Decoding Algorithms for Random Linear Network Codes

Janus Heide, Morten Pedersen, Frank Fitzek

Decoding Algorithms for Random Linear Network Codes

Janus Heide, Morten V. Pedersen, and Frank H.P. Fitzek

Falculty of Engineering and Science
Aalborg University, Aalborg, Denmark
jah@es.aau.dk

Abstract. We consider the problem of efficient decoding of a random linear code over a finite field. In particular we are interested in the case where the code is random, relatively sparse, and use the binary finite field as an example. The goal is to decode the data using fewer operations to potentially achieve a high coding throughput, and reduce energy consumption. We use an on-the-fly version of the Gauss-Jordan algorithm as a baseline, and provide several simple improvements to reduce the number of operations needed to perform decoding. Our tests show that the improvements can reduce the number of operations used during decoding with 10-20% on average depending on the code parameters.

Keywords: Network Coding; Algorithms; Implementation

1 Introduction

When implementing and deploying Network Coding (NC) at least two performance criteria are important; the magnitude of overhead added by the code, and the speed at which encoding, recoding and decoding can be performed. We consider the last issue and note that it is trivial to implement encoding such that the minimal number of operations are used. As recoding is similar to encoding we turn our attention to the problem of fast decoding.

A popular approach to network coding is Random Linear Network Coding (RLNC), introduced in [2]. It is based on finite fields, and it has been shown that high coding throughput can be obtained with this code when the binary finite field is used [1]. Additionally, as a randomly drawn element from the binary field is zero with high probability (50%), the resulting code will be sparse.

As data is encoded and recoded in a random way, there is no special structure or shortcut to exploit when performing decoding. Instead we are left with the tedious task of determining the inverse of operations performed during encoding/recoding. Additionally we prefer to perform decoding as packets arrive in order to avoid a large decoding delay when the final packet arrives. We know the resulting code is sparse, and therefore propose some simple mechanisms to utilize this fact. We have implemented these and their impact on the number of operations used during decoding.

In the remainder of this paper we introduce the used encoding approach, several decoding optimizations, and their measured impact on the decoding.

2 Coding Algorihtms

We consider encoding packets from some data to be sent from a source to a sink, we denote this data the generation. The generation consists of g pieces, called symbols each with a size of m bits, where g is called the generation size, and thus the generation contains $g \cdot m$ bits of data. The g symbols are arranged in the matrix $\boldsymbol{M} = [\boldsymbol{m}_1; \boldsymbol{m}_2; \ldots; \boldsymbol{m}_g]$, where $\boldsymbol{m}_i$ is a column vector. In practise some original file or data stream may be split into several generations, but here we only consider a single generation.

To generate a new encoded symbol $\boldsymbol{x}$, $\boldsymbol{M}$ is multiplied with a randomly generated coding vector $\boldsymbol{g}$ of length g, $\boldsymbol{x} = \boldsymbol{M} \times \boldsymbol{g}$. In this way we can construct $g + r$ coded symbols and coding vectors, where r is any number of redundant symbols as the code is rateless. When a coded symbol is transmitted on the network it is accompanied by its coding vector, and together they form a coded packet. A practical interpretation is that each coded symbol, is a combination or mix of the original symbols from the generation. The benefit is that nearly infinite coded symbols can be created.

$$\overbrace{\boxed{\text{Existing protocol header}}\,\boxed{\text{Coding vector } \boldsymbol{g}\;\Big|\;\text{Coded symbol } \boldsymbol{x}}}^{\text{Coded packet}}$$

2.1 Decoding

A sink must receive g linearly independent symbols and coding vectors from the generation to decode the data successfully. All received symbols are placed in the matrix $\hat{\boldsymbol{X}} = [\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_g]$ and all coding vectors are placed in the matrix $\hat{\boldsymbol{G}} = [\hat{\boldsymbol{g}}_1, \hat{\boldsymbol{g}}_2, \ldots, \hat{\boldsymbol{g}}_g]$, we denote $\hat{\boldsymbol{G}}$ the decoding matrix. Thus the vectors and symbols are row vectors in $\hat{\boldsymbol{G}}$ and $\hat{\boldsymbol{X}}$ respectively as this is more convenient during recoding. Hence we may perform any row operation on $\hat{\boldsymbol{G}}$ if we perform the same row operation on $\hat{\boldsymbol{X}}$.

The original data $\boldsymbol{M}$ can then be decoded as $\hat{\boldsymbol{M}} = \hat{\boldsymbol{X}} \times \hat{\boldsymbol{G}}^{-1}$ by the decoder. The problem is how to achieve this in an efficient way. We note that row operations on $\hat{\boldsymbol{X}}$ are more computationally expensive compared to operations on $\hat{\boldsymbol{G}}$, as generally $m >> g$.

Elements in the matrices are indexed row-column, thus $\hat{\mathbf{G}}[i, j]$ is the element in $\hat{\mathbf{G}}$ on the intersection between the i'th row and the j'th column. The i'th row in the matrix is indexed as $\hat{\mathbf{G}}[i]$. Initially no packets have been received, thus $\hat{\boldsymbol{G}}$ and $\hat{\boldsymbol{X}}$ are zero matrices. As we operate in the binary finite field we denote bitwise XOR of two bit strings of the same length as $\oplus$.

Algorithm 1: Decoder *initial state*

```
Input: $g, m$
Data: $\hat{\boldsymbol{G}} \leftarrow \mathbf{0}_{g \times g}$          ▷ The decoding matrix
Data: $\hat{\boldsymbol{X}} \leftarrow \mathbf{0}_{g \times m}$          ▷ The (partially) decoded data
Data: rank $\leftarrow 0$                                       ▷ the rank of $\hat{\boldsymbol{G}}$
```

2.2 Basic

As a reference we use the *basic* decoder algorithm, see Algorithm 5, described in [1]. This algorithm is a modified version of the Gauss-Jordan algorithm. On each run the algorithm attempts to get the decoding matrix into reduced echelon form. First the received vector and symbol $\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{x}}$ is forward substituted into the previous received vectors and symbols $\hat{\boldsymbol{G}}$ and $\hat{\boldsymbol{X}}$ respectively, and subsequently backward substitution is performed. If the packet was a linear combination of previous received packets it is reduced to the zero-vector $\mathbf{0}_g$ and discarded.

Algorithm 2: ForwardSubstitute

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$
pivotPosition ← 0                                  ▷ 0 Indicates that no pivot was found
for $i \leftarrow 1 : g$ do
    if $\hat{\boldsymbol{g}}[i] = 1$ then
        if $\hat{\boldsymbol{G}}[i, i] = 1$ then
            $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} \oplus \hat{\boldsymbol{G}}[i]$              ▷ substitute into new vector
            $\hat{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{x}} \oplus \hat{\boldsymbol{X}}[i]$              ▷ substitute into new symbol
        else
            pivotPosition ← $i$                       ▷ pivot element found
            break
return pivotPosition
```

Algorithm 3: BackwardsSubstitute

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$, pivotPosition
for $i \leftarrow$ (pivotPosition $- 1) : 1$ do
    if $\hat{\boldsymbol{G}}[i,$ pivotPosition$] = 1$ then
        $\hat{\boldsymbol{G}}[i] \leftarrow \hat{\boldsymbol{G}}[i] \oplus \hat{\boldsymbol{g}}$                   ▷ substitute into old vector
        $\hat{\boldsymbol{X}}[i] \leftarrow \hat{\boldsymbol{X}}[i] \oplus \hat{\boldsymbol{x}}$                   ▷ substitute into old symbol
```

Algorithm 4: InsertPacket

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$, pivotPosition
$\hat{\boldsymbol{G}}$[pivotposition] = $\hat{\boldsymbol{g}}$
$\hat{\boldsymbol{X}}$[pivotposition] = $\hat{\boldsymbol{x}}$
```

Algorithm 5: DecoderBasic

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$
pivotPosition = ForwardSubstitute($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$)
if pivotPosition > 0 then
    BackwardsSubstitute($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$, pivotPosition)
    InsertPacket($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$, pivotPosition)
    rank++
return rank
```

2.3 Suppress Null (SN)

To avoid wasting operations on symbols that does not carry novel information, we record the operations performed on the vector. If the vector is reduced to the zero vector, the packet was linearly dependent and the recorded operations are discarded. Otherwise the packet was novel and the recorded operations are executed on the symbol. This reduces the computational cost when a linear dependent packet is received. This is most likely to occur in the end phase of the decoding, thus it is most beneficial for small generation sizes. In real world scenarios the probability of receiving a linearly dependent packet can be high, in which cases this approach would be beneficial. To implement this, line 1 in Algorithm 5 is replaced with Algorithm 7.

Algorithm 6: ExecuteRecipe

```
Input: $\hat{\boldsymbol{x}}$,recipe
for $i \leftarrow 1 : g$ do
    if recipe$[i] = 1$ then
        $\hat{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{x}} \oplus \hat{\boldsymbol{X}}[i]$          ▷ substitute into symbol
```

Algorithm 7: ForwardSubstituteSuppressNull

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$
pivotPosition $\leftarrow 0$          ▷ 0 Indicates that no pivot was found
recipe $\leftarrow \mathbf{0}_g$
for $i \leftarrow 1 : g$ do
    if $\hat{\boldsymbol{g}}[i] = 1$ then
        if $\hat{\boldsymbol{G}}[i, i] = 1$ then
            $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} \oplus \hat{\boldsymbol{G}}[i]$          ▷ substitute into new vector
            recipe$[i] \leftarrow 1$
        else
            pivotPosition $\leftarrow i$          ▷ pivot element found
            break
if pivotPosition $> 0$ then
    ExecuteRecipe($\hat{\boldsymbol{x}}$,recipe)
return pivotPosition
```

2.4 Density Check (DC)

When forward substitution is performed there is a risk that a high density packets is substituted into a low density packet. The density is defined as Density$(\boldsymbol{h}) = \frac{\sum_{k=1}^{g}(h_k \neq 0)}{g}$, which is the number of non-zeros in the vector, and where $\boldsymbol{h}$ is the coding vector. Generally a sparse packet requires little work to decode and a dense packet requires much work to decode. When a vector is substituted into a sparse vector, the resulting vector will with high probability have higher density and thus *fill-in* occur. To reduce this problem incoming packets can be sorted based on density during forward substitution. When it is detected that two vectors have the same pivot element their densities are compared. The vector with the lowest density is inserted into the decoding matrix. The low density packet is then substituted into the high density packet, and the forward substitution is continued with the resulting packet. To implement this, line 1 in Algorithm 5 is replaced with Algorithm 8.

Algorithm 8: ForwardSubstituteDensityCheck

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$
pivotPosition ← 0                                ▷ 0 Indicates that no pivot was found
for $i \leftarrow 1 : g$ do
    if $\hat{\boldsymbol{g}}[i] = 1$ then
        if $\hat{\boldsymbol{G}}[i,i] = 1$ then
            if Density($\hat{\boldsymbol{g}}$) < Density($\hat{\boldsymbol{G}}[i]$) then
                $\hat{\boldsymbol{g}} \leftrightarrow \hat{\boldsymbol{G}}[i]$          ▷ swap new vector with old vector
                $\hat{\boldsymbol{x}} \leftrightarrow \hat{\boldsymbol{X}}[i]$          ▷ swap new symbol with old symbol
            $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} \oplus \hat{\boldsymbol{G}}[i]$
            $\hat{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{x}} \oplus \hat{\boldsymbol{X}}[i]$
        else
            pivotPosition ← $i$                  ▷ pivot element found
            break
return pivotPosition
```

2.5 Delayed Backwards Substitution (DBS)

To reduce the *fill-in* effect the backwards substitution is postponed until the decoding matrix has full rank. Additionally it is not necessary to perform any backwards substitution on the vectors because backwards substitution is performed starting from the last row. Hence when backwards substitution of a packet is complete, that packet has a pivot element for which all other encoding vectors are zero. This approach is only semi on-the-fly, as only some decoding is performed when packets arrive. Therefore the decoding delay when the final packet arrive will increase. This is implemented with Algorithm 9

Algorithm 9: DecoderDelayedBackwardsSubstitution

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$
pivotPosition = ForwardSubstitute($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$)
if pivotPosition > 0 then
    InsertPacket($\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$, pivotPosition)
    rank++
if rank = $g$ then
    BackwardsSubstituteFinal()
return rank
```

Algorithm 10: BackwardsSubstituteFinal

```
for $i \leftarrow g : 2$ do
    for $j \leftarrow (i-1) : 1$ do                          ▷ All rows above
        if $\hat{\boldsymbol{G}}[j, i] = 1$ then
            $\hat{\boldsymbol{X}}[j] \leftarrow \hat{\boldsymbol{X}}[j] \oplus \hat{\boldsymbol{X}}[i]$    ▷ substitute into the symbol
```

2.6 Density Check, and Delayed Backwards Substitution (DC-DBS)

When DC and DBS are combined vectors are sorted so sparse vectors are kept at the top of the decoding matrix while dense vectors are pushed downwards. Because backwards substitution is performed only when the rank is full, no fill-in occurs during backwards substitution, as only fully decoded packets are substituted back. To implement this, line 1 in Algorithm 9 is replaced with Algorithm 8.

2.7 Suppress Null, Density Check, and Delayed Backwards Substitution (SN-DC-DBS)

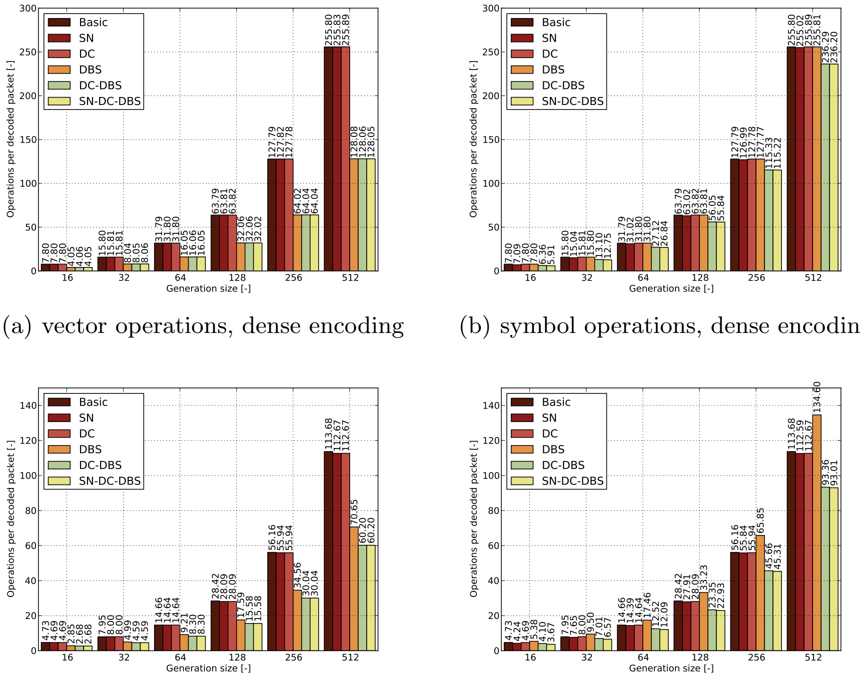
To reduce the cost of receiving linear dependent packets we include SN, by replacing line 1 in Algorithm 9 with Algorithm 11.

Algorithm 11: ForwardSubstitute-SN-DC

```
Input: $\hat{\boldsymbol{x}}$,$\hat{\boldsymbol{g}}$
pivotPosition $\leftarrow 0$                       ▷ 0 Indicates that no pivot was found
recipe $\leftarrow \mathbf{0}_g$
for $i \leftarrow 1 : g$ do
    if $\hat{\boldsymbol{g}}[i] = 1$ then
        if $\hat{\boldsymbol{G}}[i, i] = 1$ then
            if Density($\hat{\boldsymbol{g}}$) < Density($\hat{\boldsymbol{G}}[i]$) then
                ExecuteRecipe($\hat{\boldsymbol{x}}$,recipe)
                recipe $\leftarrow \mathbf{0}_g$                       ▷ reset recipe
                $\hat{\boldsymbol{g}} \leftrightarrow \hat{\boldsymbol{G}}[i]$              ▷ swap new vector with old vector
                $\hat{\boldsymbol{x}} \leftrightarrow \hat{\boldsymbol{X}}[i]$             ▷ swap new symbol with old symbol
            $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} \oplus \hat{\boldsymbol{G}}[i]$                ▷ substitute into new vector
            recipe$[i] \leftarrow 1$
    else
        pivotPosition $\leftarrow i$                     ▷ pivot element found
        break
if pivotPosition $> 0$ then
    ExecuteRecipe($\hat{\boldsymbol{x}}$,recipe)
return pivotPosition
```

3 Results

We have decoded a large number of generations with each of the optimizations, and measured the used vector and symbol operations which is $\oplus$ of two vectors, and two symbols respectively. We have considered two densities while encoding, $d = \frac{1}{2}$ and $d = \frac{\log_2(g)}{g}$ which we denote dense and sparse respectively. $d = \frac{1}{2}$ gives the lowest probability of linear dependence, and $d = \frac{\log_2(g)}{g}$ is a good trade-off between linear dependence and density. As a reference the mean number of both vector and symbol operations during encoding of one packet can be calculated as $\frac{g}{2}$ and $\log_2(g)$ for the dense and sparse case respectively.



(a) vector operations, dense encoding

(b) symbol operations, dense encoding

(c) vector operations, sparse encoding

(d) symbol operations, sparse encoding

Fig. 1. Vector and symbol operations during decoding, when $d = \frac{1}{2}$ and $d = \frac{\log_2(g)}{g}$.

With the *Basic* algorithm the number of operations during encoding and decoding is identical for the dense case, see Fig 1(a) and 1(b) and calculate $\frac{g}{2}$. For the sparse case the number of operations is significantly higher for decoding than for encoding, see Fig 1(c) and 1(d), and calculate $\log_2(g)$.

When *Suppress Null* is used the number of reduced symbol operations can be observed by subtracting the number of symbol operations from the number of vector operations. The reduction is highest for small g, which is where the ratio of linearly dependent packet is largest. For $g = 16$ and $g = 32$ the reduction in symbol operations is 9.5% and 4.4% respectively, for high $g's$ the reduction is approximately 0%.

Density Check reduces the number of operations for the sparse case marginally, but has no effect for the dense case.

The *Delayed Backwards Substitution* decreases the number of vector operations with approximately 40% when the encoding is sparse, and 50% when the encoding is dense as no operations needs to be performed on the vectors during backwards substitution. Interestingly the number of symbol operations increase significantly for the sparse encoding.

In *DC-DBS*, *density check* and *delayed backwards substitution* are combined, and the number of both data and vector operations are significantly reduced. For the sparse case the reduction is approximately 50% of the vector operations, and almost 20% of the symbol operations. For the dense case the reduction is approximately 50% of the vector operations, and almost 10% of the symbol operations. Interestingly the number of vector and symbol operations is lower for decoding compared to encoding, in the dense case. Hence the combination of *DC* and *DBS* is significantly better than the two alone, as the expensive symbol operations are reduced.

With *SN-DC-DBS* we additionally include *Suppress Null*, the number of symbol operations is reduced slightly for high g and significantly for low g.

4 Conclusion

The considered decoding optimizations have been shown to reduce the number of necessary operations during decoding. The reduction in symbol decoding operations is approximately 10%, and 20%, when the density during encoding is dense and sparse respectively. In both cases the number of vector operations is approximately halved. Surprisingly decoding can in some cases be performed with fewer operations than encoding.

Acknowledgment

This work was partially financed by the CONE project (Grant No. 09-066549/FTP) granted by the Danish Ministry of Science, Technology and Innovation, and the ENOC project in collaboration with Nokia, Oulu.

References

1. Heide, J., Pedersen, M.V., Fitzek, F.H., Larsen, T.: Network coding for mobile devices - systematic binary random rateless codes. In: The IEEE International Conference on Communications (ICC). Dresden, Germany (14-18 June 2009)
2. Ho, T., Koetter, R., Médard, M., Karger, D., ros, M.: The benefits of coding over routing in a randomized setting. In: Proceedings of the IEEE International Symposium on Information Theory, ISIT '03 (June 29 - July 4 2003)