MAT357 Project 3

Adam De Broeck

January 4, 2024

The Compression Algorithm

The main design of this compression algorithm, unlike many traditional approaches which use function transformations, is to strategically remove and replace parts of a file's linguistic data. More aptly put, this algorithm serves to compress text data by taking advantage of the patterns in human language — specifically English. By removing deterministically select words from a piece of literature, one can reduce a file's size during encoding — so long as there is a suitable way to reconstruct the text when being decoded.

The advantage of selectively removing parts of text from a piece of literature is that the text follows sentence structure and is highly likely to have explicit repetitive patterns in it. In order to decode a piece of literature that has been encoded in this manner (destructive), this algorithm uses n-grams as a form of pattern recognition and prediction. Due to the nature of this algorithm, the data after encoding will be extremely lossy, albeit still readable (to some degree). For future reference, it is assumed that all textual data used in this algorithm uses plain ASCII characters with ANSI encoding, although the compression rate is mostly theoretical anyway based on character lengths.

As a general overview, the typical flow of data compression for this algorithm is like so: Two text files are presented, the text file to be encoded, and the text file to use as a key. First, a 64-bit seed is generated based on the first 4 characters present in the key file — this is to make sure the results of the encoding are deterministic based on the key file used. Next, the text file to be encoded is scanned and every word is pseudo-randomly removed from the file and replaced with a vertical bar character `"|"`. This removal is based on the pre-processor definition that specifies the compression rate `#define COMPRESSION_RATE 0.5`. This value is ranged from `0.0f` to `1.0f` which represents the compression percentage. After the file has

been scanned and words have been replaced, the encoding portion of this algorithm is complete.

In order to decode the encoded text file, the key file is first opened and parsed into an array of tokens (these tokens can be letters, words, or groups — more on that later). The array is then traversed and buckets of tri-grams, bi-grams, and uni-grams are created alongside the following token found. Every iteration of each respective pattern adds to the counter for the specific n-gram in order to build the database which represents the key-file. Once the n-gram database has been built, the encoded file is traversed and each representation of the vertical bar symbol `"|"` is replaced with a generated word based on the historical data surrounding the symbol to be replaced. The historical data is compared against the n-gram database using a weighted lookup table which determines the next token to be generated based on the frequency of an occurrence of a given n-gram.

The vertical bar was chosen to keep compressed file sizes down to minimum as only "printable" characters are represented with a single byte for the given file encoding. Between the rest of the printable characters, the vertical bar `"|"` and at `"@"` symbol were the least likely to be used in plain text. Additionally, all code done in this project was written in `C++`. Furthermore, all random values were deterministically calculated with pre-determined seeds and a Mersenne Twister implementation from `C++`'s `STL` (MT19937).

An Overview of N-Grams

As a quick re-cap, n-grams are typically short sequences of tokens which can represent patterns or historical data. While n-grams have applied usages beyond this in the fields of computer science and probability / set theory in mathematics, these topics are outside the scope of this project. This compression algorithm will only use the basic concepts of n-grams in order to predict/generate missing words.

The tokens being used in this project vary, however, the two most common tokens being used are words and letters. For this project, `tri-grams (3-grams)`, `bi-grams (2-grams)`, and `uni-grams (1-grams)` were used to generate the n-gram database. An example of what a basic set of **letter** tri-grams might look like (ignoring punctuation) is as follows:

```
"The quick brown fox"

The, heq, equ, . . . , nfo, fox
```

Additionally, when using words as tokens, they can typically be broken up as follows:

```
"The quick brown fox"
```

```
The quick, quick brown, brown fox
```

When parsing for n-grams in the decoding process, 3 ”buckets” are formed using the tri-gram, bi-gram and uni-gram principles for each bucket. Doing so allows matching more complex patterns as a priority before matching patterns that have less complexity. The idea, is that the probability of complex patterns matching is low, but complex patterns are more likely to have the contextual evidence necessary to complete the missing token in way that makes sense for the given context.

In code, the representation of the n-gram lookup tables was a hash map of grams and hash maps of strings and integers (what a mouthful!). An example of that code is as follows:

```
struct TriGram
{
    std::string third;
    std::string second;
    std::string first;

    bool operator==(const TriGram &rhs) const
    {
        return (third == rhs.third) &&
               (second == rhs.second) &&
               (first == rhs.first);
    }
};

// Now organize the words into Ngrams
std::unordered_map<TriGram, std::unordered_map<std::string, int>> triGrams;
std::unordered_map<BiGram, std::unordered_map<std::string, int>> biGrams;
std::unordered_map<UniGram, std::unordered_map<std::string, int>> uniGrams;

size = keywords.size();
std::string currentWord;

. . .
```

Weighted Lookup Table Implementation

Another crucial component needed to make n-grams useful as a tool for regenerating missing text is weighted lookup tables. Weighted lookup tables in computer science functionally act as an effective means of simulating weighted probabilities without extra effort. In this case, a weighted lookup table is similar to an average probability question in a discrete mathematics course — e.g., what are the odds that a blue marble will be selected from a bag of 3 green marbles, 6 blue marbles, and 11 white marbles? Of course, in this case it's easy to see that the total number of marbles is $3 + 6 + 11 = 20$, so the odds of getting a blue marble are going to be $\frac{6}{20}$. However, what if the number of marbles for each changes? Or a marble needs to be selected at random instead of iterating over them and selecting the probability of each one by one? This is where a weighted lookup table is useful.

Assume there's a list of selections (key) and a weights (value) as key + value pairs. That might look like so:

Selection	Weight
quick	3
brown	1
fox	2

To get a random selection while taking its weight into consideration, the total weights in the entire list can be added together and a random value between 0 and that total (exclusive) can be taken. Then, by iterating through each selection and removing that selection's value the correct selection can be obtained by checking if the index is less than 0; The reason this works — and by extension, is uniformly fair — isn't immediately obvious. Also, wouldn't the order of the selection items change the outcome?

It becomes apparent why this works if the weights are thought of like equally sized spaces/squares. In the previous example, this means that the `quick` selection takes up 3 squares, `brown` takes 1, and `fox` takes 2. If a ball were to land, uniformly and randomly, in one of these 6 squares it would have a fair probability of $\frac{3}{6}$, $\frac{1}{6}$, and $\frac{2}{6}$ respectively. Changing the positions of the squares wouldn't change the probability of their outcome provided the ball had not yet been tossed. The same thing can be said about the marbles in a bag situation — moving the marbles around in the bag doesn't change their probability of being grabbed.

In the case of the lookup table, the position that the "ball drops" is already determined, the index subtraction is just to arrive at the selection

efficiently. As a final example, here's what the previous table would look like if a random value of 3 was selected:

Selection	Weight
quick	3
brown	1
fox	2

$$3 - 3 = 0 \rightarrow 0 \not< 0$$
$$0 - 1 = -1 \rightarrow -1 < 0$$
$$\hookrightarrow \texttt{brown}$$

In this case, brown can be seen as the selected index for the value 3 in this weighted lookup table. The probabilities of each possible starting value can also be calculated by making a truth table for each possible starting value.

Start Value	Selection
0	quick
1	quick
2	quick
3	brown
4	fox
5	fox

And here's a code example of a weighted table being populated:

```
// Store possible trigrams
if(!thirdLastWord.empty() &&
   !secondLastWord.empty() &&
   !firstLastWord.empty() &&
   !currentWord.empty())
{
    TriGram triGram;
    triGram.third = thirdLastWord;
    triGram.second = secondLastWord;
    triGram.first = firstLastWord;

    // Find the gram in the map and store its future word if found.
    // Otherwise, make a new entry for the future word / map key
    if(triGrams.find(triGram) != triGrams.end())
```

```
    {
        std::unordered_map<std::string, int> futureGrams = triGrams.at(triGram);
        if(futureGrams.find(currentWord) != futureGrams.end())
        {
            ++futureGrams.at(currentWord); // Future word found, incrememt
        }else
        {
            // No future word, make and insert 1
            futureGrams.insert(std::make_pair(currentWord, 1));
        }
        triGrams.at(triGram) = futureGrams;
    }else
    {
        // No map entry, make entry + future word and insert 1
        std::unordered_map<std::string, int> futureGrams;
        futureGrams.insert(std::make_pair(currentWord, 1));
        triGrams.insert(std::make_pair(triGram, futureGrams));
    }
}
```

Letter Based Compression Implementation

Now that the tech used is out of the way, it's time for the implementation details of the algorithm. The first attempt at this algorithm was to use letter sized tokens and delete portions of words which could then be reconstructed with n-grams later. Of course, since the deleted portions of the text needed some marker / indicator showing where a part of the text was deleted, the minimum number of consecutive letters that could be deleted would have to be 2 otherwise the file size wouldn't change (technically if the file was written to as a raw binary file this isn't true, however, for the purposes of this project only ANSI encoded text files were used).

Initial results were a little underwhelming. As a general note, all examples in this project unless otherwise stated used a compression rate of `0.5` which means `50%` of the original text was thrown away. The biggest problem with this approach was the contextual fluidity of the letters being added. Because the history of letters being used took into consideration only the letters and no punctuation, letters could generate using history that crosses punctuation boundaries — i.e., if the history for the current generation was `"lazy dog. Th_"`, the n-gram would try to use `"gTh"` as the history for the next letter despite these letters not even pertaining to the same word.

The two example pieces of literature that were used while working on

this project were *The Great Gatsby* (text) and the script to the movie *The Bee Movie* (key). Here's an excerpt of the encoded text from *The Great Gatsby*:

```
Chapter 1


In my younger and more vulnerable yea| | fat|r g|e me |me advice
that I've b|n turni| over in my m|d ever since.

"When|er you |el like criticizing any o|," he to| me, "just
remember t|t all the people in t|s world haven't |d the advantages
t|t you've had."
```

And here's that same text after it had been decoded using the script from *The Bee Movie* as a key:

```
Chapter 1


In my younger and more vulnerable yeahe at father gice me ntme advice
that I've bran turnias over in my mbod ever since.

"Whenaner you rcel like criticizing any one," he tone me, "just
remember tint all the people in ters world haven't hed the advantages
tint you've had."
```

In order to fix the boundary breaches across punctuation (the original implementation was much worse than the example, however, the example text generated from that version is too old to be able to re-generate) and make the text slightly more understandable, a method of detecting and truncating n-grams that traversed punctuation boundaries was implemented. In addition to this new method of truncating the n-grams, generation of tokenized n-grams was extended to generate "future" tokens of the same size as the removed text during the encoding process. This means that instead of randomly selecting 2 (example) letters twice in a row, the expected pattern of 2 letters is saved during the n-gram process which makes it much more likely to generate coherent blocks of letters.

Unfortunately, even after these improvements to the token regeneration, the text still isn't in a particularly useful state. It's passable as readable, however, it's not a pleasant experience. Additionally, the amount of compression being achieved by this technique is very small — about 10–20% depending on the text and compression level used.

An example of the new decoded text is as follows:

```
Chapter 1



In my younger and more vulnerable year t father gice me nme advice
that I've bran turnish over in my mand ever since.

"Whenaner you rel like criticizing any one," he ton me, "just
remember tast all the people in ters world haven't had the advantages
trot you've had."
```

After reading enough of this decoded text, the work starts to sound like it was written by an 1800's western slightly-less-literate cattle rancher.

Word Based Compression Implementation

After enough experimentation with letter based compression, a new approach was clearly needed. That being said, what if the token sizes used were actually words instead of letters? How might this affect generation and sentence structure? There's something to be said about the granularity of using letters as tokens. However, to that regard, the same granularity that letters provide is also the reason for the wild variance of generation mid-word. Naturally, letters as tokens also do not scale for larger words.

The solution to all of this was to increase the scale of the tokens. The same procedure used for letter sized tokens can be used for word sized tokens with some slight tweaks. For words, additional logic is needed to group and identify whole words which complicates the n-gram generation process (it also increases runtime complexity).

The first attempt at breaking up the text into words was actually way too slow to be usable to any reasonable degree. The reason for this was the reliance on `C++`'s `STL` implementation of `std::find_first_of`. This is a method used on strings to find the first occurrence of a character. The problem with this is that this introduces multiple loops **per char** in the string which multiples the traversal time of the original implementation causing the scalability of the decoding process to increase to $O(N^2)$. Of course, the fix for this was to search for and modify the string **in place** instead of looping back over past `chars`.

Those changes to the code for separating words from the input string and for separating words during n-gram generation is as follows:

```
// Get a vector of all the words in the keyfile
std::vector<std::string> keywords;

bool word = IsAlphabetical(fileContents[0]);
std::string parseString;
long size = keyContents.size();

for(int i = 0; i < size; ++i)
{
    char val = keyContents[i];

    if(word) // Currently extracting a word
    {
        if(!IsAlphabetical(val))
        {
            // Current char is not a word so push back the word and start
            // a new working string
            keywords.push_back(parseString);
            parseString = val;
            word = false;
        }else
        {
            parseString += val;
        }
    }else // Currently extracting a non-word
    {
        if(IsAlphabetical(val))
        {
            // Current char is a word so push back the word and start
            // a new working string
            parseString = val;
            word = true;
        }
    }
}
```

This implementation proved to have much different results — most of which were significantly more readable. The file size reductions due to the compression for this method also proved to be significantly higher than the letter token method — about 25–50% reduced file size depending on the compression level used and the input file.

One thing to note during both of these implementations is that they require a somewhat lengthy **key** file in order to generate / decode an encoded file. This might seem like it should add to the file size, however, this type of encoding and decoding is not uncommon in computer science. This is typically the purpose of **codecs** during file encoding and decoding, whether that be some known constant or a whole array of constants. The purpose of these files is to act as a general purpose and shared key used to encode and decode files. For something like Discrete Cosine Transform, this would be akin to the constants used in the equation for encoding and decoding `JPEG`s.

Below is the same encoded example text from before but using word sized tokens for n-gram generation instead of letters:

```
Chapter 1



In my younger | more | | my father | me | advice
that | | turning | | my mind | since.

"Whenever | | like criticizing | one," he told |, "|
remember | all | people | this world | had the |
| you've |."
```

And here's the decoded version of the text using a `SEED_OFFSET` of 3:

```
Chapter 1



In my younger a more I think my father paid me I advice
that all the turning Of course my mind off since.

"Whenever those petunias like criticizing Yeah one," he told you, "humans
remember the all the people You this world What had the human
race you've gotta."
```

Obviously, the text is significantly more coherent from a reading perspective. The context and/or narrative aren't perfect, however, and could use the same punctuation improvements as the letter token attempt. The

biggest pitfalls come from the n-gram generation not being able to find any relevant words and therefore choosing a random word. Choosing the correct text file to use as a key is, well… **key**, to getting a good result.

Compression Results

Because there isn't really a good way to perform Monte Carlo testing on literature files due to the results needing some form of readable English-based input and output, the first 3 chapters from *The Great Gatsby* were used as a benchmark. Readability also isn't really a good metric to test here, so it's assumed that the garbled nature of the text is an expected result of this process. The main goal of the text's readability is that it is somewhat possible to decipher as the bare minimum.

It may be possible to generate lorem ipsum as a testing benchmark, however, since the nature of lorem ipsum is by definition meaningless, there would be no known texts that would realistically match text from a block of lorem ipsum — except for another block of lorem ipsum of course, but at that point the resulting text is still meaningless garbage and thus wouldn't pass the minimum requirement of "readability".

All that being said, below is a table of the first three chapters of *The Great Gatsby* compressed using the script of *The Bee Movie* as a key along with the compression technique used, the compression rate used and the effective filesize reduction.

Text & Type	Raw Size	Encoded Size 50%	Size Reduced 50%	Encoded Size 75%	Size Reduced 75%	Encoded Size 100%	Size Reduced 100%
Chapter 1 - Letter	33KB	31KB	6%	29KB	12%	28KB	15%
Chapter 2 - Letter	24KB	22KB	8%	21KB	13%	20KB	17%
Chapter 3 - Letter	33KB	30KB	9%	29KB	12%	28KB	15%
Chapter 1 - Word	33KB	24KB	28%	19KB	42%	14KB	58%
Chapter 2 - Word	24KB	17KB	29%	14KB	42%	11KB	54%
Chapter 3 - Word	33KB	23KB	30%	18KB	45%	14KB	58%
Chapter 1 - Zip	33KB	15KB	55%	15KB	55%	15KB	55%
Chapter 2 - Zip	24KB	11KB	54%	11KB	54%	11KB	54%
Chapter 3 - Zip	33KB	15KB	55%	15KB	55%	15KB	55%

As seen above, the letter tokenization barely compresses the file at all and makes it quite illegible. Conversely, the word tokenization performs very well from a file size reduction standpoint and even manages to beat zipped file compression (albeit, zipped file compression is lossless and doesn't require the file to be in English) This can be seen ever further in the graph below:

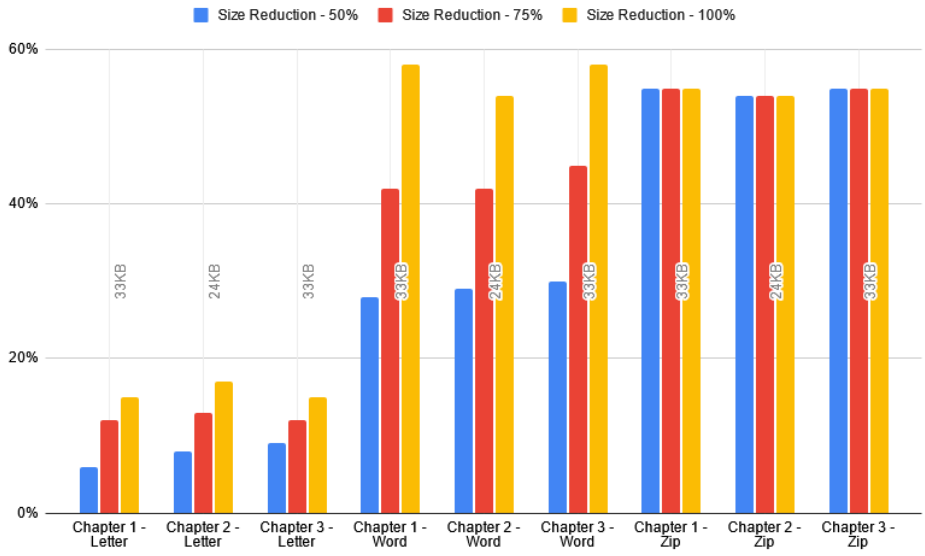

Figure 1: Side by side comparison of different test compression methods.

Possible Extensions

This project served as a proof of concept for n-gram based file compression. There are a ton of possible extensions to this technique that would either improve or restructure the idea. One such example could be to check bi-directionality when generating n-gram databases and "future" words. As it currently stands, both the letter and word tokenization only look at what comes previously in the file — this is true for both n-gram database generation and token generation. Allowing bi-direction traversal would help to increase the reliability of the generation.

Another possible technique for extension would be to take grammatical sentence structure into consideration when generating tokens. For example, if the immediately previous word in the sentence was a noun it wouldn't make much sense to place another noun. This type of validation is simple, yet often has numerous edge cases.

As far as compression goes, this type of technique leaves a decent amount of repetitive patterns in the encoded file which could be shrunk further using existing techniques — e.g., "| | | | | | |" can be shrunk to "||8|". Additionally, as far as readability goes this compression algorithm doesn't take into consideration punctuation for word based tokenization. Not only

could the accuracy of the chosen words be increased (similar to what was achieved with letter tokenization), but the finalized punctuation could at least be fixed to make the inserted words less jarring to read.

Finally, it's entirely possible to use an encoded file as the key file for another file, causing infinite recursion of "encoded" files. This is possible because the part of the file which is used for seed generation is in the first 4 characters which are always preserved when encoding.

Conclusions

Ultimately, in its current state, this algorithm doesn't have any real world practical use. This is because while the data received after decoding is legible, it isn't always contextually accurate to the original piece of literature. This algorithm does pose interesting possibilities for using n-grams or maybe even LLMs (large language models) in the future for file compression of linguistic texts.

The results of decoding from this algorithm serve an almost better purpose as entertainment. Some lines received after decoding are genuinely hilarious. On that note, setting the compression rate to 100% — i.e., removing all the original text — can sometimes create very interesting and/or funny results. Other times, however, it can create downright **terrifying** moments. Here's an example of one of those moments:

```
Chapter 1


In my younger This harmless little contraption This couldn't hurt a fly let
alone a bee Look at what has happened to bees.

"who have never been afraid to change," the world What, "about
Bee Columbus Bee Gandhi Bejesus Where I'm from we'd never sue humans
We were thinking."

. . .

on, I'm sorry the Krelman just closed out, Wax monkey's always open The
Krelman opened up again What. happened A bee died Makes an opening See He's dead Another
dead, one Deady Deadified Two more dead Dead from the neck up Dead.
from the neck up Dead from the neck up Dead from the neck up Dead
from the neck up Dead from the neck up Dead from the neck; up
Dead from the neck up Dead from the neck up
Dead from. the neck, up Dead from the neck up Dead from the, neck...
```