Robust implementation of LT Codes encoding/decoding process.

Industrial Training Defense (EC4T001) Report submitted in partial fulfillment for the award of degree of

Bachelor of Technology

in

Electronics and Communication Engineering

by

Anurag Paul
20EC01045

Under the supervision of

Dr. Siddhartha S. Borkotoky

SCHOOL OF ELECTRICAL SCIENCES

INDIAN INSTITUTE OF TECHNOLOGY BHUBANESWAR

May 15 - July 15, 2023

Acknowledgement

I express my deep sense of gratitude and sincere regards to my supervisor Dr. Siddhartha S. Borkotoky. The friendly discussion helped immensely in selecting this topic and the generous encouragement throughout my dissertation work helped in continuing this project work. Sir has immensely assisted in providing all opportunities and facilities for the project work. I am thankful to sir for helping me in this work.

I am indebted to my parents who have inspired us to face all the challenges and win all the hurdles in life. Finally, I would like to thank all those who directly or indirectly guided me during my work.

Abstract

Fountain codes are record-breaking sparse-graph codes for channels with erasures, such as the internet, where files are transmitted in multiple small packets, each of which is either received without error or not received. Standard file transfer protocols simply chop a file up into K packet-sized pieces, then repeatedly transmit each packet until it is successfully received.

A back channel is required for the transmitter to find out which packets need re-transmitting. In contrast, fountain codes make packets that are random functions of the whole file. The transmitter sprays packets at the receiver without any knowledge of which packets are received. Once the receiver has received any N packets, where N is just slightly greater than the original file size K, the whole file can be recovered. LT codes are introduced, the first rate-less erasure codes that are very efficient as the data length grows.

Contents

List of Figures

Introduction

LT codes[1] are the first realization of a class of erasure codes that we call universal erasure codes. The symbol length for the codes can be arbitrary, from one-bit binary symbols to general l-bit symbols. We analyze the run time of the encoder and decoder in terms of symbol operations, where a symbol operation is either an exclusive-or of one symbol into another or a copy of one symbol to another.

0.1 Some encoding details

The process of generating an encoding symbol is conceptually very easy to describe:

- Randomly choose the degree d of the encoding symbol from a degree distribution. The design and analysis of a good degree distribution is a primary focus of the remainder of this work.
- Choose uniformly at random d distinct input symbols as neighbors of the encoding symbol.
- The value of the encoding symbol is the exclusive-or of the d neighbors.

Work done

Input:

```
ad = '/MATLAB Drive/O/L out/b.jpg';
% file path of the file
r = 2;       % the wanted redundancy or rep
SYC = 1;     % SYSTEMATIC LT Codes, T/F
VSE = 1;     % VERBOSE, increase output verbosity
P = 2^16;    % PACKET_SIZE
ro = 0.01;   % ROBUST_FAILURE_PROBABILITY
EPN = 1e-4;  % EPSILON = 0.0001
```

Figure 1: Input[2].

Encoding:

```
symbol = Symbol.empty(0,dq);
for i = 1:dq % i = symbol_index
    % Get the random selection, generated precedently
    % (for performance)

    % selection_indexes, deg = generate_indexes(i, r[i], n)
    [si, d] = geni(i, pr(i), nb, SYC); %

    % Xor each selected array within each other gives the
    % drop (or just take one block if there is only one
    % selected)
    drop = fb(si(1), :); % si(1)+ pf = f blocks
    for n = 2: d % bitwise_xor
        drop = bitxor(drop, fb(si(n),:)); % pf
        % drop = drop ^ blocks[selection_indexes[n]]
    end

    % Create symbol, then log the process
    symbol(i) = Symbol(i, d, drop); %i, d, drop
    symbol(i).log(nb, SYC);

    logo("Encoding", i, dq, EPN, P, dq) % , start_time
    %yield symbol
end
```

```
symbol 1, degree = 1     1
-- Encoding: 1/2 - 50.00% symbols at 1.63 MB/s
~0.08s
symbol 2, degree = 1     1
```

```
[~] = toc;
fprintf("\n---- Correctly dropped %d symbols " + ...
    "(packet size=%d)", dq, P);
```

```
---- Correctly dropped 2 symbols (packet size=65536)
```

Figure 2: Encoding[3].

Decoding

```
sbc = 0; % solved_blocks_count
isc = 0; % teration_solved_count
tic     % start_time
while isc > 0 || sbc == 0

    isc = 0;

    % Search for solvable symbols
    while 0<length(symbol) % symbol in enumerate(symbols)

        % Check the current degree. If it's 1 then we can
        % recover data
        if symbol(1).deg == 1 % i

            isc = isc + 1;
            bi = symbol(1).nes; % i block_index = next(iter
            syl = symbol(1);    % i
            symbol(1) = [];     % symbols.pop(i)

            % This symbol is redundant: another already
            % helped decoding the same block
            if bll(bi)      % is not None b, ~isnan(
                continue
            end
            bll(bi) = 1;
            bls(bi,:) = syl(1).data; % i

            if VSE
                fprintf("Solved block_%d with symbol_%d\n" ...
                    , bi, syl.ind);
            end
```

Figure 3: Decoding.

```
            % Update the count and log the processing
            sbc = sbc + 1;
            logo("Decoding", sbc, nb, EPN, P, ns)

            % Reduce the degrees of other symbols that
            % contains the solved block as neighbor.

            % reduce_neighbors()
            % Loop over the remaining symbols to find for a
            % common link between each symbol and the last
            % solved block.

            % To avoid increasing complexity and another for
            % loop, the neighbors are stored as dictionnary
            % which enable to directly delete the entry after
            % XORing back.

            for os = 1:length(symbol) % other_symbol
                if symbol(os).deg > 1 && ~isempty(find( ...
                        symbol(os).nes==bi, 1))

                    % XOR the data and remove the index from
                    % the neighbors
                    symbol(os).data = bitxor(bls(bi), ...
                        symbol(os).data)
                    symbol(os).nes(bi) = []; % .remove

                    symbol(os).deg = symbol(os).deg - 1

                    if VSE
                        fprintf("XOR block_%d with " + ...
                            "symbol_%d : %d", bi, ...
                            symbol(os).ind, symbol(os).nes);
                        % list( .keys()
                    end
                end
            end
        else
            symbol(1) = [];
        end % break here while testing
    end
end
```

Figure 4: Decoding continued.

```
b = bls';   b = b(:);
b = typecast(b(1:1:ceil(f/8)),'uint8');

fid = fopen(fcy, 'w','n',oen);
fwrite(fid,b(1:f)); % shrinked_data
fclose('all');

fprintf("Wrote %d bytes in %s", dir(fcy).bytes, fcy)
```

```
Wrote 181586 bytes in /MATLAB Drive/b-copy.jpg
```

Figure 5: Writing down the recovered blocks in a copy.

References

[1] M. Luby. “LT codes”. In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* 2002, pp. 271–280. DOI: `10.1109/SFCS.2002.1181950`.

[2] *Fountain Code: Matlab Implementation of LT Codes.* URL: `https://github.com/AnuragPaul0/LT-Codes`.

[3] *Efficient Python Implementation of LT Codes.* URL: `https://github.com/Spriteware/lt-codes-python`.