



Copy model and text generation

Algorithmic Information Theory

Pedro Figueiredo - 97487

Renato Dias - 98380

Index

| | |
|----------------------------------|----|
| 1. Introduction ----- | 2 |
| 2. Discussed approaches ----- | 4 |
| 3. cpm structure ----- | 6 |
| 3.1 Command line arguments ----- | 7 |
| 3.2 Hashmaps ----- | 8 |
| 3.3 Word data structure ----- | 9 |
| 3.4 Workflow ----- | 9 |
| 4. cpm_gen structure ----- | 8 |
| 4.1 Data preprocessing ----- | 7 |
| 4.2 Text generation ----- | 6 |
| 5. Results ----- | 11 |
| 6. Conclusion ----- | 11 |

1. Introduction

The objective of this work was to build 2 programs in C++, a copy model (cpm) and an automatic text generator (cpm_gen). The first one aims to read a file provide the estimated total number of bits for encoding a certain file, as well as the average number of bits per symbol. The second aims to generate text that follows the model implemented in the first program as a basis for text generation.

2. Discussed approaches

The first step towards designing a solution for the proposed project was to form a discussion around the approach we'd like to take.

We started by identifying a few of the characteristics we needed to have.

We started out by deciding the information we wanted to have associated with every word instance.

We decided on this attributes:

- Number of hits
- Number of fails
- Probability
- Number of consecutives fails
- Next symbol

We then analyzed 2 different design strategies: the first one, for every unique word in the hashmap, would associate an array of structures with the attributes given above (a new structure for every instance of the word that was followed by a different symbol).

Though this approach has some advantages in terms of compression, it demands more RAM to store all the structures being stored during the processing stage, so we chose to follow a different approach.

For every word, we will only store 1 structure only, with 1 symbol. This means the copy model will be able to operate with minimal RAM demands, allowing it to be used in devices of smaller computational skills.

3. cpm structure

3.1 Command line arguments

The program takes in 5 command line arguments (explained in order): the **name of the text file** to process, the **length of a word** to consider, a **threshold value**, α , used to calculate the probability associated to each word ($P(\text{hit}) \approx (N_h + \alpha)/(N_h + N_f + 2\alpha)$), a **minimum probability value**, and a **maximum number of consecutive failures**. These last 2 values serve as failsafes. When they are hit, the next symbol in the word structure is considered obsolete and replaced by a new symbol.

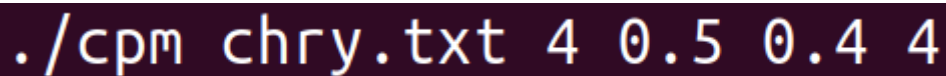


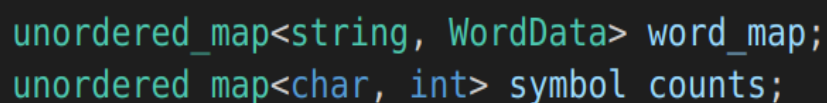
Fig.1 - Example of a command to run cpm file

```
if (argc != 6) {  
    cout << "Usage: " << argv[0] << " <filename> <word_length> <threshold> <min_prob> <max_num_fails>" << endl;  
    return 1;  
}
```

Fig.2 - Command line arguments

3.2 Hashmap

Then, the program initializes two hashmaps (or unordered maps): one to store information about each word, the other unordered map is used to store the count of each symbol in the sequence.



```
unordered_map<string, WordData> word_map;  
unordered_map<char, int> symbol_counts;
```

Fig.3 - Hashmaps initialization

3.3 Word data structure

This is the data structure that contains data about a word (number of hits, number of fails, estimated probability (calculated with the formula $P(\text{hit}) \approx (N_h + \alpha)/(N_h + N_f + 2\alpha)$), the next symbol that occurs after the word and the number of consecutive fails).

Every word in the file will have only one instance of this structure in the hashmap at a time.

Whenever a new reference to the word is found, we check if the nextSymb char corresponds to the symbol that follows it. We then update the remaining fields accordingly (incrementing hits or fails and num_consecutive_fails, depending on whether the symbol is correctly guessed), and update prob accordingly.

```
struct WordData {  
    int hits;  
    int fails;  
    double prob;  
    char nextSymb;  
    int num_consecutive_fails;  
};
```

Fig.4 - Structure 'WordData'

3.4 Workflow

The program iterates over the sequence of characters in the input file, taking each substring of a length defined as a command line argument. For each substring, the program checks if the substring is in the word map. If it is not, the program initializes a new entry in the map with a hit count and a fail count of 0, an estimated probability of 0.5 ($P(\text{hit}) \approx \alpha/2\alpha$), and the next symbol that occurs after the substring. If the substring is already in the map, the program updates the hit and fail counts based on whether the next symbol after the substring matches the expected next symbol.

The program then checks if the next symbol needs to be updated, by checking the number of consecutive failures or if the estimated probability is

below the minimum threshold. The program then updates the estimated probability of the word using the hit count, fail count, and smoothing factor, and updates the next symbol that occurs after the word.

```
string sequence;
file >> sequence;

int len = sequence.length();
for (int i = 0; i <= len - word_length; i++) {
    string word = sequence.substr(i, word_length);

    if (word_map.find(word) == word_map.end()) {
        // Word not found in map, initialize entry
        word_map[word] = {0, 0, 0.5, sequence[i + word_length]};
    } else {
        // Word found in map, update values
        if (sequence[i + word_length] == word_map[word].nextSymb) {
            word_map[word].hits++;
            word_map[word].num_consecutive_fails = 0;
        } else {
            word_map[word].fails++;
            word_map[word].num_consecutive_fails++;
        }

        if (word_map[word].num_consecutive_fails >= max_num_fails || word_map[word].prob < min_prob) {
            word_map[word].nextSymb = sequence[i + word_length];
            if (word_map[word].num_consecutive_fails >= max_num_fails) {
                word_map[word].num_consecutive_fails = 0;
            }
        } else {
            word_map[word].prob = static_cast<double>(word_map[word].hits + threshold) / (word_map[word].hits + word_map[word].fails + 2 * threshold);
            word_map[word].nextSymb = sequence[i + word_length];
        }
    }
}
```

Fig.5 - Hashmap update

After this, the program iterates over the entire sequence of characters and updates the count of each symbol in the sequence in the symbol counts map.

Finally, the program outputs the information stored in the word map: for each word in the map, the program outputs the hit count, fail count, estimated probability, next symbol, and number of consecutive failures. The program then calculates the total number of bits required to encode the sequence based on the formula $\sum(-\log_2(P(\text{word})))$ for estimated probabilities of each word in the map, and outputs the estimated total number of bits and the average number of bits per symbol.

```
double total_bits = 0.0;

for (auto it = word_map.begin(); it != word_map.end(); it++) {
    total_bits += -log2(it->second.prob);
}

double avg_bits_per_symbol = total_bits / total_symbols;
```

Fig.6 - Total number of bits and average number of bits per symbol calculation

4. cpm_gen structure

4.1 Data preprocessing

For the cpm_gen program, we do the same data preprocessing as we do for cpm program, obtaining a list of words, following symbols and probabilities that we'll put to use in text generation.

4.2 Text generation

The information above can be used to generate text. The algorithm implemented works as follows:

- Choose a starting word at random from the words in the input text;
- Output the chosen word;
- Use the expected character associated with the chosen word to generate the next character;
- Find all words in the input text that start with the last (word_length - 1) characters of the previous word, and select the one with the highest probability of generating the next character;
- Repeat steps 2-4 until the desired length of text has been generated.

```
// Generate text
srand(time(0)); // seed random number generator
string seed = sequence.substr(0, word_length);
string generated_text = seed;
vector<string> possible_words;
for (int i = 0; i < max_length - word_length; i++) {
    string curr_word = generated_text.substr(i, word_length);
    for (auto it = word_map.begin(); it != word_map.end(); it++) {
        if (it->first.substr(0, word_length - 1) == curr_word.substr(1)) {
            possible_words.push_back(it->first);
        }
    }
    int randomIndex = rand() % possible_words.size();
    generated_text += possible_words[randomIndex];
    possible_words.clear();
}

cout << "Generated text:" << endl;
cout << generated_text << endl;

return 0;
```

Fig.7 - Text generation algorithm

5. Results

When we run the cpm file with the command in the figure 1, for example, we obtain the following data:

```
List of words and their associated data:
TCCG: hits = 3104, fails = 6438, prob = 0.25, nextSymb = C, num_consecutive_fails = 0
ACGT: hits = 4452, fails = 11193, prob = 0.25, nextSymb = C, num_consecutive_fails = 2
CTTA: hits = 24113, fails = 63261, prob = 0.375, nextSymb = G, num_consecutive_fails = 2
CGAT: hits = 3415, fails = 6975, prob = 0.25, nextSymb = G, num_consecutive_fails = 0
AACG: hits = 3715, fails = 8506, prob = 0.25, nextSymb = G, num_consecutive_fails = 0
TTAC: hits = 31533, fails = 55394, prob = 0.25, nextSymb = T, num_consecutive_fails = 0
TGCG: hits = 2678, fails = 6893, prob = 0.25, nextSymb = T, num_consecutive_fails = 0
CGCC: hits = 3239, fails = 7260, prob = 0.25, nextSymb = T, num_consecutive_fails = 1
TCGC: hits = 2442, fails = 5133, prob = 0.25, nextSymb = C, num_consecutive_fails = 1
GTTG: hits = 20587, fails = 53209, prob = 0.375, nextSymb = A, num_consecutive_fails = 0
GTGC: hits = 19372, fails = 38462, prob = 0.25, nextSymb = A, num_consecutive_fails = 1
GCAG: hits = 25970, fails = 68568, prob = 0.25, nextSymb = C, num_consecutive_fails = 0
CTCG: hits = 3386, fails = 8609, prob = 0.386364, nextSymb = T, num_consecutive_fails = 1
TATA: hits = 50526, fails = 95883, prob = 0.25, nextSymb = A, num_consecutive_fails = 1
GCGT: hits = 2868, fails = 5753, prob = 0.39916, nextSymb = A, num_consecutive_fails = 0
GACC: hits = 16579, fails = 29354, prob = 0.25, nextSymb = C, num_consecutive_fails = 0
GATC: hits = 17653, fails = 33460, prob = 0.25, nextSymb = T, num_consecutive_fails = 0
TGGT: hits = 28772, fails = 71062, prob = 0.397321, nextSymb = T, num_consecutive_fails = 0
CAAC: hits = 25681, fails = 45363, prob = 0.388889, nextSymb = A, num_consecutive_fails = 2
CGAA: hits = 3705, fails = 7591, prob = 0.375, nextSymb = T, num_consecutive_fails = 0
TCGA: hits = 4488, fails = 7855, prob = 0.25, nextSymb = G, num_consecutive_fails = 1
ATTG: hits = 32120, fails = 78605, prob = 0.361111, nextSymb = G, num_consecutive_fails = 0
TACC: hits = 19937, fails = 37396, prob = 0.25, nextSymb = T, num_consecutive_fails = 2
GGAT: hits = 20364, fails = 53522, prob = 0.25, nextSymb = G, num_consecutive_fails = 0
CGGG: hits = 3197, fails = 8638, prob = 0.397959, nextSymb = A, num_consecutive_fails = 0
TGGG: hits = 28930, fails = 75305, prob = 0.25, nextSymb = A, num_consecutive_fails = 0
CCCA: hits = 29861, fails = 73848, prob = 0.25, nextSymb = G, num_consecutive_fails = 2
ATCC: hits = 25276, fails = 48271, prob = 0.25, nextSymb = T, num_consecutive_fails = 2
TGCC: hits = 28342, fails = 53651, prob = 0.25, nextSymb = A, num_consecutive_fails = 0

Estimated total number of bits: 464.334
Average number of bits per symbol: 2.04839e-05
```

Fig.8 - cpm results for command '/cpm chry.txt 4 0.5 0.4 3'

When we run the cpm_gen file with the command "/cpm_gen chry.txt 4 0.5 0.4 3 1000" (the command line arguments are the same except the last one that represents the maximum length of the generated text), for example, we obtain the following data:

Generated text:

CAATTAATAATAAAAAATGATTTAAAAATGATAATAATCATATTAAAGAAAAAATAAAAAATCATGTGGACGATCATCATCTATTATTACAAAGAAATAAAAAAACAAATGATGATGAAGATCATTTAATAATAATCAAAAAAT
ATAATAAAAAATGATCTCCACAGATGATGATTATTGTTAGTAAGAAGCAGGAAAAAATAAAAAATGAAAAAACAAATAATCATGATGAAAAAACAAACAAAGATGATCATGACAGGATGTGGCGGTTGTGTGACGACGAGCGTCGGA
TATACCTCCACGATTTACTACACATCTCTAGTATCTCTTCCAAATATTTTGTAGTAATTTTGTATACCACTACTTATTAATAAGCAGATGAATAAAAAAGATGATTAAAGAAAAATAAAAAAACCAATCAACCAAGGAATTAT
TGAGGATCATTTGTGCGAGTTATGTGCGAAGAAAAATGAGAATATCTCCAGCATATCATGATGTTGTAATAAGCATGATGATTAATAATAAAAAATCATTTACAACTCCTTATTAATAAGAAAAAATAATAATTTTGTATATA
ATCATTAATCATATTAAAAACAAAAAATAATTTATGATGATGATGTCCTCAACTCTCTCCAGCAACCACTACAGATTTGCTGATGATCATATAAGCGTTGTATTATTTATTTATTTCTGCTGTCTGTTTTATATAT
AGCTGGTAATAATAAAGCAAAAAAGAACGTCGACAGTAGAGGAGTGGTGGATGAAGAAATAAAGAAAGAAATAATAAAGAAACAAATAATCATATAAATAAAGAAAAAGAAACAAACCAACTACCGACAGAAACAAATAAACAACAA
ATAATCTCCACCATTTACTAGAAGTTGTTATGAAGAATAAAGAAAGAAACAAAGAAATAAAGAAAGAACCATCAACAAAGAACTCATCAATAAAGAACGATGAACCATCTGCGAGGATGTCTCCCAACTCTCGATCGTCAACG
ACGACCAACATATTTATTTATGATGTTGGGCTGTGGTGTGGAGCAGCAGCGGTGGTCTGTTGATGTTGTGGTGGAGGATATGATGAGGACAACTGGCGAGGACCGCGGAGACCGCGGCTCGCTCGACGAGGAAGATAGAG
GATTTACTAGAATATCTCTCCGCTCTGTTCTCCAGCAAGCCGAGCATGATTTCTAGTATAATGACTACTGCTAATACGACATCAAAAATCTACTCTCTGCTCTTACGAGTAGTAAATGATGCTGATCATATCTCTCTG
CTCTGTTCTCCAGCAATCAACACATCATTTTATAATAATTTCTGTTCTGTTTGTATAGCACTGTATTATTAATAATTTATTTCTGTTGTTGTTCTGATAGGCTGTGATCACCCTCCCAACTCTCTGATGACCTCTCTGAT
CTTATCATTTATTAATAATAATGATACATAAGGAGGAGCGGCGACAGAGACGATGATTTGGAGGAATAATTATGATAAAAAACAAAAAACAACTCATGTTGGAGATATAAGTATGATTTTTTAACTAATAAGGACAGAGAAAAAAT
AAATAAAGAAATATGATATAATAAAAAACAAATATAAAAAAATGATAGCAAGAAAAATCATATAAAAAACCTCCACACAAACACAGCCGACCCCAAGCAATAAAGAAATGAAAAAATTTTTTGTATATAATGATGAAGGTCGGTGG
TGAGGAGAGGAGGATGATTTCTCTTCTCATCATCTATGATGATGTTCTGTCGACACAGCTCGGAAGATTTTTTAAATAATTTATTTGTCGGCTCTGTTGGAGGACAGAAGATGAAGAAAAAATAAAAAAGAGCTAGTATGACGAGAAGAT
TACGACAGCGATCATAGTAAAGATATCTTCCACTCTTCATCAGCAGCGCCGACCAATAATAAATCAAAATATATGACGACTCTTAATAATAATGGTATGATGATTTATTTGTTCTTAATAAAAAATCATTAAGAACACAGCAAGA
ATAATGTTGGAGGATATAATAAGAAGTCCTAAAAATCATAGTAAGAAATATTAATAAATAATGAAAAAATAATAAAGAAAAAAGAAATAGCTCCGAGTATGATTATATTTTTATAACAACTATAAATCAACCACTGCTCG
TCCCCCTCTCTACTATATTTATTAAGAAATATATCAACAAAAAGAGACGAATAAAGAGGAGGAGAAAAAATAATGATAAATAAAATGATTAATAAAATTTGTTCTTCTTATAGGAGCTGATCATCAACATAAAAAAT
TATTAATAGCACTTTATTAACATGATACAAAGATTTCTTCCAGCATATAGTATGATTATTAATAAGAAACAAACAAATATAAATAATGAAAAATCAACAAATAAACAAGAAATCATTTCTTATATAATTTGGATG
CATTTGATGATGATCATGATGAGGATCATGTGGAGGATTTATCTCGAGCTCTGATCTCCCCCAACCAACCTCTGCTCTTCCACTCTCTGCTACTGTTCTGCTGACGACAGCGGGCATCAAGAACACCAACAGAACCACTTTCTTACG
ACAGCAAGCAGCAGGACCATCTTCTTGTATAATCTCGACGCTGGTGGTGAAGATATGTCGAAGATAAATCATCATGATTAATAATAATCATATTAGCGCGCCGACGAGCGCTGCTTGTATGCTGATCATTTGTTATCATTTAT
TACTATGATTTCTTATTTATCATTTATACGACCTCTAATATTTCTTATTTATTTGTTCTCTTATGATGCCGCTGTTGTTGCTGCTGCTGTTGTTCTTCTCTCTGATGATGTTCTTCTTACGAGGCGGCTCAGG
AGTACTGCTGGTGGTGGTTTAAAAATGATACATAAAATCATATAAAGAAATAACTATAAAGAAACAGGAGGATGAATAAAGAAACAAAAAATAAGAAATGAAAAAGTACGGGCTCATGTGACGACGACGCGTGCACAGCGCT
GTAATATGATGAAGAGAGGAGGCGGAGAGGATGTTGATGATAAGGGGGGTAGTATGATGGGCGACATTTATGATGATCAACATGACGAAAAAATAAATAATAAAGAAAAAGAGGAGAAACGACGACGAGAAAGAAATCATATCAAG
ATAATAATGATCATTAACAAAGAGAGGATGAAAAAAACAAACCAACCAAAATAAATAGTAAGAAATATCTGACCAATGATACAAACATGATGACAAAGAAAGAAAGAAATAGACGAGGAATATAAGAAAGAAATATAAAGAT
AAGAAAGCAAAAAACACACCCCAAGATAAAGCAGACCAACCAAGCTACTGACTACTCAGCAGGACCGGACAGGAAGAGGAGCAATTAATAACAACATCATATAAATAATGATATAAAGAAAGACCAACCAATAAAAATCATATAA
AATGATGATGAAACATGATCATCCCCCTCTCTCTCAACCAACCCGACCCACTGTTTGTACTATAATAGCTCTCTACAGGACAGCAACGATGCTGTTTGTGTTGATGATCTGTTTTTAAAGCAAGGAGGAGGAAATAAAAAAT
TATGATTAATAAACAAGGAGAAAAAATAAAGAAAAAACAACAAAGAACCATCAACAAAGAGGAGATGAAGAAATATCATTTTAAAGAAACAAAGAAAGAAACAAACAAAGGAGATGAATAACGACCTCCGACATCATCATATACTTCA
CAACACATCATCATCAACAAAGAGAGATGAAGAACCG

Fig.9 - cpm results for command '/cpm_gen chry.txt 4 0.5 0.4 3 1000'

```
Estimated total number of bits: 7412.65
Average number of bits per symbol: 0.000327006
```

Fig.10 - cpm results for command '/cpm chry.txt 6 0.5 0.4 3'

```
Estimated total number of bits: 399.538
Average number of bits per symbol: 1.76255e-05
```

Fig.11 - cpm results for command '/cpm chry.txt 4 1 0.4 3'

```
Estimated total number of bits: 572.25
Average number of bits per symbol: 2.52446e-05
```

Fig.12 - cpm results for command '/cpm chry.txt 4 0.5 0.2 3'

```
Estimated total number of bits: 464.362
Average number of bits per symbol: 2.04852e-05
```

Fig.13 - cpm results for command '/cpm chry.txt 4 0.5 0.4 4'

Figures 8,10,11,12 e 13 display the compression results for different entry values (Figure 8 serves as the comparison for all the others).

When comparing them we can conclude the following:

- When comparing copy models with **word sizes as the only difference (Fig 10)**, we notice that a larger size correlates to a larger number of bits being used, both in total and per symbol. This can be attributed to the fact that longer words create a more diverse set of words, reducing the possibility of repetition in the text.
- As for the impact that the **threshold (Fig 11)** has, we feel it's pretty minute, as it's impact is immediately smoothed by the hit probability formula.
- In regards to the **minimum probability value(Fig 12)**, we were surprised by the small impact it had on repeated runs. However, we attribute the small impact of it's alterations on the final result to the fact the minimum number of repeated fails provides a strong mitigating force.
- In the **minimum number of repeated fails case (Fig 13)**, we also noted limited impact, likely due to the mitigating factor of minimum probability.

6. Conclusion

In discussing our approach to the copy model, there were 2 attributes we focused on: compression quality and RAM usage.

We looked into finding a compromise between both, and believe our program is capable of achieving a solid compromise on both fronts.

The fact that we only store one word structure in *word_map* for any instance of a word means our copy model has pretty efficient RAM usage, while still being able to have a good level of compression.

There are some drawbacks however, as the compression capability cannot be fully maximized. If we kept all instances of a word followed by a different symbol, we would likely achieve higher levels of compression, but this put a much higher strain on RAM, which we chose to not do.

In regards to the text generator, we focused on allowing it to reproduce the patterns of the text it derives from.

To achieve those goals, we take the words that start with a given symbol and perform sampling of these to build a text, guaranteeing that there will be similarities in structure.

Some improvements could have been made on its efficiency, but they had to be sacrificed for time constraints.