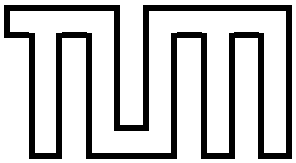


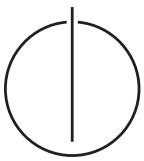
DEPARTMENT OF INFORMATICS

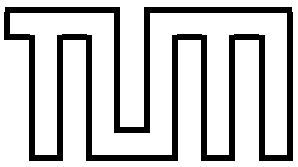
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Querying Compressed Data: LIKE Pattern Matching in the FSST Domain

Călin-George Pop





DEPARTMENT OF INFORMATICS

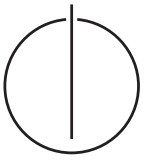
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Querying Compressed Data: LIKE Pattern Matching in the FSST Domain

Abfragen auf komprimierten Daten: LIKE-Textmusterabgleich in der FSST-Domäne

Author:	Călin-George Pop
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Adrian Riedl, M.Sc.
Submission Date:	December 18, 2025



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

Munich, May 26, 2026 Călin-George Pop

Acknowledgments

I would like to express my sincere gratitude to several individuals who have supported me throughout the process of writing this thesis.

First and foremost, I am deeply grateful to my advisor, Adrian Riedl, M.Sc., for his expert advice, valuable guidance and unwavering support from the initial concept of the thesis to the last bug, found a mere three days before submission. His insights were crucial in shaping this thesis. I also extend my sincere thanks to my supervisor, Prof. Dr. Thomas Neumann, for providing me with the opportunity to work on such an interesting and engaging topic.

Furthermore, I am grateful for the technical support I received. My thanks go to Maximilian Reif, Simon Ellmann, and Theresa Schmidkonz for their exceptional kindness in lending me a laptop for a month when my personal computer broke down, which allowed me to continue my work without any interruption.

This thesis also benefited immensely from the input of many others, especially my friends. I would like to specifically thank Vincent and Frank for proofreading it; their valuable suggestions and corrections significantly improved the quality and clarity of the final version of the thesis. A special mention goes to Răzvan for joining me at the Bad Omens concert in Paris; this much-needed break allowed me to return to my work with a refreshed and clear mind, which was instrumental in the polishing phase.

Finally, I owe my deepest gratitude to my parents and my brother. Your constant love, encouragement, and support have been the foundation of everything I have achieved. Without you, I would not be here today.

Abstract

Filtering compressed data efficiently poses a significant computational challenge, even when using compression algorithms that offer point access to columns. Currently, the state-of-the-art pattern matching algorithms on strings often require a costly decompression step, which is frequently the most expensive part of the matching process. By eliminating the need for decompression, we demonstrate that string matching runtime can be improved by up to 15x in extreme cases.

We restrict our attention to SQL `LIKE` expressions for simple text pattern matching and to datasets compressed using the FSST compression algorithm. To match all compressed strings corresponding to a given pattern, we build a deterministic finite automaton whose structure is determined by both the specified pattern and the symbols in the current symbol table. The final parsing process is achieved by traversing the automaton with the strings from the input dataset.

We describe the algorithm for constructing the corresponding deterministic finite automaton for each pattern type, taking into account the wildcards present in the pattern. Once the final automaton is built, we assess the benefits of compiling it in two different frameworks, taking into consideration the compilation speed.

Our results demonstrate that multiple factors influence the performance of the presented algorithm compared to the baseline. However, creating the automaton and then compiling proves worthwhile for the majority of workloads. Moreover, for workloads where our approach is significantly slower, we outline some immediate optimisations that can be applied to mitigate this effect.

Contents

Acknowledgments **v**

Abstract **vii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 State of the Art . . . 2
1.3 Structure . . . 3

2 Building Blocks **5**
2.1 SQL LIKE Keyword . . . 5
2.2 Fast Static Symbol Table (FSST) . . . 6
2.3 Finite Automata . . . 8
2.4 Single-Pattern String Matching . . . 9
2.4.1 Knuth-Morris-Pratt Algorithm . . . 9
2.4.2 Boyer-Moore Algorithm . . . 12
2.5 Multi-Pattern String Matching . . . 15

3 Single Pattern Matching **17**
3.1 End Match Automata . . . 17
3.2 Start Match Automata . . . 20
3.3 Middle Match Automata . . . 25
3.3.1 Initial Phase . . . 25
3.3.2 Linking-Starts Phase . . . 29
3.3.3 Creating Fallback States . . . 30

4 Underscore Pattern Matching **35**
4.1 Creating States for Underscore Symbols . . . 36
4.2 Start Match Automata . . . 37
4.3 End Match Automata . . . 41
4.4 Middle Match Automata . . . 44

5 Construction and Structuring of Automata **53**
5.1 Connecting Sub-Automata . . . 53
5.2 Level Assignment . . . 55

6 Benchmark **59**
6.1 Hardware Specification . . . 59
6.2 Datasets . . . 59

6.3 Approaches 60
6.3.1 Interpreted Approach 61
6.3.2 C++ Compiled Code 61
6.3.3 LLVM Compiled Approach 62
6.4 Results 65

7 Conclusion **69**
7.1 Empirical Factors Affecting Performance 69
7.2 Immediate Optimization Potential 70
7.3 Future Work and Outlook 70

List of Figures **73**

List of Tables **75**

Listings **77**

Bibliography **79**

1 Introduction

The current digital era is characterised by an unprecedented, exponential surge in the amount of data that is created; [Kna24] estimated that, between 2016 and 2021, the mean data size handled by companies grew 10 times, from 1.45 petabytes to 14.6 petabytes, and around 2.5 exabytes of data were approximated to be generated daily. Due to this enormous data size, compression algorithms, such as the Lempel-Ziv family of algorithms (for example, LZ77) or dictionary-based algorithms, have become indispensable for storage and transmission.

Moreover, the rapid advancement and adoption of Artificial Intelligence, as well as the consequent growth of Data Science methodologies, have intensified the requirement for high-speed access to massive datasets. Training complex deep learning architectures often necessitates mining vast repositories of historical, archived data, called cold data. Crucially, the efficiency of tasks like data preparation and data engineering hinges on the ability to quickly locate and verify patterns across terabytes of compressed data.

As a consequence, database systems and modern analytical platforms are constantly striving to achieve two often-conflicting goals: minimising the runtime of complex queries and achieving minimal disk storage requirements for housing massive volumes of user data. This creates immense pressure to employ practical compression algorithms, while not significantly hindering the performance of queries written by users. This thesis is based on the idea that there is an optimal solution that satisfies both demands.

1.1 Motivation

Data centres commonly use internal compression algorithms to significantly reduce the size of stored data. A major issue with this approach is that the necessary decompression step slows down data extraction significantly when analytical systems query the stored data. To overcome this performance bottleneck, it is crucial to enable direct data extraction in a compressed manner, thereby significantly reducing the amount of data sent over the network. A proposed solution is BtrBlocks [Kus+23], a transparent storage system that stores data in compressed formats known to the user. When data extraction is required, the compressed data is transmitted efficiently over the network, and the user performs the decompression if required.

For integer data types, not only is the decompression fast due to the small size in bytes of typical integer values (commonly up to 8 bytes), but most integer compression algorithms also allow for modifications that enable direct filtering on compressed elements [Spi+24]. By contrast, string decompression is often substantially more expensive because strings can grow arbitrarily large. Moreover, the pool of lightweight, general-purpose string compression algorithms suitable for database systems is scarce; this issue is further reinforced by the widespread use of strings as a default representation of data whose inherent type is not immediately evident [BNL20]. Such data types include IP addresses, which require 4 bytes in

their integer form but up to 15 bytes as strings, and UUIDs, which require 16 bytes in their integer form but up to 36 bytes as strings.

The performance of analytical query engines on large datasets is frequently bottlenecked by the CPU and I/O overhead required for data decompression; consequently, BtrBlocks focuses on algorithms that achieve fast decompression. For string compression, BtrBlocks primarily uses FSST [BNL20], as it has been observed to perform well across various datasets, both in terms of decompression speed and compression rate. Since integer decompression is fast, queries that depend on integer values are not significantly hindered by the decompression process. For user queries on string data, however, we must consider whether the costly decompression process is actually necessary and if it can be avoided. For equality predicates, since compressing is a deterministic process, decompression is avoided completely; our goal is to enable fast querying with `LIKE` SQL patterns on FSST-compressed data.

1.2 State of the Art

Most database systems lack native, automatic column-level compression for standard data types. For instance, PostgreSQL provides compression only through its TOAST (The Oversized-Attribute Storage Technique) mechanism [Pos25], which applies compression only to values exceeding 2 KB in size. Consequently, TOAST offers no practical benefit for typical string attributes in a dataset.

Conversely, MySQL allows for manual, column-wise compression through the use of the `COMPRESS` and `DECOMPRESS` functions. This functionality requires the explicit inclusion of the relevant compression library [MyS]. The most commonly used compression library is zlib, which employs a combination of Huffman Coding and the LZ77 algorithms. Consequently, for datasets containing short strings, the computational overhead associated with the compression process frequently exceeds the size of the uncompressed data, rendering the algorithm ineffective.

DuckDB offers multiple algorithms for encoding the data, including the aforementioned FSST algorithm for strings [Duc]. However, given the complexity of arbitrary pattern matching on FSST compressed data and the fact that its exact mechanism is not explicitly documented, knowing that the algorithm runs regardless of the string encoding, this thesis proceeds under the assumption that the data is decompressed just-in-time for pattern matching, and that the query plan generated is equivalent to the query plan presented on the left of Figure 1.1. Our goal is to ultimately introduce the `ENCODEDLIKE` filtering operator, optimising the query plan in the process, to the form illustrated on the right of Figure 1.1.

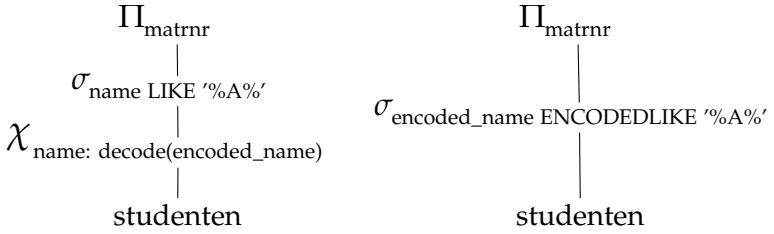


Figure 1.1: Query plans for pattern matching on compressed data (unoptimised on the left, optimised on the right)

1.3 Structure

The thesis is structured as follows: Chapter 2 lays the groundwork for our subsequent research by introducing the foundational concepts required for our study. Section 2.1 introduces the `LIKE` pattern in SQL, detailing the available wildcards and explaining how the text encoding influences its behaviour. Following this, we present FSST, the compression algorithm employed in this work, in Section 2.2. Section 2.3 then recapitulates basic finite automata theory. We conclude this chapter by examining the state-of-the-art string matching techniques. Sections 2.4 and 2.5 provide a focused overview of the leading algorithms for single-pattern and multiple-pattern matching, respectively.

Chapter 3 details the algorithms available for constructing finite automata designed to match strings containing a single sub-pattern, without underscore wildcards. We approach this topic systematically, starting with an explanation of the construction algorithm for end patterns in Section 3.1, followed by the method to construct automata for start patterns in Section 3.2. We finalise the chapter with a discussion detailing the more complex algorithm for generating automata matching middle sub-patterns (Section 3.3).

Chapter 4 extends the algorithms from the previous chapter to handle single sub-patterns that contain the underscore wildcard. We begin, in Section 4.1, by explaining the integration of the underscore symbols into the finite automata construction. Building on this foundation, the chapter then presents the algorithmic extensions necessary for creating automata based on pattern type. Section 4.2 covers start automata, and Section 4.3 details end automata. We conclude the chapter with the algorithm constructing middle automata for such sub-patterns in Section 4.4.

Chapter 5 addresses the task of matching patterns containing multiple sub-patterns, thereby integrating the single-pattern algorithms developed in the previous chapters. To achieve this, we first present a method for connecting the individual sub-automata into a unified structure in Section 5.1. Section 5.2 introduces an optimisation using an early stopping mechanism, which we achieve by assigning a level attribute to each state within the final automaton. Finally, we provide the basic parsing functionality that enables matching arbitrary patterns against the input text.

Chapter 6 details the experimental setup and presents the empirical results of this research. We start by outlining the hardware specifications of the machine used for benchmarking in Section 6.1, followed by a description of the datasets employed in Section 6.2. Subsequently, Section 6.3 introduces the different approaches benchmarked, which are based on the finite automaton previously built. This chapter concludes with a comprehensive presentation and analysis of the benchmarking results in Section 6.4.

Chapter 7 concludes this research by summarising the findings and outlining directions for future research. Section 7.1 begins by formally analysing the variables of a dataset empirically correlated to the performance of the previously presented algorithms. Following this analysis, Section 7.2 outlines specific ideas for significantly improving the algorithm's efficiency; finally, in Section 7.3, we present broader research directions.

2 Building Blocks

This chapter presents the fundamental concepts and background information necessary to comprehend the remainder of the thesis. We begin by explaining how the SQL `LIKE` expression works, including its features and practical applications, taking data encoding into account, in Section 2.1. In Section 2.2, we describe the FSST compression algorithm, before quickly reviewing the basic theory of finite automata in Section 2.3. Finally, we present the state-of-the-art algorithms for checking whether a single specific string appears in a text (Section 2.4) and also outline the existing approaches that simultaneously check for the existence of multiple patterns within a string (Section 2.5).

2.1 SQL LIKE Keyword

According to the SQL-92 standard [Iso], the `LIKE` keyword is used to determine whether a string matches or not a given pattern (also a character string). The general syntax of a `LIKE` predicate is `<string> LIKE <pattern> [ESCAPE <escapeCharacter>]`; by default, the escape character is '\'. Two special symbols are used as wildcards: '%' means "any character, any number of times" (i.e., '.*' from regular expressions), and '_' means "any single character" (i.e., '.' from regular expressions); in order to match the actual '%' and '_' characters, we escape them using the escape character; to match the escape character, we use the escape character twice.

For example, for the pattern "red%_green_%blue", the matching strings start with the word "red", end with the word "blue", and contain the word "green" somewhere in the middle of the word. Additionally, there must be at least one character in between the "red" and "green", and at least one character in between "green" and "blue". We observe that any sequence of consecutive '%' wildcards is redundant, and it is equivalent to matching a single '%' wildcard.

Moreover, the two wildcards are commutative: for a sequence "$s_0 s_1 \dots s_{m+n-1}$" satisfying $s_i \in \{\text{'\%'}, \text{'\_'}\}, \forall i \in \{0, 1, \dots, m+n-1\}, m \geq 1$ the count of '%' and n the count of '_', and a permutation σ of the indexes, the sequences "$s_0 s_1 \dots s_{m+n-1}$" and "$s_{\sigma(0)} s_{\sigma(1)} \dots s_{\sigma(m+n-1)}$" are equivalent and interchangeable inside of any `LIKE` pattern; both are interchangeable with "%_ _ ... _ _" or "_ _ ... _ _%", where both have n underscore wildcards. Therefore, the previously presented pattern is equivalent to "red_%green_%blue", "red%_green%_blue" and "red_%green%_blue".

In the case of ASCII-encoded data, the intended behaviour is simple, as a character is equivalent to one byte. For other encodings, however, `LIKE` matching is more complicated due to the multitude of ways a character is encoded; for example, the special character Ö is represented by at least three different sequences of bytes:

I. `0xC3 0x96` (UTF-8)
II. `0xD6` (Latin-1, ISO-8859-1)
III. `0x4F 0xCC 0x88` (Unicode, Normalization Form D)

To ensure consistency within the database, a single encoding should be maintained per string column; then, matching "%Ö%" is equivalent to finding the bytes presented above, for the given encoding. However, PostgreSQL exhibits behaviour that can lead to conflicting string encodings within a single database. This was demonstrated by first establishing a database with UTF-8 encoding as the default text encoding, in which we created and then populated a table with UTF-8 strings. Subsequently, we added a trigger to perform Normalisation Form D of Unicode on all string columns during row insertion; re-inserting the previous strings in the table leads to inconsistent string encodings within the string columns. We present such a table below, along with the byte representation of the string column. We observe that `LIKE` matching now exhibits strange behaviour: the Unicode encoded string matches '%O%' (since its first byte is `0x4F`, the ASCII code of 'O'), but not '%Ö%', whereas the UTF-8 encoded string matches '%Ö%', but not '%O%'. Without examining the internals (i.e., the byte sequence), this appears to be inconsistent behaviour within the same table.

```
SELECT id, german_word, ENCODE(german_word::bytea, 'hex') AS byte_hex FROM test;
```

id	german_word	byte_hex
1	Öl	0xC3 0x96 0x6C
2	Überraschung	0xC3 0x9C 0x62 0x65 0x72 0x72 0x61 0x73 0x63 0x68 0x75 0x6E 0x67
3	Schön	0x53 0x63 0x68 0x6F 0x6E
4	Öl	0x4F 0xCC 0x88 0x6C
5	Überraschung	0x55 0xC3 0xBC 0x62 0x65 0x72 0x72 0x61 0x73 0x63 0x68 0x75 0x6E 0x67
6	Schön	0x53 0x63 0x68 0x6F 0xCC 0x88 0x6E

(6 rows)

```
SELECT id, german_word, ENCODE(german_word::bytea, 'hex') AS byte_hex FROM test WHERE
                                    german_word LIKE '%O%';
```

id	german_word	byte_hex
4	Öl	0x4F 0xCC 0x88 0x6C

(1 row)

```
SELECT id, german_word, ENCODE(german_word::bytea, 'hex') AS byte_hex FROM test WHERE
                                    german_word LIKE '%Ö%';
```

id	german_word	byte_hex
1	Öl	0xC3 0x96 0x6C

(1 row)

2.2 Fast Static Symbol Table (FSST)

For database systems, around half of the columns are estimated to have string-like data types [Vog+18], which are often used for data whose actual type is unclear. However, file compression algorithms like LZ4 are not fit for compressing these strings due to the overhead introduced, often higher than the actual string length, and dictionary encoding fails when a string value appears only once on average; as a consequence, FSST [BNL20] was introduced for a light-weight compression algorithm, well-suited for small strings.

FSST replaces frequently occurring, at most 8-byte long substrings with 1-byte codes. When decoding a compressed string, each byte is decoded iteratively and concatenated to obtain the final result. Knowing that the string data we aim to compress is always cold, the symbol table is created once and never modified. For a symbol to be stored on 1 byte, the symbol table stores at most 255 different symbols; from now on, by $\{x_0, x_1, \ldots, x_{n-1}\}$ we mean the

dataset (uncompressed)
SC␣FC␣FCSB␣SA
SC␣Freiburg
Real␣Madrid␣CF
Atletico␣de␣Madrid␣SAD
FC␣Augsburg
...

symbol table

	symbol	length
0	SC␣	3
1	FC	2
2	␣SA	3
3	Real␣	5
4	␣CF	3
5	␣de␣	4
6	Madrid	6
7	Atletico	8
8	burg	4
9	Augs	4
10	Frei	4
11	␣	1
...		

dataset (compressed)
{0, 1, 11, 1, 255, 83, 255, 66, 3}
{0, 10, 8}
{3, 6, 4}
{7, 5, 6, 2, 255, 68}
{1, 11, 9, 8}
...

Figure 2.1: Example of an uncompressed dataset, an arbitrary symbol table and the compressed dataset for the presented symbol table

sequence of those bytes, in the presented order. We reserve the byte `255` for escaping a byte (i.e., the sequence {`255, x`} is decoded into the raw byte value `x`, rather than the `x`-th symbol). We present an example uncompressed dataset, along with a possible symbol table and the compressed version of the dataset in Figure 2.1.

For constructing the optimal symbol table given a dataset, we note that counting the appearances of each substring of lengths from 1 to 8 is infeasible, as the number of appearances of overlapping strings is not independent of each other. Moreover, greedily picking the longest symbol has been observed not to maximise compression efficiency, and testing all symbol tables is impractical, given that checking the compression rate of a single symbol table is $\mathcal{O}(N)$, where N is the file size. As a consequence, an iterative, bottom-up algorithm is employed, which yields an acceptable symbol table in terms of compression rate very quickly.

Initially, the symbol table is empty; at each iteration, the corpus is first iterated over and encoded using the current symbol table; this part computes the compression factor in order to analyse the overall quality of the symbol table and, at the same time, counts the number of appearances of each symbol in the compressed corpus. Afterwards, these counts are used to construct the next symbol table by selecting the symbols that have the highest apparent gains. At any step, all symbols from the previous step are taken into consideration, along with all possible concatenations of two symbols, all one-byte symbols and all one-byte extensions of all previous symbols.

Given that the speed of constructing the symbol depends heavily on the corpus at hand, a sample of $16KB$ is used rather than the entire dataset; the observed compression ratios are comparable to those achieved when using the entire dataset. Furthermore, the number of iterations for which the algorithm runs is limited to 5, as empirically it has been observed to yield satisfactory results. Data compression should be deterministic; as such, we present in Property 2.1 an important feature of the compression algorithm.

Property 2.1. Given a symbol table, encoding a string "$s_0 s_1 \dots s_{k-1}$" involves greedily picking

the longest symbol $s = "s_0 s_1 \dots s_{l-1}"$, completely contained within the string ($l \leq k$), then re-applying this algorithm for the rest of the uncompressed string $"s_l s_{l+1} \dots s_{k-1}"$. For example, for the previously presented symbol table, the string "␣de␣" is encoded as $\{5\}$ rather than $\{11, 255, 100, 255, 101, 11\}$; to handle such sequences, we provide the following definition.

Definition 2.1. A sequence of symbols $S = \{sym_0, sym_1, \dots, sym_{m-1}\}$ is valid if there is a string $s_0 s_1 \dots s_{l-1}$ that compresses to S using the FSST Algorithm. For example, the previously presented sequence $\{11, 255, 100, 255, 101, 11\}$ is not valid.

To further improve encoding speed, a lossy perfect hash table `hashTab[4096]` is implemented to facilitate finding the longest symbol starting with the string to be compressed; the size is chosen to fit the table in the L1 cache. The hash function depends only on the first three bytes of the symbol; as such, we know the following property of symbols:

Property 2.2. No two symbols have the same three-byte prefix; for example, for the previous example, no other symbols start with the prefixes "Rea", "Mad", "Atl", and so on.

To handle potential clashes, only the symbol with the highest apparent gain is kept for a selected prefix, while the other symbols are replaced by the next candidates, ranked by apparent gain, that have not been picked yet. Moreover, to simplify an algorithm from Chapter 3, when building a symbol table, we ensure the following holds:

Property 2.3. The last byte of the i-th indexed symbol (with $i < 255$) is different from i; for example, the 65-th symbol does not end with the character 'A'.

When every selected symbol ends with the same final character, we drop the symbol with the lowest apparent gain and replace it with the next candidate whose final character differs. Afterwards, we simply resolve any remaining conflicts through symbol swaps until the condition is satisfied.

In addition to the perfect hash table, a 2-dimensional array `shortCodes[256][256]` is stored, containing information about one-byte and two-byte symbols. Initially, `shortCodes[x][y] = 511` represents free slots. Afterwards, for all 2-byte symbols $sym_i = "xy"$, `shortCodes[x][y] = i`; finally, for all 1-byte symbols $sym_j = "x"$, all empty slots `shortCodes[x][*]` have their values set to j; using this array, finding the longest symbol to match the string happens in one step.

2.3 Finite Automata

We begin this section by recalling the definition of finite automata [BT14]:

Definition 2.2. A non-deterministic finite automaton is a quintuple $\mathcal{M} = \{\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F}\}$, where $\mathcal{Q}$ is the finite non-empty set of states, Σ is the finite non-empty alphabet, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{P}(\mathcal{Q})$ is the transition function, $\mathcal{I}$ is the set of initial states, and $\mathcal{F}$ is the set of final states.

However, parsing a non-deterministic finite automaton is not only complicated, but also slow: due to $\delta(q, c)$ taking arbitrarily many values, each parsing step has multiple current states; to simplify parsing, deterministic finite automata are introduced:

Definition 2.3. A Deterministic Finite Automaton (DFA) is a quintuple $\mathcal{M} = \{\mathcal{Q}, \Sigma, \delta, S, \mathcal{F}\}$, where $\mathcal{Q}, \Sigma, \mathcal{F}$ are the same as in the previous definition, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the transition function, and $S \in \mathcal{Q}$ is the start state. To simplify parsing even further, there is a state ε indicating an erroneous state, and, for each state q, there is a default state q_{def} to which q transitions when a transition is not explicitly defined; unless specified otherwise, $q_{def} = \varepsilon$. When drawing an automaton, the colour of a state indicates, by looking at the legend, its default transition state.

The definition above implies that, for any state $q \in \mathcal{Q}$ and any symbol of the alphabet $c \in \Sigma$, there is a unique state to arrive at when transitioning from q through c, with the resulting state being either a proper state of the automaton or the designated error state. From now on, questioning whether a state transitions through a symbol or not is equivalent to being able to transition directly through that symbol (i.e., without default transitions).

Definition 2.4. A DFA is called a Deterministic Acyclic Finite State Automaton if the transition function δ is acyclic; for any $q \in \mathcal{Q}, q \neq \varepsilon$ and for any sequence of characters $s_1, s_2, \ldots, s_n \in \Sigma$, $q^{(n)} \neq q$, where $q^{(0)} = q$ and $q^{(k+1)} = \delta(q^{(k)}, s_k)$ [Dac+00].

2.4 Single-Pattern String Matching

For this section, we assume that we have a string $s = "s_0 s_1 \ldots s_{n-1}"$ of length n and a single pattern $p = "p_0 p_1 \ldots p_{m-1}"$ of length $m > 0$. Our goal is to determine if the string s contains the pattern p. Subsection 2.4.1 introduces the Knuth–Morris–Pratt (KMP) algorithm, the first efficient algorithm for string matching; subsequently, Subsection 2.4.2 details the widely adopted Boyer–Moore algorithm, an alternative approach to KMP.

2.4.1 Knuth-Morris-Pratt Algorithm

The KMP algorithm [KMP77] aims to avoid traversing the same state in a string multiple times by pre-processing the pattern before running the algorithm. For the pre-processing phase, we require the following definitions:

Definition 2.5. Let $x = "x_0 x_1 \ldots x_{l-1}"$ be a string.
Any substring y of x of the form $"x_0 x_1 \ldots x_{b-1}"$ is called a *prefix of* x; when $y \neq x$ (i.e., $b < l$), y is called a *proper prefix of* x.
Any substring y of x of the form $"x_b x_{b+1} \ldots x_{l-1}"$ is called a *suffix of* x; when $y \neq x$ (i.e., $b > 0$), y is called a *proper suffix of* x.
A substring $y = "x_0 x_1 \ldots x_{b-1}" = "x_{l-b} x_{l-b+1} \ldots x_{l-1}"$ that is both a proper prefix and a proper suffix of x (i.e., $b < l$) is called a *border* of x.

The pre-processing involves creating an array `lps` of size $m + 1$ of shifts: firstly, `lps[0] := -1`; for $i > 0$, set `lps[i]` to the length of the largest border of $"p_0 p_1 \ldots p_{i-1}"$, and to 0 if no such border exists [Rie23]; more intuitively, this is equivalent to finding the minimal $j > 0$ for which $"p_0 p_1 \ldots p_{i-j-1}" = "p_j p_{j+1} \ldots p_{i-1}"$, and then doing `lps[i] := i - j`, or `lps[i] := 0` if no such j exists. We present in Table 2.1 the pre-processing array for the pattern $p = "abcabcacab"$.

i	0	1	2	3	4	5	6	7	8	9	10
s[i]	a	b	c	a	b	c	a	c	a	b	END
longest border	∅	∅	∅	∅	a	ab	abc	abca	∅	a	ab
j	∅	∅	∅	∅	3	3	3	3	∅	8	8
lps[i]	-1	0	0	0	1	2	3	4	0	1	2
compressed_lps[i]	-1	0	0	-1	0	0	-1	4	-1	0	2

Table 2.1: Preprocessing data for the pattern "abcabcacab"

The key to the KMP algorithm is avoiding backtracking within the string pattern: the pre-processed array `lps[i]` contains the furthest point in the pattern we can shift to, given that we have just encountered a mismatch. For $i \in \{1, 2, \ldots, m\}$, we define the sequences $\texttt{lps}^{(k)}\texttt{[i]}$ as follows: $\texttt{lps}^{(0)}\texttt{[i]} = \texttt{lps[i]}$, and then, iteratively, $\texttt{lps}^{(k+1)}\texttt{[i]} = \texttt{lps}[\texttt{lps}^{(k)}\texttt{[i]}]$, until $\texttt{lps}^{(k)}\texttt{[i]} = 0$. When running into a mismatch at the i-th character in the pattern, we retry to match, iteratively, the next character of the text with the $\texttt{lps}^{(k)}\texttt{[i]}$-th entry of the pattern, for $k = 0, 1, \ldots$, until we either find a match or reach the start of the pattern. We present the original implementation of the KMP parsing approach [Rie23] below.

```
bool parseKMP(string, pattern)
    lps = preprocess(pattern);
    patIdx = 0, strIdx = 0;
    while (strIdx < string.size())
        while (patIdx != 0 and pattern[patIdx] != string[strIdx])
            patIdx = lps[patIdx];
        if (pattern[patIdx] == string[strIdx])
            if (patIdx == pattern.size() - 1)
                return true;
            ++patIdx;
        ++strIdx;
    return false;
```

To avoid having two different while loops, we can implement the parsing in a single loop. Using the previously built `lps` table for the pattern "abcabcacab", as presented in Table 2.2, we show how to search for this pattern through a string using KMP. At the fifth step of the search, we observe a problem of the current implementation, called the "bouncing ball" problem [Gus97]: after finishing matching "abc", if we have a mismatch (i.e., the character following "abc" is not 'a'), the KMP tries matching the pattern "a" on that exact same character, which we know that fails.

To fix this, `compressed_lps[i]` stores instead the length of the longest border of the prefix "$p_0 p_1 \ldots p_{i-1}$" if and only if the character following the border (i.e., p_i) is different than the one causing the mismatch; when doing so, the matching algorithm is also adapted to account for possible -1 indexes. Formally, the `compressed_lps` array is given by

$$\texttt{compressed\_lps[i]} = \begin{cases} -1, & \text{if } i = 0 \\ \texttt{compressed\_lps}[\texttt{lps[i]}], & \text{if } i > 0 \text{ and } p_i = p_{\texttt{lps[i]}} \\ \texttt{lps[i]}, & \text{otherwise;} \end{cases}$$

for the pattern "abcabcacab", we have also computed this value in Table 2.1.

strIdx	patIdx	b a b c b a b c a b c a a b c a b c a b c a c a b c	
0	0	a b c a b c a c a b	mismatch; `++strIdx`
1	0	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
2	1	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
3	2	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
4	3	a b c a b c a c a b	mismatch; `patIdx` ← `lps[3] = 0`
	0	a b c a b c a c a b	mismatch; `++strIdx`
5	0	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
6	1	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
7	2	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
8	3	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
9	4	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
10	5	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
11	6	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
12	7	a b c a b c a c a b	mismatch; `patIdx` ← `lps[7] = 4`
	4	a b c a b c a c a b	mismatch; `patIdx` ← `lps[4] = 1`
	1	a b c a b c a c a b	mismatch; `patIdx` ← `lps[4] = 0`
	0	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
13	1	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
14	2	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
15	3	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
16	4	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
17	5	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
18	6	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
19	7	a b c a b c a c a b	mismatch; `patIdx` ← `lps[7] = 4`
	4	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
20	5	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
21	6	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
22	7	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
23	8	a b c a b c a c a b	match; `++strIdx`, `++patIdx`
24	9	a b c a b c a c a b	match; return `true`

Table 2.2: KMP search for the pattern "abcabcacab" in the string "babcbabcabcaabcabcabcacabc"

The KMP approach can be easily adapted into a deterministic finite automaton for a single sub-pattern [Rie23]. Initially, for the sub-pattern $p = "p_0 p_1 \dots p_{m-1}"$ containing no wildcards, we create one state for each character: q_0 acts as the start state, then $\delta(q_0, p_0) = q_1$, $\delta(q_1, p_1) = q_2, \dots, \delta(q_{m-1}, p_{m-1}) = q_m$, and q_m is the end state; the default transition of all states from this automaton is the start state q_0; we draw the initial, non-deterministic finite automaton in Figure 2.2.

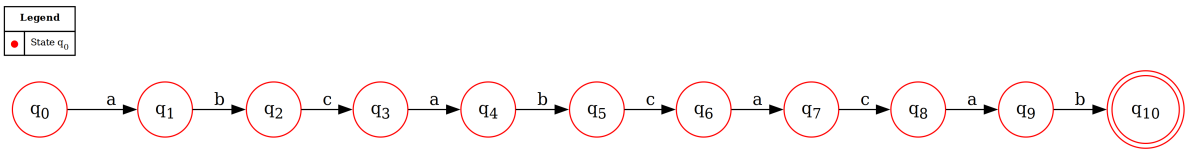


Figure 2.2: Non-deterministic finite automaton for the pattern "`abcabcacab`"

To determinize this non-deterministic finite automaton, we first distinguish between forward transitions and other transitions. During the algorithm, we have two pointers: `qCurrent`, pointing to the current state, and `qLps`, pointing to the trailing state before `qCurrent`; initially, `qCurrent` is set to the state q_1 following q_0, and `qLps` is set to the start state q_0. At each iteration step, we copy all new transitions of `qLps` into `qCurrent`, regardless of the direction; then, `qCurrent` = $\delta_{forward}$(`qCurrent`, c) (its unique forward transition), and `qLps` is moved forward as follows: if `qLps` transitions through c, `qLps` = δ(`qLps`, c); otherwise, `qLps` = q_0. We present

in Figure 2.3 the determinized version of the automaton from Figure 2.2.

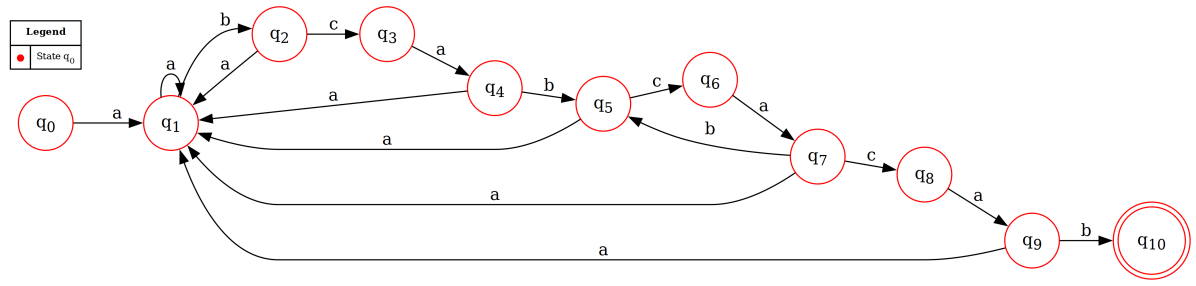


Figure 2.3: Determinized finite automaton for the pattern "`abcabcacab`"

2.4.2 Boyer-Moore Algorithm

The KMP algorithm processes the string one character at a time. Therefore, the complexity of the algorithm is $\mathcal{O}(n)$ in both the worst case and on average, where n is the length of the string checked. The Boyer-Moore (BM) algorithm [BM77] aims to significantly improve the best-case runtime complexity, albeit at the expense of the worst-case complexity. The general idea is to adjust the index `startIdx` at which we begin checking for a match within the string, and when a mismatch is encountered, shift `startIdx` according to certain rules. For the proper functioning of those rules, the pattern is traversed backwards, so `patIdx` is initially set to $m-1$ and, when running into a matching character, `patIdx` is decreased.

The MP algorithm employs two rules: the Bad Character Rule and the Good Suffix Rule, both of which occur only when a mismatch occurs [Hor80; Ryt80].

Bad Character Rule. The Bad Character Rule (BCR) is based upon the following two observations: firstly, if `string[startIdx + patIdx]` does not appear in the pattern, then the earliest a match may start is from the (`startIdx + patIdx + 1`)th character; as a consequence,

$$\begin{aligned}&\texttt{startIdx}^{(new)} = \texttt{startIdx}^{(old)} + \texttt{patIdx}^{(old)} + 1, \text{ and } \texttt{patIdx}^{(new)} = m-1 \implies \\ &\texttt{startIdx}^{(new)} + \texttt{patIdx}^{(new)} = \texttt{startIdx}^{(old)} + \texttt{patIdx}^{(old)} + m.\end{aligned}$$

Moreover, if the last occurrence of the character `c := string[startIdx + patIdx]` in `pattern` is $\delta_{bad}(\texttt{c})$ characters from the end of the pattern, we shift `startIdx` to align the character `c` in the string with the character `c` in the pattern as follows:

$$\texttt{startIdx}^{(new)} + m - 1 - \delta_{bad}(\texttt{c}) = \texttt{startIdx}^{(old)} + \texttt{patIdx}^{(old)},$$

given that the last occurrence of `c` is $\delta_{bad}(\texttt{c})$ characters away from the end, and the end is at the $(m-1)$th character; for $\texttt{patIdx}^{(new)}$, we start from the end of the pattern to check if we created any mismatches by shifting, so $\texttt{patIdx}^{(new)} = m-1$. Therefore,

$$\texttt{startIdx}^{(new)} + \texttt{patIdx}^{(new)} = \texttt{startIdx}^{(old)} + \texttt{patIdx}^{(old)} + \delta_{bad}(\texttt{c}).$$

Denoting `strIdx := startIdx + patIdx`, when mismatching on the character `c`, we set

$$\texttt{strIdx}^{(new)} = \begin{cases} \texttt{strIdx}^{(old)} + m, & \text{if c is not in the pattern,} \\ \texttt{strIdx}^{(old)} + \delta_{bad}(\texttt{c}), & \text{if c is in the pattern.} \end{cases}$$

To avoid branching, for the characters c in the alphabet that do not appear in the pattern, we set $\delta_{bad}(\texttt{c}) = m$. However, we observe that

$$\texttt{startIdx}^{(new)} = \texttt{startIdx}^{(old)} + \texttt{patIdx}^{(old)} + \delta_{bad}(\texttt{c}) - (m-1),$$

so, if $\texttt{patIdx}^{(old)} + \delta_{bad}(\texttt{c}) < m-1$, $\texttt{startIdx}^{(new)} < \texttt{startIdx}^{(old)}$, effectively starting earlier in the string; this is called a *negative shift*. We present below in Table 2.3 [Rie23] an example in which negative shift causes the matching function not to end, using that `strIdx = startIdx + patIdx` and $\delta_{bad}(a) = 1$. To avoid this behaviour, we require that `startIdx` is increasing when a shift is performed.

startIdx	patIdx	strIdx	a	a	a	a	b	a	b	
0	3	3	a	b	a	b				mismatch; strIdx+ = $\delta_{bad}(a)$
1	3	4		a	b	a	b			match; --patIdx
1	2	3		a	b	a	b			match; --patIdx
1	1	2		a	b	a	b			mismatch; strIdx+ = $\delta_{bad}(a)$
0	3	3	a	b	a	b				mismatch; strIdx+ = $\delta_{bad}(a)$
...										

Table 2.3: Bad Character Rule for searching the pattern "abab" in the string "aaaabab"

Good Suffix Rule. For the Good Suffix Rule (GSR), we start by denoting $j - i :=$ `startIdx`, $i :=$ `patIdx` and $j :=$ `strIdx`. We require to have a match "$s_{j+1}s_{j+2}\dots s_{m+(j-i)-1}$" = "$p_{i+1}p_{i+2}\dots p_{m-1}$", and there is a mismatch on the current character (i.e., $p_i \neq s_j$). Our goal is to use the known suffix "$p_{i+1}p_{i+2}\dots p_{m-1}$" (called the good suffix) to shift the pattern. Originally, s_{j+1} is aligned with p_{i+1}, s_{j+2} is aligned with $p_{i+2}, \dots, s_{m+(j-i)-1}$ is aligned with p_{m-1}; we split the good suffix shift in three different cases:

I. we align the entire good suffix with another non-prefix substring of the pattern, i.e. there is some k, $1 \leq k < i$, such that "$p_{k+1}p_{k+2}\dots p_{k+m-i-1}$" = "$p_{i+1}p_{i+2}\dots p_{m-1}$" and $p_k \neq p_i$; pick the largest k which satisfies this property. After shifting, we align s_{j+1} with $p_{k+1} = p_{i+1}$, s_{j+2} with $p_{k+2} = p_{i+2}, \dots$, and $s_{m+(j-i)-1}$ with $p_{k+m-i-1} = p_{m-1}$; this aligns p_0 with s_{j-k}, so

$$\begin{aligned}\texttt{startIdx}^{(new)} &= j - k = \texttt{strIdx}^{(old)} - k \implies \\ \texttt{strIdx}^{(new)} &= \texttt{startIdx}^{(new)} + (m-1) = \texttt{strIdx}^{(old)} + (m-1-k);\end{aligned}$$

as a consequence, in this case, $\delta_{good}(i) = m - 1 - k$, where k depends on the index i;

II. if no such k exists, then we align the prefix of the pattern with the good suffix, equivalent to finding the maximal $l \leq m - i - 1$ such that "$p_0p_1\dots p_{l-1}$" = "$p_{m-l}p_{m-l+1}\dots p_{m-1}$", if such an l exists. After shifting, we align $s_{m+(j-i)-1}$ with $p_{l-1} = p_{m-1}$, $s_{m+(j-i)-2}$ with $p_{l-2} = p_{m-2}$, and so on; $p_0 = p_{m-l}$ is then aligned with $s_{m+(j-i)-l}$, so

$$\begin{aligned}\texttt{startIdx}^{(new)} &= m + (j-i) - l = \texttt{strIdx}^{(old)} + (m-l) - i \implies \\ \texttt{strIdx}^{(new)} &= \texttt{startIdx}^{(new)} + (m-1) = \texttt{strIdx}^{(old)} + (m-l) - i + (m-1) \\ &= \texttt{strIdx}^{(old)} + (2\cdot m - i - l - 1);\end{aligned}$$

as such, here, $\delta_{good}(i) = 2 \cdot m - i - l - 1$, where l depends on the index i;

	A	B	C
$\delta_{bad}(c)$	0	1	3

	0	1	2	3	4	5	6
$\delta_{good}(i)$	10	9	8	7	8	3	1

startIdx	patIdx	strIdx	A B A B C A B A C A B A B	
0	6	6	A B A C A B A	mismatch; `strIdx += max(`$\delta_{bad}(B), \delta_{good}(6)$`)`
1	6	7	A B A C A B A	aligned 5th character using BCR
	6	7	A B A C A B A	match; `--patIdx, --strIdx`
	5	6	A B A C A B A	match; `--patIdx, --strIdx`
	4	5	A B A C A B A	match; `--patIdx, --strIdx`
	3	4	A B A C A B A	match; `--patIdx, --strIdx`
	2	3	A B A C A B A	mismatch; `strIdx += max(`$\delta_{bad}(B), \delta_{good}(2)$`)`
5	6	11	A B A C A B A	aligned first 3 characters with GSR
	6	11	A B A C A B A	match; `--patIdx, --strIdx`
	5	10	A B A C A B A	match; `--patIdx, --strIdx`
	4	9	A B A C A B A	match; `--patIdx, --strIdx`
	3	8	A B A C A B A	match; `--patIdx, --strIdx`
	2	7	A B A C A B A	match; `--patIdx, --strIdx`
	1	6	A B A C A B A	match; `--patIdx, --strIdx`
	0	5	A B A C A B A	match; `--patIdx, --strIdx`
	-1	4	A B A C A B A	return `true`

Table 2.4: BM search for pattern "ABACABA" in the string "ABABCABACABAB"

III. if no such l exists, then we skip the currently matched suffix, and

$$\texttt{startIdx}^{(new)} = m + (j - i) = \texttt{strIdx}^{(old)} + (m - i) \implies$$
$$\texttt{strIdx}^{(new)} = \texttt{startIdx}^{(new)} + m - 1 = \texttt{strIdx}^{(old)} + 2 \cdot m - i - 1,$$

therefore, $\delta_{good}(i) = 2 \cdot m - i - 1$ in this case. We notice that this is the previously obtained formula in (II) with $l = 0$.

By performing the maximum shift at each mismatch between the good suffix shift and the bad character shift, given that the good suffix shift is always positive, we no longer perform negative shifts; we present the parsing algorithm below.

```
bool parseBM(string, pattern)
    deltaBad = preprocessBad(pattern);
    deltaGood = preprocessGood(pattern);
    strIdx = pattern.size() - 1;
    while (strIdx < string.size())
        patIdx = pattern.size() - 1;
        while (patIdx >= 0 and pattern[patIdx] == string[strIdx])
            --patIdx;
            --strIdx;
        if (patIdx == -1)
            return true;
        strIdx += max(deltaBad(string[strIdx]), deltaGood(patIdx));
    return false;
```

We present an example of running the algorithm in Table 2.3; we notice that, during the algorithm, only three characters are candidates for starting the pattern there. The best-case complexity of the algorithm is $\Omega(n/m)$, while the worst-case complexity is $\mathcal{O}(m \cdot n)$, where m is the length of the pattern and n is the length of the string checked.

2.5 Multi-Pattern String Matching

For the problem we want to solve, neither KMP nor BM is applicable, as both algorithms check for equality with a single string, whereas in our case, we have multiple strings that we accept. For example, for the symbol table in Figure 2.1, the pattern "%␣%" is matched by the uncompressed version of a string if and only if its compressed form contains one of the symbols $0, 2, 3, 4, 5$ or 11, not preceded by an escape byte. For this, we require an algorithm that matches multiple possible strings simultaneously. The simplest algorithm achieving this is the Aho-Corasick Algorithm [AC75], which finds all the different occurrences of words within a string that appear in an initial array, called a dictionary.

Initially, a prefix tree (i.e., a trie) is created from all the words in the dictionary, which functions as a deterministic finite automaton. This is achieved by creating an initial state to start from, called `qStart`, having an empty prefix, and, for all words "$w_0w_1 \dots w_{n-1}$" in the dictionary, ensure that there are automaton states with all the prefixes "w_0", "w_0w_1", ..., "$w_0w_1 \dots w_{n-1}$".

Listing 2.1: Initialise the prefix tree given a dictionary of words

```
State* initialisePrefixTree(dictionary)
    qStart = createState();
    qStart.defaultTransition = qStart;
    for (word: dictionary)
        qCurrent = qStart;
        for (i = 0; i < word.size(); ++i)
            // qCurrent currently has the prefix "word[0] word[1] ... word[i -1]"
            // ensure there is a state with prefix "word[0] word[1] ... word[i]"
            if (not qCurrent.canTransition(word[i]))
                qCurrent.addTransition(word[i], createState());
            // transition to the state with prefix "word[0] word[1] ... word[i]"
            qCurrent = qCurrent.transition(word[i]);
    return qStart;
```

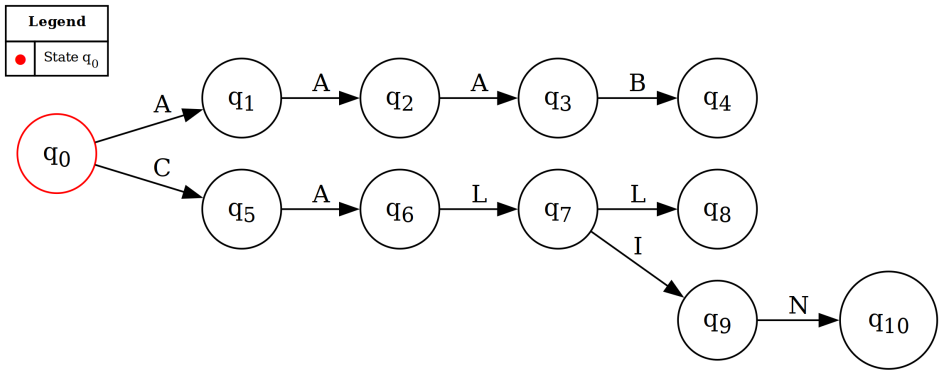


Figure 2.4: Initial generated trie for the dictionary ["AAAB", "CALIN", "CALL", "CALI"]

Afterwards, for all the characters c through which the start state (in this case, q_0) does not transition, we add a transition from the start state to itself; this is equivalent to setting the default transition of the start state to itself. The transitions created until now are called

"goto transitions", and are represented by the function $g : \mathcal{Q} \times \Sigma \to \mathcal{Q}$. For the list of words ["AAAB", "CALIN", "CALL", "CALI"], we present in Figure 2.4 the finite automaton built using only the "goto" transitions g, and in Algorithm 2.1 the algorithm for building the trie for an arbitrary dictionary of words.

Running into a mismatch, which means $g(q, c)$ has no value, is called a failure; a failure does not necessarily mean starting from scratch. Each state has a unique failure state to which it tries falling back when unable to transition; this state is returned by the failure function f.

We compute the failure functions using a breadth-first search algorithm: for all states q of depth 1, we set $f(q) = q_0$, where q_0 is the start state, as we are unable to fallback to any other state. When computing the failure function for states of depth d, the failure function is assumed to have been computed for all states of depth $\leq d - 1$.

For each state q of depth $d - 1$ and for each character c such that q transitions through c, set $q_{fail} := f(q)$, and repeat $q_{fail} \leftarrow f(q_{fail})$ until q_{fail} transitions through c (i.e., $g(q_{fail}, c)$ makes sense); such a state is always found, given the default transition of q_0 is q_0. Afterwards, $f(g(q, c)) = g(q_{fail}, c)$. To create a deterministic finite automaton, instead of iterating multiple times over failure states, for each symbol c through which q does not transition, but q_{fail} does, we set $\delta(q, c) = \delta(q_{fail}, c)$; the second transition might be a fallback transition.

To find all matches of words in the dictionary, we have an output function o that reports, for each state, whether a pattern has been completed or not; for the initial end states, the output function is set when building the prefix tree. For example, for the trie in Figure 2.4, we have $o(q_4) = "AAAB", o(q_8) = "CALL", o(q_9) = "CALI"$ and $o(q_{10}) = "CALIN"$. If the current state has an output, we print it to report that we matched a word. When computing the failure function, we merge the outputs of a state q with the outputs of its failure state $f(q)$. For the previously presented example, all states, along with their corresponding failure states, are shown below.

state	q_0	q_1	q_5	q_2	q_6	q_3	q_7	q_4	q_8	q_9	q_{10}
failure state	$-$	q_0	q_0	q_1	q_1	q_2	q_0	q_0	q_0	q_0	q_0

To adapt the algorithm to return only whether a string contains any of the words in the dictionary, rather than finding all occurrences, we first ensure that there are no words w_1, w_2 in the dictionary such that w_1 is a prefix of w_2; otherwise, w_2 is redundant, knowing it is never matched without matching w_1. Thereafter, for any word "$w_0 w_1 \dots w_{n-1}$" in the dictionary, when reaching the last character w_{n-1}, we connect $\delta(q, w_{n-1}) = E$, where q has the prefix "$w_0 w_1 \dots w_{n-2}$", rather than creating a state with the prefix "$w_0 w_1 \dots w_{n-1}$"; E is a final state, and reaching E means we matched one of the words. Building on a similar idea to [Gic25], we employ a similar algorithm for the task at hand.

3 Single Pattern Matching

This chapter focuses on building a deterministic finite automaton for one single sub-pattern, containing no wildcards; the equivalent SQL patterns are "$p_0p_1 \dots p_{n-1}$%", "%$p_0p_1 \dots p_{n-1}$" or "%$p_0p_1 \dots p_{n-1}$%", where p_i is not the underscore wildcard character. There is a fourth usage as well, checking whether a string is equal to "$p_0p_1 \dots p_{n-1}$"; however, in a database system, this predicate can be optimised out into a string equality predicate, so we do not cover this case. We start by explaining the algorithm for building an automaton for end patterns.

3.1 End Match Automata

Given an end pattern "%$p_0p_1 \dots p_{n-1}$", we generate the set $\mathcal{S}$ of all sequences of symbols $s = \{sym_0, sym_1, \dots sym_{m-1}\}$ such that s is valid, the string obtained by decompressing s ends with this pattern and the sequence is minimal (i.e. the string obtained by decompressing $\{sym_1, \dots, sym_{m-1}\}$ does not end with the pattern). For the end automaton, when parsing, we start from the end of the compressed string and go backwards; consequently, the automaton has only one start state. However, there 9 end states in total: when parsing a sequence from $\mathcal{S}$, we end in E_i, where i is the index in sym_0 of the character corresponding to p_0. We present the values of $\mathcal{S}$ and E_i for a toy example below in Table 3.1b.

(a) Symbol table

0	"AB"
1	"ABCD"
2	"BCD"
3	"CA"
4	"CDE"
5	"FAB"

(b) Values of $\mathcal{S}, S_i, E_i$

"ABC%"		"%ABC%"			"%ABC"	
$\mathcal{S}$	E_i	$\mathcal{S}$	S_i	E_i	$\mathcal{S}$	E_i
0, 4	E_1	0, 3	S_0	E_1	3, 255, 66, 255, 67	E_1
0, 255, 67	E_0	0, 4	S_0	E_1	5, 255, 67	E_1
1	E_3	5, 3	S_1	E_1	0, 255, 67	E_0
0, 3	E_1	1	S_0	E_3		
		3, 2	S_1	E_2		
		3, 255, 66, 255, 67	S_1	E_0		
		5, 255, 67	S_1	E_0		
		0, 255, 67	S_0	E_0		
		5, 4	S_1	E_1		

Table 3.1: Example of a symbol table and patterns, along with the corresponding matching sequences

We build the automaton greedily from the set $\mathcal{S}$; indeed, the automaton cannot be built greedily only if there is $s^{(0)} = \{c_0, c_1, \dots, c_{k-1}\}, s^{(1)} = \{c_{k_0}, c_{k_0+1}, \dots, c_{k-1}\} \in \mathcal{S}, k_0 \neq 0$ (i.e., $s^{(1)}$ is a subsequence of $s^{(0)}$); with this assumption, there are two cases:

I. if $c_{k_0-1} \neq 255$, then decompressing $\{c_1, \ldots, c_{k-1}\}$ ends with the pattern, violating the minimality principle;
II. if $c_{k_0-1} = 255$, Property 2.3 is violated because decoding $\{255, c_{k_0}\}$ and decoding $\{c_{k_0}\}$ end with the same byte; as such, the assumption made is false.

To find the set $\mathcal{S}$, we split $\mathcal{S}$ into disjoint subsets $\mathcal{S}_k$, depending on where the last symbol not fully contained within the pattern in the compressed form ends. For $k = 0$, as a convention, the last not fully contained symbol in the pattern ends right before the pattern. As such, $\mathcal{S}_0$ contains one element: the compressed representation of "$p_0 p_1 \ldots p_{n-1}$". For other values of k, the last not fully contained symbol within the pattern ends with "$p_0 p_1 \ldots p_{k-1}$", and then "$p_k p_{k+1} \ldots p_n$" is compressed. Consequently, each set $\mathcal{S}_k$ is easily written as a cartesian product:

$$\mathcal{S}_k = \{s \in symTable | s \text{ ends with, not equal to "} p_0 p_1 \ldots p_{k-1}\text{"}\} \times \{\left(c_0^{(k)}, c_1^{(k)}, \ldots, c_{j_k-1}^{(k)}\right)\},$$

$$\text{where } \left(c_0^{(k)}, c_1^{(k)}, \ldots, c_{j_k-1}^{(k)}\right) = \begin{cases} \text{compress("} p_k p_{k+1} \ldots p_n \text{")} & \text{if } k < n \\ () & \text{if } k = n \\ \emptyset & \text{if } k > n \end{cases}$$

We aim to build the automaton without materialising all possible sequences of symbols, using the sets $\mathcal{S}_k$. For $k = 0$, the sequence of symbols to match is $\{c_0^{(0)}, c_1^{(0)}, \ldots, c_{j_0-1}^{(0)}\}$; we set working state q_{cur} to be the start state S, ensure that $\delta(q_{cur}, c_{j_0-1}^{(0)})$ exists, and then set $q_{cur} = \delta(q_{cur}, c_{j_0-1}^{(0)})$; afterwards, ensure that $\delta(q_{cur}, c_{j_0-2}^{(0)})$ exists, and set $q_{cur} = \delta(q_{cur}, c_{j_0-2}^{(0)})$, and so on; finally, ensure that $\delta(q_{cur}, c_0^{(0)}) = E_0$. Consequently, when traversing $\{c_{j_0-1}, c_{j_0-2}, \ldots, c_0\}$ starting from S, we end up in E_0. We present the algorithm below in Algorithm 3.1.

```
void createEndAutomaton(qStart, qEnds, symTable, pattern, k=0)
    compressed = symTable.compress(pattern);
    qCurrent = qStart;
    for (i = compressed.size() - 1; i >= 1; --i)
        if (!qCurrent.canTransition(compressed[i]))
            qCurrent.addTransition(compressed[i], createState());
        qCurrent = qCurrent.transition(compressed[i]);
    qCurrent.addTransition(compressed[0], qEnds[0]);
```

Listing 3.1: Creating the end automaton for $k = 0$

For $k > 0$, the elements in $\mathcal{S}_k$ are of the form $\{s, c_0^{(k)}, c_1^{(k)}, \ldots, c_{j_k-1}^{(k)}\}$; therefore, we use a similar algorithm to the one from before, but $\delta(q_{cur}, c_0^{(k)}) \neq E_0$, where q_{cur} is the state obtained by parsing $\{c_{j_k-1}^{(k)}, c_{j_k-2}^{(k)}, \ldots, c_1^{(k)}\}$ starting in S; in this case, we ensure that we have a non-end state to transition through $c_0^{(k)}$, and then, from that state, we transition through all possible symbols s to the respective end states. We present the algorithm for building the end automaton for $k > 0$ below in Algorithm 3.2. Since the maximum length of a symbol is defined to be 8 bytes, k takes values between 0 and $min(n, 7)$.

As 255 acts only as the escape character, signifying that the following byte is a raw byte, and is not a valid byte in neither ASCII nor UTF-8 encoding, matching the compressed string leads to false positives if we mistake the k-th symbol for the byte k and ignore the preceding escape

character, which currently happens when parsing the compressed string backwards. With the example symbol table from 3.1a, the compressed string $s = \{255, 0, 255, 67\}$ is mistaken for a match for the compressed end pattern $p =$ "ABC", if we do not look ahead after finishing the match to find the escaped character 255. Consequently, we introduce the notion of a pseudo-end.

```
1  void createEndAutomaton(qStart, qEnds, symTable, pattern, k)
2      [prefix, restPattern] = split(pattern, k);
3      // assume the function below does not return exact matches
4      symbols = symTable.findWithSuffix(prefix);
5      if (symbols.empty())
6          return;
7      compressed = symTable.compress(restPattern);
8      qCurrent = qStart;
9      for (i = compressed.size() - 1; i >= 0; --i)
10         if (!qCurrent.canTransition(compressed[i]))
11             qCurrent.addTransition(compressed[i], createState());
12         qCurrent = qCurrent.transition(compressed[i]);
13     for (symIdx: symbols)
14         endIdx = symTable.symbols[symIdx].size() - k;
15         qCurrent.addTransition(symIdx, qEnds[endIdx]);
```

Listing 3.2: Creating the end automaton for $k > 0$

Definition 3.1. A state q is called a pseudo-end for an automaton if it acts as an accept state in case there are no more symbols in the original string, its default transition is an accept state, but it has a subset of transitions leading to other non-accept states; when drawing a finite automaton pseudo-end states are represented by triple circles, to distinguish them from ordinary accept states, which are drawn with double circles.

To avoid modifying the original end states, we create 9 pseudo-end states $\{E_0', \ldots, E_8'\}$ mirroring the original end states $\{E_0, \ldots, E_8\}$, which have the following properties: $\delta(E_i', x) = E_i$ for $x \neq 255$ and $\delta(E_i', 255) = \varepsilon$. When parsing a string, if we reach a pseudo-end, we do not increment the string index; however, if we reach a regular end state, we do, as we have consumed one more symbol than we are supposed to. We present the automaton for the pattern in 3.1b below in Figure 3.1.

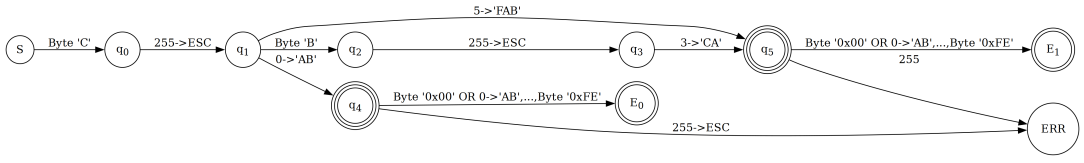


Figure 3.1: Automaton for the pattern "%ABC" for the symbol table 3.1a

Having generated the backwards automaton for the end pattern, we now explain the algorithms for generating the forward automata. We start with the simpler case: the automata for start patterns.

3.2 Start Match Automata

Given a start pattern "$p_0p_1 \dots p_{n-1}$%", we aim to generate the set $\mathcal{S}$ of all possible sequences of symbols $s = \{sym_0, sym_1, \dots sym_{m-1}\}$ such that s is valid, the string obtained by decompressing s starts with this pattern and the sequence of symbols is minimal (i.e. the string obtained by decompressing $\{sym_0, \dots, sym_{m-2}\}$ does not start with the pattern). The built automaton has only one start state; however, there are 9 end states: when parsing a sequence from $\mathcal{S}$, we end in E_i, where i is the index in sym_{m-1} of the character following the character corresponding to p_{n-1}. For the start match, it is not as trivial to split the set $\mathcal{S}$ into disjoint subsets for which creating the automaton is trivial.

Our goal is to build a single-start, deterministic, acyclic, finite-state automaton for the current pattern. To simplify this, we first generate the set of sequences of symbols that we aim to match. Intuitively, Definition 2.4 means that the innate structure of the automaton defines a Directed Acyclic Graph (DAG). This type of automaton is designed to match sets of strings while minimising the number of states [Dac+00]. Define $S : \{0, 1, \dots, 255\}^* \to \mathcal{P}(\{0, 1, \dots, 255\}^*)$ as the function that, for a start pattern "$p_0p_1 \dots p_{n-1}$", generates its set of matching sequences $\mathcal{S}$; we aim to compute $S(\text{"}p_0p_1 \dots p_{n-1}\text{"})$ for an arbitrary pattern.

The set $S(\text{"}p_0p_1 \dots p_{n-1}\text{"})$ highly depends on the value of n, and is split into three distinct cases: $n = 1$, $n = 2$ and $n \geq 3$; we assume that the pattern is not empty ($n > 0$). After finding the value of $S(\text{"}p_0p_1 \dots p_{n-1}\text{"})$, in the three cases, we construct a deterministic acyclic finite automaton matching the sequences of symbols in the respective set. For $n = 1$, define the set of monosymbolic sequences, with the symbol starting with "p_0", as $\mathcal{P}ref = \{\{i\} | i\text{th symbol starts with "}p_0\text{"}\}$ and let

$$\mathcal{S}^{(1)} = \begin{cases} \mathcal{P}ref & \text{if "}p_0\text{" is a symbol} \\ \mathcal{P}ref \cup \{\{255, p_0\}\} & \text{if "}p_0\text{" is not a symbol} \end{cases}$$

We aim to prove that $\mathcal{S}^{(1)} = S(\text{"}p_0\text{"})$. Choose $c = \{c_0, c_1, \dots, c_{k-1}\} \in S(\text{"}p_0\text{"})$, where $k \geq 1$ and $c_0, c_1, \dots, c_{k-1}$ are arbitrary bytes. If $c_0 = 255$, then $k \geq 2$ and decompressing c is be the actual byte c_1, followed by decompressing $\{c_2, c_3, \dots, c_{k-1}\}$; we infer that $c_1 = p_0$. We must have $k \leq 2$; otherwise, decompressing $\{c_0, c_1, \dots, c_{k-2}\}$ starts with the pattern and, consequently, the minimality condition is violated; as such, $c = \{255, p_0\} \in \mathcal{S}^{(1)}$. This case occurs only when "p_0" is not a standalone symbol; alternatively, $\{255, p_0\}$ is not a valid sequence of symbols. If $c_0 \neq 255$, then decompressing $\{c_0\}$ is at least one character long; by minimality, $k = 1$ and $c = \{c_0\}$; otherwise, decompressing $\{c_0, c_1, \dots, c_{k-2}\}$ starts with the pattern. Moreover, $c \in \mathcal{P}ref$, knowing that decompressing c_0 starts with "p_0". We conclude that $c \in \mathcal{S}^{(1)}$ and $S(\text{"}p_0\text{"}) \subseteq \mathcal{S}^{(1)}$.

Conversely, pick an arbitrary $c \in \mathcal{S}^{(1)}$, we show that c satisfies the three required properties; by choice, it is trivial that the string obtained by decompressing c starts with "p_0" and that the sequence is minimal, since it is one symbol long (or 2 symbols long, when the first byte of the sequence is the escape byte). For the validity of the sequence, if "p_0" is not a symbol, then compressing "p_0" is $\{255, p_0\}$. Similarly, for a symbol $\text{sym}_i = \text{"}p_0s_1s_2 \dots s_{l-1}\text{"}, l \geq 0$ starting with "p_0", compressing "$p_0s_1s_2 \dots s_{l-1}$" is $\{sym_i\}$. As such, all elements in $\mathcal{S}$ satisfy all three conditions, so $\mathcal{S}^{(1)} = S(\text{"}p_0\text{"})$. To build the automaton given the start state q, then, we define $\delta(q, c) = E_1$ for all symbols c starting with "p_0"; if "p_0" is not a standalone symbol, we create an escape state q_{esc}, and do $\delta(q_{esc}, p_0) = E_0$ and $\delta(q, 255) = q_{esc}$; as a convention, transitions

through raw bytes go to E_0. We present the algorithm for creating the automaton for patterns of length 1 in Algorithm 3.3. As this function is called recursively, to avoid creating strings multiple times, we create the string only once, and then pass the string index to the function; $n = 1$ is equivalent to `idx = pattern.size() - 1`. For forward automata, we add transitions to states from `qEnds` using a special function `addEndTransition(symbol, qEnds, endIndex)`.

```
1  State* createStartAutomaton(qStart, qEnds, symTable, pattern, idx)
2      if (qStart == NULL)
3          qStart = createState();
4      for (i = 0; i < symTable.nSymbols; ++i)
5          sym = symTable.symbols[i];
6          if (sym.empty() or sym[0] != pattern[idx])
7              continue;
8          qStart.addEndTransition(i, qEnds, 1);
9      if (symTable.symIdx(pattern.substr(idx, 1)) == -1)
10         qTemp = createState();
11         qTemp.addEndTransition(pattern[idx], qEnds, 0);
12         qStart.addTransition(255, qTemp);
13     return qStart;
```

Listing 3.3: Creating the start automaton for `idx = pattern.size() - 1`

For $n = 2$, let $\mathcal{P}ref = \{\{i\} | i\text{th symbol starts with "}p_0p_1\text{"}\}$ the set of monosymbolic sequences, whose symbol starts with the pattern and define

$$\mathcal{S}^{(2)} = \begin{cases} \mathcal{P}ref & \text{if "}p_0p_1\text{" is a symbol} \\ \mathcal{P}ref \cup (\{i\} \times S(\text{"}p_1\text{"})) & \text{if "}p_0p_1\text{" is not a symbol and "}p_1\text{" is the } i\text{th symbol} \\ \mathcal{P}ref \cup (\{255, p_0\} \times S(\text{"}p_1\text{"})) & \text{if neither "}p_0p_1\text{" nor "}p_0\text{" are symbols} \end{cases}$$

We show that $S(\text{"}p_0p_1\text{"}) \subseteq \mathcal{S}^{(2)}$, and that for all sequences $s \in \mathcal{S}^{(2)} \setminus S(\text{"}p_0p_1\text{"})$, s matches the pattern and is minimal, but is not valid. Pick an arbitrary sequence of symbols $c = \{c_0, c_1 \ldots, c_{k-1}\} \in \mathcal{S}^{(2)}$, where $c_0, c_1, \ldots, c_{k-1}$ are arbitrary bytes; there are two cases:

I. if $c \in \mathcal{P}ref$, minimality stems from $k = 1$ and, by the definition of $\mathcal{P}ref$, decompressing c starts with "p_0p_1";
II. otherwise, $c \in \{i\} \times S(\text{"}p_1\text{"})$ ($c \in \{255, p_0\} \times S(\text{"}p_1\text{"})$); then, decompressing $\{c_0\}$ ($\{c_0, c_1\}$) is "p_0" and, by definition, decompressing $\{c_1, c_2, \ldots, c_{k-1}\}$ ($\{c_2, c_3, \ldots, c_{k-1}\}$) starts with "$p_1$", while decompressing $\{c_1, c_2, \ldots, c_{k-2}\}$ ($\{c_2, c_3, \ldots, c_{k-2}\}$) does not.

As a consequence, decompressing c matches "p_0p_1" minimally; notice that not all sequences in $\mathcal{S}^{(2)}$ are valid: if $sym_0 =$ "XYZ", $sym_1 =$ "X" and $sym_2 =$ "YZ", for the pattern "XY", $\mathcal{S}^{(2)}$ contains the sequence $\{1, 2\}$, despite that it is not a valid sequence by Property 2.1. Conversely, pick an arbitrary sequence of symbols $c = \{c_0, c_1, \ldots, c_{k-1}\} \in S(\text{"}p_0p_1\text{"})$. There are three cases:

I. if "p_0p_1" is a standalone symbol, by Property 2.1, decompressing c_0 must start with "p_0p_1", so $c \in \mathcal{P}ref = \mathcal{S}^{(2)}$;
II. if "p_0p_1" is not a symbol, but "p_0" is the ith symbol, for $c \notin \mathcal{P}ref$, then $c_0 = i$ and "p_1" is matched minimally by $\{c_1, c_2, \ldots, c_{k-1}\}$, so $\{c_1, c_2, \ldots, c_{k-1}\} \in S(\text{"}p_1\text{"})$. We conclude that $c \in \mathcal{P}ref \cup (\{i\} \times S(\text{"}p_1\text{"})) = \mathcal{S}^{(2)}$;

III. if neither "p_0p_1" nor "p_0" are symbols, for $c \notin \mathcal{P}ref$, then $c_0 = 255, c_1 = p_0$ and "p_1" is matched minimally by $\{c_2, c_3, \ldots, c_{k-1}\}$, so $\{c_2, c_3, \ldots, c_{k-1}\} \in S("p_1")$. We conclude that $c \in \mathcal{P}ref \cup (\{255, p_0\} \times S("p_1")) = \mathcal{S}^{(2)}$ and that $S("p_0p_1") \subseteq \mathcal{S}^{(2)}$.

Consequently, we have proven that $c \in \mathcal{S}^{(2)}$. The implementation is similar to the case $n = 1$. Denoting q as the start state, for all symbols c starting with "p_0p_1", we define $\delta(q, c) = E_2$. If "p_0p_1" is not a symbol, we build the automaton for the start pattern "p_1" starting at q_0; if "p_0" is the c-th symbol, we connect the automata by doing $\delta(q, c) = q_0$, otherwise, q_{esc} is created, and we define $\delta(q_{esc}, p_0) = q_1$ and $\delta(q, q_{esc}) = 255$. We present the algorithm below in Algorithm 3.4.

```
1  State* createStartAutomaton(qStart, qEnds, symTable, pattern, idx)
2      if (qStart == NULL)
3          qStart = createState();
4      for (i = 0; i < symTable.nSymbols; ++i)
5          sym = symTable.symbols[i];
6          if (sym.size() < 2 or sym[:2] != pattern.substr(idx, 2))
7              continue;
8          qStart.addEndTransition(i, qEnds, 2);
9      if (symTable.symIdx(pattern.substr(idx, 2)) != -1)
10         return qStart;
11     // use the case n = 1 previously mentioned
12     qNext = createStartAutomaton(NULL, qEnds, symTable, pattern, idx + 1);
13     symIdx = symTable.symIdx(pattern.substr(idx, 1));
14     if (symIdx != -1)
15         qStart.addTransition(symIdx, qNext);
16     else
17         qTemp = createState();
18         qTemp.addTransition(pattern[idx], qNext);
19         qStart.addTransition(255, qTemp);
20     return qStart;
```

Listing 3.4: Creating the start automaton for `idx = pattern.size() - 2`

For $n \geq 3$, by Property 2.2, there is at most one symbol starting with the prefix "$p_0p_1p_2$". Define

$$\mathcal{D} = \begin{cases} \{j\} \times S("p_2p_3 \ldots p_{n-1}") & \text{if "}p_0p_1\text{" is the } j\text{th symbol} \\ \{j\} \times S("p_1p_2 \ldots p_{n-1}") & \text{if "}p_0p_1\text{" is not a symbol, but "}p_0\text{" is the } j\text{th symbol} \\ \{255, p_0\} \times S("p_1p_2 \ldots p_{n-1}") & \text{if neither "}p_0p_1\text{" nor "}p_0\text{" are symbols} \end{cases}$$

representing all sequences, not necessarily valid, that start with the given pattern and encode "p_0p_1" deterministically. Now, let

$$\mathcal{S}^{(n)} = \begin{cases} \{i\} \times S("p_lp_{l+1} \ldots p_{n-1}") & \text{if "}p_0p_1 \ldots p_{l-1}\text{" is the } i\text{th symbol, } 3 \leq l \leq n \\ \{\{i\}\} \cup \mathcal{D} & \text{if "}p_0p_1 \ldots p_{n-1}s_n \ldots s_{n+l}\text{" is the } i\text{th symbol for a suffix} \\ \mathcal{D} & \text{otherwise} \end{cases}$$

Similarly to the case when $n = 2$, we show that $S("p_0p_1 \ldots p_{n-1}") \subseteq \mathcal{S}^{(n)}$ and that any $c \in \mathcal{S}^{(n)} \setminus S("p_0p_1 \ldots p_{n-1}")$ matches the pattern minimally when decompressed. Pick an

arbitrary sequence of bytes $c = \{c_0, c_1, \ldots, c_{k-1}\} \in \mathcal{S}^{(n)}$. If "$p_0p_1 \ldots p_{l-1}$" with $3 \leq l \leq n$ is the ith symbol, decompressing $c_0 = i$ returns "$p_0p_1 \ldots p_{l-1}$", and decompressing $\{c_1, c_2, \ldots, c_{k-1}\}$ starts with "$p_lp_{l+1} \ldots p_{n-1}$", so the pattern is matched when decompressing the sequence of symbols. Minimality stems from the fact that decompressing $\{c_1, c_2, \ldots, c_{k-2}\}$ does not start with "$p_lp_{l+1} \ldots p_{n-1}$". If "$p_0p_1 \ldots p_{n-1}s_n \ldots s_{n+l}$" is the ith symbol, for some non-empty suffix, $c = \{i\}$ is trivially minimal and matches the pattern. Finally, for $c \in \mathcal{D}$, by construction, there are three possibilities:

I. if "p_0p_1" is the j-th symbol, the string obtained by decompressing c_0 returns "p_0p_1", decompressing $\{c_1, c_2, \ldots, c_{k-1}\}$ matches "$p_2p_3 \ldots p_{n-1}$", but the string obtained by decompressing $\{c_1, c_2, \ldots, c_{k-2}\}$ does not;
II. if "p_0p_1" is not a symbol, but "p_0" is the j-th symbol, decompressing c_0 returns "p_0", decompressing $\{c_1, c_2, \ldots, c_{k-1}\}$ matches "$p_1p_2 \ldots p_{n-1}$", but the string obtained when decompressing $\{c_1, c_2, \ldots, c_{k-1}\}$ does not;
III. if neither "p_0p_1" nor "p_0" are symbols, decompressing $\{c_0, c_1\}$ returns "p_0", decompressing $\{c_2, c_3, \ldots, c_{k-1}\}$ matches "$p_1p_2 \ldots p_{n-1}$", but decompressing $\{c_1, c_2, \ldots, c_{k-1}\}$ does not; as such, in all cases, minimal matching occurs.

Therefore, for any arbitrary sequence $c \in \mathcal{S}^{(n)}$, decompressing c matches the start pattern minimally. Pick $c = \{c_0, c_1, \ldots, c_{k-1}\} \in S("p_0p_1 \ldots p_{n-1}")$, we now prove that $c \in \mathcal{S}^{(n)}$. For this, there are three main cases to follow:

I. if "$p_0p_1 \ldots p_{l-1}$" with $3 \leq l \leq n$ is the ith symbol, by Property 2.1 and Property 2.2, $c_0 = i$ and $\{c_1, c_2, \ldots c_{k-1}\}$ is left to match "$p_lp_{l+1} \ldots p_{n-1}$" minimally; as such, $\{c_1, c_2, \ldots, c_{k-1}\} \in S("p_lp_{l+1} \ldots p_{n-1}")$ and $c \in \{i\} \times S("p_lp_{l+1} \ldots p_{n-1}") = \mathcal{S}^{(n)}$.
II. if "$p_0p_1 \ldots p_{n-1}s_n \ldots s_{n+l}$" is the ith symbol for a non-empty suffix, c_0 either decompresses to sym_i, case in which, to ensure minimality, $c = \{c_0\}$, or to "p_0p_1" or "p_0". In the latter case, if "p_0p_1" is the jth symbol, then, $c_0 = j$ by Property 2.1, and $\{c_1, c_2, \ldots, c_{k-1}\}$ is left to match "$p_2p_3 \ldots p_{n-1}$" minimally; consequently, $c \in \mathcal{D}$. The cases in which "p_0" is the jth symbol and in which neither "p_0p_1" nor "p_0" are symbols are treated similarly; as such, $c \in \mathcal{S}^{(n)}$.
III. otherwise, showing that $c \in \mathcal{S}^{(n)}$ is equivalent to showing $c \in \mathcal{D}$, case treated the same as above.

Consequently, $c \in \mathcal{S}^{(n)}$, so $S("p_0p_1 \ldots p_{n-1}") \subseteq \mathcal{S}^{(n)}$ and all $c \in \mathcal{S}^{(n)} \setminus S("p_0p_1 \ldots p_{n-1}")$ are not valid, and, as such, never given as input to the matching algorithm. The algorithm building the automaton starting at the state q matching the sequences in $\mathcal{S}^{(n)}$ is split into two cases:

I. if there is a symbol $c =$ "$p_0p_1 \ldots p_{l-1}$" starting with "$p_0p_1p_2$" completely contained within the pattern, we create the automaton starting at q_0 for the start pattern "$p_lp_{l+1} \ldots p_{k-1}$", and we define $\delta(q, c) = q_0$;
II. otherwise, if there is a symbol c completely containing the pattern, we define $\delta(q, c) = E_n$, and we are left to match the sequences from $\mathcal{D}$. If "p_0p_1" is the c-th symbol, then we build the automaton starting at q_0 for the start pattern "$p_2p_3 \ldots p_{k-1}$" and connect $\delta(q, c) = q_0$; otherwise, we create the automaton for the start pattern "$p_1p_2 \ldots p_{k-1}$" starting at q_0 and, if "p_0" is the c-th symbol, we define $\delta(q, c) = q_0$, otherwise $\delta(q_{esc}, p_0) = q_0$ and $\delta(q, 255) = q_{esc}$.

We present the implementation of the algorithm below in Algorithm 3.5 and, as an example,

we show the automaton for the start pattern from 3.1b in Figure 3.2. Finally, having generated the automata for the start and end patterns, we now examine the algorithm for creating the more complex automata for the middle patterns.

```
1  State* createStartAutomaton(qStart, qEnds, symTable, pattern, idx)
2      if (qStart == NULL)
3          qStart = createState();
4      symIdx = symTable.findWithPrefixOrEqual(pattern.substr(0, 3));
5      symSize = symIdx == -1 ? -1 : symTable.symbols[symIdx].size();
6      numChars = std::min(pattern.size() - idx, symSize);
7      matches = symIdx != -1 and
8          pattern[idx:idx + numChars] == symTable.symbols[symIdx][:numChars];
9      if (matches and idx + symSize <= pattern.size())
10         startIdx = idx + symSize;
11         qNext = createStartAutomaton(NULL, qEnds, symTable, pattern, startIdx);
12         qStart.addTransition(symIdx, qNext);
13         return qStart;
14     if (matches)
15         qStart.addEndTransition(symIdx, qEnds, pattern.size() - idx);
16     symIdx = symTable.symIdx(pattern.substr(idx, 2));
17     if (symIdx != -1)
18         qNext = createStartAutomaton(NULL, qEnds, symTable, pattern, idx + 2);
19         qStart.addTransition(symIdx, qNext);
20         return qStart;
21     qNext = createStartAutomaton(NULL, qEnds, symTable, pattern, idx + 1);
22     symIdx = symTable.symIdx(pattern.substr(idx, 1));
23     if (symIdx != -1)
24         qStart.addTransition(symIdx, qNext);
25     else
26         qTemp = createState();
27         qStart.addTransition(255, qTemp);
28         qTemp.addTransition(pattern[idx], qNext);
29     return qStart;
```

Listing 3.5: Creating the start automaton for `idx <= pattern.size() - 3`

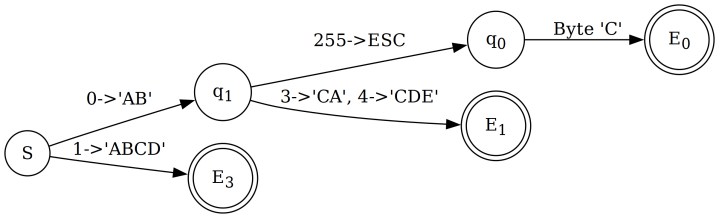


Figure 3.2: Automaton for the pattern "*ABC*%" for the symbol table 3.1a

3.3 Middle Match Automata

Given a middle pattern "%$p_0p_1 \dots p_{n-1}$%", we generate the set $\mathcal{S}$ of all possible sequences of symbols $s = \{sym_0, sym_1, \dots sym_{m-1}\}$ such that s is valid, the string obtained by decompressing s contains this pattern and the sequence of symbols is minimal (i.e. the strings obtained by decompressing $\{sym_0, \dots, sym_{m-2}\}$ and $\{sym_1, \dots, sym_{m-1}\}$ do not contain the pattern).

For this type of automaton, we have multiple starts and multiple ends, so, for each sequence in $\mathcal{S}$, we start in a state S_i, where i is the index in sym_0 of the byte matching p_0, and we end in a state E_j, where j is the index in sym_{m-1} of the character following the character corresponding to p_{n-1}. As a symbol is at most of length eight, nine different start states are required: one for each possible index, and one for when the automaton needs to start from the start of the symbol. Creating a middle match automaton has three different phases: the initial phase, the start-linking phase, and the fallback creation phase. We discuss each phase individually.

3.3.1 Initial Phase

For the initial phase, we must consider how a pattern "$p_0p_1 \dots p_{n-1}$" is matched by a sequence of symbols. Initially, if the entire pattern is not contained within a symbol, we split the cases depending on where the last symbol, not fully contained within the symbol, ends, similar to when building the automaton for an end pattern.

If a symbol ends with "$p_0p_1 \dots p_{l-1}$", then "$p_lp_{l+1} \dots p_{n-1}$" is treated as a start pattern; for $l \geq 1$, we find $\mathcal{S}uff^{(l)} = \{i | i\text{th symbol ends with "}p_0p_1 \dots p_{l-1}\text{"}\}$ and create the automaton for the start pattern "$p_lp_{l+1} \dots p_{n-1}$", having q_l as its start state; finally, for all $c \in \mathcal{S}uff^{(l)}$, we add a transition from the $(len(c) - l)$th start to q_l through c. Given that the sequence of symbols $\{c_0, c_1, \dots, c_{k-1}\}$ (starting with "$p_lp_{l+1} \dots p_{n-1}$" when decompressed) takes q_l to some end state E, then $\{sym, c_0, c_1, \dots, c_{k-1}\}$ takes the corresponding start state to E as well. We look at the following example, by initially defining the following symbol table:

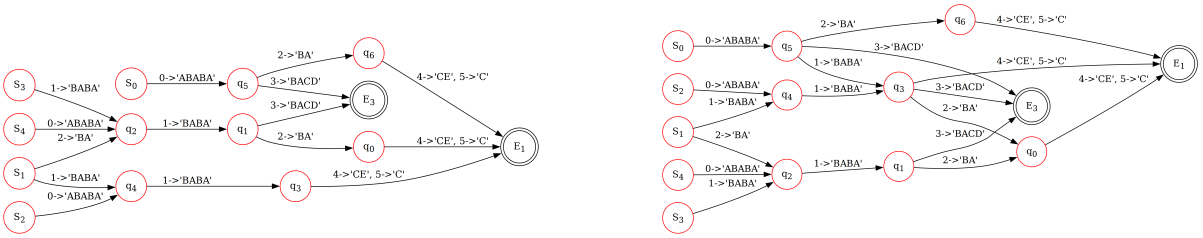
0	1	2	3	4	5
"ABABA"	"BABA"	"BA"	"BACD"	"CE"	"C"

Table 3.2: Symbol table for searching the middle pattern "%ABABABAC%"

and then choosing the middle pattern "%$ABABABAC$%". Using the previous algorithm, the created finite automaton is the one presented in Figure 3.3a; we ignore the potentially escaped characters for simplicity purposes. As starting from the 2th character of a symbol (consuming more bytes of the symbol) should also allow starting from the 4nd character of the same symbol (consuming fewer bytes of the symbol), then we should be able to reach E_3 from S_2 through the sequence $\{0, 1, 3\}$. Therefore, we define $\delta(q_3, 3) = E_3$, making the sequence of symbols $\{1, 1, 3\}$ (decompressed as $B\underline{ABABABACD}$) end up in E_3 starting from S_2.

To be able to build this the first time we create the sub-automata, we create the sub-automata from "weakest" (i.e., larger values of the start indexes) to "strongest" (i.e., smaller values of the start indexes), and we keep an array `transitionStates` mapping, at each step, each symbol c to $\delta(S_i, c)$, with i as small as possible, or `NULL` if it does not exist.

When we create the sub-automaton obtained by transitioning through the ith index of a symbol, the sub-automatons obtained by transitioning through the jth index of that same



(a) The middle automaton obtained by the naive algorithm

(b) The extended middle automaton obtained after the initial phase

Figure 3.3: Automata for the pattern "%$ABABABAC$%" for the symbol table 3.2

symbol, if they exist, should already be created, for all $j > i$, in order to know what transitions to copy. For this, notice that for a symbol $sym = s_0 s_1 \dots s_{k-1}$, starting from index $i \neq 0$ implies $s_i s_{i+1} \dots s_{k-1} = p_0 p_1 \dots p_{k-i-1}$ is used as a prefix of the pattern, and the start automaton of the pattern "$p_{k-i} p_{k-i+1} \dots p_{n-1}$" is created. Therefore, we must ensure that the start automatons for the patterns "$p_{k-j} p_{k-j+1} \dots p_{n-1}$" for all $j > i$ already exist. Define $n_{max} = min(8, n) - 1$; then, the steps for building the automaton are:

- create the automaton for the start pattern "$p_1 p_2 \dots p_{n-1}$" having start state q_1
- find $\mathcal{S}uff^{(1)}$, for $sym \in \mathcal{S}uff^{(1)}$, $\delta(S_{len(sym)-1}, sym) = q_1$
- create the automaton for the start pattern "$p_2 p_3 \dots p_{n-1}$" having start state q_2
- find $\mathcal{S}uff^{(2)}$, for $sym \in \mathcal{S}uff^{(2)}$, $\delta(S_{len(sym)-2}, sym) = q_2$
- for $sym \in \mathcal{S}uff^{(1)} \setminus \mathcal{S}uff^{(2)}$, $\delta(S_{len(sym)-2}, sym) = q_1$
- for $sym \in \mathcal{S}uff^{(1)} \cap \mathcal{S}uff^{(2)}$, copy the sub-automaton starting from q_1 into q_2

  ...
- create the automaton for the start pattern "$p_{n_{max}} p_{n_{max}+1} \dots p_{n-1}$" having start state $q_{n_{max}}$
- find $\mathcal{S}uff^{(n_{max})}$, for $sym \in \mathcal{S}uff^{(n_{max})}$, $\delta(S_{len(sym)-n_{max}}, sym) = q_{n_{max}}$
- for $sym \in \mathcal{S}uff^{(n_{max}-1)} \setminus \mathcal{S}uff^{(n_{max})}$, $\delta(S_{len(sym)-n_{max}}, sym) = q_{n_{max}-1}$
- for $sym \in \mathcal{S}uff^{(n_{max}-1)} \cap \mathcal{S}uff^{(n_{max})}$, copy the automaton starting from $q_{n_{max}-1}$ into $q_{n_{max}}$
- for $sym \in \mathcal{S}uff^{(n_{max}-2)} \setminus (\mathcal{S}uff^{(n_{max})} \cup \mathcal{S}uff^{(n_{max}-1)})$, $\delta(S_{len(sym)-n_{max}}, sym) = q_{n_{max}-2}$

  ...
- create the automaton for the start pattern "$p_0 p_1 \dots p_{n-1}$" having as a start state $q_\infty(S_0)$.

Subsequently, we justify why copying sub-automata is valid and faster than creating a new sub-automaton when a symbol appears twice in the transitions. First, at any step l, we define the set $\mathcal{T}^{(l)} = \{\texttt{transitionStates}^{(l)}\texttt{[i]} | i \in \mathcal{S}uff^{(l)}, \texttt{transitionStates}^{(l)}\texttt{[i]} \neq \texttt{NULL}\}$, where

$$\texttt{transitionStates}^{(l)}\texttt{[i]} = \begin{cases} q_{l_0}, & \text{if for a start state } S \text{ and } l_0 < l \text{ maximal, } \delta(S, i) = q_{l_0} \\ \texttt{NULL}, & \text{otherwise.} \end{cases}$$

is just the value of `transitionStates[i]` at the l-th iteration of the algorithm. We claim that, apart from the last step when we are looking at S_0, $\mathcal{T}^{(l)}$ has at most one value; for this, assume that $\mathcal{T}^{(l)}$ has at least 2 different values, q_{l_0} and q_{l_1} with $l_0 < l_1$, implying that there is a symbol i_0 such that $\texttt{transitionStates}^{(l)}[i_0] = q_{l_0}$ and a symbol i_1 such that $\texttt{transitionStates}^{(l)}[i_1] = q_{l_1}$, with $i_0, i_1 \in \mathcal{S}uff^{(l)}$, equivalent with i_0 and i_1 suffixed by

"$p_0 p_1 \dots p_{l_0-1}$". Moreover, there is no $l_0 < l' < l$ such that i_0 is suffixed by "$p_0 p_1 \dots p_{l'-1}$" by maximality. However, because of the assumption, sym_{i_1} is suffixed by "$p_0 p_1 \dots p_{l_1-1}$", and both sym_{i_0} and sym_{i_1} are suffixed by "$p_0 p_1 \dots p_{l-1}$", so, because $l > l_1$, picking $l' = l_1$ yields a contradiction. Consequently, $\mathcal{T}^{(l)}$ has at most one element.

Now, we prove that, if there is an element in $\mathcal{T}^{(l)}$, shallow copying its transitions leads to a formally correct sub-automaton; shallow copying involves redirecting transitions through symbols to already existing states in the source sub-automaton. Assume that, at any step apart from the last one, one of the symbols suffixed by "$p_0 p_1 \dots p_{l_0-1}$" has been previously used to transition from some start state to q_{l_1}. We claim that, by shallow copying the sub-automaton starting from q_{l_1} in the sub-automaton starting from q_{l_0}, the result stays formally correct.

To observe this, notice that any symbol s with which a start state transitions into q_{l_0} must be suffixed by "$p_0 p_1 \dots p_{l_0-1}$"; because of the hypothesis, we also can affirm that "$p_0 p_1 \dots p_{l_0-1}$" is suffixed by "$p_0 p_1 \dots p_{l_1-1}$", so that holds also for any such s. We draw our conclusion by starting at a later index in the symbol; given that the algorithm for shallow copying the sub-automaton has the signature `shallowCopy(State* dest, const State* src)`, we present the algorithm for copying all such transitions, called splitting the starts, in Algorithm 3.6. For simplicity purposes, we postpone splitting the starts until we have built all sub-automata. For S_0, the states that are shallow copied are the ones S_0 directly transitions into.

```
void splitStarts(qStarts)
    transitionStates = std::array<State*, 255>{NULL};
    postStartMapping = std::unordered_map<State*, State*>{};
    orderedPostStarts = std::list<State*>{};
    for (uint8_t idx = 8; idx >= 1; --idx)
        qStart = qStarts[idx];
        for ([symbol, qNext]: qStart.transitions)
            if (transitionStates[symbol] != NULL and
                    !postStartMapping.contains(qNext))
                orderedPostStarts.push_back(qNext);
                postStartMapping[qNext] = transitionStates[symbol];
        for ([symbol, qNext]: qStart.transitions)
            transitionStates[symbol] = qNext;
    for (qPostStart: orderedPostStarts)
        shallowCopy(qPostStart, postStartMapping[qPostStart]);
    for ([symbol, qNext]: starts[0].transitions)
        if (symbol == 255 or transitionStates[symbol] == NULL)
            continue;
        shallowCopy(qNext, transitionStates[symbol]);
```

Listing 3.6: Splitting all starts

Splitting the starts implies that states immediately following the start states (also called post-starts) copy states of other sub-automata, and we have proven that each such state has a unique transition state (if not `NULL`). Therefore, we store a mapping from each post-start state to its transition state, and we traverse the starts in reverse order to update both the mapping and the array of transition states. To make sure that, in case q_0 has transition state q_1 and q_1 has transition state q_2, we copy q_2 in q_1 before we copy q_1 in q_0, we store a list of the states in

the topological order we traversed them, and we shallow copy using that order.

As 255 is a special character, and, for a middle automaton, $q_{def} = S_0$ rather than $q_{def} = \varepsilon$, we ensure that S_0 transitions through 255 so that parsing $\{255, 0\}$ does not match the symbol 0. As a consequence, if S_0 cannot transition through 255, a special state with no transitions is created; by doing this, parsing $\{255, 0\}$ leads back to $q_{def} = S_0$.

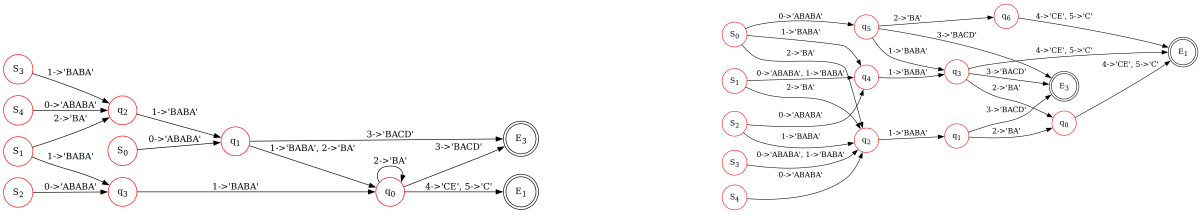
In the beginning of the section, we intentionally left out the symbols completely containing the current pattern; for these symbols $c =$ "$s_0 s_1 \dots s_{m-1}$", find all indexes $0 < i \leq m - n$ which satisfy "$s_i s_{i+1} \dots s_{i+n-1}$" = "$p_0 p_1 \dots p_{n-1}$", and then add $\delta(S_i, c) = E_{i+n}$. The transitions from other starts, respectively all other states, through c are added later when adding the other transitions of all starts, respectively when creating the fallback states. From now on, we call this process middle linking, and we present the implementation below in Algorithm 3.7.

```
1  void middleLinking(qStarts, qEnds, symTable, pattern)
2      if (pattern.size() > 7)
3          return;
4      for (symIdx = 0; idx < symTable.nSymbols; ++idx)
5          sym = symTable.symbols[symIdx];
6          if (sym.size() < pattern.size() + 1)
7              continue;
8          for (i = 1; i <= sym.size() - pattern.size(); ++i)
9              if (sym.substr(i, pattern.size()) == pattern)
10                 endIdx = i + pattern.size();
11                 qStarts[i].addEndTransition(symIdx, qEnds, endIdx);
```

Listing 3.7: Adding the symbols containing the pattern

When creating a sub-automaton, we may call `createStartAutomaton` with the same value of `idx` multiple times. As building a start automaton is a slow process, we determine when it is feasible to return the same automaton with all subsequent function calls. For this, before creating any sub-automata, we initialise a vector of the size of the pattern called `cache`, in which, after creating the start automaton for an index `idx`, we store the start of said sub-automaton in `cache[idx]`.

Figure 3.4: Automaton for the pattern "%$ABABABAC$%" for the symbol table 3.2 after



(a) Caching with `enableCaching` always `true`

(b) Finishing linking the starts

The naive solution is `cachedCreateStartAutomaton` always returns `cache[idx]` when it is not `NULL`. However, by building the sub-automata with caching always enabled, as in the previous example, we notice that the acyclicity of the automaton is compromised. This happens because, starting from S_2, both $\{0, 1, 4\}$ and $\{0, 1, 2, 4\}$ lead to E_1; as such, when splitting the starts, a transition between q_0 and itself through 2 is created. To fix this, we store

a boolean `enableCaching` at each step, set initially to `true`; if `transitionStates[symbol] != NULL` for any symbol we have in $\mathcal{S}uff$, set `enableCaching` to `false`, as a `shallowCopy` call happens later, which might introduce cycles otherwise. In this case, instead of pointing directly to `cache[idx]`, we create a deep copy of it, which is faster than creating the start automaton from scratch. We present the modified code for creating a start automaton below in Algorithm 3.8.

```
State* cachedCreateStartAutomaton(qStart, qEnds, symTable, pattern,
                      idx, cache, enableCaching)
    if (cache[idx] != NULL)
        if (enableCaching)
            return cache[idx];
        else
            if (qStart == NULL)
                qStart = createState();
            deepCopy(qStart, cache[idx]);
            cache[idx] = qStart;
            return qStart;
    ...
    // same implementation as before, but recursive calls
    // are to cachedCreateStartAutomaton instead
    cache[idx] = qStart;
    return qStart;
```

Listing 3.8: Cached creation of an automaton

To create the fallback states correctly later, for the sub-automaton starting from S_0, we do not use the cached version of the start pattern automaton building. We finally present the implementation for `suffixLinking` below in Algorithm 3.9; consequently, the automaton generated using caching for the same example is the same as the one presented in Figure 3.3b. This happens because 3 total automatons are built, and the `shallowCopy` function is called twice by `splitStarts`; the "weakest" sub-automaton, starting in q_2 remains unchanged, but `shallowCopy`(q_4, q_2) and then, later, `shallowCopy`(q_6, q_4) are called, modifying the two other sub-automata. Having finished the initial phase of building the automaton, we move onto the second phase, the linking-starts phase.

3.3.2 Linking-Starts Phase

For the linking-starts phase, we want to allow for a start S_i starting at the ith index of a symbol to start later in the symbol, from the jth index, i.e., S_i has the transitions of S_j, for all $j > i$; through starts splitting, we previously ensured this behaviour for symbols through which both S_i and S_j transition. To reduce the number of states and transitions, first, we remove all the start states that do not have any transitions; as S_0 has at least one transition (through 255, as previously mentioned) and the other starts are not scoped within this class, we set the pointer to be `NULL` when the start has no transitions.

```
void suffixLinking(symbols, i, transitionStates, cache, qStarts,
        qEnds, symTable, pattern)
    disableCaching = std::any_of(
        symbols.begin(), symbols.end(), [&](uint8_t symbol) {
        return transitionStates[symbol] != NULL;
    });
    qPostStart = cachedCreateStartAutomaton(NULL, qEnds, symTable,
        pattern, i + 1, cache, !disableCaching);
    for (symbol: symbols)
        symIdx = symTable.symbols[symbol].size() - i - 1;
        qStarts[symIdx].addTransition(symbol, postStart);
        transitionStates[symbol] = qPostStart;
```

Listing 3.9: Suffix linking process

The algorithm for doing this is much simpler than in the case of common symbols: iterate from the highest to the lowest indexed starts; when in S_i, we keep a data structure which maps each symbol c to $\delta(S_j, c)$, where S_j can transition through c and j is as small as possible; all transitions corresponding to symbols through which S_i does not transition are then copied, and the data structure is updated using the transitions of S_i. We present the implementation of this algorithm in Algorithm 3.10, and we show the state of the automaton after finishing linking the starts for the example middle match in Figure 3.4b; the only change between Figure 3.3b and Figure 3.4b is some transitions from the start states. Having finished linking the start states, we now proceed to the last phase of creating the middle match automaton: creating the fallback states.

```
void linkStarts(qStarts)
    for (uint8_t idx = 1; idx < 9; ++idx)
        if (qStarts[idx].transitions.empty())
            qStarts[idx] = NULL;
    symToState = std::unordered_map<uint8_t, State*>{};
    for (qStart: reverse(qStarts))
        if (qStart == NULL)
            continue;
        for ([symbol, qNext]: symToState)
            if (!qStart.canTransition(symbol))
                qStart.addTransition(symbol, qNext);
        for ([symbol, qNext]: qStart.transitions)
            symToState[symbol] = qNext;
```

Listing 3.10: Link starts of a middle automaton

3.3.3 Creating Fallback States

Currently, the built automaton goes only forward, with the exception of the default transition, which is the start state of the automaton S_0. However, for Figure 3.4b, starting from S_0 and traversing the sequence $\{0, 1, 1\}$ from S_0 currently leads to S_0, but traversing the prefix $\{1, 1\}$

leads to q_3; as such, we must arrive in q_3 rather than S_0. For this, we define the prefix of a state as the set of all sequences of symbols through which we reach that state by starting from a start state S_i. We initially consider the case in which sub-automaton caching is disabled. Apart from the first symbol, which can take multiple values, all the sequences have the same symbols in the other positions. We then define the depth of a state as the length of its prefix sequences.

State	q_0	q_1	q_2	q_3	q_4	q_5	q_6
Prefix	$\{0\|1\|2\}$	$\{0\|1\|2,1\}$	$\{0\|1\|2,1,2\}$	$\{0\|1,1\}$	$\{0\|1\}$	$\{0\}$	$\{0,2\}$
Depth	1	2	3	2	1	1	2

Table 3.3: Prefix sequences for states of Figure 3.4b

The first symbol of the prefix taking multiple values does not affect the algorithm, as the value is not utilised anywhere. For an arbitrary state q and for all possible follow-up characters $x \in \{0, 1, \ldots, 255\}$ through which q does not transition, knowing q has the prefix $\{s_0^{(0)}|s_0^{(1)}|\ldots|s_0^{(k_0-1)}, s_1, s_2, \ldots, s_{k-1}\}$, find the smallest $k_1 > 0$ such that there is a state $q^{(fallback)}$ with prefix $\{s_{k_1}, s_{k_1+1}, \ldots, s_{k-1}, x\}$, and define $\delta(q, x) = q^{(fallback)}$. For example, for Figure 3.4b, q_3 has prefix $\{0|1,1\}$, so $\delta(q_3, 1) = q_3 \neq q_{def} = S_0$; similarly, $\delta(q_1, 1) = \delta(\delta(S_0, 1), 1) = q_3$.

Naively checking if there is a state with prefix $\{s_{k_1}, s_{k_1+1}, \ldots, s_{k-1}, x\}$ for all $k_1 > 0$ leads to a $\mathcal{O}(k^2)$ complexity. However, the following solution reduces the algorithm's runtime from quadratic to linear. For a state q_i having prefix $\{s_0^{(0)}|s_0^{(1)}|\ldots|s_0^{(k_0-1)}, s_1, s_2, \ldots, s_{k-1}\}$, choose $q_i^{(0)} = S_0$ and then, iteratively, $q_i^{(m+1)} = \delta(q_i^{(m)}, s_{m+1})$; we define now the trailing state $q_i^{(trail)} = q_i^{(k-1)}$. With this choice, we have $q_i^{(trail)}$ as the state having the prefix $\{s_{k_2}, s_{k_2+1}, \ldots, s_{k-1}\}$, with $k_2 > 0$ minimal; the transitions from $q_i^{(trail)}$ are then copied into q_i. Knowing $q_i^{(trail)}$ has depth at most $k-1$, we create all fallback states of states of such depth before traversing q_i; we show inductively why this choice makes sense.

For $k = 1$, the working state is a state immediately following a start state, and creating the fallback states implies for all x such that q_i does not transition through x, but S_0 does, we add $\delta(q_i, x) = \delta(S_0, x)$; the previous statement trivially holds.

Now, assume that the previous statement holds for all $k_0 < k$, and we aim to prove it also holds for k; pick an arbitrary x such that q_i does not transition through x. If there is no $1 \leq k_1 \leq k$ such that there is a state with the prefix $\{s_{k_1}, s_{k_1+1}, \ldots, s_{k-1}, x\}$, then $\delta(q_i, x) = S_0 = q_{def}$, so there is no need to add a transition. Otherwise, pick k_1 such that the fallback state $q_i^{(fallback)}$ has the prefix $\{s_{k_1}, s_{k_1+1}, \ldots, s_{k-1}, x\}$; then, k_1 is minimal. Our goal is to show $\delta(q_i^{(trail)}, x) = q_i^{(fallback)}$; $q_i^{(trail)}$ has the prefix $\{s_{k_2}, s_{k_2+1}, \ldots, s_{k-1}\}$, with k_2 minimal. The state preceding $q_i^{(fallback)}$ has the prefix $\{s_{k_1}, s_{k_1+1}, \ldots, s_{k-1}\}$, so, by the minimality of k_2, we conclude $k_2 \leq k_1$; if $k_1 = k_2$, $\delta(q_i^{(trail)}, x) = q_i^{(fallback)}$ trivially. Otherwise, the fallback state of $q_i^{(trail)}$, denoted by $q^{(fallback)}$, has the prefix $\{s_{k_1'}, s_{k_1'+1}, \ldots, s_{k-1}, x\}$ with $k_1' > k_2$ minimal; to ensure minimality of k_1', given the minimality of k_1, we have $k_1' = k_1$; we conclude that $q^{(fallback)} = q_i^{(fallback)}$, but $q^{(fallback)} = \delta(q_i^{(trail)}, x)$, so the hypothesis is proven; consequently, we only copy the transitions of $q_i^{(trail)}$.

There is an edge case for $q_{esc} = \delta(S_0, 255)$, which has the prefix is $\{255\}$; if the transitions of S_0 are copied in q_{esc}, then through $\{255, x\}$, rather than treating x as a raw byte, x is treated as

a symbol. Consequently, to avoid this behaviour, we set the trailing state of q_{esc} to be NULL, by convention. To ensure that, when reaching a state of depth $d+1$, the fallback states of states of depth $\leq d$ have already been created, we employ a breadth-first search algorithm, having in the queue at each step a pair $(q, q^{(trail)})$, where q copies the transitions of $q^{(trail)}$, and, then, both are moved forward through the same symbol. As each state is visited only once, we store a set of the states we have visited in the past; we present the implementation below in Algorithm 3.11, and we draw the final automaton in Figure 3.5.

```
void createFallbackStates(qStarts)
    queue = std::queue<std::tuple<State*, const State*>>{};
    queue.emplace(qStarts[0].transition(255), NULL);
    topoSorted = std::deque<State*>{};
    trailingMappings = std::unordered_map<const State*, const State*>{};
    trailingMappings[qStarts[0].transition(255)] = NULL;
    for (qStart: qStarts)
        for ([symbol, qNext]: qStart.transitions)
            if (traversed.contains(qNext) and allStates.contains(qNext))
                traversed.insert(qNext);
                queue.emplace(qNext, qStarts[0]);
    while (!queue.empty())
        [qState, qTrailing] = queue.front();
        queue.pop();
        for ([symbol, qNext]: qState.transitions)
            if (!trailingMappings.contains(qNext) and allStates.contains(qNext))
                qFallback =
                    transitionTrailing(qTrailing, symbol, trailingMappings);
                trailingMappings[qNext] = qFallback;
                queue.emplace(qNext, qFallback);
    for (qCur: topoSorted)
        qCur.copyTransitions(trailingMappings[qCur]);
```

Listing 3.11: Creating the fallback states

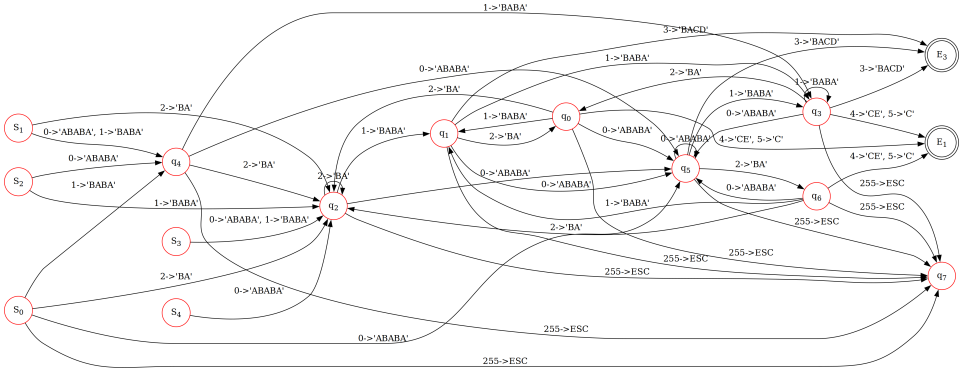


Figure 3.5: Final automaton for the pattern "%$ABABABAC$%" for the symbol table 3.2

To avoid adding transitions on the fly, we store a mapping for the trailing states and the

order in which to add them, which is determined by the order in which we traversed the states. Consequently, because the trailing state aims to use its trailing transitions when advancing, we implement the function `transitionTrailing` below in Algorithm 3.12; if the current state transitions through the required symbol, return that transitioned state. Otherwise, first check the trailing state for a transition, then the trailing state of the trailing state, and so on. The case where the current state is `NULL`, and the symbol is 255 occurs later. Finally, if no trailing state transitions through the symbol, we arrive in the default transition.

In the case of sub-automaton caching, the problem becomes more challenging, as a state has multiple such prefixes, given that we transition into a cached state from multiple sub-automata. Empirically, disabling caching when the transition state is not `NULL` helps against states having multiple trailing states.

```
State* trailingTransition(qCurrent, symbol, trailingMappings)
    if (qCurrent == NULL)
        return symbol == 255 ? NULL : qCurrent.defaultTransition;
    qTrailing = qCurrent;
    while (qTrailing != NULL)
        if (qTrailing.canTransition(symbol))
            return qTrailing.transition(symbol);
        qTrailing = trailingMappings[qTrailing];
    return qCurrent.defaultTransition;
```

Listing 3.12: Transition a trailing state

Having presented the algorithm for creating the fallback states, the last functionality before connecting everything is ensuring that there are no `NULL` entries in `qStarts`; if we start at the i-th index of a symbol, but S_i is `NULL`, then we must begin later in the symbol (i.e., from S_j with $j > i$), but as soon as possible (i.e., j is minimal). Consequently, we start in S_j, where $j > i$ as small as possible such that S_j is not `NULL`. If no such S_j exists, then we overflow and instead start in S_0 through the symbol following the current one; we skip the implementation of the algorithm, although the function has the signature `precomputeStartPositions(qStarts)`. We put everything together in the function building the automaton for the middle patterns completely, which we present below in Algorithm 3.13.

```
1  void buildMiddleAutomaton(qStarts, qEnds, symTable, pattern)
2      max_n = pattern.size() >= 8 ? 7 : pattern.size() - 1;
3      transitionStates = std::array<State*, 255>{NULL};
4      cache = std::vector<State*>{NULL, pattern.size()};
5      for (i = 1; i <= max_n; ++i)
6          [prefix, _] = split(pattern, i);
7          symbols = symTable.findWithSuffix(prefix);
8          if (symbols.empty())
9              continue;
10         suffixLinking(symbols, i, transitionStates, cache,
11             qStarts, qEnds, symTable, pattern);
12     middleLinking(qStarts, qEnds, symTable, pattern);
13     createStartAutomaton(qStarts[0], qEnds, symTable, pattern, 0);
14     splitStarts(qStarts);
15     if (!qStarts[0].canTransition(255))
16         qStarts[0].addTransition(255, createState());
17     linkStarts(qStarts);
18     createFallbackStates(qStarts, qEnds);
19     precomputeStartPositions(qStarts);
```

Listing 3.13: Build complete middle automaton

Having completed the construction of finite automata for single patterns without any wildcard characters, the next chapter addresses the extension of this approach to patterns containing underscores.

4 Underscore Pattern Matching

This chapter explains how to build an automaton matching a single sub-pattern containing underscores. Here, as the FSST-algorithm [BNL20] downcasts the data to bytes regardless of the original encoding of the data (i.e. ASCII, UTF-8), rather than an underscore matching any single character (potentially multiple-bytes long in the case of UTF-8 encoded data), an underscore matches any single byte. A consequence of the FSST-algorithm downcasting UTF-8 encoded data is that, for example, an emoji (one single UTF-8 character), bytewise represented as `0xF0 0x9F 0x91 0xA8 0xE2 0x80 0x8D 0xF0 0x9F 0x8F 0xAB`, is split into multiple symbols; these symbols are possibly invalid as standalone characters.

The equivalent patterns in SQL treated here are either "$p_0p_1 \dots p_{n-1}$%" (for start patterns), "%$p_0p_1 \dots p_{n-1}$%" (for middle patterns) and "%$p_0p_1 \dots p_{n-1}$" (for end patterns), where p_i is not a '%' wildcard, but there are '_' wildcards present in the pattern. As a convention, when splitting the pattern in such sub-patterns, we move all underscores initially situated in the beginning of a sub-pattern to the end of the previous sub-pattern. Therefore, for a middle pattern, $p_0 = $ '_' implies that it is the first sub-pattern, and we construct the automaton for the sub-pattern with that in mind. Moreover, we aim to have $p_0 \neq$ '_' in the case of end patterns; so, in the case we have only an end pattern which starts with an underscore, we move those underscores to a start pattern having only underscores.

The challenge in this section involves efficiently creating states for the underscore wildcards, whose number of transitions increases rapidly, and connecting the sub-automata created for the sub-subpatterns. Moreover, the algorithms presented in Chapter 3 are no longer valid due to the exact connections created between sub-automata. For the examples shown in this chapter, we utilise a previously built symbol table for the firstname dataset presented in the FSST paper [BNL20]. An example end automaton is shown in Figure 4.1, where we observe the high number of transitions in the automaton despite the simple pattern.

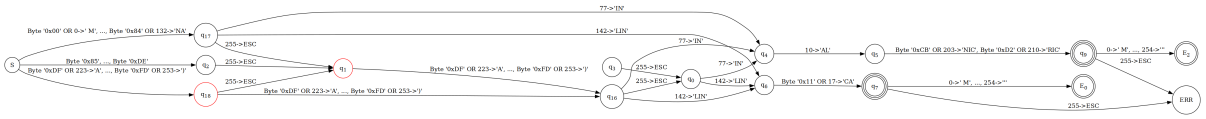


Figure 4.1: Automaton for the pattern "%CALIN__"

We store the pattern as a vector of string sub-subpatterns `std::vector<std::string> subpatterns` and a vector of unsigned integers `std::vector<uint8_t> numUnderscores`, representing the number of underscores between the sub-subpatterns. The first element `numUnderscores.front()` represents the number of underscores before all string sub-subpatterns; the last element `numUnderscores.back()` is the number of underscores after all the string sub-subpatterns. Otherwise, `numUnderscores[i]` represents the number of underscores between `subpatterns[i - 1]` and `subpatterns[i]`. Having explained all the prerequisites, we start this chapter by explaining how to create states for underscore wildcards.

4.1 Creating States for Underscore Symbols

We observe that, when creating the automaton for such patterns, underscore wildcards are treated differently from other symbols. Moreover, the behaviour of creating these states changes depending on whether they have any string sub-patterns following or not.

For the most part, the algorithm for creating the underscore states is the same: we create a state for each underscore wildcard and another state acting as the default escape state for the previously mentioned state; for the rest of the section, we call the previously described states **underscore**, respectively **escaped underscore** states. Subsequently, for each state and for all symbols, determine the state where it transitions through that symbol; the transition is not always possible. This function has the signature `createUnderscoreZone`; the boolean `reqExact` indicates whether the function is called for the last entry of `numUnderscores` or not. We present the implementation below, in Algorithm 4.1, and the previously mentioned parameter is used for the `addExactEndTransition` function call, determining the state into which an underscore state transitions through a non-escaped symbol; we discuss its implementation later.

```
std::vector<State*> createUnderscoreZone(num, reqExact, qEnds)
    qUnderscores = std::vector<State*>{num, NULL};
    for (i = numUnderscores - 1; i >= 0; --i)
        qUnderscores[i] = createState();
        qEscape = createState();
        escapeDestination = i == numUnderscores - 1 ? qEnds[0] : qUnderscores[i + 1];
        for (byte = 0; byte < 255; ++byte)
            qEscape.addTransition(byte, escapeDestination);
        qUnderscores[i].addTransition(255, qEscape);
        for (sym = 0; sym < symTable.nSymbols; ++sym)
            symLen = symTable.symbols[sym].size();
            if (symLen != 0)
               qUnderscores[i].addExactEndTransition(sym, 1, symLen - 1, qEnds,
                       qUnderscores.data() + i + 1, num - i - 1, reqExact);
    return qUnderscores;
```

Listing 4.1: Creating the underscore states

The underscore states created, and the end states are consecutive (i.e., one byte away from each other); given that the sequence $\{255, c\}$ consumes any one single byte, it transitions `qUnderscores[i]` to either `qEnds[0]` (if $i = num - 1$, i.e. there are no underscore states of higher index), or to `qUnderscores[i + 1]` otherwise.

For other symbols, we call the function handling adding transitions to the end states, both in the exact and inexact cases, when there is (a potentially empty) vector of underscore states in the middle; we present this function in Algorithm 4.2. If there are exactly enough bytes in the symbol as the number of underscore states (`qUnderscoreCnt == restBytes`, both of which can be 0), we land in `qEnds[0]`. Otherwise, if the underscore states consume all the bytes of the symbol inexactly (`qUnderscoreCnt > restBytes`), we add a transition to the corresponding underscore state; note that it does not matter if we require an exact match, as these transitions are exact by default. Moreover, all transitions landing in other underscore states are valid due to the order in which we create the underscore states.

If $i =$ `numUnderscores.size()` $- 1$, we connect the state to the end state corresponding to the index in the symbol after consuming the underscore wildcards. Otherwise, we are careful

that the transitions lead to valid exact matches; for this, a prerequisite is that `qEnds` contains the pointers to the start states created for a special type of multiple starts finite automaton for `subpatterns[i + 1]`, which is presented later. Instead of doing $\delta(q, c) = E_i$, where E_i is the end state from `qEnds` corresponding to the i-th index in the symbol c and, subsequently, the i-th start state for the following automaton, we say $\delta(q, c) = \delta(E_i, c)$ if E_i transitions through c (otherwise, we add no transition); this means we start earlier in the symbol, but we end in the same spot as before.

```
1  void addExactEndTransition(sym, endIdx, restBytes, qEnds,
2                        qUnderscore, qUnderscoreCnt, reqExact)
3      if (qUnderscoreCnt == restBytes)
4          addEndTransition(sym, qEnds, 0);
5          return;
6      if (restBytes < qUnderscoreCnt)
7          addTransition(sym, qUnderscores[restBytes]);
8          return;
9      idx = endIdx + qUnderscoreCnt;
10     if (reqExact)
11         if (!qEnds[idx].canTransition(sym))
12             return;
13         addTransition(sym, qEnds[idx].transition(sym));
14     else
15         addEndTransition(sym, qEnds, idx);
```

Listing 4.2: Creating the underscore states

Line 13 is necessary because, if `reqExact` is set to true, there is a sub-automaton following the one we are currently building; as such, assume it matches "$p_{k_0}p_{k_0+1}\dots p_{k-1}$", with $p_{k_0} \neq$ '_' and, for simplicity, denote $i :=$ `endIdx`, $j :=$ `qUnderscoreCnt` and $c :=$ `sym`. As S_{i+j} transitions through c, by the building process, "$c_{i+j}c_{i+j+1}\dots c_{len(c)-1}$" matches "$p_{k_0}p_{k_0+1}\dots p_{k_1}$", where $k_1 := k_0 + len(c) - i - j - 1$. As the previous j characters are underscore wildcards, matching any single byte, then "$c_ic_{i+1}\dots c_{len(c)-1}$" matches "$p_{k_0-j}p_{k_0-j+1}\dots p_{k_1}$" and, due to the context, "$c_0c_1\dots c_{i-1}$" matches "$p_{k_0-i-j}p_{k_0-i-j+1}\dots p_{k_0-j-1}$"; we conclude the any sequence of symbols starting at the state q for which we added the transition through q matches the start pattern "$p_{k_0-i-j}p_{k_0-i-j+1}\dots p_{k-1}$". Having explained the new method for adding a transition to end states and the process for creating states for underscore wildcards, we now delve into the algorithm for constructing an automaton for a start underscore pattern.

4.2 Start Match Automata

Before presenting the algorithm for building the automaton for start matches, we discuss the scope of the start states $S \neq S_0$; since we aim not to split a symbol during parsing, such states are omitted in the final automaton. Consequently, our goal is to determine how long such a start state must exist before being de-allocated. Instead of reallocating states multiple times, we allocate them once and then clear the transitions of these states, reusing them.

We mentioned that we construct the sub-automata for underscore sub-patterns starting from the last to the first string sub-pattern, as Algorithm 4.1 requires `qEnds` to correspond to the start states of the next sub-automaton, apart from the last one. Therefore, when creating the sub-automaton for the k-th sub-pattern, the start states of the $(k+1)$-th sub-automaton are supposed to still be valid, but are free to reuse after the end of the building phase; as

such, we reuse the starts of the $(k+1)$-th sub-automaton when creating the $(k-1)$-th one. We assume a struct `TemporaryStarts` does this automatically, and calling `get()` returns a set of start states with no transitions, which are currently not in use.

For this algorithm, we create a specialisation of the `createStartAutomaton` function call named `exactCreateStartAutomaton`, taking as additional parameters a boolean `reqExact` specifying whether the transitions to the end are exact or not, and a vector of underscore states `qUnderscores`; the differences between this and `createStartAutomaton` are:

- for $n = 1$ (`idx = pattern.size() - 1`), we replace Line 8 from Algorithm 3.3 by

```
qStart.addExactEndTransition(sym, 1, sym.size() - 1, qEnds,
    qUnderscores.data(), qUnderscores.size(), reqExact)
```

- for $n = 2$ (`idx = pattern.size() - 2`), we replace Line 8 from Algorithm 3.4 by

```
qStart.addExactEndTransition(sym, 2, sym.size() - 2, qEnds,
    qUnderscores.data(), qUnderscores.size(), reqExact)
```

- for $n \geq 3$ (`idx <= pattern.size() - 3`), we replace Line 15 from Algorithm 3.5 by

```
qStart.addExactEndTransition(symIdx, pattern.size() - idx, size + idx -
pattern.size(), qEnds, qUnderscores.data(), qUnderscores.size(), reqExact)
```

Similarly, we create a function `exactCachedCreateStartAutomaton`, where we replace the recursive calls to `createStartAutomaton` from Algorithm 3.8 with recursive calls to `exactCreateStartAutomaton`, and `exactMiddleLinking`, where we replace Line 11 from Algorithm 3.7 by

```
qStarts[i].addExactEndTransition(symIdx, endIdx, sym.size() - endIdx, qEnds,
    qUnderscores.data(), qUnderscores.size(), reqExact).
```

To ensure the minimality of the built automaton, as we create no fallback states in this case, we set `enableCaching = true` all the time; therefore, the specialisation of Algorithm 3.9 for underscore patterns does not require a `transitionStates` parameter, removes Line 12, and replaces Lines 7 and 8 with

```
qPostStart = exactCachedCreateStartAutomaton(NULL, qEnds, symTable, pattern, i
    + 1, cache, true, qUnderscores, reqExact).
```

The signature of the previously mentioned function is

```
exactSuffixLinking(symbols, i, cache, qStarts, qEnds, symTable, pattern,
    qUnderscores, reqExact).
```

Having introduced all functionalities, we now present the function that builds the sub-automaton for a sub-pattern, as shown below in Algorithm 4.3. For simplicity, the function returns the start states of the middle automaton, allowing them to be reused later. Accordingly, we build the entire automaton as follows: we create the underscore states for the $(k+1)$th entry in `numUnderscores`, where `k = subpatterns.size()`; we pass them to the `buildCurrent` function call, along with the end states `qEnds`. Afterwards, we update `qEnds` as the start states of the newly built sub-automaton, and we create the underscore states for the kth entry in `numUnderscores`, and so on.

```
std::vector<State*> buildCurrent(pattern, symTable, qTempStarts, qEnds, qUnderscores, reqExact)
    qStarts = std::vector<State*>{9, NULL};
    qStarts[0] = createState();
    for (i = 1; i < 9; ++i)
        qStarts[i] = &qTempStarts[i - 1];
    max_n = pattern.size() >= 8 ? 7 : pattern.size() - 1;
    cache = std::vector<State*>{NULL, pattern.size()};
    for (i = 1; i <= max_n; ++i)
        [prefix, _] = split(pattern, i);
        symbols = symTable.findWithSuffix(prefix);
        if (symbols.empty())
            continue;
        exactSuffixLinking(symbols, i, cache, qStarts, qEnds, symTable,
            pattern, qUnderscores, reqExact);
    exactMiddleLinking(qStarts, qEnds, symTable, pattern, qUnderscores, reqExact);
    exactCachedCreateStartAutomaton(qStarts[0], qEnds, symTable, pattern,
        0, cache, true, qUnderscores, reqExact);
    return qStarts;
```

Listing 4.3: Building the automaton for a sub-pattern

When reaching the first entry of `subpatterns`, there are two possibilities:

I. the first entry of `numUnderscores` is 0 (i.e. there are no underscores in front of the first sub-pattern). In this case, we create a start sub-automaton for the current sub-pattern, and we set the start state of the automaton as the start state of the newly created start sub-automaton;
II. the first entry of `numUnderscores` is not 0. In this case, we build a middle sub-automaton for the first sub-pattern using the previous algorithm, and then we create the underscore states for `numUnderscores[0]`. The start state of the automaton is now the first state in the created underscore states (i.e. `qUnderscores[0]`).

We present the full algorithm in Algorithm 4.4; for simplicity, rather than adding 255 transitions to the same state q_{dest} for escaped underscore states q, we set the default transition of q to q_{dest}. To observe an example, we use the fact that the previously mentioned symbol table for the firstname dataset has the following properties:

- $sym_{17} = "CA", sym_{142} = "LIN", sym_{242} = "P", sym_{236} = "O"$;
- $sym_{223}, sym_{224}, \ldots, sym_{254}$ are of length 1;
- no symbol of length ≥ 2 bytes ends with $'P'$;
- no symbol starts with "PO", "OP" or "CAL";
- $sym_{40}, sym_{61}, sym_{71}, sym_{108}, sym_{175}, sym_{188}$ start with $'P'$.

For the start pattern "$CALIN\_POP$", we present the automaton Algorithm 4.4 builds in Figure 4.3; indeed, any encoded string matching this pattern starts with $\{17, 142\}$. Given that no symbol of length ≥ 2 ends with P, there are two types of sequences reaching the start of the automaton for the subpattern "POP" (q_0): $\{255, c\}$, achieved through q_4, or one-byte symbols. Using Property 2.1, any encoded string starting with "POP" starts with $\{242, 236\}$; finally, q_1 transitions through all symbols starting with $'P'$ to E_1.

```
State* buildStartAutomaton(
    subpatterns, numUnderscores,
    symTable, qEnds
)
    tempStarts = TemporaryStarts{};
    cnt = subpatterns.size();
    for (idx = cnt - 1; idx >= 1; -idx)
       reqExact = (idx != cnt - 1);
       qUnderscores = createUnderscoreZone(
          numUnderscores[idx + 1], reqExact,
          qEnds
       );
       qStarts = tempStarts.get();
       qEnds = buildCurrent(
          subpatterns[idx],
          symTable, qStarts, qEnds,
          qUnderscores, reqExact
       );
    reqExact = numSubpatterns > 1;
    qUnderscores = createUnderscoreZone(
      numUnderscores[1], reqExact,
      qEnds
    );
    qStart = NULL;
    if (pattern.numUnderscores[0] == 0)
        qStart = exactCreateStartAutomaton(
            NULL, qEnds, symTable,
            subpatterns[0], 0, qUnderscores,
            reqExact
        );
    else
        qStarts = tempStarts.get();
        qEnds = buildCurrent(
            subpatterns[0],
            symTable, qStarts, qEnds,
            qUnderscores, reqExact
        );
        qUnderscores = createUnderscoreZone(
            numUnderscores[0],
            true,
            qEnds
        );
        qStart = qUnderscores[0];
    return qStart;
```

Listing 4.4: Building the automaton for a start pattern

```
std::vector<State*> buildFwdEndAutomaton(
    subpatterns, numUnderscores,
    symTable, qStart, tempStarts,
    allqUnderscores
)
    cnt = subpatterns.size();
    qEnds = std::vector<State*>{9, NULL};
    qEnds[0] = qStart;
    for (idx = cnt - 1; idx >= 0; --idx)
        qUnderscores = createUnderscoreZone(
            numUnderscores[idx + 1], true,
            qEnds
        );
        allqUnderscores.push_back(
            qUnderscores
        });
        qStarts = tempStarts.get();
        qEnds = buildCurrent(subpatterns[idx],
            symTable, qStarts, qEnds,
            qUnderscores, true
        );
    return qEnds;
```

Listing 4.5: Building the forward automaton for an end pattern

```
std::vector<State*> buildMiddleAutomaton(
    subpatterns, numUnderscores,
    symTable, qEnds
)
qStart = createState();
...
qStarts = tempStarts.get();
qStarts = buildFirst(subpatterns[0],
    symTable, qStarts, qEnds,
    qStart, qUnderscores,
    reqExact
);
if (numUnderscores[0] != 0)
    qUnderscores = createFirstUnderscoreZone(
        numUnderscores[0], qStarts
    );
    qStarts[0] = qUnderscores[0];
return qStarts;
```

Listing 4.6: Building the automaton for a middle pattern

Figure 4.2: A side by side comparison of the algorithms building underscore automata

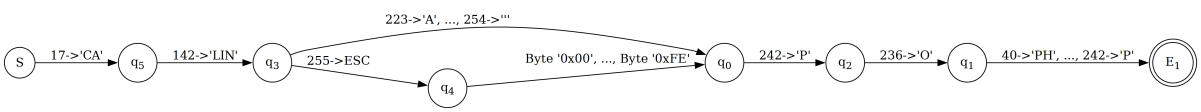


Figure 4.3: Automaton for the pattern "CALIN_POP%"

Having finished building automata for underscore start patterns, we now delve into the other single-start automaton we built, the automata created for end underscore patterns.

4.3 End Match Automata

We aim to do a two-way search eventually, with the end match automaton moving backwards in the compressed string; the other connected automata move forward in the string. In the case of underscore patterns, we initially create a multiple-starts-forward automaton, which we then invert into a backwards automaton.

The algorithm for building the forward automaton is similar to Algorithm 4.4. One key difference is that we must end exactly at the end of the compressed string, achieved by having `qEnds[i] = NULL`, for $i \neq 0$, and setting `reqExact = true`; by doing so, Algorithm 4.2 does not add any transitions unless it transitions exactly to `qEnds[0]`. The other key difference is that all sub-patterns are treated as middle patterns, and the end pattern does not start with underscores. We present this algorithm in Algorithm 4.5. Afterwards, we have a finite automaton with multiple start states `qEnds` and a unique end state `qStart`; we pass `tempStarts` to the function so that the start states do not go out of scope when the function finishes.

In general, inverting a deterministic finite automaton returns a non-deterministic one [BT14]; for this, we first observe that any symbol c through which an escaped underscore state transitions is the raw byte c rather than the c-th symbol, and that all underscore states are returned, at some point, by the `createUnderscoreZone` function call.

Using the automaton caching mechanism at all steps implies that the string suffix (i.e., what is left to match from the original pattern) of a state q is "$p_l p_{l+1} \dots p_{n-1}$" for some $l > 0$. In the case $p_{l-1} = \text{'\_'}$, q is the i_{l-1}th underscore state of the j_{l-1}th `createUnderscoreZone` call, where the indexes depend only on $l-1$, yielding the uniqueness; otherwise, let i_{l-1} be the index of p_{l-1} in the corresponding sub-pattern, then q is returned by `exactCachedCreateStartAutomaton` with the index i_{l-1}; thus, unique by having caching always enabled.

Using the previous result, we show that if there exist some states $q, q_0 \neq q_1$ and a byte c such that $\delta(q_0, c) = \delta(q_1, c) = q$, then c represents the c-th symbol when transitioning from q_0 and is escaped to represent the c-th byte when transitioning from q_1 (or conversely); otherwise, q_0 and q_1 have the same string suffix. Moreover, by Property 2.3, q is the successor of an underscore state, so q_1 is an escaped underscore state; therefore, the default transition of q_1 is q rather than ε.

For inverting the automaton, we create two additional data structures: the first is a queue `queue` of states for the breadth-first search algorithm. To avoid changing the transitions on the fly, we store a map from each state to its new set of inverted transitions `newTransitions` (δ_{new}). This map is also used to avoid traversing a state twice; a state that is used as a key in this mapping has been previously visited.

We set initially $\delta_{new}(E_i', 255) = \varepsilon$, for all pseudo-ends E_i', in case we copy E_i' later. Subsequently, we iterate through all transitions of the starts S_i of the previously constructed multiple starts finite automaton $\delta(S_i, c) = q$, and we redirect q to the corresponding pseudo-end $\delta_{new}(q, c) = E_i'$. If we have not already added q to `queue`, we do so.

Having traversed the states whose transitions are not reversed, but rather redirected, we traverse the automaton freely. For each state q in the queue, there are two possibilities:

I. the default transition of q is $q_0 \neq \varepsilon$, equivalent to q being an escaped underscore state; since q has no actual transitions, we simply add q_0 to the queue. The actual inverting of transitions is done later;

II. the default transition of q is ε. For each transition $\delta(q, c) = q^{(0)}$, if we have not visited $q^{(0)}$, we add it to the queue and, to invert the transition, $\delta_{new}(q^{(0)}, c) = q$. Notice that the value of $\delta_{new}(q^{(0)}, c)$ is set only once: for any other q' such that $\delta(q', c) = q^{(0)}$, q' is an escaped underscore state, and, for escaped underscore states, we do not set δ_{new} explicitly.

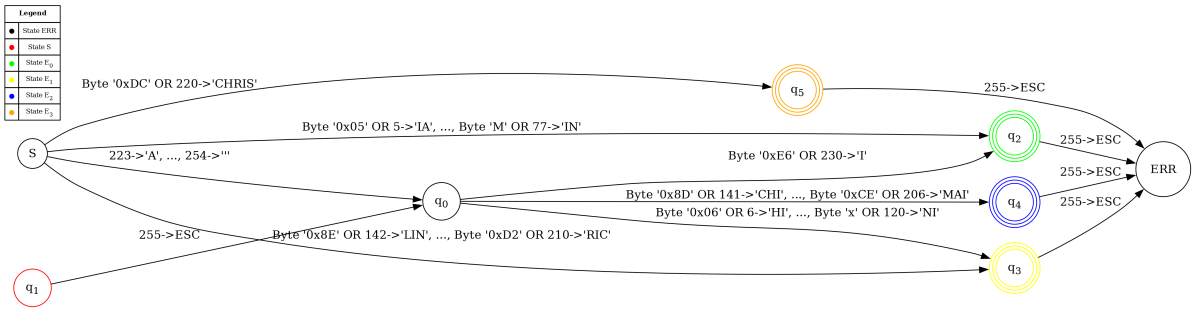
Finally, after handling the potential transition clashes, we set $\delta(q, c) = \delta_{new}(q, c)$ for all states q and all symbols c; the algorithm for inverting the automaton is presented in Algorithm 4.7.

```
1  void invert(qStart, qEnds, qErr, pseudoEnds, allqUnderscores)
2      queue = std::queue<State*>{};
3      newTransitions =
4          std::unordered_map<State*, std::unordered_map<uint8_t, State*>>{};
5      for (qPseudoEnd: pseudoEnds)
6          newTransitions[qPseudoEnd][255] = qErr;
7      enqueue = [&newTransitions, &queue] (qNext) {
8          if (!newTransitions.contains(qNext))
9              newTransitions[qNext] = std::unordered_map<uint8_t, State*>{};
10             queue.emplace(qNext);
11     }
12     for (state: qEnds)
13         for ([symbol, qNext]: state.transitions)
14             enqueue(qNext);
15             newTransitions[qNext][symbol] = pseudoEnds[idx];
16     while (!queue.empty())
17         qCurrent = queue.front();
18         queue.pop();
19         if (qCurrent.defaultTransition != qErr)
20             enqueue(qCurrent.defaultTransition);
21             continue;
22         for ([symbol, qNext]: qCurrent.transitions)
23             enqueue(qNext);
24             newTransitions[qNext][symbol] = qCurrent;
25     handleClashes(defaults, newTransitions);
26     for ([state, transitions]: newTransitions)
27         state.transitions = transitions;
```

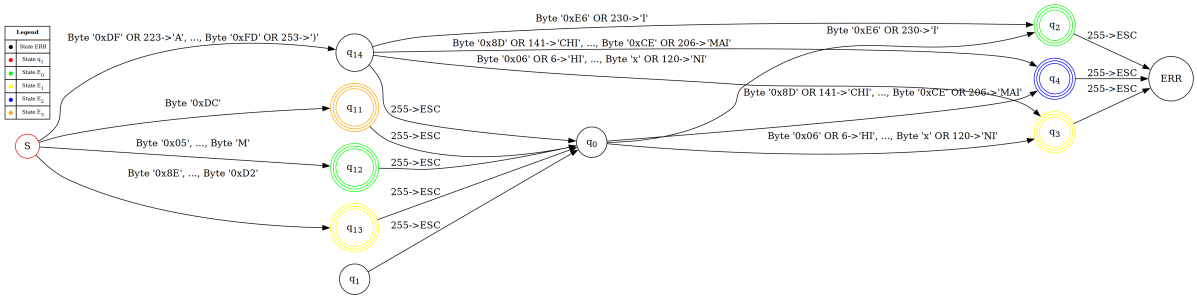
Listing 4.7: Inverting the multiple starts finite automaton

To handle transition clashes, we utilise the fact that we have the transitions in both directions. For a state q with an initial default transition of $q^{(0)}$, i.e. $\delta(q, c) = q^{(0)}$, for all c, the trivial method of inverting is to set the default transition of $q^{(0)}$ to q. We present an automaton in Figure 4.4a, built for the end pattern "$I\_$" with the previously explained approach; by only changing the default transition of S, a sequence of symbols ending with $\{255, 223\}$ leads to the error state, which is not the intended behaviour. Similarly, adding a transition from $q_0 = \delta_{new}(S, 223)$ to $\delta_{new}(q_1, 255) = q_0$ breaks the acyclicity of the automaton. As a consequence,

we copy the state q_0 into a state q_{14}, and $\delta_{new}(S, c) = q_{14}$, for $c \in \{223, 224, 225, \ldots, 253, 254\}$; finally, $\delta_{new}(q_{14}, 255) = q_0$.



(a) Initial end automaton for the pattern "$I\_$"



(b) Final end automaton for the pattern "$I\_$"

For this, we know all states with non-error state default transitions are the escaped underscore states; by picking an underscore state q, the default transition of $q' = \delta(q, 255)$ is $q_{def} \neq \varepsilon$. For each such triplet, firstly, we set the default transition of q_{def} to q', and the new default transition of q' to ε. Afterwards, for each $q_{def}^{(0)}$ that q_{def} transitions to in the inverted automaton, we store the list of symbols for which this happens. For example, for Figure 4.4a, there is only one underscore state $q := q_0$ with $q' := \delta(q, 255) = q_1$ and the default transition of q' is $q_{def} := S$; the mapping will be, as such, $\{q_5 : [220], q_2 : [5, 12, 13, \ldots, 77], q_0 : [223, 224, 225, \ldots, 254], q_3 : [142, 143, 144, \ldots, 210]\}$.

For each different state $q_{def}^{(0)} = \delta_{new}(q_{def}, c)$ for some c, if it is transitioned into only through q_{def} in the inverted automaton, we copy the transition $\delta_{new}(q_{def}^{(0)}, 255) = \delta_{new}(q', 255) = q$; otherwise, since the state is transitioned into from other states in the inverted automaton, we copy it into $q_{def}^{(1)}$ by copying its new transitions from the `newTransitions` mapping, its default transition and its pseudo-end index (not necessarily containing a value). Thereafter, $\delta_{new}(q_{def}^{(1)}, 255) = \delta_{new}(q', 255) = q$ and, finally, for all c such that $\delta_{new}(q_{def}, c) = q_{def}^{(0)}$, we redirect the transition $\delta_{new}(q_{def}, c) = q_{def}^{(1)}$. We know $q_{def}^{(0)}$ is transitioned in the inverted automaton only from q_{def} using that $\delta_{new}(q, c) = q'$ if and only if $\delta(q', c) = q$, consequently, by checking whether $\delta(q_{def}^{(0)}, c)$ is the same for all possible symbols c. We present the algorithm below in Algorithm 4.8.

A potential problem with the previous algorithm is calling the function `copyState` for an arbitrary state q, and then changing $\delta_{new}(q, c)$, for some c, at some later point. We avoid this by traversing the escaped underscore states in a topologically sorted manner.

```
void handleClashes(allqUnderscores, newTransitions, qErr)
    copyState = [&newTransitions](state){
        copy = createState();
        newTransitions[copy] = newTransitions[state];
        copy.defaultTransition = state.defaultTransition;
        copy.endIdx = state.endIdx;
    };
    for (qUnderscore: allqUnderscores)
        qDefault = qUnderscore.transition(255);
        qSource = qDefault.defaultTransition;
        qSource.defaultTransition = qDefault;
        qDefault.defaultTransition = qErr;

        transitioningFromSource = std::unordered_map<State*, std::vector<uint8_t>>{};
        for ([symbol, qDestination]: newTransitions[qSource])
            transitioningFromSource[qDestination].push_back(symbol);
        for ([qDestination, symbols]: transitioningFromSource)
            isUnique = std::all_of(
                qDestination.transitions.begin(), qDestination.transitions.end(),
                [qSource] (kv) { return kv.second == qSource; }
            );

            qTarget = qDestination;
            if (!isUnique)
                qTarget = copyState(qDestination);
                newTransitions[qTarget] = newTransitions[qDestination];
                for (symbol: symbols)
                    newTransitions[qSource][symbol] = qTarget;
            newTransitions[qTarget][255] = qUnderscore;
```

Listing 4.8: Handle clashing transitions when inverting an automaton

For the previous example, $\delta_{new}(q_0, 230) = q_2$, $\delta_{new}(q_1, 255) = q_0$ and $\delta_{new}(q_0, 6) = q_3$. Therefore, we copy q_2 into q_{12}, q_0 into q_{14} and q_3 into q_{13}; we notice that copies of pseudo-ends are pseudo-ends as well, with the same index. However, no other state transitions directly to q_5, so, instead of copying it, we add the transition $\delta_{new}(q_5, 255) = q_0$; the final automaton is presented in Figure 4.4b. Having completed the construction of automata for start and end underscore patterns, we now proceed to describe the algorithm for building the automaton for a middle underscore pattern.

4.4 Middle Match Automata

The algorithm building an automaton for a middle underscore pattern is also similar to Algorithm 4.4. However, one key difference is that the default transition of automata states is not the error state ε, but rather the start state of the automaton created for the first sub-pattern; consequently, we create this start state at the beginning of the building phase. In case `numUnderscores[0]` is not 0, the default transition is not the first underscore state because we never fallback before the first sub-pattern, as we have already matched the bytes for those underscore wildcards.

To build the sub-automata for sub-patterns which are not the first sub-pattern, we use Algorithm 4.3. For the first sub-pattern, we implement algorithms inspired by Section 3.3. Disabling caching when creating a start sub-automaton ensures that the new sub-automaton is built from scratch, rather than reusing cached states. This guarantees that the only shared states are those related to the final states, preventing unintended state overlaps during construction. However, the current algorithm does not differentiate between `qEnds`, which represent the start states of the second sub-automaton and the actual end states.

To preserve the intended behaviour, before creating an automaton with caching turned off, we create a data structure `endMappings` that maps a state from any other sub-automaton to an equivalent new state, created specifically for the current sub-automaton. When transitioning to a state q that lies outside of the current sub-automaton but belongs to the automaton built for the pattern (i.e., using `qEnds`), we recursively deep copy q into q' and set the mapping of q to be q', if we have not done that already. The cache is used for write-only purposes in this case, and the function incorporating this behaviour has the signature

```
exactNonCachedCreateStartAutomaton(qStart, qEnds, symTable, pattern, i, cache,
                                   qUnderscores, reqExact, endMappings).
```

The general structure of the `buildFirst` algorithm is the same as Algorithm 3.13, with some changes to the functions called. For this type of sub-automaton, we handle suffix linking with `exactSuffixLinking`; we enable caching because the function splitting the starts presented later cannot generate cycles anymore. For middle linking, we use `exactMiddleLinking`. Afterwards, since the fallback states start prefix parsing from S_0, we should avoid cycles in our fallback creation algorithm. Consequently, we disable caching for building the start sub-automaton from the start of the pattern by replacing Line 13 of Algorithm 3.13 with

```
std::unordered_map<State*, State*> endMappings{};
exactNonCachedCreateStartAutomaton(qStarts[0], qEnds, symTable, pattern, 0,
    cache, qUnderscores, reqExact, endMappings);
```

Splitting starts is more complicated for this type of automata, as the underscore states act as bottleneck states connecting different sub-automata; consequently, with the previous definition of a prefix, due to the aggressive caching used, an arbitrary state q has multiple prefixes of the form

$$\{s_0^{(0)}|s_0^{(1)}|\dots s_0^{(l_0-1)}, s_1, \dots, s_{k_0}^{(0)}|s_{k_0}^{(1)}|\dots|s_{k_0}^{(l_1-1)}, s_{k_0+1}, \dots, s_{k_1}^{(0)}|s_{k_1}^{(1)}|\dots|s_{k_1}^{(l_2-1)}, \dots, s_{k-1}\}.$$

For example, for the initial automaton built for the pattern "%N_S%", presented in Figure 4.5, the state q_5 has the prefix $\{225, 255, 0|1|\dots|254\}$ through the path $S_0 \to q_4 \to q_6 \to q_5$ and the prefix $\{225, 223|224|\dots|254\}$ through the path $S_0 \to q_4 \to q_5$. Algorithm 3.6 does not behave as intended in this case anymore due to the multiplicity of prefixes: although $\delta(S_0, 35) = q_3$ and $\delta(S_1, 35) = q_1$, copying the sub-automaton starting at q_1 in the sub-automaton starting at q_3 leads to the sequence $\{56, 12\}$ (decoded as "$NOIS$") matching the pattern. This happens because $sym_{35} =$ "NN" and the second symbol of the pattern is an underscore wildcard rather than a character; as a consequence, we should not start from the first index of the 56th symbol, since $sym_{56}[1] = 'O' \neq 'N'$. The solution is to copy q_3 into a state q_8, modify $\delta(S_0, 35) = q_8$ and copy the sub-automaton starting at q_1 into the sub-automaton starting at q_8.

The algorithm finds all possible combinations `(qCurrent, qSource)` by traversing the automaton exhaustively through all symbols; for each such tuple, we traverse it only once.

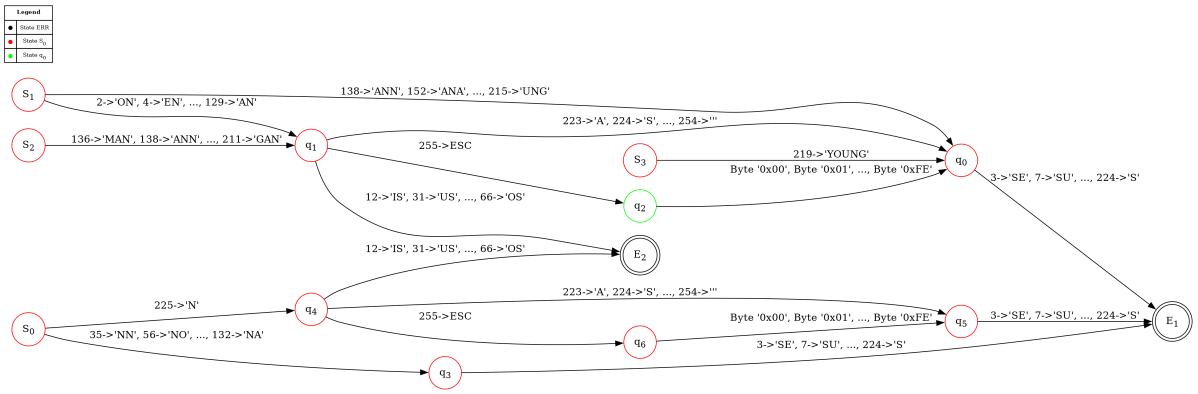


Figure 4.5: Initial middle automaton for the pattern "%N_S%"

For simplicity, define a struct `QueueObject` containing two states q_1, q_2 and a symbol; here, we store tuples `(qPrev, symbol, qSource)`, where the transition of `qPrev` through `symbol` is `qCurrent`; we explain the underlying reason below.

To minimise the number of state copies created, we store a data structure `parentMappings` mapping each state to the state from which it was originally copied; for all states q in the initial automaton, we have that $parent(q) = q$. The parent-child relation extends transitively over copying operations: if $parent(q') = q$ and q'' is a copy of q', then $parent(q'') = q$. When we traverse the pair (q_{prev}, c, q_{src}), with $q_{cur} = \delta(q_{prev}, c)$ and $q_{parent} = parent(q_{cur})$, we have two possibilities:

I. we have not previously reached a copy of q_{parent} with the intent to copy q_{src}; as a consequence, we copy q_{parent} into q_{copy} and set $parent(q_{copy}) = q_{parent}$. Afterwards, we set q_{copy} as a state copying q_{src} in the future and redirect $\delta(q_{prev}, c) = q_{copy}$; this redirection is the reason why we add q_{prev} to the queue and not q_{cur};
II. we have reached a copy q_{copy} of q_{parent} with the intent to copy q_{src} in the past; we simply redirect $\delta(q_{prev}, c) = q_{copy}$ in this case.

The data structure for the previously explained task is `copyMappings` mapping each pair of states (q_{parent}, q_{src}) to the copy q_{copy} of q_{parent}, copying the transitions of q_{src} later. For each start state, we call the `splitOne` function, handling the splitting of the current start, given `transitionStates` and the previous two data structures, shared between all `splitOne` function calls; the copying of states suggests, at least intuitively, that cycles should not arise.

For faster traversal, we store a set of the visited triplets (q_{prev}, c, q_{src}), as there is no point traversing such a triplet twice; the actual copying of the source states is done after traversing the sub-automaton and, for this, a mapping `copies` is stored, signalling for each newly created state q_{copy} the state q_{src} whose transitions it copies. We present the entire algorithm for splitting one start in Algorithm 4.9; the algorithm for splitting all starts is now trivial: iterate backwards through the starts, split the current start S_i using `splitOne`, and then update `transitionStates` with the transitions of S_i. This function has the signature `linkUnderscoreStarts(qStarts)`, and the automaton after splitting the starts for the previous example is shown in Figure 4.6.

```
void splitOne(qStart, parentMappings, transitionStates, copyMappings)
    queue = std::queue<QueueObject>{};
    copies = std::unordered_map<State*, const State*>{};
    visited = std::unordered_set<QueueObject>{};
    for ([symbol, qNext]: qStart.transitions)
        if (symbol == 255 or transitionStates[symbol] == NULL)
            continue;
        queueObj = QueueObject{start, symbol, transitionStates[symbol]};
        queue.push(queueObj);
        visited.insert(queueObj);

    while (!queue.empty())
        [qPrev, symbol, qSrc] = queue.front();
        queue.pop();
        qCur = qPrev.transition(symbol);
        qParent = parentMappings.contains(qCur) ? parrentsMapping[qCur] : qCur;

        if (copyMappings[qParent].contains(qSrc))
            qDest = copyMappings[qParent][qSrc];
            qPrev.transitions[symbol] = qDest;
            qCur = qDest;
        else
            qNew = createState();
            qNew.transitions = qCur.transitions;
            copies[qNew] = qSrc;
            parentMappings[qNew] = qParent;
            copyMappings[qParent][qSrc] = qNew;
            qPrev.transitions[symbol] = qNew;
            qCur = qNew;

        for ([symbol, qNext]: qCur.transitions)
            if (qSrc.canTransition(symbol) and allStates.contains(qNext))
                queueObj = QueueObject{qCur, symbol, qSrc.transition(symbol)};
                if (!visited.contains(queueObj))
                    queue.push(queueObj);
                    visited.insert(queueObj);

    for ([qDest, qSrc]: copies)
        shallowCopy(qDest, qSrc);
```

Listing 4.9: Split one start for an underscore automaton

If S_0 cannot transition through 255, likewise Algorithm 3.13, a special state q_{esc} is created, and $\delta(S_0, 255) = q_{esc}$; the only purpose of q_{esc} is ensuring that the byte following 255 is treated as the raw byte value rather than a symbol. Thereafter, we link the starts through the function presented in Algorithm 3.10, ensuring that if $\delta(S_i, c) = q$ and $j < i$, then $\delta(S_j, c)$ does not lead to the default transition S_0.

For creating the fallback states, we combine Algorithm 3.11 and Algorithm 4.9; we traverse the automaton using the breadth-first search algorithm, and, at each step of the algorithm, the queue contains prefixes $\{\mathbf{p}_0, p_1, \ldots, p_{k-1}\}$, where $\mathbf{p}_0 = \{p_0^{(0)} | p_0^{(1)} | \ldots | p_0^{(k_0-1)}\}$. For each such prefix, the fallback state is defined as the state reached from S_0 without defaulting through the

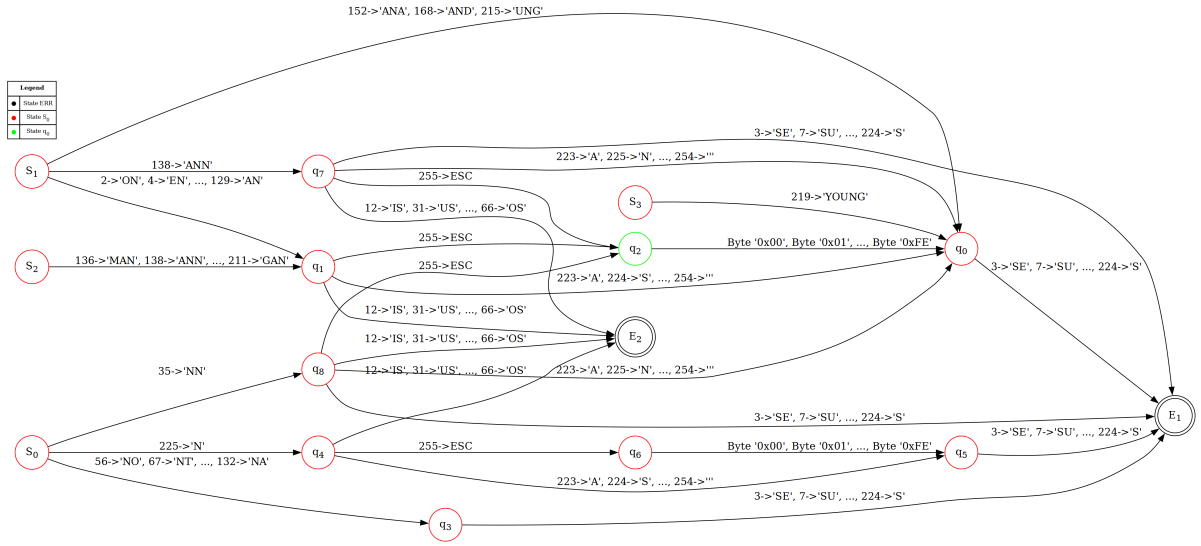


Figure 4.6: Middle automaton after start splitting for the pattern "%N_S%"

sequence $\{p_{k_1}, p_{k_1+1}, \ldots, p_{k-1}\}$, with k_1 minimal. Likewise in Algorithm 4.9, each parent state q has copies of it; each copy corresponds to a different trailing state $q_{trailing}$ reached through some symbol sequence, as the mapping from states to trailing states is one-to-many in this case.

The queue we use for the traversing algorithm contains the same struct used for start splitting `QueueObject`; here, the triplet consists of the previous state q_{prev}, the symbol c used for transitioning and the trailing state $q_{prev}^{(trail)}$ of q_{prev}, and we store the previous states because $\delta(q_{prev}, c)$ and $\delta(q_{prev}^{(trail)}, c)$ are susceptible to change from one step to another. To initialise the queue, the algorithm is slightly different: for each non-null start state S_i and each transition $\delta(S_i, c) = q$, if we have not traversed q yet, add the triplet $(S_i, c, \texttt{NULL})$ to the queue; we do not add q multiple times to the queue because the first symbol of the prefix is irrelevant for the trailing state. This functionality is implemented by a function with the signature `initQueue(qStarts)`, returning the initialised queue object.

Similarly to Algorithm 3.11, we store a data structure `trailingMappings` mapping each state to its unique trailing state; when we transition a trailing state through a symbol, using the induction argument previously introduced in Subsection 3.3.3, we advance the trailing state using Algorithm 3.12. Notice that, to preserve the behaviour of the escaped state $q_{esc} = \delta(S_0, 255)$ previously explained in Algorithm 3.11, we set its trailing state to `NULL`.

To keep track of the copies of states in the original automaton which we create, we require two of the data structures used in Algorithm 4.9: `parentMappings`, mapping each newly created state to its parent, and `splitMappings` (called `copyMappings` in the previous algorithm), which maps each pair (q_{parent}, q_{trail}) to a copy q_{copy} of q_{parent} whose trailing state is q_{trail}. If q_0 has the trailing state q_1 and q_1 has the trailing state q_2, we must copy the transitions of q_2 into q_1 before we copy the transitions of q_1 into q_0; for this, the first time we traverse a state q, we add it to a deque, called `topoSorted`, also present in Algorithm 3.11. As such, by breadth-first searching, we guarantee that all states q appear in `topoSorted` after its trailing state $q^{(trail)}$.

When traversing the queue, we transition both q_{prev} and $q_{prev}^{(trail)}$ through the current symbol to obtain q_{cur} and $q_{cur}^{(trail)}$, and we set q_{parent} to be the parent of q_{cur}. We have four cases:

I. it is the first time we have arrived in q_{cur} (i.e., `trailingMappings` does not have q_{cur} as a key); in this case, the trailing state of q_{cur} is set to $q_{cur}^{(trail)}$ and we add q_{cur} to `topoSorted`;

II. we have previously visited q_{cur} and the official trailing state of q_{cur} is $q_{cur}^{(trail)}$ (i.e., `trailingMappings` maps q_{cur} to $q_{cur}^{(trail)}$); we simply continue the loop;

III. we have previously visited q_{cur}, the official trailing state of q_{cur} is not $q_{cur}^{(trail)}$, but we have a copy q_{copy} of q_{parent} with this trailing state (i.e., `splitMappings` maps $(q_{parent}, q_{cur}^{(trail)})$ to q_{copy}); we redirect $\delta(q_{prev}, c) = q_{copy}$ and continue the loop;

IV. we have previously visited q_{cur}, the official trailing state of q_{cur} is not $q_{cur}^{(trail)}$, and we do not have a copy q_{copy} of q_{parent} with this trailing state (i.e., `splitMappings` does not have $(q_{parent}, q_{cur}^{(trail)})$ as a key); we create a new state q_{new}, copying the transitions of q_{cur}; q_{new} has as trailing state $q_{cur}^{(trail)}$, and we push it to `topoSorted`; finally, we set the parent of q_{new} to q_{parent} and we redirect $\delta(q_{prev}, c) = q_{new}$.

After handling all possible cases, we set $q_{cur} = \delta(q_{prev}, c)$ in case the transition changed. Finally, for all c such that q_{cur} transitions through c and $\delta(q_{cur}, c)$ is a state in the current automaton (i.e., not an end state), if we have never added the triplet $(q_{cur}, c, q_{cur}^{(trail)})$ to the queue, we add it to both the queue and the set of visited triplets. Upon termination of the loop, we copy the transitions of the trailing states in the order provided by `topoSorted`. We present the entire algorithm in Algorithm 4.10.

At each step, q_{prev} has the prefix $\{\mathbf{p}_0, p_1, p_2, \ldots, p_{k-1}\}$, where $\mathbf{p}_0 = \{p_0^{(0)} | p_0^{(1)} | \ldots | p_0^{(k_0-1)}\}$ is a vector of possible symbols, and $q_{prev}^{(trail)}$ has the prefix $\{p_{k_1}, p_{k_1+1}, \ldots, p_{k-1}\}$, where k_1 is minimal; we have previously shown inductively that $q_{prev}^{(trail)}$ is achievable by parsing $\{p_1, p_2, \ldots, p_{k-1}\}$ starting at S_0 and allowing fallback transitions, and the symbol c is chosen so that $\delta(q_{prev}, c)$ is a non-trivial transition. Since we redirect transitions of states during the automaton traversal, we seek whether or not the states become invalid (i.e., the prefix of either q_{prev} or $q_{prev}^{(trail)}$ changes).

The length of the prefix of a state is equivalent to the distance from a start state. Because the approach traverses the automaton using breadth-first search, at each point in the queue, we have at most two different prefix lengths (distances to a start). When at depth d in the automaton, the queue looks, in terms of prefixes, like

$$\Big[\Big(\Big\{\mathbf{p}_0^{(0)}, p_1^{(0)}, \ldots, p_{d-1}^{(0)}\Big\}, c^{(0)}, \Big\{p_{k_0}^{(0)}, p_{k_0+1}^{(0)}, \ldots, p_{d-1}^{(0)}\Big\}\Big), \Big(\Big\{\mathbf{p}_0^{(1)}, p_1^{(1)}, \ldots, p_{d-1}^{(1)}\Big\}, c^{(1)}, \Big\{p_{k_1}^{(1)}, p_{k_1+1}^{(1)}, \ldots, p_{d-1}^{(1)}\Big\}\Big), \ldots,$$
$$\Big(\Big\{\mathbf{p}_0^{(m-1)}, p_1^{(m-1)}, \ldots, p_{d-1}^{(m-1)}\Big\}, c^{(m-1)}, \Big\{p_{k_{m-1}}^{(m-1)}, p_{k_{m-1}+1}^{(m-1)}, \ldots, p_{d-1}^{(m-1)}\Big\}\Big), \Big(\Big\{\mathbf{p}_0^{(m)}, p_1^{(m)}, \ldots p_d^{(m)}\Big\}, c^{(m)}, \Big\{p_{k_m}^{(m)}, p_{k_m+1}^{(m)}, \ldots, p_d^{(m)}\Big\}\Big),$$
$$\ldots, \Big(\Big\{\mathbf{p}_0^{(n-1)}, p_1^{(n-1)}, \ldots, p_d^{(n-1)}\Big\}, c^{(n-1)}, \Big\{p_{k_{n-1}}^{(n-1)}, p_{k_{n-1}+1}^{(n-1)}, \ldots, p_d^{(n-1)}\Big\}\Big)\Big].$$

When traversing q_{prev} having the prefix $\left\{\mathbf{p}_0^{(0)}, p_1^{(0)} \ldots, p_{d-1}^{(0)}\right\}$, through symbol $c^{(0)}$, and $q_{prev}^{(trail)}$ having the prefix $\left\{p_{k_0}^{(0)}, p_{k_0+1}^{(0)}, \ldots, p_{d-1}^{(0)}\right\}$, the only transition we change is $\delta(q_{prev}, c^{(0)})$, so only the state with prefix $\left\{\mathbf{p}_0^{(0)}, p_1^{(0)} \ldots, p_{d-1}^{(0)}, c^{(0)}\right\}$ may be changed. Since this state has a prefix of length $d+1$ and the prefix of any trailing state currently in the queue is at most of

```
void createUnderscoreFallbackStates(qStarts)
    visited = std::unordered_set<QueueObject>{};
    queue = initQueue(qStarts);
    splitMappings =
        std::unordered_map<State*, std::unordered_map<const State*, State*>>{};
    trailingMappings = std::unordered_map<const State*, const State*>{};
    parentMappings = std::unordered_map<State*, State*>{};
    topoSorted = std::deque<State*>{};
    while (!queue.empty())
        [qPrev, symbol, qPrevTrailing] = queue.front();
        queue.pop();
        qCur = qPrev.transition(symbol);
        qTrailing = transitionTrailing(qPrevTrailing, symbol, trailingMappings);
        qParent = parentMappings.contains(qCur) ? parentMappings[qCur] : qCur;
        if (!trailingMappings.contains(qCur))
            // case (I)
            trailingMappings[qCur] = qTrailing;
            topoSorted.push_back(qCur);
        else if (qTrailing == trailingMappings[qCur])
            // case (II)
            continue;
        else if (splitMappings[qParent].contains(qTrailing))
            // case (III)
            qPrev.transitions[symbol] = splitMappings[qParent][qTrailing];
            continue;
        else
            // case (IV)
            qNew = createState();
            qNew.transitions = qCur.transitions;
            splitMappings[qParent][qTrailing] = qNew;
            trailingMappings[qNew] = qTrailing;
            topoSorted.push_back(qNew);
            parentMappings[qNew] = qParent;
            qPrev.transitions[symbol] = qNew;
        qCur = qPrev.transition(symbol);
        for ([symbol, qNext]: qCur.transitions)
            queueObj = QueueObject{qCur, symbol, qTrailing};
            if (!visited.contains(queueObj) and allStates.contains(qNext))
                queue.push(queueObj);
                visited.insert(queueObj);
    for (qCur: topoSorted)
        qCur.copyTransitions(trailingMappings[qCur]);
```

Listing 4.10: Create fallback states for an underscore automaton

length d ($k_i \geq 1, \forall i \in \overline{0 \dots n}$), the trailing states are valid throughout the algorithm. For the traversing states, having the state with the prefix $\left\{\mathbf{p}_0^{(0)}, p_1^{(0)} \dots, p_{d-1}^{(0)}, c^{(0)}\right\}$ already in the queue implies having previously visited the state with the prefix $\left\{\mathbf{p}_0^{(0)}, p_1^{(0)} \dots, p_{d-1}^{(0)}\right\}$; however, this also implies traversing $\left(q_{prev}^{(0)}, c^{(0)}, q_{trail}^{(0)}\right)$ twice, due to the symbol path uniquely identifying the trailing state, which is a contradiction. We conclude that the states in the queue are always

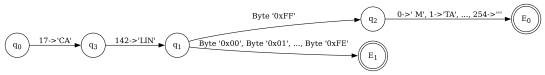
valid.

Having finished creating the fallback states for the automaton, we use `precomputeStarts` to pre-compute the start states, which are currently `NULL`. The final algorithm follows Algorithm 3.13 closely, implementing the presented changes, with signature `buildFirst(pattern, symTable, qStarts, qEnds, qStart, qUnderscores, reqExact);` we set `reqExact` to true only in the case `subpatterns.size()` is 1.

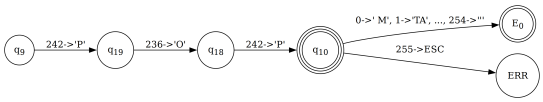
In the case that the first entry of `numUnderscores` is non-zero, by convention, there is no automaton preceding this automaton to build; as a consequence, `qStarts[i]` is not used later, for $i > 0$. To create the underscore states preceding the automaton, we implement the function `createFirstUnderscoreZone(numUnderscores, qEnds)`, with the parameter `reqExact` set to false. The key difference from `createUnderscoreZone` is how we handle transition to end states; we postpone the presentation of the algorithm to Chapter 5. Finally, we set `qStarts[0] = qUnderscores[0];` we showcase a comparison of this algorithm and Algorithm 4.4 in Figure 4.2. Having finished building individual sub-automata for sub-patterns, we now explain how to connect the sub-automata to simplify parsing.

5 Construction and Structuring of Automata

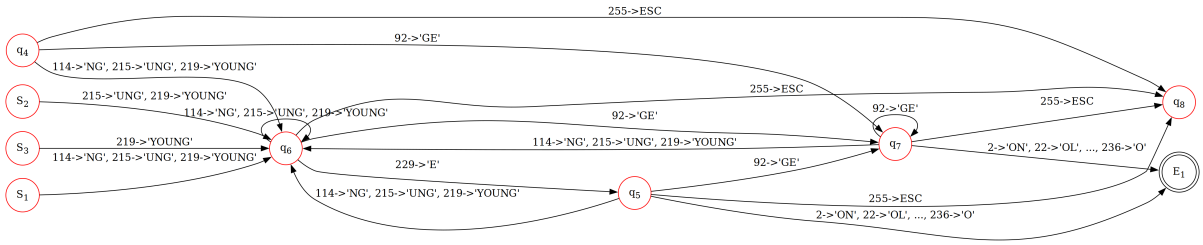
The previous chapters build the automata for individual sub-patterns; as such, we assume that we have built a vector of automata $\mathcal{F} = \{\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{n-1}, \mathcal{F}_n\}$ for a pattern containing multiple sub-patterns, where $\mathcal{F}_0$ and $\mathcal{F}_n$ represent the (potentially empty) automata for the start pattern, respectively for the end pattern. There are two possible directions for the automata; $\mathcal{F}_{fwd} = \{\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{n-1}\}$ represents the vector of automata going forward (for start or middle patterns), and $\mathcal{F}_{bwd}$ is the automaton going backwards (for the end pattern). For example, for the pattern "$CALIN\_\%GEO\%POP$", we present the three generated sub-automata below in Figure 5.1. We begin this chapter by explaining the algorithm that connects all the previously built sub-automata.



(a) The automaton $\mathcal{F}_0$ generated for start sub-pattern "$CALIN\_$"



(b) Tbe automaton $\mathcal{F}_2$ generated for end sub-pattern "POP"



(c) The automaton $\mathcal{F}_1$ generated for middle sub-pattern "GEO"

Figure 5.1: All sub-automata built for the pattern "$CALIN\_\%GEO\%POP$"

5.1 Connecting Sub-Automata

Having the vector of forward automata $\mathcal{F}_{fwd}$, our goal is to connect them in order to avoid splitting symbols when parsing. Since there is only one backward automaton, we do not need to connect it with any other automaton. As a convention, both $\mathcal{F}$, representing the automaton by connecting the sub-automata from $\mathcal{F}_{fwd}$, and $\mathcal{F}_{bwd}$ connect to the same set of end states $\{E_0, E_1, \ldots, E_8\}$.

For forward automata, we do not split a symbol c when parsing: if we used its first k bytes "$c_0 c_1 \ldots c_{k-1}$" for the automaton $\mathcal{F}_m$, with $m \neq n-1$, we should be able to use the remaining bytes "$c_k c_{k+1} \ldots c_{len(c)-1}$" for the following automaton $\mathcal{F}_{m+1}$, which is equivalent to starting, in

the sub-automaton $\mathcal{F}_{m+1}$, from a start state $S_j^{(m+1)}$ with $j \geq k$. By construction,

$$\texttt{precomputedStarts}^{(m+1)}\texttt{[i]} = \begin{cases} S_j^{(m+1)}, & j \geq i \text{ minimal}, \\ S_0^{(m+1)}, & \text{no such } j \text{ exists}, \end{cases}$$

and, by the algorithm linking the starts, if $S_k^{(m+1)} \neq S_0^{(m+1)}$ and there exists $j \geq k$ minimal such that $\delta(S_j^{(m+1)}, c) = q^{(m+1)}$, with $q^{(m+1)} \in \mathcal{F}_{m+1}$, we have defined $\delta(S_k^{(m+1)}, c) = q^{(m+1)}$. Therefore, when adding a transition to `qEnds[k]` from a state $q^{(m)} \in \mathcal{F}_m$ through c (i.e., using the first k bytes of the symbol c), we instead have $\delta(q^{(m)}, c) = q^{(m+1)}$, so the symbol c is not split. If either $S_k^{(m+1)} = S_0^{(m+1)}$ or $S_k^{(m+1)}$ does not transition through c, we cannot use the remaining bytes of the symbol c, so instead we transition to the start $S_0^{(m+1)}$ of the sub-automaton $\mathcal{F}_{m+1}$; we present the implementation below in Algorithm 5.1.

```
void addEndTransition(symbol, qEnds, endIdx)
    if (endIdx == 0 or qEnds[endIdx] == qEnds[0])
        addTransition(symbol, qEnds[0]);
    else
        // qEnds[endIdx] transitions, by default, to qEnds[0]
        addTransition(symbol, qEnds[endIdx].transition(symbol));
```

Listing 5.1: Adding an end transition

For the previous implementation to make sense, the sub-automaton having the start states in `qEnds` must already exist. As a consequence, given $\mathcal{F}_{fwd} = \{\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{n-1}\}$, we construct them in opposite order, i.e. $\mathcal{F}_{n-1}, \mathcal{F}_{n-2}, \ldots, \mathcal{F}_0$. By doing this, the connections between sub-automata are created on the fly, and none of this is done explicitly. To finalise the full forward automaton $\mathcal{F}$, we set the start state of $\mathcal{F}$ to be single start of the automaton $\mathcal{F}_0$ built for the start pattern, if not empty, and $S_0^{(1)}$ of $\mathcal{F}_1$ otherwise; we show the connected automaton for the pattern "*CALIN_%GEO%POP*" below in Figure 5.2.

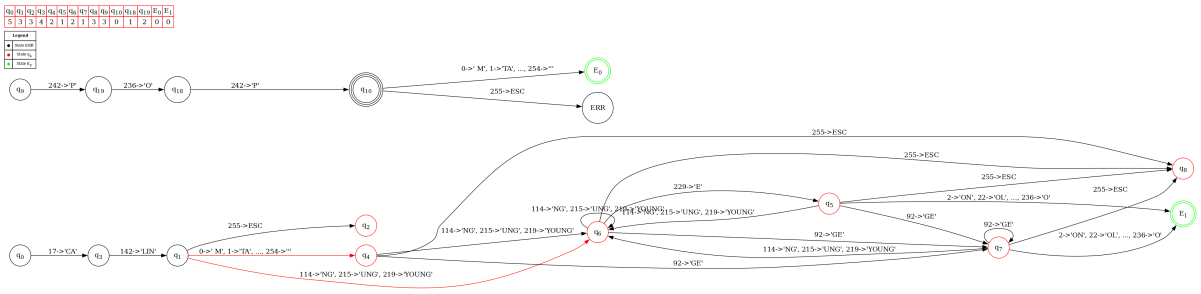


Figure 5.2: Connecting the automata for the pattern "*CALIN_%GEO%POP*"

For parsing, we define a struct `Index` containing a `uint8_t` for the position in the last symbol and a `uint64_t` for the position in the compressed string where parsing finishes. The parsing algorithm for the forward automaton is as follows: if there is no forward automaton, we set the symbol index to 0 and the string index to 0, and then return. Otherwise, while there are symbols left in the compressed string and the current state is not an accept state, advance the current state through the current symbol. When arriving in the error state, return

an empty struct; otherwise, increment the index in the compressed string. After finishing the loop, if the current state is an accept state, we return its current index as the position of the last symbol and decrement the index by one for the position in the compressed string, as we incremented it once more when we finished the loop. Otherwise, we return an empty struct.

For the backwards automaton, the algorithm is similar, with a few key differences: in the case there is no end automaton, we set the symbol index to 8 and the string index to the last index of a character in the compressed string (`string.size() - 1`). Otherwise, we use the same approach as before, but we decrement the index, and, after the loop, there are three different possibilities compared to the previous two:

I. `qCurrent` is an accept state; therefore, we return its accept index as the position in the symbol we start from, and the index in the compressed string is incremented twice, as we consumed one more symbol to transition from a pseudo-end to the accept state;
II. `qCurrent` is a pseudo-end; then, we return its pseudo-end index as the symbol index and the index in the compressed string is incremented once, as we consumed only one extra symbol;
III. otherwise, we do not have a match, and we return an empty struct.

We showcase a comparison of the two algorithms in Figure 5.3. To decide whether to accept or reject a compressed string, we initially parse the backwards automaton; if we fail to do so, we return false. Thereafter, we parse the forward automaton as well; if we fail, we return false. Finally, using the string and symbol indexes from the previous parsing algorithms (`symIdxBwd`, `strIdxBwd`) and (`symIdxFwd`, `strIdxFwd`), we accept if either `strIdxFwd` $<$ `strIdxBwd` (i.e., we end matching the forward automaton before the backwards automaton) or if `strIdxFwd` $=$ `strIdxBwd` and `symIdxFwd` $\leq$ `symIdxBwd` (i.e., we end in the same symbol in both cases, but we end earlier in the symbol in the forward automaton).

However, we observe that a sequence of symbols $\{c_0, c_1, \ldots, c_{k-1}\}$ does not match the pattern drawn in Algorithm 5.2 if $k < 7$, so we can immediately reject the sequence. This holds because the shortest path from the start of the forward automaton q_0 to any end state contains 5 bytes ($q_0 \xrightarrow{17} q_3 \xrightarrow{142} q_1 \xrightarrow{114} q_6 \xrightarrow{229} q_5 \xrightarrow{2} E_1$) and the shortest path from the start of the end automaton to an end state is 3 bytes ($q_9 \xrightarrow{242} q_{19} \xrightarrow{236} q_{18} \xrightarrow{242} q_{10}$; q_{10} is a pseudo-end state), and the two paths share at most one byte. As a consequence, we assign to each state the shortest distance to an end state; this approach is presented in the following section.

5.2 Level Assignment

Under normal circumstances, we require a reverse breadth-first search to obtain the shortest distance from each state to an end state, which we refer to as a level from now on. However, this approach is memory-intensive and computationally expensive, as it requires a vector of parents for each state; computing this data structure involves a complete traversal of the automaton.

Define $\mathcal{L} : \mathcal{Q} \to \mathbb{N}$, the level function mapping a state to its level; then, for an end state $E \in \mathcal{F}$, $\mathcal{L}(E) = 0$ by definition. For forward automata, we use the fact that all sub-automata are built from right (i.e., states with a lower level) to left (i.e., states with a higher level). As a consequence, at the point where $\delta(q, c) = q_{dest}$ is defined, the sub-automaton starting from q_{dest} has already been constructed, so the value of $\mathcal{L}(q_{dest})$ is known. In this case, if either

```
std::optional<Index> parseForward(
    qStart, string, qErr
)
    idx = 0;
    if (qStart == NULL)
        return {0, idx};
    qCurrent = qStart;
    while (idx < string.size() and
        !isAccept(qCurrent))
        c = string[idx];
        qCurrent = qCurrent.transition(c);
        if (qCurrent == qErr)
            return std::nullopt;
        ++idx;
    if (isAccept(qCurrent))
        return {
            getAcceptIdx(qCurrent),
            idx - 1
        };
    return std::nullopt;
```

Listing 5.2: Parsing a compressed string through the forward automaton

```
std::optional<Index> parseBackwards(
    qStart, string, qErr
)
    idx = string.size() - 1;
    if (qStart == NULL)
        return {8, idx};
    qCurrent = qStart;
    while (idx >= 0 and
        !isAccept(qCurrent))
        c = string[idx];
        qCurrent = qCurrent.transition(c);
        if (qCurrent == qErr)
            return std::nullopt;
        --idx;
    if (isAccept(qCurrent))
        return {
            getAcceptIdx(qCurrent),
            idx + 2
        };
    if (isPseudoEnd(qCurrent))
        return {
            getPseudoEndIdx(qCurrent),
            idx + 1
        };
    return std::nullopt;
```

Listing 5.3: Parsing a compressed string through the backwards automaton

Figure 5.3: A side-by-side comparison of the algorithms building underscore automata

$\mathcal{L}(q)$ has not been set or $\mathcal{L}(q_{dest}) + 1 < \mathcal{L}(q)$, we define $\mathcal{L}(q) = \mathcal{L}(q_{dest}) + 1$. Moreover, the implementation of the `deepCopy` function copies the levels of the sub-automaton states, rather than recomputing them. However, the states having smaller levels are not necessarily built first in the case of end automata; initially, we assign to each pseudo-end state $\mathcal{L}(q) = 0$. For end patterns without underscore wildcards, Algorithm 3.1 builds the first sub-automaton. Therefore, initially, the only state in the automaton is `qStart`, and we replace Lines 4-8 in Algorithm 3.1 with

```
qCurrent = qEnds[0];
for (i = 0; i < compressed.size() - 1; ++i)
    qNext = createState();
    qNext.addTransition(compressed[i], qCurrent);
    qCurrent = qNext;
qStart.addTransition(compressed[compressed.size() - 1], qCurrent);
```

The algorithms are equivalent: instead of starting to add transitions from the start state, we start by adding transitions to the pseudo-end state. No branching is required here, as this is the first end sub-automaton built. However, for $k \neq 0$, part of the initial path might already exist; as a consequence, we find the largest i_k such that the start state has already created

the transitions for the sequence $\{c_{j_k-1}^{(k)}, c_{j_k-2}^{(k)}, \ldots, c_{i_k}^{(k)}\}$, and define q_{end} as the state reached through this sequence; if $i_k = 0$, we do not create any new states, and we connect q_{end} to the pseudo-ends through all possible symbols c. In the case $i_k \neq 0$, we create a new state q_{cur}, which we connect to the corresponding pseudo-end for all possible symbols c (and, as a consequence, $\mathcal{L}(q_{cur}) = 1$), and then, for all $j < i_k - 1$, we create a new state q_{new}; afterwards, we define $\delta(q_{new}, c_j^{(k)}) = q_{cur}$ and set $q_{cur} = q_{new}$. Finally, we define $\delta(q_{end}, c_{i_k-1}) = q_{cur}$. We replace the lines 8-15 from Algorithm 3.2 with

```
idx = compressed.size() - 1;
path = std::vector<State*>{};
qEnd = qStart;
path.push_back(qEnd);
while (idx >= 0 and qEnd.canTransition(compressed[idx]))
    qEnd = qEnd.transition(compressed[idx]);
    path.push_back(qEnd);
    --idx;
if (idx == -1)
    for (symIdx: symbols)
        endIdx = symTable.symbols[symIdx].size() - k;
        qEnd.addTransition(symIdx, qEnds[endIdx]);
else
    qCurrent = createState();
    for (symIdx: symbols)
        endIdx = symTable.symbols[symIdx].size() - k;
        qCurrent.addTransition(symIdx, qEnds[endIdx]);
    for (i = 0; i < idx; ++i)
        qNext = createState();
        qNext.addTransition(compressed[i], qCurrent);
        qCurrent = qNext;
    qEnd.addTransition(compressed[idx], qCurrent);
pathIdx = path.size() - 2;
while (pathIdx >= 0 and path[pathIdx].level > path[pathIdx + 1].level + 1)
    path[pathIdx].level = path[pathIdx + 1].level + 1;
    --pathIdx;
```

However, knowing that some states in our current path have already been created, if defining $\delta(q_{end}, c_{i_k-1}) = q_{cur}$ modifies $\mathcal{L}(q_{end})$, we propagate this change within the path through which we reach q_{end}, defined as $\{S, q_{j_k-1}, q_{j_k-2}, \ldots, q_{i_k} = q_{end}\}$; if $\mathcal{L}(q_{i_k+1}) \leq \mathcal{L}(q_{i_k}) + 1$, we stop; otherwise, we set $\mathcal{L}(q_{i_k+1}) = \mathcal{L}(q_{i_k}) + 1$ and we move the index forward. Similarly, if $\mathcal{L}(q_{i_k+2}) \leq \mathcal{L}(q_{i_k+1}) + 1$, we stop; otherwise, we set $\mathcal{L}(q_{i_k+1}) = \mathcal{L}(q_{i_k}) + 1$, and we repeat this until we either stop changing the level or we do not have states left in the path.

For end patterns containing underscores, the algorithm is significantly simpler, as we construct it as a forward automaton and then invert it, which involves running a reverse breadth-first search algorithm. As a consequence, we also store the breadth in the queue; initially, the states transitioned into by the start states have a level of 1. We change Lines 7-11 of Algorithm 4.7 to

```
enqueue = [&newTransitions, &queue] (qNext, level) {
    if (!newTransitions.contains(qNext))
        qNext.level = level;
        newTransitions[qNext] = std::unordered_map<uint8_t, State*>{};
        queue.emplace(qNext, level);
}
```

and Line 23 to `enqueue(qNext, currentLevel + 1)`; we guarantee by breadth-first search that we assign to each state its corresponding level, given that we traverse each state only once. Having the value of $\mathcal{L}(q)$ for all states q, we finally optimise the parsing function by modifying Line 8 of Algorithm 5.3 to `idx + 1 >= qCurrent.level`; then, we pass `endStrIdx` to `parseForward`, and we replace Line 8 of Algorithm 5.2 to `idx + qCurrent.level <= endStrIdx`. By doing so, we ensure that both algorithms terminate as early as possible. Having presented the algorithm, we then proceed to compare its performance with that of the state-of-the-art approaches.

6 Benchmark

In this chapter, we present the results obtained when evaluating the performance of the presented algorithm and various optimisations. We begin by introducing the hardware specifications of the server on which we run our experiments in Section 6.1. Afterwards, we present the datasets we use, along with their different scale factors and the LIKE patterns used in Section 6.2, and we explain the various approaches we benchmark in Section 6.3; finally, we present the results of the benchmarking process in Section 6.4.

6.1 Hardware Specification

We run the experiments on a single thread of a dual-socket machine with Intel(R) Xeon(R) CPU E5-2660 v2 processors at 2.20GHz; the machine has a total of 251 GiB of DDR3-DRAM running at 1866 MHz, and the operating system is a 64-bit Ubuntu 24.04.2 LTS with a Linux 6.8.0-64-generic kernel. The processor implements the SSE4.2 SIMD instruction set, enabling accelerated vector operations. We implement the matching algorithm on a fork of the code presented in [BNL20], where we upgraded the C++ version to C++ 20.

6.2 Datasets

To evaluate the performance of the code, we generated LIKE patterns on three different datasets: TPC-H, StackOverflow[1] and IMDB[2,3]; while the data distribution of TPC-H is unnatural, as each word in the queried columns are generated from an uniform distribution, the distributions of the other two datasets more accurately reflect real-world behaviour, as they are derived from actual data.

For TPC-H, we generate datasets for the scale factors 1, 10 and 100. To emulate this behaviour on the other datasets, we initially extract the queried columns in individual files named *table*_uncompressed, where *table* is the name of the table we extract the column from; the format of the file is as follows: the first 8 bytes of the file represent the number of strings n, and the next 8 bytes represent the total sum of lengths L of all the strings $s_0, \ldots, s_{n-1}$. Afterwards, the next L bytes represent the buffer, containing all the strings in the dataset concatenated; finally, the last $8 \times n$ bytes match each string to the index in the buffer where it starts from. To generate a sub-sample for the scale factor sf, we sample n Bernoulli random variables $X_0, X_1, \ldots, X_{n-1} \sim \text{Bernoulli}(p)$, with parameter $p = sf * 2^{30}/L$; we select the string s_i for the result file if and only if $X_i = 1$, and the format of the end file is the same as the one previously mentioned.

[1] downloaded using a script provided by [Sch+25]

[2] `plot` and `quotes` downloaded from `https://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/)`

[3] `films` and `actors` downloaded from `https://datasets.imdbws.com/`

Using this algorithm, we generate the scale factors $0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50$ for the Stack Overflow dataset and $0.001, 0.005, 0.01, 0.05, 0.1, 0.2$ for the IMDB dataset. After generating the datasets for different scale factors, we run the FSST algorithm, which creates and saves an FSST symbol table on the disk. We then compress all the data using this symbol table; the compressed file retains the same format previously presented. For the three datasets, we then generate LIKE patterns with no underscores for the compressed columns, such that we have all three primary pattern types:

I. containing a start pattern (and potentially multiple middle patterns);
II. containing an end pattern (and potentially multiple middle patterns);
III. containing only middle patterns.

We explain the reasoning behind this pattern classification later. After generating the patterns, we then mask one arbitrary character to create sets of patterns with underscores. Finally, we mask ≥ 2 arbitrary characters for the sets of patterns with at least 2 underscores. We split the patterns into groups based on the number of underscores, as the number of states grows exponentially with this number: even short, simple patterns generate large automata. We present examples of patterns generated for the previously mentioned datasets in Table 6.1.

TPC-H			Stack Overflow			IMDB	
table	column	pattern	table	column	pattern	table	pattern
part	p_type	"MEDIUM POLISHED%"	Comments	Text	"Why do%you%?"	name	"Marlon %"
supplier	s_comment	"%Customer%Complaints"	PostHistory	Comment	"%sometimes%know."	title	"% Scene"
orders	o_comment	"%special%requests%"	Posts	Body	"%why%happening%"	quotes	"%Harvey%It's just%"
part	p_type	"MED_UM POLISHED%"	Comments	Text	"Why_do%you%?"	name	"_arlon %"
supplier	s_comment	"%Customer%Compl_ints"	PostHistory	Comment	"%sometimes%know_"	title	"% Sc_ne"
orders	o_comment	"%speci_l%requests%"	Posts	Body	"%why%happenin_%"	quotes	"%_arvey%It's just%"
part	p_type	"_EDIUM POLI__ED%"	Comments	Text	"Why do%y__%?"	name	"_a_lon %"
supplier	s_comment	"%Cu_tomer%Complai_ts"	PostHistory	Comment	"%some_ime_%know."	title	"% ___ne"
orders	o_comment	"%speci_l%requ_sts%"	Posts	Body	"%wh_%hap_enin_%"	quotes	"%_arvey%It_s just%"

Table 6.1: A subset of benchmarked patterns, grouped by dataset, table and column queried

6.3 Approaches

Our competitor, the Hybrid String Search Decoded algorithm (HSSDecoded), involves first decoding the compressed string using the symbol table and then running a Hybrid String Search (HSS) on the decompressed string. For each sub-pattern, HSS initially finds instances of the longest sub-subpattern not containing any underscore wildcards (called `needle`) within the text (called `haystack`). This is achieved by two different algorithms. The first such algorithm is SIMDSearch [SPR16], suitable for `needles` of length ≤ 16. It loads `needle` into a SIMD register and utilises SSE4.2 instructions to find its first instance within `haystack`. The other possible algorithm is the Two-Way Search algorithm [CP91], combining both the KMP and BM algorithms. Given the drop in performance of the SIMDSearch algorithm for `needles` of length > 12 [Rie+23], we adjust the algorithm to use SIMDSearch only if the `needle` length is ≤ 12. The implementation of the full algorithm is directly taken from the Umbra database system [NF20].

To avoid decoding, we present three different approaches. In Subsection 6.3.1, we explain the interpreted approach, closest to what we have done up to this point. Afterwards, we compile the automaton into C++ code in Subsection 6.3.2. Finally, to reduce the compilation

cost, we present an approach that generates and compiles LLVM-IR in Subsection 6.3.3.

6.3.1 Interpreted Approach

For the interpreted approach, we store all previously built automata, ensuring that states remain valid memory addresses throughout the execution of the program; in the implementation of a finite automaton, all states reside in deque objects for the same reason as above; in this case, we use the previously presented `parseForward` and `parseBackwards` to check for matches.

The primary flaw of this implementation is that the accessed data is scattered across the heap. A `deque` is composed of a linked list of multiple memory blocks; therefore, the implementation requires significantly more random heap accesses, resulting in poor cache locality. Moreover, each state stores a `std::unordered_map` object allocated on the heap containing its transitions. States having a large number of transitions have a large number of collisions in the hash table of `std::unordered_map`, as the key is an unsigned integer on 8 bits, leading to slow look-up.

Indeed, using `perf` with sampling every 1000 cycles for the in-memory algorithm, for some arbitrary patterns on the TPC-H dataset, we observe that roughly 55% of the cycles are spent in the `find` function of the hash table implementation in C++. Furthermore, this approach achieves approximately 0.64 instructions per cycle, representing a low level of efficiency. To avoid pointer chasing, we implement automaton-specific C++ code in the following subsection, preserving the same parsing functionality previously presented.

6.3.2 C++ Compiled Code

Initially, if we have a forward automaton, for all states in the connected forward automaton, we create an enum class `FwdState` mapping each state to a unique enum object by utilising the unique memory address of each state.

For parsing, the function has the signature `parseFwd` and, apart from the compressed string, takes as a parameter also a pointer to the individual attributes of an `Index` object, which we modify in case of successful parsing. The boolean value returned determines whether the string was parsed successfully. Initially, we set the working state as the start state (i.e., `auto q = FwdState::q109859499273072;`) and we set the working level to the level of the start state (i.e., `uint64_t level = 2;`). While there are `level` bytes in the string, the state q and the current byte c in the compressed string determine our action:

I. $q = E_i$ is an end state; we decrement the value of the index in the string, we set the index in the symbol to i, and we return true;
II. $q = \varepsilon$ is the error state, then we return false;
III. q is not an end state and $\delta(q, c) = q'$; we have a switch case on the value of c which sets $q = q'$ and the working level to the level of q'; the default case does the same for the default transition. Afterwards, we increment the string index and continue the loop.

After the loop finishes, if q is not an end state, we return false; otherwise, we proceed as in (I). We present a small code snippet we generate to match the automaton presented earlier in Figure 3.2, but with the unreachable end states removed for simplicity, in Figure 6.1. Similarly, in the same file, if we have a backwards automaton, we generate the function `parseBwd(const uint8_t* string, int64_t len, int64_t* strIdx, uint8_t* symIdx)`. The differences between `parseFwd` and `parseBwd` are the `while` condition and the different way to approach

states, knowing there are both end states and pseudo-end states in the automaton: whenever we arrive in an end state, increment the string index by 2, set the symbol index as the index of the current end state, and return true. In comparison, we treat a pseudo-end state like a normal state within the while loop. However, in the end, if we run out of bytes and we are in a pseudo-end state, we increment the string index by 1, set the symbol index as the index of the pseudo-end state and return true; otherwise, we return true if and only if we end up in an accept state.

Finally, we distinguish three possible cases for the main function `parse(const uint8_t* string, int64_t len)`, exposed to external components via the `extern` keyword:

I. there is no backwards automata. We define `int64_t strIdx = 0` and `uint8_t symIdx`, and return `parseFwd(string, len, &strIdx, &symIdx)`. In this case, the code for the `parseBwd` function is not generated;

II. there is no forward automata. We define `int64_t strIdx = len - 1` and `uint8_t symIdx`, and return `parseBwd(string, len, &strIdx, &symIdx)`. In this case, the code for the `parseFwd` function is not generated;

III. there is both a forward automaton and a backwards automaton, so we must do a two-way search. Initially, if the incremented length of the string is less than the sum of the levels of the two start states, we instantly reject the string. Otherwise, we run the string through both previously constructed functions, and if either fails, we return false. Finally, the criteria for accepting in case both return true are the same as before. We present an example of the implementation below.

```
1  extern "C" bool parse(const uint8_t* string, int64_t len)
2      if (len < 5)
3          return false;
4      int64_t endStrIdx = len - 1;
5      uint8_t endSymIdx;
6      bool canParse = parseBwdState(string, len, &endStrIdx, &endSymIdx);
7      if (!canParse)
8          return false;
9      uint8_t startSymIdx;
10     int64_t startStrIdx = 0;
11     canParse = parseFwdState(string, endStrIdx + 1, &startStrIdx, &startSymIdx);
12     return canParse and (startStrIdx < endStrIdx or startSymIdx <= endSymIdx);
```

Finally, we compile the entire code into a shared library and use the `parse` function within our code. However, the generated C++ code can be thousands of lines long, and as a consequence, its compilation is too slow to be practical. The `clang` compiler we use generates LLVM-IR code; if we generate LLVM-IR directly and then compile it, we significantly reduce the compilation time. As such, the following subsection focuses on LLVM code generation.

6.3.3 LLVM Compiled Approach

Similar to the C++ implementation, for readability purposes, we emulate an enum class by assigning each state in the forward automaton to a unique integer, which is later constant-folded. The label of this integer is uniquely identified by the memory address of the state in the deque object. For the same automaton we compiled in the previous subsection, we define the following internal constants:

```
@fwd.q101495874758560 = internal constant i32 0
```

```
enum class FwdState {
  q101495874758560, // E0
  q101495874758640, // E1
  q101495874758800, // E3
  q101495874759296, // ε
  q101495874760208, // S
  q101495874759712, // q0
  q101495874759792, // q1
}
bool parseFwd(const uint8_t* string,
           int64_t len, int64_t* strIdx,
           uint8_t* symIdx) {
  auto q = FwdState::q101495874760208;
  uint64_t level = 1;
  while (*strIdx + level <= len) {
    switch(q) {
      case FwdState::q101495874758560: // E0
        *symIdx = 0;
        *strIdx -= 1;
        return true;
      case FwdState::q101495874758640: // E1
        *symIdx = 1;
        *strIdx -= 1;
        return true;
      case FwdState::q101495874758800: // E3
        *symIdx = 3;
        *strIdx -= 1;
        return true;
      case FwdState::q101495874759296: // ε
        return false;
      case FwdState::q101495874760208: // S
        switch (compressed[*strIdx]) {
          case 1: // δ(S,1) = E3
            q = FwdState::q101495874758800;
            level = 0;
            break;
          case 0: // δ(S,0) = q1
            q = FwdState::q101495874759792;
            level = 1;
            break;
          default: // default(S) = ε
            q = FwdState::q101495874759296;
            break;
        }
        break;
      case FwdState::q101495874759712: // q0
        switch(compressed[*strIdx]) {
          case 67: // δ(q0,0) = E0
            q = FwdState::q101495874758560;
            level = 0;
            break;
          default: // default(q0) = ε
            q = FwdState::q101495874759296;
            break;
        }
        break;

      case FwdState::q101495874759792: // q1
        switch(compressed[*strIdx]) {
          case 4: // δ(q1,4) = E3
            q = FwdState::q101495874758640;
            level = 0;
            break;
          case 3: // δ(q1,3) = E3
            q = FwdState::q101495874758640;
            level = 0;
            break;
          case 255: // δ(q1,255) = q0
            q = FwdState::q101495874759712;
            level = 1;
            break;
          default: // default(q1) = ε
            q = FwdState::q101495874759296;
            break;
        }
        break;
    }
    *strIdx += 1;
  }

  switch(q) {
    case FwdState::q101495874758560:
      *symIdx = 0;
      *strIdx -= 1;
      return true;
    case FwdState::q101495874758640:
      *symIdx = 1;
      *strIdx -= 1;
      return true;
    case FwdState::q101495874758800:
      *symIdx = 3;
      *strIdx -= 1;
      return true;
    default:
      return false;
    }
}
```

Figure 6.1: C++ generated code for the automaton in Figure 3.2

```
@fwd.q101495874758640 = internal constant i32 1
@fwd.q101495874758800 = internal constant i32 2
@fwd.q101495874759296 = internal constant i32 3
@fwd.q101495874760208 = internal constant i32 4
@fwd.q101495874759712 = internal constant i32 5
@fwd.q101495874759792 = internal constant i32 6
```

Afterwards, if there is at least one forward automaton, we generate the `parseFwd` function, which has the same functionality as the one presented in the previous subsection. We allocate memory for the current level and current working state with the `alloca` keyword, and we jump into the while loop. In the body of the loop, we load the actual integer value of the state in `fwd.qVal` (q) and the current working symbol in `fwd.elemVal` (c) using `load` function calls, and we create a switch statement based on the current value in `fwd.qVal`. Each automaton state has a unique switch label to jump to, of the form "fwd.q101495874758640.switch". As in Subsection 6.3.2, depending on the current state q, we proceed as follows:

I. $q = E_i$ is an end state; then, we set the index within the symbol to i, decrement the index within the string, and return true. For example, for E_1, the generated code is

```
fwd.q101495874758640.switch: ; preds = %fwd.loop.body
    store i8 3, ptr %fwd.symIdx, align 1
    %fwd.newVal5 = sub i32 %fwd.strIdxVal2, i8 1
    store i32 %fwd.newVal5, ptr %fwd.strIdx, align 4
    ret i1 true
```

II. $q = \varepsilon$ is the error state, then we simply return false;

```
fwd.q101495874759296.switch: ; preds = %fwd.loop.body
    ret i1 false
```

III. q is not an end state and $\delta(q, c) = q'$. The transition into q' results in the same behaviour irrespective of the particular state q from which it is taken: we set the current level to the level of q', and we set the current working state to q'. Thereafter, we jump at the end of the switch. As such, to reduce the number of total labels in the intermediate representation, for each state we generate a label of the form "transition.fwd.q101495874758640", and a switch statement maps each symbol to which such label to jump to; the default jump is just the label of the default transition of q.

```
fwd.q101495874759792.switch: ; preds = %fwd.loop.body
    switch i8 %fwd.elemVal, label %transition.fwd.q101495874759296 [
        i8 4, label %transition.fwd.q101495874758640
        i8 3, label %transition.fwd.q101495874758640
        i8 -1, label %transition.fwd.q101495874759712
    ]
...
transition.fwd.q101495874758640: ; preds = %fwd.q101495874759792.switch
    store i64 0, ptr %fwd.level, align 4
    %fwd.q101495874758640.value = load i32, ptr @fwd.q101495874758640, align 4
    store i32 %fwd.q101495874758640.value, ptr %fwd.q, align 4
    br label %fwd.merge

transition.fwd.q101495874759712: ; preds = %fwd.q101495874759792.switch
    store i64 1, ptr %fwd.level, align 4
    %fwd.q101495874759712.value = load i32, ptr @fwd.q101495874759712, align 4
    store i32 %fwd.q101495874759712.value, ptr %fwd.q, align 4
    br label %fwd.merge
```

Similarly, if there is a backwards automaton, we generate the `parseBwd` function, and, finally, the `parse` function combining all the functions previously generated; the signature of the parse function is the same as the one previously presented `bool parse(const uint8_t* string, int64_t len)`. After we have finished generating LLVM-IR code, we optimise the current module for the current machine with the most aggressive level of optimisations and aggressive inlining, prioritising speed over size. Having presented all the approaches, we now compare their runtimes in the next section.

6.4 Results

For benchmarking, we compute eight different metrics for each different run setup: the average, median, min, max speed-up factor, along with the 25^{th} and 5^{th} quantiles of this speed-up factor, compared to the previously explained HSSDecoded approach. Apart from statistics on the speed-up factors, in each case, we compute the median number of strings we parse in the time it takes to create the automaton (for the interpreted, C++ and LLVM approaches) and the median number of strings parsed in the time to compile the automaton (for the C++ and LLVM approaches). Given that we create the automaton thrice, we assign the actual automaton creation time to the maximum time out of the three.

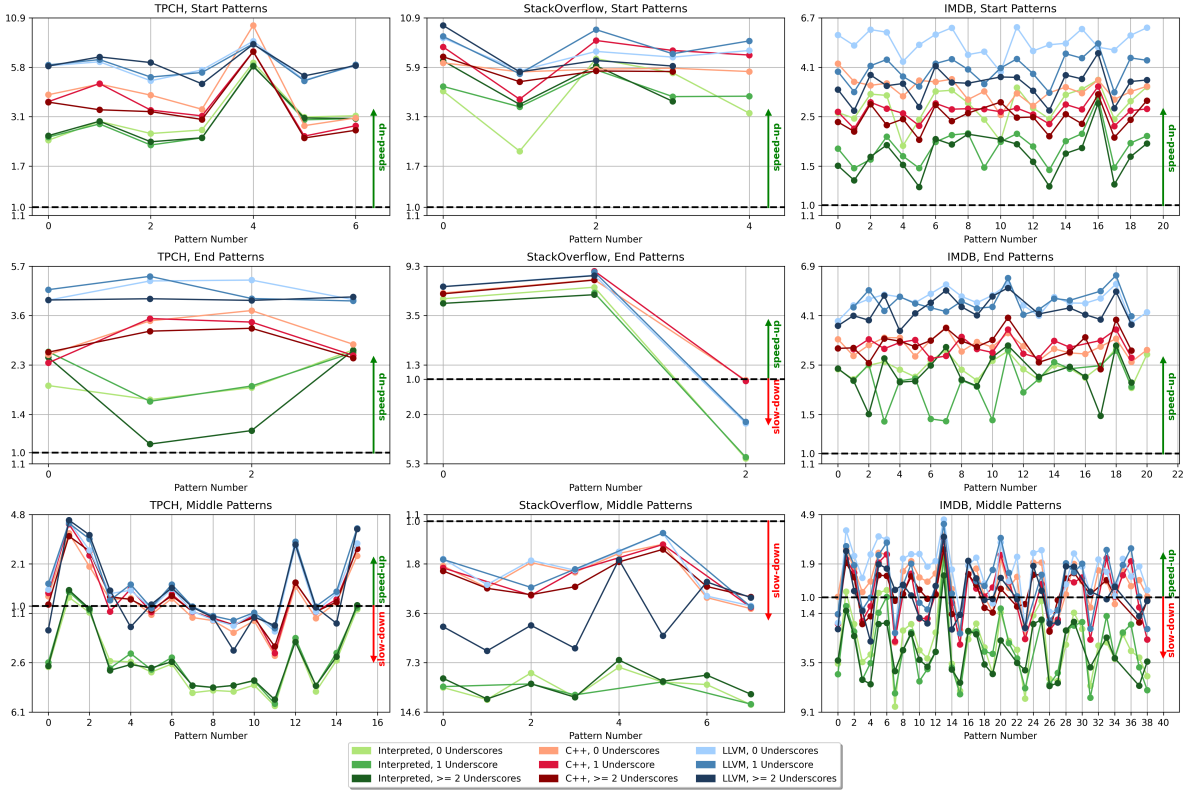


Figure 6.2: The average speed-up/slow-down, grouped by dataset and pattern type

To interpret the results, we present nine distinct plots in Figure 6.2, which separate the results by dataset (TPC-H, StackOverflow, or IMDB) and pattern class (Start, End, or Middle). On each plot, the x-axis positions are determined by the original pattern (before masking).

The measured performance metric is the speed-up factor (or slow-down factor), and the colour code of the visualisation depends on execution approach (Interpreted for green hues, C++ for red hues and LLVM for blue hues) and the number of underscores in the final masked pattern (the deeper the colour, the more underscores). We notice that the higher the number of underscores, the slower the algorithm.

The best-performing algorithm among the three presented is the approach that generates LLVM code, and all approaches are significantly faster than HSSDecoded in the case of start and end patterns. Only one end pattern is slower than the HSSDecoded approach, which happens because the end pattern is not selective at all (only 50% of the strings are rejected by the end pattern; as such, the run-time is dominated by the middle match).

For middle matches, despite parsing symbol by symbol (empirically, ≤ 3.3 bytes at a time), our compiled approaches are competitive with HSSDecoded (8 bytes at a time, when SSE 4.2 is enabled) on the TPC-H and IMDB datasets, but on StackOverflow, the presented approaches are around 2 times slower on average. This discrepancy in runtimes is caused by the fact that strings in StackOverflow are much longer than those in the other datasets; below, we present the average string lengths for all the datasets used. The scale factor is not important for this statistic, as the true distribution of the data is the same regardless of the scale factor.

TPC-H			Stack Overflow			IMDB	
table	column	avg len	table	column	avg len	table	avg len
orders	o_comment	48.5	Comments	Text	158.6	actors	13.6
part	p_name	32.7	PostHistory	Comment	765.6	films	20
part	p_type	20.6	Posts	Body	1158.2	plot	67.8
supplier	s_comment	62.5				quotes	48.6

Table 6.2: Average lengths of strings from the benchmarked datasets

We define a transition $\delta(S, c) = q$ to be redundant if S is the start state of a middle automaton and q is the default transition of S, $\delta_{def}(S) = S$; intuitively, this is redundant because we do not move the state of the automaton forward, and we still have to match the entire current middle pattern after doing the transition. Moreover, we define a sequence of traversed states to be redundant if all the transitions are redundant; such sequences are of the form $S \to S \to S \to \cdots \to S$, where S is the start state of a middle automaton. We aim to estimate the average length of redundant sequences when matching strings in the StackOverflow dataset.

We pick arbitrary middle patterns out of those benchmarked; on average, 95.5% of the taken transitions are redundant, the smallest redundancy rate exhibited by a pattern is 91.4%, and the median redundancy rate is 96.3%. Furthermore, whenever we visit the start state of a middle automaton, we count the length of the maximal redundant sequence starting at that point; for example, if we jump away from $S \to q \neq S$, the length is 0, if the sequence is $S \to S \to q \neq S$, the length is 1, and so on; we observe that, on average, when arriving in a start state of a middle automaton, the next 20 bytes are not used for transitioning; we present the full statistics in Table 6.3. Finally, when disabling SSE4.2, we are competitive with the original approach, leading us to believe that the high number of redundant transitions is the memory bottleneck of our algorithm; we showcase the complete numbers and comparisons of

pattern	redundant sequence length average	median	25^{th} quantile	# start transitions
table Posts, column Body				
"%why%happening%"	64.6	28	8	3, 5
"%why%happenin_%"	64.6	28	8	3, 5
"%cannot%fathom%C__%"	69	33	10	7, 4, 2
table PostHistory, column Comment				
"%SELECT *%LIMIT 1%"	80.8	24	5	2, 2
"%define%then%double%"	47.7	22	6	8, 17, 7
"%SELECT *%LIMIT _%"	80.9	25	5	2, 2
"%define%then%dou_le%"	47.7	22	6	8, 17, 7
table Comments, column Text				
"%why%C++%"	42	30	12	4, 2
"%how is%this%"	20.9	14	5	7, 17
"%_o we%actually%"	19	13	5	10, 12
"%_o we%a_tually%"	19	13	5	10, 22
"%w_y%C_+%"	27.5	18	7	7, 2
"%ho_ _s%this%"	20.8	14	5	7, 17

Table 6.3: Parsing statistics for different patterns on the StackOverflow dataset (SF=1)

different approaches, grouped by datasets, number of underscores, and pattern types, in Table 6.4. We observe the extreme compilation cost of C++. While the compilation cost of LLVM is equivalent to running the function on N tuples, the compilation cost of C++ is equivalent to running it on around $8 - 10 \times N$ tuples; however, there are ways to cover this compilation cost such that it is hardly noticeable when querying [KLN21; Gru+23].

			average speed-up factor			median speed-up factor			min speed-up factor			max speed-up factor		
Type	# '_'	Dataset	Interp.	C++	LLVM	Interp.	C++	LLVM	Interp.	C++	LLVM	Interp.	C++	LLVM
Start	0	TPCH	328%	460%	**598%**	291%	381%	**576%**	208%	207%	**455%**	624%	**1019%**	820%
		Stack	438%	576%	**706%**	395%	603%	**691%**	177%	307%	**545%**	**1047%**	790%	971%
		IMDB	279%	326%	**527%**	285%	334%	**520%**	176%	138%	**375%**	390%	546%	**730%**
	1	TPCH	306%	395%	**595%**	274%	368%	**592%**	206%	170%	**460%**	601%	762%	**792%**
		Stack	468%	692%	**772%**	414%	**760%**	740%	312%	235%	**526%**	1094%	1031%	**1510%**
		IMDB	185%	261%	**401%**	187%	264%	**405%**	137%	103%	**279%**	327%	401%	**549%**
	≥ 2	TPCH	309%	370%	**603%**	278%	367%	**607%**	207%	160%	**444%**	600%	755%	**787%**
		Stack	476%	545%	**693%**	446%	556%	**623%**	309%	284%	**487%**	1050%	809%	**1598%**
		IMDB	170%	244%	**342%**	167%	239%	**344%**	109%	94%	**243%**	308%	414%	**492%**
End	0	TPCH	199%	312%	**460%**	181%	307%	**462%**	137%	173%	**380%**	264%	468%	**533%**
		Stack	388%	506%	**545%**	391%	589%	**607%**	17%	74%	38%	**1046%**	886%	910%
		IMDB	242%	301%	**481%**	239%	324%	**477%**	176%	136%	**363%**	324%	460%	**738%**
	1	TPCH	222%	281%	**447%**	251%	238%	**435%**	153%	165%	**358%**	263%	468%	**551%**
		Stack	342%	532%	**542%**	354%	472%	**563%**	17%	69%	39%	1065%	**1105%**	1070%
		IMDB	211%	301%	**472%**	215%	312%	**461%**	129%	143%	**371%**	318%	491%	**704%**
	≥ 2	TPCH	198%	279%	**422%**	249%	253%	**419%**	104%	169%	**360%**	266%	449%	**514%**
		Stack	419%	557%	**619%**	364%	556%	**582%**	309%	284%	**487%**	**939%**	866%	899%
		IMDB	234%	311%	**439%**	224%	324%	**425%**	141%	114%	**342%**	327%	521%	**710%**
Middle	0	TPCH	45%	124%	**155%**	36%	93%	103%	18%	27%	**61%**	137%	**502%**	496%
		Stack	10%	50%	**53%**	10%	**52%**	51%	7%	23%	**24%**	14%	93%	95%
		Stack no SSE	20%	99%	106%	20%	90%	96%	17%	53%	**55%**	28%	166%	**180%**
		IMDB	51%	143%	**184%**	39%	130%	**188%**	11%	22%	**41%**	312%	455%	**520%**
	1	TPCH	51%	145%	**182%**	39%	99%	116%	18%	26%	**65%**	149%	**533%**	477%
		Stack	10%	48%	**50%**	9%	**44%**	42%	7%	**22%**	18%	14%	95%	91%
		Stack no SSE	20%	96%	100%	19%	92%	81%	16%	**51%**	32%	28%	165%	**177%**
		IMDB	47%	125%	**156%**	38%	116%	**138%**	13%	23%	**42%**	252%	407%	**466%**
	≥ 2	TPCH	50%	142%	**167%**	36%	104%	100%	20%	32%	**44%**	154%	409%	**493%**
		Stack	10%	**47%**	30%	10%	**44%**	22%	7%	**24%**	14%	16%	**76%**	73%
		Stack no SSE	16%	74%	43%	17%	67%	39%	10%	**38%**	21%	27%	**154%**	73%
		IMDB	42%	117%	122%	34%	110%	107%	16%	31%	**39%**	197%	387%	**424%**

			25^{th} speed-up factor quantile			5^{th} speed-up factor quantile			median # tuples/construction			median # tuples/compilation	
Type	# '_'	Dataset	Interp.	C++	LLVM	Interp.	C++	LLVM	Interp.	C++	LLVM	C++	LLVM
Start	0	TPCH	264%	266%	**535%**	216%	215%	**466%**	966	1485	**1890**	8.45M	**1.05M**
		Stack	303%	531%	**608%**	183%	355%	**554%**	415	510	**577**	1.31M	**184.68K**
		IMDB	240%	243%	**480%**	185%	172%	**423%**	1304	1673	**2401**	6.88M	**1.06M**
	1	TPCH	237%	261%	**531%**	208%	193%	**490%**	1934	2646	**4657**	10.24M	**1.21M**
		Stack	354%	486%	**600%**	323%	396%	**534%**	519	**1133**	1034	2.66M	**268.43K**
		IMDB	156%	207%	**349%**	142%	141%	**314%**	1377	2065	**3073**	7.28M	**833.88K**
	≥ 2	TPCH	245%	207%	**534%**	217%	174%	**485%**	2706	3155	**5856**	9.92M	**1.32M**
		Stack	359%	487%	**541%**	324%	349%	**528%**	1671	1175	**2444**	2.55M	**322.06K**
		IMDB	148%	193%	**321%**	120%	131%	**258%**	1787	2949	**3686**	8.30M	**824.76K**
End	0	TPCH	169%	235%	**421%**	148%	198%	**400%**	969	1458	**2702**	7.34M	**1.38M**
		Stack	184%	383%	**553%**	19%	80%	39%	472	663	**670**	1.88M	**311.34K**
		IMDB	220%	234%	**450%**	200%	147%	**379%**	1231	1689	**2401**	8.38M	**1.51M**
	1	TPCH	183%	201%	**409%**	162%	166%	**360%**	3996	3862	**6455**	9.79M	**1.86M**
		Stack	323%	269%	**527%**	19%	80%	40%	641	**1413**	1054	2.87M	**360.36K**
		IMDB	182%	230%	**429%**	135%	150%	**403%**	3105	4313	**6995**	10.57M	**1.83M**
	≥ 2	TPCH	113%	194%	**390%**	104%	170%	**370%**	4751	6274	**7763**	10.72M	**1.95M**
		Stack	356%	462%	**537%**	324%	348%	**526%**	2933	4182	**4477**	9.43M	**490.08K**
		IMDB	207%	234%	**392%**	150%	152%	**361%**	5847	8070	**10656**	18.63M	**2.20M**
Middle	0	TPCH	24%	80%	**90%**	20%	56%	68%	267	817	**967**	2.29M	**309.72K**
		Stack	8%	**34%**	**34%**	7%	26%	**27%**	12	67	**77**	163.02K	**21.27K**
		Stack no SSE	19%	76%	75%	18%	59%	**63%**	11	56	**65**	167.71K	**20.64K**
		IMDB	28%	95%	124%	15%	45%	**61%**	277	914	**1145**	2.75M	**373.29K**
	1	TPCH	27%	89%	95%	21%	63%	72%	1056	3477	**3848**	5.79M	**381.45K**
		Stack	8%	**33%**	32%	7%	**26%**	24%	41	200	**225**	417.46K	**26.42K**
		Stack no SSE	18%	73%	74%	17%	**56%**	42%	41	182	**208**	414.17K	**25.71K**
		IMDB	24%	66%	83%	15%	40%	**53%**	472	1334	**1742**	5.31M	**316.97K**
	≥ 2	TPCH	27%	**89%**	73%	22%	**65%**	48%	1991	**5713**	5391	12.10M	**468.59K**
		Stack	9%	**37%**	19%	8%	**29%**	16%	72	**332**	166	938.32K	**9.70K**
		Stack no SSE	14%	**51%**	34%	11%	**44%**	23%	77	**313**	193	963.60K	**9.36K**
		IMDB	24%	**79%**	71%	18%	**51%**	47%	1343	**4834**	4448	11.87M	**423.13K**

Table 6.4: Full benchmarking results, grouped by dataset, approach, pattern type and number of underscores

7 Conclusion

In this chapter, we discuss the potential factors that affect the performance of the presented approaches by interpreting the previously obtained results. Through our experiments, we are confident that our approach offers numerous benefits by avoiding the need to decompress strings. However, no experimental study is exhaustive enough to establish with complete certainty that the proposed algorithm is consistently faster (or slower) than the baseline. Moreover, selecting the appropriate algorithm at runtime is essential to avoid the performance degradation presented in Table 6.4. We begin by introducing the empirical factors that influence the performance of the implemented algorithms.

7.1 Empirical Factors Affecting Performance

Sub-pattern length. For sub-patterns with a length of ≤ 12, the Hybrid String Search algorithm employs SSE4.2 instructions, an approach which has been shown to offer significant speed advantages over the traditional string search algorithms. This performance gain is notable: for the dataset with the longest strings (StackOverflow), the use of SSE instructions resulted in a performance improvement by a factor of two. Conversely, the length of the pattern influences the execution of the proposed automaton-based approach only on a linear scale, as the lengths of paths in the automaton are proportional to the pattern size.

Distribution of characters in the sub-patterns. The distribution of characters appearing in sub-patterns influences the effectiveness of the automaton-based approach, as characters that do not appear often in symbols lead to less branching within the matching algorithm. As the jump lengths of the BM algorithm are low when characters are repeating, the execution time of the Hybrid String Search heavily depends on this distribution as well.

Text length. For patterns containing start sub-patterns or end sub-patterns, we reject a large number of strings in the dataset instantly. As a consequence, the longer the string to check for matches, the faster the automaton approach, since we do not incur the cost of decoding the string to immediately reject it. For middle matches, on the other hand, longer strings imply a smaller speed-up factor, as "bad" characters are parsed through and ignored much more quickly (up to 16 bytes at a time, compared to 1 symbol at a time for the automaton approach). Moreover, for queries with start or end patterns of low selectivity, the majority of the cost of checking for a match still lies in the middle sub-patterns; for the StackOverflow dataset, for example, roughly half of the Comments from PostHistory end with a dot and, as a consequence, the runtime of checking if a string matches "%cannot%." is dominated by checking if it contains "%cannot%". This fact explains the surprisingly low values of the 25^{th} and 5^{th} quantiles of speed-up factors for end patterns on the StackOverflow dataset in Table 6.4.

Compression and symbol table quality. The compression rate of the symbol table decreases when the average symbol length in bytes within the dataset increases. As a consequence, a

higher-quality symbol table increases the number of bytes processed per iteration, thereby narrowing the performance gap between the automaton approach and SSE HSS.

7.2 Immediate Optimization Potential

State Minimisation. The current algorithm constructs the minimal matching automaton for all start and end sub-patterns, as the sub-automaton is either built greedily (if no underscore wildcards are present) or with caching always enabled (if the sub-pattern contains underscore wildcards). However, for middle sub-patterns not containing any underscore wildcards, we might disable caching to avoid forming cycles, even in the case that no cycles would be formed. Furthermore, for middle sub-patterns containing underscore symbols, when splitting the start states and creating the fallback states, we create multiple copies of states, which eventually become equivalent. As such, the final sub-automaton might have equivalent states that can be merged.

Multi-Character Parsing. Parsing the text one character at a time is highly inefficient and results in performance degradation when the symbol table is suboptimal.

In the case of single-start automata, the matching process for a compressed input sequence can be optimised by avoiding standard byte-wise traversal. Instead of state-by-state processing, we first identify the longest deterministic prefix sequence of the automaton, and the compressed input sequence is then checked against this value. As a consequence, we are able to reject instantaneously strings not matching the prefix; otherwise, we start the traversal in the automaton at the state with this prefix, and we shift the index in the compressed string accordingly. For example, for the automaton presented in Figure 3.1, this sequence is $\{67, 255\}$; any compressed string not ending with $\{255, 67\}$ is rejected instantly, and, if a match occurs, traversal starts from q_1.

For middle automata, to avoid traversing redundant sequences one byte at a time, we can either use SSE (16 bytes at a time) or blockwise (8 bytes at a time) instructions to find the first symbol that a start state transitions through, and then ensure it is not preceded by an escape. This is achievable only if the aforementioned start state has ≤ 16 transitions. However, deciding which implementation is the most efficient is heavily data-reliant. If the matching symbols are common throughout the string, then the number of instructions per byte increases when working on 16 bytes at a time; specifically, this occurs if the parsed sequence is of the form $S \to q_0 \to S \to q_1 \to S \to ....$

Boyer-Moore Style Jumps. To improve the parsing speed even further, an idea is to define jump rules, similar to the Boyer-Moore algorithm, that take into account the variable length of a matching sequence.

7.3 Future Work and Outlook

FSST Optimisation. The performance of the presented algorithms depends heavily on the quality of the FSST compression algorithm. However, the compression rate is approximately 2 regardless of the true distribution of the data. As a consequence, further research on possible algorithms for building the symbol table is required to more accurately assess the potential performance gain of this algorithm on analytical data.

ILIKE matching on FSST compressed data. The `ILIKE` keyword in SQL performs LIKE pattern matching in a case-insensitive manner. Given the relevance of character case for the FSST symbols, it would be worthwhile to explore adaptations of the presented algorithms that enable case-insensitive matching.

Regex matching on FSST compressed data. While the `LIKE` and `ILIKE` keywords cover most string-filtering use cases, certain patterns can only be expressed using regular expressions. Given that the presented implementation relies on the assumption that only the '%' and '_' underscores may appear, a fundamentally different approach would be required to support regex matching on FSST encoded data.

Outlook. The previously presented algorithms successfully implement `LIKE` pattern matching algorithms onto FSST encoded data, including all supported SQL wildcards. Empirical analysis demonstrates a significant reduction in the computational cost of the matching function, particularly for patterns containing start or end subpatterns. However, performance optimisation for middle patterns on abnormally long strings remains a challenge. Despite this, avenues for further optimisations have been identified. The overall observed performance improvement suggests a potential expansion of our algorithm to support more complex pattern-matching algorithms.

List of Figures

1.1 Query plans for pattern matching on compressed data (unoptimised on the left, optimised on the right) . . . 2

2.1 Example of an uncompressed dataset, an arbitrary symbol table and the compressed dataset for the presented symbol table . . . 7
2.2 Non-deterministic finite automaton for the pattern "abcabcacab" . . . 11
2.3 Determinized finite automaton for the pattern "abcabcacab" . . . 12
2.4 Initial generated trie for the dictionary ["AAAB", "CALIN", "CALL", "CALI"] . 15

3.1 Automaton for the pattern "%*ABC*" for the symbol table 3.1a . . . 19
3.2 Automaton for the pattern "*ABC*%" for the symbol table 3.1a . . . 24
3.3 Automata for the pattern "%*ABABABAC*%" for the symbol table 3.2 . . . 26
3.4 Automaton for the pattern "%*ABABABAC*%" for the symbol table 3.2 after . . 28
3.5 Final automaton for the pattern "%*ABABABAC*%" for the symbol table 3.2 . . 32

4.1 Automaton for the pattern "%CALIN__" . . . 35
4.2 A side by side comparison of the algorithms building underscore automata . . 40
4.3 Automaton for the pattern "CALIN_POP%" . . . 40
4.5 Initial middle automaton for the pattern "%N_S%" . . . 46
4.6 Middle automaton after start splitting for the pattern "%N_S%" . . . 48

5.1 All sub-automata built for the pattern "*CALIN*_%*GEO*%*POP*" . . . 53
5.2 Connecting the automata for the pattern "*CALIN*_%*GEO*%*POP*" . . . 54
5.3 A side-by-side comparison of the algorithms building underscore automata . . 56

6.1 C++ generated code for the automaton in Figure 3.2 . . . 63
6.2 The average speed-up/slow-down, grouped by dataset and pattern type . . . 65

List of Tables

2.1 Preprocessing data for the pattern "abcabcacab" 10
2.2 KMP search for the pattern "abcabcacab" in the string "babcbabcabcaabcabcab-cacabc" 11
2.3 Bad Character Rule for searching the pattern "abab" in the string "aaaabab" . . 13
2.4 BM search for pattern "ABACABA" in the string "ABABCABACABAB" 14

3.1 Example of a symbol table and patterns, along with the corresponding matching sequences 17
3.2 Symbol table for searching the middle pattern "%ABABABAC%" 25
3.3 Prefix sequences for states of Figure 3.4b 31

6.1 A subset of benchmarked patterns, grouped by dataset, table and column queried 60
6.2 Average lengths of strings from the benchmarked datasets 66
6.3 Parsing statistics for different patterns on the StackOverflow dataset (SF=1) . . 67
6.4 Full benchmarking results, grouped by dataset, approach, pattern type and number of underscores 68

Listings

2.1 Initialise the prefix tree given a dictionary of words 15

3.1 Creating the end automaton for $k = 0$ 18
3.2 Creating the end automaton for $k > 0$ 19
3.3 Creating the start automaton for `idx = pattern.size() - 1` 21
3.4 Creating the start automaton for `idx = pattern.size() - 2` 22
3.5 Creating the start automaton for `idx <= pattern.size() - 3` 24
3.6 Splitting all starts 27
3.7 Adding the symbols containing the pattern 28
3.8 Cached creation of an automaton 29
3.9 Suffix linking process 30
3.10 Link starts of a middle automaton 30
3.11 Creating the fallback states 32
3.12 Transition a trailing state 33
3.13 Build complete middle automaton 33

4.1 Creating the underscore states 36
4.2 Creating the underscore states 37
4.3 Building the automaton for a sub-pattern 39
4.4 Building the automaton for a start pattern 40
4.5 Building the forward automaton for an end pattern 40
4.6 Building the automaton for a middle pattern 40
4.7 Inverting the multiple starts finite automaton 42
4.8 Handle clashing transitions when inverting an automaton 44
4.9 Split one start for an underscore automaton 47
4.10 Create fallback states for an underscore automaton 50

5.1 Adding an end transition 54
5.2 Parsing a compressed string through the forward automaton 56
5.3 Parsing a compressed string through the backwards automaton 56

Bibliography

[AC75] A. V. Aho and M. J. Corasick. "Efficient string matching: an aid to bibliographic search." In: *Communications of the ACM* 18.6 (1975), pp. 333–340.

[BM77] R. S. Boyer and J. S. Moore. "A fast string searching algorithm." In: *Commun. ACM* 20.10 (Oct. 1977), 762–772. ISSN: 0001-0782. DOI: 10.1145/359842.359859.

[BNL20] P. Boncz, T. Neumann, and V. Leis. "FSST: fast random access string compression." In: *Proc. VLDB Endow.* 13.12 (July 2020), 2649–2661. ISSN: 2150-8097. DOI: 10.14778/3407790.3407851.

[BT14] J. Brzozowski and H. Tamm. "Theory of átomata." In: *Theoretical Computer Science* 539 (2014), pp. 13–27.

[CP91] M. Crochemore and D. Perrin. "Two-way string-matching." In: *Journal of the ACM (JACM)* 38.3 (1991), pp. 650–674.

[Dac+00] J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson. "Incremental Construction of Minimal Acyclic Finite-State Automata." In: *Computational Linguistics* 26.1 (Mar. 2000), pp. 3–16. ISSN: 0891-2017. DOI: 10.1162/089120100561601. eprint: https://direct.mit.edu/coli/article-pdf/26/1/3/1797487/089120100561601.pdf.

[Duc] DuckDB Team. *Lightweight Compression in DuckDB*. DuckDB Blog. Accessed: December 8, 2025.

[Gic25] D. Gichev. *Searching and Indexing FSST-Compressed String Data*. Practical Course: Implementation of Database Systems. Presentation. 2025.

[Gru+23] F. Gruber, M. Bandle, A. Engelke, T. Neumann, and J. Giceva. "Bringing compiling databases to RISC architectures." In: *Proceedings of the VLDB Endowment* 16.6 (2023), pp. 1222–1234.

[Gus97] D. Gusfield. "Algorithms on stings, trees, and sequences: Computer science and computational biology." In: *Acm Sigact News* 28.4 (1997), pp. 41–60.

[Hor80] R. N. Horspool. "Practical fast searching in strings." In: *Software: Practice and Experience* 10.6 (1980), pp. 501–506.

[Iso] *ISO/IEC 9075:1992 — Information Technology: Database Language SQL*. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt. Accessed: 2025-11-27. 1992.

[KLN21] T. Kersten, V. Leis, and T. Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra." In: *The VLDB Journal* 30.5 (2021), pp. 883–905.

[KMP77] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. "Fast Pattern Matching in Strings." In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024. eprint: https://doi.org/10.1137/0206024.

[Kna24] K. Knapton. *Drowning in Data for Want of Information: Is Data Minimization Really Possible?* IDC Blog. Accessed: 2025-12-06. Sept. 2024.

[Kus+23] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. "Btrblocks: Efficient columnar compression for data lakes." In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–26.

[MyS] MySQL. *MySQL 8.4 Reference Manual: COMPRESS() Function*. Online Documentation. Accessed: December 8, 2025.

[NF20] T. Neumann and M. J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *CIDR*. Vol. 20. 2020, p. 29.

[Pos25] PostgreSQL Global Development Group. *The Oversized-Attribute Storage Technique (TOAST)*. PostgreSQL. 2025. URL: https://www.postgresql.org/docs/current/storage-toast.html (visited on 12/08/2025).

[Rie23] A. Riedl. "Efficient Code Generation for Pattern Matching in DBMS." Submitted on January 18, 2023. MA thesis. Technische Universität München, School of Computation, Information and Technology - Informatics, Jan. 2023.

[Rie+23] A. Riedl, P. Fent, M. Bandle, and T. Neumann. "Exploiting Code Generation for Efficient LIKE Pattern Matching." In: *VLDB Workshops*. 2023.

[Ryt80] W. Rytter. "A correct preprocessing algorithm for Boyer–Moore string-searching." In: *SIAM Journal on Computing* 9.3 (1980), pp. 509–512.

[Sch+25] T. Schmidt, V. Leis, P. Boncz, and T. Neumann. "SQLStorm: Taking Database Benchmarking into the LLM Era." In: *Proc. VLDB Endow.* 18.11 (July 2025), 4144–4157. ISSN: 2150-8097. DOI: 10.14778/3749646.3749683.

[Spi+24] J. Spindler, P. Fent, A. Riedl, and T. Neumann. "Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression." In: *Proceedings of the VLDB Endowment. ISSN* 2150 (2024), p. 8097.

[SPR16] E. Sitaridi, O. Polychroniou, and K. A. Ross. "SIMD-accelerated regular expression matching." In: *Proceedings of the 12th International Workshop on Data Management on New Hardware*. DaMoN '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450343190. DOI: 10.1145/2933349.2933357.

[Vog+18] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. "Get real: How benchmarks fail to represent the real world." In: *Proceedings of the Workshop on Testing Database Systems*. 2018, pp. 1–6.