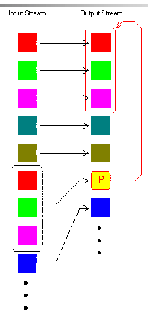
LZ77



© Klein S. T. and Wiseman Y.

Lempel & Ziv's methods

- Let $x_1,\ldots,x_n$ be a sequence of items.
- We want to find a sub-sequence $x_k,\ldots,x_m$ which holds: $P(x_k,\ldots,x_m) > \prod_{i=k}^{m} P(x_i)$
- For example $p(qu) > p(q)\cdot p(u)$.
- Huffman and Arithmetic don't look on the item's environment.

significantly bigger

© Klein S. T. and Wiseman Y.

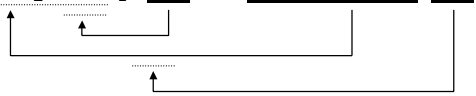
Implementation of LZ77

- Pointers should be distinguished from other items.
- char, (offset, length), char, (offset, length),...
  pointer pointer
- A null pointer is (0,0).
- If two pointers are adjacent, the first character of the second string will be encoded separately.
- Offset is accommodated in n bits so the maximum offset is 2^n

© Klein S. T. and Wiseman Y.

LZ77

```
An outcry in Spain is an outcry in vain
```

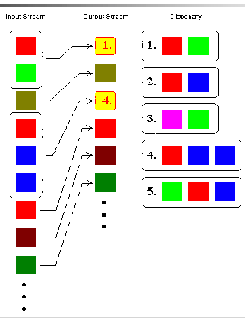


```
An outcry in Spa(6,3)is a(22,12)v(21,3)

aaaaaaaaaa
a(1,9)
```


© Klein S. T. and Wiseman Y.

LZ78



© Klein S. T. and Wiseman Y.

Some versions of LZ77

- LZSS - A flag bit for distinguishing pointers from the other items.
- LZR - No limit on the pointer size.
  - Example: $1234_{10} = 10011010010_2$
    10010110001101010010
    3 4 11
- LZH - Compress the pointers in Huffman coding.

© Klein S. T. and Wiseman Y.

LZW algorithm

- dictionary ← single characters
- w ← first character of input

step:
- k ← next character
- if EOF then output(code(w)); exit
- else
  - if wk ∈ dictionary then w ← wk
  - else
    - Output (code(w))
    - dictionary ← wk
    - w ← k
  - goto step

LZW

- A version of LZ78.
- If x∈dictionary then all prefix(x)∈dictionary
- Finds the longest string in dictionary. If $x_1,\ldots,x_n$ in dictionary but $x_1,\ldots,x_{n+1}$ not in dictionary, add $x_1,\ldots,x_{n+1}$ to dictionary.

Dictionary considerations

- Dictionary size is determined according to pointer's length. Pointer's length can be changed during processing.
- When dictionary is full, one of these can be done:
  - Delete the whole dictionary.
  - Delete strings according to LRU.
  - Each string has a counter. A string whose counter is the lowest will be deleted.
  - Stop to update the dictionary.
  - Double the dictionary's size.

An example to LZW

- T=ababcbababaaaaaaa

w	a	b	a	ab	c	b	ba	b	ba	bab	a	a	aa	a	aa	aaa	a
wk	ab	ba	ab	abc	cb	ba	bab	ba	bab	baba	aa	aa	aaa	aa	aaa	aaaa	
out	1	2		4	3		5			8	1		10			11	1
dict'	ab	ba		abc	cb		bab			baba	aa		aaa			aaaa	

- Dictionary:

code	1	2	3	4	5	6	7	8	9	10	11	12
string	a	b	c	ab	ba	abc	cb	bab	baba	aa	aaa	aaaa

Some features of LZW

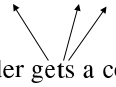
- In order to reduce dictionary size:

code	1	2	3	4	5	6	7	8	9	10	11	12
string	a	b	c	1b	2a	4c	3b	5b	8a	1a	10a	11a

- If there is a long string like "Jerusalem", all its sub-strings will be in the dictionary too.
  - J, Je, Jer, Jeru, Jerus, Jerusa, Jerusal, Jerusale, Jerusalem.
  - LZMW - concatenate last two strings instead of string and a single item.
  - Strings like aa...a will have in LZMW, O(log(n)) entries in the dictionary instead of O(√n) in LZW.

LZW decoding

code	1	2	4	3	5	8	1	10	11	1
decoded text	a	b	ab	c	ba	bab	a	aa	aaa	a
dictionary		ab	ba	abc	cb	bab	baba	aa	aaa	aaaa

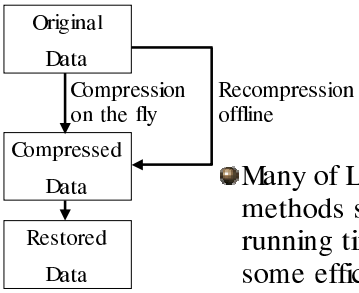


- In these stages the decoder gets a code which isn't in the dictionary yet.
- In such cases, the last code's string concatenated with first item in the string, will be the new string.

Optimal partition

- D = { abc, ab, cdef, d, de, ef, f }
- T = abcdef
- Greedy method will yield abc-de-f.
- A better partition would be ab-cdef.
- If the words in D are encoded by 1, 2, 3, 4, 5, 6 and 6 bits respectively, abc-d-ef is the optimal partition.

Recompression



- Many of Lempel-Ziv methods save some running time by losing some efficiency.

Encoding

- f:D→{0,1}* assigns to each element of the dictionary a binary string.
- f produces a code which is uniquely decipherable.
- A fixed length code can be good, if the occurrences of the elements of D in S is nearly uniform.
- We look for the minimal $\sum_{j\geq 0}\left|f(S_{i_j+1}...S_{i_{j+1}})\right|$

Definitions

- Let $S=S_1S_2...S_n$ be a string. Each S_i belongs to an alphabet Σ.
- Let D be a dictionary contains words from Σ.
- An increasing sequence of indices $i_0=0, i_1, i_2,...$ Should be found s.t.: $S=S_1S_2...S_n=S_1...S_{i_1}S_{i_1+1}...S_{i_2}...$ with $S_{i_j+1}...S_{i_{j+1}}\in D$ for $j=0,1,...$

Improving the graph

- Before adding edge (i,j) to E, check if it is possible to reach vertex j directly from every vertex in {k|(k,i)∈E}, without passing through vertex i.
- If so, we assume the triangle inequality which holds for many practical schemes. Hence edge (i,j) can be pruned.
- If after having checked all the edges emanating from vertex i, none of have been adjoined to E, vertex i and all the incoming edges on vertex i can be pruned.

Reduction to The Shortest Path

- Vertex i corresponds to the character S_i for i≤n and n+1 corresponds to the end of the text.
- An ordered pair (i,j) with i<j belongs to E if and only if $S_i...S_{j-1}\in D$.
- The label L_{ij} is defined for edge (i,j)∈E as $\left|f(S_i...S_{j-1})\right|$
- Dijkstra's worst case complexity for finding the shortest path is the highest between $O(|V|^2)$ and $O(|E|+|V|\log|V|)$.
- In this case the graph contains no cycle, since all of the edges are of the form (i,j) with i<j, so the complexity is O(|E|).

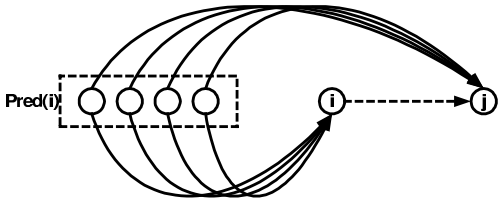
The pruning algorithm

- E←∅; V←{1,...,n,n+1}
- for i ←1 to n
  - Pred(i)←{k|(k,i)∈E}
  - Succ_Candidates(i)←{j|S_i..S_{j-1}∈D}
  - add_edge ←FALSE
  - for all j∈Succ_Candidates(i)
    - all_connected←TRUE
    - for all k∈Pred(i)
      - if (k,j)∉E then all_connected←FALSE
    - if not all_connected then
      - E←E∪{(i,j)}
      - Added_edge ←TRUE
  - if not added_edge then
    - V←V\{i}
    - E←E\{(j,i)|j∈Pred(i)}

 © Klein S. T. and Wiseman Y.

Pruning

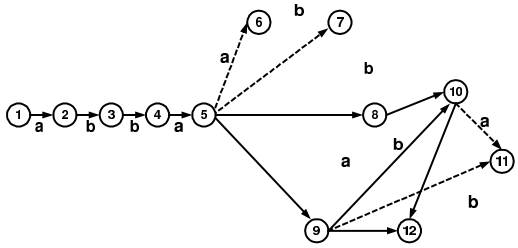
- $L_{kj} \le L_{ki} + L_{ij}$ for $k < i < j$



 © Klein S. T. and Wiseman Y.

Example (cont.)

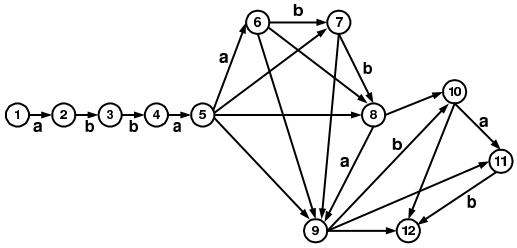
- The graph after pruning



 © Klein S. T. and Wiseman Y.

Example

- A graph for the string abbaabbabab



 © Klein S. T. and Wiseman Y.

Results of recompression

File	Time			Space	
	Optimal	Prune	size	Optimal	Prune
			bytes	No. of edges	Percents remain
English	0.75	0.39	36521	220357	28.9
French	1.55	1.12	53580	222270	47.9
Finnish	1.22	0.47	34615	203772	29.3
German	2.15	1.07	55121	282662	32.9
Hebrew	1.50	1.05	48193	235260	33.7
paper1	1.33	0.76	53161	287574	29.4
prog1	3.05	0.94	71646	1187012	10.7
trans	3.94	0.83	93695	2031230	7.2
bib	2.55	1.52	111261	626136	25.2
moricons	10.02	1.57	118864	5320050	16.8

 © Klein S. T. and Wiseman Y.

Shortest Path Algorithm

- SPL(1)←0
- for i←2 to n+1
  - if i∈V then
    - SPL(i)←∞
    - for all j∈Pred(i)
      - t←SPL(j)+L_{ji}
      - if SPL(i)>t then
        - SPL(i)←t
- Each edge (j,i) is referenced exactly once, so the time complexity is O(|E|).

 © Klein S. T. and Wiseman Y.