



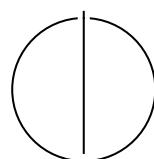
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Improved Symbol Table Construction for
FSST Compression**

Hedi Chehaidar





SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

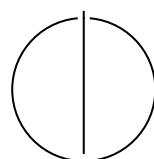
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Improved Symbol Table Construction for
FSST Compression**

**Verbesserte Symboltabellenkonstruktion für
die FSST-Komprimierung**

Author: Hedi Chehaidar
Examiner: Prof. Dr. Thomas Neumann
Supervisors: Prof. Dr. Andreas Kipf, Mihail Stoian
Submission Date: January 14, 2026



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, January 14, 2026

Hedi Chehaidar

Acknowledgments

First of all, I want to thank God for the guidance and strength that he gave me to complete this thesis.

I would like to express my sincere gratitude to all those, who supported me morally and technically throughout this work.

I would like to thank my supervisor Mihail Stoian for the continuous guidance, insightful feedback, and trust in my ideas. The discussions we had, from discussing ideas to detailed implementation improvements, helped shape the thesis and improve its quality. I am thankful for the encouragements and for pushing me to always explore more optimization possibilities. I am grateful for the practical insights, constructive criticism, and technical expertise, especially regarding algorithmic correctness and performance evaluation. The feedback I received helped turn initial ideas into well-structured and measurable contributions.

I am grateful to the broader research and open-source community whose work laid the foundation for this thesis. The availability of high-quality implementations, documentation, and discussions around compression methods made it possible to build upon existing work and focus on meaningful improvements.

A special thanks goes to everyone who shared tools, datasets, or prior experience that helped accelerate experimentation and evaluation.

Finally, I want to thank my friends and family for their constant support, patience, and encouragement throughout this journey. Their motivation helped me stay focused and enthusiastic, especially during long debugging sessions and performance tuning phases.

This thesis would not have been possible without the collective contributions of all these people, and I am truly thankful for their support and inspiration.

Abstract

Modern analytical database systems process large volumes of string data, where efficient compression is essential for reducing memory footprint and improving performance. While the Fast Static Symbol Table (FSST) compression scheme provides excellent decompression speed and random-access capabilities, its effectiveness heavily depends on the quality of the constructed symbol table.

The original FSST algorithm relies on heuristic, greedy symbol selection, which can lead to suboptimal symbol choices and limit achievable compression ratios.

This thesis presents several enhancements to the FSST compression method that improve symbol table construction while preserving FSST's core advantages. The central contribution is a refined symbol selection process that systematically identifies more effective symbols and avoids redundant or conflicting choices.

First, a dynamic programming approach is introduced to evaluate and select higher-quality symbols within each generation, enabling a more globally informed optimization compared to the original greedy strategy. Second, an additional frequency counter is incorporated to accelerate the discovery of longer symbols and to explicitly favor them in subsequent generations, improving the exploitation of longer recurring patterns in the data. Third, a symbol pruning mechanism is applied to eliminate conflicting and redundant symbols, ensuring a more compact and effective symbol table.

Together, these techniques significantly improve the robustness and quality of the symbol table generation process. Experimental evaluation demonstrates that the proposed enhancements lead to consistently improved compression ratios compared to the original FSST algorithm, while maintaining its fast decompression and random-access properties. The results show that careful algorithmic refinement of symbol selection can yield substantial gains without altering the lightweight and practical nature of FSST, making the improved approach well suited for use in modern analytical systems.

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Outline	2
2 Background	4
2.1 Dynamic Programming vs. Greedy	4
2.1.1 Greedy Algorithms	4
2.1.2 Dynamic Programming	5
2.2 Datastructures	5
2.2.1 Max Heap	5
2.2.2 Trie	7
3 Related Work	8
3.1 Block-based Compressors	8
3.1.1 LZ4	8
3.1.2 Zstd	9
3.1.3 Limitations	9
3.2 FSST	10
3.2.1 General Idea	10
3.2.2 Decompression	11
3.2.3 Compression	11
3.2.4 Symbol Table Construction	12
4 Approach	15
4.1 Dynamic Programming	15
4.2 3 rd Counter	18
4.3 Symbol Pruning	18

Contents

5 Evaluation	20
5.1 Environment	20
5.2 Benchmarking Data	20
5.3 Results	20
5.3.1 DP approach	20
6 Discussion & Future Work	21
7 Conclusion	22
List of Figures	23
List of Tables	24
Bibliography	25

1 Introduction

1.1 Motivation

Modern data management systems increasingly operate on large-scale, string-heavy datasets. In analytical databases, strings are omnipresent and appear in many forms, including URLs, file paths, identifiers, log messages, categorical attributes, and semi-structured data originating from web, cloud, and enterprise applications. Studies of real-world database workloads show that string columns often constitute a substantial fraction of the overall data volume, both in terms of storage and memory consumption. Prior analyses of analytical benchmarks and production systems report that strings can account for a significant portion of columnar storage, frequently dominating memory usage in dictionary-encoded or compressed representations [1], [2].

The growing prevalence of string data places strong demands on compression techniques used in analytical systems. Effective compression reduces memory footprint, improves cache utilization, and lowers memory bandwidth pressure, all of which are critical for high-performance query processing. At the same time, analytical workloads require fast random access to individual values, as queries often scan and decode only a subset of columns or rows. This combination of requirements makes general-purpose, block-based compressors such as Zstandard or LZ4 less suitable despite their strong compression ratios, as they typically require decompressing entire blocks before accessing individual strings.

To address this gap, lightweight string compression schemes have been proposed that prioritize fast decompression and random access. One prominent example is the FSST compression algorithm [3]. FSST compresses strings by replacing frequent byte sequences with compact symbols from a statically constructed symbol table. During decompression, symbols can be expanded independently, allowing direct access to individual strings without scanning neighboring data. As a result, FSST achieves decompression speeds that are competitive with, and often superior to, more heavyweight compression schemes, making it attractive for use in modern analytical databases.

However, the compression effectiveness of FSST critically depends on the quality of

its symbol table. The original FSST algorithm constructs this table using a greedy approach that iteratively selects symbols based on local heuristics. While this strategy is computationally efficient and aligns with FSST’s design goal of lightweight processing, it can lead to suboptimal symbol choices. In particular, greedy selection may favor short or locally frequent symbols that conflict with longer or more informative sequences, limit the discovery of beneficial longer symbols, or introduce redundancy within the symbol table. Furthermore, once a symbol is selected, its impact on future generations of symbols is not globally optimized, which can prevent the algorithm from converging toward a symbol set that maximizes overall compression gain.

These limitations suggest that there is room for improvement in FSST’s symbol table construction without compromising its core advantages. By revisiting the greedy nature of symbol selection and incorporating more informed decision-making into the construction process, it is possible to improve compression ratios while retaining FSST’s fast decompression and random-access properties. This thesis explores such improvements, focusing on enhanced symbol selection strategies that address the shortcomings of the original greedy approach.

1.2 Thesis Outline

This thesis is structured as follows.

Chapter 2 (Background) introduces the fundamental concepts required to understand the techniques developed in this thesis. It begins with an overview of dynamic programming and contrasts it with greedy algorithmic approaches, highlighting their respective strengths and limitations. The chapter then introduces the core data structures used throughout the thesis, namely tries and max-heaps, which play a central role in symbol generation and selection.

Chapter 3 (Related Work) reviews existing work in the area of data compression. It first discusses widely used block-based compression algorithms such as LZ4 and Zstandard, explaining their general design principles and trade-offs. The chapter then presents the original FSST compression algorithm, detailing its symbol table construction and encoding process, which form the basis for the improvements proposed in this thesis.

Chapter 4 (Approach) describes the main contributions of this work. It introduces three enhancements to the FSST symbol table construction process: a dynamic programming-based method for improved symbol selection, the introduction of a third frequency counter to accelerate the discovery and prioritization of longer symbols,

and a symbol pruning strategy to eliminate conflicting and redundant symbols. Each technique is explained in detail and integrated into the overall compression pipeline.

Chapter 5 (Evaluation) evaluates the proposed enhancements experimentally. It presents the datasets, experimental setup, and benchmarking methodology used in the evaluation. The chapter analyzes the impact of the proposed techniques on compression ratio and runtime, comparing the improved FSST variants against the original algorithm.

Chapter 6 (Discussion & Future Work) reflects on the achieved improvements and discusses directions for future work, with emphasis on potential optimization opportunities to reduce the runtime overhead introduced by the enhanced symbol selection techniques.

Chapter 7 (Conclusion) summarizes the contributions and findings of the thesis.

2 Background

2.1 Dynamic Programming vs. Greedy

Many algorithmic problems involve selecting a sequence of decisions that together optimize a global objective. Two widely used approaches for solving such problems are greedy algorithms and dynamic programming (DP). While both aim to construct efficient solutions, they differ fundamentally in how they explore the solution space and reason about optimality.

2.1.1 Greedy Algorithms

Greedy algorithms build a solution incrementally by making locally optimal decisions at each step. At every stage, the algorithm selects the option that appears best according to a predefined heuristic, without reconsidering previous choices. This strategy is attractive due to its simplicity, low computational overhead, and ease of implementation.

Greedy approaches are particularly effective when a problem exhibits the greedy-choice property, meaning that a locally optimal decision can be shown to lead to a globally optimal solution.

An easy example illustrating the greedy-choice property is the problem where we are given a sequence of distinct numbers and we want to take a subset of K numbers with the goal of maximizing the sum of those numbers. The intuitive and correct way to choose the numbers is by selecting the largest K numbers of the sequence for the subset (denoted by S). We can prove this by contradiction: suppose there is an optimal solution where a number of the chosen subset does not belong to S , we can swap that number with an unchosen number of the set S and we would have a strictly better solution, which leads to contradiction.

An example of a problem that cannot be solved greedily and requires DP is the 0/1 Knapsack problem.

Despite this limitation, greedy algorithms are frequently used in performance-critical systems where execution speed and simplicity are prioritized over absolute optimality.

2.1.2 Dynamic Programming

Dynamic programming (DP) addresses optimization problems by systematically exploring all relevant subproblems and combining their solutions to obtain a globally optimal result. The core idea is to decompose a problem into overlapping subproblems, solve each subproblem once, and store its result to avoid redundant computation.

Unlike greedy algorithms, dynamic programming evaluates the long-term consequences of decisions. By considering multiple possible choices at each stage and selecting the one that minimizes or maximizes a well-defined cost function, DP can guarantee optimality when the problem satisfies optimal substructure and overlapping subproblems.

We define the 0/1 Knapsack problem formally as follows: there are n items where each item i has value v_i and weight w_i and the knapsack capacity is W . We need to choose a subset of items that fits the capacity while maximizing the total value. Define the DP table: $dp[i][c] = \text{maximum value achievable using the first } i \text{ items with capacity } c$. So the result is stored in $dp[n][W]$. The base cases are $dp[i][0] = 0$ for all i and $dp[0][c] = 0$ for all c . The recurrence is defined for each item i and capacity c as follows: If $w_i > c$ (item does not fit): $dp[i][c] = dp[i - 1][c]$. Otherwise choose best of taking or not taking the item: $dp[i][c] = \max(dp[i - 1][c], dp[i - 1][c - w_i] + v_i)$. Time complexity is $O(nW)$.

The primary trade-off of dynamic programming is computational complexity. DP solutions often require more time and memory than greedy alternatives, especially when the state space is large. As a result, practical DP implementations frequently rely on carefully designed cost models, state compression, or bounded problem sizes to remain efficient.

2.2 Datastructures

Efficient algorithmic design relies not only on the choice of optimization strategy but also on the use of appropriate data structures. This section introduces the two core data structures used throughout this thesis: max-heaps and tries. Both play an important role in managing candidates and efficiently representing string data.

2.2.1 Max Heap

A max heap is a complete binary tree-based data structure that maintains the heap property: the key of each node is greater than or equal to the keys of its children. As a result, the maximum element is always stored at the root of the heap.

The primary operations supported by a max heap include insertion, extraction of the maximum element, and key updates. Each of these operations can be performed in logarithmic time with respect to the number of elements in the heap. This efficiency makes max-heaps particularly well suited for priority-based selection tasks, where the most valuable or promising element must be accessed repeatedly.

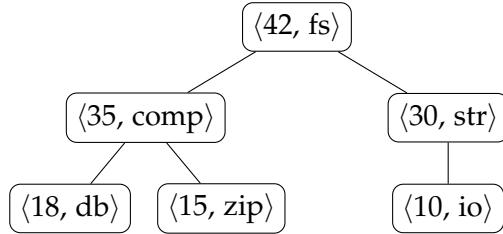


Figure 2.1: Initial max heap ordered by frequency.

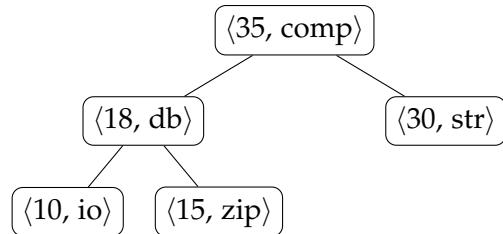


Figure 2.2: Max heap after extracting the maximum element ⟨42, fs⟩.

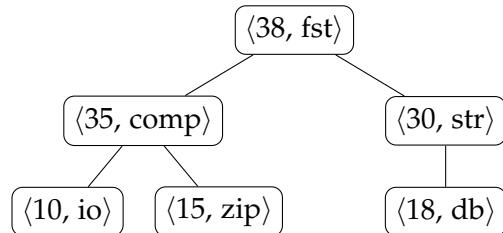


Figure 2.3: Max heap after inserting the element ⟨38, fst⟩.

In the context of algorithmic optimization, max-heaps are often used to manage candidate sets ordered by a score or heuristic value. They enable fast retrieval of the currently best candidate while allowing dynamic updates as new candidates are generated or existing ones are reweighted.

2.2.2 Trie

A trie, also known as a prefix tree, is a tree-based data structure used to store and retrieve strings efficiently by exploiting their shared prefixes. Each node in a trie represents a prefix of one or more strings, and edges correspond to individual characters or bytes. Strings are represented by paths from the root to terminal nodes which can also store other data related to the strings (like mappings).

Tries provide efficient operations for prefix-based queries, insertion, and lookup, all of which can be performed in time proportional to the length of the string rather than the number of stored strings. This makes them particularly suitable for applications involving multiple search operations for strings with common prefixes.

Following trie stores the 5 strings "fs", "fst", "db", "dbms" and "zip" with a mapping for each string as an integer stored in the corresponding terminal node.

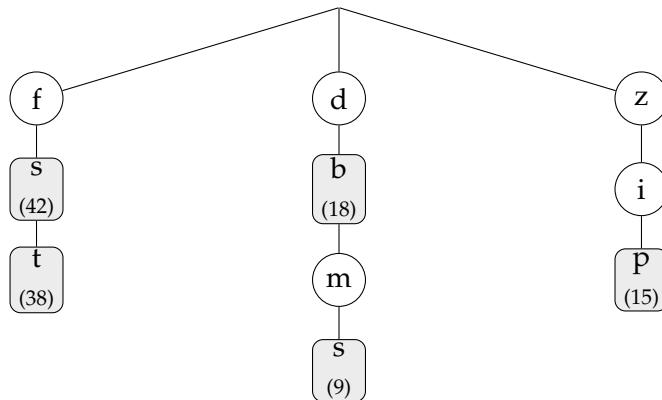


Figure 2.4: Example trie storing short strings. Terminal nodes are annotated with integer metadata.

3 Related Work

3.1 Block-based Compressors

General-purpose block-based compression algorithms are widely used across storage systems, operating systems, and data processing pipelines due to their ability to achieve high compression ratios on diverse data types. These compressors operate by partitioning the input data into blocks of fixed or variable size and compressing each block independently. This design enables parallel compression and decompression, robust error containment, and adaptability to different data distributions.

3.1.1 LZ4

One of the most prominent examples of such algorithms is LZ4, a lightweight compressor based on the Lempel–Ziv 77 (LZ77) family of dictionary compression techniques. The LZ77 general technique works by scanning a block with a usual size of 128 KB as input, detecting byte sequences that appeared before. Repeated sequences are encoded as a pair $(\text{offset}, \text{length})$ where offset references the first appearance of the sequence in the output stream.

What distinguishes LZ4 is that during compression a hash table is used to quickly identify matches in the input and greedily accepting the first match found. This approach minimizes CPU usage, providing compression speed > 500 MB/s per core.

LZ4 decompression is very similar to just performing memory copy ('memcpy') operations, making it bounded by memory bandwidth with speed of multiple GB/s per core, typically reaching RAM speed limits on multi-core systems.

LZ4 prioritizes decompression speed over compression ratio, making it suitable for performance-critical workloads where fast data access is required. It achieves compression by identifying repeated byte sequences within a block and encoding them as references to earlier occurrences [4].

While LZ4 offers very fast decompression, its relatively small window size and simple matching strategy limit its ability to exploit longer-range redundancies, especially across block boundaries.

3.1.2 Zstd

Zstandard (Zstd) [5] represents a more advanced block-based compression approach developed by Facebook that balances compression ratio and performance. Zstandard combines LZ-style dictionary matching with entropy coding and supports large compression windows, allowing it to capture long-distance repetitions within a block. Zstd encodes unmatched byte sequences, called literal bytes, in addition to the offset and length attributes of match pairs using Finite State Entropy (FSE) and Huffman coding techniques. As a result, Zstandard often achieves significantly higher compression ratios than lightweight compressors such as LZ4, particularly on structured or repetitive data. However, this improved compression comes at the cost of increased decompression complexity and reduced random-access efficiency, as individual values typically cannot be decompressed without processing the surrounding block.

Table 3.1: Comparison of different block-based compressors with the Silesia compression corpus [5], [6]

Compressor name	Ratio	Compression	Decompress.
zstd 1.5.6 -1	2.887	510 MB/s	1580 MB/s
zlib 1.2.11 -1	2.743	95 MB/s	400 MB/s
brotli 1.0.9 -0	2.702	395 MB/s	430 MB/s
zstd 1.5.6 –fast=1	2.437	545 MB/s	1890 MB/s
zstd 1.5.6 –fast=3	2.239	650 MB/s	2000 MB/s
quicklz 1.5.0 -1	2.238	525 MB/s	750 MB/s
lzo1x 2.10 -1	2.106	650 MB/s	825 MB/s
lz4 1.9.4	2.101	700 MB/s	4000 MB/s
lzf 3.6 -1	2.077	420 MB/s	830 MB/s
snappy 1.1.9	2.073	530 MB/s	1660 MB/s

3.1.3 Limitations

While block-based compressors are highly effective for bulk storage and sequential access patterns, their design fundamentally limits random access performance. In analytical database systems, queries often access individual values or small subsets of columns rather than scanning entire blocks. In such scenarios, block-based compression requires decompressing full blocks even when only a small portion of the data is needed, leading to unnecessary computation and memory traffic. This limitation has been highlighted in prior work on column-oriented storage and analytical query processing, where fine-grained access and low-latency decompression are critical for

performance.

Several systems attempt to mitigate this issue by using smaller block sizes or by combining block-based compression with dictionary encoding. However, reducing block size typically degrades compression efficiency, while hybrid approaches increase system complexity. As discussed in the FSST paper, these trade-offs make block-based compressors less attractive for scenarios where fast random access to compressed strings is a primary requirement [3].

Consequently, lightweight compression schemes that avoid block-level dependencies have gained attention in the context of analytical databases. These schemes trade some compression efficiency for predictable decompression costs and fine-grained access. FSST belongs to this class of compressors, providing symbol-based compression that allows individual strings to be decompressed independently. Understanding the strengths and limitations of block-based compressors is therefore essential for positioning FSST and the improvements proposed in this thesis relative to existing compression techniques.

3.2 FSST

3.2.1 General Idea

Fast Static Symbol Table (FSST) [3] is a lightweight string compression algorithm that works by replacing substrings (symbols) in the original data with one-byte codes. A symbol table is constructed from a sample of the data, mapping symbols to codes. The term “static” refers to the fact that any string can be compressed or decompressed independently using the symbol table, without requiring any prior context, unlike block-based compressors discussed in the previous section. Individual strings in databases are typically no larger than 200 bytes, with the majority being less than 30 bytes. This makes the random-access property of FSST more valuable compared to block-based compressors, which often need to decompress an entire block of 128 KB to retrieve a single string. FSST also supports compressed query processing, since equality between original strings is equivalent to equality between their compressed representations. It may also be possible to perform more complex operations on compressed strings, such as pattern matching, which can reduce the need to decompress strings when processing a wide range of queries.

Symbols have lengths ranging from 1 to 8 bytes, and each symbol is mapped to a code from 0 to 254, with code 255 reserved as an escape code that plays an important role during decompression. Consequently, the total number of usable symbols is 255. This results in a worst-case symbol table memory overhead of $8 \times 255 + 255$ bytes, which is very unlikely in practice, as many symbols tend to be only 2 to 3 bytes long.

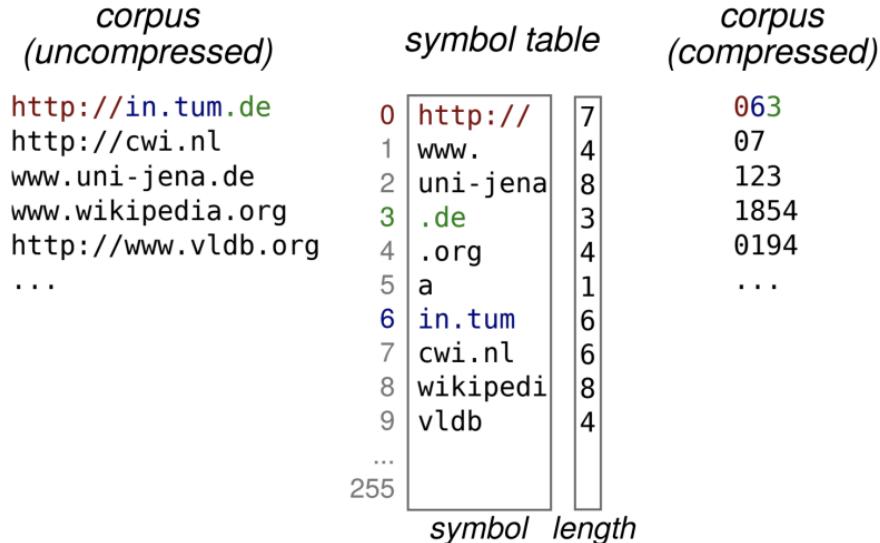


Figure 3.1: FSST compression example [3]

3.2.2 Decompression

FSST decompression is fairly simple. The algorithm iterates over the encoded input stream byte for byte and appends the corresponding symbol from the symbol table to the output stream. The exception of the escape code (code 255) can be checked with an if statement, where in that case the algorithm appends the following raw byte after the escape code as is and continues with the next byte.

The original FSST open-source implementation [7] optimizes decoding to avoid performing an if statement for every byte and instead loads 4 bytes every time and then checks if they contain the escape code. FSST decoding is therefore comparably fast approaching 2 GB/s according to the original paper's evaluation.

The improvements of this thesis do not affect the FSST desompression method in any way. Decompression was used in this thesis solely to verify robustness of the added contributions.

3.2.3 Compression

FSST compression is performed independently for each string in the corpus. A loop iterates over the bytes of the string. In each iteration, the longest matching symbol starting at the current position is used for encoding, and the current position in the

string is updated accordingly. If no symbol in the table matches the current position, the byte is escaped by appending the escape code to the output stream, followed by the raw byte from the string.

The function used to find the longest matching symbol is called *findLongestSymbol*, and its implementation is shown in Figure 3.2. The same function is also used during symbol table construction.

This greedy approach enables fast encoding at the cost of a suboptimal compression factor. An alternative approach is presented in Section 4.1.

3.2.4 Symbol Table Construction

The construction of the symbol table is the most important component of the FSST algorithm with respect to the achieved compression factor, as the encoding quality is tightly dependent on the set of symbols selected for the table. Selecting symbols is challenging because their effectiveness depends on one another whenever they overlap. This is the dependency issue described in the original paper. So the only reliable way to assess the significance of a symbol is to compress the sample using the current symbol table. However, evaluating all possible symbol tables is not feasible due to the enormous number of possibilities ($\binom{8N}{255}$ where N denotes the sample size).

The FSST algorithm shown in Figure 3.2 constructs the symbol table by iterating over a sample of the corpus for a fixed number of generations. In each generation, the current table is refined by discovering new symbol combinations and retaining those with the highest static gain. The gain of a symbol is computed as $gain = length \times frequency$, where length is the symbol length (between 1 and 8 bytes) and frequency is the number of times the symbol is selected when compressing the sample using the previous table.

In the original implementation, the number of generations is set to five and the sample size to 16 KB. The sample is processed incrementally, such that in each generation a larger fraction of the sample is considered for symbol selection.

The *buildSymbolTable* function is the main component of the algorithm, in which the five iterations over the sample are performed. Two counters, *count1* and *count2*, are used to record symbol frequencies when compressing the text using the previous symbol table. The counter *count1* tracks individual symbols, while *count2* records the concatenation of two symbols or the concatenation of a symbol and a literal byte.

The indices of these counters range from 0 to 511. Indices from 0 to 255 correspond to escaped bytes (literal bytes), while indices from 256 to $256 + st.nSymbols - 1$ (up to 511) denote actual symbols.

In each generation, the counters are populated using the *compressCount* function, which compresses the text with the current symbol table. This function counts individual symbols in *count1* and records all pairs of successive symbols, as well as

3 Related Work

```

class SymbolTable:
    def __init__(st): # constructor
        st.nSymbols = 0
        st.sIndex[257] = 0*256
        st.symbols[512] = ']'*512
        # the first 256 symbols are escaped bytes
        for code in range(0,256):
            st.symbols[code] = chr(code)

    def insert(st, s):
        st.symbols[256+st.nSymbols] = s

    def findLongestSymbol(st, text):
        var letter = ord(text[0])
        # try all symbols that start with this letter
        for code in range(st.sIndex[letter], st.sIndex[letter+1]):
            if (text.startswith(st.symbols[code])):
                return code # symbol, code >= 256
        return letter # non-symbol byte (will be escaped)

    # compress the sample and count the frequencies
    def compressCount(st, count1, count2, text):
        var pos = 0
        var prev, code = st.findLongestSymbol(text[pos:])
        while ((pos += st.symbols[code].len()) < text.len()):
            prev = code
            code = st.findLongestSymbol(text[pos:])
            # count the frequencies
            count2[prev][code] += count1[prev][code]
            # we also count frequencies for the next byte only
            if (code >= 256):
                nextByte = ord(text[pos])
                count1[nextByte] += count2[prev][nextByte]
                count2[prev][nextByte] +=

    def makeTable(st, count1, count2): # pick top symbols
        var res = SymbolTable()
        var candis = []
        for code1 in range(0,256+st.nSymbols):
            # single symbols (+all bytes 0..255) are candidates
            gain = st.symbols[code1].len() * count1[code1]
            heapq.heappush(candis, (gain, st.symbols[code1]))
            for code2 in range(0,256+st.nSymbols):
                # concatenated symbols are also candidates
                s = (st.symbols[code1]+st.symbols[code2])[::8]
                gain = s.len() * count2[code1][code2]
                heapq.heappush(candis, (gain, s))
            # fill with the most worthwhile candidates
            while (res.nSymbols < 255):
                res.insert(heapq.heappop(candis))
        return res.makeIndex()

    def makeIndex(st): # make index for findLongestSymbol
        # sort the real symbols and init the letter index
        var tmp = sort(st.symbols[256:256+st.nSymbols])
        for i in range(0,st.nSymbols).reverse():
            var letter = ord(tmp[i][0])
            st.sIndex[letter] = 256+i
            st.symbols[256-i] = tmp[i]
            st.symbols[256] = 256+st.nSymbols # sentinel
        return st

    def buildSymbolTable(st, text): # top-level entry point
        var res = SymbolTable()
        for genLevel in [1,2,3,4,5]:
            var count1[512][512] = 0
            var count2[512][512] = 0
            st.compressCount(res, count1, count2, text)
            res = st.makeTable(res, count1, count2)
        return res

```

Figure 3.2: FSST initial idea symbol table construction [3]

combinations of a symbol followed by a literal byte, in *count2*.

Real symbols are stored in *st.symbols* starting from index 256 in alphabetical order. Symbols that begin with the same character are sorted in decreasing order of length. As a result, *st.findLongestSymbol* terminates after finding the first matching symbol. The implementation details of this function in the open-source code are discussed later.

A new symbol table is then constructed from the counters using the *makeTable* function. Each symbol, literal byte, and their concatenations are inserted into a max heap, where the key is the computed gain of the candidate symbol. The 255 symbols with the highest static gains are selected for the next symbol table. Finally, the *makeIndex* function sorts the symbols as described above and initializes the *st.index[letter]* table, which stores the position (code) of the first symbol beginning with a given letter.

The FSST algorithm evolved from the previously described version to a branchless

implementation that enables the use of SIMD instructions (AVX-512), significantly reducing encoding time. The primary bottleneck in encoding is the *findLongestSymbol* function. Therefore, a redesign of the symbol storage was introduced as a first optimization.

Symbol codes are now stored in two separate data structures: a 256×256 array named *shortCodes* for symbols of length one and two bytes, and a hash table *hashTab* of size 1024 (1 KB) for symbols with lengths ranging from three to eight bytes. A lossy perfect hashing scheme is employed for longer symbols, using a hash function based on the first three bytes of the symbol. If inserting a symbol would result in a collision, the symbol is discarded, retaining only the symbol with the highest gain among colliding symbols of length three or greater.

For each character A for which no entry in $\text{shortCodes}[A][*]$ is defined, the corresponding single-byte symbol is inserted if it exists. This guarantees that an access to *shortCodes* always returns the longest matching symbol of length one or two. The *findLongestSymbol* function first queries the *hashTab* to search for a matching symbol, then consults *shortCodes* if no match is found, and finally falls back to returning the literal byte. The function's return value is computed using a conditional move (MOV) instruction, thereby avoiding branching.

Although the original FSST algorithm is highly optimized for runtime, particularly when using the SIMD variant, the greedy strategy employed by the *findLongestSymbol* function does not always select the optimal symbol, both during compression and during symbol table construction. This leaves room for improving the compression factor by adopting a different optimization strategy that more carefully considers conflicts between overlapping symbols, as discussed in the following section.

4 Approach

4.1 Dynamic Programming

The first contribution of the thesis to the FSST algorithm of Figure 3.2 is replacing the *st.findLongestSymbol* method with a DP function that gives "one" best symbol given the position in the text to encode (or training text in `compressCount`) with a single table lookup in $O(1)$ runtime. This table called *opt* is constructed simultaneously with the DP table.

To understand the utility of the used DP function in the code, let's first define the more intuitive DP function for $0 \leq i \leq n$ (where n is the text length) as:

$dp[i]$ = smallest compressed size of the first i bytes of the given text.

So the result for the whole text is in $dp[n]$. The base case is $dp[0] = 0$ as the compression of an empty string is always an empty string. The recurrence is defined for each other value as:

$$dp[i] = \min(2 + dp[i - 1], 1 + \min_{\substack{0 \leq j < i \\ s[j:i] \in symbols}} dp[j])$$

where $s[j : i]$ denotes the substring of the text from position j to $i - 1$ (0-indexed). The formula simply chooses between either escaping the current byte or selecting one symbol that minimizes the DP value. In the first case, an overhead of two bytes must be added resulted in the escape byte followed by the raw byte from the string during encoding. In the second case, there is an added one byte for the code of the chosen symbol. Then all candidate symbols are the ones that match a suffix of the first i bytes, thus the constraints under the second *min*.

The problem with this formula is that the candidate symbols are ending at the position i , while building the *opt* table requires knowing one best symbol beginning at a position i . So implementing this DP function will not help by filling the *opt* table simultaneously, as we still need another dedicated function that also iterates over candidate symbols again at each position, but this time the candidate symbols are beginning at the current position and choosing one best symbol will depend on the DP

values computed beforehand.

As a consequence, the following DP function is used instead, defined for $0 \leq i \leq n$ as:

$$dp[i] = \text{smallest compressed size of the last } n - i \text{ bytes of the given text.}$$

This is equivalent to say that $dp[i]$ is the smallest compressed size of the suffix of the given text starting at position i (0-indexed). So the result for the whole text is in $dp[0]$. The base case is $dp[n] = 0$ as the suffix from position n is an empty string. The recurrence is defined for each other position as:

$$dp[i] = \min(2 + dp[i + 1], 1 + \min_{\substack{i < j \leq n \\ s[i:j] \in symbols}} dp[j]).$$

The two options to choose from are the same as in the previous formula, but candidate symbols to choose from must now begin at position i as we are now calculating the results for suffixes instead of prefixes of the given text.

This DP problem is a segmentation problem, where each segment is either a symbol, the escape code or a raw byte of the string that must follow the escape code. All segments have the same cost and we want to minimize the number of segments used to compress a given text. The first formula builds a bottom-up solution, while the second formula builds a top-down solution. Both should give the same result for the whole text as they are solving the same problem. The code in Figure 4.1 represents the python implementation of the DP function *st.buildDP* that fills both the DP and the *opt* tables in a single pass.

As the code shows, the DP table is filled from n to 0 in descending order, where for each position other than n the DP value is initialized with the corresponding value for the option of escaping the current byte. Then the inner loop iterates over the candidate symbols that begin with the current byte and match the next chunk of the text *data*. This code uses the *st.sIndex* table, whose construction was shown in Figure 3.2. So the iteration over the symbols is in decreasing order of their lengths. As a tie-breaker between symbols that yield the same DP value, this implementation chooses a symbol with the highest length (i.e. occurs first in the iteration), thus the use of ' $<$ ' instead of ' \leq ' in the if statement for updating the DP and *opt* tables. There is an exception to this case, when choosing to escape the byte also yields the smallest DP value, then no symbol will be chosen.

This approach requires performing a memory comparision of at most 8 bytes for every symbol that begins with a certain byte. To avoid that, a trie is used in the C++ implementation of the contributions [8] to store the symbols in addition to the *hashTab*

```
n = len(data) # data is the given text to encode, or to use for compressCount
self.dp = [0] * (n+1)
self.opt = [0] * n
for i in reversed(range(n)):
    self.opt[i] = data[i]
    self.dp[i] = self.dp[i+1] + 2
    # start is the index of first symbol beginning with the byte at i
    start = self.sIndex[data[i]]
    end = self.sIndex[data[i] + 1]
    for code in range(start, end):
        sym = self.symbols[code]
        L = len(sym)
        if (
            i + L <= len(data)
            and self.dp[i] > 1 + self.dp[i + L]
            and data[i:i + L] == sym
        ):
            self.dp[i] = 1 + self.dp[i + L]
            self.opt[i] = code
```

Figure 4.1: Python implementation of the DP function.

and *shortCodes* arrays. The trie is implemented as a vector of *TriNodes*, whose structure is shown in Figure 4.2.

```
struct TriNode {
    int symbolCode; // code of a symbol ending here, -1 if none
    int child[256]; // child indices, -1 if absent
    TriNode() : symbolCode(-1) {
        for (int i=0;i<256;i++) child[i] = -1;
    }
};
```

Figure 4.2: TrieNode structure

Each node stores the code of the symbol ending at that node and the indices for the children nodes, where the maximum number of children is 256 (an edge for every possible byte). The maximum number of nodes in the trie is $(8 * 255 + 1)$, as all symbols are disjoint in the worst case and there are 255 actual symbols at most, with an additional node for the root. This makes the trie memory overhead around 2MB in the extreme case.

Now finding candidate symbols can be done efficiently by traversing the trie and comparing one byte at each iteration instead of a whole symbol, with the possibility to break from the inner loop when there are no symbols beginning with a certain prefix. The inner loop of the current implementation is shown in Figure 4.3.

A similar traversal is used for adding each symbol into the trie, where instead of breaking when *node* == -1, a new node is created and the child entry is updated accordingly. Here, breaking is faster in practice than continuing the 8 full iterations without branching, as in the most cases no symbols will be found after the second or third iteration, due to the high number of symbols of lengths two and three in the symbol table.

After building the DP and *opt* tables, choosing the next symbol in *compressCount* or in the compression method can be done by a simple lookup in the *opt* table. The implementation wraps this operation in a function called *findBestSymbol*. Results of this DP approach on the compression factor (CF) will be discussed in Section 5.3.1.

4.2 3rd Counter

4.3 Symbol Pruning

```
// walk trie for real symbols (1..8 bytes)
int node = 0;
int limit = (int) min<size_t>(Symbol::maxLength, n - (size_t)i);
for (int off=0; off<limit; ++off) {
    u8 bb = data[i + off];
    node = trie[node].child[bb];
    if (node == -1) break;
    int code = trie[node].symbolCode;
    if (code != -1) {
        u32 L = (u32)(off + 1);
        u32 cost = 1u + dpCost[i + (int)L]; // real symbol always emits 1 byte
        if (cost <= bestCost) {
            bestCost = cost;
            bestCode = (u16) code;
        }
    }
}
```

Figure 4.3: Trie traversal in st.buildDP

5 Evaluation

5.1 Environment

5.2 Benchmarking Data

5.3 Results

5.3.1 DP approach

6 Discussion & Future Work

7 Conclusion

List of Figures

2.1	Initial max heap ordered by frequency.	6
2.2	Max heap after extracting the maximum element ⟨42, fs⟩.	6
2.3	Max heap after inserting the element ⟨38, fst⟩.	6
2.4	Example trie storing short strings. Terminal nodes are annotated with integer metadata.	7
3.1	FSST compression example [3]	11
3.2	FSST initial idea symbol table construction [3]	13
4.1	Python implementation of the DP function.	17
4.2	TrieNode structure	18
4.3	Trie traversal in st.buildDP	19

List of Tables

3.1 Comparison of different block-based compressors with the Silesia compression corpus [5], [6]	9
--	---

Bibliography

- [1] Peter A. Boncz, Martin L. Kersten, “Monet: An impressionist sketch of an advanced database system,” *Proc. IEEE BIWIT workshop, San Sebastian (Spain)*, pp. 240–251, 1995.
- [2] D. J. Abadi, S. R. Madden, and N. Hachem, “Column-stores vs. row-stores,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, L. V. S. Lakshmanan, R. T. Ng, and D. Shasha, Eds., New York, NY, USA: ACM, 2008, pp. 967–980. doi: 10.1145/1376616.1376712.
- [3] P. Boncz, T. Neumann, and V. Leis, “Fsst: Fast random access string compression,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2649–2661, 2020. doi: 10.14778/3407790.3407851.
- [4] Y. Collet, *lz4*, <https://github.com/lz4/lz4>, Original release: 2011-03-25. Accessed: 2026-01-10.
- [5] *facebook/zstd*, <https://github.com/facebook/zstd>, Original release: 2015-01-24. Accessed: 2026-01-10.
- [6] *Silesia compression corpus*, <https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia>, Visited: 2026-01-10.
- [7] P. Boncz, *fsst*, <https://github.com/cwida/fsst>, Original release: 2020-03-03. Accessed: 2026-01-10.
- [8] H. Chehaidar, *btrfsfst*, <https://github.com/Hedi-Chehaidar/btrfsfst>, Original release: 2025-12-07. Accessed: 2026-01-13.