Lempel-Ziv-78 and Lempel-Ziv-Welch

Devid Duma

Abstract

Simple, yet efficient Lempel-Ziv-78 and Lempel-Ziv-Welch algorithms. ID numbers of lookup tables are saved in base 128 as characters. The first bit of each character denotes whether that character is the last one representing the ID number or not.

Keywords – Lempel-Ziv-78, Lempel-Ziv-Welch

1 Introduction

C++17 I make use of C++17 for it's superior IO streams and buffers.

Dataset I am going to use the Pizza&Chili Corpus database as a reference for our DNA sequence. I used the sequence called *"dna.50MB"*, the 50MB version of the larger 300MB sequence. This proved to be more than enough for our needs. The download link is provided here http://pizzachili.dcc.uchile.cl/texts/dna/

2 Compression Algorithms

2.1 Lempel-Ziv-78

Implementation Let us have a look at Lempel-Ziv-78 compression algorithm.

C++17 code

```
class LZ78 {
private:

public:
    map<string, long> table;
    vector<tuple<long, char>> compressed_data;

    LZ78() {
    }

    void compress(const string& data) {

        long pos = 0;
        while(pos < data.length()) {

            // find if string exists in dictionary
            long length = 0;
            long id = 0;

            while(pos + length + 1 < data.length() && this->table.find(data.substr(pos, length + 1)) != this->table.end()) {
                length++;
                id = this->table[data.substr(pos, length)] + 1;
            }

            // save next substring to table
            this->table[data.substr(pos, length + 1)] = this->table.size();

            // move pos to += length + 1
            pos += length + 1;
            char next_char;
            // next_char of tuple <id, next_char> is at new position
            if(pos > data.length()) {
                next_char = NULL;
            } else {
                next_char = data[pos - 1];
            }

            this->compressed_data.emplace_back(tuple(id, next_char));
        }
    }

    static string writeSegment(long x) {
        // I chose modulo 127 rather than 128, to avoid char(-1) == EOF
        int base = 128;
        string x_base;

        if(x > 0) {
            while(x > 0) {
                x_base += char(x % base);
                x = x / base;
            }
        } else {
            x_base = char(0);
        }
        // Flip range [0 ; 127] to [-1 ; -128]
        x_base[x_base.length() - 1] = (x_base[x_base.length() - 1] + 1) * (-1);

        return x_base;
    }

    static string writePair(long id, char next_char) {
        string id_base = writeSegment(id);

        string result;
        if(next_char != NULL) {
            result = id_base + next_char;
        } else {
            result = id_base;
        }

        return result;
    }

    string printCompressed() {
        string compressed_data_string;

        for(tuple pair : this->compressed_data) {
            compressed_data_string += LZ78::writePair(get<0>(pair), get<1>(pair));
        }

        return compressed_data_string;
    }
};
```

2.2 Lempel-Ziv-Welch

Implementation Let us have a look at Lempel-Ziv-Welch compression algorithm.

C++17 code

```
class LZW {
private:

public:
    map<string, long> table;
    vector<long> compressed_data;

    LZW() {
        // initialize lookup table
        for(int i = 0; i < 256; i++) {
            string charToString;
            charToString += char(i);
            table.insert({charToString, i});
        }
    }

    void compress(const string& data) {

        long pos = 0;
        while(pos < data.length()) {

            // find if string exists in dictionary
            long length = 0;
            long id = 0;

            while(pos + length < data.length() && this->table.find(data.substr(pos, length + 1)) != this->table.end()) {
                length++;
                id = this->table[data.substr(pos, length)];
            }

            // save next substring to table
            this->table[data.substr(pos, length + 1)] = this->table.size();
            // cout << id << " " << data.substr(pos, length + 1) << endl;

            // move pos to += length
            pos += length;

            // emplace back id
            this->compressed_data.emplace_back(id);
        }
    }

    static string writeSegment(long x) {
        int base = 128;
        string x_base;

        if(x > 0) {
            while(x > 0) {
                x_base += char(x % base);
                x = x / base;
            }
        } else {
            x_base = char(0);
        }
        // Flip range [0 ; 127] to [-1 ; -128]
        x_base[x_base.length() - 1] = (x_base[x_base.length() - 1] + 1) * (-1);

        return x_base;
    }

    static string writePair(long id) {
        string result = writeSegment(id);

        return result;
    }

    string printCompressed() {
        string compressed_data_string;

        for(long id : this->compressed_data) {
            compressed_data_string += LZW::writePair(id);
        }

        return compressed_data_string;
    }
};
```

2.3 Output Format

Lempel-Ziv-78 The output format is *id* and *next_character*, which are printed sequentially without any delimiting character between them. There is also no delimiting character between tuples.

If the *id* is 0, then the *id* **will** be printed.

Lempel-Ziv-Welch The output format is *id* only. IDs are printed sequentially without any delimiting character between them.

If the *id* is 0, then the *id* **will** be printed.

2.4 Encoding numbers as characters

Encoding numbers as characters When printing out numbers with base 10 in the output file, we are wasting a lot of bits.

Numbers are outputted with base 128, so 2^7. Characters in files are encoded with 8 bits, and 7 of those bits are now used to save the number. If 7 bits are not enough, the next octets of bits (characters) will contain the remaining information of the number.

The remaining bit in each octet (the first bit of the character) is used to denote if that character is the last character representing that number. If it is the last character, it's first bit will have value 1 and 0 otherwise.

3 Complexity

The Lempel-Ziv complexity was first presented in the article *On the Complexity of Finite Sequences (IEEE Trans. On IT-22,1 1976)*[2]. This complexity measure is related to Kolmogorov complexity, but the only function it uses is the recursive copy (i.e., the shallow copy).

The underlying mechanism in this complexity measure is the starting point for some algorithms for lossless data compression, like LZ77, LZ78 and LZW. Even though it is based on an elementary principle of words copying, this complexity measure is not too restrictive in the sense that it satisfies the main qualities expected by such a measure: sequences with a certain regularity do not have a too large complexity, and the complexity grows as the sequence grows in length and irregularity.

The Lempel-Ziv complexity can be used to measure the repetitiveness of binary sequences and text, like song lyrics or prose. Fractal dimension estimates of real-world data have also been shown to correlate with Lempel-Ziv complexity [3][4].

4 Decompression Algorithms

4.1 Lempel-Ziv-78 Decompression

Implementation The decompression algorithm for Lempel-Ziv-78 is shown below.

C++17 code

```
1
2 class LZ78Dec {
3 private:
4     string data;
5
6 public:
7     vector<string> table;
8     int counter = 0;
9
10    explicit LZ78Dec(string data) {
11        this->data = std::move(data);
12    }
13
14    string decompress() {
15        string decompressed_data;
16
17        while(!this->data.empty()) {
18            tuple pair = readPairFromData();
19            long id = get<0>(pair);
20            char next_char = get<1>(pair);
21
22            // if id = 0, then no need for table lookup
23            if(id > 0) {
24                decompressed_data += this->table[id - 1];
25                // add to table
26                this->table.emplace_back(this->table[id - 1]↩
    + next_char);
27            } else {
28                // add to table
29                this->table.emplace_back(string(1, next_char)↩
    );
30            }
31
32            if(next_char != NULL) {
33                decompressed_data += next_char;
34            }
35        }
36
37        return decompressed_data;
38    }
39
40    tuple<long, char> readPairFromData() {
41        long base = 128;
42
43        long pos = 0;
44
45        long id = 0;
46        long power_up = 1;
47        while(pos < this->data.length()) {
48            if(this->data[pos] < 0) {
49                id += (this->data[pos] + 1) * (-1) * ↩
    power_up;
50                break;
51            } else {
52                id += (this->data[pos]) * power_up;
53            }
54
55            power_up *= base;
56            pos += 1;
57        }
58        pos += 1;
59
60        char next_char;
61        // next_char and remove tuple
62        if(pos < this->data.length()) {
63            next_char = this->data[pos];
64            this->data = string(this->data.begin() + pos ↩
    + 1, this->data.end());
65        } else {
66            next_char = NULL;
67            this->data = "";
68        }
69
70        this->counter++;
71        return tuple(id, next_char);
72    }
73 };
```

4.2 Lempel-Ziv-Welch Decompression

Implementation The decompression algorithm for Lempel-Ziv-Welch is shown below.

C++17 code

```
1
2 class LZWDec {
3 private:
4     string data;
5
6 public:
7     vector<string> table;
8     int counter = 0;
9
10    explicit LZWDec(string data) {
11        this->data = std::move(data);
12
13        // initialize lookup table
14        for(int i = 0; i < 256; i++) {
15            string charToString;
16            charToString += char(i);
17            this->table.emplace_back(charToString);
18        }
19    }
20
21    string decompress() {
22        string decompressed_data;
23        string old;
24
25        while(!this->data.empty()) {
26            // read id from data
27            long id = readIdFromData();
28
29            string neu;
30
31            // watch out for the case when id points to the ↩
    not yet written old id
32            if(id == this->table.size()) {
33                neu = old + old[0];
34            } else {
35                neu = this->table[id];
36            }
37
38            // if first character read, then no need to add to ↩
    table
39            if(!old.empty()) {
40                // add to table
41                this->table.emplace_back(old + neu[0]);
42            }
43
44            // cout << id << " " << this->table.back() ↩
    << " " << neu << endl;
45
46            // add sequence to decompressed_data
47            decompressed_data += this->table[id];
48
49            // update old tuple
50            old = neu;
51        }
52
53        return decompressed_data;
54    }
55
56    long readIdFromData() {
57        long base = 128;
58
59        long pos = 0;
60
61        // build up id from characters
62        long id = 0;
63        long power_up = 1;
64        while(pos < this->data.length()) {
```

```
65            if(this->data[pos] < 0) {
66                id += (this->data[pos] + 1) * (-1) * power_up;
67                break;
68            } else {
69                id += (this->data[pos]) * power_up;
70            }
71
72            power_up *= base;
73            pos += 1;
74        }
75
76        // remove id
77        if(pos < this->data.length()) {
78            this->data = string(this->data.begin() + pos + 1, this->data.end());
79        } else {
80            this->data = "";
81        }
82
83        this->counter++;
84        /*
85        if(id > this->table.size())
86            cout << endl << "ID is bigger than table size: " << data.length() << " " << pos << " " << id << " " << this->table.size() << endl;
87        */
88        return id;
89    }
90 };
```

Description The decompression algorithm follows the same logic used in the compression algorithm regarding numbers. Since the first bit of a character denotes the end of the number, decompression is possible.

Correctness The compression and then decompression of some input DNA sequence always results in the original sequence. The correctness of the compression and decompression algorithm is therefore assured.

5 Experiment and Results

Compression ratios tested on 50MB DNA sequences.

Lempel-Ziv-78 achieved circa 230% compression ratio.

Lempel-Ziv-Welch achieved circa 300% compression ratio.

Remarks The larger the filesize of DNA sequencing, the larger the compression ratio.

The compressed files now have a wider alphabet than before. The probability distribution of characters should be more uniform.

The Huffman encoding efficiency should also increase, although I was not able to showcase it because of compatibility issues when reading files.

6 Conclusion

Compression ratios tested on 50MB DNA sequences for Lempel-Ziv-78 is 230% and compression ratio for Lempel-Ziv-Welch is 300%. Implementation in C++ is crucial, for making use of it's superior IO streams and buffers.

References

[1] J. Ziv, A. Lempel, *Compression of individual sequences via variable-rate coding*. 10.1109/TIT.1978.1055934.

[2] J. Ziv, A. Lempel, *On the Complexity of Finite Sequences*. 10.1109/TIT.1976.1055501.

[3] Thomas Burns, Ramesh Rajan, *Combining complexity measures of EEG data: multiplying measures reveal previously hidden information*. 10.12688/f1000research.6590.1.

[4] Thomas Burns, Ramesh Rajan, *A Mathematical Approach to Correlating Objective Spectro-Temporal Features of Non-linguistic Sounds With Their Subjective Perceptions in Humans*. 10.3389/fnins.2019.00794.