$\leq O(\Sigma \log \Sigma) + N * [2 * H_0(X) + 1]$$

$$L_a[mtf] \leq 2 * H_0(X) + O(1)$$

MTF: higher compression

Alphabet of words

How to keep efficiently the MTF-list:

- **Search tree**
  - Leaves contain the words, ordered as in the MTF-List
  - Nodes contain the size of their descending *subtree*
- **Hash Table**
  - keys are the words (of the MTF-List)
  - *data* is a pointer to the corresponding tree leaves

- Each ops takes $O(\log \Sigma)$
- Total cost is $O(n \log \Sigma)$

Run Length Encoding (RLE)

If spatial locality is very high, then

abbbaacccca => (a,1),(b,3),(a,2),(c,4),(a,1)

In case of binary strings → just numbers and one bit

Properties:

- Exploit *spatial locality*, and it is a *dynamic code*

There is a memory

- $X = 1^n 2^n 3^n \dots n^n$ →

  $Huff(X) = n^2 \log n \;>\; Rle(X) = n\,(1+\log n)$