6. Dictionary models for text compression

Previous techniques:

- Predictive, statistical
- One symbol at a time

Dictionary coding:

- Substrings replaced by pointers to a dictionary
- Pointers are coded (often fixed-length codes)
- Dictionary can be *static, semi-adaptive* or *adaptive*
- Dictionary can be *implicit* or *explicit*

Can be proved:

- Each dictionary scheme has an equivalent statistical scheme achieving at least the same compression.

Viewpoints on dictionary models

Advantages:

- Simple
- Fast
- Practical

Design decisions:

- Selection of substrings to be included in the dictionary
- Restricting the length of substrings
- Restricting the *window* where the dictionary is taken from in adaptive methods
- Encoding of references to the dictionary

Parsing strategies in dictionary modelling

Division of the message into substrings:

- *Greedy*: Choose the longest matching substring at each step from left to right.
- *Longest-fragment-first* (*LFF*): Choose the substring matching somewhere in the unparsed parts of the message.
- *Optimal*: Create a graph of all matching phrases and determine its shortest path.

Dictionary modelling approaches

(1) Static dictionary:

- Fixed for all sources
- Known to the encoder and decoder
- Choice of substrings (words, phrases) is a problem.
- Depends too much on the message type
- E.g. a complete English dictionary would be too large and not at all source-specific.

Dictionary modelling approaches (cont.)

(2) Semi-adaptive dictionaries:

- Create a dictionary *D* for the current source message
- Finding an *optimal* dictionary is NP-complete
- Size |*D*| is usually fixed
- Typical heuristic: Find approximately equi-frequent substrings and use fixed-length codes ($\lceil \log_2 |D| \rceil$ bits)
- Using e.g. Huffman coding does not usually pay.

Dictionary modelling approaches (cont.)

(3) Adaptive dictionaries:

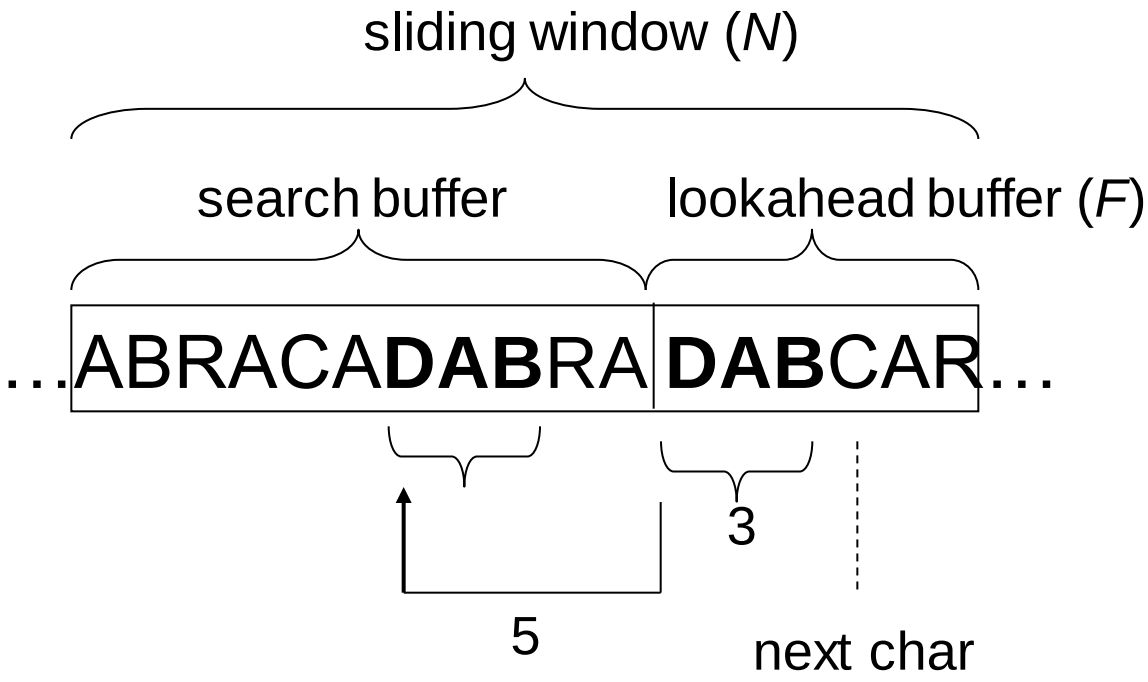
Two large ‘families’ of methods:

- ***LZ77*: *Implicit*** dictionary; any substring from the processed part of the message

- ***LZ78*: *Explicit***, evolving dictionary; only selected substrings of the processed part.

[’L’ → Abraham Lempel, ’Z’ → Jacob Ziv]

Illustrating the idea of LZ77 coding



Code triple : <5, 3, C>

Code structure in LZ77

- Substring code consists of triples <*offset, length, char*>
- Offset = distance of the *longest match* from the end of the search buffer
- Length = length of the matching substring
- Char = symbol following the match in the lookahead buffer
- Triple size = $\lceil \log_2(N-F) \rceil + \lceil \log_2 F \rceil + \lceil \log_2 q \rceil$ bits, when using ***fixed-length codes*** for the components.

Features of LZ77

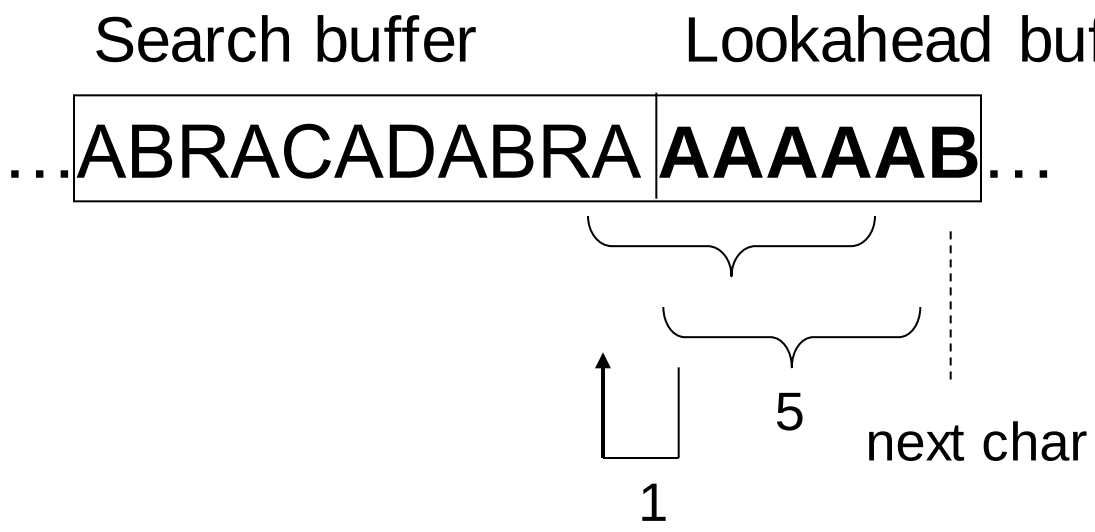
Special case:

- Longest match extends to the search buffer
- Decoder can recover the substring simply by copying symbols from left to right

Optimality of LZ77:

- Approaches the best possible semi-adaptive method that has full knowledge of the statistics of the source.

Example: matching pattern extends to the lookahead buffer



Code triple : <1, 5, B>

Some members of the LZ77 family

LZR (Rodeh, Pratt, Even, 1981):

- No window; the complete processed part is used
- Variable-length coding of arbitrarily large offsets

LZSS (Storer, Szymanski, 1982):

- No character extension of matches
- Flag bit tells, whether the codeword represents a single symbol, or an offset & length pair.

Some members of the LZ77 family (cont.)

LZB (Bell, 1987):

- Match length is γ-coded
- Shorter offsets for the front part of the message
- Some other tunings

LZH (Brent, 1987):

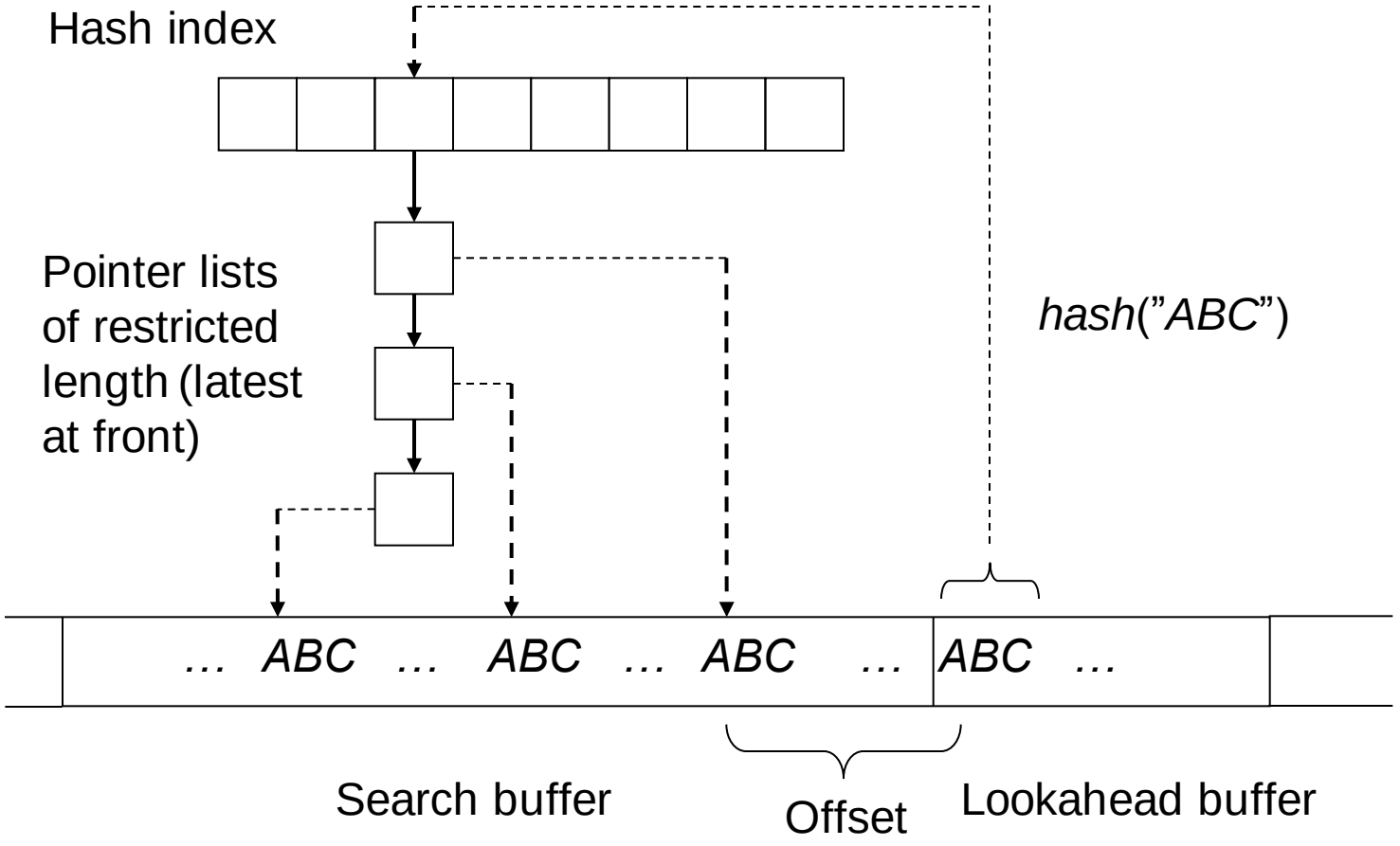
- Huffman coding of the components of references

Some members of the LZ77 family (cont.)

GZip (Gailly, 90's):

- Part of *Gnu software* (for Unix)
- Fast searching of matches by three-character hashing
- Raw symbols are encoded in case of no match
- Two ***Canonical*** Huffman codes:
  1) Lengths of matches *and* raw symbols
  2) Offsets (when matching succeeded)
- Semi-adaptive blockwise coding (64 K at a time)
- Reads the input only once
- Either greedy or look-ahead parsing
- Outperforms most other LZ-variants

GZip: Data structure



Drawbacks of LZ77

- Small window results in short matches.
- Large window results in long offsets.
- Distinct code values are reserved for all instances of a repeating pattern.
- Searching for the longest match may be slow.

6.2. LZ78 family of adaptive dictionary methods

Features of LZ78:

- Explicit dictionary, grows dynamically.
- Both encoder and decoder build the dictionary in an identical manner.
- The code consists of <*index*, *symbol*> pairs.
- Matching substring appended by the successor symbol is the next dictionary entry.
- In principle, the dictionary grows without bounds
- In practice, the size is restricted; overflow cases can be handled by flushing, pruning or freezing the dictionary

LZ78 example

Source: “wabba-wabba-wabba-wabba-woo-woo-woo”

Lookahead buffer	Encoder output	Dictionary index	Dictionary entry
wabba-wabba-...	<0, w>	1	w
abba-wabba-w...	<0, a>	2	a
bba-wabba-wa...	<0, b>	3	b
ba-wabba-wab...	<3, a>	4	ba
-wabba-wabba...	<0, ->	5	-
wabba-wabba-...	<1, a>	6	wa
bba-wabba-wa...	<3, b>	7	bb
a-wabba-wabb...	<2, ->	8	a-
wabba-wabba-...	<6, b>	9	wab
ba-wabba-woo...	<4, ->	10	ba-
wabba-woo-wo...	<9, b>	11	wabb
a-woo-woo-wo…	<8, w>	12	a-w
oo-woo-woo	<0, o>	13	o
o-woo-woo	<13, ->	14	o-
woo-woo	<1, o>	15	wo
o-woo	<14, w>	16	o-w
oo	<13, o>	17	oo

Optimality of LZ78

- The compression performance is *asymptotically optimal*, if the message is generated by a *stationary*, *ergodic* source.
- Convergence to the optimum is quite slow
- LZ77 family has generally slightly better compression performance in practice.

Some members of the LZ78 family

LZW (Welch, 1984):

- One of the most famous LZ variants
- The code consists of only references to the dictionary; the appended symbols are omitted.
- The dictionary must be initialized with all symbols of the alphabet.
- The decoder can decide the new entry to be added to the dictionary only after seeing the next match (overlap of one symbol).
- Small problem: reference to the yet unsolved entry; Solution: unsolved symbol equals the first symbol of the match.
- Typical dictionary size: 4096 entries; 12-bit references.

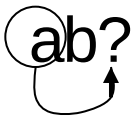
LZW example

Source: ”aababababaaa...”

Index	Substring	Derived from
0	a	
1	b	
2	aa	0+a
3	ab	0+b
4	ba	1+a
5	aba	3+a
6	abaa	5+a
…	…	…

LZW example: decoder steps

Index	Development of dictionary for coded indexes					
		0	0	1	3	5
0	a	a	a	a	a	a
1	b	b	b	b	b	b
2	aa	a?	aa	aa	aa	aa
3	ab		a?	ab	ab	ab
4	ba			b?	ba	ba
5	aba				ab?	aba
6	abaa					aba?
…	…		…			



Some members of the LZ78 family (cont.)

Unix compress (= LZC):

- Close variant of LZW.
- Reference lengths grow gradually to the maximum.
- Compression performance is monitored; if it gets too bad, the dictionary is discarded and rebuilt.

GIF (Graphics Interchange Format):

- Similar to Unix compress
- Some tuning for image data
- Blockwise processing (max 255 bytes)
- Not comparable with the best (but *lossy*) image compressors

Some members of the LZ78 family (cont.)

V.42 bis:

- *V.42* = CCITT recommendation procedure for data transmission in telephone networks.
- *V.42 bis* = related data compression.
- Modification of LZW.
- After reaching the maximum dictionary size, the method reuses unextended entries.
- Upper bound for lengths of encoded substrings.
- Latest dictionary entry cannot be used immediately.

LZT (Tischer, 1987):

- Replacement of least recently used dictionary entries by new ones (= LRU strategy).

Some members of the LZ78 family (cont.)

LZJ (Jakobsson, 1985):

- All unique substrings $\leq h$ included in the dictionary.
- Prunes entries, starting from those that occurred only once
- Encoding is faster than decoding.

LZFG (Fiala, Greene, 1989):

- One of the most effective LZ variants.
- A kind of combination of LZ77 and LZ78.
- Sliding window, arbitrarily long substrings
- Stored strings have matched strings as prefixes
- Data structure: *Patricia trie*
- Code: reference to a node + possible end position of the match (if not unique).

LZFG: Phasing-in technique

- Defines variable-length code codes for integers from range [0, m–1], where m need not be a power of 2.
- Almost fixed-length code: lengths differ at most by one
- Numbers [0, $2^{\lceil \log_2 m \rceil} - m - 1$] encoded with $\lfloor \log_2 m \rfloor$ bits,
- Numbers [$2^{\lceil \log_2 m \rceil} - m$, $m - 1$] encoded with $\lfloor \log_2 m \rfloor + 1$ bits

- Used in many other compression methods, as well.

LZFG: Example of phasing-in technique

$m = 10$, and thus $2\lceil \log m \rceil - m - 1 = 5$

0 = 000	5 = 101
1 = 001	6 = 1100
2 = 010	7 = 1101
3 = 011	8 = 1110
4 = 100	9 = 1111

LZFG: *Start-Step-Stop* codes

- Several (n) different lengths of simple binary codes.
- The first 2^{start} numbers encoded with *start* bits.
- The next $2^{start+step}$ numbers are encoded with *start*+*step* bits.
- The next $2^{start+2 \cdot step}$ numbers encoded with *start*+2·*step* bits, etc.
- The biggest code length used is *stop*.
- The number of different codes available is

$$\frac{2^{stop+step} - 2^{start}}{2^{step} - 1}$$

LZFG: *Start-Step-Stop* codes (cont.)

Example: Start-step-stop (1, 2, 5)

0 = 1 0	2 = 01 000	10 = 00 00000
1 = 1 1	3 = 01 001	11 = 00 00001
	4 = 01 010	12 = 00 00010
	.....	.....
	9 = 01 111	41 = 00 11111

- The last group of codes can be phased-in, if the total number of codes is $\geq m$.
- Normal k-length binary code = start-step-stop (k, 1, k).
- γ-code = start-step-stop (0, 1, ∞).

Performance comparison for Calgary Corpus

[Widely used test data; the results are borrowed from the literature]

File	Size	LZ77	LZSS	LZH	GZIP	LZ78	LZW	LZJ'	LZFG	PPMZ
bib	111261	3.75	3.35	3.24	**2.51**	3.95	3.84	3.63	2.90	**1.74**
book1	768771	4.57	4.08	3.73	**3.26**	3.92	4.03	3.67	3.62	**2.21**
book2	610856	3.93	3.41	3.34	**2.70**	3.81	4.52	3.94	3.05	**1.87**
geo	102400	6.34	6.43	6.52	**5.34**	5.59	6.15	6.05	5.70	**4.03**
news	377109	4.37	3.79	3.84	**3.06**	4.33	4.92	4.59	3.44	**2.24**
obj1	21504	5.41	4.57	4.58	**3.83**	5.58	6.30	5.19	4.03	**3.67**
obj2	246814	3.81	3.30	3.19	**2.63**	4.68	9.81	5.95	2.96	**2.23**
paper1	53161	3.94	3.38	3.38	**2.79**	4.50	4.58	3.66	3.03	**2.22**
paper2	82199	4.10	3.58	3.57	**2.89**	4.24	4.02	3.48	3.16	**2.21**
pic	513216	2.22	1.67	1.04	**0.82**	1.13	1.09	2.40	0.87	**0.79**
progc	39611	3.84	3.24	3.25	**2.67**	4.60	4.88	3.72	2.89	**2.26**
progl	71646	2.90	2.37	2.20	**1.81**	3.77	3.89	3.09	1.97	**1.47**
progp	49379	2.93	2.36	2.17	**1.81**	3.84	3.73	3.14	1.90	**1.48**
trans	93695	2.98	2.44	2.12	**1.61**	3.92	4.24	3.52	1.76	**1.24**
aver.	**224402**	**3.94**	**3.43**	**3.30**	**2.70**	**4.13**	**4.71**	**4.00**	**2.95**	**2.12**