1. Grundlagen

Im Folgenden werden Grundbegriffe und Zusammenhänge der Kompression, Approximation und insbesondere der LZ-Zerlegung erläutert.

1.1 Kompression

Kompression ist die Transformation von Daten in eine kleinere Menge an Daten. Es gibt zwei Möglichkeiten, diese Transformation auszuführen, verlustfrei oder verlustbehaftet. Alle verlustfreien Ansätze basieren darauf, Redundanzen innerhalb eines Datensatzes zu eliminieren, während verlustbehaftete Methoden ausgewählte Daten löschen. Der bedeutende Unterschied zwischen diesen beiden Ansätzen ist, dass sich nur aus einer verlustfreien Kompression das Original wiederherstellen lässt.
Die Güte einer Kompression, der *Kompressionsgrad*, ist definiert als:

$$\text{Kompressionsgrad} = \frac{\text{Größe der Ausgabe}}{\text{Größe der Eingabe}} \tag{1.1}$$

Während der *Kompressionsfaktor* das Inverse des Kompressionsgrades ist:

$$\text{Kompressionsfaktor} = \frac{\text{Größe der Eingabe}}{\text{Größe der Ausgabe}} \tag{1.2}$$

Keine verlustfreie Kompressionsmethode kann einen Kompressionsgrad < 1 garantieren, ein Kompressionsgrad ist immer von der Eingabe abhängig. Sollte theoretisch eine solche Methode existieren, könnte man sie immer wieder auf ihre eigenen Ausgaben anwenden und jegliche Datenmenge in nur einem einzelnen Bit codieren, was offensichtlich unmöglich ist [?].
Im Folgenden beschäftige ich mich ausschließlich mit verlustfreien Verfahren, der Begriff der Kompression bezieht sich deshalb immer auf die verlustfreie Kompression.

1.2 Strings

Eine fundamentale Art Informationen darzustellen ist der *String*. Ein *String* s der Länge n ist eine Folge von Symbolen $s[1], s[2], \cdots, s[n-1]$ aus einem Alphabet Σ.
Ein String u der Länge m ist Substring eines Strings w der Länge n, wenn für ein $i < n-m$ gilt: $u[1]u[2]\cdots u[m-1] = w[i]w[i+1]\cdots w[m-1]$
Ein Substring u über W besitzt einen *vorherigen Substring*, wenn für u mit Länge m beginnend an Index i ein x existiert mit $x < i$ und $w[i+0]w[i+1]\cdots w[m-1] = w[x+0]w[x+1]\cdots w[x-1]$.
Für Substrings existieren zwei Kategorien von besonderer Bedeutung, Präfixe und Suffixe. Präfixe sind Substrings die an Index 1 beginnen, während Suffixe Substrings sind die an Index n enden. Als *echte Substrings* bezeichnet man Substrings die kürzer sind als der String in dem sie liegen. Im Folgenden soll $Substring[i, l]$ den Substring der an Index i beginnt und an Index l endet bezeichnen.
In *C++* existieren zwei Konzepte die als String bezeichnet werden, der *C-style char array* und die *std::string* Klasse. Das *C-style char array* ist ein kontinuierlicher Block an Speicher mit fester Größe. In diesem Block liegen *chars*, die Symbole darstellen, direkt hintereinander. Die Größe des *arrays* kann nachträglich nicht verändert werden.
Die *std::string* Klasse spezifiziert Objekte, die ein *char array* enthalten. Objekte der Klasse String bieten weitere Funktionalitäten und die Möglichkeit, die Länge des Strings zu verändern [**?**].

1.3 LZ-Zerlegung

Eine LZ-Zerlegung ist eine Aufteilung eines String T in sich nicht überschneidende Substrings, diese Substrings bezeichnet man als Faktoren. Für jeden Faktor gilt, dass er entweder ein Substring ist zu dem ein vorheriger Substring existiert oder der Faktor aus einem einzelnen Symbol besteht. Des weiteren ist jeder Faktor maximal in seiner möglichen Länge. Dass heißt für jeden Faktor $f[i,l]$, also jeden Substring$[i,l]$ in der LZ-Zerlegung $LZ()$ der Eingabe T gilt:

Die Faktoren überschneiden sich nicht:

$$\forall f_x[i_x,l_x] \in LZ(T).\ i_x = l_{x-1} + 1 \vee (x = 1 \wedge i_x = 1) \tag{1.3}$$

Die Faktoren sind einzelne Symbole oder besitzen vorherige Substrings:

$$\forall f_x[i_x,l_x] \in LZ(T).\ l_x = i_x + 1 \vee \exists k \in \mathbb{N}.k < i \wedge \bigwedge_{j=1}^{l-i} T[k+j] = T[i+j] \tag{1.4}$$

Die Faktoren sind in ihrer möglichen Länge maximal:

$$\forall f_x[i_x,l_x] \in LZ(T).\ \nexists k \in \mathbb{N}.\ k > l_x \wedge \exists j \in \mathbb{N}.j < i \wedge \bigwedge_{h=1}^{k} T[j+h] = T[i+h] \tag{1.5}$$

Die LZ-Zerlegung erlaubt es, einen String komprimiert darzustellen. Diese Kompression wird erreicht, indem in der LZ-Zerlegung Faktoren durch Referenzen ersetzt werden. Jede Referenz ist ein Tupel aus zwei Zahlen, das einen vorher liegenden Substring identifiziert Das erste Element des Tupels enthält Informationen über die Position des vorher liegenden Substrings, während das zweite Element die Länge des Substrings angibt. Die Positionsangabe kann dabei ein *offset* oder ein absoluter Index über den faktorisierten String sein. Ein *offset* findet häufiger Verwendung, da für größere Strings ein absoluter Index $\lceil \log_2 n \rceil$ Bits belegt, wobei $n = |Eingabe|$.
Allerdings werden nicht alle Substrings durch Referenzen ersetzt. Wenn die zugehörige Referenz mehr Speicher benötigt, als der Faktor selbst, lohnt es sich nicht den Substring zu ersetzten. Ebenfalls wird keine Referenz gespeichert, wenn der Faktor ein einzelnes Symbol ist [**?**].

1.4 Approximation

Zu jedem Optimierungsproblem existiert eine Menge an richtigen Lösungen und in dieser eine Teilmenge an optimalen Lösungen. Ein Approximationsalgorithmus erzeugt für eine Eingabe eine richtige Lösung, die aber nicht zwangsläufig optimal sein muss.
Die *Approximationsgüte* ρ ist eine Garantie, wie sehr die erzeugte Lösung s_e maximal von einer optimalen Lösung s_{opt} abweicht. Um komplexe Ergebnisse vergleichen zu können, evaluiert man beide durch eine Bewertungsfunktion *f(x)* [**?**].

$$\rho = maximum \left\{ \frac{f(s_e)}{f(s_{opt})}, \frac{f(s_{opt})}{f(s_e)} \right\} \tag{1.6}$$

Da eine Abweichung vom Optimum sowohl größer als auch kleiner als „1“ sein kann ist der größere der beiden Brüche die Güte.

1.5 LZ-Approximation

Zu jedem beliebigen String existiert eine LZ-Zerlegung. Eine Approximation dieser Zerlegung besteht, wenn die Faktoren in ihrer Länge nicht zwingend maximal sind. Die Faktoren der Approximation sind weiterhin entweder einzelne Symbole oder besitzen vorherige Substrings.
Eine *c-optimale Approximation* besteht dann, wenn keine c aufeinander folgenden Faktoren selber einen Faktor bilden. Daraus folgt, dass in einem Faktor der LZ-Zerlegung maximal c-1 ganze Faktoren liegen können. Es kann aber sein, dass sowohl ein echtes Suffix und ein echtes Präfix der umliegenden Faktoren ebenfalls in dem Faktor der LZ-Zerlegung enthalten sind. Im ungünstigsten Fall wird damit ein Faktor vollkommen zwischen zwei Faktoren der LZ-Zerlegung aufgeteilt (vgl. Abb. 1.1).

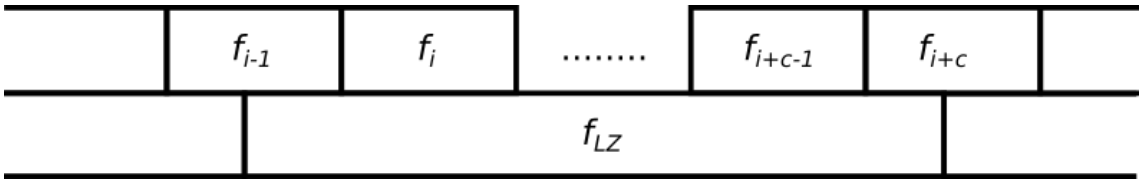

Abbildung 1.1: Der Faktor f_{LZ} erstreckt sich von f_{i-1} bis f_{i+c}

Das bedeutet, dass wir maximal c-mal mehr Faktoren in der *c-optimale* LZ-Approximation finden, als in der LZ-Zerlegung. Die Anzahl der Faktoren in der Approximation ist also durch die Anzahl der Faktoren in der LZ-Zerlegung multipliziert mit c nach oben beschränkt.

$$\rho = max \left\{ \frac{c * f(s_{opt})}{f(s_{opt})}, \frac{f(s_{opt})}{c * f(s_{opt})} \right\} \tag{1.7}$$

$$\rho = c \tag{1.8}$$

Aus einer *c-Optimalität* der Approximation folgt eine Güte der Approximation von c [**?**].

1.6 Hashfuntionen & Hashmaps

Eine *Hashfunktion* h ist eine deterministische, polstellenlose Funktion von einem *Universum* U auf eine beschränkte Teilmenge der natürlichen Zahlen $h : U \to \{0, 1, 2 \cdots, n\}$. Die Eingaben bezeichnet man als Schlüssel und die Ausgaben als Hashcodes, eine *Hashfunktion* definiert damit *key value pairs*. Wenn unterschiedliche Schlüssel den gleichen *Hashcode* besitzen, bezeichnet man dies als *Kollision*. Wenn die Menge der Eingaben größer ist als die Menge der Hashcodes, sind diese unvermeidlich.
Eine $Hashfunktion$ wird anhand folgender zentralen Kriterien bewertet:

- Verteilung: Alle möglichen *Hashcodes* sollen gleich wahrscheinlich getroffen werden
- Dichte: Die Menge aller *Hashcodes* soll möglichts klein sein
- Wahrscheinlichkeit einer Kollision: Kollisionen sollen so unwahrscheinlich wie möglich sein

Eine Hashmap ist eine Datenstruktur die $look-ups$,$insertions$ und $deletion$ in amortisiert konstanter Zeit ermöglicht. Dies ist einer der häufigsten Anwendungsfälle der Hashfunktionen. Beim Einfügen eines Elementes e der wird *hashcode* von e, $h(e)$, berechnet und e unter Index $h(e)$ abgespeichert, dabei kann es hier zu einer Kollision kommen. Um zu überprüfen, ob ein Element e in der hashmap enthalten ist genügt es den *hashcode* zu berechnen und nur den so erhaltenen Index zu untersuchen. Das Löschen eines Elementes verläuft ebenso, außer das nach dem Untersuchen das entsprechende Element gelöscht wird.
Um mit Kollisionen effizient umzugehen, existieren viele Ansätze,zuden bekanntesten zählen: die verschiedenen Elemente einfach in einer Liste zu verkettet, hinter jedem Index eine zweite hashmap zu bilden und das entsprechende Element einfach in dem nächsten freien Index zu speichern. Alle Kollisionsresolutionsmethoden haben ihre eigenen Vor- und Nachteile.

1.7 *Fingerprint* und *rolling Hash*

Als *Fingerprint* bezeichnet man den *Hashcode* eines Strings. *Fingerprints* erlauben es Strings schneller zu vergleichen. Um herauszufinden, ob zwei Strings s_1, s_2 gleich sind müssen wir alle Symbole miteinander vergleichen. Sollten die beiden Strings sich nur im letzten Symbol unterscheiden so haben wir alle vorherigen Symbole umsonst überprüft. Vergleichen wir aber vorher beide *Fingerprint* s können wir nicht gleiche Strings direkt erkennen, dies ist wesentlich effizienter da *Fingerprints* meist deutlich kürzer sind als die Strings selber. Es kann aber sein, dass durch eine Kollision unterschiedliche Strings den gleichen *Fingerprint* besitzen. Daher recht es nicht aus nur die *Fingerprints* zu vergleichen. Wenn die sich zwei *Fingerprints* gleich sind müssen danach alle Symbole des Strings untersucht werden.
Ein *rolling Hash* ist eine effiziente Methode um aus dem *Fingerprint* eines Substrings w den Fingerpint des nachfolgenden Substrings u zu erhalten. Die allgemeine Formel eines $rollingHashes$ ist $h(u) = h(w) - f(w_0) + f(u_m)$, wo bei m die Länge der Substrings und $f()$ die Transformation der einzelnen $chars$ ist.
Diese Bedingung erfüllen nicht alle $Hashfunktionen$.
Eine gängige Methode ist den *Fingerprint* von w als $h(w) = w_0 * p^n + w_1 * p^{n-1} + ... + w_n * p^0$, mit $p \in PRIME$, zu definieren. So lässt sich der *Fingerprint* von u wie folgt berechnen: $h(u) = (h(w) - w_0 * p^n) * p + u_n$.

1.8 Beschreibung des Algorithmus

Der in dieser Arbeit umgesetzte Algorithmus besteht aus drei Phasen. Die erste Phase generiert eine $O(\log n) - \textit{optimale LZ-Approximation}$, welche in Phase 2 und Phase 3 weiter optimiert werden.

1.8.1 Phase 1

Die erste Phase beginnt damit die Eingabe T in $\textit{Substrings}\ F_0, F_1, F_2 ... F_n$ der Länge m aufgeteilt wird, wobei gilt $m = 2^x, x \in \mathbb{N}$. Für jedes dieser Teilstücke untersuchen wir, ob ein vorheriger Substring existiert. Sollte kein vorheriger Substring existieren so spalten wir F_i in ein Präfix und ein Suffix der Länge $m/2$ auf F_{iP}, F_{iS}. Als nächstes wiederholen wir die beiden vorhergegangenen Schritte mit allen daraus erhaltenen Prä- und Suffixen. Dies tun wir so lange bis wir keine Prä- und Suffixe mehr erhalten oder die Länge der Prä- und Suffixe eins ist.

Faktor & Cherry, Chain

Um Speicher zu sparen, speichern wir nicht alle Substrings, sondern repräsentieren sie durch Verweise der Art $(\textit{Position}, \textit{Länge}, \textit{absteigendeChain}, \textit{aufsteigendeChain})$. Wir speichern aber auch nicht alle dieser Faktoren. Sollte von einem Substring nur das Präfix oder nur das Suffix einen vorherigen Substring besitzen, löschen wir den Faktor des gefundenen Substrings. Darauf hin fügen wir der entsprechenden $\textit{Chain}$ (absteigend, wenn nur das Suffix gefunden wurde, aufsteigend, wenn nur das Präfix) ein $1 - \textit{Bit}$ hinzu und der anderen ein $0 - \textit{Bit}$. Diese $\textit{Bitvektoren}$ kodieren daher alle bisher gefundenen Faktoren. Sollten sowohl Prä- als auch Suffix einen vorherigen Substring besitzen(oder beide eine Länge von eins haben) so sprechen wir nun von einer $\textit{Cherry}$. Am Ende der Phase 1 erhalten wir eine *LZ-Approximation* in Form von Cherrys und ihren assoziierten Ketten.

1.8.2 Phase 2

In der zweiten Phase versuchen wir die erhaltene $LZ - Approximation$ zu optimieren, indem wir Faktoren vereinigen. Jede Cherry besitzt zwei Chains, eine aufsteigende und eine absteigende. Für alle aufsteigenden Chains makieren wir das Suffix der Cherry als aktiv. Daraufhin überprüfen wir, ob der nächste Faktor F_i mit dem aktiven F_c vereinigt wird. Um dies zu entscheiden, testen wir, ob ein vorheriger Substring für den Substring existiert der an gleicher Stelle wie F_c startet aber zweimal die Länge von F_i hat. Solle so ein vorheriger Substring existieren dann vereinigen wir F_i mit F_c. Sollte dieser nicht existieren so markieren wir F_i als aktiv und speichern F_c ab.
Diese Schritte wiederholen wir mit dem nächsten Faktor in der Chain bis kein nächster mehr existiert.
Absteigende Chains funktionieren dementsprechend genauso.

1.8.3 Phase 3

In der letzten Phase versuchen wir erneut Faktoren zu vereinigen.
Wir testen für einen Faktor F_i (im ersten Durchlauf F_0) ob er zusammen mit dem nächsten Faktor F_{i+1} einen vorherigen Substring besitzt. Sollte dies der Fall sein vereinigen wir F_i und F_{i+1} und wieder holen den schritt mit F_{i+2}, andernfalls wieder holen wir ihn mit F_{i+1}.
Diese Methode lassen wir fünfmal über alle Faktoren laufen.
Als Ergebnis erhalten wir am Ende eine *2-optimale LZ-Approximation.*