Content-assisted File Decoding for Nonvolatile Memories

Yue Li, **Yue Wang**, **Anxiao (Andrew) Jiang**
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77843
{yli, yuewang, ajiang}@cse.tamu.edu

Jehoshua Bruck
Department of Electrical Engineering
California Institute of Technology
Pasadena, CA 91125
bruck@caltech.edu

***Abstract*—Nonvolatile memories (NVMs) such as flash memories play a significant role in meeting the data storage requirements of today's computation activities. The rapid increase of storage density for NVMs however brings reliability issues due to closer alignment of adjacent cells on chip, and more levels that are programmed into a cell. We propose a new method for error correction, which uses the random access capability of NVMs and the redundancy that inherently exists in information content. Although it is theoretically possible to remove the redundancy via data compression, existing source coding algorithms do not remove all of it for efficient computation. We propose a method that can be combined with existing storage solutions for text files, namely *content-assisted decoding*. Using the statistical properties of words and phrases in the text of a given language, our decoder identifies the location of each subcodeword representing some word in a given input noisy codeword, and flips the bits to compute a most likely word sequence. The decoder can be adapted to work together with traditional ECC decoders to keep the number of errors within the correction capability of traditional decoders. The combined decoding framework is evaluated with a set of benchmark files.**

I. Introduction

Nonvolatile memories (NVMs), such as flash memories, have excellent speed and storage capacity. They have emerged as a crucial technology for storage systems. However, accompanying the improvement in data density, the reliability issue of NVMs are attracting more and more attention [1]. In this paper, we propose a new method for error correction named *content-assisted decoding*. Our method uses the fast random access capability of NVMs and the redundancy that inherently exists in information content. Although it is theoretically possible to remove the redundancy via data compression, existing source coding algorithms do not remove all of it for efficient computation. Our method can be combined with existing storage solutions for text files. With dictionaries storing the statistical properties of words and phrases of the same language, our decoder first breaks the input noisy codeword into subcodewords, with each subcodeword corresponding to a set of possible words. The decoder then flips the bits in each noisy subcodeword to select a most likely word sequence as the correction. Consider the example in Figure 1. The

	Codeword	**Text**
Huffman encoding	$(1,0,0,0,0,1,1,1)$	*I am*
ECC encoding	$(1,0,0,0,0,1,1,1,\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{1})$	*I am*
Noise received	$(1,0,\underline{1},0,0,1,\underline{0},1,\mathbf{0},\underline{\mathbf{0}},\mathbf{1},\mathbf{1})$	*IIaa*
ECC decoding failure	$(1,0,\underline{1},0,0,1,\underline{0},1,\mathbf{0},\underline{\mathbf{0}},\mathbf{1},\mathbf{1})$	*IIaa*
Content-assisted decoding	$(1,0,0,0,0,1,1,1,\mathbf{0},\underline{\mathbf{0}},\mathbf{1},\mathbf{1})$	*I am*
ECC decoding success	$(1,0,0,0,0,1,1,1,\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{1})$	*I am*

Fig. 1. An example on correcting errors in the codeword of a text.

English text *"I am"* is stored using the Huffman coding: $\{I \to (1,0), \sqcup \to (0,0), a \to (0,1), m \to (1,1)\}$, where $\sqcup$ denotes a space. The information bits are encoded with a $(12,8)$-shortened Hamming code which corrects single bit errors (the bold bits denote the parity check bits). Assume that three errors (marked by the underlines) are received by the codeword. The number of errors exceeds the code's correction capability, and ECC decoding fails. Our decoder takes in the noisy codeword, and corrects the errors in the information symbols by looking up a dictionary which contains two words $\{I, am\}$. This brings the number of errors down to one. Therefore, the second trial of ECC decoding succeeds, and all the errors are corrected. Our approach is suitable for natural languages, and can potentially be extended to other types of data where the redundancy in information content is not fully removed by data compression. The scheme takes advantage of the fast random access speed provided by flash memories for fast dictionary look-up and content verification. For performance evaluation, we have tested a decoding framework that combines a soft decision decoder of low-density parity-check (LDPC) codes and our scheme with a set of text file benchmarks. Experimental results show that our decoder indeed increases the correction capability of the LDPC decoder.

The rest of the paper is organized as follows. Section II presents the preliminaries, and defines the text file decoding problem. Section III specifies the algorithms of the content-assisted file decoder. Section IV discusses implementation details and experimental results.

II. The Models of File Decoding

We first define a few notations used throughout this paper. Let $\mathbf{x}$ denote a binary *codeword* $(x_1, x_2, \cdots, x_n) \in \{0,1\}^n$, and we use $\mathbf{x}[i:j]$ to represent the *subcodeword* $(x_i, x_{i+1}, \cdots, x_j)$. Let the function $\text{length}(\mathbf{x})$ compute the

This work was supported in part by the NSF CAREER Award CCF-0747415 and the NSF grant CCF-1217944. The authors thank the Texas A&M University Brazos HPC cluster for providing computing resources to support the research reported here.

length of a codeword $\mathbf{x}$, and we use $\mathrm{d_H}(\mathbf{x}_1, \mathbf{x}_2)$ for computing the Hamming distance between two codewords of the same length. Let $\mathcal{A}$ be an alphabet set, and let $s \in \mathcal{A}$ be a symbol. We denote a *space* by $\sqcup \in \mathcal{A}$. A *word* $\mathbf{w} \triangleq (s_1, \cdots, s_n)$ of length n is a finite sequence of symbols without any space. A *phrase* $\mathbf{p} \triangleq (\mathbf{w}_1, \sqcup, \mathbf{w}_2)$ is defined as a combination of two words separated by a space. Define a text $\mathbf{t} \triangleq (\mathbf{w}_1, \sqcup, \mathbf{w}_2, \sqcup, \cdots, \sqcup, \mathbf{w}_n)$ as a sequence of words separated by $\sqcup$. A *word dictionary* $D_w \triangleq \{[\mathbf{w}_1 : p_1], [\mathbf{w}_2 : p_2], \cdots\}$ is a finite set of records where a record $[\mathbf{w} : p]$ has a key $\mathbf{w}$ and a value $p > 0$. The value p is an average probability that the word $\mathbf{w}$ occurs in a text. Similarly, a *phrase dictionary* $D_p \triangleq \{[\mathbf{p}_1 : p_1], [\mathbf{p}_2 : p_2], \cdots\}$ stores the probabilities that a set of phrases appear in any given text. The dictionary look-up operations denoted by $D_w[\mathbf{w}]$ and $D_p[\mathbf{p}]$ return the probabilities of words and phrases, respectively. We use the notation $\mathbf{w} \triangleright D_w$ (or $\mathbf{p} \triangleright D_p$) to indicate that there is a record in D_w (or D_p) with key $\mathbf{w}$ (or $\mathbf{p}$). Let π_s be a bijective mapping between a symbol and a binary codeword, and let $\mathbf{x}_s = \pi_s(\sqcup)$. In this paper, the mapping π_s is used during data compression before ECC encoding, and it encodes each symbol separately. In the example of Section I, π_s refers to the Huffman codebook. The bijective mapping between a word $\mathbf{w} = (s_1, \cdots, s_n)$ and its binary codeword is defined as $\pi_w(\mathbf{w}) \triangleq (\pi_s(s_1), \cdots, \pi_s(s_n))$, and the bijective mapping from a text to its binary representation is defined as $\pi_t(\mathbf{t}) \triangleq (\pi_w(\mathbf{w}_1), \mathbf{x}_s, \cdots, \mathbf{x}_s, \pi_w(\mathbf{w}_n))$ where $\mathbf{x}_s = \pi_s(\sqcup)$. We use π_s^{-1}, π_w^{-1} and π_t^{-1} to denote the corresponding inverse mappings.

The model of the data storage channel is shown in Figure 2.

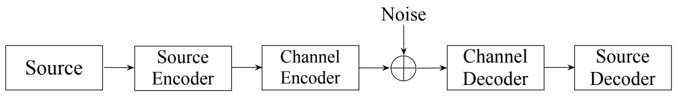


Fig. 2. The channel model for data storage.

A text $\mathbf{t}$ is generated by the source. The text is compressed by the source encoder, producing a binary codeword $\mathbf{y} = \pi_t(\mathbf{t}) \in \{0,1\}^k$. The compressed bits are fed to a channel encoder, obtaining an ECC codeword $\mathbf{x} = \psi(\mathbf{y}) \in \{0,1\}^n$ where $n > k$. Here we assume a systematic ECC is used. The codeword is then stored by memory cells, and receives an additive error $\mathbf{e} \in \{0,1\}^n$. In this paper, a binary symmetric channel (BSC) with bit-flipping rate f is assumed. When the cells are read, the channel outputs a noisy codeword $\mathbf{x}' = \mathbf{x} \oplus \mathbf{e}$ where $\oplus$ is the bit-wise exclusive-OR over codewords. The noisy codeword is first corrected by a channel decoder, producing an estimated ECC codeword $\hat{\mathbf{y}} = \psi^{-1}(\mathbf{x}')$. The source decoder decompresses the corrected codeword, and returns an estimated text $\hat{\mathbf{t}} = \pi_t^{-1}(\hat{\mathbf{y}})$ upon success.

This work focuses on designing better channel decoders ψ^{-1} for correcting bit errors in text files. We propose a new decoding framework which connects a traditional ECC decoder with a *content-assisted decoder* (CAD) as shown in Figure 3. A noisy codeword is first passed into an ECC decoder. If decoding fails, the decoding output is passed to

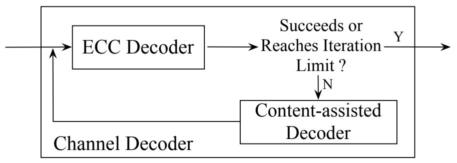


Fig. 3. The work-flow of a channel decoder with content-assisted decoding.

CAD. With the statistical information stored in D_w and D_p, the CAD selects a word for each subcodeword to form a likely text as the correction for the noisy codeword. The corrected text is fed back to the ECC decoder. The iteration continues until either the ECC decoder succeeds or an iteration limit is reached. The text file decoding problem for our CAD is defined as follows.

Definition 1. *Let* $\mathbf{t}$ *be some text generated from the source, and let* $\mathbf{x}' \in \{0,1\}^n$ *be a noisy channel output codeword of* $\mathbf{t}$*. Given two dictionaries* D_w *and* D_p*, the text file decoding problem for the CAD is to find an estimated text* $\hat{\mathbf{t}}$ *which is the most likely correction for* $\mathbf{x}'$, i.e.

$$\underset{\hat{\mathbf{t}}}{\operatorname{argmax}} \Pr\{\hat{\mathbf{t}} \mid \mathbf{x}', D_p, D_w\}.$$

III. The Content-Assisted Decoding Algorithms

The CAD approximates the solution to the problem in Definition 1 in the three steps: (1) estimate space positions in the noisy codeword to divide the codeword into subcodewords, with each subcodeword representing a set of words in D_w. (2) Resolve ambiguity by selecting a word for each subcodeword to form a most likely sequence. (3) Perform post-processing to revert the aggressive bit flips done in (1) and (2). We describe the algorithm of each step in this section.

A. Creating dictionaries

The dictionaries D_w and D_p are used in our decoding algorithms. To create the dictionaries, we simply count the frequencies of words and phrases of two words which appear in a relatively large set of different texts in the same language as the texts generated by the source. Fast dictionary look-up is achieved by storing the dictionaries in a content-addressable way thanks to the random access in flash memories, *i.e.*, the probability in a dictionary record is addressed by the value of the corresponding word or phrase. As we show later in section IV, the completeness of the dictionaries effects the decoding performance.

B. Codeword segmentation

The codeword segmentation function σ takes in a noisy codeword and a word dictionary, then flips the minimum number of bits to make the corrected codeword represent a text, *e.g.*, a sequence of valid words separated by spaces. If $\sigma(\mathbf{x}, D_w) = ((\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k), (i_1, i_2, \cdots, i_{k-1}))$, where the number of records $|D_w|$ is bounded by some constant K, and $i_j \in \mathbb{N}$ is the index of the first bit of the j-th space in $\mathbf{x}$, the subcodeword $\mathbf{x}_1 = \mathbf{x}[1 : i_1 - 1]$, $\mathbf{x}_k = \mathbf{x}[i_{k-1} + \mathrm{length}(\mathbf{x}_s) : \mathrm{length}(\mathbf{x})]$, and $\mathbf{x}_j = \mathbf{x}[i_{j-1} + \mathrm{length}(\mathbf{x}_s) : i_j - 1]$ for $j \in \{2, 3, \cdots, k-1\}$. The mapping σ is required to satisfy the following properties: (1) for each subcodeword $\mathbf{x}_j$, $\exists \mathbf{w} \triangleright D_w$ such that $\mathrm{length}(\mathbf{x}_j) = \mathrm{length}(\pi_w(\mathbf{w}))$.

(2) $\mathrm{d_H}(\mathbf{x}, (\mathbf{x}_1, \mathbf{x}_s, \mathbf{x}_2, \mathbf{x}_s, \cdots, \mathbf{x}_s, \mathbf{x}_k))$ is minimized. Intuitively, as the bit-flip rate f is very small (which is common for NVM channels), the segmentation function is a maximum likelihood decoder which flips the minimum number of bits of the codeword. Let the cost function $\mathrm{c}(i,j)$ return the minimum number of flips taken to convert the subcodeword $\mathbf{x}[i:j]$ to represent a text. We have the following recurrence:

$$\mathrm{c}(i,j) \triangleq \begin{cases} \min\{\mathrm{g}(i,j), \mathrm{h}(i,j)\} & \text{if } i < j \\ \infty & \text{otherwise} \end{cases},$$

where

$$\begin{aligned} \mathrm{g}(i,j) &\triangleq \min\nolimits_{\mathbf{w} \triangleright D_w} \mathrm{d_H}(\pi_w(\mathbf{w}), \mathbf{x}[i:j]), \\ \mathrm{h}(i,j) &\triangleq \min\nolimits_{k \in [i+1, j-\mathrm{length}(\mathbf{x}_s)]} \\ &\quad \mathrm{c}(i,k-1) + \mathrm{c}(k+\mathrm{length}(\mathbf{x}_s), j) + \\ &\quad \mathrm{d_H}(\mathbf{x}[k:k+\mathrm{length}(\mathbf{x}_s)-1], \mathbf{x}_s). \end{aligned}$$

The function $\mathrm{g}(i,j)$ computes the minimum number of flips taken to turn $\mathbf{x}[i:j]$ into the codeword of a word in D_w. The function $\mathrm{h}(i,j)$ computes the minimum flip cost taken to obtain a codeword representing a text with at least two words.

Example 2. *Consider the example in section* I. *The input noisy codeword* $\mathbf{x}' = (1,0,1,0,0,1,0,1)$, *and the word dictionary* $D_w = \{[I : 0.5], [am : 0.5]\}$. *We have* $\sigma(\mathbf{x}', D_w) = (((1,0),(0,1,0,1)),(3))$. *Starting from* $c(1,8)$, *we recursively compute* $\mathrm{c}(i,j)$ *for all* $i < j$. *The results are shown in Figure* 4(b). *For instance, to compute* $\mathrm{c}(5,8)$, *we first compute* $\mathrm{g}(5,8) = 1$ *as the subcodeword can be turned to represent the word "I" with* 1 *bit-flip. We then compute* $\mathrm{h}(5,8) = \infty$. *This is because* $\mathrm{length}(\mathbf{x}_s) = 2$ *and the minimum codeword length of a word in* D_w *is* 2, *therefore it is impossible to split the subcodeword* $(0,1,0,1)$ *by a space. Finally, we have* $\mathrm{c}(5,8) = \min(1,\infty) = 1$.

Our objective is to compute $\mathrm{c}(1,n)$ given an input codeword of length n, and find out the space positions which help achieve the minimum cost. When $\mathrm{c}(i,j)$ is computed recursively starting from $\mathrm{c}(1,n)$, some entries will be recomputed unnecessarily. For instance, in example 2, the entry $\mathrm{c}(4,5)$ needs to be computed when we compute $\mathrm{c}(1,7)$ and $\mathrm{c}(2,8)$. A good way for speeding up such computation is to use dynamic programming techniques shown in Algorithm 1, which computes the final result iteratively starting from $\mathrm{c}(1,2)$, an entry computed in the previous iteration is saved for later iterations. The algorithm treats $\mathrm{c}(i,j)$ as the entries of a two dimensional table. Starting from $\mathrm{c}(1,2)$, the table the algorithm fills each entry diagonally across the table as shown in Figure 4(a). The corresponding space locations for breaking the subcodeword $\mathbf{x}[i:j]$, or the set of words that $\mathbf{x}[i:j]$ can be flipped to represent is recorded using a two dimensional table m. In practice, as f is close to 0, the average number of errors in the codeword of a word is small. Computing the set of possible words S_w for a given noisy codeword can be accelerated by passing an additional Hamming distance limit d to reduce the search space, i.e. instead of searching the whole D_w as in $g(i,j)$, we search the set $\{\mathbf{w} \mid \mathbf{w} \triangleright D_w, \mathrm{d_H}(\pi_w(\mathbf{w}), \mathbf{x}[i:j]) < d\}$ to skip the words which are too far from the noisy codeword in terms of d and Hamming distance metric. As we are more

Algorithm 1 CodewordSegmentation($\mathbf{x}, D_w$)

```
$n \leftarrow \mathrm{length}(\mathbf{x})$, $l \leftarrow \mathrm{length}(\mathbf{x}_s)$
Let $c$ and $m$ be two $n \times n$ tables
Let $wordSets$ and $spaces$ be two empty lists
for $t$ from 1 to $n$ do
    for $i$ from 1 to $n - t + 1$ do
        $j \leftarrow i + t - 1$
        $d_{\min} \leftarrow \min_{\mathbf{w} \triangleright D_w} \mathrm{d_H}(\pi_w(\mathbf{w}), \mathbf{x}[i:j])$
        $S_w \leftarrow \{\mathbf{w} \mid \mathbf{w} \triangleright D_w, \mathrm{d_H}(\pi_w(\mathbf{w}), \mathbf{x}[i:j]) = d_{\min}\}$
        $k' \leftarrow 0$
        for $k$ from $i+1$ to $j - l$ do
            $d' \leftarrow c(i,k) + c(k+l,j) + \mathrm{d_H}(\mathbf{x}_s, \mathbf{x}[k:k+l-1])$
            if $d' < d_{\min}$ then
                $d_{\min} \leftarrow d'$
                $k' \leftarrow k$
        if $k' = 0$ then
            $m(i,j).words \leftarrow S_w$
        else
            $m(i,j).words \leftarrow \emptyset$
            $m(i,j).space \leftarrow k'$
        $c(i,j) \leftarrow d_{\min}$
TraceBack$(1, n, spaces, wordSets, m, l)$
return $wordSets$ and $spaces$
```

Algorithm 2 TraceBack($i, j, spaces, wordSets, m, l$)

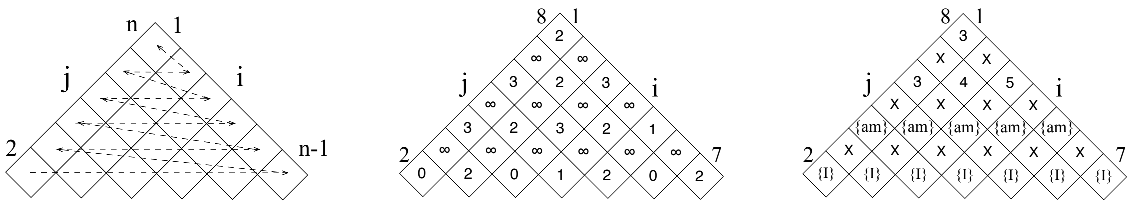
```
if $m(i,j).words = \emptyset$ then
    $k \leftarrow m(i,j).space$
    TraceBack$(i, k-1, spaces, m, l)$
    $spaces.$append$(k)$
    TraceBack$(k+l, j, spaces, m, l)$
else
    $wordSets.$append$(m(i,j).words)$
```

interested in the space locations than the value of $\mathrm{c}(i,j)$, after the entries of c and m have been filled, Algorithm 2 is used to recursively trace back the solution path recorded in m. The results are the ordered space locations and the sets of words for the codewords between the spaces. Assume that K is a constant which is much smaller than N, and that the codeword of each word has limited length bounded by some constant. The time complexity of our dynamic programming algorithm is $\mathcal{O}(n)$. This is because only $\mathcal{O}(n)$ entries need to be computed and each computation takes $\mathcal{O}(1)$ time. The algorithm requires $\mathcal{O}(n^2)$ space for storing the tables c and m.

Example 3. *For the example in section* I, *the tables* c *and* m *computed by Algorithm 1 are shown in Figure* 4(b) *and* 4(c). *The minimum flipping cost is* $c(1,8) = 2$, *and the index of the estimated space is* $m(1,8).space = 3$. *With the estimated space, the subcodeword* $\mathbf{x}[1:2] = (1,0)$ *can be flipped to denote a word in the set* $\{I\}$, *and the subcodeword* $\mathbf{x}[5:8] = (0,1,0,1)$ *can be flipped to denote a word in the set* $\{am\}$.

C. Ambiguity resolution

Given the subcodewords $(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k)$ between the estimated spaces, and a list of word sets $(\mathbf{W}_1, \mathbf{W}_2, \cdots, \mathbf{W}_k)$ computed from the codeword segmentation algorithm, for $i \in \{1, \cdots, k\}$ we select a word $\mathbf{w}_i$ from $\mathbf{W}_i$ to form a most probable text $\hat{\mathbf{t}} = (\mathbf{w}_1, \sqcup, \mathbf{w}_2, \sqcup, \cdots, \sqcup, \mathbf{w}_k)$. The codeword $\pi_t(\hat{\mathbf{t}})$ is a correction for the input noisy codeword. Specifically,



(a) Iterative table filling. (b) Table c. (c) Table m.

Fig. 4. The examples of codeword segmentation. In Figure (c): A number in the table denotes the index of the first bit of an estimated space; a set of word means the subcodeword can be flipped to any of the word in the set. The cross $\times$ means a subcodeword can neither be flipped to represent a word nor to a text with at least two words.

this step is to compute

$$\operatorname{argmax}_{(\mathbf{w}_1,\mathbf{w}_2,\cdots,\mathbf{w}_k)\in\mathbf{W}_1\times\mathbf{W}_2\cdots\times\mathbf{W}_k} \Pr\{(\mathbf{w}_1,\mathbf{w}_2,\cdots,\mathbf{w}_k),(\mathbf{x}_1,\mathbf{x}_2,\cdots,\mathbf{x}_k)\}.$$

Let the function $\mathrm{P}(\mathbf{w}_i)$ compute the maximal joint probability when some word $\mathbf{w}_i$ is selected from $\mathbf{W}_i$ and appended to the previously selected word sequence $(\mathbf{w}_1,\mathbf{w}_2,\cdots,\mathbf{w}_{i-1})$. For $i\in[2,k]$, we have

$$\mathrm{P}(\mathbf{w}_i) \triangleq \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in\mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \Pr\{(\mathbf{w}_1,\cdots,\mathbf{w}_i),(\mathbf{x}_1,\cdots,\mathbf{x}_i)\}.$$

Assume the words in a text form a one-step Markov chain, *i.e.*, for $i\geq 2$, $\Pr\{\mathbf{w}_i \mid (\mathbf{w}_1,\mathbf{w}_2,\cdots,\mathbf{w}_{i-1})\} = \Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}$. Therefore, we rewrite the equation above as:

$$\begin{aligned}
&\mathrm{P}(\mathbf{w}_i)\\
&=\max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in\mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}\\
&\quad \Pr\{\mathbf{w}_1\}\Pr\{\mathbf{w}_2 \mid \mathbf{w}_1\}\cdots\Pr\{\mathbf{w}_{i-1} \mid \mathbf{w}_{i-2}\}\prod_{k=1}^{i-1}\Pr\{\mathbf{x}_k \mid \mathbf{w}_k\}\\
&=\max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in\mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}\\
&\quad \Pr\{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1}),(\mathbf{x}_1,\cdots,\mathbf{x}_{i-1})\}\\
&=\max_{\mathbf{w}_{i-1}\in\mathbf{W}_{i-1}} \Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}\Pr\{\mathbf{w}_i|\mathbf{w}_{i-1}\}\\
&\quad \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-2})\in\mathbf{W}_1\times\cdots\times\mathbf{W}_{i-2}}\\
&\qquad \Pr\{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1}),(\mathbf{x}_1,\cdots,\mathbf{x}_{i-1})\}\\
&=\max_{\mathbf{w}_{i-1}\in\mathbf{W}_{i-1}} \Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}\Pr\{\mathbf{w}_i|\mathbf{w}_{i-1}\}\,\mathrm{P}(\mathbf{w}_{i-1}).
\end{aligned} \tag{1}$$

and $\mathrm{P}(\mathbf{w}_1) = \Pr\{\mathbf{w}_1\}\Pr\{\mathbf{x}_1 \mid \mathbf{w}_1\}$. The conditional probability $\Pr\{\mathbf{x}_k \mid \mathbf{w}_k\}$ is computed from the channel statistics by $\Pr\{\mathbf{x}_k \mid \mathbf{w}_k\} = f^{\mathrm{d_H}(\pi_w(\mathbf{w}_k),\mathbf{x}_k)}(1-f)^{\mathrm{length}(\mathbf{x}_k)-\mathrm{d_H}(\pi_w(\mathbf{w}_k),\mathbf{x}_k)}$. The probabilities $\Pr\{\mathbf{w}_1\} = D_w[\mathbf{w}_1]$ and $\Pr\{\mathbf{w}_k \mid \mathbf{w}_{k-1}\} = D_p[(\mathbf{w}_{k-1},\sqcup,\mathbf{w}_k)]$ are looked up from the dictionaries:

The derived recurrence suggests that the optimization problem can be mapped to the problem of trellis decoding, which is again solved by dynamic programming. The trellis for our problem has k time stages. The observed codeword at the i-th stage is $\mathbf{x}_i$ for $i\in\{1,\cdots,k\}$. There are $|\mathbf{W}_i|$ vertices at stage i with each representing an element $\mathbf{w}$ of $\mathbf{W}_i$ and being associated with the conditional probability $\Pr\{\mathbf{w} \mid \mathbf{x}_i\}$. The weight of the directed edge from a vertex at stage i with word $\mathbf{w}_x$ to a vertex of stage $i+1$ with word $\mathbf{w}_y$ is the conditional probability $\Pr\{\mathbf{w}_y \mid \mathbf{w}_x\}$. An example of the mapping is shown in Figure 5. Our target is to compute the sequence which achieves $\max_{\mathbf{w}_k\in\mathbf{W}_k}\mathrm{P}(\mathbf{w}_k)$, which leads to the Viterbi path in the corresponding trellis starting from a vertex in stage 1 and ending at a vertex in stage k.

The dynamic programming algorithm for solving our trellis decoding problem is specified in Algorithm 3, which is adapted from the Viterbi decoding [2]. The final solution is computed iteratively, starting from $P(\mathbf{w}_1)$ according to the recurrence. When the last iteration is finished, we trace back along the Viterbi path recorded in the table s, collecting the selected words to form an estimated text $\hat{\mathbf{t}}$. The complexity of the Viterbi decoding algorithm is $\mathcal{O}(n^2k)$ where $k=\mathcal{O}(N)$ is the length of the input codeword list, and $n=\max_{i\in[1,k]}|\mathbf{W}_i| = \mathcal{O}(K)$ is the cardinality of the biggest input word set. As K is a constant which is much smaller than N, the Viterbi decoding for our case has time complexity $O(N)$. The algorithm requires $\mathcal{O}(nk)=\mathcal{O}(N)$ space for storing the tables p and s.

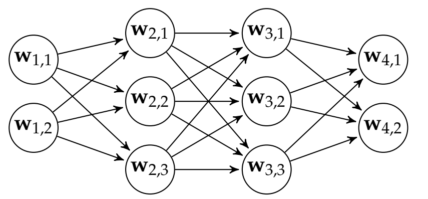


Fig. 5. Example of the mapping to trellis decoding. The word sets $\mathbf{W}_1 = \{\mathbf{w}_{1,1},\mathbf{w}_{1,2}\}$, $\mathbf{W}_2 = \{\mathbf{w}_{2,1},\mathbf{w}_{2,2},\mathbf{w}_{3,2}\}$, $\mathbf{W}_3 = \{\mathbf{w}_{3,1},\mathbf{w}_{3,2},\mathbf{w}_{3,3}\}$ and $\mathbf{W}_4 = \{\mathbf{w}_{4,1},\mathbf{w}_{4,2}\}$ respectively corresponds to the subcodewords $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and $\mathbf{x}_4$.

Algorithm 3 Viterbi$((\mathbf{W}_1,\cdots,\mathbf{W}_k),(\mathbf{x}_1,\cdots,\mathbf{x}_k),f,D_w,D_p)$

```
$n \leftarrow \max_{l\in[1,k]}|\mathbf{W}_l|$
Let $p$ and $s$ be two $n\times k$ tables
$p_{\max} \leftarrow 0$, $index \leftarrow 0$
for $t$ from 1 to $k$ do
    for $i$ from 1 to $|\mathbf{W}_t|$ do
        $p' \leftarrow f^{\mathrm{d_H}(\pi_w(\mathbf{W}_t[i]),\mathbf{x}_t)}(1-f)^{\mathrm{length}(\mathbf{x}_t)-\mathrm{d_H}(\pi_w(\mathbf{W}_t[i]),\mathbf{x}_t)}$
        if $t = 0$ then
            $p(i,t) \leftarrow p'\cdot D_w[\mathbf{W}_t[i]]$
        else
            $p_{\max} \leftarrow 0$, $index \leftarrow 0$
            for $j$ from 1 to $|\mathbf{W}_{t-1}|$ do
                $p'' \leftarrow p'\cdot D_p[(\mathbf{W}_{t-1}[j],\sqcup,\mathbf{W}_t[i])]\cdot p[j,t-1]$
                if $p'' > p_{\max}$ then
                    $p_{\max} \leftarrow p''$
                    $index \leftarrow j$
            $p(i,t) \leftarrow p_{\max}$
            $s(i,t) \leftarrow index$
$words \leftarrow [\mathbf{W}_k[index]]$
for $t$ from $k$ to 2 do
    $i \leftarrow s(index,t)$
    $words$.appendToFront$(\mathbf{W}_{t-1}[i])$
    $index \leftarrow i$
return $words$
```

D. Post-processing

If unknown words or phrases occur in the input codeword, additional errors will be introduced during codeword segmentation and ambiguity resolution. Unknown words (phrase) refers to new or rare words (phrases) which are not included in D_w (D_p). Upon encountering an unknown word, the codeword segmentation algorithm tends to split its codeword into subcodewords representing known words with the space symbol. Such segmentation introduces additional bit errors. We use a simple post-processing step which undoes the bit-flips issued by such aggressive segmentation. The idea is to use the phrase dictionary D_p to check whether two adjacent words returned by the Viterbi decoder is known to D_p. If so, the post-processor simply accepts the segmentation, otherwise the corresponding bits in the initial noisy codeword are used to replace the codewords for those unknown phrases. The complexity of this step is $\mathcal{O}(k) = \mathcal{O}(N)$.

IV. Experiments

A. Implementation detail

Our implementation supports the use of basic punctuation in the input text files, including ',', '.', '?' and '!'. This is done by adding another function in the definition of $c(i, j)$ when $i < j$. The function measures the number of flips taken to turn a subcodeword to represent a word followed by a punctuation. During ambiguity resolution, overflow may occur in the multiplications of probabilities when N is large. We thus use a logarithmic version of Eq.(1). Using additions instead of multiplications of floating point numbers significantly delays the overflow. A smoothing technique is used for computing $\Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}$. The probability $\Pr\{\mathbf{w}_i\}$ will be used if the phrase $(\mathbf{w}_{i-1}, \sqcup, \mathbf{w}_i)$ is unknown to D_p. The reason is that returning 0 for unknown phrases suddenly makes the whole joint probability be 0 in Eq.(1) and cancels the path.

B. Evaluation

We evaluated decoding performance of the channel decoder combining the LDPC sum-product decoder and the CAD. We compared the bit error rates (BER) of the combined channel decoder with those of the scheme using the LDPC sum-product decoding alone. The test inputs include 2 self-collected paragraphs and 8 paragraphs randomly extracted from the Canterbury Corpus, the Calgary Corpus, the Large Corpus [3], and the large text compression benchmark [4] (see Table I). The dictionaries are built using the books randomly extracted from Project Gutenberg [5]. The functions π_s and π_s^{-1} are implemented with Huffman coding. A $(3584, 3141)$-random LDPC code is used as the ECC. The iteration limit of the sum-product decoder is 32. The iteration threshold for the LDPC-CAD exchange is 3. The bit-flip rate of the BSC is 0.012, which makes the sum-product decoder fail to converge with high probability. The decoding BERs for complete and incomplete dictionaries are shown in Table I and Table II, respectively. The BERs for each benchmark are averaged from 1000 experiments. In Table I, the combined channel decoder significantly outperforms the traditional decoder thanks to the completeness of the dictionaries. The performance for the benchmark `world192` is not as good as others. This is because `world192` has much more punctuation but fewer words than other benchmarks do, and more errors occur in the punctuations which the CAD is not good at correcting. In Table II, to see the effectiveness of the post-processor, we also show the performance of the combined decoder without the post-processor. The completeness of the dictionaries determines the decoding performance. For instance, the benchmarks `world192` and `enwiki` have considerable number of words and phrases which are unknown to our dictionaries. The combined decoder without post-processing introduces additional errors by aggressively breaking the codewords of the unknown words into subcodewords separated with spaces. In such cases, the post-processor is able to recognize and revert most of the over-aggressive bit-flips. This greatly reduces the number of additional errors introduced due to the "ignorance" of the CAD. For the benchmark `confintro`, the performance of the decoder without post-processing is much better than that of the decoder using post-processing. This is because `confintro` has only a few unknown words but many technical phrases which are unknown to D_p. The unknown phrases makes the post-processor tend to revert reasonable corrections done in the previous steps.

TABLE I
The decoding BERs when the dictionaries are complete

Name	Category	From	ECC only	Combined
`email`	Email discussion	Calgary	8.6×10^{-3}	1.9×10^{-6}
`lcet`	Lecture notes	Canterbury	8.4×10^{-3}	0.0
`alice`	Novel	Canterbury	8.3×10^{-3}	2.6×10^{-6}
`confintro`	Call for paper	Self-made	8.7×10^{-3}	0.0
`bible`	The bible	Large	8.3×10^{-3}	3.2×10^{-6}
`asyoulike`	Shakespeare play	Canterbury	8.9×10^{-3}	3.8×10^{-6}
`plrabn`	Poetry	Canterbury	8.6×10^{-3}	0.0
`news`	Web news	Self-made	8.6×10^{-3}	8.4×10^{-6}
`enwiki`	Wikipedia texts	Large text	8.3×10^{-3}	0.0
`world192`	The world fact book	Large	8.3×10^{-3}	4.9×10^{-5}

TABLE II
The decoding BERs when the dictionaries are incomplete.

Name	ECC only	Combined	After PP	UW%	UP%
`email`	8.6×10^{-3}	1.2×10^{-3}	6.0×10^{-4}	0	14
`lcet`	8.4×10^{-3}	9.3×10^{-4}	1.2×10^{-3}	0	24
`alice`	8.3×10^{-3}	7.6×10^{-5}	0.0	0	2
`confintro`	8.7×10^{-3}	5.1×10^{-5}	3.5×10^{-3}	0.9	41
`bible`	8.3×10^{-3}	7.5×10^{-4}	1.1×10^{-3}	0.7	29
`asyoulike`	8.9×10^{-3}	4.1×10^{-4}	9.6×10^{-4}	0.8	15
`plrabn`	8.6×10^{-3}	7.2×10^{-3}	5.0×10^{-3}	2	33
`news`	8.6×10^{-3}	1.2×10^{-3}	2.1×10^{-3}	2	29
`enwiki`	8.3×10^{-3}	1.6×10^{-2}	4.0×10^{-3}	11	34
`world192`	8.3×10^{-3}	2.6×10^{-2}	9.2×10^{-3}	25	31

References

[1] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2012.

[2] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, april 1967.

[3] The Canterbury Corpus: http://corpus.canterbury.ac.nz/, 2012.

[4] Large Text Compression Benchmark: http://mattmahoney.net/dc/text.html, 2012.

[5] Project Gutenberg: http://www.gutenberg.org/, 2012.