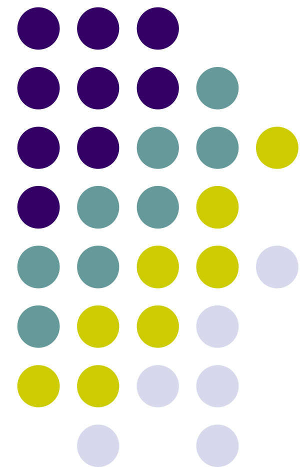
Guaranteed Synchronization of Huffman Codes with Known Position of Decoder

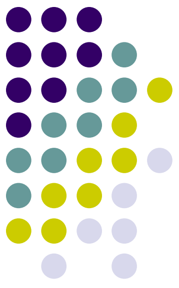


Marek Biskup, Wojciech Plandowski

University of Warsaw, Poland

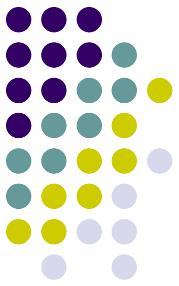
Data Compression Conference '09, 2009-03-16

Agenda



- Synchronization of Huffman Codes
- Guaranteed synchronization
- New method: Guaranteed synchronization of Huffman codes with known start position of decoder
- Performance evaluation + applications

Huffman Codes



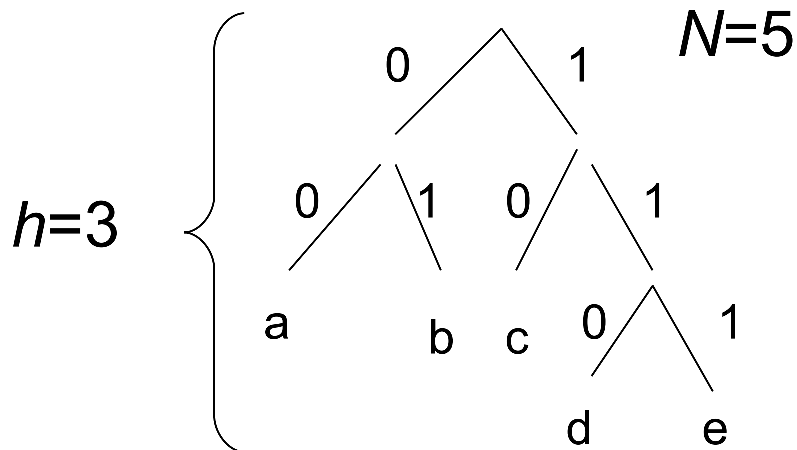
$a \to 00$

$b \to 01$

$c \to 10$

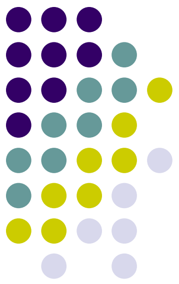
$d \to 110$

$e \to 111$

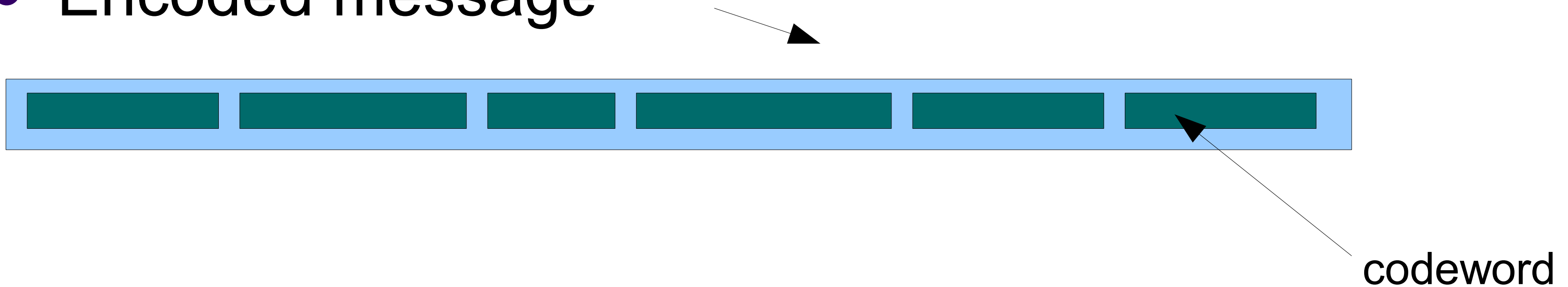


- Each letter has a corresponding binary string (its codeword)
- The codewords form a proper binary tree
- The depth of a letter depends on its probability
- The code is uniquely decodable

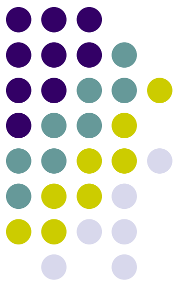
Codeword alignment



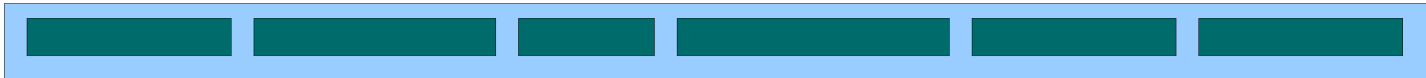
- Encoded message



Codeword alignment

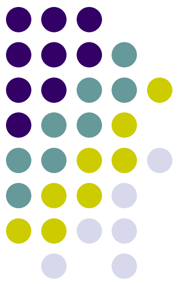


- Encoded message

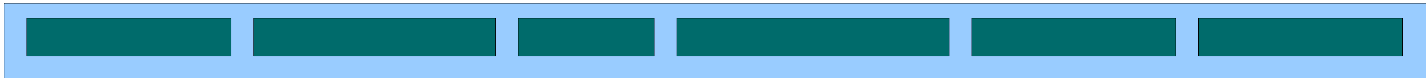


- Decoder's perspective:

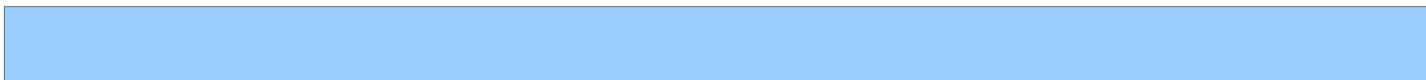
Codeword alignment



- Encoded message

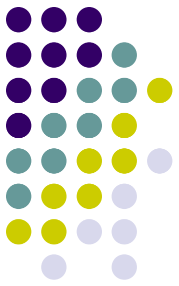


- Decoder's perspective:



Decode from here

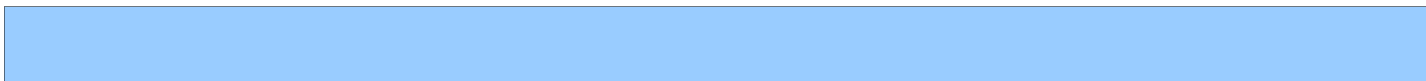
Codeword alignment



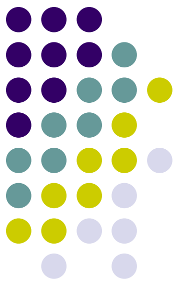
- Encoded message

- Decoded message

- Decoder's perspective:

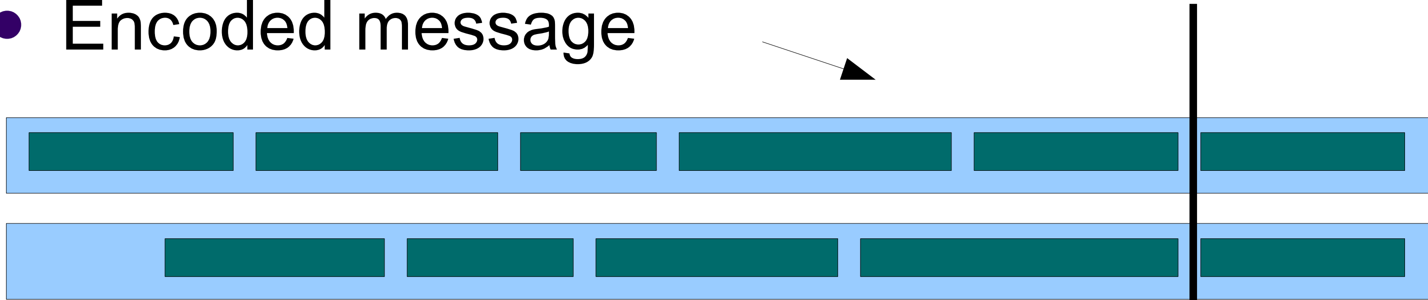


Decode from here

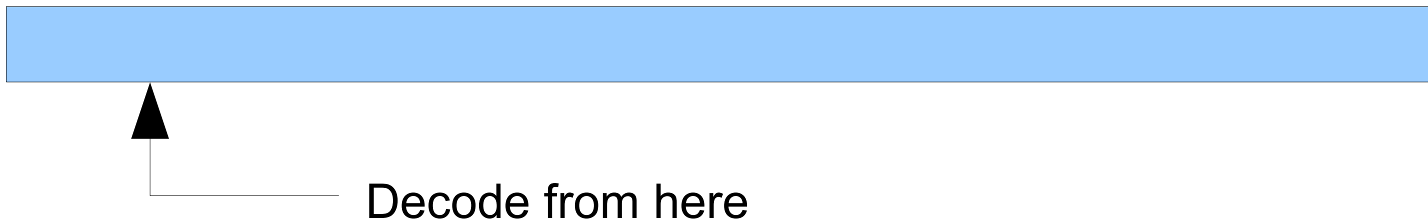


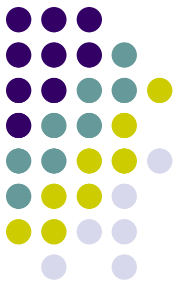
Codeword alignment

- Encoded message
- Decoded message



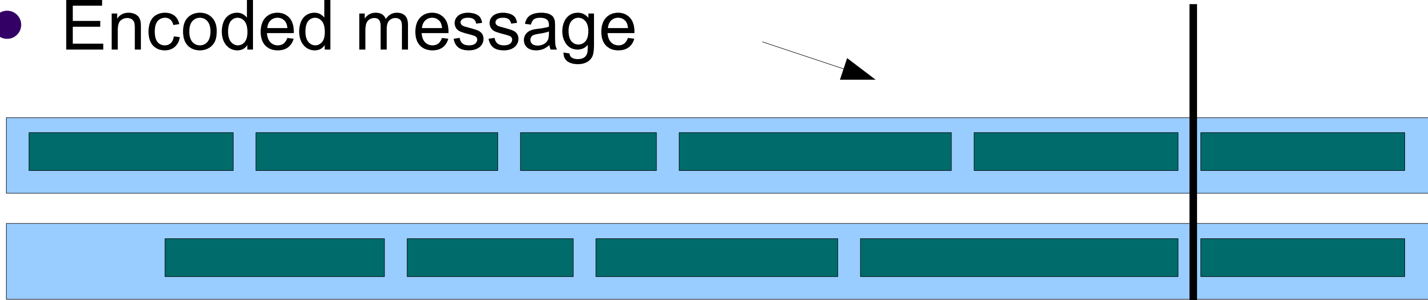
- Decoder's perspective:





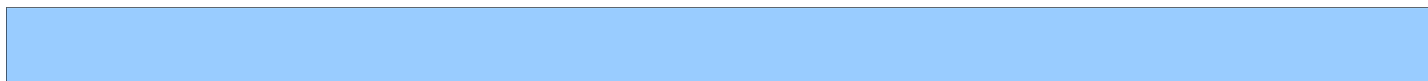
Codeword alignment

- Encoded message



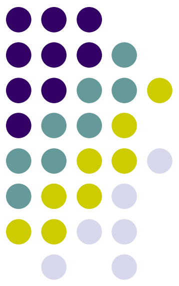
- Decoded message

- Decoder's perspective:



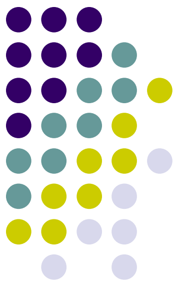
Decode from here

Misalignment

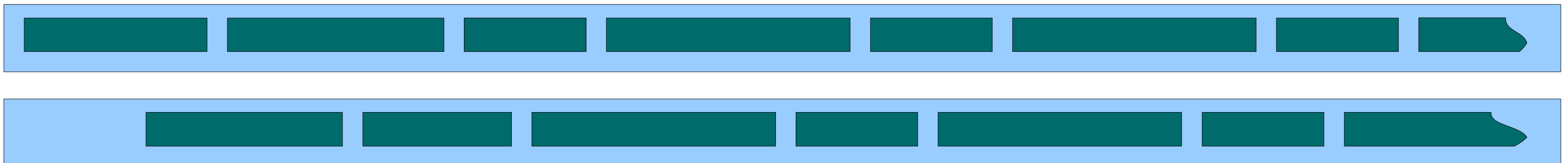


- Reasons:
  - Decoding a fragment
  - Parallel decoding
  - Bit corruption
- Consequences:
  - (part of) message is decoded incorrectly
- Good news
  - Resynchronization occurs quite quickly
  - After resynchronization the rest is decoded correctly

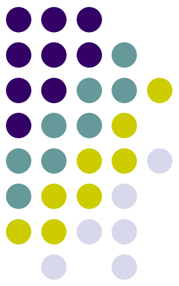
Problems



- Synchronization is only statistical
  - no upper bound on the synchronization delay
- Example:

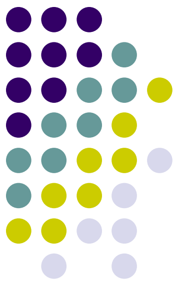


Guaranteed synchronization

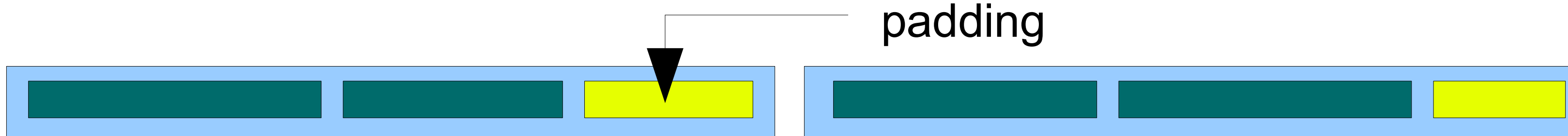


- Goal:
  - limit the synchronization delay to < L bits

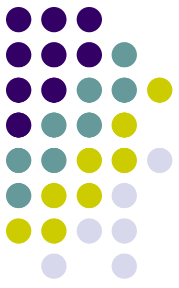
Guaranteed synchronization



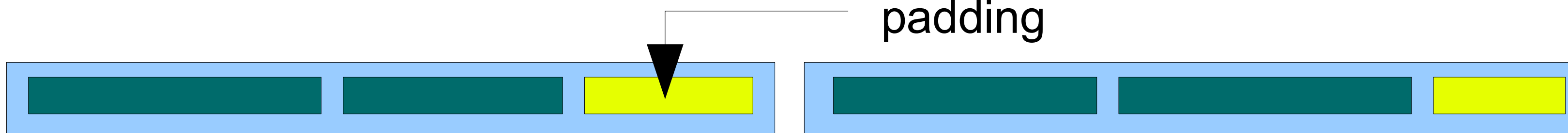
- Goal:
  - limit the synchronization delay to < L bits
- **Simple approaches**:
  - Divide the data into fixed-size blocks



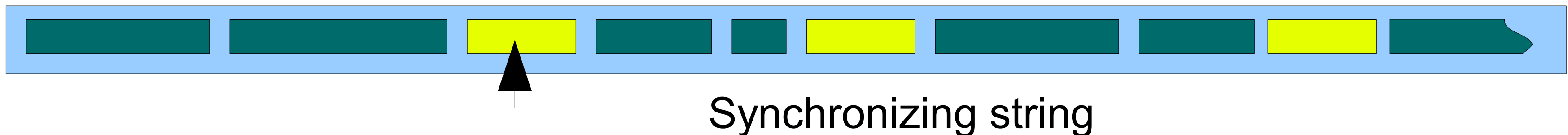
Guaranteed synchronization



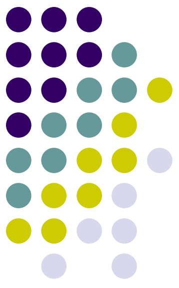
- Goal:
  - limit the synchronization delay to < L bits
- **Simple approaches**:
  - Divide the data into fixed-size blocks



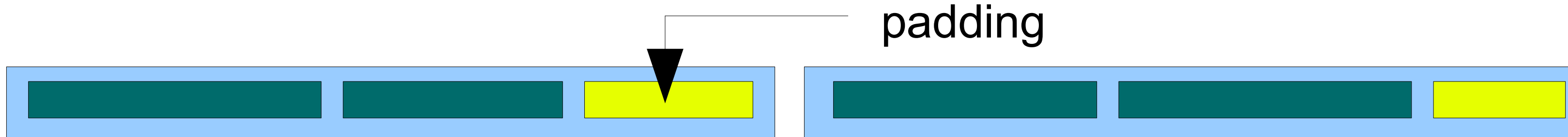
  - Insert a synchronizing string in regular intervals



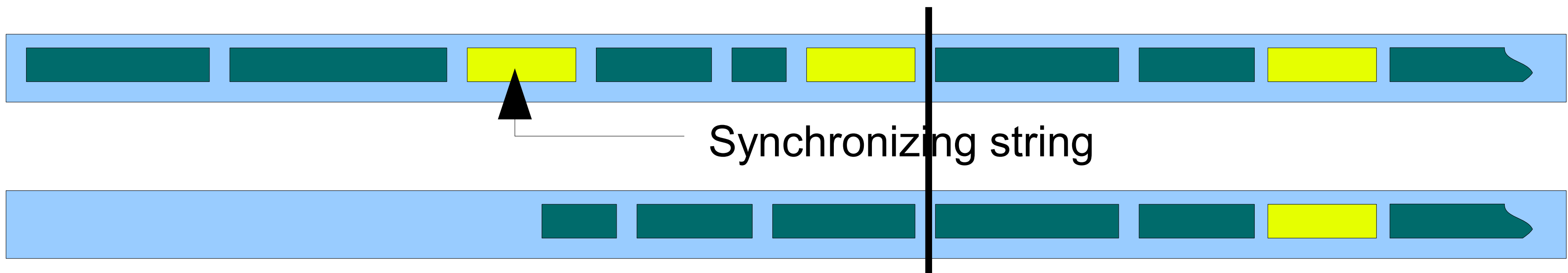
Guaranteed synchronization



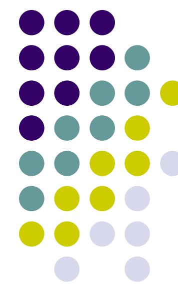
- Goal:
  - limit the synchronization delay to < *L* bits
- **Simple approaches**:
  - Divide the data into fixed-size blocks



  - Insert a synchronizing string in regular intervals

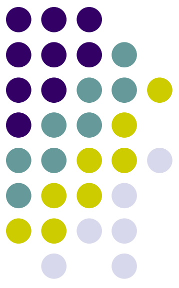


Guaranteed synchronization



- Additional goals:
  - keep the code optimal
  - exploit statistical synchronization
- Realization: M.Biskup, DCC'08:
  Guaranteed synchronization of Huffman codes
  - Second type: arbitrary start position
  - Little redundancy
  - Large computational overhead

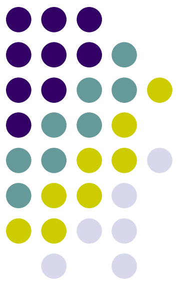
Guaranteed synchronization with known start position



- First type:
  - Divide data into blocks
    - The decoder may start only at position Ki, $i \in N$
  - Equivalently, the decoder always knows the position where it starts
  - But insert the marker only if necessary

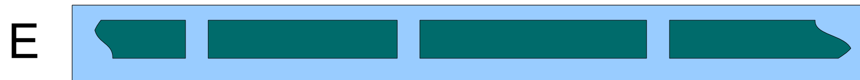
- In comparison to Biskup '08:
  - Much faster
  - Even less redundancy
  - But doesn't handle bit insertion/deletion errors

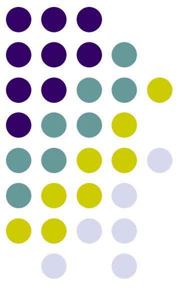
Encoder



- Blocks of length K; encode the first block normally, then encode the second block:

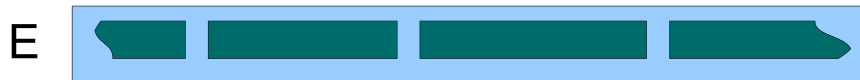
E



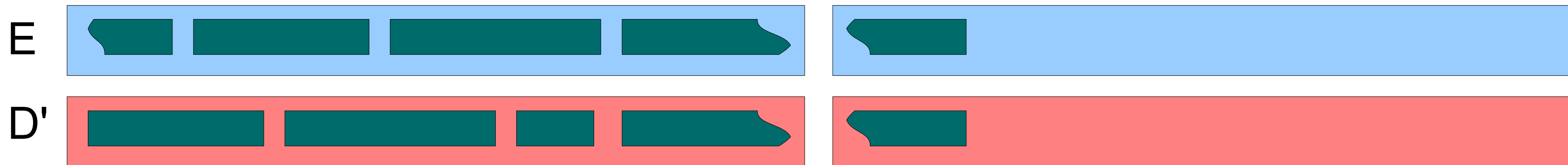


Encoder

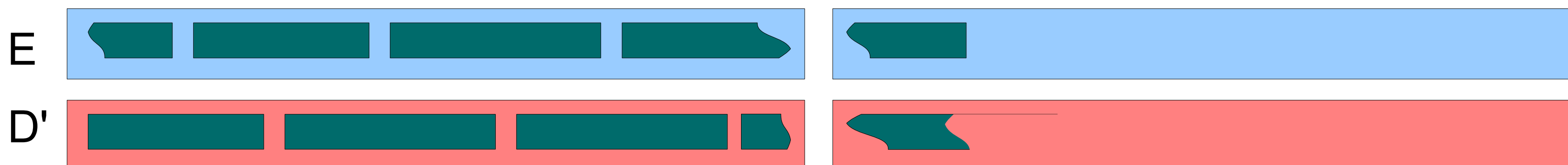
- Blocks of length K; encode the first block normally, then encode the second block:

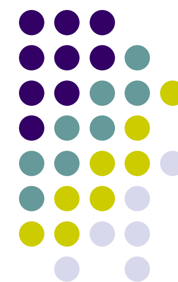


- Check the result of decoding this block
  - Resynchronization:



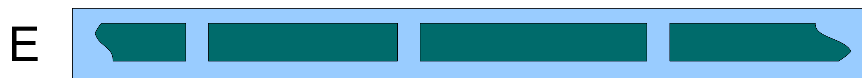
  - No resynchronization:



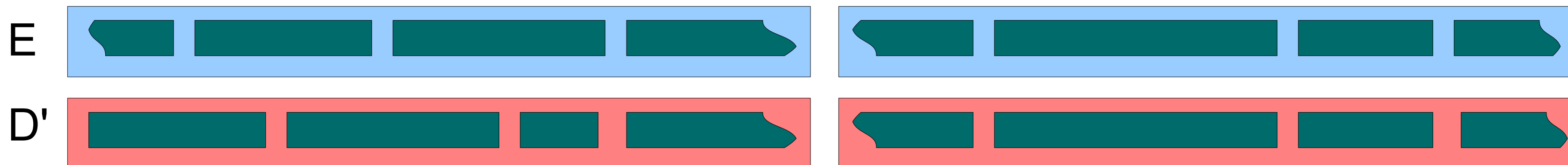


Encoder

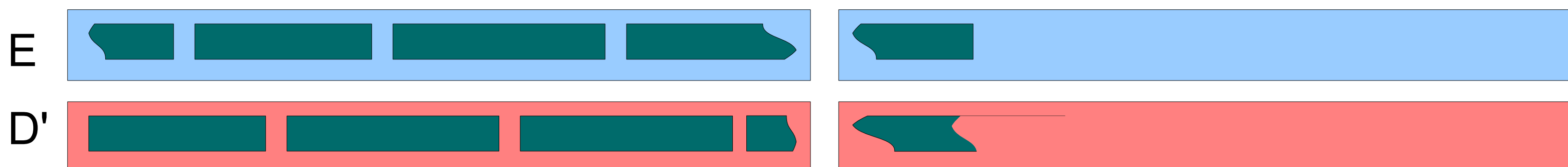
- Blocks of length K; encode the first block normally, then encode the second block:



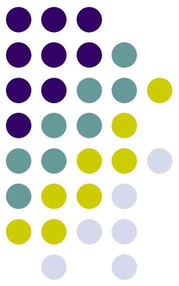
- Check the result of decoding this block
  - Resynchronization:



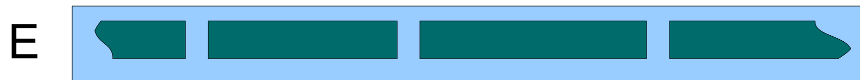
  - No resynchronization:



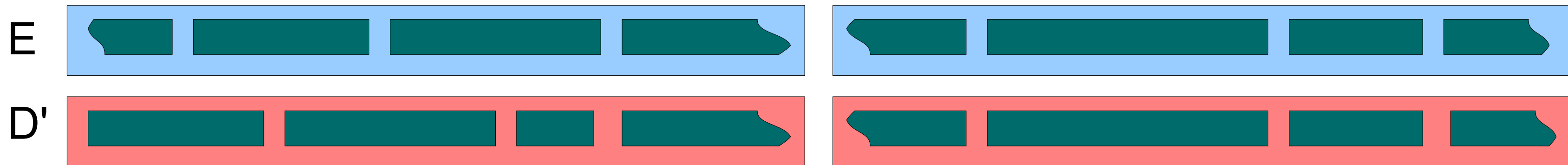
Encoder



- Blocks of length K; encode the first block normally, then encode the second block:

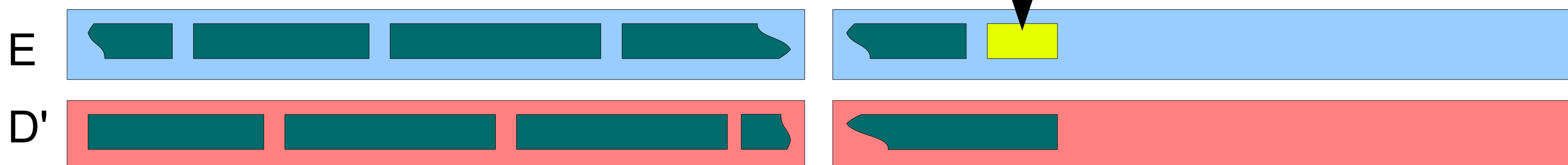


- Check the result of decoding this block
  - Resynchronization:

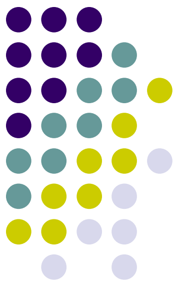


  - No resynchronization:

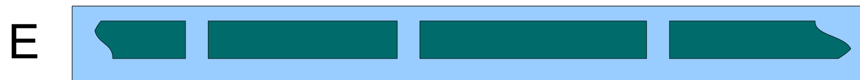
Leaf string



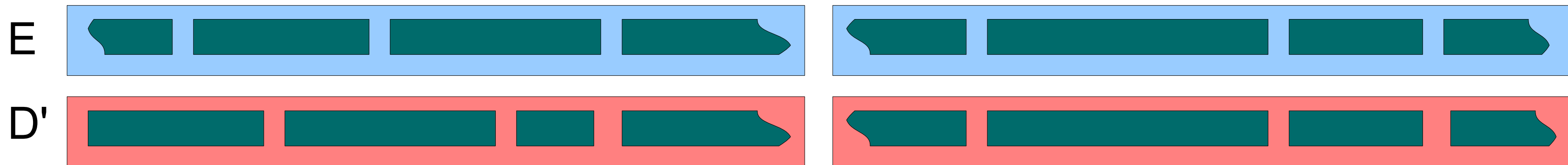
Encoder



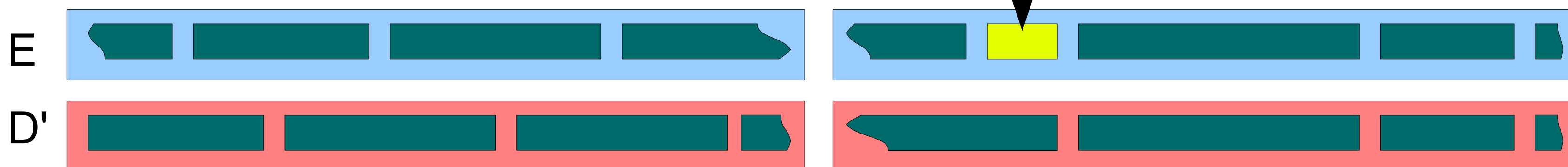
- Blocks of length K; encode the first block normally, then encode the second block:



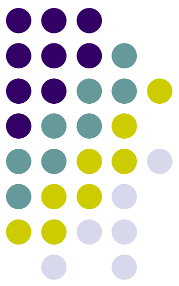
- Check the result of decoding this block
  - Resynchronization:



  - No resynchronization:



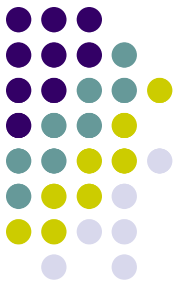
Encoder algorithm (simple!)



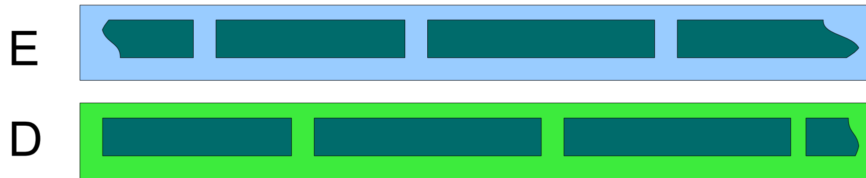
Algorithm 1: Encoder

```
1  i ← 0;                                   ◁ i is used only for the description of the algorithm
2  p ← 0;                     ◁ the current position relative to the bit iK (the position of D_{iK})
3  q ← ε;                                   ◁ the state of decoder D_{iK} at the current position
4  while (a ← M.NEXTSYMBOL()) ≠ NULL do
5     output c(a);                                                ◁ encode the symbol normally
6     q ← δ*(q, c(a));                                                ◁ update the state of D_{iK}
7     p ← p + |c(a)|;                                             ◁ update the position of D_{iK}
8     if p ≥ K then                   ◁ make sure D_{iK} is in sync and start with D_{(i+1)K}
9        s ← suffix of c(a) of length p − K;          ◁ the overflow from i^th K-bit block
10       output L(q);                         ◁ synchronize D_{iK}; recall that L(ε) = ϵ
11       q ← δ*(ε, sL(q));                 ◁ the state of D_{(i+1)K} at the current position
12       p ← |sL(q)|;                        ◁ the number of bits processed by D_{(i+1)K}
13       i++;                                            ◁ proceed to the next K-bit block
```

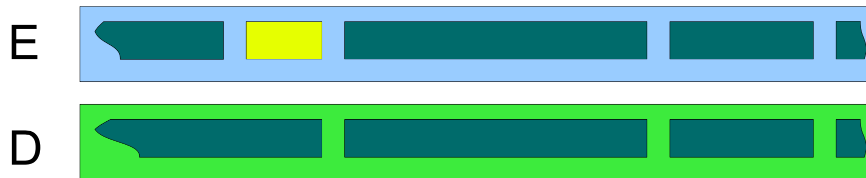
Decoder



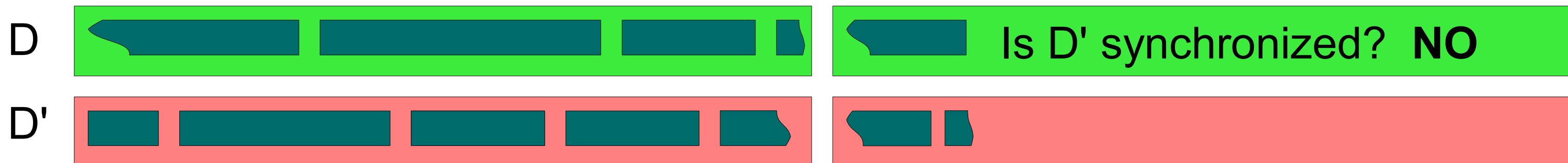
- Decoder the first block (K bits) normally



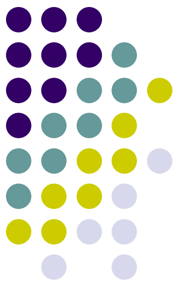
- Decode the second block



- Track the decoder of the second block

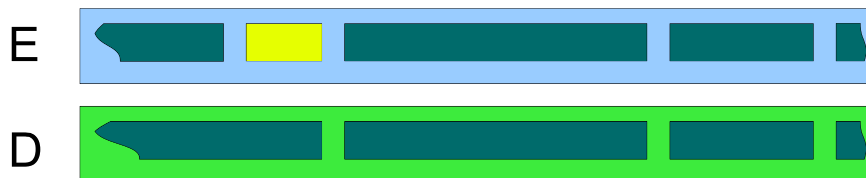


Decoder



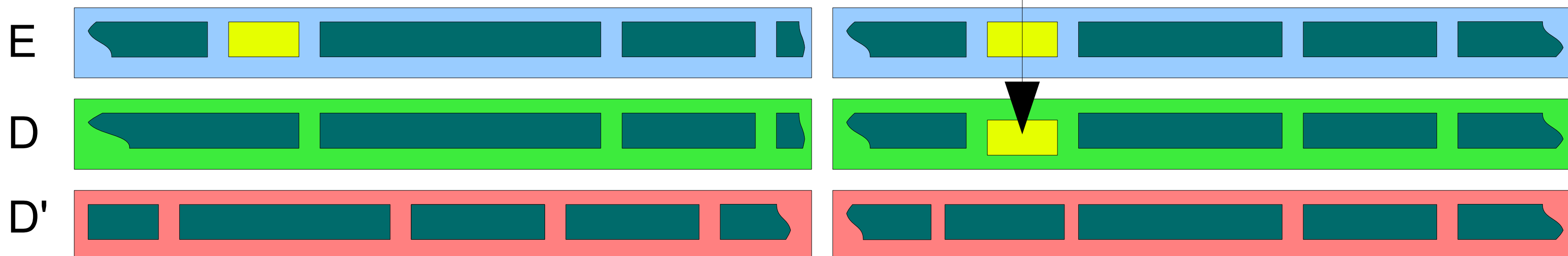
- Decoder the first block (K bits) normally

- Decode the second block

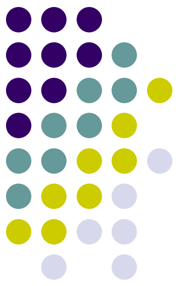


Leaf string
detected by decoder

- Track the decoder of the second block



Evaluation – time

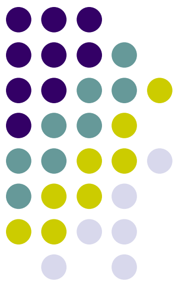


- Time overhead:
  - Silesia Corpus files (~10MB)
  - Two possible implementations

	Linear	Sublinear
Processing:	$O(M)$	$o(M)$
Preprocessing:	$O(N)$	$O(N^2)$
Small code (256)	60%	16%
Large code (4096)	65%	300%

- Legend: *M* – message length *N* – code size

Evaluation – redundancy (%)



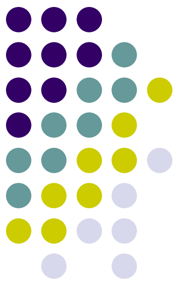
- Tests on Silesia Corpus (large files)
- Small codes (256 elem.)

	K=50	K=100	K=200	K=500
Simple code	0,097	0,018	0,002	3,9E-5
Canonical code	0,106	0,019	0,002	4,0E-5
Fixed-order code	0,014	0,002	2,6E-4	6,7E-6

- Large codes (4096 elem.)

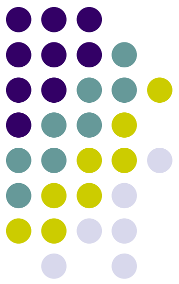
	K=50	K=100	K=200	K=500
Simple code	0,321	0,094	0,017	4,3E-4
Canonical code	0,371	0,089	0,011	1,7E-4
Fixed-order code	0,038	0,003	9,4E-5	5,0E-6

Application: Parallel Huffman decompression

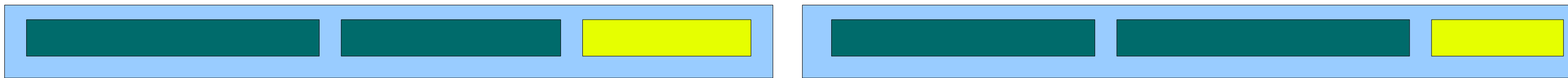


- Requirements:
  - divide data into blocks
  - more blocks than processors
  - Increase granularity at the end
- Encoder support (to find codeword alignment)
  - But see Klein and Wiseman, 2003
- Methods:
  - WC (Whole Codewords)
  - LogH
  - KSP (Guaranteed Synchronization with Known Start Position)

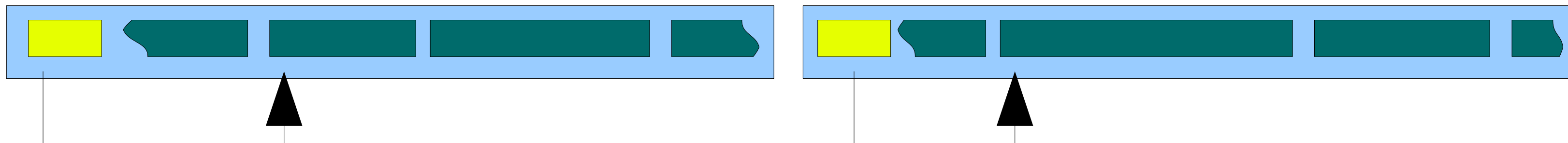
Division into blocks



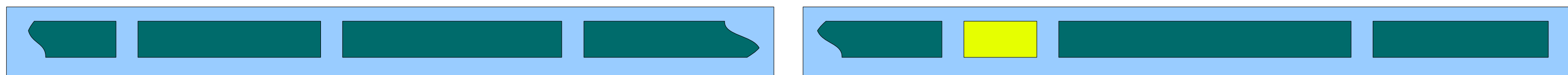
- Whole Codewords, – padding



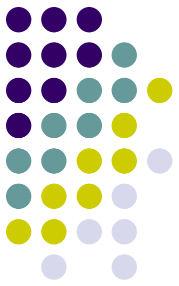
- LogH, – pos. of the first complete codeword



- KSP: – (optional) resynchronization marker



Tests on Jpeg

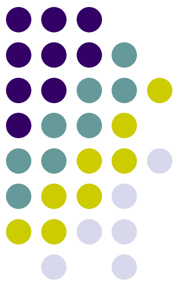


- Redundancy [%]

	Q = 1			Q = 10		
K	**WC**	**LogH**	**KSP**	**WC**	**LogH**	**KSP**
50	10,72	11,07	0,91	13,29	11,33	0,84
100	4,78	5,24	0,19	6,30	5,32	0,11
200	2,52	2,56	0,03	3,03	2,58	0,01
500	0,92	1,01	5E-3	1,18	1,01	1E-3
10000	0,04	0,05	0 bits	0,06	0,05	1 bit

- KSP is much better
  - For large blocks any method is good.

Summary



- Modification of Huffman coding
  - the decoder always resynchronizes in L bits
  - start position of the decoder must be known
- Application
  - division of Huffman data into blocks
  - limiting error propagation
- Advantages:
  - little redundancy,
  - reasonable processing overhead
  - simple algorithm