PPM - Prediction by Partial Matching

20. September 2012

Zusammenfassung

Der PPM-Algorithmus ist eine prinzipiell relativ einfache Möglichkeit, Daten sehr stark zu komprimieren. In dieser Arbeit soll der Algorithmus und einige Verbesserungen vorgestellt, sowie hinsichtlich Laufzeit und Speicherverbrauch bewertet werden.

1 Einleitung

In dieser Arbeit soll das Kompressionsverfahren *Prediction by Partial Matching* (PPM) vorgestellt und erläutert werden, sowie ein Überblick über die Effizienz des Algorithmus' gegeben werden.

PPM ist ein Algorithmus zur verlustfreien Kompression, der sich besonders gut zur Komprimierung von natürlichsprachlichem Text eignet. Das generelle Verfahren wurde zuerst Mitte der 1980-er Jahre beschrieben (vgl. [2]) und Anfang der 1990-er Jahre implementiert, wobei zu diesem Zeitpunkt speziell der Speicherverbrauch noch kritisch war (vgl. [5]). Auf diesen Umstand wird auch in der Zusammenfassung noch eingegangen (Abschnitt 5). Zunächst soll jedoch in Abschnitt 2 die generelle Funktionsweise von PPM erläutert werden, gefolgt von Verbesserungen und Erweiterungen, die die erreichbare Kompression noch erhöhen (Abschnitt 3). Schließlich werden noch einige Details zur Implementierung von PPM, sowie daraus folgende Abschätzungen zu Laufzeit und Speicherverbrauch vorgestellt (Abschnitt 4).

Die vorliegende Arbeit orientiert sich in großen Teilen an [6], wobei die PPM* betreffenden Teile sich an [1] orientieren.

2 Funktionsweise von PPM

Bevor die eigentliche Funktionsweise von PPM beschrieben wird, ist es wichtig, sich klar zu machen, dass PPM an sich kein Kompressionsalgorithmus ist. PPM erzeugt lediglich ein Modell der Daten, d.h. eine Wahrscheinlichkeitsverteilung für die Eingabezeichen. Dieses Modell ist adaptiv[1] und kann

[1] D.h. es passt sich nach jedem gelesenen Eingabezeichen an die bisher gelesenen Daten an

zur Kompression verwendet werden. Die eigentliche Kompression geschieht allerdings durch andere Algorithmen, beispielsweise Arithmetische Codierung oder Huffman-Codierung.

Aus diesem Grund wird in diesem Kapitel nur selten auf die eigentliche Kompression eingegangen, sondern hauptsächlich auf die Codierung bzw. die Modellierung einzelner Zeichen, d.h. die Zuweisung von Wahrscheinlichkeiten zu Eingabezeichen.

Kontexte

Als Kontext eines Zeichens z wird eine feste Menge von z unmittelbar vorausgehenden Zeichen bezeichnet. So sind z.B. in dem String $s = aabacad$ einige Kontexte für das Zeichen d die Zeichenketten ca, aca oder $aabaca$, aber auch das Zeichen a sowie der leere String (der dann ein Kontext der Länge 0 wäre). Ein Kontext der Länge n ist somit ein Markov-Modell n-ter Ordnung[2].

In diesem Sinne arbeitet auch die Adaptive Arithmetische Codierung mit Kontexten, allerdings erhöht sich die Länge des Kontextes mit jedem gelesenen Zeichen um 1, da alle bisher gelesenen Zeichen als Kontext genutzt werden.

Im Gegensatz dazu arbeitet PPM mit Kontexten fester (maximaler) Länge, die typischerweise relativ kurz sind (z.B. 5 oder 6 Zeichen, vgl. [6]).

Außerdem ist die Art und Weise, wie die Kontexte zur Vorhersage von Zeichen (d.h. zur Generierung von Wahrscheinlichkeiten für Zeichen) verwendet werden, bei PPM und Arithmetischer Codierung generell verschieden: bei Arithmetischer Codierung werden lediglich Häufigkeiten für die im Kontext enthaltenen Zeichen berechnet und auf dieser Basis Wahrscheinlichkeiten berechnet. Dadurch können allerdings nur Wahrscheinlichkeiten für Zeichen berechnet werden, die im gegenwärtigen Kontext bekannt sind. Dies ist speziell bei kürzeren Kontexten unvorteilhaft, da in diesen nur relativ wenige unterschiedliche Zeichen vorkommen können. PPM dagegen „merkt sich“, welche Zeichen in welchem Kontext bereits gesehen wurden und mit welcher Häufigkeit sie aufgetreten sind.

Das bedeutet formal: wurden bisher die Zeichen $a_1a_2a_3 \dots a_{n-2}a_{n-1}$ gelesen und ist das aktuelle Zeichen a_n, so sind alle Zeichenketten $a_ia_{i+1} \dots a_{n-1}$, $i \in \{1, 2, \dots n-1\}$, sowie der leere String Kontexte des Zeichens a_n. Für einen Kontext c mit beliebiger, aber fester Länge l würde die arithmetische Codierung ihre Vorhersage für a_n basierend auf den Zeichen in c berechnen. PPM hingegen „kennt“ sämtliche vorherigen Vorkommen von c, sowie die in diesen Fällen c nachfolgenden Zeichen, d.h. PPM kennt eine Menge Z von (Zeichen, Häufigkeit)-Paaren, für die gilt $Z = \{(z_i, h_i) \mid c\ wurde\ h_i - mal\ als\ Kontext\ von\ z_i\ gesehen\}$. PPM berechnet dann die Vorhersage für

[2] für eine mathematische Betrachtung von Markov-Modellen siehe z.B. [3], Kap. 12

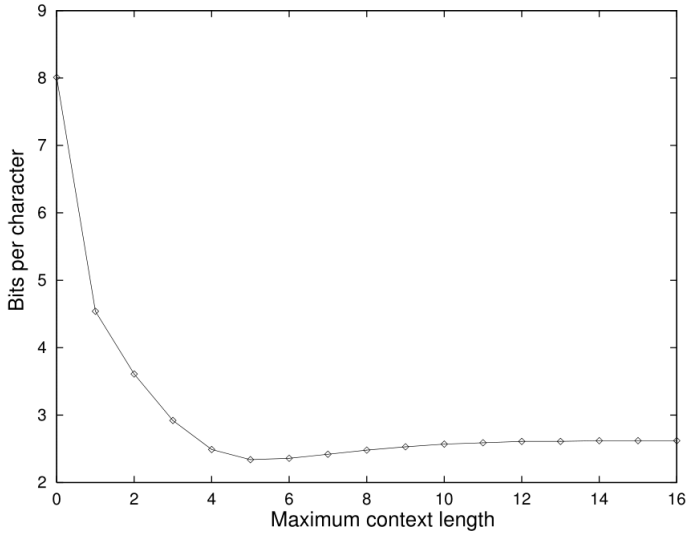

Abbildung 1: Zusammenhang zwischen Kontextlänge und Kompression, Quelle: [1]

a_n basierend auf Z.

Ein kleines Beispiel um diesen Unterschied zu verdeutlichen: angenommen, die zu codierenden Daten sind englischer Text, der gegenwärtige Kontext habe die Länge 1 und enthalte das Zeichen t, und das nächste Zeichen sei ein h.

Die Arithmetische Codierung würde in diesem Kontext für das nächste Zeichen eine sehr schlechte Schätzung abgeben, da der Kontext nur das t enthält, dem daher eine sehr hohe Wahrscheinlichkeit zugeordnet würde, während allen anderen Zeichen eine vergleichsweise kleine Wahrscheinlichkeit zugeordnet würde.

PPM hingegen würde auf Zahlen zurückgreifen, die durch die Analyse der bisher bereits codierten Zeichen ermittelt wurden. Wurde z.B. bereits eine gewisse Menge Text eingelesen, so wird die Wahrscheinlichkeit für ein h, das einem t folgt, ca. 30% betragen[3], was eine wesentlich bessere Vorhersage ist, als die durch Arithmetische Codierung gemachte.

Eine wesentliche Rolle spielt die maximale Länge der Kontexte, da dies die erreichbare Kompression maßgeblich beeinflusst. Allerdings führen bereits sehr kurze Kontexte zu einer hohen Kompression und zu lange Kontexte verschlechtern die erreichbare Kompression sogar wieder, siehe Abbildung 1.

[3]Diese Wahrscheinlichkeit gilt im Mittel für englischen Text; dies gilt laut [6], und wurde auch durch eigene Experimente bestätigt

Kontextwechsel

Falls ein Zeichen im gegenwärtigen Kontext nicht codiert werden kann, d.h. unbekannt ist, schaltet PPM auf einen kürzeren Kontext um. Dabei wird zuerst ein Escape-Symbol codiert, um den Kontextwechsel explizit, d.h. auch für den Decoder erkennbar, zu machen. Dieser Wechsel auf einen kürzeren Kontext wird so lange wiederholt, bis das Zeichen in einem Kontext gefunden wird, oder bis es auch in einem Kontext der Länge 0 nicht gefunden wurde. Im letzteren Fall wird auf einen Spezialkontext gewechselt („Kontext der Länge -1“), in dem alle Zeichen des Eingabealphabets die selbe Wahrscheinlichkeit haben (also z.B. eine Wahrscheinlichkeit von $\frac{1}{256}$ für jedes mögliche Zeichen, wenn die Eingabezeichen Bytes sind).

Nachdem ein Zeichen dann (spätestens im Kontext der Länge „-1“) gefunden wurde, wird es mit der entsprechenden Wahrscheinlichkeit codiert und anschließend werden die Wahrscheinlichkeiten für dieses Zeichen in den entsprechenden Kontexten angepasst. Ist beispielsweise die maximale Kontextlänge n, d.h. ein Zeichen x wurde im Kontext $a_1a_2a_3 \dots a_{n-1}a_n$ gesehen (d.h. die Eingabe war $\dots a_1a_2 \dots a_{n-1}a_nx$), so wird die Wahrscheinlichkeit für x in allen Kontexten $a_i \dots a_n, i \in \{1, \dots, n\}$ angepasst.

Auf diese Weise „lernt“ das Modell neue Zeichen, d.h. Zeichen die entweder noch gar nicht, oder nicht im gegenwärtigen Kontext gesehen wurden. Dies funktioniert auf die gleiche Art auch am Anfang der Daten, d.h. zu einem Zeitpunkt zu dem noch kein Kontext der Länge n vorhanden ist. Ist z.B. $n = 3$, so gibt es für das zweite Zeichen noch keinen Kontext entsprechender Länge. In diesem Fall wechselt der Algorithmus also ebenfalls einfach auf einen Kontext kürzerer Länge (wobei trotzdem ein Escape-Zeichen für jeden Kontextwechsel codiert werden muss).

Beispiel

Die bisher vorgestellte Funktionsweise soll an einem kurzen Beispiel verdeutlicht werden. Die Eingabedaten seien gegeben als String $s = aababa$ über dem Eingabealphabet $U = \{a, b\}$ und die maximale Kontextlänge sei $n = 2$. Das Escape-Symbol sei $\triangle$. Da noch keine Zeichen gelesen wurden, ist das Anfangsmodell dann

Tabelle 1: Ausgangsmodell

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : 1$	$a : \frac{1}{2}$
			$b : \frac{1}{2}$

Das erste zu codierende Zeichen x ist dann $x = a$ mit einem leeren Kontext c. Dieses Zeichen wird dann codiert als $1, 1, 1, \frac{1}{2}$, da zuerst auf

den Kontext der Länge „-1“ gewechselt (d.h. dreimal das Escape-Symbol codiert) wird, und dann das a mit der Wahrscheinlichkeit $\frac{1}{2}$ codiert wird. Anschließend wird das Zeichen a mit einem Kontext der Länge 0 dem Modell hinzugefügt:

Tabelle 2: Modell nach Schritt 1

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : \frac{1}{2}$	$a : \frac{1}{2}$
		$a : \frac{1}{2}$	$b : \frac{1}{2}$

Mit diesem neuen Modell wird dann das zweite a codiert, dessen Kontext jetzt $c = a$ ist. Da dieser Kontext jedoch nicht bekannt ist, wird das zweite a als $1, 1, \frac{1}{2}$ codiert: es ist in Kontexten der Länge 2 bzw. der Länge 1 nicht bekannt, aber (aus dem vorigen Schritt) im Kontext der Länge 0, d.h. das zweite a wird codiert als zwei Escape-Symbole, gefolgt vom eigentlichen a. Danach wird das Modell angepasst (Tabelle 3).

Die Modelle nach den weiteren Schritten sind dann in den Tabellen 4 bis 7 zu sehen.

Wie zu sehen ist „lernt“ das Modell, welche Zeichen in welchem Kontext gesehen wurden, und wie oft sie gesehen wurden. Je mehr Eingabedaten also schon gesehen wurden, desto besser wird das Modell.

3 Verbesserungen

Die erste Möglichkeit, die Kompressionsrate von PPM zu verbessern ist, die Art und Weise wie die Escape-Wahrscheinlichkeiten berechnet werden, zu verändern. Das in Kapitel 2 gezeigte Beispiel hat für die Escape-Wahrscheinlichkeiten folgende einfache Formel benutzt:

Bezeichne $c{:}x$ das Zeichen x im Kontext c, $P(c{:}x)$ die Wahrscheinlichkeit von x im Kontext c, und $H(c{:}x)$ die Häufigkeit mit der x im Kontext c bisher gesehen wurde. Dann ist $X = \{x \mid P(c{:}x) \neq 0, x \neq \triangle\}$ die Menge aller im Kontext c bekannten Zeichen. Damit gilt:

$$P(c{:}\triangle) = \frac{|X|}{|X| + \sum_{x \in X} H(x)}$$

Tabelle 3: Modell nach Schritt 2

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : \frac{1}{3}$	$a : \frac{1}{2}$
	$a \to a : \frac{1}{2}$	$a : \frac{2}{3}$	$b : \frac{1}{2}$
	$\triangle : \frac{1}{2}$		

D.h. für jedes Zeichen, das im Kontext c bekannt ist, wird die Escape-Häufigkeit in diesem Kontext um eins erhöht.

Dies ist jedoch nur eins von mehreren heuristischen Verfahren, die zwar relativ gute Kompression ermöglichen, aber keinen theoretisch fundierten Hintergrund besitzen (vgl. [6]).

Um die mit diesem Ansatz mögliche Kompression zu verbessern kann die Wahrscheinlichkeit der Escape-Symbole z.B. mit einer Poisson-Verteilung modelliert werden. Die damit erreichbaren Verbesserungen der Kompressionsrate sind allerdings gering (vgl. [6]).

Eine andere Möglichkeit bietet der Ausschluss bestimmter Zeichen beim Wechsel auf einen kürzeren Kontext. Dem liegt folgende Idee zugrunde: falls ein Zeichen x im gegenwärtigen Kontext c nicht codiert werden kann, dann kann x keins der in c bekannten Zeichen sein, d.h. $x \notin X$. Dann können aber in kürzeren Kontexten alle in diesen Kontexten bekannten Zeichen y mit $y \in X$ unberücksichtigt bleiben. Formal: Bezeichne $c_n : x$ das Zeichen x in seinem Kontext c der Länge n, d.h. die n Zeichen, die x unmittelbar vorhergehen. Sei außerdem $X_n = \{x \mid P(c_n : x) \neq 0, x \neq \triangle\}$. Dann gilt für die Wahrscheinlichkeiten im Kontext c_{n-1} nicht mehr

$$P(c_{n-1} : x) = \frac{H(c_{n-1} : x)}{|X_{n-1}| + \sum_{x\prime \in X_{n-1}} H(x\prime)}$$

sondern

$$P(c_{n-1} : x) = \frac{H(c_{n-1} : x)}{|X_{n-1}| - |X_n \cap X_{n-1}| + \sum_{x\prime \in (X_{n-1} \setminus (X_n \cap X_{n-1}))} H(x\prime)}$$

d.h. die Zeichen, die bereits im Kontext c_n bekannt waren, werden im Kontext c_{n-1} nicht mehr berücksichtigt, da sie bereits im längeren Kontext codiert worden wären. Daher erhöhen sich die Wahrscheinlichkeiten aller übrigen Zeichen im Kontext c_{n-1}, was eine bessere Codierung (d.h. Kompression) ermöglicht.

Auf die gleiche Weise können im Kontext c_{n-2} alle in den Kontexten c_n und c_{n-1} bekannten Zeichen ausgeschlossen werden. Allgemein können im Kontext c_i alle Zeichen, die in den Kontexten $c_n, c_{n-1}, \ldots, c_{i+1}$ (d.h. in allen längeren Kontexten) bekannt sind, ausgeschlossen werden.

Diese beiden Verbesserungen (Ausschluss und Verwendung von Poisson-Verteilungen für die Escape-Wahrscheinlichkeiten) lassen sich auch kombinieren. Selbst wenn die Escape-Wahrscheinlichkeiten nicht durch Poisson-Verteilungen modelliert werden (z.B. weil ein möglichst einfacher Algorithmus gewünscht ist), sollte die Verbesserung durch Ausschluss trotzdem verwendet werden, da sich die Kompressionsrate teilweise deutlich verbessern lässt.

Eine weitere Möglichkeit die Kompressionsrate zu verbessern ist die Verwendung von Kontexten unbeschränkter Länge, d.h. theoretisch unendlich

langer Kontexte. Dieses Verfahren heißt PPM* und basiert auf dem Prinzip von *deterministischen* Kontexten. Ein Kontext heißt deterministisch, wenn er genau ein Zeichen vorhersagt, d.h. in allen Fällen in denen der deterministische Kontext c bisher gesehen wurde, folgte auf c immer das gleiche Zeichen x.

PPM* versucht nun, das nächste Zeichen entweder mit dem kürzesten deterministischen Kontext zu codieren, oder, falls es keinen solchen gibt, mit dem längsten nichtdeterministischen Kontext. Die Idee dabei ist, dass die meisten Zeichen zwar mit nichtdeterministischen Kontexten codiert werden (die dann meist auch, wie bei PPM, eine Länge von ca. 5 oder 6 Zeichen haben), in einigen Fällen jedoch ein Zeichen auch einen deterministischen Kontext hat, mit dem es codiert werden kann. Diese deterministischen Kontexte ermöglichen typischerweise (vgl. [1]) eine sehr präzise Vorhersage, und damit eine sehr gute Kompression. Damit erreicht PPM* im Vergleich zu PPM eine im Schnitt gut 5% bessere Kompressionsrate (vgl. [6], [1]).

4 Implementierungsdetails

Aufgrund der schon bei kurzen Kontexten sehr großen Anzahl möglicher Kontexte ist es für effiziente Implementierungen von PPM wichtig, eine möglichst geeignete Datenstruktur zu verwenden. Die für diese Zwecke am besten geeignete Datenstruktur ist ein Trie (auch Präfixbaum genannt).

Tries

Ein Trie ist eine Baumstruktur, die Zeichenketten speichert indem in jedem Knoten (außer dem Wurzelknoten) ein Zeichen gespeichert wird. Beim Durchlaufen des Baums kann man dann die in den Knoten gespeicherten Zeichen „aufsammeln“ und verketten, und erhält so die im Trie gespeicherten Zeichenketten. Geht man dabei einen Pfad von der Wurzel bis zu einem Blattknoten, so erhält man eine Zeichenkette maximaler Länge für diesen Pfad. Geht man einen Pfad nicht bis zu einem Blattknoten, so erhält man eine Zeichenkette, die ein Präfix aller Zeichenketten ist, die man erhält, wenn man von der Wurzel durch den gelesenen Pfad bis zu einem Blattknoten geht. Abbildung 2 zeigt einen einfachen Trie, der die Wörter „Hallo“ und „Halt“ enthält.

Vom Wurzelknoten ganz links ausgehend kann man beide Wörter als Pfade bis zu den Blättern darstellen. Die Präfixe („H“, „Ha“, „Hal“ und „Hall“) sind jeweils durch Pfade darstellbar, die nicht in Blättern enden. Soll der Trie jetzt nur die vollständigen Wörter „Halt“ und „Hallo“ enthalten, so sind zusätzliche Markierungen nötig, die entweder die Präfixe als „nicht im Trie befindlich“ markieren, oder die Wörter als „im Trie befindlich“ markieren. Normalerweise wird die zweite Option gewählt und es würden dann z.B. im

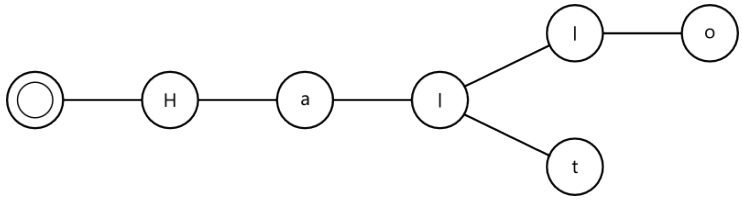

Abbildung 2: Ein einfacher Trie

obigen Trie die Knoten „o“ und „t“ markiert, um zu zeigen, dass der Trie „Halt“ und „Hallo“ enthält.

Wegen der Besonderheiten des PPM-Algorithmus' sind diese Markierungen im PPM-Trie jedoch nicht notwendig: Werden im Zeitpunkt t die Kontexte $c_n^t = a_1a_2 \dots a_{n-1}a_n$, $c_{n-1}^t = a_2a_3 \dots a_{n-1}a_n$, …, $c_1^t = a_n$ in den Trie eingefügt, dann wurden im Zeitpunkt $t-1$ die Kontexte $c_n^{t-1} = a_0a_1 \dots a_{n-1}$, $c_{n-1}^{t-1} = a_1a_2 \dots a_{n-1}$, …, $c_1^{t-1} = a_{n-1}$ eingefügt, d.h. alle Präfixe aller in t eingefügten Kontexte wurden bereits in $t-1$ in den Trie eingefügt, d.h. alle Knoten im Trie (bzw. ihre korrespondierenden Kontexte) sind gültige Wörter, weshalb die Markierungen überflüssig sind.

Zwei Punkte sind jedoch zu beachten: Erstens werden in den PPM-Trie nicht nur die eigentlichen Kontexte eingefügt, sondern immer das gegenwärtige Zeichen mit allen Kontexten. Das heißt, dass im Zeitpunkt t nicht die Zeichenketten c_n^t, c_{n-1}^t, …, c_1^t in den Trie eingefügt werden, sondern die Zeichenketten $c_n^t x^t$, $c_{n-1}^t x^t$, …, $c_1^t x^t$ und x^t, wobei x^t das aktuelle Zeichen im Zeitpunkt t ist. Es wird also das aktuelle Zeichen mit allen Kontexten (einschließlich des Länge-0-Kontexts) in den Trie eingefügt[4].

Zweitens enthalten die Knoten im PPM-Trie nicht nur ein Zeichen, sondern auch eine Häufigkeit. Diese gibt an, wie häufig das in dem Knoten enthaltene Zeichen im Kontext des Pfades von der Wurzel zu diesem Knoten gesehen wurde. Das bedeutet: sei u der Knoten in dem das Zeichen enthalten ist und sei p der Pfad vom Wurzelknoten bis zum Elternknoten von u. Dann beschreibt p einen gültigen Kontext für das Zeichen in u, und die in u gespeicherte Häufigkeit gibt an, wie oft das Zeichen in u im Kontext p gesehen wurde.

Zusätzlich können die Knoten noch je einen zusätzlichen Zeiger („Vine Pointer“, siehe [6]) enthalten. Diese Zeiger sind nicht notwendig für die korrekte Funktion des PPM-Tries, ermöglichen jedoch eine Laufzeit-Verbesserung: Für einen Knoten u der einen Kontext der Länge n beschreibt, zeigt der Vine Pointer auf den Knoten v, der den Kontext der letzten $n-1$ Zeichen von u beschreibt, d.h. wenn u den Pfad $a_1a_2 \dots a_n$ beschreibt, dann

[4]Wenn ab jetzt im Zusammenhang mit einem Trie von einem Kontext die Rede ist, so ist immer die Kombination aus Kontext (im Sinne des PPM-Algorithmus) und dem aktuellen Zeichen gemeint.

zeigt sein Vine Pointer auf den Knoten v, der den Pfad $a_2a_3 \dots a_n$ beschreibt.

PPM* verwendet eine ähnliche Datenstruktur, da allerdings die Kontextlänge nicht beschränkt ist, werden Patricia-Tries verwendet, um die Anzahl der Knoten zu reduzieren. Diese Art von Trie ermöglicht es, in einem Knoten nicht nur ein Zeichen, sondern eine ganze Zeichenkette zu speichern. Dies ist beispielhaft in Abbildung 3 dargestellt, wo die gleichen zwei Zeichenketten wie oben („Hallo“ und „Halt“) diesmal in einem Patricia-Trie gespeichert wurden, wobei Knoten mit nur einem Nachfolger mit diesem zusammengeführt wurden.

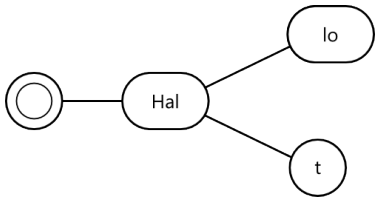

Abbildung 3: Der Trie aus Abbildung 2 als Patricia-Trie

Dabei ist anzumerken, dass die Knoten, die mehr als ein Zeichen enthalten, genau die Knoten sind, die deterministische (Teil-)Kontexte enthalten.

Speicherverbrauch und Laufzeit

Für ein Alphabet mit k Zeichen kann jeder Knoten des Tries bis zu k Kindknoten besitzen. Die Tiefe des Tries ist, falls der PPM-Algorithmus Kontexte der Länge n verwendet, beschränkt auf $n + 1$ (Kontexte der Länge 0 bis n), den Wurzelknoten nicht mit eingerechnet, da dieser keine Informationen speichert. Die Gesamtzahl der Knoten beträgt also $\mathcal{O}(k^{n+1})$, ist also exponentiell in der Kontextlänge. Der Trie benötigt dabei umso mehr Platz, je mehr unterschiedliche Kontexte gesehen wurden, d.h. je größer die Variabilität und damit je schlechter die Komprimierbarkeit der Eingabedaten ist. Gut komprimierbare Daten (d.h. Daten mit wenigen verschiedenen Kontexten) ergeben also deutlich kleinere Tries.

Da in praktischen Instanzen der Trie allerdings sehr groß werden kann, gibt es unterschiedliche Strategien, den Speicherverbrauch zu begrenzen. Die einfachste Lösung ist, bei Erreichen einer gewissen Größe das gesamte Modell zu verwerfen und ein neues zu erstellen. Dies führt allerdings dazu, dass die Kompressionsrate zu zufälligen Zeitpunkten stark einbrechen kann, da ein neues Modell angefangen wurde. Außerdem müssen Decoder und Encoder entweder den gleichen kritischen Wert für die Modellgröße verwenden, oder der Encoder muss ein Spezialsymbol codieren, wenn er das Modell verwirft. Eine bessere Möglichkeit besteht darin, die in den Knoten gespeicherten Häufigkeiten zu bestimmten Zeitpunkten anzupassen (z.B. bei

Erreichen einer gewissen Modellgröße, oder nach einer bestimmten Anzahl codierter Zeichen). Dabei kann z.B. ein bestimmter Wert von den Häufigkeiten abgezogen werden, oder alle Häufigkeiten werden halbiert. Auf diese Weise können Knoten, die eine Häufigkeit von 0 haben, aus dem Modell entfernt werden, häufiger gesehene Knoten bleiben aber erhalten. Auf diese Weise werden periodisch unwahrscheinliche Kontexte aus dem Modell entfernt. Außerdem bekommen auf diese Weise neuere Daten eine höhere Priorität als ältere, d.h. das Modell passt sich schneller an sich verändernde Eingabedaten an.

Die Laufzeit für das Nachschlagen eines Kontexts ist linear in der Tiefe des Tries (d.h. linear in der Kontextlänge n). Da allerdings ggf. mehrfach gesucht werden muss (falls ein Kontext noch unbekannt ist), ist die Laufzeit für das Suchen eines gültigen Kontexts schlechtestenfalls $\mathcal{O}(n^2)$: es müssen alle $n+1$ Kontexte (d.h. Kontexte der Längen n bis 0) gesucht werden (jeweils $\mathcal{O}(n)$).

Updates der Datenstruktur sind in Linearzeit in der Tiefe des Tries (d.h. linear in n) möglich. Wie bereits beschrieben sind alle Präfixe aller Kontexte die zu einem gegebenen Zeitpunkt t in den Baum eingefügt werden müssen bereits in den vorherigen Schritten in den Baum eingefügt worden, d.h. es müssen nur maximal $n+1$ neue Knoten eingefügt werden: das aktuelle Zeichen wird jeweils an die entsprechenden Knoten der bekannten Kontexte angehängt, insgesamt $n+1$ mal für die Kontexte der Länge 0 bis n. Falls diese Knoten bereits im Baum enthalten sind, so müssen ihre Häufigkeiten aktualisiert werden, was für jeden Knoten ebenfalls in Linearzeit möglich ist. Soweit wären die Updates allerdings in $\mathcal{O}(n^2)$, da einzelne Updates zwar in $\mathcal{O}(n)$ möglich sind, allerdings $n+1$ Updates nötig sind. Um diese Updates zu beschleunigen sind die Vine Pointer hilfreich: Wie bereits beschrieben zeigt der Vine Pointer für einen Knoten u, der den Kontext $c_n = a_1 a_2 \dots a_n$ beschreibt, auf den Knoten v, der den Kontext $c_{n-1} = a_2 a_3 \dots a_n$ beschreibt. Dementsprechend muss im Schritt t nur der Knoten für den Kontext c_n^t gesucht werden ($\mathcal{O}(n)$). Der Knoten für c_{n-1}^t ist dann in $\mathcal{O}(1)$ zu finden, indem man dem Vine Pointer des c_n^t-Knotens folgt. Für das nächste Update folgt man dann dem Vine Pointer des c_{n-1}^t-Knotens usw. Lediglich wenn keine Vine Pointer vorhanden sind, dauert das Update länger. Allerdings werden die Vine Pointer sofort eingefügt, wenn ein Knoten in den Trie eingefügt wird. Damit dauern Updates, die Knoten einfügen müssen, im Worst Case weiterhin $\mathcal{O}(n^2)$. Da der Trie aber nach einer gewissen Zeit (spätestens wenn er vollständig ist, d.h. alle möglichen Knoten enthält) stabil bleibt, amortisieren sich diese „teuren“ Einfüge-Updates und die Update-Laufzeit wird asymptotisch linear.

5 Zusammenfassung

Das in dieser Arbeit vorgestellte Kompressionsverfahren ermöglicht eine sehr hohe Kompressionsrate und auf PPM-Verfahren basierende Programme gehören zu den besten verfügbaren Kompressionsprogrammen (in Bezug auf die Größe der komprimierten Daten; vgl. [4]). Dies ist bemerkenswert, wenn man bedenkt, wie vergleichsweise einfach das dem Algorithmus zu Grunde liegende Prinzip ist. Allerdings wird die hohe Kompressionsrate mit enormem Speicherbedarf und hoher Laufzeit erkauft, was eine größere Verbreitung unwahrscheinlich macht.

Ein weiteres, auf dem PPM-Prinzip aufbauendes Kompressionsverfahren ist das Prediction-by-Partial-Precision-Matching-Verfahren (PPPM), welches für Bilder gedacht ist. Dieses Verfahren nutzt benachbarte Pixel zur Schätzung der Varianz des Vorhersagefehlers (vgl. [6]). Im Gegensatz zum hier vorgestellten PPM-Verfahren sollten Kontexte allerdings nicht sofort „gelernt“ werden, sondern erst, nachdem sie bereits einige Male gesehen wurden. Andernfalls würde die Vorhersage durch relativ seltene Pixel (bzw. Pixelumgebungen) zu stark beeinflusst. Eine etwas ausführlichere Erläuterung dieses Verfahrens findet sich in [6], weitergehende Informationen sind allerdings schwer zu finden.

Literatur

[1] John G. Cleary, W. J. Teahan, and Ian H. Witten. Unbounded Length Contexts for PPM. In *The Computer Journal*, pages 52–61, 1995. Download von `http://www.cs.ucf.edu/courses/cap5015/Cleary.pdf`, zuletzt abgerufen am 1.6.2012.

[2] John G. Cleary and Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. In *IEEE Transactions on Communications*, volume 32, pages 396–402, 1984. Download von `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1096090&tag=1`, zuletzt abgerufen am 1.6.2012.

[3] Lutz Dümbgen. *Stochastik für Informatiker*. Springer, 2003.

[4] Matt Mahoney. Large Text Compression Benchmark. `http://mattmahoney.net/dc/text.html`. zuletzt abgerufen: 2.6.2012.

[5] Mark Nelson. Arithmetic coding + statistical modeling = data compression. `http://marknelson.us/1991/02/01/arithmetic-coding-statistical-modeling-data-compression/`, Februar 1991. zuletzt abgerufen: 30.5.2012.

[6] David Salomon. *Data Compression - The Complete Reference, Fourth Edition*. Springer, 2007.

Tabelle 4: Modell nach Schritt 3

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : \frac{2}{5}$	$a : \frac{1}{2}$
$aa \to b : \frac{1}{2}$	$a \to a : \frac{1}{4}$	$a : \frac{2}{5}$	$b : \frac{1}{2}$
$\triangle : \frac{1}{2}$	$b : \frac{1}{4}$	$b : \frac{1}{5}$	
	$\triangle : \frac{2}{4}$		

Tabelle 5: Modell nach Schritt 4

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : \frac{2}{6}$	$a : \frac{1}{2}$
$aa \to b : \frac{1}{2}$	$a \to a : \frac{1}{4}$	$a : \frac{3}{6}$	$b : \frac{1}{2}$
$\triangle : \frac{1}{2}$	$b : \frac{1}{4}$	$b : \frac{1}{6}$	
$ab \to a : \frac{1}{2}$	$\triangle : \frac{2}{4}$		
$\triangle : \frac{1}{2}$	$b \to a : \frac{1}{2}$		
	$\triangle : \frac{1}{2}$		

Tabelle 6: Modell nach Schritt 5

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : \frac{2}{7}$	$a : \frac{1}{2}$
$aa \to b : \frac{1}{2}$	$a \to a : \frac{1}{5}$	$a : \frac{3}{7}$	$b : \frac{1}{2}$
$\triangle : \frac{1}{2}$	$b : \frac{2}{5}$	$b : \frac{2}{7}$	
$ab \to a : \frac{1}{2}$	$\triangle : \frac{2}{5}$		
$\triangle : \frac{1}{2}$	$b \to a : \frac{1}{2}$		
$ba \to b : \frac{1}{2}$	$\triangle : \frac{1}{2}$		
$\triangle : \frac{1}{2}$			

Tabelle 7: Modell nach Schritt 6

2	1	0	-1
$\triangle : 1$	$\triangle : 1$	$\triangle : \frac{2}{8}$	$a : \frac{1}{2}$
$aa \to b : \frac{1}{2}$	$a \to a : \frac{1}{5}$	$a : \frac{4}{8}$	$b : \frac{1}{2}$
$\triangle : \frac{1}{2}$	$b : \frac{2}{5}$	$b : \frac{2}{8}$	
$ab \to a : \frac{2}{3}$	$\triangle : \frac{2}{5}$		
$\triangle : \frac{1}{3}$	$b \to a : \frac{2}{3}$		
$ba \to b : \frac{1}{2}$	$\triangle : \frac{1}{3}$		
$\triangle : \frac{1}{2}$			