# TSBK08 - Datakompression

*Entropi-estimering, Huffman-kodning och LZW-kodning*

**Fredrik Wallström**
**19920808-2694**
**fredrik.wallstrom.1992@gmail.com**

Handledare : Harald Nautsch
Examinator : Harald Nautsch

LINKÖPINGS UNIVERSITET

Linköpings universitet
SE–581 83 Linköping
013-28 10 00 , www.liu.se

## Upphovsrätt



## Copyright



Fredrik Wallström
© 19920808-2694
fredrik.wallstrom.1992@gmail.com

**Sammanfattning**

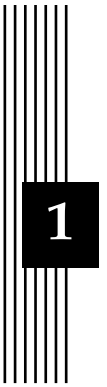
Entropi-estimering, Huffman-koding och LZW-kodning implementerades i ett C++ program i kursen TSBK08 - Datakompression. Att komprimera filer i dagens samhälle är viktigt för att spara plats på lagringsenheten, det känns således viktigt att lära sig hur komprimering med vissa metoder fungerar och hur bra de presterar. Huffman-kodning ger en optimal komprimerad fil och passar bra att komprimera de flesta filtyperna med. LZW-kodning är en bra komprimeringsmetod som passar bra för att komprimera stora filer med symbolsekvenser som upprepas ofta. Huffman-kodning har sina nackdelar jämfört med LZW-kodning, vi behöver spara statistik över hur källan ser ut. LZW-kodning har också sina nackdelar jämfört med Huffman-kodning, den ger inte en optimal kod utan behöver ha en stor källa med många upprepade symboler för att bli effektiv.

# Författarens tack

Framför ett tack till Harald Nautsch för en väl genomförd kurs, TSBK08 - Datakompression.

# Innehåll

# Figurer



# 1 Introduktion

## 1.1 Motivation

Att komprimera filer är väldigt användbart i dagens samhälle. Dels för att spara filer i ett mindre format, vilket leder till de tar upp mindre plats på lagringsenheten, och för att det går snabbare att överföra informationen mellan olika användare. Det finns flertalet metoder för att komprimera filer och olika metoder passar bättre till vissa filtyper. Den här rapporten kommer att undersöka två olika komprimeringsmetoder, Huffman-kodning och Lempel-Ziv-Welch-kodning (LZW).

## 1.2 Mål

Målet med denna rapport är att undersöka två olika komprimeringsmetoder, Huffman-kodning och LZW-kodning. Hur fungerar dessa komprimeringsmetoder och vilka olika fördelar och nackdelar finns, samt hur bra de presterar vid komprimering av olika filtyper.

## 1.3 Frågeställningar

De frågeställningar som denna rapport kommer att behandla listas nedan.

1. Hur stor plats på lagringsenheten sparar vi då en fil komprimeras med Huffman-kodning respektive LZW-kodning.
2. Hur nära kommer vi den optimala takten (entropin) när vi komprimerar en fil med Huffman-kodning respektive LZW-kodning.
3. Vilka filtyper passar bäst att komprimera med Huffman-kodning respektive LZW-kodning.

## 1.4 Avgränsningar

Det finns flera olika metoder för komprimering av filer. Denna rapport undersöker endast Huffman-kodning och LZW-kodning och tar ej hänsyns till övriga existerande metoder för

komprimering. Tilläggas bör att vid referering till Huffman-kodning menas endast statisk Huffman-kodning, det vill säga då källans frekvenser är kända på förhand och uppskattas inte samtidigt som filen kodas.

# 2 Teori

Nedan presenteras hur Entropi-estimering, Huffman-kodning och LZW-kodning fungerar.

### Entropi-estimering

Det går att mäta hur bra komprimerad en sekvens är, detta gör man genom mäta takten på koden. Takten på koden ges av [2].

$$Takt\ R = \frac{genomsnittligt\ antal\ bitar\ per\ kodord}{genomsnittligt\ antal\ symboler\ per\ kodord} \quad Bitar\ per\ symbol$$

Det ger alltså ett tal som säger i snitt hur många bitar vi har kodat varje symbol med.

Entropin för en källa är en teoretisk lägre gräns för hur många bitar vi behöver för att koda varje symbol. Takten kan alltså inte vara under denna gräns. Entropin av en källa ges av [2].

$$Entropy\ H(X) = -\sum p(X) \log p(X)$$

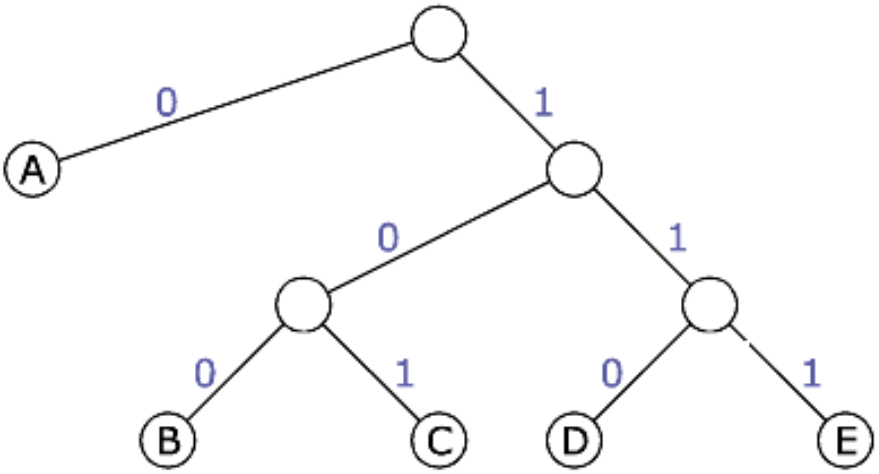
Där p(X) är sannolikheten för given symbol och log är bas 2 logaritm. Vi summerar över alla symboler i källan.

### Huffman-kodning

Huffman-kodning är en metod för att komprimera filer till ett mindre format. Metoden utvecklades av den dåvarande studenten David A. Huffman år 1952. Idén med metoden är att tilldela varje symbol som ska kodas en sekvens av olika bitar, 0 eller 1, exempel:

Symbol A kodas till 10101010
Symbol B kodas till 1000000
Symbol C kodas till 11111111

I metoden kommer de symboler som förekommer flest gånger i källan, det vill säga den symbol som har högst sannolikhet, att tilldelas minst antal bitar. Det betyder att de symboler




Figur 2.1: Ett exempel på ett Huffman-träd.

som förekommer minst gånger i källan kommer tilldelas mest antal bitar. Exempel kan symbolen D kodas med bitarna 10 och symbolen E kodas med bitarna 1001010101. Det betyder att symbolen D förekommer oftare i sekvensen än symbolen E. Alltså kommer sekvensen att komprimeras på bästa möjliga vis. Redan här inses att med Huffman-kodning behövs information om hur källan ser ut. Det vill säga, man behöver veta sannolikheterna för de olika symbolerna i källan. Nedan beskrivs mer ingående hur komprimering och avkomprimering fungerar i Huffman-kodning [1].

**Komprimering**

Att komprimera en källa med Huffman-kodning förutsätter att det finns en given sannolikhetstabell för källan. Det finns alltså en lista som mappar en symbol mot dess sannolikhet, vi kallar här varje symbol och dess sannolikhet för en nod. Metoden fungerar då enligt följande:

1. Ta ut de två symbolerna (noderna) som har lägst sannolikhet i listan.
2. Addera dessa noders sannolikheter och lägg till den som en ny nod i listan, denna nod får ingen symbol. Resterande noder behålls.
3. Börja sedan om från ett om det finns fler än en stycken noder kvar i listan.

Med den här metoden byggs en trädstruktur upp över hur de olika symbolerna kan tilldelas olika bitar. Lövnoderna i trädet representerar de olika symbolerna. Roten av trädet representeras av en nod med sannolikhet 1 eftersom summan av alla sannolikheter i en sekvens alltid summeras till 1. För att tilldela en sekvens av bitar till alla symboler traverserar man igenom hela trädet och bildar en lista som mappar respektive symbol sin egna unika sekvens av bitar. Här är det förutbestämt att varje gång man går vänster i trädet adderar man en 0:a till sekvensen av bitar och varje gång man går höger adderar man en 1:a till sekvensen, eller vice versa. Figur 2.1 visar hur ett Huffman-träd kan se ut [1].

När mappningen mellan symbol och sekvens av bitar existerar kan den givna källan komprimeras. Detta görs genom att läsa av en symbol i i taget ifrån källan och sedan matcha den symbolen med symbolerna som finns i listan som mappar symboler mot en sekvens av bitar. Då ges en sekvens av bitar som adderas till den resulterande bitsekvensen. Denna procedur upprepas tills det inte finns fler symboler att läsa av i sekvensen som ska komprimeras.

### Avkomprimering

För att avkomprimera en Huffman-kod behövs samma sannolikhetslista som användes vid komprimeringen av en källa. Förutsatt att denna lista är känd, kan ett Huffman-träd byggas upp på samma vis som vid komprimeringen. Med Huffman-trädet kan man sedan traversera ifrån roten och ner beroende på vilken bit, 0 eller 1, man stöter på i den sekvens som ska avkodas. När man sedan kommer till en lövnod betyder det att en symbol kan avkodas, denna symbol adderas sedan till den resulterande sekvensen. Proceduren startar sedan om ifrån rotnoden och avslutas då det inte längre finns bitar kvar att läsa av i bitsekvensen [1].

## LZW-kodning

LZW-kodning är en metod för att komprimera filer till ett mindre format. Metoden utvecklades av Abraham Lempel, Jacob Ziv, och Terry Welch år 1984. Metoden är en utveckling av den tidigare metoden LZ78 metoden som utvecklades av Abraham Lempel och Jacob Ziv år 1978. Idén med LZW komprimering är att skapa en ordbok för olika symboler och kombinationer av symboler. Med kombinationer av symboler menas symboler som förekommer efter varandra i källan och inte alla möjliga kombinationer av symboler ifrån källan. Nedan beskrivs mer ingående hur komprimering och avkomprimering fungerar i LZW-kodning [3].

### Komprimering

För att komprimera en given källa med LZW följs dessa steg.

1. Initialisera ordboken så den innehåller alla symboler av längd 1, lämligtvis alla möjliga bytes, 0-255
2. Hitta den längsta möjliga strängen, W, av symboler i ordboken som matchar den aktuella strängen som avläses.
3. Addera motsvarande index i ordboken för W till resultatet.
4. Addera W plus nästa symbol ifrån källan som avläses till ordboken.
5. Gå till steg två så länge som det finns kvar symboler att läsa i källan.

På detta vis erhålls en sträng med index som för sig motsvarar en eller flera symboler [3].

### Avkomprimering

Avkomprimeringen för en sekvens med LZW-kodning fungerar som följer. Först intialiseras samma ordbok som användes vid komprimeringen, med undantag att man nu mappar index mot en sträng istället för tvärtom. Man läser sedan av ett index ifrån sekvensen och slår upp det indexet i ordlistan, motsvarande symbol/symboler adderas till resultatet. Under tiden byggs ordlistan upp på samma sätt som den byggdes upp under komprimeringen. Det är dock lite annorlunda under avkomprimeringen, när man avläser en symbol, W, vet vi att den sista symbolen i W är ett prefix till nästa symbol, X, som ska vara i ordboken, så vi måste därför vänta tills vi läser av nästa symbol, Y, innan vi kan addera X till ordboken. Avkomprimeringen ligger på så vis ett steg efter med att bygga upp ordboken [3].

# 3 Metod

Nedan gås igenom hur entropi-estimering, Huffman-kodning och LZW-kodning har genomförts.

### Entropi estimering

För att göra en entropi estimering av givna källor, som angavs i laborationens uppgift, skrevs ett program i C++. Programmet utgick ifrån en funktion som beräknade entropin för en given källa enligt formeln som beskrivs i teori avsnittet. Funktionen tar in en sannolikhetstabell som mappar varje symbol mot dess sannolikhet att den förekommer i källan. Denna sannolikhetstabell behövde alltså genereras innan entropin kunde beräknas.

För att generera en sannolikhetstabell estimerades sannolikheterna för varje symbol i källan. Detta gjordes genom att först spara hela filen i en lista, för att sedan iterera över den och addera 1 i en ny lista för varje påträffad symbol i källan. Denna nya lista mappar således symboler mot hur många gånger den förekommer i källan, det vill säga, en frekvenstabell. Denna frekvenstabell användes sedan för att beräkna sannolikhetstabellen genom att dividera varje frekvens med källans storlek. Den önskade entropin kunde sedan beräknas.

I laborationen har även betingade entropier beräknats för källorna. En betingad entropi betyder att vi beräknar entropi då vi känner till hur källan ser ut K steg tillbaka i tiden. De betingade entropierna som har beräknats är då K = 1 och då K = 2, detta utrycks genom följande, H(Xi | Xi-1) och H(Xi | Xi-1, Xi-2). För att beräkna de betingade entropierna krävdes en sannolikhetstabell för par av symboler och för tripplar av symboler. Dessa sannolikhetstabeller togs fram på motsvarande sätt som för en symbol, med undantag att två respektive tre symboler lästes in i taget. Dessa symboler adderades sedan genom att konkatenera dess bit representation, detta skapar en unik ny symbol. Vi har då sannolikhetstabeller för par och tripplar av symboler. Sedan beräknades den gemensamma entopin, H(X1, X2) och H(X1, X2, X3) med samma entropi funktion som användes för singlar av symboler. Efter det användes kedjeregeln, H(X1, X2, ..., Xn) = H(X1) + H(X2 | X1) + ... + H(Xn | X1, ..., Xn-1), för att beräkna den betingade entropin.

### Huffman-kodning

För att komprimera godtyckliga filer med Huffman-kodning skrevs ett program i C++. Grundmetoden som användes var den som beskrevs i teori avsnittet. Först lästet filen in och varje symbol sparades i en lista. Sedan byggdes en frekvenstabell upp på samma vis som gjordes för att estimera entropin. Utifrån denna frekvenstabell byggdes sedan ett Huffman-träd, som beskrivs i teoriavsnittet. Detta Huffman-träd bestod av noder, structar i C++, som innehåller symbolen, frekvensen för symbolen och två stycken pekare till andra noder. För att bygga upp Huffman-trädet används en prioritetskö, det behövs för att noderna i kön behöver vara sortera med den lägsta frekvensen först. Sedan tas de två första noderna ut ur kön och deras frekvens summeras och en ny nod bildas med de två gamla noderna som barn till den nya noden. Den nya noden läggs sedan in i prioritetskön igen. Detta upprepas till bara en nod återstår i prioritetskön, denna nod är rotnoden i Huffman-trädet.

När Huffman-trädet är uppbyggt byggdes sedan en så kallad kodningskarta upp, det vill säga en lista som mappar symboler mot sekvenser av bitar. Detta gjordes genom att rekursivt traversera genom hela Huffman-trädet och varje gång vi gick till höger barn adderades en 1:a och varje gång vi gick till vänster barn adderades en 0:a.

Med denna kodningskarta kodades sedan källan. Detta gjordes genom att återigen öppna upp filen och gå igenom symbol för symbol. För varje symbol i filen adderades motsvarande sekvens av bitar i kodningskartan till en resulterande sträng. Denna sträng skrevs sedan till en ny fil och källan har därmed kodats med Huffman-kodning.

En viktig detalj att nämna här är att den frekvenstabell som genererades i början av Huffman-kodningen sparades i den kodade filen, som en så kallad header. Detta är ett måste eftersom vi behöver veta frekvenserna för varje symbol när vi ska avkomprimera den kodade Huffman-filen.

För att avkomprimera den komprimerade Huffman-filen läses först headern av och frekvenstabellen erhålls. Med denna frekvenstabell byggs sedan samma Huffman-träd upp som vid komprimering av källan. Sedan avkomprimeras Huffman-filen genom att börja från rotnoden och sedan läsa av bit för bit. Vid en 0:a går vi ner i vänstra subträdet och vid en 1:a går vi ner i högra subträdet. När vi stöter på en lövnod adderar vi den motsvarande symbolen till resultatsträngen och börjar om ifrån rotnoden. På så vis byggs orginalfilen upp igen.

### LZW-kodning

För att komprimera godtyckliga filer med LZW-kodning skrevs ett program i C++. Grundmetoden som användes utgicks ifrån den som beskrevs i teori avsnittet. Först lästes den givna filen in och varje symbol sparades i en lista. Sedan byggdes ordboken upp, bestående av alla möjliga bytes, 0-255. Sedan itererades det över varje symbol den sparade filen, om den symbolen redan existerade i ordboken så lästes en nästa symbol in och konkateneras till den föregående symbolen. Tillexempel om A läses in och den symbolen redan existerar i ordboken läses nästa symbol B in och konkateneras till A, det ger resultatsymbolen AB. Sedan återupprepas proceduren och det undersöks om symbolen AB finns i ordboken eller inte.

Låt oss nu säga att symbolen AB inte finns i ordboken. Det betyder att vi ska lägga till AB i ordboken och skriva A:s motsvarande index i ordboken till resultatfilen. A:s index skrivs med X antal bitar där X är bas 2 logaritmen av nuvarande storleken på ordboken. Detta upprepas till det inte finns fler symboler kvar att läsa vilket resulterar i en komprimerad LZW-fil.

För att avkomprimera en komprimerad LZW-fil läses först LZW-filen in och varje bit sparades i en lista. Sedan byggdes samma motsvarande ordbok upp som vid komprimering av filen med undantaget att denna gång mappas index mot bytes. Sedan läser vi av X antal bitar ifrån listan där X är bas 2 logaritmen av nuvarande storleken på ordboken. Dessa byter transformeras till motsvarande siffra, med denna siffra slås motsvarande symbol upp i ordboken. Under tiden i avkomprimeringen byggs ordboken upp så att den blir identisk med den ord-

bok som användes vid komprimeringen av källan. Hur detta görs förklaras i teori avsnittet. På detta vis erhålls en identisk källa som innan komprimering.

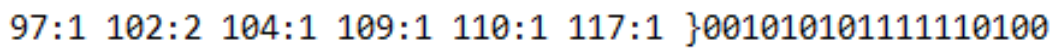
# 4 Resultat

Resultatet av Entropi-estimeringen, Huffman-kodningen och LZW-kodningen presenteras i figur 4.1. För varje fil, som presenteras i raderna, ser vi entropin, H(Xi), H(Xi | Xi-1) och H(Xi | Xi-1, Xi-2). Vi ser även vilken takt (rate) vi kan koda med Huffman-kodning och LZW-kodning. Vi ser även hur mycket plats vi sparar med Huffman-kodning och LZW-kodning för respektive fil.

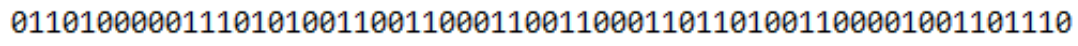
Att komprimera en fil med Huffman-kodning lyckades bra. Låt oss säga att vi har en fil med innehållet "huffman". Den komprimerade Huffman-filen ser vi i figur 4.2. I filen ser vi

| | H(Xi) | H(Xi\|Xi-1) | H(Xi\|Xi-1, Xi-2) | Rate - Huffman | Rate - LZW | Space saving - Huffman | Space saving - LZW |
|---|---|---|---|---|---|---|---|
| alice29.txt | 4.56768 | 3.41859 | 2.4851 | 4.61244 | 3.27403 | 41.9887% | 59.0746% |
| asyoulik.txt | 4.80812 | 3.41762 | 2.53807 | 4.84465 | 3.51403 | 39.0329% | 56.0746% |
| cp.html | 5.22914 | 3.46713 | 1.73784 | 5.26716 | 3.67845 | 31.8315% | 54.0193% |
| fields.c | 5.0077 | 2.94981 | 1.46953 | 5.0409 | 3.5591 | 31.8318% | 55.5112% |
| grammar.lsp | 4.63227 | 2.80336 | 1.28392 | 4.66434 | 3.88928 | 29.5216% | 51.384% |
| kennedy.xls | 3.57347 | 2.7773 | 1.71153 | 3.59337 | 2.49216 | 54.8923% | 68.848% |
| lcet10.txt | 4.66912 | 3.497 | 2.61228 | 4.69712 | 3.05877 | 41.1349% | 61.7653% |
| plrabn12.txt | 4.53136 | 3.36604 | 2.71689 | 4.57534 | 3.29493 | 42.6833% | 58.8134% |
| ptt5 | 1.21018 | 0.823655 | 0.705193 | 1.66091 | 0.969744 | 79.0348% | 87.8782% |
| sum | 5.32899 | 3.29792 | 1.93048 | 5.36504 | 4.2045 | 28.596% | 47.4438% |
| xargs.1 | 4.89843 | 3.1934 | 1.54757 | 4.92382 | 4.42039 | 27.7354% | 44.7451% |
| bible.txt | 4.34275 | 3.2691 | 2.47861 | 4.38495 | 2.45466 | 45.1745% | 69.3167% |
| E.coli | 1.99982 | 1.98142 | 1.96323 | 2 | 2.12143 | 74.999% | 73.4822% |
| world192.txt | 4.99831 | 3.66047 | 2.77064 | 5.04114 | 2.57203 | 36.954% | 67.8496% |

Figur 4.1: Resultat av entropi-estimeringen, Huffman-kodningen och LZW-kodningen.




Figur 4.2: En Huffman-fil som komprimerat en fil med innehållet "huffman".




Figur 4.3: En LZW-fil som komprimerat en fil med innehållet "huffman".

den bitsekvens som filen har komprimerats till. Vi ser även den så kallade headern, den representerar frekvenstabellen som även nämndes i teori- och metodavsnittet. Vi ser tillexempel först 97:1, vilket representerar ASCII-tecken 97 och efter kolonet ser vi många 97 det finns i filen, det vill säga 1. ASCII-tecken 97 motsvarar symbolen a. Vi ser sedan att det finns två stycken 102, vilket motsvarar symbolen f. Resultatet av detta exempel på Huffman-kodning blir att den har komprimerats från 8*7 = 56 bitar till 36*8 + 18 = 306 bitar. Detta är såklart ingen komprimering överhuvudtaget, filen har blivit större, varför det blivit så diskuteras senare. Vid avkomprimering av Huffman-filen läses frekvenstabellen in först. för att sedan bygga upp Huffman-trädet och läsa av bit för bit i Huffman-filen. Detta fungerar utmärkt.

Att komprimera en fil med LZW-kodning lyckades bra. Låt oss säga att vi har en fil med innehållet "huffman". Den komprimerade LZW-filen ser vi i figur 4.3. I filen ser vi den bitsekvens som filen har komprimerats till. Resultatet av detta exempel på LZW-kodning blir att den har komprimerats från 8*7 = 56 bitar till 62 bitar. Detta är inte heller någon komprimering, utan filen har blivit större, varför det blivit så diskturas senare. Att avkomprimera LZW-filen görs genom att läsa av bitsekvensen, vilket fungerar utmärkt.

# 5 Diskussion

Nedan diskuteras implementationen, fördelar och nackdelar med Entropi-estimeringen, Huffman-kodningen samt LZW-kodningen.

## 5.1 Entropi-estimering

Entropin var till en början lite problematisk att implementera. Det uppstod problem när sannolikheterna för par av symboler och tripplar av symboler skulle estimeras. Första som gjordes var att addera två de två motsvarande ASCII-tecknen, insåg sedan att symbolerna AB kommer mappas till samma symbol BA och detta är förstås inte tanken. Detta löstes sedan genom att konkatenera deras bitsekvensen som förklaras i metodavsnittet. Efter det kunde de betingade entropierna beräknas på korrekt och enkelt sätt genom att använda kedjeregeln.

## 5.2 Huffman-kodning

Huffman-kodningen implementerades smärtfritt. Fördelen med Huffman-kodning är att den alltid generar en optimal kod, detta eftersom vi alltid tar och adderar de två minsta sannolikheterna när vi bygger upp Huffman-trädet. Det gör att de symboler som förekommer ofta i källan kommer att kodas med få bitar. Huffman-kodning kommer heller inte att ha några onödiga extra 0:or framför kodordet som kan förekomma i LZW-koden. Detta leder till en optimal kod, vilket också visas i resultatet då vi kommer väldigt nära den optimala entropin. En annan fördel med Huffman-kodning är att den är väldigt enkel att implementera, när Huffman-trädet är uppbyggt är det bara läsa av symbol för symbol i källan och och slå upp den symbolen i Huffman-trädet, vilket ger den motsvarande bitsekvensen. Den stora nackdelen med Huffman-kodning är att vi måste spara statistik för hur källan ser ut. I mitt fall har jag sparat hela frekvenstabellen i en så kallad header, detta gör att den komprimerade filen således blir större än vad den egentligen behöver vara. I exemplet som presenterades i resultatavsnittet ser vi att den komprimerade filen blir större än orginalfilen. Detta beror alltså på att vi måste addera frekvenstabellen till den komprimerade filen för att kunna avkomprimera den. Utan frekvenstabellen skulle filen endast kodas med 18 bitar, vilket kan jämföras med LZW-kodningen som gav hela 62 bitar.

## 5.3 LZW-kodning

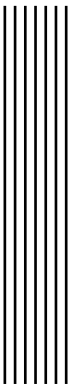
LZW-kodningen implementerades utan problem. Enda problemet var att först visste jag inte hur många bitar man skulle koda symbolens index med, så jag begränsade min ordbok till $2^{20}$, det vill säga jag skrev varje index med 20 bitar. Blev ordboken full, så slutade jag lägga in fler symboler och körde på de symboler jag hade i ordboken. Denna metod presterade bra ändå tyckte jag, speciellt på stora filer, eftersom vid stora filer fylls ändå ordboken. Vid små filer däremot blev inte resultatet lika tillfredställande, en fil blev större vid komprimering än vad den var innan komprimering tillexempel. Föreläsningen om LZW-kodning var sedan givande. Ändrade till att skriva antalet bitar till att bero på nuvarande storlek på ordboken istället, detta betyder att i värsta skriver vi med optimalt antal bitar. Däremot kan vi ibland skriva fortfarande skriva onödiga bitar eftersom om vi får en träff med en symbol som motsvarar index 3, som motsvarar bitarna 11, och storleken på ordboken för tillfället är 1500 så kommer vi skriva med 11 bitar, det vill säga 00000000011, vilket inte är optimalt såklart. Den dynamiska tillämpningen att enbart skriva med 2 bitar, 11, har inte implementerats och vet inte om det går heller. Problemet är att vi inte vet hur många bitar vi ska läsa av när vi avkomprimerar den LZW-kodade filen. Fördelen med LZW-kodning är att vi inte behöver veta någonting om hur den tidigare källan ser ut. Det vill säga vi behöver tillexempel inte veta frekvenserna för varje symbol som vi behöver veta vid Huffman-kodning. Detta leder till att den LZW-kodade filen inte behöver bli onödigt stor. LZW-kodning kodar i allmänhet annars väldigt bra. Nackdelen med LZW-kodning är som nämnts ovan att den inte presterar en optimal kod. Vi kan tillexempel inte koda 11 med bara bitarna 11 utan vi måste koda den beroende på hur stor ordboken är för tillfället. LZW-kodning bygger på upprepningar av symboler och är således inte så användbar om det inte förekommer mycket upprepningar i källan. LZW-kodning är således användbar för stora källor, då finns chans för fler upprepningar i källan.

# 6 Slutsats

Både Huffman-kodning och LZW-kodning är bra metoder för komprimering av data. De båda har sina fördelar respektive nackdelar. Nästan i alla fall kan vi komprimera filen mer om vi använder LZW-kodning istället för Huffman-kodning. Det beror på att med Huffman-kodning måste vi sparar frekvenstabellen, vilket tar extra plats i den kodade filen. Den främsta anledningen är dock att i min implementation av Huffman-kodningen kodar vi bara en symbol i taget och tar inte hänsyn till källans minne av symboler, vilket vi gör i LZW-kodning.

I varje fall med Huffman-kodning kommer vi väldigt nära entropin (optimala takten) som kan presteras då vi kodar med en symbol i taget. Så Huffman-koden presterar relativt sätt väldigt bra. I LZW-koden är takten ungefär lika med den betingade entropin då vi vet en symbol tillbaka i tiden. Detta betyder alltså att vi ungefär kodar med dubbla symboler i LZW-koden.

Utifrån detta experiment är det svårt att dra slutsats om vilka filtyper som passar bättre till de olika kodningsmetoderna. Huffman-kodningen passar bra till de flesta typer av filer eftersom takten är nära entropin i alla fall av filtyper. LZW-kodning drar fördelar av återupprepning av symboler, vilket betyder att den presterar bäst på filer med upprepade symboler. Det kan vara både txt-, gif- eller pngfiler.



# Litteratur


[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest och Clifford Stein. *Introduction to algorithms, 3rd edition*. 3. utg. MIT Press, 2009, s. 428–437.

[2] Harald Nautsch. *TSBK08 - Data compression*. 2018. URL: http://www.icg.isy.liu.se/en/courses/tsbk08/lect2.pdf.

[3] Terry Welch. "A Technique for High-Performance Data Compression". I: *Computer* 17.6 (1984), s. 8–19. DOI: 10.1109/mc.1984.1659158.