Untersuchung des Huffman-Algorithmus auf seine Effizienz

:)

Wettbewerb "Jugend Forscht" 2004

Markus Kramer (19 Jahre)

Arbeitsgemeinschaft "Jugend Forscht"
des Christian-Gymnasiums Hermannsburg
Leitung: StD Thomas Biedermann

Inhalt

1. Was ist der Huffman-Algorithmus?

Dieses Projekt ist eine kleine Untersuchung des Huffman-Algorithmus, einem Verfahren zur verlustfreien Kompression beliebiger Daten. Ich möchte von vornherein sagen, dass die hier vorgestellten Lösungen und Lösungswege möglicherweise nicht optimal sind, da sie in der Praxis bereits weiter optimiert und angepasst wurden. Sie bieten aber einen Überblick über die Stärken und Schwächen des Verfahrens.

1952 von Dr. David Huffman entwickelt, wird der Algorithmus vielseitig eingesetzt: Neben der Verwendung in Kompressionsprogrammen wie WinZip kommt er auch in Multimedia-Formaten wie MP3 vor (in letzterem spielt er eine etwas untergeordnete Rolle, da MP3 verlustbehaftet arbeitet). (Vgl. [1]).

1.1 Wie arbeitet der Algorithmus?

Wenn man einen Text am Computer schreibt, wird jedes Zeichen durch ein Byte (= 8 Bit) ausgedrückt (ASCII-Code). Das bedeutet, dass der Buchstabe e genau so viel Speicherplatz braucht wie das y. In der Praxis kommt aber das e viel häufiger vor als das y. Idee des Huffman-Algorithmus ist es, Zeichen, die häufig in einem Text vorkommen, durch kurze Bitfolgen zu ersetzten, während Zeichen, die relativ selten vorkommen, durch eine längere Bitfolge dargestellt werden. Damit verringert sich die durchschnittliche Codelänge eines Zeichens, es findet eine Kompression statt [1]. Das Problem: wenn man keine festen Grenzen (definierte Blockgröße) für einen Zeichencode hat, kann man beim Dekodieren nicht genau sagen, mit welchen Zeichen man es zu tun hat. Beispiel: wenn a = 11_b und b = 110_b und c = 011_b, ist die Bitfolge 11011_b nicht eindeutig dekodierbar, obwohl die einzelnen Codes unterschiedlich sind. Würde man ein Trennzeichen einführen, müsste man dieses mit speichern und damit die Kompressionsrate verschlechtern. Daher muss man alle Zeichen so kodieren, dass sie eindeutig auszulesen sind. Das bedeutet, dass kein Code einen Anfang haben darf, der bereits ein anderer Code ist [1]. Beispiel: mit $a = 00_b$ $b = 01_b$ $c = 10_b$ $d = 110_b$ e = 111_b lässt sich der Code $\underline{10}01\underline{110}111\underline{100}0\underline{10}_b$ eindeutig dekodieren: cbdecac

Bleibt nur noch die Frage, wie man jedem Zeichen einen eindeutig dekodierbaren, an seine Häufigkeit des Auftretens angepassten Code zuordnen kann. Die Antwort: mit einem Binärbaum. Da man in einem Binärbaum mit den Anweisungen „nach links (= 0)“ bzw. „nach rechts (= 1)“ navigieren kann, kann man jedem Blatt des Baumes eine eindeutige „Wegbeschreibung“ zuordnen. Dadurch stellen die Blätter unsere Zeichen dar, während die restlichen Knoten nur zum Verzweigen dienen und keine Information enthalten. Dieser Binärbaum sollte nicht balanciert sein, da sonst alle Codes die gleiche Länge haben würden

(was man ja vermeiden will!). Er sollte auch nicht zu stark „rechtslastig“ bzw. „linkslastig“ werden, da sonst ebenfalls die durchschnittliche Codelänge zu lang würde.

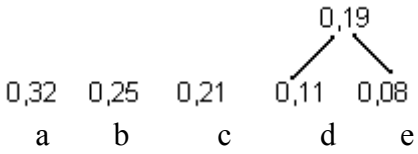
1.2 Konstruktion des Huffman Baums

Der Huffman-Baum wird von den Blättern zur Wurzel aufgebaut [2]. Man beginnt mit einem Feld von Zeichen, die nach ihrer Häufigkeit sortiert sind. Dies sind die Knoten, die später die Blätter des Baumes werden. Für das Beispiel verwende ich die Daten aus Tabelle 1.

Wahrscheinlichkeit	0,35	0,25	0,21	0,11	0,08
Zeichen	a	b	c	d	e

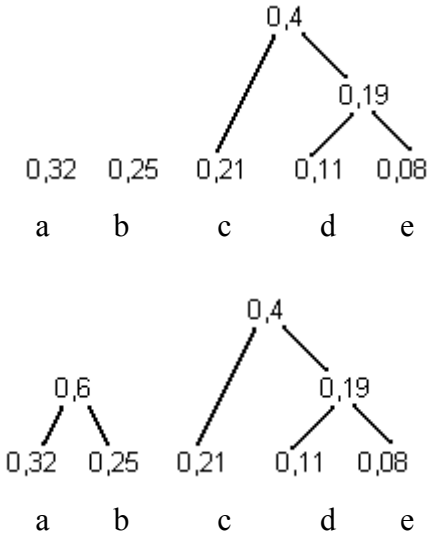
(Tab. 1)

Jetzt verbindet man die beiden Knoten mit den geringsten Wahrscheinlichkeiten in einem Vaterknoten, dessen Wahrscheinlichkeit gleich der Summe der Wahrscheinlichkeiten seinen Kinder ist [2]:

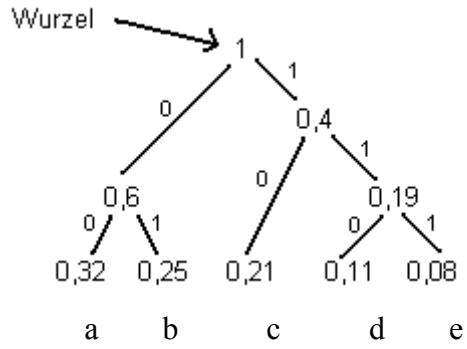


(Abb. 1)

Diesen Vorgang wiederholt man, bis nur noch ein freier Knoten übrig ist. Dieser Knoten ist dann die Wurzel des Baums.



(Abb. 2a)



(Abb. 2b)

Für links = 0 und rechts = 1 folgt daraus:

00_b	01_b	10_b	110_b	111_b
a	b	c	d	e

(Tab. 2)

Damit ist den häufig auftretenden Zeichen ein kurzer und den weniger häufig auftretenden Zeichen ein langer Code zugeordnet. Man sieht jetzt schon, dass die Effizienz des Verfahrens stark von den Eingangsdaten abhängt. Jetzt kann man anhand des Baumes bzw. der entstandenen Codewort-Tabelle jedes Zeichen des Eingangstextes durch den dazugehörigen Code ersetzen. Damit man den Text später wieder dekodieren kann, muss der dazugehörige Huffman-Baum mit gespeichert bzw. übertragen werden. Das bedeutet, dass bei der Kompression eines kurzen Text mehr Platz für den „Schlüssel“ nötig ist als für den Code selbst. Daher eignet sich das Verfahren nicht für kurze Texte oder andere kleine Datenmengen.

Durch den ASCII-Code wird jedem Zeichen ein in der Regel 8 Bit langes Bitmuster zugeordnet. In Bilddateien dagegen wird jedem Pixel je nach Einstellung der Farbtiefe ein 8, 16 oder 24 Bit langes Bitmuster zugeordnet, in dem die Helligkeit der einzelnen Grundfarben (bzw. die Graustufe) festgelegt wird. Da es für eine Kompression keine Rolle spielt, ob Zeichen oder Zahlenwerte verwendet werden (in jedem Fall handelt es sich um Bitfolgen definierter Länge), ist es sinnvoll, allgemein von Symbolen zu sprechen.

1.3 Die verschiedenen Versionen des Algorithmus

(Siehe [1])

- Im **statischen** Verfahren ist die Wahrscheinlichkeitsverteilung fest vorgegeben. Daher braucht man weder die Eingangsdaten zu untersuchen noch den Baum zu erstellen. Dieses Verfahren ist daher für zeitkritische Anwendungen geeignet.
- Im **dynamischen** Verfahren werden die Eingangsdaten auf ihre Symbole und deren Häufigkeit untersucht, ein Binärbaum mit Codetabelle erzeugt und dann die Daten kodiert.
- Das **adaptive** Verfahren kombiniert die beiden anderen Verfahren. Man geht von einem vorgegebenem Baum aus und erweitert ihn bei Bedarf.

2. Implementation - das Testprogramm

Da man den Huffman-Algorithmus immer nur in abgewandelten Formen antrifft, kommt man nicht darum herum, ihn für eine Untersuchung selbst zu implementieren. Ich habe mich dabei für das dynamische Verfahren entschieden, weil es am wenigsten an seien Einsatzort gebunden ist.

Primär habe ich versucht, das Programm so zu schreiben, dass es übersichtlich ist und man den Algorithmus leicht nachvollziehen kann. An ein paar Stellen ist das Programm auf Sonderfälle optimiert, z.B. wenn die Eingangsdaten bei 1 Byte Symbolgröße analysiert werden sollen. Eine allgemein gültige Lösung würde auch funktionieren, doch wäre sie viel langsamer und damit noch weiter von der Praxis entfernt.

Das Programm bietet die Möglichkeit, eine Quelldatei zu komprimieren bzw. eine komprimierte Datei zu dekomprimieren. Des weiteren kann man die Symbolgröße, die zur Kompression benutzt werden soll, festlegen. Damit man das Vorgehen verfolgen kann wurde eine kleine Statusanzeige und eine Ergebnisanzeige vorgesehen. In eine Protokolldatei werden die Informationen geschrieben, die das Programm zur Laufzeit ermittelt. Sie wurde auch zum Debuggen benutzt, weshalb das Programm relativ langsam arbeitet, wenn die Protokollierung aktiviert ist. Während der Testphase wurden die Statusfunktionen und die Protokolldatei weitgehend außer Kraft gesetzt, damit sie die zeitlichen Messwerte nicht verfälschen. Um die verlustfreie Kompression nachzuweisen, habe ich eine Funktion zum Vergleichen von 2 Dateien eingebaut.

2.1 Modul zur Kompression einer Datei

Lade die Quelldatei komplett in den Speicher
Unterteile die Daten in gleich große Blöcke
Erstelle eine Symbol- und Wahrscheinlichkeitstabelle der Daten
Erstelle einen Huffman-Baum aus der Symboltabelle
Ordne jedem Symbol in der Tabelle rekursiv einen Code zu
Code anhand von Symboltabelle und Eingangs-Datenblöcken aufstellen
Symboltabelle und Code in eine Datei schreiben

(Abb. 3)

Abbildung 3 zeigt den groben Ablauf der Kompressionsphase. Ein umfangreicher Schritt ist das Erstellen der Symboltabelle, was je nach Größe der Quelldaten und Symbolgröße lange dauern kann. Das Erklären dieses Teilmoduls wäre in dieser Arbeit eher unwichtig, weswegen ich mich auf die wesentlichen Teile, z.B. das Erstellen des Baumes beschränken werde.
In die komprimierte Datei wird neben einem Header sowohl die Symboltabelle als auch der Code geschrieben, damit man den Huffman-Baum später wieder rekonstruieren kann. Wichtig ist, dass dabei der gleiche Baum wie beim Komprimieren entsteht, sonst hätte man ein verfälschtes Ergebnis.

2.2. Modul zur Dekompression einer komprimierten Datei

Lade die komprimierte Datei komplett in den Speicher
Unterteile den Speicher in die Bereiche für Symboltabelle und Code
Erstelle einen Huffman-Baum aus der Symboltabelle
Dekodiere den Code mit Hilfe des Baumes
Schreibe die dekomprimierten Daten in eine Datei

(Abb. 4)

Für die Dekompression sind ähnliche Schritte wie bei der Kompression nötig. Allerdings entfällt z.B. der Schritt zur Datenanalyse, weil die Symboltabelle einfach geladen werden

kann. Den Baum muss man im Rechnerspeicher trotzdem rekonstruieren, da man durch ihn den gespeicherten Code dekodieren muss.

2.3 Modul zur Erstellung des Huffman-Baumes

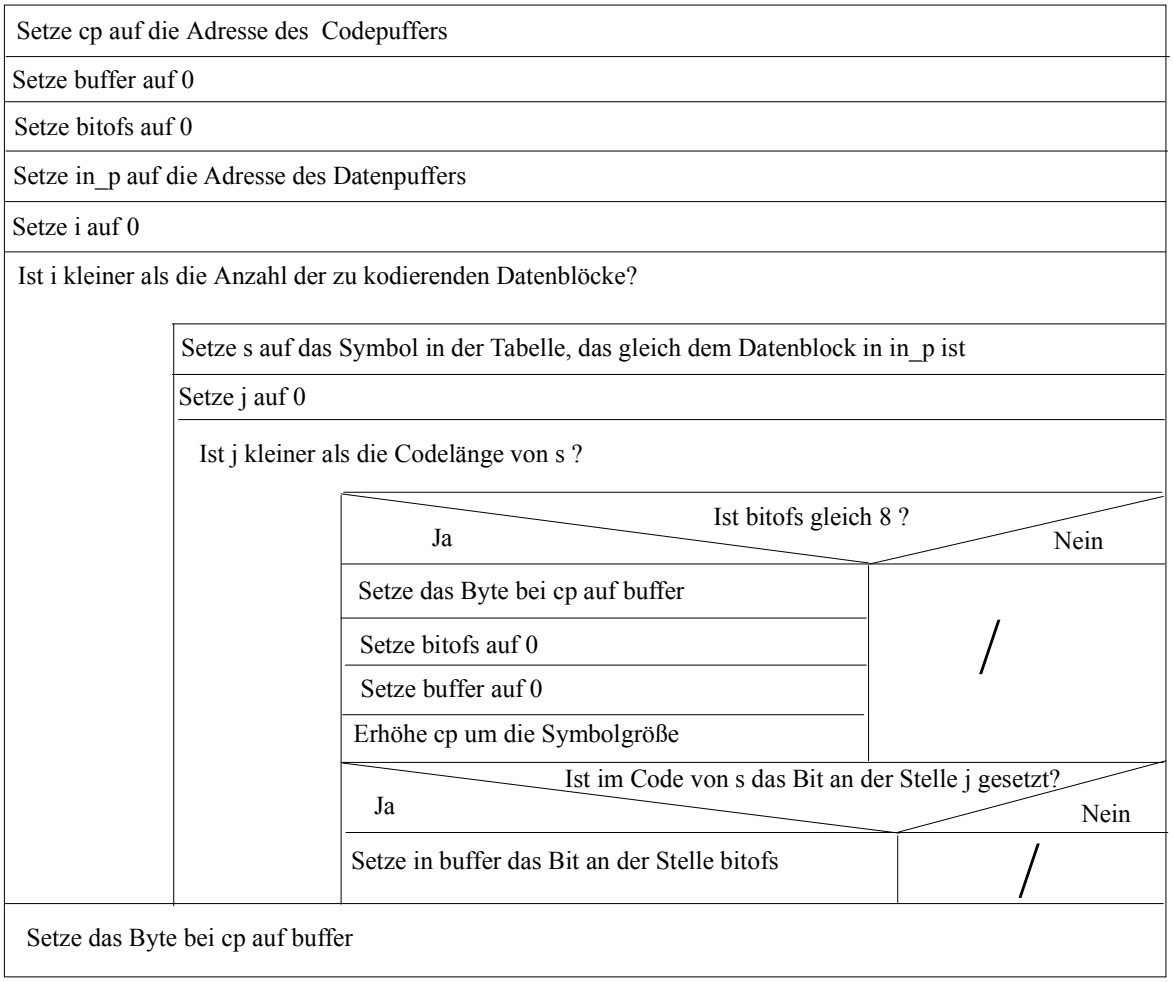
Erstelle ein Feld aus 2*n Knoten-Zeigern
Setze i auf 0 und m auf 0
Ist i kleiner als n ?
Erstelle einen neuen Knoten an der Stelle n
Kopiere die Symboldaten des Tabellenelements an der Stelle i in den Knoten an der Stelle i
Erhöhe m um die Wahrscheinlichkeit des Knotens an der Stelle i
Erhöhe i um 1
Setze i auf 0 und j auf n
Ist die Wahrscheinlichkeit des Knotens an der Stelle i ungleich m ?
Erstelle einen neuen Knoten an der Stelle j
Setze den rechten Nachfolger des neuen Knotens auf den Knoten bei i+1
Setze den linken Nachfolger des neuen Knotens auf den Knoten bei i
Wahrscheinlichkeit des neuen Knotens auf die Summe der Wahrscheinlichkeiten seiner beiden Nachfolger setzen
Erhöhe i um 2
Erhöhe j um 1
Sortiere das Zeiger-Feld von i bis j-1 nach der Wahrscheinlichkeit
Merke den Zeiger an der Stelle j-1
Lösche das Zeiger Feld
Gebe den gemerkten Zeiger zurück

(Abb. 5)

Abbildung 5 zeigt den Algorithmus, den ich zum Erstellen des Baumes verwendet habe. Wie man sieht, ist er recht einfach: es tauchen nicht einmal Verzweigungen auf. Das Modul benötigt eine fertige Symboltabelle mit n Einträgen, um den Baum zu erstellen. Diese

Einträge müssen bereits aufsteigend nach ihrer Wahrscheinlichkeit sortiert sein, damit der Baum korrekt erstellt wird. Zu bemerken ist, dass m die absolute Wahrscheinlichkeit (=Häufigkeit des Auftretens) der Symbole angibt. Gleitkommazahlen könnten Rundungsfehler hervorrufen, was das Ergebnis stark verfälschen könnte.
Dieser Algorithmus hat einen enormen Nachteil:
Je mehr Symbole in der Tabelle enthalten sind, desto öfter muss die Konstruktionsschleife durchlaufen werden. Für n Symbole braucht man n-1 Durchläufe, also muss man n-1 mal das Teilfeld sortieren. Da mit zunehmender Symbolgröße die Anzahl der Möglichen Symbole exponentiell zunimmt ($n_{\max} = 2^x$), dauert es immer länger mit diesem Algorithmus den Baum zu erstellen.
Quicksort hilft dabei nicht weiter, weil es immer das gesamte Feld sortiert, obwohl nur ein Element falsch liegt. Deshalb habe ich ein optimiertes Sortierverfahren entwickelt, jedoch mit mäßigem Erfolg: es ist zwar schneller als Quicksort, braucht aber trotzdem noch zu lange.

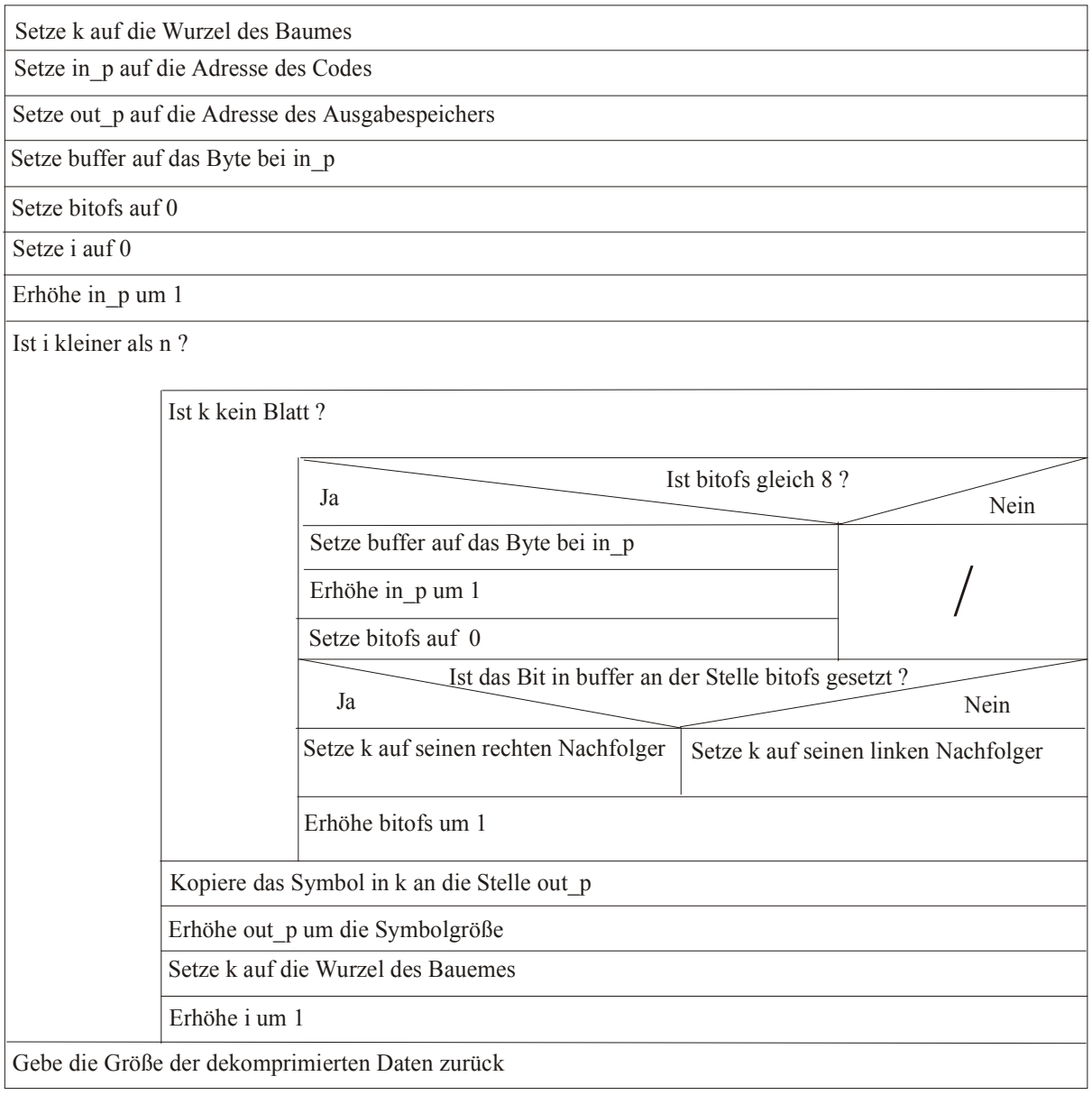
2.4 Modul zu Kodierung der Daten



(Abb. 6)

Der Algorithmus zum Kodieren der Daten kommt - wie man sieht - sogar ohne den Huffman-Baum an sich aus. Es werden nur die Symbol-Codes gebraucht, die sich aber wiederum nur mit Hilfe des Baumes erstellen lassen. Das Programm tut an dieser Stelle nichts anderes als Datenblock für Datenblock durchzugehen, und den zugehörigen Symbol-Code in einen Puffer zu schreiben. Dies geschieht linear, d.h. die Dauer des Kodierens steigt proportional zur Anzahl der Datenblöcke. Allerdings gibt es auch in diesem Modul eine Problemstelle, nämlich das Ausfindigmachen eines Symbols in der Tabelle. Je mehr Symbole man hat, desto länger dauert die Suche.

2.5 Modul zur Dekodierung der Daten

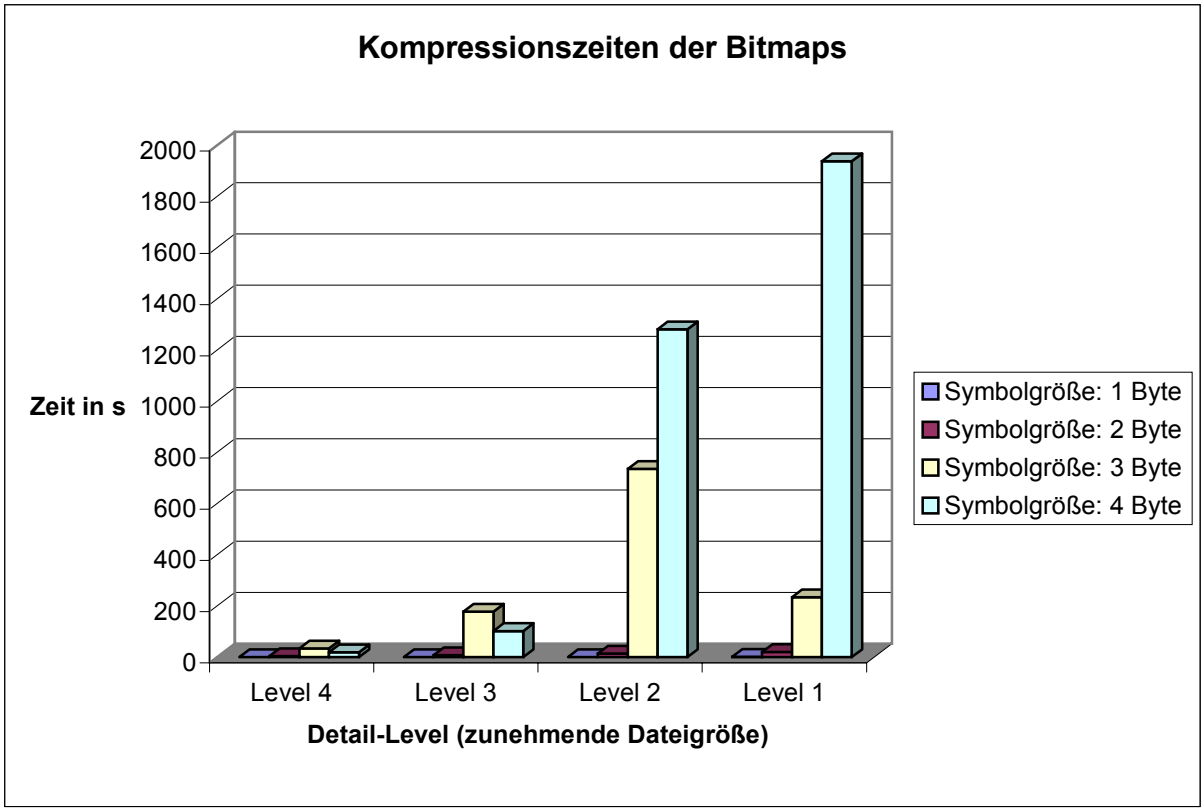


(Abb. 7)

Abbildung 7 zeigt das Verfahren, wie man anhand des Huffman-Baumes einen Code dekomprimieren kann. Dabei ist n die Anzahl der kodierten Datenblöcke. Jeder kodierte Block muss wiederhergestellt werden, das bedeutet, dass der Baum n mal durchlaufen werden muss. Dabei wird anhand des Codes entschieden, wie man zu verzweigen hat. Um den Code einfacher auswerten zu können, ist ein 1 Byte großer Puffer vorhanden, der bei Bedarf nachgeladen wird. Die Dekodierdauer ist damit abhängig von der Komplexität des Baumes und der Anzahl der kodierten Datenblöcke.

3. Tests

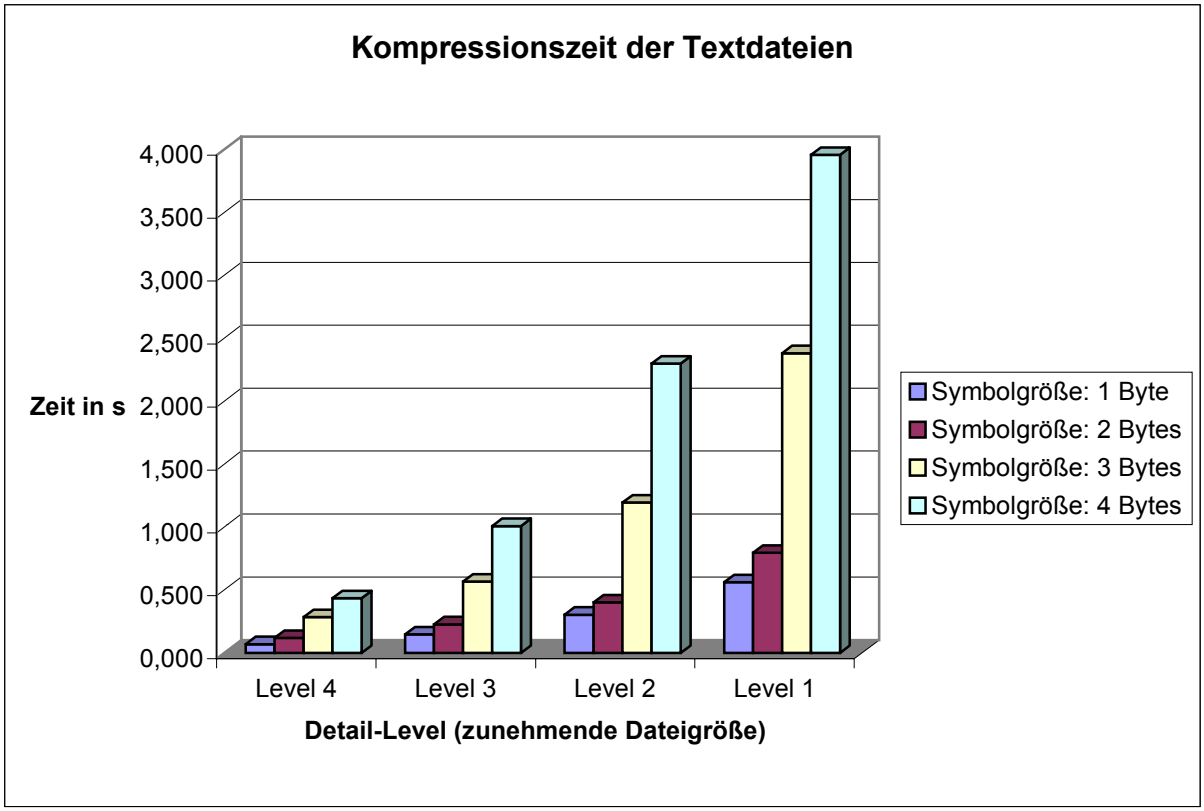
Mit dem von mir erstellten Programm habe ich nun ein paar Tests durchgeführt. Dabei wurde das Verhalten des Algorithmus bei Bild- und Textdateien untersucht. Als Bild habe ich eine 2Mb große Windows-Bitmap verwendet, da diese nicht selbst schon komprimiert sind. Als Textdatei wurde ein Teil der DirectX 7 SDK Dokumentation in Form von reinem ASCII-Code verwendet. Damit werden nur die Zeichen gespeichert, Textformatierung gibt es nicht. Von jeder Datei existieren 4 Versionen. Jede ist etwa halb so groß wie ihre Vorgängerdatei. Das macht 8 Dateien, die vom Programm komprimiert und wieder dekomprimiert werden müssen. Nach jeder Dekompression wurde mit der eingebauten Vergleichsfunktion sicher gestellt, dass kein Datenverlust auftrat, was während der Tests auch nicht vorgekommen ist. Um zu testen, wie sich der Algorithmus mit größeren Symbolen verhält, wurde jede der Dateien mit 4 verschiedenen Symbolgrößen komprimiert. Das macht dann 32 Tests, also zu viele um hier alle einzeln zu erläutern. Deshalb sind hier nur ein paar Ergebnisse beschrieben. Als Testsystem wurde ein 1GHz Athlon mit 256Mb RAM und Windows98 SE verwendet. Während der Tests liefen keinerlei zusätzliche Programme im Hintergrund und sogar die Maus wurde nicht bewegt, da sich dies als starker Störfaktor bei kurzen Kompressionszeiten erwiesen hat.



(Abb. 8)

3.1 Gemessene Kompressionszeiten im Vergleich

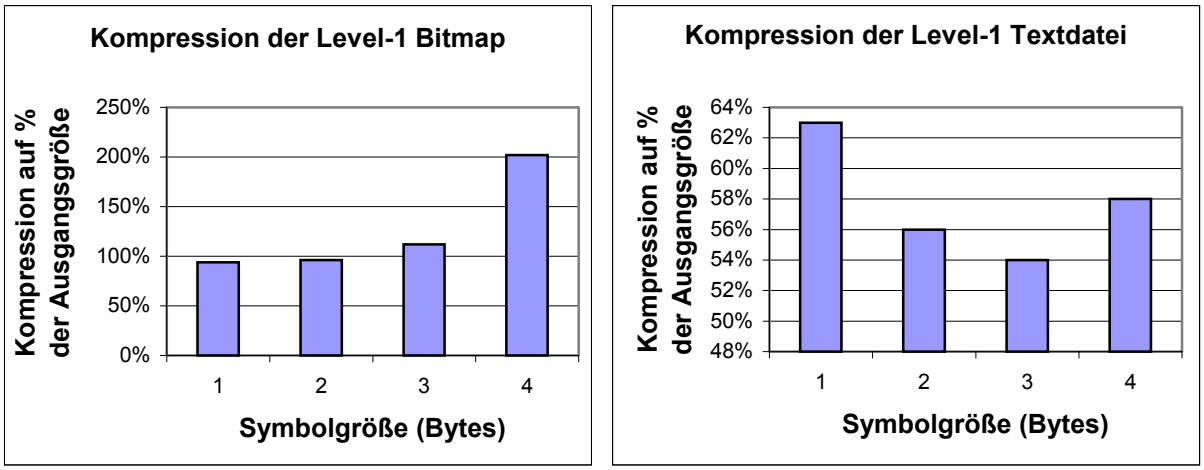
Abbildung 8 zeigt die Zeit, die zur Kompression der Bilddateien nötig war. Dabei ist die Level-1 Datei 2Mb groß, die Level-2 Datei 1Mb, usw. Das Diagramm zeigt die Zeit, die zur Kompression der einzelnen Dateien nötig war in Abhängigkeit von der Symbolgröße. Wie man deutlich sehen kann, nimmt die Dauer mit zunehmender Symbolgröße katastrophal zu. Das liegt in erster Linie daran, dass es sehr lange dauert den Huffman-Baum zu erstellen. Wie bereits gesagt, nimmt die Anzahl der maximal möglichen Symbole exponentiell zu, und da in Bilddateien meistens sehr viele verschiedene Farben gleich häufig vorkommen, ist das Ergebnis auch durchaus nachvollziehbar.



(Abb. 9)

Abbildung 9 zeigt die Zeiten, die beim Test mit den Textdateien gemessen wurden. Hierbei ist die Level-1 Datei 1Mb groß, die Level-2 Datei 500kb, usw. Im Gegensatz zu den Bilddateien ist die Zunahme der Kompressionsdauer hier nicht so extrem, aber doch noch zu stark. Da in Textdateien nur druckbare Zeichen vorkommen, nimmt die maximale Anzahl an Symbole bei größerer Symbolgröße weniger stark zu. Bei der Level-1 Datei mit Symbolgröße 1 Byte wurden z.B. nur 96 Symbole gefunden, in der Level-1 Bilddatei hingegen alle 256. Das bedeutet, dass bei der Level-1 Bilddatei mit 4 Bytes Symbolgröße $256^4 = 4\,294\,967\,296$ mögliche Symbole auftreten können, während es bei der Level-4 Textdatei nur $96^4 = 84\,934\,656$ Symbole geben kann, also etwa 2 Prozent der bei der Bilddatei möglichen Kombinationen.

3.2 gemessene Kompressionsraten im Vergleich



(Abb. 10)

Neben der Zeit, die zur Kompression nötig ist, spielt natürlich auch die Kompressionsrate eine wichtige Rolle. Abbildung 10 zeigt, wie gut die Level-1-Bitmap bzw. -Textdatei bei den verschiedenen Symbolgrößen komprimiert wurde. Während man bei der Textdatei mit allen 4 Symbolgrößen eine Kompression auf 63% bis 54% der Ausgangsgröße findet, wird die Bilddatei praktisch gar nicht komprimiert, bzw. findet sogar eine Vergrößerung der Datei statt! Wie bereits erwähnt hängt das damit zusammen, dass sehr viele Symbole mit gleicher Wahrscheinlichkeit auftreten und alle Symbole mit in die komprimierte Datei geschrieben werden müssen, damit man später den Huffman-Baum wiederherstellen kann. Das Optimum für Textdateien liegt nach den Tests bei der Symbolgröße 2-3 Bytes. In den anderen Detail-Levels lagen die Kompressionsraten von 2 und 3 Bytes Symbolgröße sehr dicht bei einander, wobei bei 2-Bytes das Minimum lag. Für die Bilddateien ist das Optimum hingegen „besser ein anderes Verfahren verwenden“, da eine zu starke Gleichverteilung der Symbol-häufigkeiten auftritt.

4. Fazit

Der dynamische Huffman-Algorithmus eignet sich gut für Daten, die eine geringe Anzahl unterschiedlicher Symbole beinhalten, wie z.B. Textdateien. Bei wenigen Symbolen kann man die Symbolgröße auf 2 bis 3 Bytes erhöhen, ohne die Kompressionsrate oder –dauer zu verschlechtern. Das Verfahren ist dagegen ungeeignet für Daten mit vielen, gleich häufigen Symbolen, da diese viel Rechenzeit und Speicherplatz brauchen. Diese Daten, z.B. Bild und Audiomaterial, werden deshalb besser mit verlustbehafteten Verfahren komprimiert. Die Komprimierung mit 1 Byte Symbolgröße ist am universellsten einsetzbar, da sie ein gutes Verhältnis von Zeit und Kompressionsrate bringt.

Damit komme ich zu dem selben Ergebnis, das auch schon vor mir festgestellt wurde, nämlich dass der Huffman-Algorithmus zwar eine gute Idee, aber ohne vorhergehende Anpassung der Quelldaten nur mäßig effizient ist.

5. Quellen

- [1] Huffman und Wavelet Kompression
(http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/Itschert-Huffman-Wavelet.pdf)
- [2] Informatik in der Oberstufe
(http://www.lehrer.uni-karlsruhe.de/~za714/material/infkurs/komp_huff.html)

6. Erläuterungen zum Anhang

Dieser Arbeit liegen einige Programme/Quelltexte bei. Sie wurden (abgesehen vom MFC-Framework) alle von mir geschrieben, ohne dass ich mich dabei an vorgegebenen Quelltexten orientiert habe. Daher sind eventuelle Fehler und schlecht optimierte Teile auf mich zurückzuführen.

Die Programme wurden erstellt mit MS-Visual-C++ 6.0 Standard.

- huffman.h	Header-Datei für die Hauptkompressor-Klasse	A1
- huffman.ccp	Implementation der Kompressor-Klasse	A2
- stab.h	Header-Datei für die Symboltabellen-Klassen	A10
- stab.ccp	Implementation der Symboltabellen-Klassen	A11

Zu den weiteren Programmen (hier habe ich auf die Angabe des Quelltextes aus Platzgründen verzichtet) gehören außerdem:

- compress.exe bzw. compress_n.exe	(Das Hauptprogramm)
- sorter.cpp	(Testprogramm für optimiertes Sortieren)