Seminar Datenkompression

ZIP-Verfahren: LZ77-Algorithmus

Tobias Völkner 5328855
Philipp Stephan 7182981

Dozent: Dr.- Ing. The Anh Vuong
Graphische Daten Verarbeitung, Informatik Institut
Johann Wolfgang Goethe Universität, Frankfurt am Main

7. Februar 2023

Inhaltsverzeichnis

1 Datenkompression

Datenkompression oder Datenkomprimierung ist ein Vorgang der Menge von digitalen Daten reduziert. Dadurch soll weniger Speicherplatz benötigt und die Übertragungszeit der Daten verringert werden. Der grundsätzliche Ansatz ist dabei, redundante bzw. nicht notwendige Informationen zu entfernen und Daten in verkürzter Form darzustellen. Dieser Vorgang wird Komprimierung genannt. Der umgekehrte Vorgang wird als Dekomprimierung bezeichnet. Dabei werden die reduzierten Daten wieder in ihren lesbaren Zustand zurückgesetzt.

Die Datenkomprimierung unterliegt gewissen Grenzen, da die Daten nicht beliebig gekürzt werden können. Deshalb unterscheidet man zwischen zwei Arten von Komprimierung: Die verlustfreie Kompression, bei der die Originaldaten ohne Verlust von Informationen aus den komprimierten Daten zurückgewonnen werden können, und die verlustbehaftete Kompression, bei der ein Teil der Informationen während der Komprimierung verloren geht. [Wik22a]

1.1 Verlustfreie Kompression

Bei der verlustfreien Kompression muss gewährleistet sein, dass aus den komprimierbaren Daten die Originaldaten wiederhergestellt werden können. Es gehen also keine Informationen verloren. Verlustfreie Kompressionsverfahren nutzen oft die Redundanz, also das mehrfache Vorkommen, von Daten aus.Verlustfreie Kompression wird deshalb auch als Redundanzreduktion bezeichnet. Der Algorithmus zum Dekomprimieren zählt dabei zu den komprimierten Daten hinzu, da die komprimierte Datei sonst wertlos wäre. Ohne eine erkennbare Struktur oder Besonderheiten, ist keine Komprimierung möglich. Verlustfreie Kompression findet oft bei der Komprimierung von Texten Anwendung. [Wik22a]

1.2 Verlustbehaftete Kompression

Bei der verlustbehafteten Kompression werden redundante/irrelevante Informationen ausgelassen, um somit die Datengröße zu reduzieren. Es wird deshalb auch von Irrelevanzreduktion gesprochen. Was als ”redundant”gilt, unterscheidet sich dabei je nach Anwendungsfall. Die Schwelle dafür kann bis zu einer Größe von einem Bit heraufgesetzt werden. Ein Modell entscheidet welche Informationen für den Empfänger wichtig sind, und welche nicht. Verlustbehaftete Kompression wird meist bei Bild-, Audio-, oder Videodateien und -übertragungen angewandt. [Wik22a]

2 Theoretische Grundlagen

In der Datenkompression gibt es verschiedene Methoden, Algorithmen und Dateiformate. Im Folgenden werden einige Grundlagen für den LZ77-Algorithmus erklärt.

2.1 Die Wörterbuchmethode

Die Wörterbuchmethode ist eine Art der verlustfreien Kompression. Bei diesem Verfahren werden wiederkehrende Zeichen oder Zeichenketten durch Abkürzungen, sogenannte Tokens, ersetzt. Um die Tokens den zugehörigen Zeichen(-ketten) zuordnen zu können, wird ein Wörterbuch angelegt. Dieses Wörterbuch muss in den komprimierten Daten mit aufgenommen werden, damit die ursprünglichen Daten wieder hergestellt werden können. Diese Methode ist besonders nützlich, wenn sich eine kleine Anzahl Muster sehr häufig im Text wiederholt. Wörterbuchmethoden werden in zwei Kategorien unterteilt. Wenn es genügend Kenntnis über den Text gibt, wird ein statisches Verfahren - mit einem festen Wörterbuch - verwendet. Ist das nicht der Fall, wird ein dynamisches Verfahren eingesetzt, bei dem das Wörterbuch im Laufe der Codierung erstellt wird. Ein großer Vorteil der Wörterbuch-Techniken ist, dass ganze Zeichenketten codiert werden können und nicht nur einzelne Zeichen. [Wik20, LF06]

Statische Verfahren Statische Wörterbuch-Verfahren sind in der Regel nur für spezielle Fälle sinnvoll. Das ist dann der Fall, wenn Zeichen bzw. Zeichenketten sehr häufig vorkommen, beispielsweise Dateien, in denen Leistungen von Studenten erfasst sind. Wörter wie ”Vorname”, ”Nachname”, ”Matrikelnummer” und ”Note” kommen dort sehr häufig vor. [LF06]

Dynamische Verfahren Bei dynamischen Verfahren wird das Wörterbuch erst bei der Codierung erstellt. Es wird also vorher keine Kenntnis über den Text benötigt und das Wörterbuch enthält nur die Informationen die auch gebraucht werden. [LF06]

2.2 Das ZIP-Dateiformat

Das ZIP-Dateiformat ist ein Dateiformat für verlustfrei komprimierte Daten und dient zur Archivierung oder zum Versand von Dateien. Das erste ZIP-Dateiformat wurde 1989 vom US-Amerikaner Phil Katz mit dem Programm PKZIP für die Kompression und PKUNZIP für die Dekompression entwickelt. Weitere ZIP-Formate basieren auf dieser ersten Herausgabe. Die Kompression in einer ZIP-Datei reduziert die Größe der Ursprungsdaten bei der Archivierung und wird als Containerformate gespeichert. Alle Dateien werden individuell komprimiert und dann der Archiv-Datei zusammengefasst. Für das ZIP-Dateiformat gibt es verschiedene Kompressions-Algorithmen. Einer der bekanntesten ist der LZ77- Algorithmus von Abraham Lempel und Jacob Ziv. [Wik22b]

3 Der LZ77-Algorithmus

LZ77 ist ein Verfahren zur verlustfreien Kompression von Daten. Es wurde 1977 von Abraham Lempel und Jacob Ziv im Fachmagazin "IEEE Transactions of Information Theory" veröffentlicht (daher der Name "Lempel-Ziv-1977"). Es handelt sich hierbei um ein Wörterbuch-Verfahren, welches sich als erstes seiner Art mehrfach auftauchende, ganze Sequenzen an Zeichen in einem Datensatz zunutze macht, anstatt die Wahrscheinlichkeiten für das Auftreten von Zeichen oder Zeichenfolgen zu verwenden (Entropiekodierung). Es ist das erste von vielen darauf folgenden LZ-Verfahren und bildete die Grundlage für weitere Kompressionsverfahren und viele Varianten, mit denen die Kompressionsrate weiter verbessert wird.

Das Ziel von Lempel und Ziv war es einen Algorithmus zu entwickeln, der ohne Vorkenntnisse über den Text und ohne Voruntersuchung der Eingabe, effektiv komprimiert. Der LZ77-Algorithmus liest die Daten nur ein einziges mal, die Eingabe erfolgt zeichenweise und baut auf den vorher eingelesenen Teilen des Datenstroms auf. Daraus folgt der Vorteil die Daten Stück für Stück verarbeiten zu können, auch wenn diese noch nicht vollständig zur Verfügung stehen. Da keine Voranalyse der zu komprimierenden Daten stattfindet, ist die Kompression nicht immer zwangsläufig optimal. Im schlimmsten Fall kann aufgrund weniger Redundanzen sogar eine Vergrößerung der Datenmenge auftreten.

Da Zeichenfolgen nur dann ersetzt werden können, wenn diese vorher schon einmal aufgetreten sind, ist die Kompressonsrate anfangs noch sehr gering und der Algorithmus liefert erst mit fortschreitendem Einlesen der Eingabedaten bessere Ergebnisse. Daraus folgt, dass zu kleine Eingabestrings zu schlechten Kompressionsraten führen. [Wan06, Wik21, But12]

3.1 Kompressionsvorgang

Ein Eingabe-Text wird komprimiert, indem man nach sich wiederholenden Zeichenketten sucht und diese durch eine Referenz ersetzt, die auf das Original verweist. Diese Referenzen werden anschließend in einem Wörterbuch gespeichert. Falls ein Zeichen zum ersten Mal auftaucht, wird dieses im Wörterbuch ohne Referenz gespeichert. Dabei wird der Eingabetext zeichenweise durch einen Buffer, das sogenannte Sliding Window, bewegt. Dieses Sliding Window ist unterteilt in den Search-Buffer und den Lookahead-Buffer. Der Search-Buffer ist dabei üblicherweise mehrere Tausend Zeichen lang, der Lookahead-Buffer mehrere Hundert Zeichen.

Der Algorithmus durchläuft dabei den Text, definiert als Zeichenkette $Z = s_1, ..., s_k$, und sucht für jedes Symbol $s_i \in Z$, ein Match in der Zeichenkette $Z_i = s_{i-1}, ..., s_{i-x}$. Dabei gilt $x \leqslant i \leqslant k$. Die Zahl x ist die Länge des Search-Buffers, der die Zeichenkette Z_i rückwärts durchläuft.
Der Searchbuffer durchsucht dabei den Text rückwärts, ausgehend des aktuellen Symbols, nach einem Match. Bei Fund eines Matches zweier Symbole s_i und $s_j : (i - x \leqslant j < i)$ wird, mithilfe des Lookahead-Buffers der Länge l, geprüft ob eine Sequenz existiert.
Für ein Match einer Sequenz wird ein Tupel der Form $(offset,\ length,\ nextSymbol)$ erstellt, definiert als:

- $offset$:= der Abstand zwischen den jeweils ersten Symbolen der beiden Sequenzen
- $length$:= die Länge der Sequenz
- $nextSymbol$:= das nächste Symbol nach Abschluss der Sequenz

Die Funktion für eine gefundene Sequenz der Länge $m \leqslant l$ liest sich daher wie folgt:

$$s_j, ..., s_{j+m} = s_i, ..., s_{i+m} \Rightarrow (i - j, m, s_{i+m+1})$$

Für den Fall, dass sich keine Sequenz finden lässt, wird folgendes Tupel t erstellt:

$$(0, 0, s_i)$$

Die Tupel werden anschließend, in Reihenfolge, in ein Wörterbuch $L := t_1, ..., t_g$ eingetragen.

Die Effektivität der Kompression des LZ77-Algorithmus hängt hauptsächlich von der Größe des Sliding Window ab. Ein kleinerer Search-Buffer führt zu weniger gefundenen Übereinstimmungen. Ein kleinerer Lookahead-Buffer führt zu kürzeren Übereinstimmungen. Bei kleinen Buffer-Größen ist die Kompressionsrate also üblicherweise gering. Werden die Buffer allerdings zu groß gewählt, bedeutet das einen höheren Bedarf an Arbeitsspeicher und die Laufzeit der Suche nach Übereinstimmungen verlängert sich. [Wan06]

Pseudocode Das Vorgehen des LZ77-Algorithmus bei der Kompression, dargestellt in Pseudocode. Im Anhang 8.1 befindet sich eine vollständige Implementierung in der Programmiersprache Javascript.

```
while der Lookahed Buffer ist nicht leer;
    durchsuche rückwärts den Text nach der längsten
           übereinstimmenden Zeichenkette mit dem Lookahead Buffer;

    if eine Übereinstimmung wurde gefunden;
        füge das Tripel (Versatz zum Rand des Lookahead Buffers,
                         Länge der gefundenen Zeichenkette,
                         erstes nicht übereinstimmendes Zeichen aus dem Lookahead Buffer)
                dem Wörterbuch hinzu;
        verschiebe das Sliding Window um die Länge+1;
    else
        füge das Tripel (0,
                         0,
                         erstes Zeichen im Lookahead Buffer)
                dem Wörterbuch hinzu;
        verschiebe das Sliding Window um 1;
```

Beispiel Codierung des Textes ”abracadabra!”. Die linke Spalte der Tabelle symbolisiert den Search-Buffer mit der Größe 12, die Mittlere den Lookahead-Buffer mit der Größe 4 und die Rechte das Wörterbuch. Gefundene Übereinstimmungen sind unterstrichen geschrieben.

Sliding Window			Wörterbuch
Search-Buffer	Lookahead-Buffer		
	a b r a	c a d a b r a !	(0, 0, a)
a	b r a c	a d a b r a !	(0, 0, b)
a b	r a c a	d a b r a !	(0, 0, r)
a b r	a c a d	a b r a !	(3, 1, c)
a b r a c	a d a b	r a !	(2, 1, d)
a b r a c a d	a b r a	!	(7, 4, !)
a b r a c a d a b r a !			

Tabelle 1: Veranschaulichung der Kompression.

3.2 Dekompressionsvorgang

Für die Dekompression wird das Wörterbuch ausgeschrieben. Dabei ist das Wörterbuch definiert als $L := t_1, ..., t_k$ und Tupel $t := (offset,\ length,\ nextSymbol)$. Für jedes Tupel aus dem Wörterbuch wird ein entsprechender Text generiert. Für den Fall, dass $offset$ und $length$ beide 0 sind, wird nur $nextSymbol$ geschrieben. Das erste Tupel eines Wörterbuchs wird immer der Form (0, 0, $nextSymbol$) sein und setzt damit den initialen Text $T := z_1$.

Typischerweise werden die ersten Tupel keine Referenzen enthalten. Jeder neu generierte Textabschnitt wird dann an T angefügt. Wenn ein Tupel mit einer Referenz auf eine Zeichenkette gefunden wird, wird diese im aktuellen T gesucht. Dabei wird die Zeichenkette $z_{l-offset}, ..., z_{l-offset+length}$ wiederholt und an T gefügt.

Zum Schluss wird $nextSymbol$ an die Zeichenkette gefügt. Sollte $nextSymbol$ leer sein, bleibt T unverändert und dies bedeutet immer, dass der letzte Tupeleintrag erreicht wurde.

Pseudocode Das Vorgehen des LZ77-Algorithmus bei der Dekompression, dargestellt in Pseudocode. Im Anhang 8.2 befindet sich eine vollständige Implementierung in der Programmiersprache Javascript.

```
for jedes Tripel (Versatz, Länge, Zeichen);
     if Länge > 0;
         durchlaufe rückwärts die bisherige Ausgabe und gib solange Zeichen aus
                 bis Länge erreicht ist,
                 bei einem Überlauf beginne erneut bei Versatz;
     Gib Zeichen aus;
```

Beispiel Decodierung des in Tabelle 1 erstellten Wörterbuchs. Unterstrichen geschriebene Buchstaben sind die für die Rekonstruktion verwendeten Buchstaben.

Wörterbuch	Text
(0, 0, a)	a
(0, 0, b)	a b
(0, 0, r)	a b r
(3, 1, c)	a b r a c
(2, 1, d)	a b r a c a d
(7, 4, !)	a b r a c a d a b r a !

Tabelle 2: Veranschaulichung der Dekompression.

3.3 Speichereinsparung

Bei der Kompression wird für die Tupel-Einträge unterschiedlich viel Platz alloziert, je nach dedizierter Länge der Buffer. Für die Länge der Buffer werden entsprechend viele Bits alloziert. Dabei würde im oben genannten Beispiel (”abracadabra!”) für einen Search-Buffer der Länge 12, $\lceil \log_2(12) \rceil = 4$ Bits alloziert und für einen Lookahead-Buffer der Länge 4, $\lceil \log_2(4) \rceil = 2$ Bits. Für die Symbole wird im Normalfall 1 Byte, also 8 Bits alloziert. Das Wort ”abracadabra!” enthält insgesamt 12 Symbole, damit 96 Bits. Das entsprechende Wörterbuch enthält 6 Einträge, also $6 \cdot (4+2+8)$ Bits = 84 Bits. Damit hätte man für das Beispiel eine Einsparung von insgesamt 12 Bits. Die Speichereinsparung verändert sich dynamisch, je nach Textart, Bufferlänge und Textlänge. [Sch19]

4 Anwendungsgebiet

Das LZ77-Verfahren wird heutzutage nur noch selten verwendet. Allerdings ist es die Grundlage für viele, darauf aufbauende Kompressionsalgorithmen, wie, unter anderem, der LZ78-Algorithmus von 1978, der Lempel-Ziv-Storer-Szymanski-Algorithmus (LZSS), der Lempel-Ziv-Welch-Algorithmus (LZW) und die Lempel-Ziv-Markov-Kompression (LZMA).

Die Nachfahren des LZ77 finden in vielen Bereichen noch heute Anwendungen, zum Beispiel: Gif, PDF, TIFF, im Unix-'compress' command und vielen mehr. Das eigentliche LZ77-Verfahren wird heutzutage zum Beispiel noch auf dem Game Boy Advance, dem AutoCAD DWG Format und weiteren eingebetteten Systemen verwendet. Kombiniert mit der Huffman-Kodierung wird LZ77 im häufig verwendeten Deflate-Algorithmus angewandt. Dieser wird unter anderem vom Grafikformat PNG und anderen sehr bekannten Kompressionsprogrammen eingesetzt. In der algorithmischen Verarbeitung von Zeichenketten wird LZ77 außerdem zur Erkennung von Regelmäßigkeiten in Strings genutzt. Da die Weiterentwicklungen allgemein effizienter sind, werden diese heutzutage meist der LZ77-Kompression bevorzugt. [Wik22a]

5 Qualitätsbewertung

Vorteile Das LZ77-Verfahren hat den großen Vorteil, dass es ohne Textkenntnis komprimieren kann. Des Weiteren ist es mit keinem Patent belegt, was bedeutet, dass die Verwendung des Algorithmus' sehr unkompliziert ist. LZ77 kommt außerdem mit wenig Systemressourcen aus und liefert dafür hinreichende Ergebnisse. Der Algorithmus dient zudem heutzutage oft als Ausgangspunkt, um neue Algorithmen zu entwerfen, da er recht einfach zu erlernen, zu programmieren und zu erweitern ist. Mit Variationen des Algorithmus lassen sich somit sehr effektive Kompressionsverfahren erstellen, die mit den heutigen komplexen Verfahren auf einem Niveau sind.

Nachteile Ein offensichtlicher Nachteil des Verfahrens ist, dass bei wenigen Wiederholungen innerhalb des Textes oder der Zeichenkette nur wenige Referenzen erstellt werden können oder die Datenmenge im worst-case sogar vergrößert werden kann. Das ist insbesondere bei kleinen oder nicht natürlichsprachigen Texten der Fall. Des Weiteren werden beinahe ausschließlich Zeichenfolgen genutzt, die sich nah um den Zeiger der aktuellen Iteration befinden, da der Lookahead- und Search-Buffer die Reichweite der gefundenen Matches limitieren. Diese Buffer sind natürlich unendlich erweiterbar, allerdings mit Befall massiver Kosten der CPU-Laufzeit bei der Kompression und Auswirkung auf die Speichergröße der Tupel durch die größere Bufferlänge.

Fazit Der LZ77-Algorithmus dient heutzutage eher als sogenannter Präprozessor, also der Vorverarbeiten des Textes, um danach mit einem anderen Kompressionsverfahren (beispielsweise der Huffman-Kodierung) stärker zu komprimieren oder er dient als Grundlage, dafür neue, mächtigere Kompressionsalgorithmus zu entwickeln. [Mü06, Wik13, Wik21, Rob00]

6 Demonstrations-Kit

Das Demonstrations-Kit besteht aus drei Bereichen. Ein Bereich für die Eingabe, ein weiterer für die Visualisierung der Kompression und ein letzter für die Visualisierung der Dekompression.

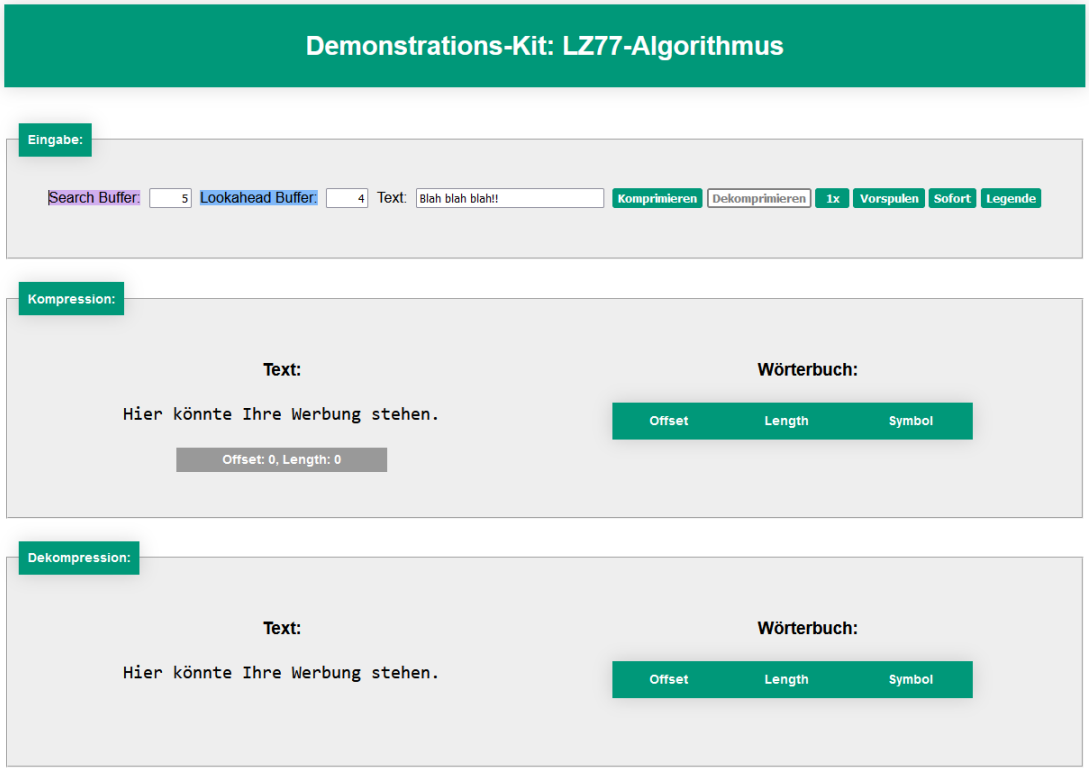

Abbildung 1: Oberfläche des Demonstrations-Kit.

Im Eingabe-Bereich lässt sich die Größe des Search- und Lookahead-Buffers festlegen, sowie einen Text von bis zu 1000 Zeichen eingeben. Dieser Text wird dann komprimiert bzw. dekomprimiert. Der Button zum Dekomprimieren wird erst nach der Komprimierung freigeschaltet. Zusätzlich lässt sich mit je einem Button die Geschwindigkeit der Visualisierung regeln, vorspulen oder sie sofort beenden.

Abbildung 2: Eingabe-Bereich des Demonstrations-Kit.

Der Kompressions-Bereich stellt die Kompression des LZ77-Algorithmus visuell anschaulich dar. Hier wird auf der linken Seite der Text zusammen mit dem Sliding Window angezeigt. Die linke, Violette Box stellt dabei den Search Buffer, die rechte, Blaue Box den Lookahead Buffer dar. Die Suche nach einer Übereinstimmung im Search Buffer wird durch **Fett gedruckte unterstrichene** Zeichen visualisiert. Übereinstimmende Zeichen werden Grün markiert, nicht mehr übereinstimmende Zeichen Rot. Leerzeichen werden durch die

Glyphe "␣" ersetzt. Unterhalb des Textes wird der Offset und die Länge der aktuell besten/längsten gefundene Übereinstimmung angezeigt. Auf der rechten Seite befindet sich das Wörterbuch mit den entsprechenden Einträgen.

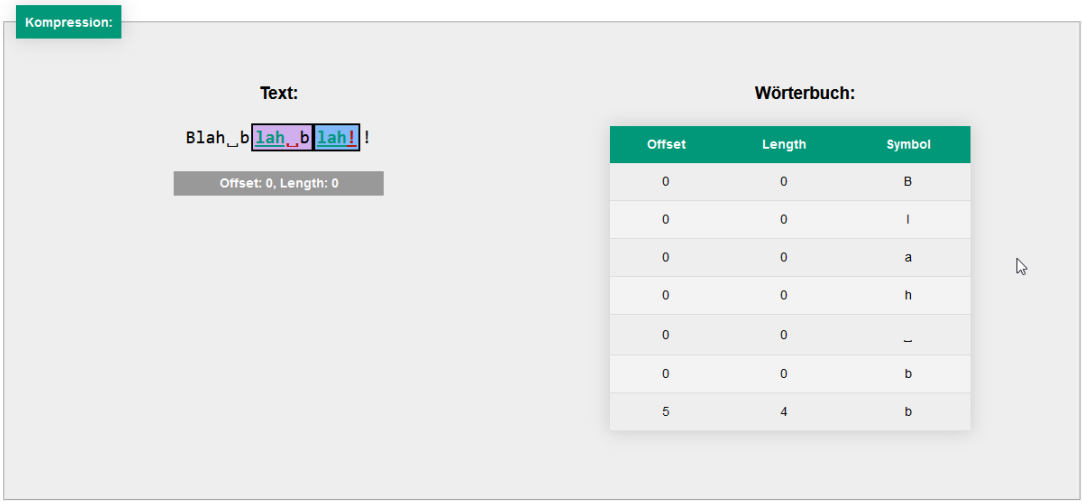

Abbildung 3: Kompressions-Bereich des Demonstrations-Kit mit Visualisierung.

Der Dekompressions-Bereich veranschaulicht die Dekompression. Hier wird ebenfalls auf der linken Seite der Text angezeigt. Neu hinzugefügte Zeichen werden hier **Fett gedruckt und unterstrichen**. Zeichen die mittels Wörterbuch-Eintrag aus dem bereits decodierten Text geholt werden, sind Grün dargestellt. Auf der rechten Seite wird auch hier das Wörterbuch angezeigt. Dieses wird von der Kompression an die Dekompression übergeben. Der aktuell bearbeitete Eintrag wird Grau hervorgehoben.

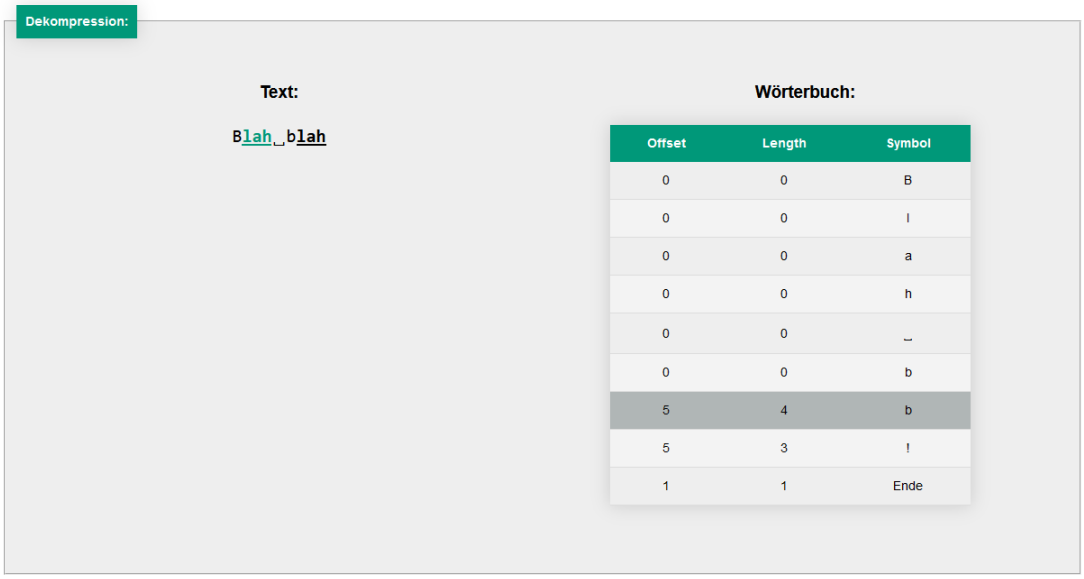

Abbildung 4: Dekompressions-Bereich des Demonstrations-Kit mit Visualisierung.

7 Referenzen

Literatur

[But12] Butting, Arvid: *Der Lempel-Ziv-Markov Chain Algorithmus*. https://tcs.rwth-aachen.de/lehre/Komprimierung/SS2012/ausarbeitungen/Lempel-Ziv.pdf, 2012. – [Online; Stand 05. Februar 2023]

[LF06] Liskiewicz, Maciej ; Fernau, Henning: *Datenkompression*. https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pdf, 2006. – [Online; Stand 09. Dezember 2022]

[Mü06] Mühlinghaus, Stefan: *Der LZ77 Algorithmus*. http://www.gm.fh-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/D_rot/Ausarbeitung.pdf, 2006. – [Online; Stand 23. Januar 2023]

[Rob00] Roberts, Eric: *Dictionary-based Compressors*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/2000-01/data-compression/lossless/lz77/index.htm, 2000. – [Online; Stand 05. Februar 2023]

[Sch19] Schwarz, Heiko: *Dictionary-based Coding*. https://iphome.hhi.de/schwarz/assets/dc/07-DictionaryCoding.pdf, 2019. – [Online; Stand 05. Februar 2023]

[Wan06] Wander, Matthäus: *Verlustfreie Datenkompression mittels der LZ-Algorithmen*. http://jbg.swznet.de/uni/lzw-ausarbeitung.pdf, 2006. – [Online; Stand 22. Januar 2023]

[Wik13] Wikibooks: *Datenkompression: Verlustfreie Verfahren: Wörterbuchbasierte Verfahren: LZ77*. https://de.wikibooks.org/wiki/Datenkompression:_Verlustfreie_Verfahren:_Wörterbuchbasierte_Verfahren:_LZ77, 2013. – [Online; Stand 23. Januar 2023]

[Wik20] Wikipedia: *Wörterbuchkompression*. https://de.wikipedia.org/wiki/Wörterbuchkompression, 2020. – [Online; Stand 24. November 2022]

[Wik21] Wikipedia: *LZ77*. https://de.wikipedia.org/wiki/LZ77, 2021. – [Online; Stand 24. November 2022]

[Wik22a] Wikipedia: *Datenkompression*. https://de.wikipedia.org/wiki/Datenkompression, 2022. – [Online; Stand 24. November 2022]

[Wik22b] Wikipedia: *ZIP-Dateiformat*. https://de.wikipedia.org/wiki/ZIP-Dateiformat, 2022. – [Online; Stand 24. November 2022]

8 Anhang

8.1 Kompressionsalgorithmus

Der LZ77-Kompressionsalgorithmus implementiert in Javascript:

```
function encode(string, searchBufferLength, lookaheadBufferLength) {
    let dictionary = new Dictionary();
    let i = 0;

    // loop through string:
    while (i < string.length) {
        let char = string.charAt(i);
        let offset = 0;
        let maxLength = 0;
        let j = i-1;

        // go backwards through searchbuffer:
        while (j >= 0 && i-j <= searchBufferLength) {
            if (string.charAt(i) == string.charAt(j)) {
                // match found
                let length = 1;
                // get length of match:
                while (length < lookaheadBufferLength && i+length < string.length) {
                    if (string.charAt(j+length) == string.charAt(i+length)) {
                        length++;
                    } else {
                        break;
                    }
                }
                // update if new best match:
                if (maxLength < length) {
                    maxLength = length;
                    offset = i-j;
                    if (i+length < string.length) {
                        char = string.charAt(i+length);
                    } else {
                        char = "Ende";
                    }
                }
            }
            j--;
        }
        // add to dictionary:
        dictionary.addEntry(new Entry(offset, maxLength, char));

        // move sliding window:
        if (maxLength == 0) {
            i++;
        } else {
            i = i+maxLength+1;
        }
    }
    return dictionary;
}
```

8.2 Dekompressionsalogrithmus

Der LZ77-Dekompressionsalgorithmus implementiert in Javascript:

```
function decode(dictionary) {
    let decodedString = "";

    // loop through dictionary:
    for (let e = 0; e < dictionary.length; e++) {
        let offset = dictionary[e].getOffset();
        let length = dictionary[e].getLength();
        let char = dictionary[e].getNextSymbol();

        // put string together:
        if (length > 0) {
            for (let i = 0; i < length; i++) {
                decodedString += decodedString.charAt(decodedString.length - offset);
            }
        }
        if (char != "Ende") {
            decodedString += char;
        }
    }
    return decodedString;
}
```