LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 134 – Abgabe zu Aufgabe A404
Wintersemester 2022/23

Arman Habibi Larissa Manalil Andrei Stoica

1 Einleitung

1.1 Überblick

Wir befinden uns seit einigen Jahrzehnten in der Big Data Era, wo wir mit einer schnell wachsenden und komplexen Datenmenge konfrontiert werden, sodass herkömmliche Methoden diese nicht oder nur schwer verarbeiten können. Datenkompression spielen daher in der heutigen Zeit eine große Rolle. Viele Computer benutzen ASCII-Kodierungum Zeichen darzustellen. Dabei steht ASCII für American Standard Code for Information Interchange, welches Symbole mit genau acht Bits representiert. Beachtet man jedoch, dass Symbole in einem Text unterschiedlich oft auftauchen, könnte man Zeichen, die öfters vorkommen, mit weniger Bits codiert als Zeichen, die selten vorkommen. Dadurch wird der Text zum Schluss platzsparend gespeichert. Dies ist das grundlegende Konzept der Huffmankodierung mit dem Ziel einer einfachen und verlustfreien Datenkompression, die von David A. Huffman im Jahr 1952 entwickelt worden ist.

1.2 Konzept

Dabei wird nach einer Häufigkeitsanalyse der Symbole im Text diese in einem binären Huffmanbaum angeordnet, welche jedem Zeichen ihre entsprechende Bitsequenz zuordnet. Die Besonderheit des Huffmanbaums liegt darin, dass die Symbole nur in dem Blättern des Baumes gespeichert werden, wodurch präfix-freie Sequenzen geschafft werden. Dies bedeutet, dass jede Bitfolge einzigartig ist und keine Bitfolge mit dem Anfang einer anderen übereinstimmt. Sobald ein Zeichen zugeordnet worden ist, beginnt direkt die Bitfolge des nächsten Zeichen. Das ermöglicht es, die Daten ohne Trennzeichen zu speichern, sodass Redundanz minimiert wird.

Als Beispiel wird der String $ABRAKADABRAB$ komprimiert, welcher die folgenden Häufigkeiten enthält. Nicht vorhandene Zeichen wurden vernachlässigt.

A	B	D	K	R
5	3	1	1	2

Daraufhin wird der präfix-freie Huffmanbaum erstellt, welcher jedem Symbol ihre Bitsequenzen zuordnet. Die Zuordnung einer Bitfolge zu einem Symbol erfolgt durch das Traversieren des Baumes von der Wurzel bis zum jeweiligen Zeichen. Muss man

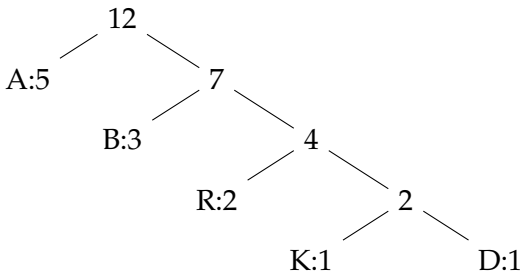

Abbildung 1: Huffmanbaum

dabei ein Knoten nach links beziehungsweise rechts, wird eine 0 beziehungsweise 1 an die kodierte Bitfolge drangehängt.

Mit der folgenden Zuweisung der Symbole kann das Beispiel nun dekodiert werden.

Symbol	**Bitsequenz**
A	0
B	10
D	1111
K	1110
R	110

ABRAKADABRAB = 01011001110011110101100 10

Ursprunglich hätte das Beispiel mit $12 \cdot 8 = 96$ Bits gespeichert werden müssen. Die Huffmankodierung reduziert das Wort auf insgesamt 1+2+3+1+4+1+4+1+2+3+1+2 = 26 Bits. Zusätzlich muss auch der Huffmanbaum abgespeichert werden, dennoch ermöglicht die Huffmankodierung bei vorallem längeren Texten und einem geeigneten, platzsparenden Format zum Speichern des Baums eine gute und einfache Datenkompression, insofern die Häufigkeiten der Symbole bekannt sind. Auf das Format zum Speichern des Baumes wird später genauer eingegangen.[3]

Im Folgenden wird eine Implementierung der Huffmankodierung in der Programmiersprache C vorgestellt, welches es ermöglicht, ASCII-Dateien zu komprimieren und zu dekodieren. Daraufhin wurde die Korrektheit dieser Implementierung anaylsiert. Zuletzt wurde die Performanz auf Kompressionsrate und durch Zeitmessungen untersucht, indem der Algorithmus mit einer nicht optimierten Version und einer Vergleichsimplementierung analysiert wurde.

2 Lösungsansatz

Der Lösungsansatz beruht darauf, dass der Benutzer eine Datei mit maximal 65536 Zeichen der erweiterten ASCII-Tabelle angibt, das von einem C Rahmenprogramm eingelesen wird und an das Huffmanprogramm zum Komprimieren gegeben wird. Das Ergebnis wird in einem speziellen Format in die vom Nutzer spezifizierte Ausgabedatei

geschrieben. Dieser kann ebenfalls komprimierte Dateien dekodieren, jedoch müssen die Daten hierfür in einem gewissen Format vorliegen, auf das später genauer eingegangen wird.

2.1 main.c

Hier befindet sich das Rahmenprogramm. Die folgenden Argumente können mittels der Funktion getopt() ausgelsesen und die zugehörigen Flags gesetzt werden. -V gibt an welche Implementierung verwendet wird, -B ob die Performanz gemessen werden soll und mit einem optionalen Argument wie oft dies geschehen soll. -o ist der Name der Ausgabedatei und -d gibt an ob das Programm encodieren oder dekodieren soll. Falls das Argument -h oder invalide Argumente gefunden werden, wird eine Hilfsnachricht ausgedruckt und das Programm mit dem Rückgabewert -1 abgebrochen. Zuletzt sollte noch der Name der Eingabedatei, ohne Argument, gefunden werden, wenn diese nicht existiert wird die Hilfe ausgegeben.

Nach den Argumenten wird die Eingabedatei ausgelsesen und in einem String, sowie die Länge des Strings, gespeichert. Nun wird je nach gesetztem -d Flag entweder die encode() oder decode() Funktion, mit dem String und dessen Länge als Parameter, aufgerufen. Zum Schluss wird das Ergebnis des Huffman Algorithmus wieder zurück in die Ausgabedatei geschrieben. Wenn irgendwo ein Fehler passiert ist und das Ergebnis nicht existiert, wird die Hilfe ausgegeben.

2.2 tree.c/heap.c

Zunächst bestand die Idee darin, den Baum als ein Array zu speichern, um die Zugriffszeiten auf Knoten zu erhöhen. Jedoch wurde klar, dass die Höhe eines Baumes vom Eingabetext und den unterschiedlichen Häufigkeiten der Zeichen stark beeinflusst wird, wodurch die Länge des Array unklar ist. Daher haben wir uns letztendlich für eine doppelt verkettete Liste entschieden. Jeder Knoten wird in einer Struktur zusammengefasst, wobei dieser ein Charakter, den dieser repräsentiert, dessen Häufigkeit und zwei Zeiger auf den linken und rechten Kindknoten speichert. Da die maximale Länge einer Datei 65536 Zeichen beträgt, reicht eine 16 Bit Variabel zum Speichern der Häufigkeit aus.

Es wird ein Min-Heap benutzt, um den Huffmanbaum darzustellen.

2.3 huffman.c

2.3.1 encode

Probleme: evtl entarteter Baum ->nicht schlimm wegen Buffer 65000 rechnung -> max 265 char mal 8 -> 2048 passt schon ungefähr

Zusätzlich muss neben den komprimierten Daten der Baum gespeichert werden, um den Text später dekodieren zu können. Um dennoch von dem Vorteil der Huffmankodierung zu gewinnen, muss ein insbesondere platzsparendes Format genutzt werden. Die Häufigkeit der einzelnen Zeichen ist für das Dekodieren irrelevant und wird daher nicht mitgespeichert. Man traversiert den Baum in Pre-Order von der Wurzel. Falls dies ein

Blattknoten ist, werden 1-Bit und darauf 8 Bits für das ASCII-Symbol gespeichert, welches das Blatt repräsentiert. Ist es kein Blattknoten, wird ein 0-Bit gespeichert. Daraufhin werden die beiden Kinder-Knoten kodiert, wobei erst das linke und dann das rechte betrachtet wird. Für den Baum in Abbildung 1 würde die Folge wie folgt aussehen:

01 01000001(A) 01 01000010(B) 01 01010010(R) 01 01000100(D) 1 01001011(K)

Beim Ausgabeformat haben wir uns dazu entschieden, den Baum genau so darzustellen, wie er im Speicher mit Bits gespeichert werden wurde. Folglich haben wir auf die Buchstaben und Leerzeichen im Beispiel verzichtet. In der nächsten Zeile steht die kodierte Bitsequenz der Quelldatei. Die Ausgabe ist dadurch nicht das effizienteste Format, dennoch sollte eine gute Mitte zwischen einem menschenlesbaren Format und der Visualisierung der genutzten Bits im Speicher getroffen werden.

2.3.2 decode

Das Dekodieren komprimierter Eingaben folgt mit der folgenden Funktion:

char *huffman_decode(size_t len, const char data[len])

Da das Ergebnis in eine Datei geschrieben wird, wurde der Rückgabeparameter der Funktionssignatur von void auf ein char-Pointer geändert. Nach Allozieren von Speicher mit der Länge von 65536 Bytes wird nach dem Leerzeichen in der zu dekodierenden Eingabe gesucht, welches den Huffmanbaum von den komprimierten Daten trennt. Ist kein Leerzeichen vorhanden oder sind andere Zeichen außer 0 und 1 im Code zu finden, wird eine passende Exception geworfen und NULL zurückgegeben.

Daraufhin wird der Baum rekursive gebildet. Man geht sequenziell durch die Bitsequenz, welche den Baum representiert. Bei einem Null-Bit erstellt man einen NULL-Knoten und ruft rekursive auf seinen linken, dann rechten Kindknoten die Methode auf. Bei einem Eins-Bit erstellt man einen Blattknoten, der das ASCII-Zeichen beinhaltet, die die nächsten 8 Bits der Bitsequenz repräsentieren.

Letztendlich wird die komprimierte Datei dekodiert. Dabei fängt man bei der Wurzel an und geht bei einer 0 beziehungsweise 1 einen Knoten nach links beziehungsweise rechts. Falls es sich um einen Blattknoten handelt, wird das entsprechende Symbol in den Buffer geschrieben und man beginnt wieder an der Wurzel. Zeigt der Zeiger zum Schluss nicht auf die Wurzel des Baums, wird wieder eine Exception ausgegeben. Ansonsten werden alle Zeiger gefreet und der Buffer zurückgegeben.

3 Korrektheit

Es wird auf die Korrektheit des Huffman Algorithmus eingegangen, da er verlustfreie Komprimierung ermöglicht und somit Genauigkeit als Thema unangemessen wäre. Um das Problem des optimalen und präfixfreien Codes zu lösen sind zunächst einige Definitionen notwendig.

3.1 Kodierungstheorie

Das *Quellalphabet* der Länge n entspricht der Anzahl an verschiedenen Zeichen die in der Quelldatei vorkommen. In diesem Fall entspricht das maximale Quellalphabet der Menge an ASCII Zeichen. Jedem Zeichen wird ein *Gewicht* w zugeordnet, welches hier der absoluten Häufigkeit des Zeichens in der Quelldatei entspricht. Die Menge der Gewichte ist somit definiert als $W = \langle w_i > 0 \mid 0 \leq i < n \rangle$. Des Weiteren wird auch ein *Codealphabet* definiert, welches die Binäre Menge $\{0, 1\}$ ist, mit dem die Quelldatei komprimiert wird. Der *Code* ordnet jedem Zeichen i des Quelllphabets ein Codewort aus dem Codealphabet der Länge l_i zu und ist definiert als die Menge $T = \langle l_i > 0 \mid 0 \leq i < n \rangle$.

Die Kraft-McMillan-Ungleichung gilt als notwendige Bedingung für präfixfreie Codes.

$$\mathcal{K}(T) = \sum_{i=0}^{n-1} 2^{-l_i} \leq 1$$

Codes die diese Bedingung erfüllen werden *zulässig* genannt, dies bedeutet dass eine Belegung an Codewörten existiert, die präfixfrei sind. Eine Menge an Codewörten ist für einen gegebenen Code *gültig*, insofern die Längen $\langle l_i \rangle$ übereinstimmen und kein Codewort präfix eines Anderen ist. Es ist somit möglich zu sagen, dass eine präfixfreier Code determiniert wurde, sobald ein zulässiger Code gefunden wurde, ohne die spezifischen Codewörter zu betrachten, da man weiß, dass eine gültige Belegung existiert.

Zuletzt sind noch die *Kosten* C eines Codes zu definieren, welches der Länge der kodierten Nachricht entspricht.

$$C(W, T) = C(\langle w_i \rangle, \langle l_i \rangle) = \sum_{i=0}^{n-1} w_i \cdot l_i$$

Ein Code $T = \langle l_i > 0 \mid 0 \leq i < n \rangle$ gilt als optimal, falls für jeden anderen zulässigen Code T' der Länge n gilt: $C(W, T) \leq C(W, T')$. Dies bedeutet auch, dass mehrere optimale Codes für eine gegebene Quelldatei existieren können. Für einen optimalen Code muss die Kraft-McMillan-Summe 1 ergeben, da bei kleineren Werten immer ein Codewort gekürzt werden kann, während größere Werte von Anfang an nicht zulässig sind.

3.2 Optimalität

Um zu zeigen, dass der Huffman Algorithmus tatsächlich optimal ist, muss zunächst die *geschwister Eigenschaft* betrachtet werden. Diese besagt, dass ein optimaler Code existiert, bei dem die zwei Zeichen mit dem geringsten Gewicht geschwister sind, also einen Elternknoten im zugehörigen Binärbaum teilen.

Angenommen es gibt einen optimalen Code $T = \langle l_i \rangle$ mit einer Häufigkeitsverteilung $W = \langle w_i \rangle$ auf n Zeichen. Da jeder nicht-Blatt Knoten zwei Kinder haben muss, existieren zwei Blatt Knoten mit der höchsten Tiefe $L = max_i l_i$. Ohne Beschränkung der Allgemeinheit sind diese Knoten w_{n-1} und w_{n-2} diejenigen mit den geringsten Gewicht. Falls

sie die selbe Tiefe haben, kann der Baum nun umstrukturiert werden, um die beiden Knoten zu Geschwistern zu machen, ohne die Kosten des Codes zu verändern.

Der Beweis der Optimalität erfolgt durch Induktion über der Anzahl an Zeichen n. [1] Als Induktionsbasis gilt n = 2, da es hier nur den optimalen Code $\langle l_i \rangle = \langle 1, 1 \rangle$ geben kann, nähmlich die beiden Codewörter 0 und 1. Es wird angenommen, dass die Aussage für $(n-1)$ Zeichen valide ist um sie für n Zeichen zu beweisen. Man nimmt an es gibt die Gewichte $W = \langle w_0, ..., w_{n-2}, w_{n-1} \rangle$, sodass für die Gewichte ohne Beschränkung der Allgemeinheit gilt $w_o \geq w_1 \geq ... \geq w_{n-1} > 0$.

Ausgehend von der geschwister Eigenschaft kann man nun sagen, dass es einen Optimalen Code $T = \langle l_0, ..., l_{n-2}, l_{n-1} \rangle$ gibt, wo die Gewichte w_{n-1} und w_{n-2} geschwister sind. Es gilt nun zu Beweisen, dass der durch Huffman hergeleitete Code H dieselben Kosten hat.

Wenn man nun die beiden Knoten w_{n-2} und w_{n-1} entfernt und dem Elternknoten die Summe der Gewichte zuordnet, erhält man den Code T' mit $(n-1)$ Zeichen und den Gewichten $W' = \langle w_0, ..., w_{n-3}, w_{n-2} + w_{n-1} \rangle$. Das Selbe geschieht nun für den Huffman Code H' mit den gleichen Gewichten $W'_H = \langle w_0, ..., w_{n-3}, w_{n-2} + w_{n-1} \rangle$ per Definition vom Huffman Algorithmus. Es gilt per Induktionsannahme:

$$C(W'_H, H') = C(W', T')$$

Ebenfalls gilt,

$$C(W', T') = C(W, T) - (w_{n-2} + w_{n-1})$$

da die Summe der Knoten w_{n-2} und w_{n-1} nun auf der Tiefe $L - 1$ liegt. Das selbe gilt wiederum für den Huffman Code:

$$C(W'_H, H') = C(W_H, H) - (w_{n-2} + w_{n-1})$$

Diese drei Gleichungen resultieren in folgendem Beweis.

$$C(W_H, H) = C(W'_H, H') + w_{n-2} + w_{n-1} = C(W', T') + w_{n-2} + w_{n-1} = C(W, T)$$

3.3 Komprimierungseffektivität

Claude Shannon definierte, 1948, die *Entropie* einer Menge an Gewichten folgendermaßen. [5]

$$\mathcal{H}(W) = -\sum_{i=0}^{n-1} w_i \cdot log_2 \frac{w_i}{m}$$

Sie lässt sich aufteilen auf die Summe der Gewichte $m = \sum_{i=0}^{n-1} w_i$ und den entropischen Kosten (in Bits) $-log_2(w_i/m) = log_2(m/w_i)$ einer Instanz eines Zeichens, welches mit einer Wahrscheinlichkeit von (w_i/m) auftritt. Die Entropie $\mathcal{H}(W)$ gibt die geringste mögliche Länge in Bits an, die der komprimierte Code haben kann, wenn man die Häufigkeitsverteilung mit einbezieht. Der *relative Effektivitätsverlust* $\mathcal{E}$ eines zulässigen Codes T und der zugehörigen Gewichte W ist durch folgende Gleichung zu berechnen.

$$\mathcal{E}(W, T) = \frac{C(W, T) - \mathcal{H}(W)}{\mathcal{H}(W)}$$

Ein relativer Effektivitätsverlust von 0 bedeutet, dass die Zeichen mit ihren entropischen Kosten komprimiert werden, wohingegend bei größeren Werten, diese nicht erreicht werden.

Die Huffmankodierung kommt, in den meisten Situationen, den entropischen Kosten sehr nahe und resultiert daher in Effektivitätsverlusten, die wenige Prozente über dem Optimalfall liegen. Jedoch gibt es auch einige Fälle, wo der Huffman Code einen großen relativen Effektivitätsverlust hat, insbesonderen, wenn w_0 im Vergleich zur Summe der anderen Zeichen sehr groß wird und w_0/m nahe an die 1 kommt. Zum Beispiel hat die Gewichtmenge $W = \langle 96, 1, 1, 1, 1\rangle$ eine Entropie von $\mathcal{H}(W) = 32.2$ Bits , ein zugehöriger optimaler Code $T = \langle 1, 3, 3, 3, 3\rangle$ mit Kosten $C(W, T) = 108$ Bits, und somit einen relativen Effektivitätsverlust von $\mathcal{E}(W, T) = 235\%$. Ein optimaler Code bedeutet somit nicht unbedingt optimale Komprimierung.

Robert Gallager [2] zeigte folgende Beziehung, zwischen einem optimalen Code $T = \langle l_i\rangle$ und der Gewichtsmenge $W = \langle w_i\rangle$ mit der Summe $\sum_{i=0}^{n-1} w_i = m$, mit der sich eine obere Schranke für den relativen Effektivitätsverlust berechnen lässt.

$$C(W, T) - \mathcal{H}(W) \leq \begin{cases} w_0 + 0.086 \cdot m, \text{ wenn } w_0 < \mathrm{m}/2 \\ w_0, \text{ wenn } w_0 \geq \mathrm{m}/2 \end{cases}$$

4 Performanzanalyse

5 Zusammenfassung und Ausblick

Insgesamt ist der Huffman Algorithmus, aufgrund seiner Optimalität bei symbolbasierter Komprimierung, immer noch sehr relevant, da er in den meisten Fällen sehr knapp an die, durch die Entropie gegebene, perfekte Komprimierung kommt. Dennoch ist es möglich die Effizienz noch weiter zu erhöhen, indem man sich nicht nur auf symbolbasierte Algorithmen beschränkt.

Es wäre somit möglich, zum Beispiel bei von Menschen geschriebenen Fließtext, einen Huffman Baum für die einzelnen, durch Leerzeichen gentrennte, Wörter zu erstellen. Dies könnte bei sehr großen Quelldateien, im Vergleich zum symbolbasierten Ansatz, effizienter sein.

Eine spezielle Anpassung auf das gespeicherte Format sollte ebenfalls zu Optimierungen führen. Man denke sich, zum Beispiel, PNG für Bild- oder FLAC für Audiodateien, da hier mehr Daten, als nur Buchstaben, abgespeichert werden müssen.

Zu der Implementierung kann man sagen, dass diese noch deutlich optimierbar ist. Zu einem ist das Verfahren mit den Min-Heap sehr anschaulich, aber dafür ineffizient, da es auch Lösungswege gibt, die dies in $\mathcal{O}(n)$ erreichen [6], unter anderem auch in-place [4]. Das Speicherformat könnte ebenfalls weiter optimiert werden, indem man, anstatt diese Menschenleslich zu strukturieren, direkt die Binärdaten reinschreibt. Zudem wäre es auch denkbar die Laufzeit, durch Parallelisierung und der Verwendung von mehreren Threads, zu reduzieren.

Zusammenfassend können wir sagen, das wir, durch das Projekt, zum optimierten Programmieren angeregt und somit ein hoher Lerneffekt erreicht wurde.

Literatur

[1] UC Berkeley. Proof of optimality of huffman coding. `https://inst.eecs.berkeley.edu/~cs170/fa20/assets/notes/huffman.pdf`. Accessed: 2023-02-04.

[2] R. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[3] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[4] Alistair Moffat and Jyrki Katajainen. In-place calculation of minimum-redundancy codes. In Selim G. Akl, Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, pages 393–402, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[5] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[6] Jan van Leeuwen. On the construction of huffman trees. In *International Colloquium on Automata, Languages and Programming*, 1976.