LZ77 Optimization

Jamie Song
Stanford University

Chendi Wu
Stanford University

Abstract

Currently, there are many compression algorithms. LZ77 is one of the most successful algorithms in use. Therefore, it is an interesting but challenging problem to research LZ77 algorithms and find room for optimizations. In this project, we separate the encoding process into table forming and table encoding. We've searched for algorithms used for both subproblems, and have implemented them. For the table forming process, we implemented both sequential search and hash-chain search heuristics for the pattern matching process, and both greeedy and optimal heuristics for the parsing process. For the table encoding process, we implemented an instinctive baseline algorithm, an optimized version, and a version using Finite State Entropy (FSE) encoding. After several unit tests ensuring the correctness of each algorithm, we performed evaluation on each combinations of the algorithms on four files, and used compression rate as the metric to evaluate our algorithms. We found out that using a certain combination works the best for all four files. We also found out some interesting features about some languages. More details will be discussed in Section 5.

1 Introduction

The project we're working on is the LZ77 optimization problem. LZ77 is an efficient and powerful tool in real life, and there leaves some room for more improvements in the algorithm. The decoding part is straightforward, while there are some interesting and challenging potential improvements left in the encoding part.
We separated the problem into two parts: optimize the pattern matching process in terms of matching efficiency and parsing strategy, and perform the entropy compression that minimizes the encoding size.
Optimizing pattern matching is a challenging problem, and there are several existing techniques such as sliding window, dynamic programming, chained-hash implementation, etc., leaving us a lot of space for discovery. After finding the matching patterns, the next problem is to find the optimal parsing strategy. In this project we implemented two strategies: greedy and optimal. More details will be introduced in section 3.
The entropy compression is designed with the expectation that the table generated from the first step has a lot of matching patterns, so the pattern part in the table would consist of a lot of empty strings. It's interesting. Meanwhile, it's different from traditional encoding schemes where no assumptions of inputs are made.
We believe that with both parts being optimized, the overall performance (in terms of compression rate) is optimized.

2 Background

We referred to the algorithm discussed in lecture for the baseline greedy sequential matching approach and the following two blog posts for optimized pattern matching algorithms:
`https://glinscott.github.io/lz/index.html` and `https://michaeldipperstein.github.io/lzss.html`.

When there are multiple matches in the search window, we need a strategy to decide which match we will take and proceed from. Thus, our implementation diverged into two: a greedy parsing strategy and an optimal parsing strategy. The greedy strategy is to always pick the longest match possible at a given position. The optimal strategy is an idea from Glin Scott's post, which involves a forward pass through all positions that computes the cost at each index, and a backward pass that picks the best option.

We implemented the baseline entropy compression heuristics completely based on the most

intuitive thoughts on compression. We followed Stanford compression Library (`https://github.com/kedartatwawadi/stanford_compression_library`) for our optimized compression algorithm, for several optimizations made in this heuristic. Then, we followed the explanation in the zstd repo (`https://github.com/facebook/zstd/blob/dev/doc/zstd_compression_format.md`) on the other optimization based on FSE compressor.

3 Algorithms

3.1 Pattern Matching

Sequential Search: This is our baseline algorithm where all possible substrings that start within the search window are compared character by character with the string currently being encoded. If there is a match with the first character of the encoding string, the match is always taken, or else the character is appended to the unmatched literals and comparison restarts at the next character. The search is not constrained within the search window, but may potentially overlap with the start of the encoding string if doing so would produce the longest possible match.

With a string of length m and a search window of length n, the runtime of the sequential search will be on the order of $O(n \times m)$.

Hash-Chain Search: This is the optimized pattern matching algorithm we resort to. A major drawback of the sequential search algorithm is that many computations are wasted in comparisons with substrings that should be known not to form a match. Thus, a performance optimization would be to track previous occurrences of the same prefix as the string currently being encoded, as they are guaranteed to produce matches, and to only compare with a fixed number of those occurrences.

The aforementioned tracking is done through maintaining two data structures: a *hash table* keyed by hashes of fixed-length prefixes and storing the index of the most recent occurrence of each prefix, and a *linked list* (chain) to trace back to all occurrences of the same prefix as in the current index, from the most recent to the most distant. While an alternative strategy is to store a list of indices corresponding to each prefix hash in the hash table, the current strategy is still more preferred, if we intend to search from a fixed window, as the tracking data structures can be kept within constant-sized memory by discarding any occurrences beyond the window in the linked list. The alternative strategy can achieve the same memory efficiency, but only at the cost of excessive modifying the lists stored within the hash-table which will incur much low-level overhead.

For each new position to encode from, we take the fixed-length prefix starting from that position, compute the hash of the prefix, consult the hash table for the position of the most recent occurrence of that prefix and use the linked list to find all past matches of the same prefix. Then, either the longest match (and offset) or all previous matches (and offsets) is returned, depending on whether the parsing strategy, which we'll discuss in the next section.

In **Algorithm 1**, we have the pseudocode of the hash-chain search algorithm, which is based on the pre-initialization of a pre-set parameter *prefix length*, a linked list on the size of the search window, a hash-table on the size of $(alphabetSize)^{prefixLength}$.

Algorithm 1 Hash-Chain Search

```
s ← encoding string
match_idx ← index of start of s
hash_key ← hash(s[match_index:match_index + prefix_length])
if hash_key exists in hash_table then
    chain[match_idx] ← hash_table[hash_key]
else
    chain[match_idx] ← -1
end if
for i in 0 ... number of hashes to search from do
    cur_search_idx ← chain[match_idx]
    if cur_search_idx < search_window_min then
        break
    end if
    cur_match_length ← MatchLengthFunc(s, cur_search_idx, match_idx)
    if cur_match_length > max match length then
        update max match length and corresponding match offset
    end if
end for
```

3.2 Parsing

Greedy: This is the baseline parsing algorithm which greedily constructs the encoding table by always taking the longest match and at all possible positions. The algorithm finishes parsing the input file in one forward pass. At the next unencoded position i, it always takes the longest possible match $s[i:j]$ and skips over all possible matches starting from indices between i and j.

Optimal: This is the more optimal parsing algorithm, which, instead of resorting to the local optimal at the next unencoded position immediately, delays the parsing decision until latter positions are parsed and compared.

The reason why this delayed approach might be more

optimal is that we're using Huffman encoding to encode the result encoding table. The optimization problem becomes minimizing the size of the code stream after the Huffman compression, instead of minimizing the size of each encoding table row directly. Thus, at a given position *i*, there might be two more optimal cases than taking the longest match: 1) taking a shorter match that is a more common pattern, and 2) skipping all matches at *i*, as the next character would form a more usable pattern. Hence, the algorithm uses an empirical heuristic introduced in **[TODO: ADD REFERENCE]** for decision making at each position.

The algorithm includes two passes: the forward pass computes a price for encoding the current position *j* as an unmatched literal or as a match among all possible previous matches. The empirical heuristic for price calculation is a function that depends on both the match length and the match offset. Prices are non-descending with the forward pass and `prices[j]` represents the cost of the best parsing strategy up until the character at *j*. Here, the underlying paradigm used is *dynamic programming* and solves the following subproblem for all *j*:

`prices[j] = min(prices[i] + matchCost(s[i:j]))`

for all match strings `s[i:j]`.

At each position, the best price, the corresponding match length and offset are stored in three lists that are later used in the backward pass.

The backward pass starts processing from the last byte and constructs the encoding table. At each position *j*, a recorded match length larger than 1 indicates there's a good match ending at position *j*, and we read back the corresponding number of bytes. Otherwise, the character at position *j* is added to the unmatched literals. Since we're in the backward pass, any addition also happens at the start.

3.3 Entropy Encoding

We've implemented three heuristics for the entropy encoding part (aka converting from the table to the bitarray and vice versa).

Baseline: The baseline algorithm is the most intuitive heuristic we implemented converting from table to binary.

In the bitarray, the first thing to encode is a 4 byte integer denoting the total number of rows in the table. Then, encode the rows one by one. Within each row, first stor the total number of characters in the pattern section of that row, and then store the pattern itself by using ascii encoding, one byte per character. Then, use two 4-byte integers to encode match length and match offset respectively.

Optimized: As indicated above, we implemented this heuristic getting the idea from Stanford Compression Library (`https://github.com/kedartatwawadi/stanford_compression_library`). We made several optimizations:

1. Since we always need to encode the number of characters per row of table, we made the table to consist of pattern length, match length, and match offset columns, and concatenate all patterns together. Then, encode the concatenated pattern using Huffman encoding.
2. We encode all integers using Elisa Delta compressor instead of the direct 4-byte binary conversion.
3. Based on the fact that Elisa Delta compressor favors small integers, we store a min_match_length somewhere in the bitarray, and within the table, the second column becomes (match length - min_match_length). In this way, the encoded integers are smaller.
4. Remove the last row of the table if match length is 0. It remains lossless compression because if the match length is 0, the match offset no longer matters. Besides, since all patterns are concatenated for encoding and decoding, the remaining unused part of the decoded pattern will be the pattern for the last row of the table. On the other hand, this process saves compression rate because if not removing the 0, the min_match_length is 0, and all the other match lengths in the table are unchanged, it means we're using one more bit in the encoding from 3) and not saving any bits.

With the four optimizations mentioned above, here is the bitarray construction:

First, **Huffman encoding:**

- The total number of bits used for Huffman.
- The total number of bits used for encoding the distribution of the concatenated pattern.
- 256 integers, each at index *i* represents the frequency of occurrence of ascii index *i*. We need to encode the distribution because we need to make sure the same Huffman tree is built for both encoding and decoding process.
- The length of encoded concatenated pattern.
- The Huffman encoded pattern.

Second, **table design:** The table now has columns pattern length, match length, and match offset.

- An Elias Delta encoded integer denoting min_match_length.

- Encoding of rows one by one. Within each row, there are three Elias Delta encoded integers denoting pattern length, match length - min_match_length, and match offset.

FSE: In addition to the optimizations mentioned in the Optimized part, we also concatenate all pattern lengths, (match length - min_match_length)'s, and match offsets. Then, encode all the four columns using FSE compression algorithm.
Here is how FSE compression works:
Encodings:

1. Build a table of number of occurrences. The total number of all occurrences needs to be a power of 2.
2. Uniformly distribute all occurrences to the table. Each row is assigned one symbol.
3. For each symbol, assign subranges based on total number of all occurrences of all symbols (denote by s), and the number of occurrences of the symbol (denote by n). The total number of subranges should be n because we want to assign each symbol to one subrange. The sum of lengths of all subranges should be s. Therefore, there should be two possible lengths, each length must be a power of 2. The shorter subrange should have length $2^{\lfloor \log_2(s/n) \rfloor}$ (denoted by l_1) bits, and the longer subrange should have length $2^{\lfloor \log_2(s/n) \rfloor + 1}$ (denoted by l_2) bits. There should be $\lfloor (s - l_2 * n)/(l_1 - l_2) \rfloor$ shorter subranges and $n - \lfloor (s - l_2 * n)/(l_1 - l_2) \rfloor$ longer subranges.
4. Encode symbol one by one. Set the original state as the of one of the occurrences of the first symbol. When reading each symbol, get all subranges of the current symbol, and find the subrange that the previous state lies in, and record the offset of previous state in the subrange. Convert the offset to binary using l bits where l is the $\log_2$ length of the subrange. It is the encoding of the symbol.
5. Convert the last state into binary because it is the starting point for the decoding process.

Decoding:

1. Get the normalized distribution and the subranges stated above.
2. Starting from the last state, read the last offset. Note that once figuring out which subrange the state belongs to, it's not hard to figure out the number of bits used to represent the offset.
3. After computing the offset, find the new state based on current state and offset. The symbol at that state is the decoded symbol. Based on the new state, read the next offset, and continue the process until finished.

Afterwards, the bitarray will consist of: FSE encoded patterns, FSE encoded pattern length list, Elias Delta encoded min_match_length, FSE encoded (match length - min_match_length) list, and FSE encoded match offset list.

4 Implementation Details

4.1 Encoder

Our encoder constructor:
`Encoder(table_type, find_match_method, binary_type, greedy_optimal, window_size, hash_num_bytes, num_hash_to_search)`
is designed flexibly to take in different combinations of parameters that makes algorithm selection and hyperparameter tuning easy.
Among these, the notable parameters are:

- `find_match_method`: allows selection from the sequential and the hash-chain matching algorithms
- `greedy_optimal`: allows selection from the greedy and optimal parsing algorithms
- `binary_type`: allows selection from the baseline, optimized and fse encoding algorithms
- `window_size`, `hash_num_bytes`, `num_hash_to_search`: allows parameter tuning that best fits machine configurations and input sources

4.2 Pattern Matching and Parsing

While our implementation follows concisely along the standard algorithms and pseudo-code whenever possible for the sake of correctness, the interface is designed to be succinct to external readers, meaning that each functionality is satisfied by only once function call with potentially different argument options, and to prioritize interchangeable use of the algorithms.
Examples of the former design methodology are the parsing algorithms: `greedy_parsing(s)` and `optimal_parsing(s)`, whereas examples of the latter are the pattern matching algorithms: `find_match_basic(s, search_idx, match_idx, greedy_optimal)` and `find_match_hashchain(self, s, search_idx, match_idx, greedy_optimal)`.

4.3 Entropy Encoding

Baseline: The baseline encoding and decoding algorithms are quite straightforward. The hardest part is in the encoding process, we used `functools.reduce` (`fold` in programming language terminology) to encode the patterns.
Optimized: We used `ProbabilityDist` as a tool to store the distribution of the concatenated pattern.
In the encoding process, we used the implemented `HuffmanEncoder` library in SCL to encode all patterns, and `EliasDeltaUintEncoder` to encode all integers.
In the decoding process, we appended another row in the table if after traversing all encoded rows, the pattern list is still not used up.
FSE: We implemented several helper functions available for both the encoding and decoding process:

- `normalizeFrequencies`: The function takes in a dict denoting the number of occurrences of each symbol. The goal is to make sure the sum of all occurrences is a power of 2, so that the later process could go on smoothly. How we implemented it is firstly to scale the sum to be the next power of 2 greater than the sum, if it's originally not a power of 2. Then, scale each occurrences in the table, and use the floor function to convert them into integers. Add 1 to the most frequent symbols until reaching total sum a power of 2.

- `formSubrange`: Given the number of occurrence of one symbol and total number of occurrences of all symbols, return the length of smaller subrange, count of smaller subrange, length of longer subrange, and count of longer subrange.

- `formUniformList`: The function takes in a distribution of symbols, and generates a uniform distribution, in the form of a list, of those symbols. The way we uniformly distribute all symbols is by enumerating all symbols in order from most frequent to least frequent and decrementing the number of occurrences by 1 on each symbol, continue until all symbols are listed out.

The above three functions are shared to both encoding and decoding. They are all deterministic. So the required components for both encoding and decoding are exactly the same.
We also encoded the distributions. To encode the distritbuion of patterns, we used the same way as we did in the Optimized heuristic, that is, storing 256 integers each denoting a number of occurrence. To encode the distribution of integers, we denote a lot of pairs of integers using Elias Delta encoder, each (k, v) pair has k to be the symbol and v to be the number of occurrences of the symbol. We also implemented several helper functions for encoder and decoder each specifically, to encode or decode integer distribution and pattern distribution.
During the actual encoding process, We first concatenate all patterns. Remove the last row if its match length is 0. Then we concatenate all pattern lengths, all (match length - min match length)'s, and all match offsets. For each lists, encoding the distribution, and encode the actual context. Then concatenating all bitarrays together to form the final encoding.
One thing we modified to the original FSE algorithm is that we add an integer denoting the total number of symbols used in the encoding. The reason for it is because if a symbol occurs a lot of times in the distribution, it is possible that the symbol is associated with subrange with length 1, and it means that this symbol will be encoded with length 0. So it will be hard to figure out when to stop the decoding algorithm because there is potentially infinite number of possible encodings with length 0 in some distributions. Therefore, we add a number of symbols for each encoding.
In the decoding process, we separate the bitarray into four concatenated lists and the decoded integer denoting min match length. Then, separately decode the four lists and combine them together to build the table.

5 Evaluation

Our main evaluation metric is the compression rate. The runtime improvement of the hash-chain algorithm is not precisely evaluated but can be immediately observed during testing. In this section, we focus our analysis on how different variables (input sources, algorithm choices, hyper-parameters) may affect the eventual compression rate and present our best compression rate in comparison to commonly-used implementations *gzip* and *zstd*.

5.1 Prefix Length and Prefix Search Range

First, we tuned our compressor with respect to two hyper-parameters: `hash_num_bytes`, meaning the length of prefix we use in the hash-chain algorithm, and `num_hash_to_search`, meaning the number of past matches we compare against.
From Figure 1, we observe that across different input sources, 6 is the best prefix length, which is also the length we use when performing the aggregated comparison in the next section. From Figure 2, we observe the expected result that the higher the number of past matches we compare against, the better the compression rate is. However, note that with a 4x increase in search range, we only get a minor compression rate improvement. Since a larger search range implies higher memory

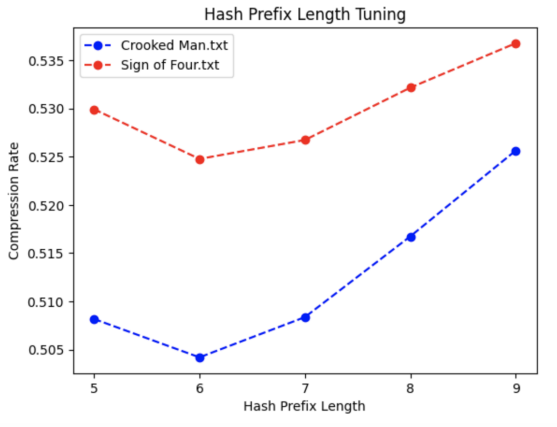

Figure 1: Prefix Length Tuning

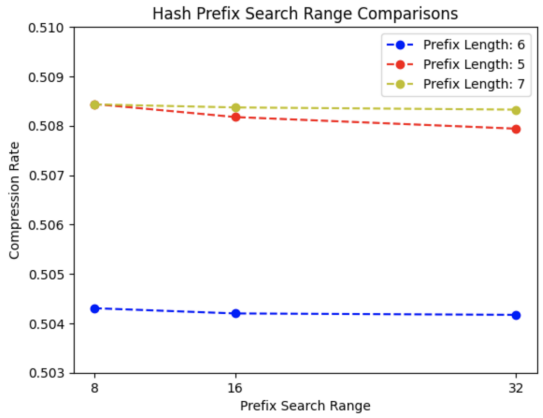

Figure 2: Number of Prefix Matches Considered

cost, we believe a search range of 8 should be sufficient.

5.2 Input Source and Algorithm Choice

We implemented 2 pattern matching algorithms, 2 parsing algorithms and 3 encoding algorithms, together contributing to 12 algorithm combinations. We used all 12 algorithm combinations to compress 4 different input sources: 2 English novels with spacing and special characters removed, 1 of the previous novels with spacing retained and 1 German novel with spacing and special characters removed.

Figure 3 shows the truncated result, generated after sorting by compression rate. The complete result is reproducible and available in our notebook file. Note that compression rate is calculated by dividing the length of the compressed file by the length of the original file, so the smaller compression rate, the better performance of the compressor. As can be observed, using **hash-chain search** as the matching algorithm, **optimal parsing** as

	File Name	Matching	Parsing	Encoding	Compression Rate
31	Verwandlung.txt	hashchain	optimal	optimized	0.458918
23	Sign of Four Spaced.txt	hashchain	optimal	optimized	0.479612
29	Verwandlung.txt	hashchain	greedy	optimized	0.489830
27	Verwandlung.txt	basic	optimal	optimized	0.499876
21	Sign of Four Spaced.txt	hashchain	greedy	optimized	0.500678
15	Crooked Man.txt	hashchain	optimal	optimized	0.504199
13	Crooked Man.txt	hashchain	greedy	optimized	0.515835
7	Sign of Four.txt	hashchain	optimal	optimized	0.524775
5	Sign of Four.txt	hashchain	greedy	optimized	0.531815

Figure 3: Algorithm Comparisons on Different Sources

File Name	Our Rate	gzip Rate	zstd Rate
Crooked Man	0.5042	0.4605	0.4637
Sign of Four	0.5248	0.4950	0.4956
Sign of Four(S)	0.4796	0.4395	0.4485
Verwandlung	0.4589	0.4038	0.4134

Table 1: Comparison with gzip and zstd

the parsing algorithm and **optimized encoding** as the encoding algorithm produces the best compression rates on all 4 sources. What's interesting to observe from the variation in input sources is that the English novel that retains spacing has better a compression rate that the one that removes spacing, and the German novel generally has better compression rates than the English novels. The reason for the former observation is that spaces are strong influencers in pattern matching, and the reason for the latter is that the German language more extensively builds longer words upon shorter, root words and is thus better for a compression scheme that relies on pattern matching.

In figure 4, our analysis focuses on which family of algorithm, among the matching, parsing and encoding families, has the most contribution to good compression rates. Again, the figure is a portion of the larger analysis efforts but well exemplifies the strong influence of the encoding algorithm family. Switching from baseline to optimized encoding is usually an at least **3x performance boost**, and switching from basic to hash-chain is usually a **1.2x boost**.

5.3 Comparison with Common Implementations

While our implementation still experiences a performance gap with the commonly-used implementations gzip and zstd (shown in Table 1), our compression rate is still in a reasonable range and can be further optimized given more time.

	File Name	Matching	Parsing	Encoding	Compression Rate
17	Sign of Four Spaced.txt	basic	greedy	optimized	0.647468
8	Sign of Four Spaced.txt	basic	greedy	fse	0.852225
16	Sign of Four Spaced.txt	basic	greedy	baseline	2.927712
19	Sign of Four Spaced.txt	basic	optimal	optimized	0.547189
9	Sign of Four Spaced.txt	basic	optimal	fse	0.706159
18	Sign of Four Spaced.txt	basic	optimal	baseline	2.353816
21	Sign of Four Spaced.txt	hashchain	greedy	optimized	0.500678
10	Sign of Four Spaced.txt	hashchain	greedy	fse	0.582223
20	Sign of Four Spaced.txt	hashchain	greedy	baseline	1.312220
23	Sign of Four Spaced.txt	hashchain	optimal	optimized	0.479612
11	Sign of Four Spaced.txt	hashchain	optimal	fse	0.567948
22	Sign of Four Spaced.txt	hashchain	optimal	baseline	1.308628

Figure 4: Algorithm Impacts

6 Conclusion

In conclusion, with the 12 possible combinations of algorithms, we find out that the combination of hashchain, optimal, optimized, with hash prefix length 6, to be the best among our implemented heuristics, using compression rate as the metric. As shown by our thoroughly evaluation and analysis, the performance of our implementation is approaching the commonly-used compressors. Although haven't defeating the compressors, we have learned a lot of different algorithms and compressors throughout the quarter.

References