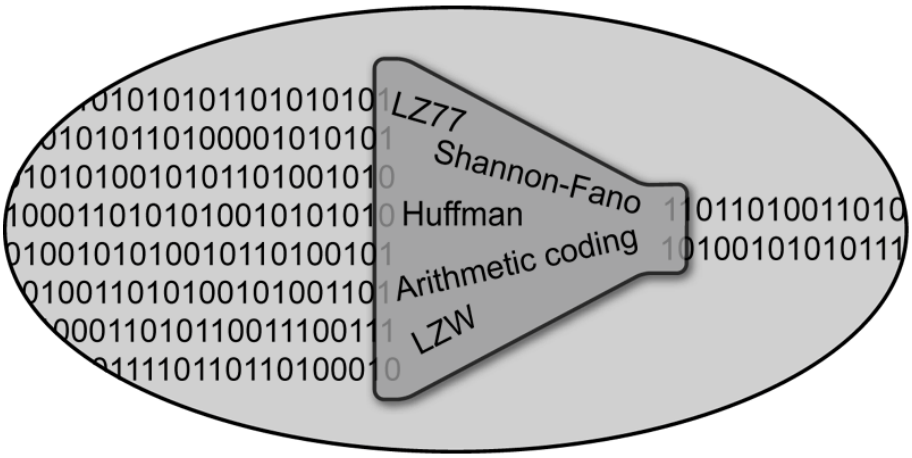
Komprimering

Implementering, og test af lossless komprimering



Adam Hayeem frosch@ruc.dk, Torbjørn Nielsen torbjni@ruc.dk

Morten Poulsen hartlev@ruc.dk, Jonas Hansen joha@ruc.dk

og Mads Danquah danquah@ruc.dk

Vejleder, Anders Madsen am@ruc.dk

2. Semester Roskilde Universitets Center

29. december 2001

Indhold

1 Abstract

Formålet med dette projekt er, at finde den komprimeringsalgoritme som er mest effektiv til at komprimere tekstfiler, hvor effektiviteten måles udfra komprimeringstiden i forhold til komprimeringsgraden. Dette har vi gjort ved at kigge på fire komprimeringsalgoritmer: LZW, LZ77, Huffman og Shannon-Fano.

Vi har herefter opstillet en analytisk model, hvor vi teoretisk og eksperimentelt har bestemt de fire algoritmers komprimeringsgrad og hastighed. Denne model viste sig at være mest anvendelig på mindre filer, som fylder mindre end en megabyte. I rapporten blev der fundet følgende konklussion:

Vi kan konkludere, at det var muligt at opstille en model over komprimeringstiden kontra komprimeringsgraden, selvom denne aldrig blev optimal p.g.a. mange fejlfaktorer.

Vi kan udfra modellen konkludere, at LZW er den mest effektive algoritme til komprimering af tekstfiler. Dette kan modellen dog kun sige er gældende for små filer.

2 Forord

Denne rapport er primært skrevet til studerende, som befinder sig på den videnskabelige basisuddannelse og har interesse for datalogi. Det antages, at læseren har et rimeligt kendskab til computere og begreberne som bruges indenfor området.

I rapporten gennemgås først den grundlæggende teori for hvordan komprimering foregår. Herefter beskrives principperne bag nogle af de klassisk brugte komprimeringsalgoritmer. Det næste afsnit beskriver hvordan gruppen selv har valgt at implementere algoritmerne, og de mest betydningsfulde aspekter fremhæves. Dette afsnit henvender sig mest til den meget datalogi interesserede læser, da det bliver ret teknisk.

I model afsnittet opstilles en model, som beskriver de valgte komprimeringsalgoritmers effektivitet ud fra et teoretisk og eksperimentalt synspunkt. Herunder gennemgås desuden en del teori, som er nødvendig for, at modellen kan opstilles.

Til sidst udføres et eksperiment på komprimeringsalgoritmerne, der skal klarlægge modellens mangler og fordele. Dette bliver uddybbet i diskussionen og leder frem til rapportens konklusion.

Vi vil gerne have lejlighed til at takke vores vejleder Anders Madsen for at have stillet kritiske spørgsmål under processen, så den røde tråd ikke forsvandt helt ud af projektet.

3 Indledning

I starten af projektperioden ville vi skrive om kryptering. Vi indså dog hurtigt at det eksperimentelle enten ville blive for avanceret for os eller for simpelt til at dække semesterbindingen.

Vi ændrede derfor planen og besluttede os for at skrive om komprimering. Komprimering virkede umiddelbart lettere at gå til, da der her var tale om to let bestemmelige aspekter, der kunne sammenlignes, nemlig hastighed og komprimeringsgrad.

Man kan komprimere på to måder, med tab og uden tab. Tabsløs komprimering er en komprimeringsmetode, hvor den komprimerede fil ikke mister nogen information, den bliver bare mindre. Komprimering med tab er derimod når den komprimeret fil mister information. Dette kan f.eks. være en sang der bliver komprimeret ned til mp3 format. Denne fil mister lyde og støj, som det menneskelige øre ikke kan opfange, og bliver derved mindre. Ved komprimering med tab, modsat tabsløs komprimering, er det ikke muligt at gendanne den originale fil ved dekomprimering.

Vi valgte at arbejde med tabsløs komprimering, da komprimering med tab er meget specifikt, og algoritmerne ofte kun kan komprimere en form for filer, som f.eks. lydfiler. Derudover virkede tabsløs komprimering mere overkommeligt.

Til selve projektet skulle der findes en række komprimeringsalgoritmer som vi selv kunne implementere og eksperimentere med. Der findes mange forskellige algoritmer, nogle er ens bare med små modifikationer, der tuner dem til en bestemt type filer eller et bestemt sprog. Vi valgte Huffman, Shannon-Fano, LZ77 og LZW, der er nogle af de mere populære algoritmer. LZ77 bliver f.eks. brugt i pakkeprogrammet Zip. Vi valgte selv at implementere algoritmerne, da vi på den måde kunne sikre os, at de havde samme arbejdsforhold.

Vores mål er at opstille en model, der viser forholdet mellem komprimeringstiden og komprimeringsgraden. Modellen vil bygge på en komprimering af “The Canterbury Corpus” som er nogle specielt udvalgte filer til afprøvning af nyudviklede komprimeringsalgoritmer. Filerne repræsenterer gennemsnitsfiler af en vis type. Ved hjælp af de eksperimentelle data vil vi kunne opstille nogle grafer for de forskellige algoritmer.

Ved at finde den teoretiske komprimeringsgrad og den teoretiske køretid for algoritmerne vil vi kunne finpudse vores model. Teorien vil også være et godt redskab til at forudsige, om vores implementeringer af algoritmerne indeholder for store fejlkilder, der forringer vores model.

Grunden til at vi har valgt at bruge to metoder til at opbygge vores model er, at begge metoder har deres mangler og fordele. F.eks. er det teoretisk umuligt at forudsige komprimeringsgraden for LZ77 og LZW, mens et eksperimentalt forsøg ofte er påvirket af mange fejlkilder. Det er derfor meningen at de to metoder skal kunne supplere hinanden, så den bedst mulige model kan opstilles.

Målsætningen med modellen er, at den skal fortælle os, hvilken algoritme der er bedst til at komprimere en tilfældig udvalgt fil med henblik på komprimeringsgraden i forhold til tidsforbruget og filstørrelsen. Modellen vil vi herefter teste ved at komprimere en række tilfældigt udvalgte filer og sammenligne disse resultater med forudsigelserne fra vores

model. Denne målsætning førte os frem til følgende problemformulering:

Vi vil, udfra en række repræsentative filer, opstille en model over specifikke tabsløse komprimeringsalgoritmers effektivitet, hvor effektivitet er målt ved hjælp af algoritmens komprimeringstid og komprimeringsgrad.
Ved hjælp af modellen vil vi prøve at finde den mest effektive (optimale) komprimeringsalgoritme

4 Teori

Informationsteorien er et meget anvendeligt redskab i komprimering, da den relativt simpelt kan forudsige, hvor meget en given fil kan komprimeres. Dette vil vi anvende i vores model til at forudsige den teoretiske komprimeringsgrad for de enkelte algoritmer.

4.1 Introduktion

Claude E. Shannon udgav i 1948 bogen "*A Mathematical Theory of Communication*". I denne beskriver han mulighederne for at nedsætte størrelsen af en information, som skal sendes fra et sted til et andet. Denne idé om en informations størrelse kaldes for *informationsteorien* og Shannon baserede den på princippet om, at budskaber, som forekommer hyppigt, har mindre informationsværdi end budskaber, som forekommer sjældent. Dette kan påvises intuitivt i vores dagligdag, hvor hyppige ord som "er" og "i" indeholder langt mindre information end de mindre brugte ord "fe" og "ø". Det ville dog være en overdrivelse at påstå dette kan overføres til al information i det virkelige liv, da der naturligvis findes undtagelser.

Til påvisning af et budskabs informationsværdi valgte Shannon at indføre en ny størrelse, som han kaldte for "selv-information" (*self-information*). Hvis $P(A)$ beskriver sandsynligheden for at finde et givent tegn A i en besked, så er *selv-informationen* af A givet ved ligningen:

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b(P(A)) \qquad ; b \in N \tag{1}$$

Da funktionen $-log(x)$ aftager, når x vokser, må $i(A)$ blive forminsket, når $P(A)$ vokser. Dette stemmer fuldstændig overens med Shannons princip, da *selv-information* af et tegn på denne måde bliver mindre, jo hyppigere tegnet er brugt.

I ligningen kan logaritmens base b være et vilkårligt naturligt tal, men det er praktisk at sætte $b = 2$, da $i(A)$ på denne måde beregnes i bit, som er computerens beregningsenhed. Desværre kan de fleste lommeregnere ikke tage log_2, så det er nødvendigt at lave en omregning via den naturlige- eller 10-logoitmen:

$$\log_2(x) = \frac{\log_{10}(x)}{\log_{10}(2)} = \frac{\ln(x)}{\ln(2)} \qquad [\text{Say00 side 15}]$$

Ofte ønsker man ikke kun at beregne *selv-information* for et enkelt tegn, men snarere for en hel besked, så det kan anslås hvor meget information den indeholder. Denne størrelse kaldes også for *entropi* og er for tegnene, $A_i = \{A_1, A_2, A_3 ... A_i\}$ givet ved:

$$H = \sum P(A_i) i(A) = -\sum P(A_i) log_b P(A_i) \qquad [\text{Say00 side 15-16}] \tag{2}$$

Entropi er især anvendelig inden for komprimering, da det kan bruges til at vurdere, hvor effektivt algoritmer komprimerer. Dette kan gøres ved at udregne den gennemsnitlige bitstørrelse $\bar{l}$ af hvert tegn i den komprimerede file. Hvis filen S indeholder $L(S)$ tegn, og den komprimerede størrelse af S er givet ved funktionen $K(S)$, må $\bar{l}$ være givet ved:

$$\bar{l} = \frac{K(S)}{L(S)} \qquad ; \text{ hvor } S \text{ er filen som komprimeres.} \tag{3}$$

Hvis $\bar{l}$ er større end entropien H, så er filen ikke komprimeret så effektivt, som det er teoretisk set er muligt. Filen må med andre ord indeholde unødvendig information. Denne metode er dog meget grov og kan ikke bruges til at vurderer alle former for komprimerings algoritmer. Shannon-Fano havde f.eks. aldrig overvejet, at sekvenser af tegn kunne grupperes og erstattes af små koder, som det er tilfældet i “dictionary-compression” algoritmerne så som LZW og LZ77. Informationsteorien er derfor kun anvendelig på komprimeringsalgoritmer, som erstatter enkelte tegn med mindre bitkoder. Disse algoritmer kaldes også for de *statistiske* komprimerings algoritmer[1]. Der kan desuden opstå mange komplikationer ved implementeringen af algoritmer på computer, som kun kan løses ved lagring af ekstra information[2].

[1] Se evt. side 4

[2] Se implementering på side 6

5 Teori om Algoritmer

5.1 Algoritmetyper

Vi har valgt at beskæftige os med to af de mest almindelige algoritmetyper, hvilket er statistiske algoritmer og ordbogs algoritmer.

5.1.1 Statistiske Algoritmer

Statistiske algoritmer kører på et princip om at nogle tegn indgår flere gange end andre i en fil. Algoritmerne laver en analyse af filen der skal komprimeres, og notere frekvens og/eller sandsynlighed for hvert tegn i filen. De tegn der optræde oftest i filen bliver derefter tildelt færre bit end dem, der optræder færrest. Dvs. i stedet for at hvert tegn fylder 8 bit, som normalt, omskrives filen så de hyppigste fylder mindre end 8 bit, og de sjældne fylder mere. Statistiske algoritmer er baseret på Shannons entropi.

5.1.2 Ordbogs Algoritmer

Ordbogs algoritmer kører efter et princip om at bestemte sekvenser af tegn optræder hyppigt, og derfor kan beskrives kortere. Der er flere metoder til at gøre dette, men ofte genererer man en "ordbogen", dvs. en kodebog på den del af filen, som er komprimeret.

5.2 Valg af Algoritmer

Algoritmerne er primært valgt ud fra et historisk perspektiv. Alle af algoritmerne er meget kendte og de bliver alle, bortset fra Shannon-Fano, brugt i dag. Desuden ligner Shannon-Fano og Huffman hinanden så meget, at implementeringen af dem er næsten ens, hvorfor det ville være omsonst ikke at bruge begge.

De valgte algoritmer er følgende:

Shannon-Fano kodning Shannon-Fano kodning var revolutionerende, da den er den første komprimeringsmetode, der er baseret på Shannon-Fanos entropi. Det er en statistisk komprimeringsmetode.

Huffman kodning Huffman kodning bygger i høj grad på samme principper som Shannon-Fano kodning, dog er den mere effektiv, både hvad komprimeringsgraden og køretid angår. Derfor er Shannon-Fano i dag afløst af Huffman.

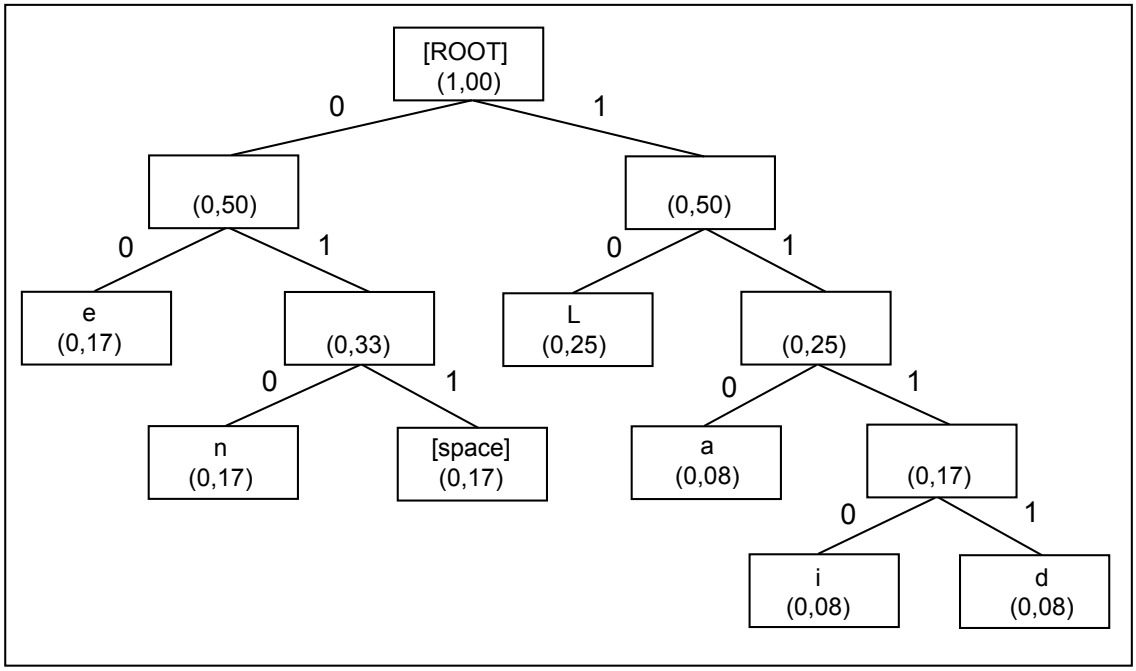
Arimetisk kodning Arimetrisk kodning er statistisk og komprimerer næsten så meget, som Shannon-Fanos teori beskriver det muligt.

LZ77 En meget anvendt algoritme i komprimeringsprogrammer i dag (i forskellige former). Det er en ordbogs algoritme, der er i stand til at komprimere en del bedre end de statistiske.

LZW LZW er en mere avanceret ordbogs algoritme end LZ77, men er patenteret, hvorfor LZ77 i højere grad bruges.

5.3 Huffman Kodning

Huffman kodning er en statistisk algoritme til pakning af filer. Den søger efter en måde, at notere tegn med høj sandsynlighed, så de fylder mindre end tegn med en lavere sandsynlighed. Huffman kodning går som Shannon-Fano kodning igennem filen, der skal pakkes, to gange; en gang for at finde sandsynligheden for tegnene, og en gang for at pakke filen.



Figur 1: Et Huffman-træ, et eksempel med teksten ”en lille and”. Det skal bemærkes at alle værdier er kun vist med to decimalers nøjagtighed som medfører f.eks. at 0,08 + 0,08 = 0,17

5.3.1 Huffman træ

For at opnå en mere optimal notering af tegn er det nødvendigt at lave en statistisk analyse af den fil, som ønskes komprimeret. Dette gøres ved at notere sandsynligheden for hvert tegns optræden i filen.

Derefter bygges et kodetræ på en sådan måde, at tegn med høj sandsynlighed ligger højest i træet - altså tæt på roden, og tegn med lavere sandsynlighed ligger længere nede. Et sådant træ kan opbygges efter følgende algoritme:

1. Opbyg en liste med sandsynligheder for alle tegn, således at hvert element, indeholder sandsynligheden for et tegn.

2. Omdan listen til en skov, således at hvert element er et træ.

3. Udvælg de to tegn (træer) med lavest sandsynlighed.

4. Sammensæt disse, så de bliver to blade på et nyt træ med en fælles rod.
5. Sæt sandsynligheden for det nye træ lig summen af sandsynligheder for dets blade.
6. Fjern de to brugte tegn (træer) fra listen og indsæt i stedet det nye træ.
7. Start forfra med punkt 3, indtil kun et træ er tilbage.

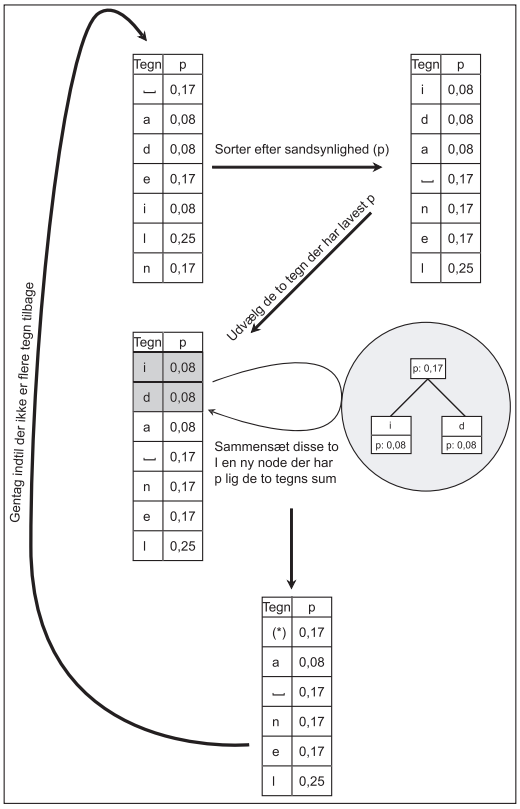
Træet på figur 1 er blevet opbygget på følgende måde: Først udtages de to tegn med lavest sandsynlighed fra listen, hvilket er "i" og "d". De bliver samlet i et træ bestående af to blade, et for hvert af tegnene, hvor træets rod bliver sat lig den samlede sandsynlighed af disse to værdier, altså $0,08 + 0,08 = 0,17$ (Se figur teksten på figur 1). Roden repræsenterer nu begge tegns samlede sandsynlighed og sættes tilbage i listen af tegn. De næste to værdier med lavest sandsynlighed er "a" ($0,08$) og den nylig indsatte rod ($0,17$). Disse værdier bliver hægtet sammen og indsættes i et nyt træ med sandsynligheden $0,17 + 0,08 = 0,25$. Da "i" og "d" allerede er hægtet sammen i et træ, og deres rod hægtet sammen med "a", bliver dette træ størrere. Den samlede sandsynlighed for "i" og "d" havde været meget større, havde algoritmen valgt et af de andre tegn til at hægte sammen med "a". Dette vil give to træer i stedet for et træ. Disse træer ville eventuelt blive hægtet sammen med et andet træ og til sidst vil der kun være et træ. Processen fortsætter, indtil alle værdier er brugt. Huffmantræet bliver derefter gemt på en kompakt form [3] i den komprimerede fil, så det kan genbruges i dekomprimeringen. Se figur 2 for et flow-diagram over algoritmen.

5.3.2 Komprimering

Når Huffmantræet er opbygget, kan oversættelsen af tegnene i filen påbegyndes. Den nye optimerede kode for de enkelte tegn findes ved at gå igennem træet. Hver gang man går til højre, noteres et 1'tal og hver gang man går til venstre, noteres et 0. Disse tal er tegnenes nye bit værdier.

[3] fylder maksimalt 258 byte

I det viste eksempel får tegnet “a” f.eks. bit-værdien “110”, mens “e” får værdien “00”.



Figur 2: Huffman algoritmen, et eksempel med teksten ”en lille and”

5.3.3 Dekomprimering

Dette foregår stort set på samme måde som komprimeringen. Der startes fra roden i kodetræet. Hver gang man har et 1’tal, går man til højre, og hver gang man har et 0, går man til venstre. Når man ender i et blad med et bogstav, hvilket også vil sige, at der ikke er videre forgreninger, udskrives dette bogstav. Derefter fortsættes fra roden af træet med næste bit.

5.4 Shannon-Fano Coding

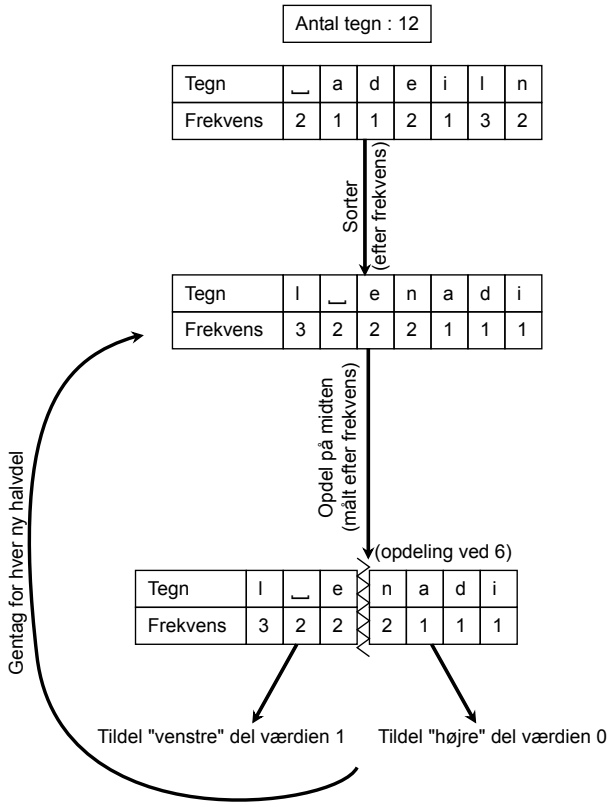
Shannon-Fano er som Huffman en statistisk algoritme. Forskellen mellem Huffman og Shannon-Fano er måden, hvorpå de opbygger deres kodetræer.

5.4.1 Shannon-Fano-træ

Shannon-Fano starter med en analyse af filen. For hvert tegn bliver frekvensen noteret. Derefter opbygges kodetræet via følgende algoritme.

1. Opbyg en liste indeholdende tegnene i filen.
2. Sorter listen, så tegnene med højest frekvens ligger først.
3. Optæl antallet af tegn i listen.
4. Opdel listen af tegn midt over set i forhold til frekvens. Første halvdel tilføjes bitkoden 0 og den anden halvdel bitkoden 1.
5. Gentag fra punkt 3 for hver halvdel.

Denne algoritme gentages for hver halvdel, indtil der ikke kan deles mere. Se figur 3 for et eksempel på denne algoritme. Komprimering og dekomprimering af filer v.h.a. denne



Figur 3: Grafisk repræsentation af Shannon-Fano algoritmen

algoritme sker på samme måde som Huffman.

5.5 Arimetisk kodning

Arimetisk komprimering er en statistisk algoritme, der benytter sig af de rationelle tal mellem 0 og 1. Denne komprimering har en stor fordel i forhold til Huffman algoritmen. Dette skyldes at Huffman algoritmen kun tildeler de komprimerede tegn hele bitkoder, selv om det optimale ifølge Shannon-Fanos entropi[4] ville være at bruge et kommatal, hvilket Arimetrisk kodning gør. Derfor er Arimetisk kodning også i stand til, at komprimere lige så godt, som Shannon-Fanos entropi siger det er muligt. Dog skal også kodebogen vedlægges, hvilket Shannon-Fanos entropi ikke tager højde for.

Hvis f.eks. entropien af tegnet *A* ligger på $0,025$ bit, så ville Huffman typisk tildele *A* en bitkode med en længde på én bit. Dette giver en forøgelse på: $\frac{100}{0.025}\% = 4000\%$, og pakningen bliver derfor langtfra optimal. Dette var dog et specielt konstrueret eksempel, og aritmetisk komprimering vil sjældent overgå Huffman med mere end et par procent. En ulempe ved aritmetisk komprimering i forhold til Huffman, er at den er yderst tidskrævende for computeren.

5.5.1 Komprimering

Først laves en statistisk analyse af filen, hvor en såkaldt interval-liste opbygges. Hvert tegns sandsynlighed p noteres. Derefter tildeles hvert tegn en del af intervallet, imellem 0 og 1 svarende til størrelsen af deres sandsynlighed p. En analyse over sætningen ´´bill gates" vil f.eks. give interval-listen i figur 4, hvor nedre og øvre værdi er grænserne for hvert tegns interval: I hele den følgende proces opererer man med en øvre og en nedre

Tegn	p	Nedre Værdi	Øvre Værdi
[space]	1/10	0,00	0,10
a	1/10	0,10	0,20
b	1/10	0,20	0,30
e	1/10	0,30	0,40
g	1/10	0,40	0,50
i	1/10	0,50	0,60
l	2/10	0,60	0,80
s	1/10	0,80	0,90
t	1/10	0,90	1,00

Figur 4: Interval Liste

grænse (ingen relation til tegnenes øvre og nedre værdier). Til at starte med er disse sat til henholdsvis 1 og 0. Algoritmen kører herefter, som følger:

1. Indlæs næste (eller første) tegn.

[4] se informationsteori side 9

2. Sæt arbejdsinterval = øvre grænse - nedre grænse.

3. Sæt ny øvre grænse = nedre grænse + arbejdsinterval $\times$ øvre værdi (for det indlæste tegn).

4. Sæt ny nedre grænse = nedre grænse + arbejdsinterval $\times$ nedre værdi (for det indlæste tegn).

5. Start forfra fra punkt 1, sålænge der er flere tegn.

Det søgte symbol vil da være værdien af den nedre grænse, når algoritmen er løbet igennem. Hvis vi følger udviklingen af de øvre og nedre grænser igennem algoritmen vil det se ud, som vist i figur 5. Heraf kan man se at det ønskede symbol, dvs. den

Nyt Tegn	Nedre grænse	Øvre grænse
	0,0	1,0
b	0,2	0,3
i	0,25	0,26
l	0,256	0,258
l	0,2572	0,2576
[space]	0,25720	0,25724
g	0,257216	0,257220
a	0,2572164	0,2572168
t	0,25721676	0,2572168
e	0,257216772	0,257216776
s	0,2572167752	0,2572167756

Figur 5: Arimetrisk kodning

komprimerede fil, er $= 0,2572167752$ (den nedre grænse ved det sidste tegn).

5.5.2 Dekomprimering

Når komprimeringsdelen af algoritmen først er forstået, er det forholdsvis let at forstå dekomprimeringsdelen.

I starten af den komprimerede fil noterer man interval-listen. Med denne som udgangspunkt, kan man let dekode. Fremgangsmåden er som følger:

1. Find det tegn, hvis interval det komprimerede symbol ligger indenfor.

2. Udskriv tegnet.

3. Sæt arbejdsinterval = tegnets øvre grænse - tegnets nedre grænse.

4. Sæt komprimerede symbol = (komprimeret symbol - tegnets nedre værdi)/arbejdsinterval.

5. Gå til 1 indtil det komprimerede symbol når værdien 0.

Hvis man benytter eksemplet fra komprimeringen, ligger symbolet 0,2572167752 mellem 0,2 og 0,3. Dermed ligger det inden for b's interval. Herefter bliver man nødt til at fjerne "b" fra det komprimerede symbol, for at kunne finde det næste. Dette gøres ved at vende processen fra komprimeringsdelen om. Det vil altså sige: trække b's nedre interval fra 0,2572167752 og derefter dividere med arbejdsintervallet (0,1). Dette giver 0,572167752, der ligger inden for i's interval. Et eksempel ses i figur 6.

Symbol	Tilhørende tegn	Tegnets interval	(Symbol-nedre værdi)/ arbejdsinterval
0,2572167752	b	0,2-0,3	0,572167752
0,572167752	i	0,5-0,6	0,72167752
0,72167752	l	0,6-0,8	0,6083876
0,6083876	l	0,6-0,8	0,041938
0,041938	[space]	0,0-0,1	0,41938
0,41938	g	0,4-0,5	0,1938
0,1938	a	0,1-0,2	0,938
0,938	t	0,9-1,0	0,38
0,38	e	0,3-0,4	0,8
0,8	s	0,8-0,9	0

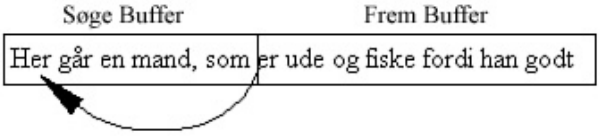
Figur 6: Arimetrisk dekodning

5.6 LZ77

5.6.1 Komprimering

Måden, hvorpå LZ77 komprimerer, er meget simpel. Der dannes en buffer, kaldet *søge-buffer*, ovenpå det stykke af filen, som algoritmen har kørt igennem. Bufferen bevæger sig i takt med, hvor langt algoritmen er nået i filen. Pointen med *søgebufferen* er, at algoritmen kan referere til den ved at benytte to tal, når den finder gentaget tekst. Dette foregår ved, at algoritmen udskriver, hvor langt tilbage i bufferen den matchende tekst blev fundet, og hvor lang den matchende tekst er. Endvidere opereres også med en *frembuffer*. Denne er dog kun defineret for at have et maksimum, som algoritmen kan kigge frem i filen.

Figur 7 demonstrere princippet for LZ77 algoritmen. I *frembufferen* findes den tekst,



Figur 7: Buffers brugt i LZ77

som algoritmen endnu ikke har komprimeret, mens *søgebufferen* indeholder den tekst, som allerede er komprimeret. Algoritmen starter med at finde den længste sekvens af tegn, som matcher i *søgebufferen* og *frembufferen*, hvilket i eksemplet ovenfor giver tegnene *e, r* i ordet *her*. Herefter udskriver algoritmen, hvor langt tilbage matchen blev fundet og dens længde, hvilket i ovenstående tilfælde giver: $(20, 2)$. Til sidst udskrives det næste tegn i teksten efter den fundne match, hvilket giver et mellemrum. Hvis der ikke findes nogen match i *søgebufferen*, udskrives værdierne $(0, 0)$ og tegnet, som ikke blev fundet. Et eksemple ses i figur 8.

SøgeBuffer	FremBuffer		Resulterende kode
	abe abe sabs	⇒	(0,0,a)
a	be abe sabs	⇒	(0,0,b)
ab	e abe sabs	⇒	(0,0,e)
abe	abe sabs	⇒	(0,0,)
abe	abe sabs	⇒	(4,4,s)
abe abe s	abs	⇒	(9,2,s)

Figur 8: Hvert trin for LZ77

Som det fremgår af den nederste linie i tabellen, bruger algoritmen den match, der er længst tilbage i vinduet, når flere match af samme størrelse findes.

Buffernes størrelser skal dog begrænses, da størrelsen af koderne afgøres af størrelsen på bufferne. Hvis man f.eks. vælger 12 bit til søge-bufferen, er det muligt at gå $2^{12} = 4096$ tegn tilbage i vinduet, mens f.eks. 8 bit begrænser rækkevidden til $2^8 = 256$ tegn. Ligeledes gælder det, at vælger man 4 bit til *frem-bufferen*, vil den længste match kunne være $2^4 = 16$ tegn, hvorimod 8 bit vil give mulighed for en sekvens på $2^8 = 256$ tegn. Hvor stor denne rækkevidde ønskes afhænger fra implementering til implementering, men jo større rækkevidden er, jo mere plads kræver det at lagre “matchen”, dvs. koden.

5.6.2 Dekomprimering

Dekomprimering af filer komprimeret af LZ77 er meget simpel. Algoritmen gør netop det, som der står i den komprimerede fil. Lad os tage et eksempel. Algoritmen vil altid møde et *(0,0,tegn1)* sekvens først i filen, da der ikke fandtes nogle tegn i *søgebufferen*, da algoritmen pakkede filen. Algoritmen får her at vide, at den skal udskrive alt fra 0 tilbage og 0 frem og derefter udskrive *tegn1* som står efter de to nuller. Derefter støder algoritmen måske på *(1,1,tegn2)* kode. Her får den at vide at den skal gå et tegn tilbage i den nydannede kode og udskrive alt der ligger mellem her og et tegn frem, altså *tegn1*. Derefter bliver *tegn2* udskrevet. Dette fortsættes, indtil algoritmen har kørt igennem al kode i den pakkede fil.

5.7 LZW

LZW [Sal97] er en ordbogs komprimerings algoritme, hvor kodebogen ikke bliver gemt i den pakkede fil. Kodebogen laves adaptivt ud fra filen der pakkes, og kan senere gendannes under udpakningsprocessen.

5.7.1 Komprimering

Først opstilles en kodebog med plads til et ønsket antal koder. Dette kunne typisk være $2^{12} = 4096$ antal koder. De første 0-255 koder i kodebogen initialiseres som de tegn, der normalt forefindes. Dette ville typisk være de 256 tegn som kan repræsentere en byte.

Så kigges på selve filen, der skal komprimeres. Algoritmen starter ved det første tegn. Algoritmens fremgangsmåde er som følger:

1. Find den længste sekvens af tegn i kodebogen, der matcher sekvensen af tegn som algoritmen er nået til.
2. Udskriv kodetallet fra kodebogen, der angiver tegn-sekvensen.
3. Såfremt kodebogen ikke er fyldt, tilføjes den netop anvendte sekvens af tegn + det næste tegn til kodebogen. Dette bliver den næste kode.
4. Hvis teksten ikke er slut, gå til punkt 1.

Eksempel: Lad os antage, at en byte kun kunne repræsentere fire tegn, hvorved vores kodebog ville indeholde: 1: a 2: b 3: e 4: _(mellemrum) Teksten, som skal kodes ser således ud: abe abe abe

Den komprimerede kode vil så være: 1234579. Skridt for skridt ser dette således ud:

Fil	Kode til komprimeret fil	Kode tilføjet til kodebogen		Nye kode betydning
a	1(=a)	5	=	ab
b	2(=b)	6	=	be
e	3(=e)	7	=	e_
_	4(=_)	8	=	_a
a	5(=ab)	9	=	abe
b				(indgår i sidste kode)
e	7(=e_)	10	=	e_a
_				(indgår i sidste kode)
a	9(=abe)			
b				(indgår i sidste kode)
e				(indgår i forrige kode)

Figur 9: Komprimering ved LZW

5.7.2 Dekomprimering

Dekomprimering foregår stort set som komprimeringen. Man starter med at opstille en kodebog i samme størrelse som ved komprimeringen, og initialisere den med de 256 tegn, som en byte kan repræsentere.

Algoritmen går derefter frem kode for kode. Hver gang den omsætter en kode til tegn, tilføjer den også næste kode til kodebogen. Dette gøres ved at tage tegnene tilhørende den nuværende kode og sammensætte dem med det næste. Denne nye kode får det næste ubrugte kode-nummer.

Eksempel: Lad os kigge på den komprimerede tekst ovenfor. Vi ved på forhånd, at de første 4 koder er defineret ved:

1: a 2: b 3: e 4: _ (mellemrum)
Og så har vi også den komprimerede tekst: 1234579
Vi går frem kode for kode. Se figur 10.

Dette giver os teksten: abe abe abe, hvilket er præcist det ønskede.

Kode fra komprimeret fil	Betydning af kode	Kode tilføjet kodebogen		Nye kode betydning
1	a	5	=	ab
2	b	6	=	be
3	e	7	=	e_
4	_	8	=	_a
5	ab	9	=	abe
7	e_	10	=	e_a
9	abe	(ingen)		(ingen)

Figur 10: Dekomprimering ved LZW

5.7.3 Omstændighed ved dekomprimering

Man kan i LZW algoritmen komme ud for at skulle bruge den kode, man lige har skabt. På denne måde kan man ved dekomprimeringen komme ud for at skulle bruge en endnu ikke defineret kode. Men for at dette kan ske, skal koden være den netop brugte kode plus det første tegn af denne.

Eksempel: 1: a 2: b
abababab Giver koden: 1235
Lad os prøve at dekomprimere den:

Dog kan man efter ovenstående teori bestemme, at eftersom 5 ikke eksisterer endnu, må den være den brugte kode plus første tegn af denne kode. Dvs: 5: aba

Kode fra komprimeret fil	Betydning af kode	Kode tilføjet kodebogen		Nye kode betydning
1	a	3	=	ab
2	b	4	=	ba
3	ab	5	=	? (her er omstændigheden, da vi ikke ved, hvad næste kode er)
5				

Figur 11: Omstændighed ved dekomprimering med LZW

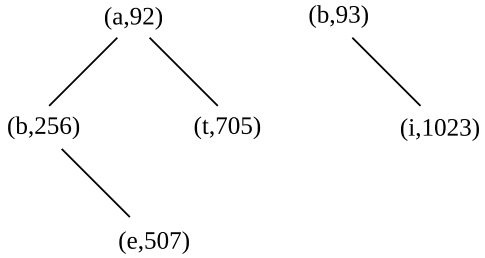
3	Ab	5	=	aba
5	Aba	-		-

Figur 12: Løsning på omstændigheden

5.7.4 Kodeskov til LZW

For at gøre søgningen hurtigere vælges en kodeskov som kodebog.

Øverst i kodebogen findes alle de 256 normale tegn en byte kan repræsentere. Hvert af disse tegn er roden i et træ, som indeholder andre tegn. Dvs. for at finde koden til tegnsekvensen “abe”, går man ned af “a”s træ, til det næste tegn “b” er fundet. Herefter søges i “b”s træ til tegnet “e” er fundet. Hermed er koden fundet. Se figur 13 for et eksempel på en LZW kodeskov.



Figur 13: En lille del af en LZW kodeskov

5.7.5 Størrelse af kodebog

Antallet af forskellige koder i kodebogen definerer også kodernes størrelse i den komprimerede fil. Hvis der f.eks. skal være $2^{12} = 4096$ forskellige koder, fylder hver kode 12 bit.

Eftersom antallet af koder konstant forøges, er det derfor en fordel at bruge variabelkodestørrelse, således at kodernes størrelse s (i bit) forøges, hver gang antallet af koder k overstiger antallet der er muligt inden for den givne kode størrelse. Dvs: $k < 2^s (< 2k)$

6 Implementering

6.1 Introduktion

I projektet har vi valgt selv at implementere de 5 komprimeringsalgoritmer, hvilket primært skyldes to årsager. For det første ville vi forbedre vores programmeringsfærdigheder og indsigt i algoritmerne, hvilket lige netop opnås når vi selv implementerer algoritmerne. For det andet ville vi give de udvalgte algoritmer så ens betingelser at arbejde under som muligt. Hvis vi i stedet havde valgt at bruge nogle allerede implementerede algoritmer, ville de højst sandsynligt være kodet på en vidt forskellig måde, da det er forskellige personer/firmaer som stod bag. Dette forsøger vi at undgå ved selv at implementere algoritmerne.

6.2 Overvejelser vedrørende implementeringen

Ved implementering af algoritmer i programkode er der mange faktorer, der skal tages hensyn til. Det er f.eks. vigtigt, inden kodningsprocessen starter, at fastslå, hvilke krav der skal stilles til implementeringen. Skal den være hurtig (have lav eksekveringshastighed), nem at vedligeholde eller måske være platformsuafhængig, så koden kan bruges på forskellige styresystemer.

I denne opgave er især hastighedsmomentet betydningsfuldt, da vi lige netop prøver at belyse forholdet mellem komprimeringshastigheden og komprimeringsgraden. En langsom implementering vil derfor give et misvisende resultat, som kan forvrænge konklusionen af denne rapport. Af samme årsag er store dele af implementeringen flyttet over i et såkaldt bibliotek (library), som vi selv har skrevet. Biblioteket fortager de operationer, som en komprimeringsalgoritme oftest har brug for, og sikrer dermed at de respektive komprimeringsalgoritmer er lige hurtige i deres basale funktionalitet, da de bruger den samme underliggende kode (se næste afsnit for nærmere beskrivelse).

Programmeringssproget C er valgt til selve implementeringen, da dette sprog er hurtigt og relativt simpelt at programmere i. De fleste komprimeringsprogrammer/algoritmer er desuden skrevet i C, hvilket umiddelbart giver en fordel, da det simplificerer en eventuel sammenligning og implementering, da koden ikke behøver at konverteres. Desuden er C standardiseret, hvilket gør det muligt for gruppen at arbejde på flere forskellige platforme (som f.eks. Linux, Windows og BeOS) uden at bekymre sig om eventuelle kompatibilitetsproblemer.

Implementeringen er valgt således, at alle komprimeringsalgoritmer indgår i et stort program, som kaldes *zar*[5]. Dette medvirker til, at afprøvningen af komprimeringsalgoritmerne bliver lettere, da vi kun behøver at udføre tests på et program. Dette har desværre også gjort implementeringen mere besværlig og kompliceret, da det har været nødvendigt at tilføje en del ekstra kode. En gennemgang af *zars* anvendelse kan findes på side A i appendikset.

[5] Navnet er en slags parodi på det professionelle pakkeprogram *rar* og arkiverings programmet *tar*

6.3 zarLib biblioteket

Det implementerede bibliotek, som vi har tildelt navnet *zarLib*, spiller to vigtige roller i implementeringen. For det første forenkler det selve kodeskrivningsprocessen, da det samler alle de basale egenskaber[6], som er nødvendige for en komprimeringsalgoritme, på et sted. Det er derfor ikke nødvendigt at skrive koden til de mest almindelige funktioner, da biblioteket sørger for dette.

Det andet hovedformål med biblioteket er at generalisere koden, så de forskellige algoritmer får så ens betingelser at arbejde under som muligt. F.eks. ville det være en stor fejlkilde, hvis de forskellige komprimeringsalgoritmer læser og skriver til filerne på en vidt forskellige måde. Dette burde helt overordnet set foregå ens i alle algoritmerne. I biblioteket bestræbes der desuden på at bruge den samme syntaks, så koden bliver let forståelig, når først bibliotekets grundprincipper er lært.

Bibliotekets vigtigste funktionaliteter kan kort beskrives, som følger:

a) Åbning og lukning af filer.

b) Manipulering (læsning/skrivning) af bits til og fra filer.

c) Håndtering af parametre fra kommandolinjen [7].

d) Fejlhåndtering.

6.4 Implementeringen

I de kommende afsnit vil der blive gennemgået, hvordan de enkelte algoritmer er blevet implementeret. Umiddelbart kunne man måske fristes til at tro, at en algoritme kan implementeres på en computer uden de store ændringer. Dette er desværre sjældent tilfældet, da aspekter såsom hastighed og hukommelse skal tages til efterretning. Implementeringen af en algoritme er derfor særdeles afhængig af computerens arkitektur og ender derfor ofte med at være meget mere avanceret end algoritmen i dens rå form.

Af samme årsag vil vægten i de følgende afsnit blive lagt på at forklare forskellene mellem de rå komprimeringsalgoritmer - dvs. algoritmerne beskrevet i teoriafsnittet, - og den implementerede udgave. Den implementerede kode vil derfor ikke blive gennemgået skridt for skridt, og den interesserede læser bedes kigge i den vedlagte kildekode, som kan ses i bilaget på side 66. I dette afsnit vil vi ikke diskutere udpakningsalgoritmerne nærmere, da de ikke har noget at gøre med målet for denne rapport.

6.5 Shannon-Fano algoritmen og Huffman algoritmen

Da implementeringen af Shannon-Fano- og Huffman algoritmen er forholdsvis ens, vil begge algoritmer blive beskrevet samlet i dette afsnit, så eventuelle gentagelser undgås.

[6] Såsom åbning af filer, fejlhåndtering og bit-manipulering.

[7] Dette kunne f.eks. være parameteren *-x*, der får biblioteket til at pakke ud. For mere information se appendiks A

Begge implementeringer stemmer temmelig overens med de rå algoritmer, men der er nogle vigtige forskelle, som er værd at pointere. Men lad os inden da først kort opsummere, hvordan algoritmerne virker:

1. Optæl antallet af tegn.
2. Sorter tegnene efter frekvens.
3. Opdel tegnene og tildel bitkoder.
4. Gentag for hver opdeling, indtil alle tegn er brugt.

I implementeringen optælles først antallet af tegnene og dernæst sorteres efter frekvens, fuldstændigt som beskrevet i algoritmerne. Dette foregår via biblioteksfunktionerne *zarCountCharFreq()* og *zarSortCharFreq()*, som er specielt optimeret til formålet. Dette betyder dog, at en konvertering bliver nødvendig, da biblioteket bruger nogle datatyper, som ikke er "fleksible" nok til Huffman eller Shannon-Fano algoritmen. Denne konvertering vil dog ikke medføre noget måleligt tidstab, da den kun er: $Q(n)$ for $n <= 256$.

De mest tidskrævende momenter i algoritmerne er opbygningen af træerne og udskrivningen af bits til den komprimerede fil; men da udskrivningen forgår via biblioteksfunktionen *zarSetBits()*, som alle de andre algoritmer (LZW, LZ77 og Arimetrisk kodning) også benytter, kan der ses bort fra hele denne proces ved evt. hastighedsberegninger.

Begge algoritmer bruger en sorteret liste til at opbygge deres træer. Huffman algoritmen er dog en smule mere avanceret, da der skal tages højde for, at træerne skal kunne flyttes rundt, når deres frekvenser ændre sig[8]. Dette medvirker til, at Huffman algoritmen bliver en smule langsommere end Shannon-Fano algoritmen i sin træopbygning.

6.6 LZW

Det første aspekt, som implementeringen af LZW skulle tage stilling til, var hvordan søgningen efter koder skulle foregå. Det kunne enten blive implementeret som en lineær søgning, hvor alle koderne søges igennem én for én, eller en hierarkisk søgning, hvor koderne opstilles i et søgetræ. Valget faldt på træet, da det matematisk set er hurtigere at søge igennem. Dette vil vi eftervise i det følgende:

Lad os forestille os, at vi har en given tekstfil t indeholdende $L(t)$ antal tegn. Teksten har den tilhørende kodebog k, med størrelsen $L_a(t)$, hvor a er antallet af tegn læst fra t. Ved lineær søgning vil det i værste tilfælde kræve, at hele kodebogen bliver gennemsøgt, før den søgte kode bliver fundet. Det vil med andre ord kræve op til $L_a(t)$ søgeoperationer, hvilket vil gentage sig, indtil alle tegnene i t er læst:

$$S(t) = \sum_{i=1}^{L(t)} L_a(t) \tag{4}$$

I træbaseret søgning, er det derimod ikke nødvendigt at søge efter koder, da man vælger at sortere træet efter bogstaver. Hvis f.eks. tegnene *A* og *B* læses, søges der først efter

[8] Se side 12 for mere information om Huffman algoritmen

tegnet *A* og så efter *B* i *A's* træ. Dette eliminerer en væsentlig mængde søgninger, da kun koderne under *A* skal gennemsøges. Da startlængden af kodebogen $L_{a=0}k$, lige netop er antallet af mulige tegn i t[9], er det maksimale antal af søgeoperationer givet ved:

$$S(t) = \sum_{i=1}^{L(t)} L_0(t) \tag{5}$$

Forskellen mellem de to søgeoperationer ΔS, lineær søgning 4 og træbaseret søgning 5 må derfor være givet ved:

$$\begin{aligned} \Delta S &= \sum_{i=1}^{L(t)} L_a(t) - \sum_{i=1}^{L(t)} L_0(t) \\ &= \sum_{i=1}^{L(t)} [(L_0(t) + \Delta L_a(t) - (L_0(t)] \\ &= \sum_{i=1}^{L(t)} \Delta L_a(t) \end{aligned} \tag{6}$$

Umiddelbart virker denne forskel ikke af meget, men da kodebogen i implementeringen kan vokse op til 24 bit, ville det ved lineær søgning kunne tage op til $2^{24} = 16.777.216$ flere søge-operation at behandle et enkelt tegn, hvilket er meget tidskrævende. Disse tal er dog approksimationer, da der ikke er taget højde for de mange operationer, det kræver at opretholde et søgetræ. Den lineære søgning kan desuden optimeres, så den altid søger fremad, når en kode er fundet. På denne måde behøver den lineære søgning ikke at begynde forfra for hvert læste tegn, og antallet af operationer kan derfor reduceres væsentligt.

Det andet element, som viste sig nødvendigt under implementeringen, var hukommelsesstyring. Desværre ville en detaljeret diskussion af denne drukne i tekniske detaljer, såsom cache, data-alignment og heap-management, men den interesserede læser bedes kigge i den vedlagte kildekode for nærmere detaljer. Overordnet var problemet dog, at man ikke kan forudsige, hvor mange koder der skal dannes for at pakke en given fil. Dette betyder, at implementeringen må antage, hvor mange koder der skal dannes (reserveres), før pakningsprocessen starter. Denne antagelse er nødvendig, fordi det ville være for langsomt at skulle allokere den hukommelse, en kode fylder, hver gang en ny kode dannes. Logisk set er naturligt, da det også i den virkelige verden er mere effektivt at hente en kasse øl ad gangen end at hente hver flaske for sig.

Men hvor stor skal denne antagelse være? 2.000 koder eller måske 2.000.000? Valget faldt på $2^{16} = 65.536$ koder, da denne størrelse ikke vil medføre urimeligt store hukommelsesbehov, og da der er plads til en rimelig mængde koder. Dette betyder dog, at implementeringen hele tiden må holde øje med, om den reserverede plads til koderne er opbrugt, så der skal allokeres ny plads.

[9] Dette hænger sammen med, at LZW algoritmen initialiserer sin kodebog med alle de mulige værdier (0-255), som et tegn kan repræsentere

Alt i alt betyder dette, at LZW algoritmen kunne være implementeret hurtigere, hvis hukommelsesforbruget var taget i betragtning. Dette vil naturligvis resultere i en lille måle-unøjagtighed i eksperimentet.

6.7 LZ77

Implementeringen af LZ77 stemmer stort set overens med den rå algoritme, og LZ77 var af samme årsag den letteste algoritme at implementere. Men lad os, før vi begynder at diskutere de tekniske detaljer, opsummere, hvordan algoritmen fungerer[10]:

1. Find den længste sekvens af tegn i *søge bufferen*, som matcher teksten i *frembufferen*.
2. Udskriv hvor langt tilbage matchen blev fundet.
3. Udskriv længden matchet.
4. Udskriv det tegn fra *frembufferen*, som ikke matchede i *søge bufferen*.
5. Opdater vinduet, så *frembufferen* starter efter det ikke matchende tegn.
6. Gentag indtil alle tegnene har været igennem *frem bufferen*.

I implementeringen foregår søgningen efter de matchende tegn lineært, hvilket betyder, at hvert eneste tegn i *søgebufferen* bliver sammenlignet med mindst et tegn fra *frembufferen*. Denne form for søgning er, som vist i afsnittet om LZW, ikke særlig hurtig; men en mere optimeret form for søgning, som f.eks. søgetræer, er desværre temmelig kompliceret at implementere for denne algoritme og kunne ikke nås inden for rammerne af dette projekt. Den rå algoritme for LZ77 beskrives desuden kun via lineær søgning i al den litteratur vi har læst, og idéen med at bruge søgetræer fik vi først meget sent i projektforløbet. Den relativt langsomme eksekveringshastighed for LZ77, forsøger implementeringen at råde bod for på forskellige måder. Bl.a. udskrives outputtet (*hvor langt tilbage matchen blev fundet* og *matchens længde*) i 16 bit, da computere lige netop er hurtigst til at manipulere størrelser[11], som er delelig med 8 bit. Af samme grund bruges der 12 bit ($2^{12} = 4096$) værdier til at repræsentere *tilbage matchet* (punkt 2), mens der bruges 4 bit ($2^4 = 16$) værdier til at holde styr på matchens længde (punkt 3). Dette medfører at komprimeringen ikke bliver helt så optimal, som det teoretisk set er muligt, da outputtet i starten af komprimeringen er større end det påkrævede. Det er f.eks. unødvendigt at *tilbage matchen* kan gå 4096 tegn tilbage, hvis der kun er læst 20 tegn fra filen.

Overordnet må der konkluderes, at implementering af LZ77 kunne have været udført mere optimalt, og LZ77 vil derfor være en betydelig fejlkilde i vores eksperiment, da implementeringen er meget langsom.

[10] Se teori om LZ77 på side 18 for mere information om LZ77 algoritmen

[11] Dette er dog en sandhed med modifikationer, da nyere maskiner er hurtigst ved størrelser delelige med 32- eller 64 bit

6.8 Arimetrisk kodning

Desværre viste implementering af Arimetrisk kodning at være meget vanskelig. Dette skyldes først og fremmest, at den overordnede algoritme ikke kunne implementeres i sin originale form, da algoritmen krævede, at meget lange komma tal skulle kunne ganges med hinanden, uden at præcision gik tabt. Dette er næsten umuligt at implementere på en computer, da komma tal lige netop har en yderst begrænset præcision. Dette problem kan dog undgås, hvis algoritmen implementeres via bit-manipulering på heltal[12]. Desværre blev hele denne proces besværliggjort af vores udviklingsmiljø (MSVC++6.0), der viste forkerte resulter af bit-manipuleringen under fejlfindings processen (debugging). Denne *bug* er *Microsoft* blevet notificeret om, og vi håber, at den vil blive rettet engang i fremtiden.

Alt i alt må vi erkende, at implementering af Arimetrisk kodning er opgivet, da den viste sig for kompliceret.

6.9 Konklusion af implementeringen

Overordnet om implementeringen kan det siges at der er en del fejlkilder, som skal tages i betragtning i analysen af eksperimentet. Målingerne for LZ77 vil f.eks. være en smule fejlagtige, da implementeringen kunne være hurtigere. Det ville derfor være logisk at tage højde for dette i resultatbehandlingen. Implementeringen af de andre algoritmer er derimod forholdsvis optimale og måleresultaterne burde derfor være relativt pæne. LZW algoritmen kunne dog være implementeret en smule hurtigere, hvis ikke hukommelsesforbruget var taget i betragtning, men dette er dog kun en lille fejlkilde, da implementeringshukommelsesstyringen ikke er specielt hastighedskrævende.

[12] Vi vil ikke beskrive dette nærmere, da det er et yderst komplekst implementeringsspørgsmål

7 Model

Den model, der skal afklare vores problemformulering, vil som beskrevet i indledningen være opbygget ud fra teoretiske og eksperimentelle data. De eksperimentelle resultater af algoritmernes tidsforbrug og komprimeringsgrad fås ved komprimering af en filsamling kaldet The Canterbury Corpus, som er en samling filer udvalgt med henblik på afprøvning af nye komprimeringsalgoritmer. Ud fra disse resultater vil der blive opstillet grafer for hver af algoritmerne.

Disse grafer bliver tilpasset med hjælp fra de teoretiske data, så evt. afvigelser kan elimineres. Resultaterne fra de teoretiske beregninger af algoritmernes tidsforbrug og komprimeringsgrad vil derefter blive opstillet som grafer. På denne måde kan de teoretiske og eksperimentelle data sammenlignes grafisk, så modellen kan opbygges.

Da modellen opbygges af eksperimentelle data, vil den være stærkt afhængig af implementeringen af vores algoritmer. Hvis en algoritme implementeres ringere end en anden, vil de eksperimentelle målinger være fejlagtige. Modellen vil derfor opbygges på grundlag af både eksperimentelle og teoretiske data, da det sikrer den mest nøjagtige model.

Den endelige model skal derefter kunne afklare, hvilken algoritme der er den mest optimal til komprimering af en given fil.

7.1 Komprimerings filer

Til udarbejdelse af modellen skulle der bruges en række filer, som er et repræsentativt gennemsnit af filer, man normalt vil komprimere. Valget faldt på ”The Canterbury Corpus”.

7.1.1 The Canterbury Corpus

The Canterbury Corpus [Cam97] blev udviklet i 1997 som en udskiftning af Calgary Corpus. Canterbury og Calgary Corpus’erne er en samling af filer, udvalgt med henblik på afprøvning af nyudviklede komprimeringsalgoritmer. Med denne samling filer har man et udgangspunkt, hvorfra man kan bedømme komprimeringsalgoritmernes effektivitet. ”The Canterbury Corpus” var en nødvendig udskiftning af ”The Calgary Corpus”, da man i løbet af årene var begyndt at optimere algoritmerne specielt til ”The Calgary Corpus”. En anden årsag til udskiftning af de forældede ”Calgary”-filer har været internettets opblomstring, da typen af filer, der komprimeres i dag, er forskellige fra de tidligere. Ved udvælgelsen af Canterbury blev vægten lagt på følgende kriterier:

1. Filerne skal være repræsentative for de filer, der vil blive anvendt i fremtiden.
2. Filerne skal være nemt tilgængelige, f.eks. via Internettet.
3. Filerne skal alle være *public domain*, altså filer der ikke er ophavsret på.
4. Der skal ikke være for mange filer, da en sammenligning hermed bliver for besværlig.

Under udvælgelsen blev der indsamlet 800 potentielle filer, der skulle være uden ophavsrettigheder. Disse filer var inddelt i følgende grupper: Engelsk tekst, C source code, HTML og binære UNIX filer. For at finde de 11 filer, der skulle bruges, komprimerede man alle filerne i de forskellige grupper flere gange med forskellige algoritmer. Herefter blev de mest repræsentative filer udvalgt via lineær regression over resultaterne. Disse filer blev til ”The Canterbury Corpus”.

Da nogle af filerne i ”The Canterbury Corpus” er meget små (mindre end 10.000 bytes) har vi valgt at erstatte dem nogle større, da det ofte er store filer som der ønskes komprimeret. I tabellen nedenfor ses de filer, som vi har udvalgt til komprimering hvor filerne markeret med ”(*)” er de filer, vi selv har udvalgt:

Fil	Type	Kategori	Size (byte)
alice29.txt	tekst	English text	152.089
asyoulik.txt	tekst	Shakespeare	125.179
cp.html	kode	HTML source	24.603
fields.c	kode	C source	11.150
tlotr.txt (*)	tekst	English text	1.871.872
kennedy.xls	binær	Excel Spreadsheet	1.029.744
lcet10.txt	tekst	Technical writing	426.754
plrabn12.txt	tekst	Poetry	481.861
ptt5	tekst	CCITT test set	513.216
sum	binær	SPARC Executable	38.240
slashdot.html(*)	kode	HTML source	1.015.808

7.2 Forsøget

Grundideen i forsøget er at køre de 4 algoritmer på alle filerne. Vi har fra starten prøvet at møde følgende kriterier for udførslen.

- Forsøget skal foregå i et miljø med så få fejlkilder som muligt.
- Forsøget skal udføres på en sådan måde, at man let kan indsamle forsøgsdata.
- Forsøget skal udføres på en sådan måde, at det let kan rekonstrueres og gentages.

7.2.1 Forsøgsmiljøet

[13] Alle forsøgene blev udført på den samme maskine og platform. Vi valgte Linux som platform, da vi mente, det ville gøre det lettere at kontrollere og minimere antallet af sideløbende processer. På denne måde kunne vi også minimere fejlkilder fra operativsystemets side.

[13] Se appendix C for en fuld teknisk redegørelse af forsøget

7.2.2 Forsøgsgang

Til lejligheden blev der skrevet et program, der kunne køre *zar* med de nødvendige parametre[14] på de forskellige filer. Dette program stod også for aflæsning og lagring af resultater på en form, der kunne læses af excel. Herefter blev excel brugt til analyse af data og opstilling af grafer.

7.3 Resultater og grafer

Resultaterne over algoritmernes kørsel på filerne kan ses i graferne nedenfor. Disse resultater vil sammen med de teoretiske udregninger blive brugt til at opbygge den endelige model.

Hver graf har en linje, som beskriver det gennemsnitlige resultat for algoritmen. Gennem hvert punkt findes der en lodret linje, som beskriver den tidsmæssige afvigelse ved de 10 test. Punktet repræsenterer derfor det gennemsnitlige resultat for de 10 tests.

Det ses på alle grafer, at *kennedy.xls* filen ligger langt fra gennemsnittet. Dette kan forklares ved, at filen indeholder flere tegn, som går igen mere en det normalt er tilfældet. Dette betyder, at komprimeringen tager kortere tid og bliver for effektiv set i forhold til gennemsnittet af de andre filer. Filen er derfor blevet fjernet fra udregningerne og dermed også fra gennemsnitslinjen. Den kan dog stadig ses på graferne.

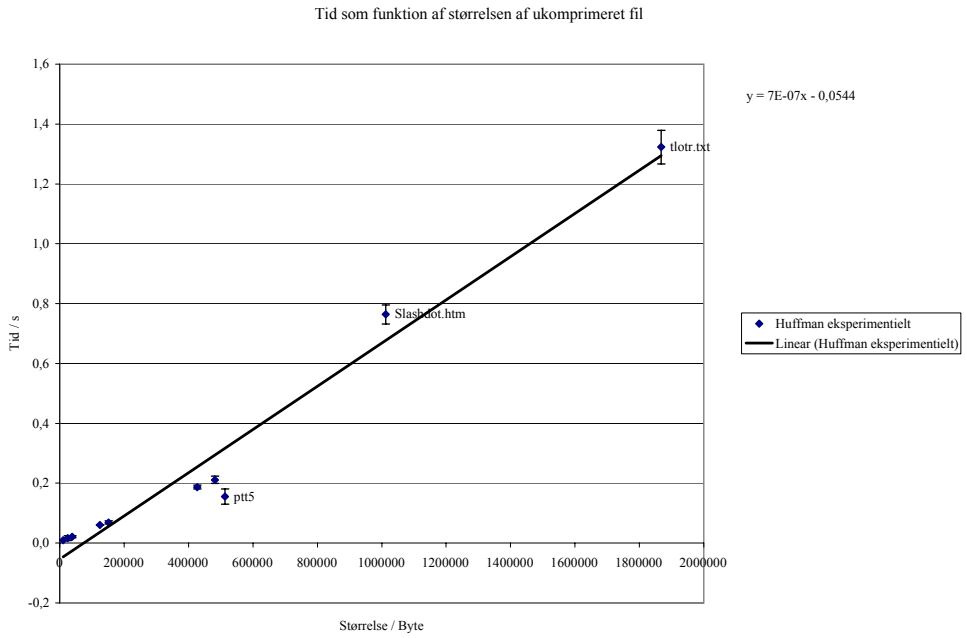
Ptt5 er også, som *kennedy.xls*, en binær fil. Den er ikke udtaget, selvom den også indeholder tegn, som findes gentaget mere end sædvanligt. Dette skyldes, at vi er lidt interesseret i binære filer, da mange tekstfiler indeholder nogen binære kode. Dette ses f.eks. i word og wordperfect filer. Filen er derfor medtaget, da graferne bliver mere realistiske.

Graferne er opbygget med de oprindelige filers størrelse ud af x-aksen og komprimeringstiden op af y-aksen. Dette giver et mål for, hvor hurtigt den enkelte algoritme er til at komprimere en fil med en given størrelse.

7.3.1 Huffman

På figur 14, ses at punkterne pænt følger en ret linje, indtil filerne fylder omkring en megabyte. Herefter tager det en del længere tid, og punkterne begynder at afvige. Årsagerne til dette kan være mange, men det skyldes sandsynligvis, at en standard implementering har en hukommelsesbegrænsning på 1 megabyte. Herefter skal styresystemet allokere mere hukommelse, hvilket er tidskrævende. Dette er dog ikke en fejlkilde i den endelige model, fordi dette er vanskeligt at undgå, da et styresystem altid sætter hukommelsesbegrænsninger.

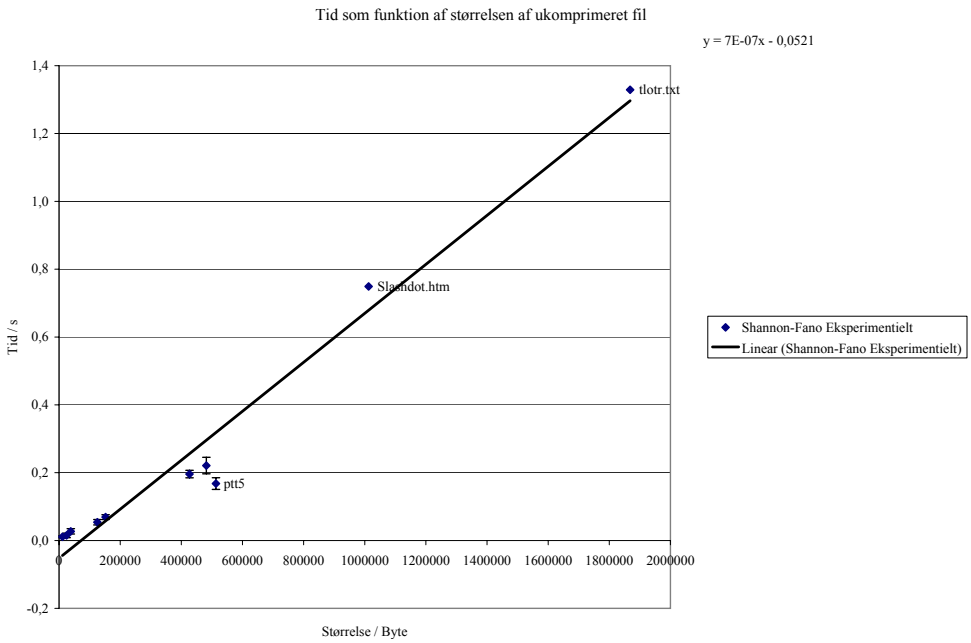
[14] Se appendix A for brugervejledning til zar



Figur 14: Graf over resultater fra test af Huffman-algoritmen.

7.3.2 Shannon

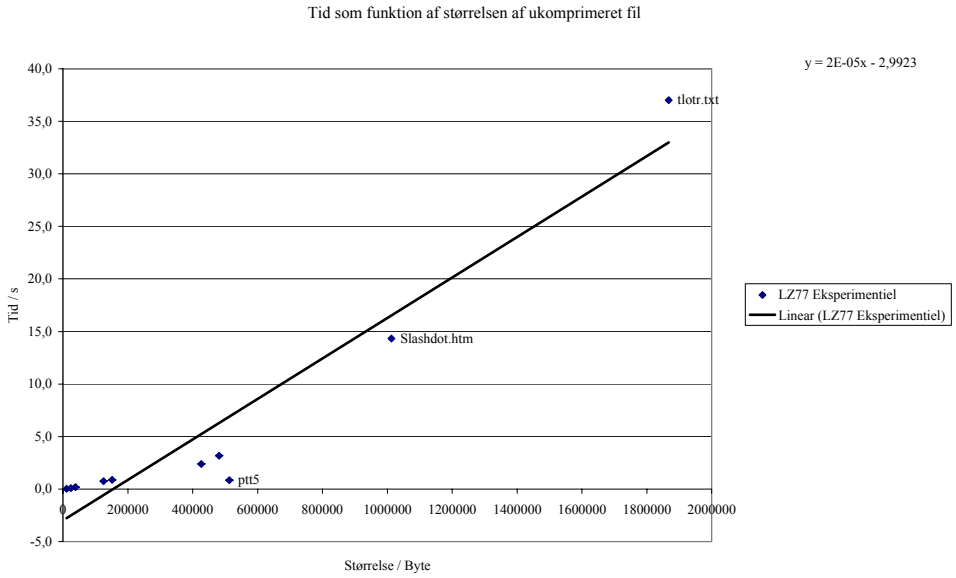
Hvis man sammenligner figur 15 fra Shannon eksperimentet med figur 14 fra Huffman eksperimentet, ser man, at graferne er næsten identiske. Ved en megabyte grænsen afviger denne graf også, hvilket er helt naturligt, da de to algoritmer er implementeret stort set ens.



Figur 15: Graf over resultaterne fra test af Shannon-Fano algoritmen.

7.3.3 LZ77

LZ77 er den algoritmen, som er længst tid om at komprimere filerne. Dette skyldes som tidligere omtalt, at algoritmen er implementeret via lineærsøgning. Af grafen ses, at filerne, som fylder mere end en megabyte, afviger meget fra den usynlige rette linje som de første 7 punkter på grafen kan danne. Grunden til, at den gennemsnitlige linje ligger så højt i forhold til den usynlige linje, er netop, at filerne som fylder over en megabyte, påvirker gennemsnitslinjen meget. Årsagen til, at *tlotr.txt* filen afviger så meget i forhold til den usynlige linje, er, at filen er relativt stor og ikke indeholder megen gentagelse.

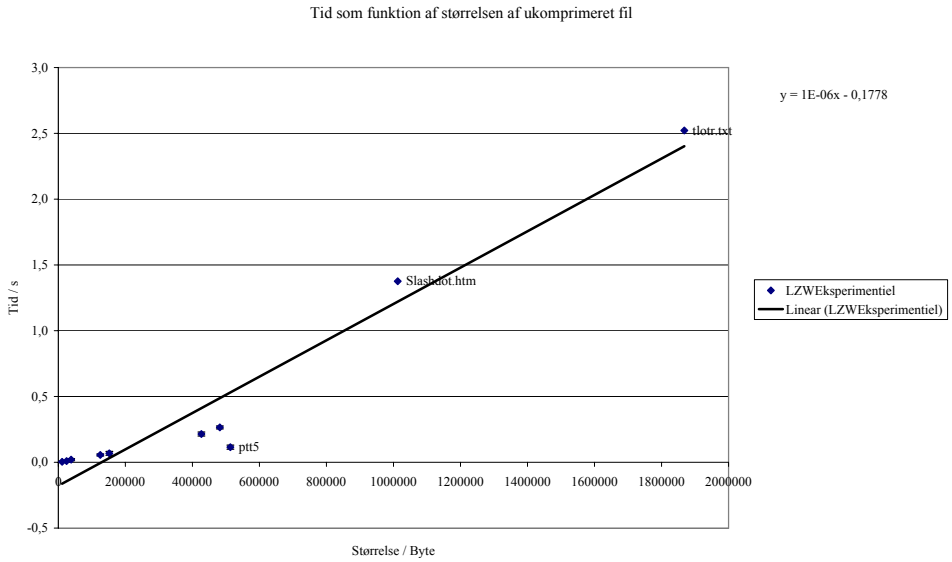


Figur 16: Resultater fra test af LZ77-algoritmen

7.3.4 LZW

De to filer med en størrelse på omkring en megabyte tager længere tid at komprimere end gennemsnitslinjen. Af grafen ses, at LZW algoritmen komprimerer filer på cirka en megabyte forholdsvis langsomt. Dette skyldes implementeringen, som ikke vil allokere for meget unødvendig hukommelse, når komprimeringen starter, og derfor må allokere den senere. Dette er yderst tidskrævende.

ptt5 filen befinder sig mod forventning lidt under gennemsnittet. Vi havde regnet med, at filen lå over gennemsnittet, da implementering af LZW algoritmen er langsommere til komprimering af binære filer. Det ses i komprimeringen af *kennedy.xls* filen som for de andre algoritmer altid har været hurtigere at komprimere end *slashdot.html*.



Figur 17: Graf over resultaterne fra test af LZW-algoritmen.

7.4 Teoretisk komprimeringsgrad

Til den teoretiske udregning af komprimeringsgraden bruges Shannon-Fanos *entropi* begreb. Denne størrelse er, som beskrevet i afsnit 4.1, velegnet til at vurdere effektiviteten af algoritmens komprimering.

Da beregningerne af entropi H er forholdsvis besværlig at udføre på større filer, har vi implementeret et program, der udfører netop dette. Programmet, som kaldes *zarEntropy*[15], kan beregne en fils entropi og maksimale komprimeringsgrad, hvor den maksimale komprimeringsgrad m er beregnet efter følgende formel:

$$m = H * l * 8 \qquad ; \text{hvor } l \text{ er filens længde og H er filens entropi} \tag{7}$$

I formlen ganges der med 8 så m enheden bliver i bytes i stedet for bits. Da Huffman algoritmens komprimeringsrate (bits pr. tegn) R ligger inden for intervallet [IDC00 side 49], $H \leq R \leq H + 1$, kan algoritmens komprimeringsgrad beregnes ved hjælp af entropien. Dette gøres "groft" ved at addere 1 til entropien.

Desværre er Shannon-Fanos komprimeringsgrad ikke så let beregnelig som Huffmans, og vi vil ikke nævne den nærmere i denne rapport. Dette er dog ikke den store fejlkilde, da Shannon-Fano algoritmen højst kan pakke ligeså effektivt som Huffman algoritmen [DC97 side 38]. De to algoritmer har stort set den samme komprimeringstid (Se evt. side 38). På figur 18 kan entropien af filerne fra *The Canterbury Corpus* aflæses.

På figuren er de teoretiske komprimeringsstørrelser, som Huffman algoritmen minimum kan opnå på filerne fra *The Canterbury Corpus* beregnet. Dette er et vigtigt element i vores rapport, da vi på denne måde har fået en teoretisk størrelse for komprimeringsgraden, som kan anvendes i vores model.

[15] Se nærmere beskrivelse af *zarEntropy* på side 57

Fil:	Entropi	Org. størrelse	Entro. størrelse	Huff. størrelse
alice29.txt	4,5676	152.089	86.834	105.846
asyoulik.txt	4,8586	125.179	78.527	91.672
cp.html	5,2671	24.603	16.622	19.274
fields.c	5,0546	11.150	7.353	8.439
kennedy.xls	3,5735	1.029.744	459.970	588.692
lcet10.txt	4,6691	426.754	249.070	302.414
plrabn12.txt	4,5313	481.861	272.929	333.164
ptt5	1,2102	513.216	77.635	136.015
sum	5,3290	38.240	25.472	30.253
Slashdot.html	5,2293	1.012.708	661.964	788.558
tlotr.txt	4,4029	1.868.014	1.028.096	1.261.587
Gennemsnit:	4,427			

Figur 18: Tabel over entropiens og Huffman algoritmens teoretiske komprimering af *The Canterbury Corpus*.

Den teoretiske komprimeringsgrad for LZ77 og LZW kan ikke findes som nævnt i afsnittet om informationsteorien.

7.5 Teoretisk køretid for algoritmer

For nogle algoritmer kan man direkte udregne en nogenlunde præcise værdi mens det for andre kun bliver en vurdering. Som beskrevet i implementeringen skal der ikke tages højde for alle de operationer, algoritmerne udfører for at kunne operere på en computer. Da de eksperimentelle tidsfaktorer allerede er udregnet, vil de tidsmæssige målinger fra teorien blive brugt til finjustering af modellen.

7.5.1 LZ77

Hver gang en kode genereres, kan den have en forskellig størrelse, og derfor er det umuligt at finde det præcise antal operationer pr. ukomprimeret tegn. Dog er det meget let at finde antal operationer pr. kode.

Da vi har valgt at køre med en søgebuffer på 12 bit, vil køretiden generelt ligge på 2^{12} operationer pr. tegn, - dvs: $4096n$, hvor n er længden af filen. Dette vil dog ikke gælde for de første koder, da søgebufferen ikke er blevet fuldvoksen. Men da dette maksimalt kan gælde for de 4096 første koder, kan man som regel se bort fra dette, da der normalt findes meget flere end "4096"tegn i filen.

7.5.2 LZW

For hver gang algoritmen får komprimeret et tegn i filen, vil den søge efter tegnet på et trin i kodetræet. Som beskrevet i afsnit 5.7 vil dette maksimalt medføre 256 søgninger pr. tegn, hvilket dog sjældent er tilfældet ved komprimering af tekstfiler, da de ofte kun indeholder 127 forskellige tegn. Antallet af søgeoperationer for en fil med længden n og antallet af koder k (i den komprimerede fil) må altså være givet ved: $(256k + n)$.

Beregning af antallet af koder Antallet af koder k kan kun beregnes ud fra den allerede komprimerede fil, da k afhænger af komprimeringsgraden. Koderne gemmes i stigende bitstørrelser. Hver gang en bit tilføjes, fordobles antallet af koder i kodebogen, da $2^{n+1} = 2 * 2^n$. Som nævnt i afsnit 6 kan LZW's kodebog vokse op til 24 bit, dvs. $k_{max} = 2^24 \approx 16 \times 10^6$.

Der startes med at initialisere algoritmens 256 koder, som beskrevet i afsnit 5.7. Derfor må algoritmen starte med at gemme i 9 bit koder, således at der er 512 (=2^9) muligheder for kodebogen. Dvs. at de første 256 (fra 256 til 512) koder er gemt i 9 bit, hvilket i den pakkede fil fylder $n_p = \frac{9 \times 256}{8} = 640$. Derefter vil bitstørrelsen stige, pga. pladsmangel i kodebogen til 10 bit, hvilket de næste 512 koder vil være. På denne måde fortsættes, hvorfor den komprimerede fils størrelse må være n_p, hvor b er den nåede bitstørrelse og k antallet af koder:

$$n_p(k) = \frac{256 \times 9}{8} + \frac{512 \times 10}{8} + \frac{1024 \times 11}{8} \ldots + \frac{2^(b-1) * (b-1)}{8} + \frac{b \times (k - n_p(2^(b-1)}{8}$$

Denne fremskrivning er svær at få omdannet således at udregningen af antal koder kan ske umiddelbart ud fra antallet af bytes i den pakkede fil. En let måde at løse dette på er ved at lave en liste over, hvor mange bytes i den pakkede fil der maksimalt kan være for en given bitstørrelse, og undersøge i hvilket interval den pakkede fil ligger. Derefter kan udregningen let foretages.

7.5.3 Huffman og Shannon-Fano

Da algoritmerne tidsmæssigt kører efter samme principper er de beskrevet samlet.

Begge algoritmer benytter et kodetræ. Før man kan bygge træet, må man finde antal tegn i teksten og hvor stor frekvens, de har. Dette tager (n) tid, hvor n er størrelsen af filen. Tegnene skal derefter sorteres efter frekvens, hvilket tager $(256log(256))$ tid, da quicksort metoden benyttes til dette. Træet kan herefter bygges.

Når Shannon-Fano kender summen af alle frekvenser, skal den finde ud af, hvor tegnene skal deles. Hvis alle 256 tegn findes med samme frekvens, må algoritmen dele tegnene i to lige store halvdele hver gang. Dette kræver 255 delinger, da den første deling kræver en deling, den næste to, den tredje 4, op til den ottende som kræver 128. Summen giver os 255.

Huffman tager længere tid om at opbygge sit træ. Hvis frekvensen er den samme for alle 256 tegn, skal algoritmen søge gennem alle tal, før den kan bygge en del af træet, dvs. at en liste bestående af tegn sorteret efter frekvens vil miste de to første elementer i listen. Frekvensen for disse elementer adderes, og listen bliver derefter søgt igennem

for at finde hvor den nye samlede frekvens skal indsættes. Hvis listen er k lang, tager det $k-2$ i tid. Da alle tegn har samme frekvens, må alle samlede to tegn sættes ind til sidst i listen. Men listen bliver mindre og mindre da man erstatter to tegns frekvens med en samlet frekvens. D.v.s. den første kode tager $k-2$ i tid, og den næste tager $k-3$ tid osv. Når $k-p=2$ hvor p er en variabel, er det ikke nødvændigt at lede mere, da de to sidste tal bliver til et samlet tal. Tidsmæssigt tager hele processen $\frac{(k-2)(k-1)}{2}$. Dette ses fra biomarginal fordelingen, kendt fra matematik.

Til sidst skal begge algoritmer erstatte tegnene i filen med de nyfunde værdier fra træet. Dette tager n tid da filen er n lang. Summen for Shannon Fano giver en køretid på $(3n+(k-1)+k*log(k))$.

Summen for Huffman giver en køretid på $(2n+\frac{(k-2)(k-1)}{2}+k*log(k)+k)$.

7.6 Sammenfatning

Den samlede model vil bestå af en sammensmeltning af de teoretiske og eksperimentelle data, så den bliver så pålidelig som muligt. Det er umiddelbart ikke muligt at sammenligne den eksperimentalt fundne komprimeringshastighed med den teoretiske, da de henholdsvis måles i sekunder og antal udførte operationer. Dog kan antallet af operationer omsættes til sekunder, hvis antallet af operationer pr. sekund beregnes. Dette beregnes ved at dele det gennemsnitlige antal af operationer $\overline{o}$ med den gennemsnitlige komprimeringstid $\overline{t}$, der bruges pr. fil for hver algoritme. Dette beskrives matematisk i den følgende formel, hvor o er antallet af operationer pr. sekund:

$$o = \frac{\overline{o}}{\overline{t}}$$

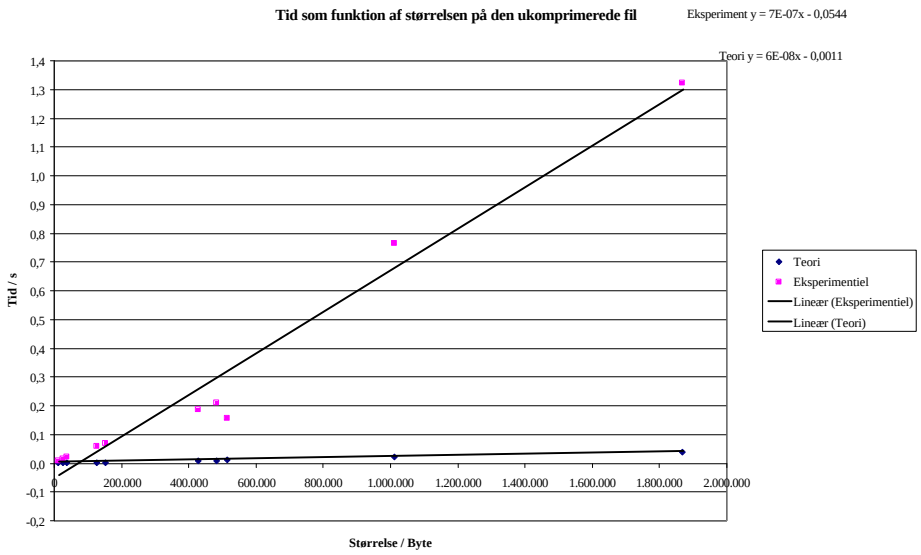
På figur 19 er antallet af operationer i *worst case* samt $\overline{o}$ beregnet for algoritmerne. Da det gennemsnitlige antal af operationer er meget højt p.g.a. LZW og især LZ77 algoritmen, bliver denne omdannelses faktor ikke så præcis for de andre algoritmer. Dette har dog ikke den store betydning, da implementeringen af Huffman og Shannon algoritmerne, efter vores vurdering, ikke er påvirket af de store fejlkilder.

			LZW	LZW	LZ77	LZ77	Shannon-Fano	Shannon-Fan	Huffman	Huffman
			Teori	Teoretisk	Teori	Teoretisk		Teoretisk	Teori	Teoretisk
	Entropi		Tid	Tid	Tid	Tid		Tid	Tid	Tid
	n	Pakket i %		s		s		s		s
Alice29.txt	152.089	42,91	39.006.761	0,269	622.956.544	4,292	1.092.317	0,008	1092317,07	0,008
Asyoulik.txt	125.179	39,27	32.112.396	0,221	512.733.184	3,533	888.461	0,006	888460,89	0,006
Cp.html	24.603	34,16	6.312.794	0,043	100.773.888	0,694	157.236	0,001	157236,48	0,001
Fields.c	11.150	36,82	2.860.815	0,020	45.670.400	0,315	67.426	0,000	67426,11	0,000
Lcet10.txt	426.754	41,64	109.431.169	0,754	1.747.984.384	12,044	3.256.208	0,022	3256207,81	0,022
Plrabn12.txt	481.861	43,36	123.623.230	0,852	1.973.702.656	13,599	3.702.099	0,026	3702099,37	0,026
Ptt5	513.216	84,87	131.450.674	0,906	2.102.132.736	14,484	3.957.048	0,027	3957048,42	0,027
sum	38.240	33,38	9.809.169	0,068	156.631.040	1,079	251.714	0,002	251714,48	0,002
Slashdot.htm	1.012.708	34,63	259.854.346	1,790	4.148.051.968	28,581	8.107.217	0,056	8107216,93	0,056
Tlotr.txt	1.868.014	44,96	478.966.843	3,300	7.651.385.344	52,719	15.451.053	0,106	15451052,88	0,106
Kennedy.xls	1.029.744	55,33	263.964.406	1,819	4.217.831.424	29,061	8.251.059	0,057	8251058,89	0,057
Gennemsnit			119.342.820		1.906.202.214		3.693.078		3.693.078	
					snit teori(t)	508.232.798				
					snit t	1,75090				
					snit teori(t) / snit t	145.134.730				

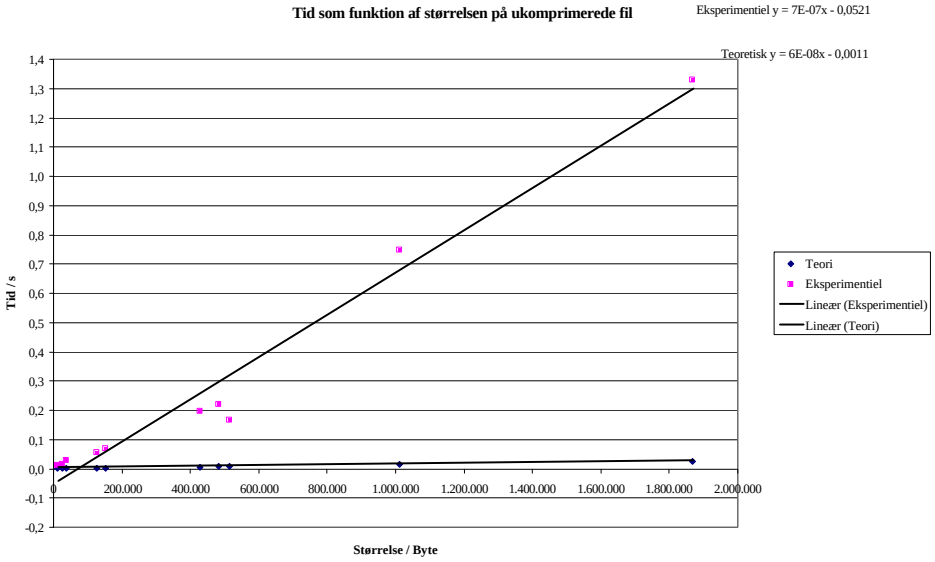
Figur 19: Antal operationer og tidsbergning for algoritmerne

7.6.1 Shannon-Fano og Huffman

Som det ses af graf 20 og 21, afviger Shannon-Fano og Huffman algoritmerne meget fra den teoretiske komprimeringshastighed. Dette skyldes som nævnt ovenfor LZW og LZ77s påvirkning af omdannelsesfaktoren. Derfor bruges kun de eksperimentalt målte hastigheder i modellen.



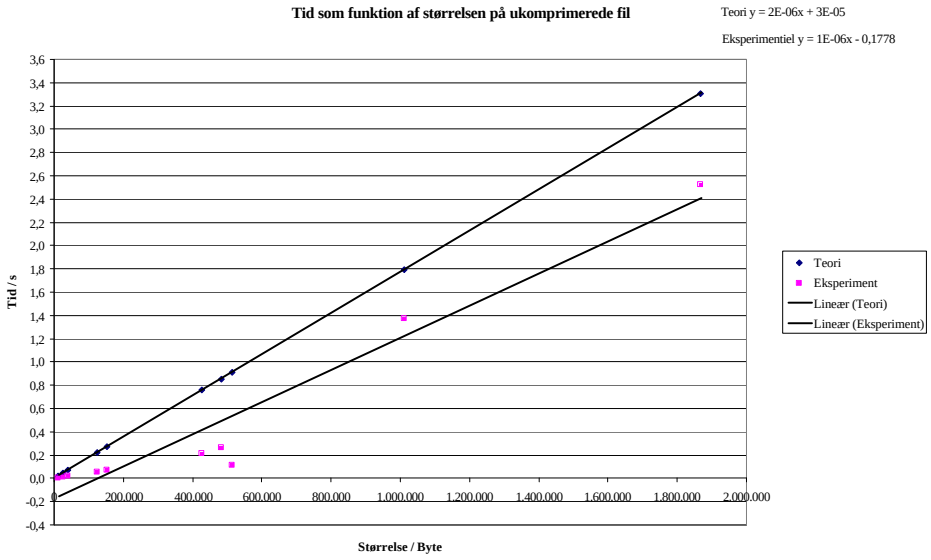
Figur 20: Den teoretiske komprimeringstid kontra den eksperimentelle for Huffman algoritmen



Figur 21: Den teoretiske komprimeringstid kontra den eksperimentelle for Shannon algoritmen

7.6.2 LZW

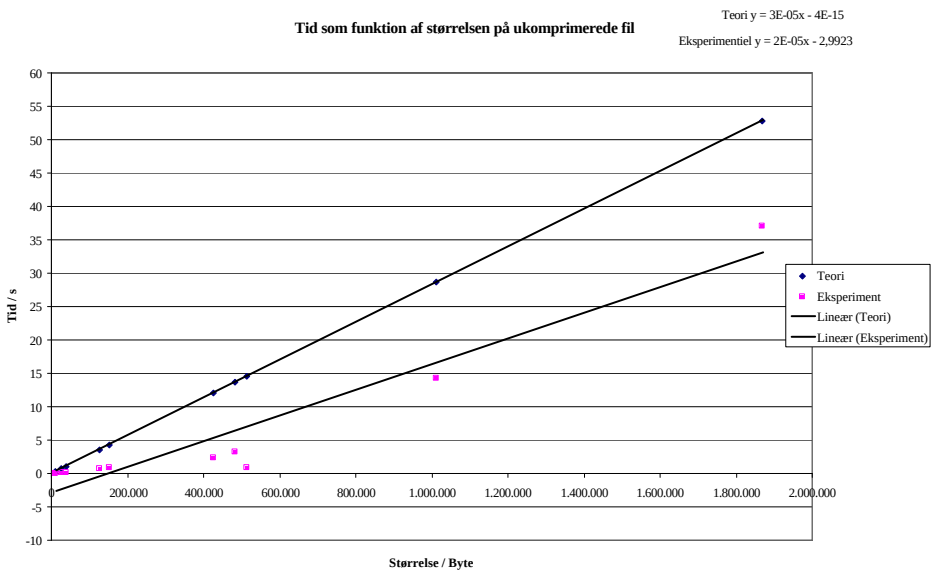
Af graf 22 ses, at den eksperimentelle hastighed ligger forholdsvis tæt på den teoretiske *worst case* linje, hvilket vil sige, at implementeringen umiddelbart virker langsom. Det ses dog, at punkterne for de mindre filer ligger en del under *worst case* linjen. Dette stemmer helt overens med analysen fortaget i resultat afsnittet, hvor vi konstaterede at en megabyte grænse havde en væsentlig betydning for hastigheden. Dette kan desværre ikke medregnes i den endelige model, da denne grænse er forskellig fra styresystem til styresystem, og modellen vil derfor bruge den lineære hældning for de mindre filer. Dette er en rimelig antagelse, da hukommelsesgrænsen for nyere styresystemer konstant forøges, efterhånden som computerteknologien udvikler sig.



Figur 22: Den teoretiske komprimeringstid kontra den eksperimentelle for LZW algoritmen

7.6.3 LZ77

Af graf 23 fremgår det, at LZ77 ligger tæt på *worst case* linjen. Dette betyder dog ikke nødvendigvis, at implementeringen er for langsom, da de andre hurtigere algoritmer trækker den teoretiske *worst case* linje ned. Dog kunne implementeringen af algoritmen være gjort anderledes, så søgningen kunne være foregået hurtigere. Dette ville også afspejle sig i den teoretiske beregning, som også ville blive væsentlig hurtigere. Vi vælger dog at bruge de eksperimentelle data, da punkterne ligger jævnt fordelt omkring den eksperimentelle gennemsnitslinje.

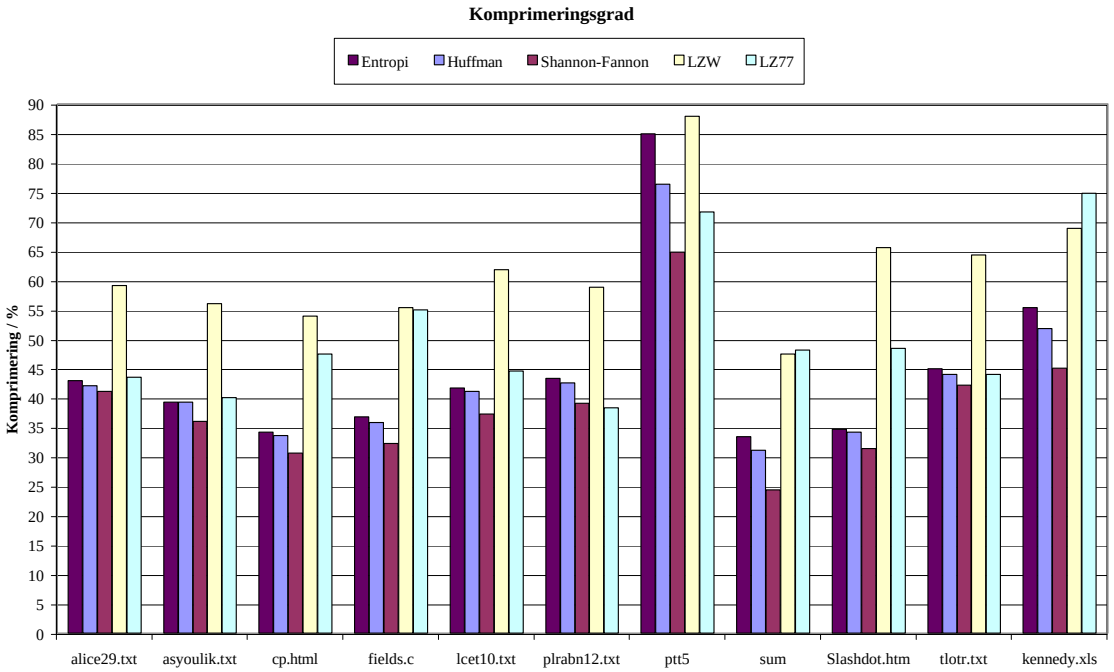


Figur 23: Den teoretiske komprimeringstid kontra den eksperimentelle for LZ77 algoritmen

7.6.4 Komprimeringsgraderne

Figur 24 viser et søjlediagram over algoritmernes komprimeringsgrader, på filerne fra *The Canterbury Corpus*. Søjlen for *entropien* viser den komprimeringsgrad som teoretisk set er mulig at opnå ifølge Shannon-Fanos informationsteori. Som det kan ses, ligger komprimeringsgraden for Huffman algoritmen utrolig tæt på *entropien*. Dette stemmer helt overens med teorien på side 36, der forudså, at Huffman algoritmen ville komprimere lige så godt eller en smule dårligere end *entropien*. Det ses også, at Shannon algoritmen altid komprimerer dårligere end Huffman, hvilket også var det forventede.

LZW er den algoritme, som generelt komprimerer bedst, hvilket skyldes, at LZW algoritmen kan gruppere hele sekvenser af tegn og erstatte dem med en lille bitkode, mens Huffman og Shannon kun kan erstatte et enkelt tegn med en mindre bitkode. LZ77 algoritmen komprimerer dog ret godt, faktisk næsten lige så godt som LZW. Dette passer godt med vores forventninger, da LZ77 er en forgænger for LZW algoritmen og derfor bygger på de samme underliggende principper.



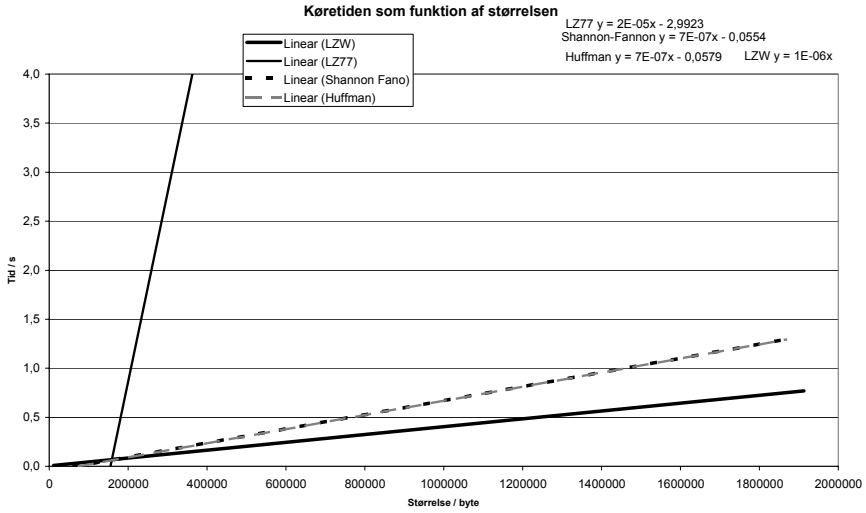
Figur 24: De teoretiske og eksperimentelle komprimeringsgrader for filerne fra *The Canterbury Corpus*

Generelt må vi vurdere, at de eksperimentelle komprimeringsgrader stemmer godt overens med teorien, og de vil derfor blive brugt direkte i modellen.

7.6.5 Den endelige model

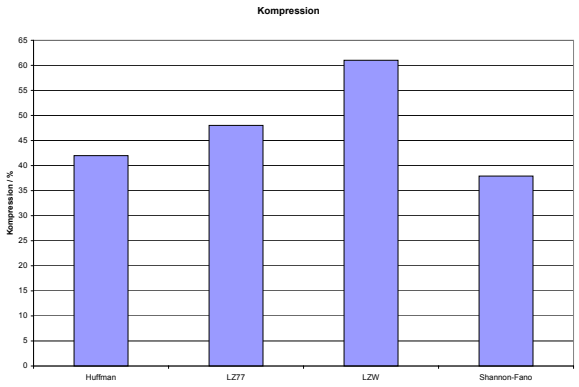
Da de eksperimentelle data ifølge analysen stemmer godt overnes med teorien, vil den endelige model primært bestå af resutaterne af disse. Dog har vi valgt at ændre de eksperimentelle resultater med hensyn til LZW, da algoritmen er for langsom på større filer på grund af en megabyte grænsen. Disse ændringer betyder, som det kan ses på graf 25, at LZW algoritmen bliver hurtigere til at komprimere større filer end både Huffman og Shannon algoritmerne. Dette stemmer ikke helt overens med vores forventninger, da Shannon og Huffman algoritmerne er kendt for at være hurtige, og da vores teoretiske udregninger forudså det modsatte. Vi skønner dog, at dette ikke bliver noget problem i modellen, da en algoritme som LZW er utrolig hurtig til at komprimere tekstfiler. Dette kan ses af formlen for LZW's teoretiske komprimeringshastighed, hvor antallet af operationer, som er nødvendige, for at LZW kan komprimere en fil, er stærkt afhængig af antallet af forskellige slags tegn, der findes i filen[16]. Dette betyder at LZW algoritmen vil være betydelig hurtigere til at komprimere tekstfiler end vores teoretiske beregninger antog, da tekstfiler typisk kun indeholder optil 127 forskellige slags tegn.

[16]Formlen som beskrives i afsnit 7.5.2 side 38 er som følger: $o = (256k + n)$, hvor o er antallet af operationer. k er kodebogens størrelse og n er den ukomprimerede fils størrelse.



Figur 25: Modellen over komprimeringstiderne for algoritmerne. Da linjerne for Huffman og Shanno algoritmerne ligger oven i hinanden, ser det ud til, der kun er 3 grafer

Til at beskrive komprimeringsgraden har vi valgt at tage den gennemsnitlige komprimeringsgrad for hver algoritme på de eksperimentelle data. Dette giver et søjlediagram (se graf 26), som nogenlunde beskriver, hvor meget de enkelte algoritmer vil komprimere en tilfældigt udvalgt tekstfil.



Figur 26: Modellen over komprimeringsgrader for algoritmerne

Modellen kan nu bruges til at forudsige, hvor hurtigt og effektivt algoritmerne kan komprimere en given fil; og herved bruges til at afklare, om der findes en optimal algoritme.

8 Ekspeirment

8.1 Test af modellen

Modellen er nu færdig og kan nu bruges til at beskrive hvor effektivt de forskellige algoritmer kan komprimere en tilfældigt udvalgt tekstfil. For at teste denne påstand har vi udvalgt en række filer, af samme type[17] som filerne i ”The Canterbury Corpus”. Derefter gentager vi eksperimentet på de nye filer under sammen betingelser, som eksperimentet blev udført på ”The Canterbury Corpus” (se evt. afsnit 7.2). Tilsidst kan vi så fastlægge i hvor høj grad vores model er i stand til at forudsige algoritmernes effektivitet over for de tilfældig udvalgte filer.

8.2 Tilfældigt udvalgte filer

Til test af vores model har vi udvalgt 35 filer opdelt i følgende grupper :

Type	Antal
HTML	8
Sparc-binær	8
Engelsk tekst	10
Excel-regneark	7

Filerne beskrives efter type i de følgende afsnit.

8.2.1 xls

De tilfældige udvalgte excel-filer består af forskellige rapporter fra gruppemedlemmernes tid i gymnasiet. Arkene indeholder en række resultater og en/flere grafer.

Filnavn	**Størrelse**	**Beskrivelse**
bio1.xls	35.840	Grafer fra et biologi-forsøg.
KPtitrereing	50.176	kemirapport, forsøgsdata og en graf.
Mappe3.xls	251.392	Resultater fra komprimering af Canterbury Corpus.
out.a20p.xls	51.712	Rå data fra komprimering af Canterburt Corpus.
saet17.xls	372.224	Fysikrapport, indeholder et stort sæt resultater.
syrestyrke.xls	40.960	Kemirapport, indeholder en række resultater.

8.2.2 html

Vi prøvet at finde sider der er så ”normale” som muligt. Hermed menes der at siderne indeholder både billeder, tabeller og tekst vægtet fornuftigt.

[17] HTML, xls, tekst og binære SPARC-filer

Filnavn	Størrelse	Navn	Kilde
berlingske.html	77.763	Berlinske tidenes forside	www.berlinske.dk
cnn_frontpage.html	73.694	CNN's forside	www.cnn.com
politiken.html	92.445	Politikens forside	wwww.politiken.dk
yahoo.html	31.326	Yahoos forside	www.yahoo.com
freshmeat.html	84.506	Freshmeat.net	www.freshmeat.net
linuxcom.html	21.268	Linux.com forside	www.linux.com
microsoftcom.html	17.266	Microsoft.com forside	www.microsoft.com

8.2.3 Engelsk tekster

Alle de brugte tekster er bøger. Her er både valgt filer indeholdende avanceret tekst, såsom "Hamlet", og filer med mindre avanceret tekst såsom "Hitchhiker's guide to the Galaxy"

Filnavn	Størrelse	Forfatter
2001 A Space Odissey.txt	375.942	Clarke, Arthur C
Hamlet.txt	200.075	Shakespeare, William
Hitchhiker's Guide to the Galaxy.txt	292.097	Adams, Douglas
Hobbit, The.txt	540.306	Tolkien, J.R.R.
Patriot Games.txt	1.163.119	Clancy, Tom
Robinson Crusoe.txt	642.698	Defoe, Daniel
In the Beginning was the Command Line	218.004	Stephenson, Neal
Neuromancer.txt	460.486	Gibson, William
Johnny Mnemonic.txt	37.920	Gibson, William

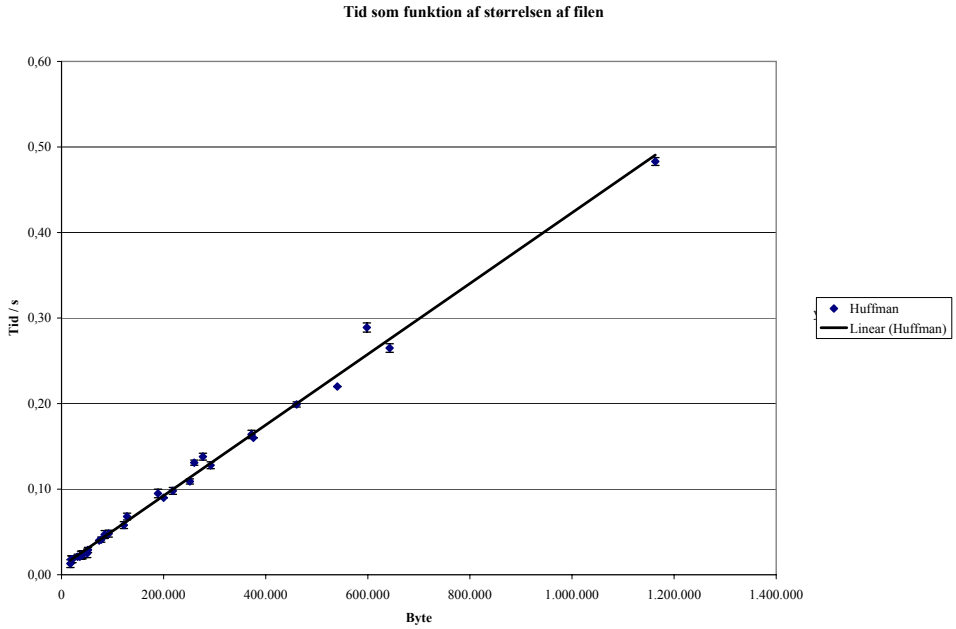
8.2.4 sparc

Alle filerne er taget fra tetex-distributionen til Sparc.

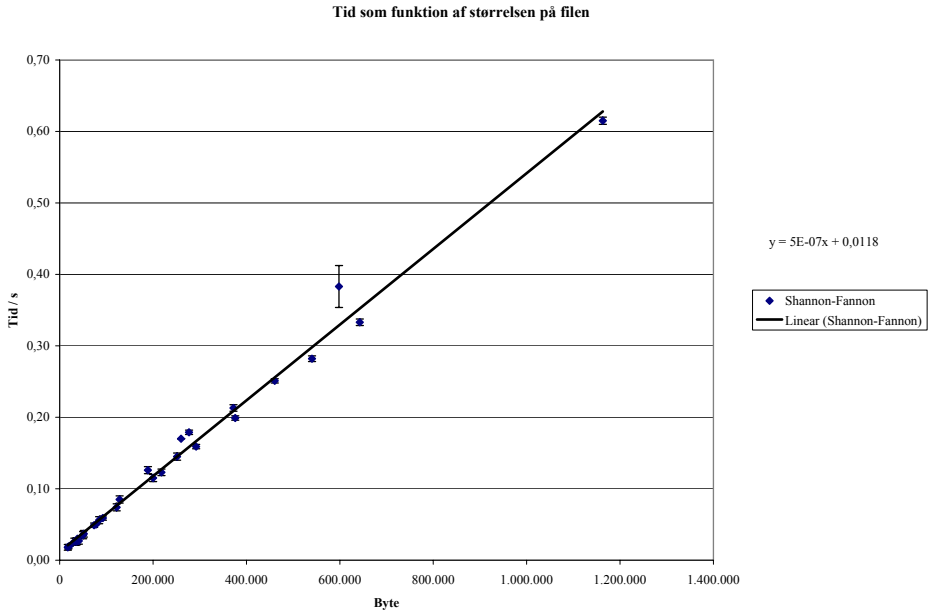
Filnavn	Størrelse
bibtex	128.292
dvips	189.020
makeindex	122.060
mf	259.992
pdftex	598.132
tex	277.000
texindex	18.856

8.3 Resultater

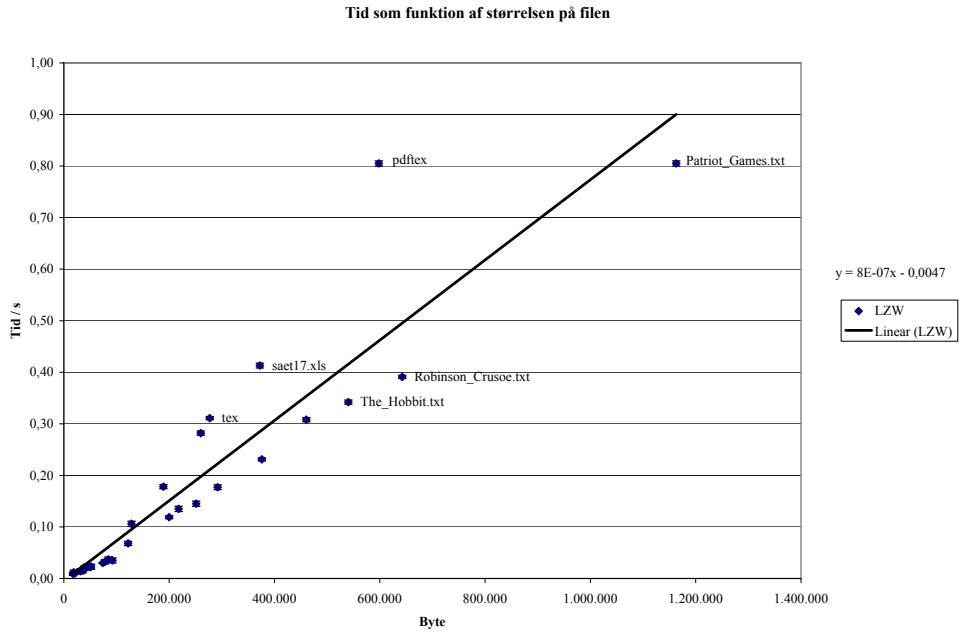
Nedenfor ses graferne for resultatet af eksperimentet. Som det kan ses, fordeler næsten alle punkterne sig pænt omkring gennmsnitslinjen for algoritmerne. Dog afviger filen *pdftex* en del fra gennemsnittet på grafen over LZW. Dette skyldes at LZW er yderst langsom til at komprimere binære filer, mens den er hurtig til komprimering af tekstfiler (Se evt. nederst afsnit 7.6.5.
Disse resultater vil blive diskuteret i sammenhold med modellen i næste afsnit.



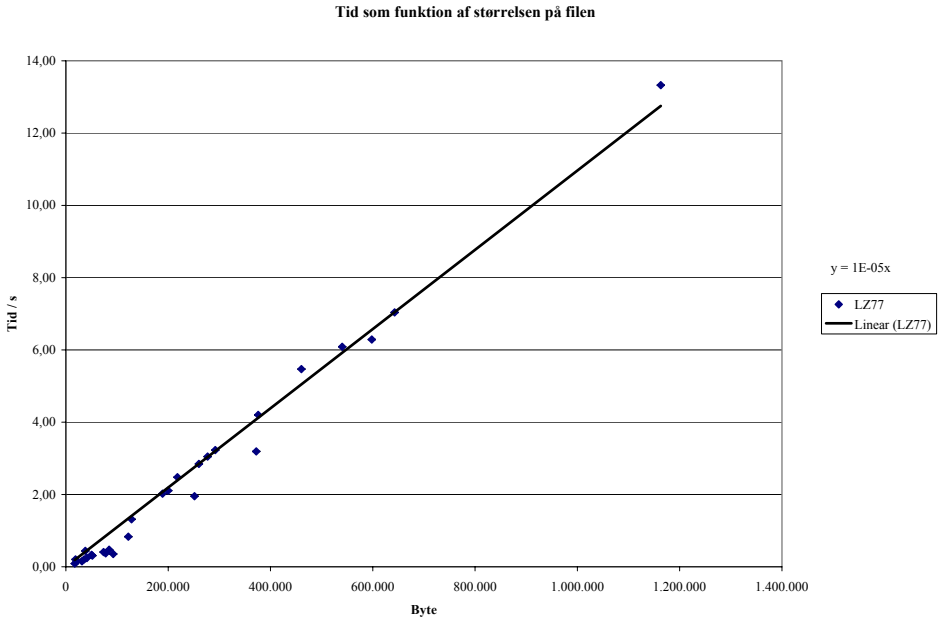
Figur 27: Graf over resultater fra test af Huffman-algoritmen



Figur 28: Graf over resultater fra test af Shannon-algoritmen



Figur 29: Graf over resultater fra test af LZW-algoritmen

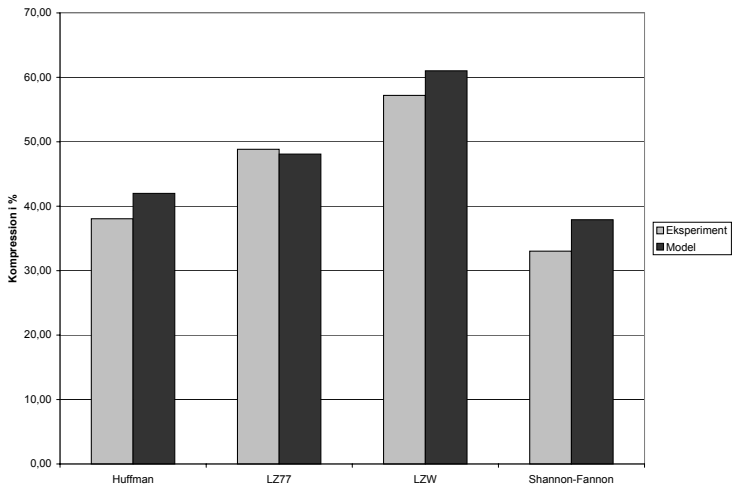


Figur 30: Graf over resultater fra test af LZ77-algoritmen

9 Diskussion

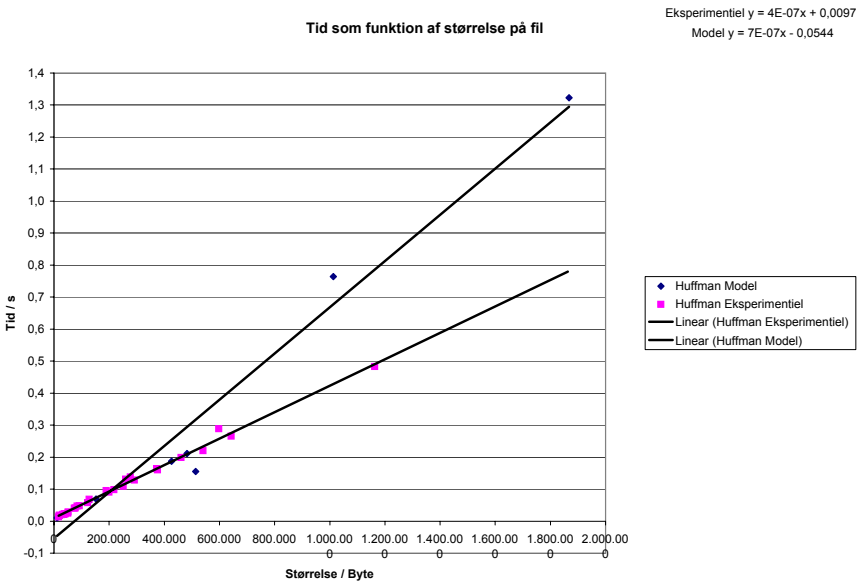
For at lave en reel diskussion af de eksperimentelle data kontra modellen, er vi nødt til at sætte dem i forhold til hinanden. Dette har vi valgt at gøre via graferne som kan ses nedenfor.

På graf 31 kan det ses, at vores model afspejler den eksperimentelle komprimeringsgrad forholdsvis godt. Dog er det en generel tendens at den eksperimentelle komprimeringsgrad ligger en smule under modellens. Dette udsving troede vi umiddelbart kunne skyldes de mange binære filer, da de typisk indeholder meget variation, hvilket gøre dem vanskelige at komprimere. Dette viste sig dog ikke at være tilfældet, da den samlede *entropi* for de binære excel filer er på 4,427, hvilket passer perfekt med gennemsnitsentropien for modellen som også lå på 4,427. Denne lave *entropi* betyder at Shannon og Huffman cirka vil komprimere filerne efter gennemsnittet. De binære filer repræsentere derfor ifølge modellen gennemsnitsfilen og har altså ikke medført udsvinget på graferne.
Vi kan desværre ikke finde nogen forklaring på dette udsving og tillægger det heller ikke den store betydning, da det kun drejer sig om 4-5 procentpoint.

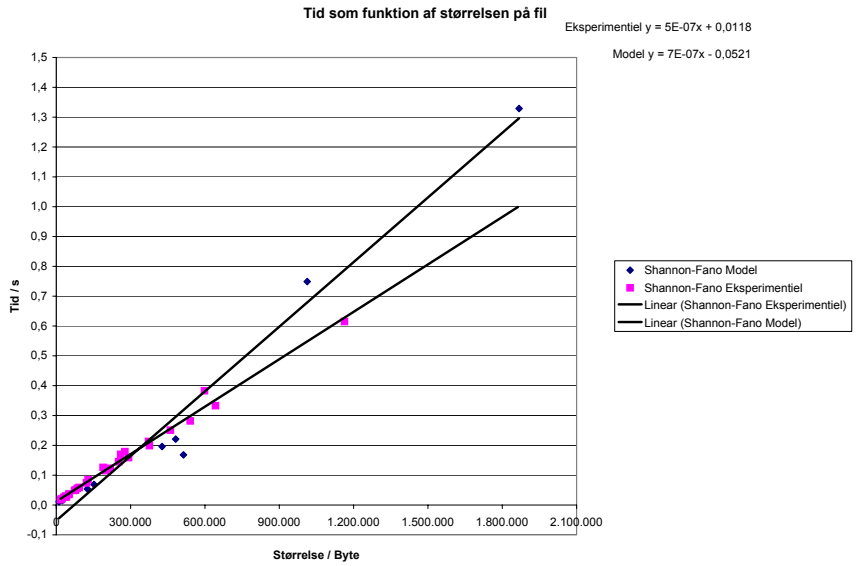


Figur 31: Model kontra gennemsnitlig eksperimental komprimeringsgrad

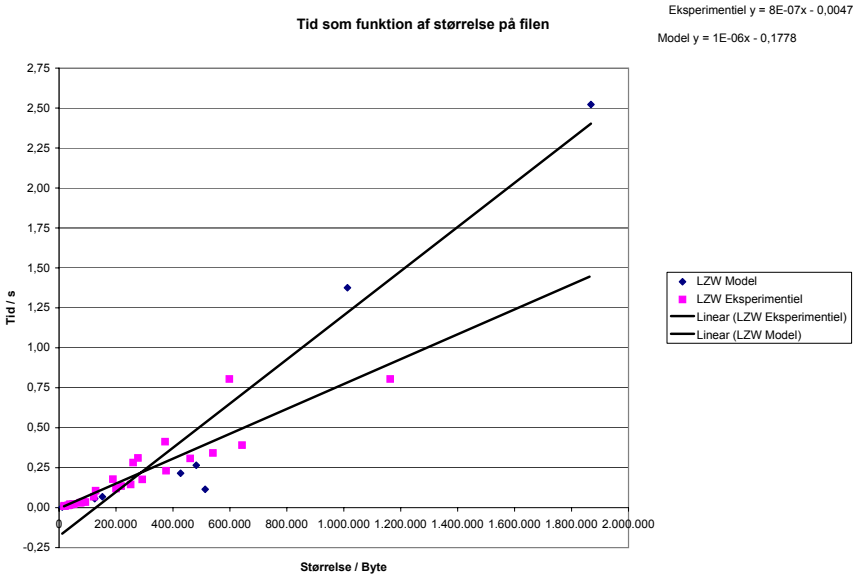
På graferne over komprimeringshastigheden kan man overordnet vurdere, at alle de eksperimentelle grafer cirka er forskubbet med den samme faktor i forhold til modellens forudsigelser. Den væsentligste årsag er, at de udvalgte filer til eksperimentet er ret små. Dette kan påvises i grafen for modellen, hvis der ses bort fra de to største filer. De andre punkter ville så danne en linje som stort set svarer til eksperimentets.
Hvis nogle af de udvalgte filer havde været større, ville modellen nok have stemt bedre overens med eksperimentet, da hukommelsesbegrænsninger (som tidligere omtalt) også ville have påvirket de eksperimentelle målinger.



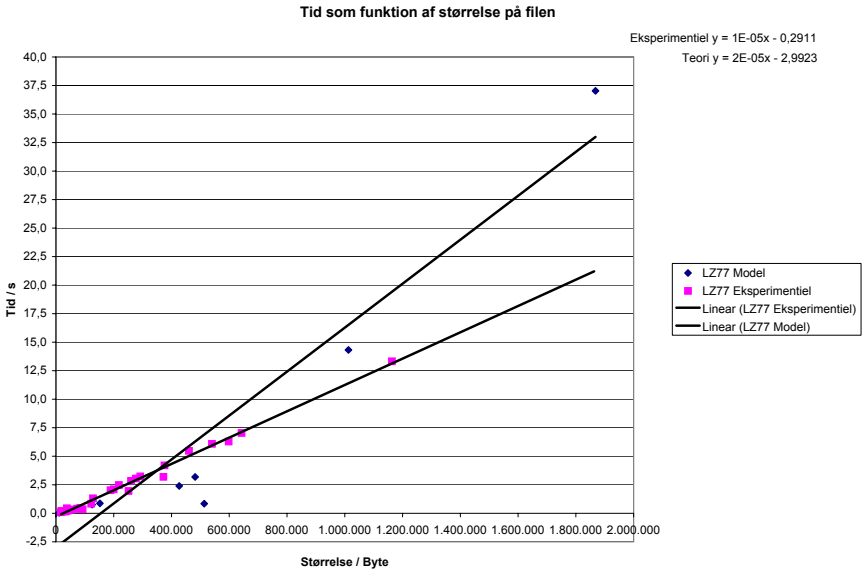
Figur 32: Model kontra eksperimentel komprimeringshastighed for Huffman.



Figur 33: Model kontra eksperimentel komprimeringshastighed for Shannon.

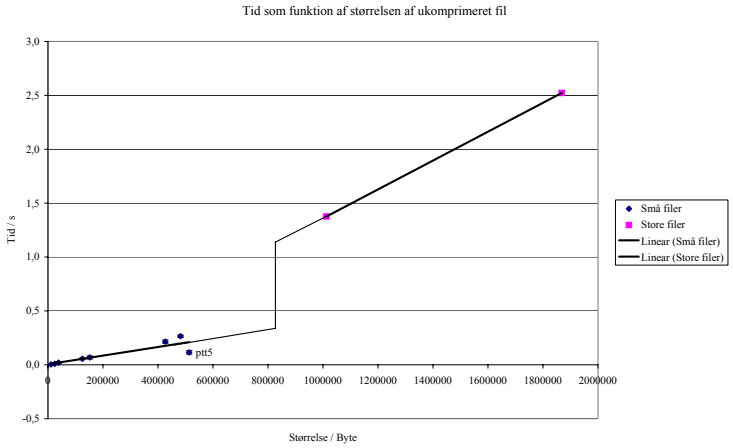


Figur 34: Model kontra eksperimentel komprimeringshastighed for LZW.



Figur 35: Model kontra eksperimentel komprimeringshastighed for LZ77.

Der kan med rette diskuteres om modellen ikke burde tage højde for, at filer som overskrider hukommelsesbegrænsningerne tager forholdsvis længere tid at komprimere end filer som ikke laver disse overskridelser. Et eksempel på dette er vist for LZW på graf 36, hvor vi har tilføjet et knæk på kurven. Desværre er denne grænse flydende og afhænger af styresystem, implementering og af den anvendte fil, og kan ikke inddrages i den endelig model.



Figur 36: Modificeret udgave af modellen for LZW

Generelt kan vi vurdere, at vores model har visse mangler, da det er begrænset hvad den kan forudsige og hvor nøjagtigt disse forudsigelser er. Vi mener dog modellen kan bruges til at lave vurderinger på mindre filer, da disse resultater ifølge graferne stemmer meget godt overens med de eksperimentelle data. Modellen begynder at afvige efter en grænse på cirka en megabyte, og dens anvendelighed på større filer er derfor begrænsede.

Udfra modellen og eksperimenterne er det umiddelbart let at finde frem til en optimal komprimeringsalgoritme med hensyn til komprimeringsgraden i forhold til hastigheden, da stortset alle LZWs resultater er gode. Dette er dog en sandhed med modifikationer, da LZW komprimerer binære filer utroligt langsom (Se evt. graf 29), og derfor ikke vil være velegnet til at komprimere visse former for tekst, så som de binære wordfiler.
Overordent må vi dog konkludere at LZW er den mest effektive algoritme.

10 Konklusion

Vi kan konkludere, at det var muligt at opstille en model over komprimeringstiden kontra komprimeringsgraden, selvom denne aldrig blev optimal på grund af mange fejlfaktorer.

Vi kan udfra modellen konkludere, at LZW er den mest effektive algoritme til komprimering af tekstfiler. Dette kan modellen dog kun sige er gældende for små filer.

11 Appendix

A Brugervejledning til zar.exe

Syntax :

zar.exe <*kommando*> [-<*switches*>] <input fil> [*output fil*]

zar.exe tager følgende parametre:

Kommandoer

c	Komprimer input-filen.
x	Dekomprimer input-filen.

Generelle switches

t	Udskriver tiden brugt på beregningerne.
y	Svare ja til alle eventuelle spørgsmål.

Komprimerings switches

sf	Benyt Shannon-Fano algoritmen (default).
h	Benyt Huffman algoritmen.
ac	Benyt aritmetisk komprimering.
77	Benyt LZ77 algoritmen.
lw	Benyt LWZ algoritmen.

Som default vil Zar komprimere inputfilen med Shannon-Fanon. Den komprimerede output-fil vil få efternavnet .zar. Ved dekomprimering vil filen få sit oprindelige efternavn med mindre andet er angivet.

B Entropi Udregning

Ofte er det svært at finde entropien for en hel tekst, specielt hvis teksten er lang. Da flere af de implementerede algoritmer's komprimeringsgrad kan forudsiges via entropien, benyttes der et program til at find entropien af en tekst. Dette program hedder *zarEntropi* og bruges via kommando-prompten. Der er to måder at benytte programmet på. Den første måde er på en enkelt tekst fil. Via prompten skriver man ´´zarEntropy" efterfulgt af et mellemrum og derefter filens navn. Programmet finder derefter entropien for den givne fil og gemmer dets resultater i en log fil. Log filen har samme navn som filen efterfulgt af ".log". Her er et eksempel: "zarEntropy mytext.txt", hvilket vil resultere i en log filen "mytext.txt.log". Desuden kan programmet bruges på flere filer ad gangen. Dette sker på samme måde som foroven bortset fra at filnavnene adskilles af et mellemrum. D.v.s. "zarEntropy mytext.txt mytext2.txt mytext3.txt"hvilket resultere i en log fil med navnet "zarEntropy.log"denne fil indeholder den samlede entropi for alle filerne.

C Forsøgsgang

Dette er en teknisk gennemgang af hele eksperimentet.

Følgende blev brugt under eksperimentet :

Computere

- Model : IBM Thinkpad A20p
- Processor : Intel Pentium III 700 Mhz

Operativsystem(Linux)

- Kerne-version : 2.2.19, standardcompilet fra Debians side.
- Distribution : Debian 2.2 [18]

Andre værktøjer

- PHP 4.0.3[19], compilet som cgi, brugt som scripting-værktøj
- GCC 1:2.95.2-13, brugt til kompilering af zar.

C.0.1 forsøgsgang (detaljeret)

For at minimere antallet af processer blev Linux-kernen booted med parameteret "single", dette gør at linux kun starter et minimum af processer, og kun logger en enkelt bruger ind. Se evt. figur 38 for en liste over kørende processer. Herved sikre vi os at der er så

```
arthur:~# ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:06 init [S]
    2 ?        SW     0:00 [kflushd]
    3 ?        SW     0:00 [kupdate]
    4 ?        SW     0:00 [kswapd]
    5 ?        SW     0:00 [keventd]
    6 ?        SW     0:00 [khubd]
  125 tty1     S      0:00 init [S]
  126 tty1     S      0:00 bash
```

Figur 37: Liste over kørende processer under eksperimentet

få fejlkilder som muligt. Derefter køres php-scriptet zartest.php[20] der komprimere hver enkelt fil med hver algoritme. Denne sekvens gentages 10 gange af scriptet.

[18] Se http://www.debian.org

[19] se http://www.php.net

[20] Scriptet kan ses i appendix D

D zartest.php

Følgende PHP[21]-script blev brugt til at udføre eksperimentet. Den viste version af scriptet er konfigureret til at køre de 4 algoritmerne (se linje 4-7) på ”The Canterbury Corpus” (se linje 11-23).

[21] Se http://www.php.net

zartest.php Page 1/1

```
#!/usr/bin/php -q
<?
#               Name               Arg
$algos =array("Shannon Fannon" => "sf",
              "Huffman" => "h",
              "LZW" => "lw",
              "L77" => "77",
#             "Arimetric" => "ar"
              );

$files = array("alice29.txt",
               "asyoulik.txt",
               "cp.html",
               "fields.c",
               "grammar.lsp",
               "kennedy.xls",
               "lcet10.txt",
               "plrabn12.txt",
               "ptt5",
               "Slashdot.html",
               "sum",
               "tlotr.txt",
               "xargs.1");

function parseResult($result){
#   Parse result and return a time/size-array
         preg_match("/: ([0-9\.]*) sec/m",$result,$time);
         $time = $time[1];                                  # get time
         preg_match("/\s*([0-9\.]*)%/",$result,$size);
         $size = $size[1];                                  # get size
         return(array("time" => $time, "size" => $size));
}

# run loop
print("Run,Filename,Algorithm,Time,Size\n");        # Print result-header
for($i=1; $i <= 10; $i++){                          # Run sequence 10 times
   while(list($key,$filename) = each($files)){      # Traverse files-array
      while(list($name,$arg) = each($algos)){       # Traverse algos-array
         $result = `./zar c -y -t $filename -$arg`; # Run zar and save output
         $presult = parseResult($result);           # Parse
         print("$i,$filename,$name,$presult[time],$presult[size]\n"); # print
      }
      reset($algos);
   }
   reset($files);
}
# done
?>
```

Figur 38: PHP-scriptet zartest.php der blev brugt under eksperimentet

E Ordforklaring

Symbolliste

ASCII står for ”American Standard Code for Information Interchange” dvs. en standart kode af 7 bit der repræsenterer hvert tegn brugt i de fleste computere.

Bit den mindste informationsenhed i en computer; står for resultatet af en ja/nej afgørelse, dvs. for talværdierne 0 eller 1, eller strøm eller ikke strøm. 8 bit udgør en byte.

Buffer Buffer er et midlertidigt lager, som oftest findes i hukommelsen.

Byte en enhed som består af en gruppe på normalt 8 bit, og som repræsenterer et tegn, fx et bogstav eller et tal, i en computer. Lagerkapaciteten i en computer måles i bytes.

Fil Selvom rapporten omhandler komprimering af tekstfiler, vil de blive beskrevet som filer. Den ikke komprimerede tekstfil vil blive kaldt “fil”. Den komprimerede tekstfil vil blive kaldt “komprimeret fil”. Også under komprimeringen.

Hexadecimal er et 16-tals system. Værdien af hvert tal er givet ved det normale talsystems første 10 tal og derefter alfabetets 5 første bogstaver gående fra 11 til 15 ($A = 10$ og $F = 15$). Hexadecimal er vældig praktisk når mennesker skal læse og omsætte binære tal, da 16-tals systemet går fint op i 2-tals systemet (eftersom $2^4 = 16$). Når man skriver hexadecimaltegn markeres det med $0x$ eller med et h efter tegnet. Et godt eksempel på hexadecimals brug er til at beskrive værdien af en byte, hvilket fylder præcist 2 hexadecimal tegn.

Kode Kode bliver brugt som begreb for de blokke komprimerings algoritmerne gemmer i, da disse ofte ikke fylder en normal byte.

Kodebog En kodebog indeholder en liste med symboler fra en kode samt deres betydning. Fungerer som en oversætter, som pakkealgoritmer enten bruger til at generere, eller oversætte koder til tegn-sekvenser. En kodebog kan være indeholdt i et kodetræ.

Køretid Den tid det tager et program at køre.

String (streng) En række af tegn hvilke computeren behandler som et. Dette kunne fx være en tekst.

Træ Et træ, ofte kaldet kodetræ i komprimering, er i datalogien en struktureret måde at sortere data på. I modsætning til et træ i den virkelige verden, vender et datalogisk træ på hovedet. Et træ består af grene og blade, hvor grenene forbinder bladene. Alle blade undtagen det øverste, dvs. start bladet i et træ, har en gren, der forbinder bladet opad. Et blad kan have et vilkårligt antal grene nedad. (se 1 for eksempel på kodetræ). I rapporten er alle kodetræer en form for kodebøger

Worst case Brugt i forbindelse med køretid for et program. Definition af det tilfælde hvor flest mulige løkker i et program udføres.

F Referencer

F.1 Tekster

[Cam97] The Canterbury Campus. *The Canterbury Corpus*. Canterbury, 1997. Webside kan findes på http://corpus.canterbury.ac.nz/credits.html

[Mia99] John Miano. *Compressed Image File Formats*. ACM Press, 1999.

[Rom97] Steven Roman. *Introduction to coding and information theory*. Springer-Verlag New York, 1997.

[Sal97] David Salomon. *Data Compression, The complete reference*. Springer-Verlag New York, 1997.

[Say00] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition edition, 2000.

[Top73] Flemming Topsøe. *Informationsteori*. Gyldendalske Boghandel, 1973.

[uni01] Berkeley university. *chapter 20 Big-O Notation*. Berkeley university, 2001. Webside kan findes på http://www.me.berkeley.edu/ e77/lecnotes/ch20/ch20.htm

[Way00] Peter Wayner. *Compression Algorithms for Real Programmers*. Morgan Kaufmann Publishers, 2000.

F.2 Figurer

Figurer

G Entropi for kennedy.xls

kennedy.xls.log Page 1/3

```
  0 ( ) : 456318
  1 ( ) : 82666
  2 ( ) : 37985
  3 ( ) : 159011
  4 ( ) : 3877
  5 ( ) : 1988
  6 ( ) : 1887
  7 ( ) : 989
  8 ( ) : 7256
  9 ( ) : 7264
 10 ( ) : 885
 11 ( ) : 939
 12 ( ) : 1040
 13 ( ) : 877
 14 ( ) : 885
 15 ( ) : 1151
 16 ( ) : 1052
 17 ( ) : 928
 18 ( ) : 1057
 19 ( ) : 1062
 20 ( ) : 869
 21 ( ) : 843
 22 ( ) : 840
 23 ( ) : 847
 24 ( ) : 842
 25 ( ) : 912
 26 ( ) : 907
 27 ( ) : 923
 28 ( ) : 869
 29 ( ) : 916
 30 ( ) : 921
 31 ( ) : 843
 32 ( ) : 823
 33 ( ) : 813
 34 ( ) : 853
 35 (#) : 1048
 36 ($) : 1048
 37 (%) : 853
 38 (&) : 814
 39 (') : 818
 40 (() : 865
 41 ()) : 865
 42 (*) : 823
 43 (+) : 812
 44 (,) : 813
 45 (-) : 814
 46 (.) : 815
 47 (/) : 865
 48 (0) : 1053
 49 (1) : 877
 50 (2) : 838
 51 (3) : 832
 52 (4) : 836
 53 (5) : 836
 54 (6) : 831
 55 (7) : 828
 56 (8) : 827
 57 (9) : 833
 58 (:) : 815
 59 (;) : 817
 60 (<) : 821
 61 (=) : 812
 62 (>) : 819
 63 (?) : 902
 64 (@) : 881
 65 (A) : 826
 66 (B) : 826
 67 (C) : 819
 68 (D) : 830
 69 (E) : 819
 70 (F) : 825
 71 (G) : 823
 72 (H) : 816
 73 (I) : 814
 74 (J) : 815
 75 (K) : 815
 76 (L) : 813
 77 (M) : 813
 78 (N) : 826
 79 (O) : 827
 80 (P) : 851
 81 (Q) : 969
 82 (R) : 829
 83 (S) : 812
 84 (T) : 825
 85 (U) : 811
 86 (V) : 812
 87 (W) : 810
 88 (X) : 814
 89 (Y) : 813
 90 (Z) : 693
 91 ([) : 579
 92 (\) : 582
 93 (]) : 585
 94 (^) : 585
 95 (_) : 689
 96 (`) : 592
 97 (a) : 597
```

kennedy.xls.log Page 2/3

```
 98 (b) : 1157
 99 (c) : 579
100 (d) : 585
101 (e) : 582
102 (f) : 579
103 (g) : 583
104 (h) : 583
105 (i) : 579
106 (j) : 583
107 (k) : 583
108 (l) : 580
109 (m) : 585
110 (n) : 580
111 (o) : 580
112 (p) : 585
113 (q) : 580
114 (r) : 580
115 (s) : 578
116 (t) : 577
117 (u) : 605
118 (v) : 577
119 (w) : 577
120 (x) : 581
121 (y) : 582
122 (z) : 577
123 ({) : 576
124 (|) : 583
125 (}) : 578
126 (~) : 578
127 (^?) : 577
128 (M-^@) : 603
129 (M-^A) : 576
130 (M-^B) : 577
131 (M-^C) : 576
132 (M-^D) : 580
133 (M-^E) : 577
134 (M-^F) : 577
135 (M-^G) : 577
136 (M-^H) : 677
137 (M-^I) : 578
138 (M-^J) : 578
139 (M-^K) : 578
140 (M-^L) : 579
141 (M-^M) : 577
142 (M-^N) : 576
143 (M-^O) : 577
144 (M-^P) : 576
145 (M-^Q) : 577
146 (M-^R) : 576
147 (M-^S) : 576
148 (M-^T) : 584
149 (M-^U) : 577
150 (M-^V) : 576
151 (M-^W) : 576
152 (M-^X) : 584
153 (M-^Y) : 576
154 (M-^Z) : 578
155 (M-^[) : 576
156 (M-^\) : 576
157 (M-^]) : 576
158 (M-^^) : 576
159 (M-^_) : 608
160 ( ) : 816
161 (¡) : 576
162 (¢) : 576
163 (£) : 576
164 (¤) : 583
165 (¥) : 576
166 (¦) : 576
167 (§) : 576
168 (¨) : 583
169 (©) : 576
170 (ª) : 576
171 («) : 576
172 (¬) : 576
173 (-) : 576
174 (®) : 576
175 (¯) : 576
176 (°) : 579
177 (±) : 577
178 (²) : 577
179 (³) : 578
180 (´) : 577
181 (µ) : 577
182 (¶) : 581
183 (·) : 640
184 (¸) : 578
185 (¹) : 576
186 (º) : 576
187 (») : 583
188 (¼) : 580
189 (½) : 579
190 (¾) : 578
191 (¿) : 578
192 (À) : 584
193 (Á) : 578
194 (Â) : 578
195 (Ã) : 616
```

kennedy.xls.log Page 3/3

```
196 (Ä) : 584
197 (Å) : 578
198 (Æ) : 578
199 (Ç) : 578
200 (È) : 615
201 (É) : 579
202 (Ê) : 579
203 (Ë) : 579
204 (Ì) : 578
205 (Í) : 578
206 (Î) : 578
207 (Ï) : 578
208 (Ð) : 604
209 (Ñ) : 577
210 (Ò) : 578
211 (Ó) : 576
212 (Ô) : 578
213 (Õ) : 577
214 (Ö) : 586
215 (×) : 578
216 (Ø) : 586
217 (Ù) : 578
218 (Ú) : 580
219 (Û) : 576
220 (Ü) : 581
221 (Ý) : 579
222 (Þ) : 578
223 (ß) : 578
224 (à) : 577
225 (á) : 577
226 (â) : 579
227 (ã) : 577
228 (ä) : 229
229 (å) : 228
230 (æ) : 228
231 (ç) : 228
232 (è) : 237
233 (é) : 237
234 (ê) : 251
235 (ë) : 230
236 (ì) : 230
237 (í) : 230
238 (î) : 229
239 (ï) : 229
240 (ð) : 235
241 (ñ) : 237
242 (ò) : 245
243 (ó) : 236
244 (ô) : 238
245 (õ) : 232
246 (ö) : 236
247 (÷) : 230
248 (ø) : 233
249 (ù) : 232
250 (ú) : 229
251 (û) : 230
252 (ü) : 230
253 (ý) : 243
254 (þ) : 238
255 (ÿ) : 238

Number of bytes in file: 1020744
The average entropy of the text is: 3.573471 bit.
The file can be compressed to: 459970 bytes.
```

H Kildekode til zar

De følgende sider indeholder kildekoden til programmet zar. Det skal bemærkes at det implementerede LZW-algoritme er patenteret, og må kun bruges til undervisningsformål.

```
//*****************************************************************************
**
// FILE:                Hoffman.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
***/
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <search.h>
#include <string.h>
#include "zarLib.h"
#include "main.h"


//*****************************************************************************
**
// Constants:

#define END_OF_STREAM   599


//*****************************************************************************
**
// Typedefs:

typedef struct NODE
{
        _dword  dwCount;
        int             nChar;
        _dword  dwBitCode;
        _dword  dwNumBits;
        struct  NODE*   pNext;
        struct  NODE*   pChild0;
        struct  NODE*   pChild1;
} NODE;


//*****************************************************************************
**
// Function prototypes:

static NODE* buildHuffmanTree(NODE* nodes, NODE* pStart);
static NODE* initHuffmanCompression(ZAR_FILE* pzSrc, NODE* nodes);
static NODE* initHuffmanExtraction(ZAR_FILE* pzSrc, NODE* nodes);
static int   loadHeader(ZAR_CHAR_FREQ* pzCharFreq, ZAR_FILE* pzFile);
static int   saveHeader(const NODE* nodes, ZAR_FILE* pzFile);
static void  assignCodes(NODE* pParent);
static void  sortNodes(NODE* nodes);
static int   nodesCompare(const void* pA, const void* pB);


//*****************************************************************************
**
// HuffmanExtract()

int HuffmanExtract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int             i, j=0;
        NODE*   root, *pStart, *pNode;
        NODE    nodes[600];

        // Print info:
        printf("--- Starting Huffman extraction ---\n");
        printf("Progress: ");

        pStart = initHuffmanExtraction(pzSrc, nodes);
```



```
        root = buildHuffmanTree(nodes, pStart);

        // build bit-codes:
        assignCodes(root);

        pNode = root;

        while (1)
        {
                // Get next bit:
                i = zarGetBit(pzSrc);

                if (i)
                // Go left:
                        pNode = pNode->pChild1;
                else
                // Go right:
                        pNode = pNode->pChild0;

                // Are we finished?
                if (pNode->pChild0 == NULL)
                {
                        // Have we reached end of stream?
                        if (pNode == pStart)
                                break;

                        zarSetByte(pzDst, pNode->nChar);
                        j++;

                        // Reset position:
                        pNode = root;
                        continue;
                }
        }

        return 1;
}

//*****************************************************************************
**
// HuffmanCompress()

int HuffmanCompress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int             i;
        NODE*   root, *pStart, *pCodes;
        NODE    nodes[600];

        // Print info:
        printf("--- Starting Huffman compression ---\n");
        printf("Progress: ");

        pStart = initHuffmanCompression(pzSrc, nodes);

        root = buildHuffmanTree(nodes, pStart);

        // build bit-codes:
        assignCodes(root);

        // Remove end code:
        nodes[END_OF_STREAM].dwBitCode = pStart->dwBitCode;
        nodes[END_OF_STREAM].dwNumBits = pStart->dwNumBits;
        pStart->dwNumBits = 0;
        pStart->dwCount   = 0;

        // Sort the nodes after code:
        pCodes = nodes +1;
        sortNodes(pCodes);

        // Write header:
```

```
        if (!saveHeader(pCodes, pzDst))
                return zarError(ERR_OUTPUT_FILE_NOT_WRITEABLE, pzDst);

        // Write the compressed output:
        while (EOF != (i = zarGetByte(pzSrc)))
                zarSetBits(pzDst, pCodes[i].dwBitCode, pCodes[i].dwNumBits);

        // Write end of stream bits:
        zarSetBits(pzDst, nodes[END_OF_STREAM].dwBitCode, nodes[END_OF_STREAM].d
wNumBits);
        zarSetBits(pzDst, 0, 8);        // Flush the last bits:

        return 1;
}


//***************************************************************************
**
// initHuffmanCompression()

static NODE* initHuffmanCompression(ZAR_FILE* pzSrc, NODE* nodes)
{
        int                             i, nLowest;
        ZAR_FREQ*               pzFreq;
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
        {
                zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                return NULL;
        }

        // Count the frequence of characters in the file:
        zarInitCharFreq(pzCharFreq);
        zarCountCharFreq(pzSrc, pzCharFreq);
        zarSortCharFreq(pzCharFreq);    // faster than qsort!

        pzFreq = pzCharFreq->freq;

        // Remove freqs with a zero-count:
        for (i=0; pzFreq[i].cxChar != 0 && i<256; i++);         // Don´t need an
y body!

        nLowest = 257 -i -1;

        // Convert to nodes and setup list:
        memset(nodes, 0, 600 *sizeof(NODE));

        for (i=1; i<257; i++)
        {
                // Add node:
                nodes[i].dwCount = pzCharFreq->freq[256 -i].cxChar;
                nodes[i].nChar  = pzCharFreq->freq[256 -i].ubChar;

                // Add to list:
                nodes[i].pNext = nodes +i +1;
        }

        // Add end of stream code:
        nodes[nLowest].pNext   = nodes +nLowest +1;
        nodes[nLowest].dwCount = 1;

        // Clean up:
        free(pzCharFreq);
        nodes[256].pNext = NULL;        // Last node doesn´t have a next node!

        // Return pointer to lowest node:
```



```
        return (nodes +nLowest);
}


//***************************************************************************
**
// initHuffmanExtraction()

static NODE* initHuffmanExtraction(ZAR_FILE* pzSrc, NODE* nodes)
{
        int                             i, nLowest;
        ZAR_FREQ*               pzFreq;
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
        {
                zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                return NULL;
        }

        // Get the frequence of characters from the file header:
        zarInitCharFreq(pzCharFreq);
        loadHeader(pzCharFreq, pzSrc);
        zarSortCharFreq(pzCharFreq);

        pzFreq = pzCharFreq->freq;

        // Remove freqs with a zero-count:
        for (i=0; pzFreq[i].cxChar != 0 && i<256; i++);         // Don´t need an
y body!

        nLowest = 257 -i -1;

        // Convert to nodes and setup list:
        memset(nodes, 0, 600 *sizeof(NODE));

        for (i=0; i<257; i++)
        {
                // Add node:
                nodes[i].dwCount = pzCharFreq->freq[256 -i].cxChar;
                nodes[i].nChar  = pzCharFreq->freq[256 -i].ubChar;

                // Add to list:
                nodes[i].pNext = nodes +i +1;
        }

        // Add end of stream code:
        nodes[nLowest].pNext   = nodes +nLowest +1;
        nodes[nLowest].dwCount = 1;


        // Clean up:
        free(pzCharFreq);
        nodes[256].pNext = NULL;        // Last node doesn´t have a next node!

        // Return pointer to lowest node:
        return (nodes +nLowest);
}



//***************************************************************************
**
// buildHuffmanTree()

static NODE* buildHuffmanTree(NODE* nodes, NODE* pStart)
{
```

```
        int     nNumNodes = 257;
        NODE*   low0 = pStart;
        NODE*   low1 = pStart->pNext;
        NODE*   tmp;
        NODE*   root = NULL;

        while (1)
        {
                // Add the lowest elements to node:
                nodes[nNumNodes].pChild0 = low0;
                nodes[nNumNodes].pChild1 = low1;
                nodes[nNumNodes].dwCount = low0->dwCount + low1->dwCount;

                // Remove the added nodes form list:
                low0 = low1->pNext;
                low1 = low0;

                // Test for end condition:
                if (low1 == NULL)
                {
                        root = nodes +nNumNodes;
                        break;
                }

                // Should we add the node to first element in list?
                if (low0->dwCount > nodes[nNumNodes].dwCount)
                {
                        low0 = nodes +nNumNodes;
                        low0->pNext = low1;
                }
                else
                {
                        // Find spot where to insert the created node in the lis
t:
                        do
                        {
                                tmp  = low1;
                                low1 = low1->pNext;
                        }
                        while (low1 && low1->dwCount < nodes[nNumNodes].dwCount)
;

                        // Insert new node:
                        (tmp->pNext = nodes +nNumNodes)->pNext = low1;
// same as tmp->pNext->pNext = low1, but faster on MS VC++ 6, because low1->pNex
t already load into register!
                }

                low1 = low0->pNext;

                nNumNodes++;
        }

        return root;
}


//***************************************************************************
**
// assignCodes()

static void assignCodes(NODE* pParent)
{
        if (pParent->pChild0 != NULL)
        {
                static NODE* pChild;

                // Add zero code to child0:
                pChild = pParent->pChild0;
                pChild->dwBitCode = pParent->dwBitCode <<1;
```



```
                pChild->dwNumBits = pParent->dwNumBits +1;

                // Move further down 0 brance:
                assignCodes(pChild);

                // Add one code to child1:
                pChild = pParent->pChild1;
                pChild->dwBitCode = (pParent->dwBitCode <<1) +1;
                pChild->dwNumBits = pParent->dwNumBits +1;

                // Move further down 1 brance:
                assignCodes(pChild);
        }
}


//***************************************************************************
**
// sortNodes()

static void sortNodes(NODE* nodes)
{
        qsort(nodes, 256, sizeof(NODE), nodesCompare);
}


//***************************************************************************
**
// nodesCompare()

static int nodesCompare(const void* pA, const void* pB)
{
        return (((const NODE*) pA)->nChar -((const NODE*) pB)->nChar);
}


//***************************************************************************
**
// saveHeader()

static int saveHeader(const NODE* nodes, ZAR_FILE* pzFile)
{
        int             j, i=0, nOk=1;

        // Start to write the file header:
        while (nOk)
        {
                // Find first entry to write::
                while (nodes[i].dwCount == 0)   // Find non-zero!
                {
                        if (++i == 255)
                        {
                                i = 255;
                                break;
                        }
                }

                // Write the index of the found entry:
                fputc(i, pzFile->pFile);

                // Find the end of the entry (3 zeros):
                j = i +1;

                do
                {
                        if (++j >= 256)
                        {
                                j   = 256;
                                nOk = 0;        // end loop
                                break;
```

```
                        }
                }
                while (nodes[j].dwCount != 0 || nodes[j +1].dwCount != 0 || node
s[j +2].dwCount != 0);


                // Store the ending index:
                fputc(j -1, pzFile->pFile);

                // Save the values between the indexs:
                if (i != 255)
                {
                        for (; i<j; i++)
                                fputc(nodes[i].dwCount, pzFile->pFile);
                }
        }

        return 1;
}

//*************************************************************************
**
// loadHeader()

static int loadHeader(ZAR_CHAR_FREQ* pzCharFreq, ZAR_FILE* pzFile)
{
        int i, from, to;
        ZAR_FREQ* pFreq = pzCharFreq->freq;

        // Extract the frequence from the header:
        while (1)
        {
                from = fgetc(pzFile->pFile);
                to   = fgetc(pzFile->pFile);

                if (from == 255)
                        break;

                // Fill the range up to i with zeros:
                for (i=from; i<=to; i++)
                        pFreq[i].cxChar = fgetc(pzFile->pFile);

                if (to == 255)
                        break;
        }

        // Read next byte:
        pzFile->nCurByte = zarGetByte(pzFile);
        pzFile->nBitMask = 0x80;                // Make sure not to read next by
te!

        return 1;
}
```

```
//*****************************************************************************
*******//
// FILE:                lz77.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
*******//
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <search.h>
#include <string.h>
#include "zarLib.h"
#include "main.h"


//*****************************************************************************
****
// Constants:

#define MIN_POINTER_BIT_SIZE            9               // in bits
#define MAX_POINTER_BIT_SIZE            16              // in bits

#define LOOK_AHEAD_BIT_SIZE                     4               // in bits
#define LOOK_AHEAD_SIZE                         (2 << (LOOK_AHEAD_BIT_SIZE -1))

#define LOOK_BACK_BIT_SIZE                      12
#define LOOK_BACK_SIZE                          (2 << (LOOK_BACK_BIT_SIZE -1))

#define MAX_BUFFER_SIZE                         (2 << 20)

#define WINDOW_SIZE                                     (LOOK_AHEAD_SIZE +LOOK_B
ACK_SIZE)

#define READ_CACHE                                      480


//*****************************************************************************
****
// Function prototypes:

static void             runCompression(_byte* pBuffer, _dword dwBufSize, int nUn
readData, ZAR_FILE* pzDst, const ZAR_FILE* pzSrc);
static void             runExtraction(_byte* pBuffer, _dword dwBufSize,  ZAR_FIL
E* pzDst, const ZAR_FILE* pzSrc);
static _dword   recacheBuffer(_byte* pBuffer, _byte** ppWndReadPos, _byte** ppWn
dBackPos, _dword dwAheadSize, const ZAR_FILE* pzSrc);
static void             resetBuffer(_byte* pBuffer, _byte** ppReadPos, int nSize
);

//*****************************************************************************
****
// lz77Compress()

int lz77Compress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        _dword  dwBufSize;
        _byte*  pBuffer;
        int             nUnreadData;

        // Print info:
        printf("--- Starting LZ77 compression ---\n");
        printf("Progress: ");

        // Get the file size of the file to compress:
        nUnreadData = zarFileSize(pzSrc);
        dwBufSize   = min(nUnreadData, MAX_BUFFER_SIZE);        // The number of
 bytes in the buffer:
        nUnreadData -= dwBufSize;
```



```
        // Allocate memory for the buffer:
        if (NULL == (pBuffer = malloc(dwBufSize)))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Read the data into the buffer:
        if (dwBufSize != zarGetBytes(pzSrc, pBuffer, dwBufSize))
        {
                free(pBuffer);
                return zarError(ERR_INPUT_FILE_NOT_READABLE, NULL);
        }

        // Compress the shit:
        runCompression(pBuffer, dwBufSize, nUnreadData, pzDst, pzSrc);  // stupi
d name, but VC++ has some bugs, which fuck functions with the same name!

        // Clean-up:
        free(pBuffer);

        return 1;
}

//*****************************************************************************
****
// lz77Extract()

int lz77Extract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        _dword  dwBufSize;
        _byte*  pBuffer;

        // Print info:
        printf("--- Starting LZ77 extraction ---\n");
        printf("Progress: ");

        // Init:
        dwBufSize = MAX_BUFFER_SIZE;

        // Allocate memory for buffer:
        if (NULL == (pBuffer = malloc(dwBufSize)))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        runExtraction(pBuffer, dwBufSize, pzDst, pzSrc);

        free(pBuffer);

        return 1;
}



//*****************************************************************************
****
// runCompression()

static void runCompression(_byte* pBuffer, _dword dwBufSize, int nUnreadData, ZA
R_FILE* pzDst, const ZAR_FILE* pzSrc)
{
        _byte* pBufEnd = pBuffer +dwBufSize;    // The end of the buffer (NEVER
go pass this point!)

        _byte*  pReadPos;               // The current position
(static)
        _byte*  pReadBack;              // The last position of the window
(static)
        _byte*  pReadAhead;             // The last position of the window  (sta
tic)

        _byte*  pFwdTmp;                // Forward pointer to search for match
```

```
        _byte*  pRwdTmp;                // Back pointer to search for match
        _byte*  pMatch;                 // The largest match found
        int     cxMatch;                // The count of the largest match
        int             cxCurMatch;             // The count of the current matc
h
        _dword  dwOut;

        // Init pointers:
        pReadBack       = pBuffer;
        pReadPos        = pBuffer;

        while (1)
        {
                // Init data:
                cxCurMatch = 0;
                pReadAhead = min(pReadPos +LOOK_AHEAD_SIZE -1, pBufEnd -1);

                do // Search for matches:
                {
                        // Init pointers:
                        pFwdTmp = pReadPos;
                        pRwdTmp = pReadBack;

                        cxMatch = 0;
                        pMatch  = pReadPos;     // smart, because (pReadPos -pMa
tch, gives index)

                        // Search up to the current position:
                        while (pRwdTmp < pReadPos)
                        {
                                // Search for match:
                                if (*pFwdTmp == *pRwdTmp)
                                {
                                        cxCurMatch = 0;

                                        do
                                        { // Go to next position:


                                                pRwdTmp++;
                                                pFwdTmp++;
                                                cxCurMatch++;

                                                if (pFwdTmp == pReadAhead || pRw
dTmp == pReadPos)
                                                        break;

                                        } while (*pFwdTmp == *pRwdTmp);

                                        // Did we find a larger match?
                                        if (cxCurMatch > cxMatch)
                                        {
                                                // Save the largest match:
                                                cxMatch = cxCurMatch;
                                                pMatch  = pRwdTmp -cxMatch;

                                                if (cxMatch == 15)
                                                        break;
                                        }

                                        pFwdTmp = pReadPos;
                                }

                                // Get the next letter:
                                pRwdTmp++;
                        }

                        // Output the current symbol:
                        zarSetByte(pzDst, pReadPos[cxMatch]);
```



```
                        // Output pointer to data (if any)
                        dwOut = ((pReadPos -pMatch) << 4) +cxMatch;
                        zarSetBytesF(pzDst, (_byte*) &dwOut, 2);        // Use f
unction version so this function will fit into the cache

                        // Update:
                        pReadPos   += cxMatch +1;
                        pReadAhead  = pReadPos +LOOK_AHEAD_SIZE -1;
                        pReadBack   = max(pBuffer, pReadPos -LOOK_BACK_SIZE +1);

                } while(pReadAhead < pBufEnd);

                // Should we read up more data?
                if (nUnreadData != 0)
                {
                        int nReadData;
                        int nTmp = pBufEnd -pReadPos;

                        nReadData = recacheBuffer(pBuffer, &pReadPos, &pReadBack
, nTmp, pzSrc);
                        pReadBack = pBuffer;
                        pBufEnd   = min(pBuffer +dwBufSize, pReadPos +nTmp +nRea
dData);
                        nUnreadData -= nReadData;
                }
                else
                {
                        // Have we raeched the end yet?
                        if (pReadPos < (pBufEnd -1))
                                pReadAhead = pBufEnd -1;
                        else
                                break;
                }
        }
}


//***************************************************************************
****
// recacheBuffer()

static _dword recacheBuffer(_byte* pBuffer, _byte** ppWndReadPos, _byte** ppWndB
ackPos, _dword dwAheadSize, const ZAR_FILE* pzSrc)
{
        _byte* pBack      = *ppWndBackPos;
        _byte* pRead      = *ppWndReadPos;
        _dword dwMoveSize = pRead -pBack +dwAheadSize;
        _byte* pGetPos    = pBuffer +dwMoveSize;

        // Move the window:
        memcpy(pBuffer, pBack, dwMoveSize);
        memset(pBuffer +dwMoveSize, 0, MAX_BUFFER_SIZE -dwMoveSize);

        // Reallocate pointers:
        *ppWndBackPos = pBuffer;
        *ppWndReadPos = pBuffer +LOOK_BACK_SIZE -1;

        // Read the last data:
        return zarGetBytesF(pzSrc, pGetPos, MAX_BUFFER_SIZE -dwMoveSize);
}

//***************************************************************************
****
// runExtraction()

static void runExtraction(_byte* pBuffer, _dword dwBufSize,  ZAR_FILE* pzDst, co
nst ZAR_FILE* pzSrc)
{
        _byte* pEndofCache, *pEndofBuffer;
        _byte  readCache[READ_CACHE], bChar;
```

```
        _byte* pReadPos, *pWritePos, *pFlush;
        _dword dwValue, dwBack, dwLen;
        _dword dwCacheSize = READ_CACHE;

        pWritePos        = pBuffer;
        pEndofBuffer = pBuffer +dwBufSize;
        pFlush      = pBuffer;

        while(dwCacheSize == READ_CACHE)
        {
                // Read up data:
                dwCacheSize = zarGetBytes(pzSrc, readCache, READ_CACHE);
                pReadPos    = readCache;
                pEndofCache = readCache +dwCacheSize;

                while (pReadPos < pEndofCache)
                {
                        // Extract the fucking byte:
                        dwValue = *((_dword*) pReadPos);
                        bChar   = (_byte) dwValue;
                        dwLen   = (dwValue >> 8)   & 0x0000000f;
                        dwBack  = (dwValue >> 12)  & 0x00000fff;


                        // Test for buffer recache:
                        if (pWritePos +dwLen +1 >= pEndofBuffer)
                        {
                                zarSetBytesF(pzDst, pBuffer, pWritePos -pBuffer)
;
                                resetBuffer(pBuffer, &pWritePos, LOOK_BACK_SIZE)
;
                                pFlush = pBuffer + (pWritePos -pBuffer);
                        }

                        // Write the output:
                        if (dwBack != 0)
                        {
                                memcpy(pWritePos, pWritePos -dwBack, dwLen);
                                pWritePos += dwLen;
                        }

                        // Write the byte:
                        *pWritePos = bChar;

                        // Update pointers:
                        pReadPos += 3;
                        pWritePos++;
                }
        }

        // Flush last data:
        if (pFlush)
                zarSetBytes(pzDst, pFlush, (pWritePos -pFlush));
}


//*************************************************************************
****
// resetBuffer()

static void resetBuffer(_byte* pBuffer, _byte** ppWritePos, int nSize)
{
        _byte* pos = *ppWritePos -nSize +1;

        // Move old memory:
        memcpy(pBuffer, pos, nSize);
        //memset(pBuffer +nSize, 0, MAX_BUFFER_SIZE -nSize);

        // Reallocate pointer:
        *ppWritePos = pBuffer +nSize -1;
```



```
}
```

```
//*****************************************************************************
**//
// FILE:                LZW.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
**//
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include "zarLib.h"


//*****************************************************************************
**
// Defines:

// The maximum bit-size the code-book can grow to:
#define MAX_CODEBOOK_SIZE                       24

// The maximum nodes that the code-book can contain:
#define MAX_NODES                                       (2 << (MAX_CODEBOOK_SIZE
 -1))


// The minimum bit-size of the code-book:
#define MIN_CODEBOOK_SIZE                       16


// The minimum number of nodes, which should be allocated:
#define MIN_NODES_ALLOC                         (2 << (MIN_CODEBOOK_SIZE -1))


// The maximum number of stacks:
#define NUM_STACKS                                      (MAX_CODEBOOK_SIZE -MIN_
CODEBOOK_SIZE)


//*****************************************************************************
**
// Typedefs:

typedef struct NODE
{
        int             nValue;       // int could be faster than _int
        _int            nCode;        // important must be 32 bit
        struct NODE*  pChild;
        struct NODE*  pNext;
        struct NODE*  pParent;
} NODE;

typedef struct CODE
{
        _pstr   pValue;
        int             nValueSize;
} CODE;


//*****************************************************************************
**
// Globals:

static int         g_nCodeSize;                 // The size of a code in bits (s
tarts at 9)
static _int        g_nNotesAtCodeSize;  // The number of notes at this codesize
static _int        g_cxNodes;                   // The count of nodes
static _int    g_nAllocNodes;           // The number of allocated nodes thus fa
r.
```



```
static _int        g_nStackPtr;

static NODE*   g_pCurNodeStack;
static NODE*   g_pNodeStacks[NUM_STACKS];

static CODE*   g_pCodeStack;
static _pstr   g_pCodeStrStack;
static _int        g_nAllocStrStack;
static _int    g_nCodeStrStackPtr;


//*****************************************************************************
**
// Function prototypes:

static int    initCompression(void);
static int        initExtraction(void);
static NODE*  getFreeNode(void);
static _pstr  getFreeCodeStr(int nSize);
static int    updateCodeSize(CODE** ppCode, CODE** ppLastNode);
static void   compress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst);
static void   addChild(NODE* pParentNode, int nValue);
static void   addSibling(NODE* pParentNode, int nValue);
static NODE*  findValue(NODE* pNode, int nValue);
static void   extract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst);
static int        getCodeValue(int nCode);


//*****************************************************************************
**
// lzwCompress()

int lzwCompress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        // Print info:
        printf("--- Starting LZW compression ---\n");
        printf("Progress: ");

        if (!initCompression())
                return 0;

        // Compress the file:
        compress(pzSrc, pzDst);

        return 1;
}


//*****************************************************************************
**
// lzwCompress()

int lzwExtract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        // Print info:
        printf("--- Starting LZW extraction ---\n");
        printf("Progress: ");

        if (!initExtraction())
                return 0;

        extract(pzSrc, pzDst);

        return 1;
}



//*****************************************************************************
**
```

```
// findCode()

static void compress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        NODE* pNode, *pLastNode;
        int       nValue, i;

        nValue = zarGetByte(pzSrc);

        while (nValue != EOF)
        {
                pLastNode = g_pNodeStacks[0] +nValue;
                pNode = pLastNode;

                do
                {
                        // Read next byte:
                        nValue = zarGetByte(pzSrc);

                        // Set pointer to next node:
                        pLastNode = pNode;
                        pNode = pLastNode->pChild;

                        if (pNode == NULL)
                        {
                                zarSetBitsF(pzDst, pLastNode->nCode, g_nCodeSize
);
                                addChild(pLastNode, nValue);
                                break;
                        }

                        pLastNode = pNode;

                        // Try to find the value:
                        pNode = findValue(pNode, nValue);

                        if (pNode == NULL)
                        {
                                zarSetBitsF(pzDst, pLastNode->pParent->nCode, g_
nCodeSize);
                                addSibling(pLastNode, nValue);
                                break;
                        }
                } while (nValue != EOF);
        }

        // Free stacks
        for (i=0; i<NUM_STACKS; i++)
                free(g_pNodeStacks[i]);
}


//*****************************************************************************
**
// findValue()

static NODE* findValue(NODE* pNode, int nValue)
{
        // Go through siblings:
        while (pNode != NULL)
        {
                // Did we find a match
                if (pNode->nValue == nValue)
                        return pNode;

                pNode = pNode->pNext;
        }

        return NULL;
}
```



```
//*****************************************************************************
**
// addChild()

static void addChild(NODE* pParentNode, int nValue)
{
        NODE* pChild;

        // Add a child:
        pParentNode->pChild = getFreeNode();
        pChild = pParentNode->pChild;

        if (pChild)
        {
                pChild->nValue  = nValue;
                pChild->nCode   = g_cxNodes;
                pChild->pParent = pParentNode;
        }
}


//*****************************************************************************
**
// addSibling()

static void addSibling(NODE* pParentNode, int nValue)
{
        NODE* pCousin;

        // Add a cousin?
        pCousin = pParentNode;

        while (pCousin->pNext != NULL)  // Find last cousin
                pCousin = pCousin->pNext;

        // Get free memory-block:
        pCousin->pNext = getFreeNode();
        pCousin = pCousin->pNext;

        if (pCousin)
        {
                // Update node data:
                pCousin->nValue = nValue;
                pCousin->nCode  = g_cxNodes;
                pCousin->pParent = pCousin->pParent;
        }
}


//*****************************************************************************
**
// initCompressTree()

static int initCompression(void)
{
        int             i;

        /* Initialize the members for the stack-memory */

        g_cxNodes       = 256;  // The first 256 nodes are already initialized!
        g_nCodeSize = 9;        // The size of the first codes

        g_nNotesAtCodeSize      = 2 << (g_nCodeSize -1);        // The number of
 notes available, before code size should increase

        // Zero all stacks:
        memset(g_pNodeStacks, 0, sizeof(NODE*) *NUM_STACKS);

        // Allocate memory for notes:
```

```
        g_nAllocNodes = MIN_NODES_ALLOC;

        if (NULL == (g_pNodeStacks[0] = calloc(g_nAllocNodes, sizeof(NODE))))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Initialize current stack:
        g_pCurNodeStack = g_pNodeStacks[0];
        g_nStackPtr = g_cxNodes;

        // Initialize the default nodes:
        for (i=0; i<256; i++)
                g_pCurNodeStack[i].nValue = g_pCurNodeStack[i].nCode = i;

        return 1;
}


//***************************************************************************
**
// initExtraction()

static int initExtraction(void)
{
        int i;

        g_cxNodes       = 256;
        g_nCodeSize = 9;

        g_nNotesAtCodeSize      = 2 << (g_nCodeSize -1);        // The number of
 codes available, before code size should increase.

        // Zero stack:
        g_pCodeStack            = NULL;
        g_pCodeStrStack     = NULL;

        // Allocate memory for codes:
        g_nAllocNodes    = MIN_NODES_ALLOC;
        g_nAllocStrStack = MIN_NODES_ALLOC;

        if (NULL == (g_pCodeStack = malloc(sizeof(CODE) *g_nAllocNodes)))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Allocate memory for strings:
        if (NULL == (g_pCodeStrStack = malloc(sizeof(char) *g_nAllocNodes)))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Initialize stacks:
        g_nStackPtr         = g_cxNodes;
        g_nCodeStrStackPtr      = 0;


        // Initialize the default codes:
        for (i=0; i<256; i++)
        {
                g_pCodeStack[i].nValueSize = 1;
                g_pCodeStack[i].pValue     = getFreeCodeStr(1);
                *g_pCodeStack[i].pValue    = (char) i;
        }

        return 1;
}


//***************************************************************************
**
// getFreeNode()

static NODE* getFreeNode(void)
{
        // Have we reached the limit of this code size?
```



```
        if (g_cxNodes == g_nNotesAtCodeSize)
        {
                // Have we reached maximum number of codes?
                if (g_nCodeSize != MAX_CODEBOOK_SIZE)
                {
                        g_nCodeSize++;
                        g_nNotesAtCodeSize <<= 1;

                        // Should we allocate some more memory?
                        if (g_cxNodes == g_nAllocNodes)
                        {
                                int nStackIdx = g_nCodeSize -MIN_CODEBOOK_SIZE;
                                g_nStackPtr = 0;

                                // Allocate memory for the new stack:
                                g_pNodeStacks[nStackIdx] = calloc(g_nAllocNodes,
 sizeof(NODE));
                                g_pCurNodeStack = g_pNodeStacks[nStackIdx];
                                g_nAllocNodes <<= 1;

                                if (g_pCurNodeStack == NULL)
                                {
                                        zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NU
LL);
                                        exit(0);
                                }
                        }
                }
                else {
                        return NULL;    // No code added.
                }
        }

        // Return next node:
        g_cxNodes++;
        return (g_pCurNodeStack +g_nStackPtr++);
}


//***************************************************************************
**
// getFreeCodeStr()

static _pstr getFreeCodeStr(int nSize)
{
        g_nCodeStrStackPtr += nSize;

        if (g_nCodeStrStackPtr >= g_nAllocStrStack)
        {
                int i;

                char* pOld = g_pCodeStrStack;
                g_nAllocStrStack <<= 1;

                g_pCodeStrStack = realloc(g_pCodeStrStack, sizeof(char) *g_nAllo
cStrStack);

                // Reallocate value pointers:
                for (i=0; i<g_nStackPtr; i++)
                {
                        g_pCodeStack[i].pValue = g_pCodeStack[i].pValue -pOld +g
_pCodeStrStack;
                }


                if (g_pCodeStrStack == NULL)
                {
                        zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                        exit(0);
                }
```

```
        }

        return (g_pCodeStrStack +g_nCodeStrStackPtr -nSize);
}


//***************************************************************************
**
// extract()

static void extract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int             nCode, nValSize;
        CODE*   pCode, *pLastCode;

        nCode = zarGetBits(pzSrc, g_nCodeSize);
        pCode = g_pCodeStack +nCode;

        while (1)
        {
                nCode = zarGetBits(pzSrc, g_nCodeSize);

                pLastCode = pCode;
                pCode     = g_pCodeStack +nCode;

                // Write value of code:
                nValSize = pLastCode->nValueSize;
                zarSetBytes(pzDst, pLastCode->pValue, nValSize);

                if (nCode == EOF)
                        break;

                // Have we reached the limit of this code size?
                if (++g_cxNodes == g_nNotesAtCodeSize)
                {
                        if (!updateCodeSize(&pCode, &pLastCode))
                                continue;
                }

                g_nStackPtr++;

                // Add code:
                g_pCodeStack[g_nStackPtr].pValue = getFreeCodeStr(nValSize +1);
                g_pCodeStack[g_nStackPtr].nValueSize = nValSize +1;
                memcpy(g_pCodeStack[g_nStackPtr].pValue, pLastCode->pValue, nVal
Size);
                g_pCodeStack[g_nStackPtr].pValue[nValSize] = g_pCodeStack[nCode]
.pValue[0];
        }

}


//***************************************************************************
**
// updateCodeSize()

static int updateCodeSize(CODE** ppCode, CODE** ppLastCode)
{
        // Have we reached maximum number of codes?
        if (g_nCodeSize != MAX_CODEBOOK_SIZE)
        {
                g_nCodeSize++;
                g_nNotesAtCodeSize <<= 1;

                // Should we allocate some more memory?
                if (g_cxNodes == g_nAllocNodes)
                {
                        CODE* pOld = g_pCodeStack;
```



```
                        g_nAllocNodes <<= 1;

                        g_pCodeStack = realloc(g_pCodeStack, sizeof(CODE) *g_nAl
locNodes);

                        *ppLastCode = *ppLastCode -pOld +g_pCodeStack;
                        *ppCode = *ppCode -pOld +g_pCodeStack;

                        if (g_pCodeStack == NULL)
                        {
                                zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                                exit(0);
                        }
                }

                return 1;
        }

        return 0;
}
```

```
//***************************************************************************
**
// FILE:                ShannonFano.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//***************************************************************************
***/
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <search.h>
#include <string.h>
#include "zarLib.h"
#include "main.h"


//***************************************************************************
**
// Constants:

#define END_OF_STREAM   256


//***************************************************************************
**
// Typedefs:

typedef struct
{
        _dword  dwCount;
        _int    ubChar;
        _dword  dwBitCode;
        _dword  dwNumBits;
} CODE;

typedef struct _NODE
{
        CODE*                   pCode;
        struct _NODE*   pChild0;
        struct _NODE*   pChild1;
} NODE;

typedef struct
{
        NODE  nodes[600];
        NODE* pCurNode;
        int       cxNodes;
} TREE;


//***************************************************************************
**
// Function prototypes:

static void freqToCode(const ZAR_CHAR_FREQ* pzCharFreq, CODE* nodes);
static void buildCompressCodes(CODE* pCode, int nNumCodes);
static void buildExtractTree(CODE* codes, int nNumCodes, TREE* tree);
static void sortCodes(CODE* codes);
static int  saveHeader(const CODE* codes, ZAR_FILE* pzFile);
static int  loadHeader(ZAR_CHAR_FREQ* zCharFreq, ZAR_FILE* pzFile);

static int  codesCompare(const void* pA, const void* pB);


//***************************************************************************
**
// ShannonFanoExtract()

int ShannonFanoExtract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
```



```
{
        int                             i, nEos;
        NODE*                   pNode;
        CODE                    codes[256 +1];
        TREE                    tree;
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Print info:
        printf("--- Starting Shannon-Fano extraction ---\n");
        printf("Progress: ");

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Get the frequence of characters from the file header:
        zarInitCharFreq(pzCharFreq);
        loadHeader(pzCharFreq, pzSrc);
        zarSortCharFreq(pzCharFreq);

        // Convert to nodes:
        freqToCode(pzCharFreq, codes);
        free(pzCharFreq);               // Get rid of unused memory.

        // Remove nodes with a zero-count:
        for (i=0; codes[i].dwCount != 0 && i<256; i++);         // Don´t need an
y body!

        nEos = i++;
        codes[nEos].dwCount = 1;


        // Build the tree:
        memset(&tree, 0, sizeof(TREE));
        tree.pCurNode = tree.nodes;

        buildExtractTree(codes, i, &tree);

        // Extract the file:
        pNode = tree.nodes;

        while (1)
        {
                // Get next bit:
                i = zarGetBit(pzSrc);

                if (i)
                // Go left:
                        pNode = pNode->pChild0;
                else
                // Go right:
                        pNode = pNode->pChild1;

                // Are we finished?
                if (pNode->pCode != NULL)
                {
                        // Have we reached end of stream?
                        if (pNode->pCode == (codes +nEos))
                                break;

                        zarSetByte(pzDst, pNode->pCode->ubChar);

                        // Reset pos:
                        pNode = tree.nodes;
                        continue;
                }
        }
```

```
        return 1;
}

//***************************************************************************
**
// ShannonFanoCompress()

int ShannonFanoCompress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int                             i, nEos;
        CODE                    codes[256 +1];
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Print info:
        printf("- - - Starting Shannon-Fano compression - - -\n");
        printf("Progress: ");

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Count the frequence of characters in the file:
        zarInitCharFreq(pzCharFreq);
        zarCountCharFreq(pzSrc, pzCharFreq);

        zarSortCharFreq(pzCharFreq);    // faster than qsort!

        // Convert to nodes:
        freqToCode(pzCharFreq, codes);
        free(pzCharFreq);                               // Get rid of unused mem
ory.


        // Remove nodes with a zero-count:
        for (i=0; codes[i].dwCount != 0 && i<256; i++);         // Don´t need an
y body!

        // Add end off stream (eos) symbol:
        nEos = i++;
        codes[nEos].dwCount = 1;

        // Build the compression tree!
        buildCompressCodes(codes, i);

        // Save end of stream code:
        codes[END_OF_STREAM].dwNumBits = codes[nEos].dwNumBits;
        codes[END_OF_STREAM].dwBitCode = codes[nEos].dwBitCode;
        codes[nEos].dwCount = 0;        // We don´t want to include this in the
saved header!


        // Sort the nodes after code:
        sortCodes(codes);

        // Write header:
        if (!saveHeader(codes, pzDst))
                return zarError(ERR_OUTPUT_FILE_NOT_WRITEABLE, pzDst);

        // Write the compressed output:
        while (EOF != (i = zarGetByte(pzSrc)))
                zarSetBits(pzDst, codes[i].dwBitCode, codes[i].dwNumBits);


        // Write end of stream bits:
        zarSetBits(pzDst, codes[END_OF_STREAM].dwBitCode, codes[END_OF_STREAM].d
wNumBits);
        zarSetBits(pzDst, 0, 8);        // Flush the last bits:
```



```
        return 1;
}


//***************************************************************************
**
// buildExtractTree()

static void buildExtractTree(CODE* pCode, int nNumCodes, TREE* pTree)
{
        int             i, j;
        NODE*   pNode = pTree->pCurNode;
        _dword  dwSum, dwSplitSum;

        // Insert node:
        pNode->pChild0 = pTree->nodes + (++(pTree->cxNodes));
        pNode->pChild1 = pTree->nodes + (++(pTree->cxNodes));

        // Left leaf:
        if (nNumCodes == 1)
        {
                // The left leaf:
                pCode->dwNumBits++;
                pCode->dwBitCode <<= 1;
                pCode->dwBitCode |=  1; // Add 1 to the end of the bits.

                // Insert left node:
                pNode->pChild0->pCode = pCode;
                return;
        }


        // Left and right leaf:
        if (nNumCodes == 2)
        {
                // The left leaf (add 1):
                pCode[0].dwNumBits++;
                pCode[0].dwBitCode <<= 1;
                pCode[0].dwBitCode |=  1; //// Add 1 to the end of the bits.
                pNode->pChild0->pCode = pCode;

                // The right leaf (add 0):
                pCode[1].dwNumBits++;
                pCode[1].dwBitCode <<= 1;
                pNode->pChild1->pCode = pCode +1;

                return;
        }


        // Get the sum of the char counts:
        for (i=0, dwSum=0; i<nNumCodes; i++)
                dwSum += pCode[i].dwCount;

        // The value for the split between left and right brance:
        dwSplitSum = dwSum /2;

        // Find the place to spilt:
        for (i=0, dwSum=0; ; i++)
        {
                dwSum += pCode[i].dwCount;

                // Add 1 bit:
                pCode[i].dwNumBits++;
                pCode[i].dwBitCode <<= 1;
                pCode[i].dwBitCode |= 1;

                // Have we reached the spilt point?
                if (dwSum >= dwSplitSum)
                {
```

```
                        // Walk down left brance:
                        i++;
                        pTree->pCurNode = pNode->pChild0;
                        buildExtractTree(pCode, i, pTree);

                        // Add 0 bit to all right brances:
                        for (j=i; j<nNumCodes; j++)
                        {
                                pCode[j].dwNumBits++;
                                pCode[j].dwBitCode <<= 1;
                        }

                        // Walk down right brance:
                        pTree->pCurNode = pNode->pChild1;
                        buildExtractTree(pCode +i, nNumCodes -i, pTree);

                        // Quit loop:
                        break;
                }
        }
}


//***************************************************************************
**
// buildTree()

static void buildCompressCodes(CODE* pCode, int nNumCodes)
{
        int             i, j;
        _dword  dwSum, dwSplitSum;

        // Left leaf:
        if (nNumCodes == 1)
        {
                // The left leaf:
                pCode->dwNumBits++;
                pCode->dwBitCode <<= 1;
                pCode->dwBitCode |=  1;         // Add 1 to the end of the bits.
                return;
        }


        // Left and right leaf:
        if (nNumCodes == 2)
        {
                // The left leaf (add 1):
                pCode[0].dwNumBits++;
                pCode[0].dwBitCode <<= 1;
                pCode[0].dwBitCode |=  1;       // Add 1 to the end of the bits.

                // The right leaf (add 0):
                pCode[1].dwNumBits++;
                pCode[1].dwBitCode <<= 1;
                return;
        }


        // Get the sum of the char counts:
        for (i=0, dwSum=0; i<nNumCodes; i++)
                dwSum += pCode[i].dwCount;

        // The value for the split between left and right brance:
        dwSplitSum = dwSum /2;

        // Find the place to spilt:
        for (i=0, dwSum=0; ; i++)
        {
                dwSum += pCode[i].dwCount;
```



```
                // Add 1 bit:
                pCode[i].dwNumBits++;
                pCode[i].dwBitCode <<= 1;
                pCode[i].dwBitCode |=  1;

                // Have we reached the spilt point?
                if (dwSum >= dwSplitSum)
                {
                        // Walk down left brance:
                        i++;
                        buildCompressCodes(pCode, i);

                        // Add 0 bit to all right brances:
                        for (j=i; j<nNumCodes; j++)
                        {
                                pCode[j].dwNumBits++;
                                pCode[j].dwBitCode <<= 1;
                        }

                        // Walk down right brance:
                        buildCompressCodes(pCode +i, nNumCodes -i);

                        // Quit loop:
                        break;
                }
        }
}

//***************************************************************************
**
// saveHeader()

static int saveHeader(const CODE* codes, ZAR_FILE* pzFile)
{
        int             j, i=0, nOk=1;

        // Start to write the file header:
        while (nOk)
        {
                // Find first entry to write::
                while (codes[i].dwCount == 0)   // Find non-zero!
                {
                        if (++i == 255)
                        {
                                i = 255;
                                break;
                        }
                }

                // Write the index of the found entry:
                fputc(i, pzFile->pFile);

                // Find the end of the entry (3 zeros):
                j = i +1;

                do
                {
                        if (++j >= 256)
                        {
                                j   = 256;
                                nOk = 0;        // end loop
                                break;
                        }
                }
                while (codes[j].dwCount != 0 || codes[j +1].dwCount != 0 || code
s[j +2].dwCount != 0);


                // Store the ending index:
                fputc(j -1, pzFile->pFile);
```

```
                // Save the values between the indexs:
                if (i != 255)
                {
                        for (; i<j; i++)
                                fputc(codes[i].dwCount, pzFile->pFile);
                }
        }

        return 1;
}


//***************************************************************************
**
// loadHeader()

static int loadHeader(ZAR_CHAR_FREQ* pzCharFreq, ZAR_FILE* pzFile)
{
        int i, from, to;
        ZAR_FREQ* pFreq = pzCharFreq->freq;

        // Extract the frequence from the header:
        while (1)
        {
                from = fgetc(pzFile->pFile);
                to   = fgetc(pzFile->pFile);

                if (from == 255)
                        break;

                // Fill the range up to i with zeros:
                for (i=from; i<=to; i++)
                        pFreq[i].cxChar = fgetc(pzFile->pFile);

                if (to == 255)
                        break;
        }

        // Read next byte:
        pzFile->nCurByte = zarGetByte(pzFile);
        pzFile->nBitMask = 0x80;                // Make sure not to read next by
te!

        return 1;
}


//***************************************************************************
**
// sortNodes()

static void sortCodes(CODE* codes)
{
        qsort(codes, 256, sizeof(CODE), codesCompare);
}


//***************************************************************************
**
// freqToCode()

static void freqToCode(const ZAR_CHAR_FREQ* pzCharFreq, CODE* codes)
{
        int i;
        const ZAR_FREQ* pFreq = pzCharFreq->freq;

        memset(codes, 0, sizeof(CODE) *257);

        for (i=0; i<256; i++)
```



```
        {
                codes[i].ubChar  = (_int) pFreq[i].ubChar;
                codes[i].dwCount = pFreq[i].cxChar;
        }
}


//***************************************************************************
**
// nodesCompare()

static int codesCompare(const void* pA, const void* pB)
{
        return (((const CODE*) pA)->ubChar -((const CODE*) pB)->ubChar);
}
```

```
//***************************************************************************
**
// FILE:                zarError.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//***************************************************************************
***/
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include "zarLocal.h"


//***************************************************************************
**
// zarError()

int zarError(int nCode, const void* pData)
{
        switch (nCode)
        {
                case ERR_INPUT_FILE_NOT_READABLE:
                        if (pData == NULL)
                                printf("Error reading input file!\n");
                        else
                                printf("The file \"%s\" can not be opened!\n", (const char*
) pData);
                        return 0;

                case ERR_OUTPUT_FILE_NOT_WRITEABLE:
                        if (pData == NULL)
                                printf("The output file can not be accessed!\n");
                        else
                                printf("The output file \"%s\" can not be accessed!\n", (const
 char*) pData);
                        return 0;

                case ERR_OUTPUT_FILE_NOT_CREATEABLE:
                        printf("The output file \"%s\" can not be created!\n", (const char*) pD
ata);
                        return 0;

                case ERR_INPUT_FILE_NOT_VALID_ZAR_FILE:
                        printf("The input file is not a valid %s-file!\n", ZAR_PROGRAM_NAME);
                        return 0;

                case ERR_ARGS_WRONG_PARAMETER:
                        printf("Wrong parameter \"%s\"\n", (const char*) pData);
                        return 0;

                case ERR_ARGS_NO_SOURCE_FILE:
                        printf("The source-file is missing!\n");
                        return 0;

                case ZAR_ERR_UNKNOWN_INTERNAL:
                        printf("Unrecoverable internal error!\n");
                        return 0;

                case ZAR_ERR_NOT_VALID_FILENAME:
                        printf("\"%s\" is not a valid filename", (const char*) pData);
                        return 0;

                case ZAR_ERR_WRONG_ZAR_VERSION:
                        if (pData == NULL)
                                printf("This program is too old to handle the compressed file!\n")
;
                        else
                                printf("This program is too old to handle \"%s\", please update to a
 newer version!\n", (const char*) pData);
                        return 0;
```



```
                case ZAR_ERR_FILE_SEEK:
                        printf("The file \"%s\" can not be searched: File corrupted!\n", (const ch
ar*) pData);
                        return 0;

                case ZAR_ERR_FEATURE_NOT_IMPLEMENTED:
                        printf("The %s is currently not implemented!\n", (const char*) pDat
a);
                        return 0;

                case ZAR_ERR_FAILED_MEMORY_ALLOC:
                        printf("Couldn´t allocate the needed memory!\n");
                        return 0;

                case ZAR_ERR_MESSAGE:
                        printf("%s\n", (const char*) pData);
                        return 0;
        }

        return 0;
}
```

```
//*****************************************************************************
**
// FILE:                zarInfo.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
***/
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "zarLocal.h"


//*****************************************************************************
**
// Globals:

extern ZAR_INIT* g_pzarInit;


//*****************************************************************************
**
// zarPrintHelp()

void zarPrintHelp(void)
{
        zarPrintShortHelp();
}


//*****************************************************************************
**
// zarPrintShortHelp(void)

void zarPrintShortHelp()
{
        int   i;
        char  szBuf[16];

        zarPrintHeader();

#ifdef __UNIX__
        printf("List of frequently used commands and switches.  Type %s -? for more help.\n\n", ZAR_PRO
GRAM_NAME);
#else
        printf("List of frequently used commands and switches.  Type %s /? for more help.\n\n", ZAR_PRO
GRAM_NAME);
#endif

        // Print the usage and the examples:
        printf("Usage:  %s\n", ZAR_HELP_USAGE);
        printf("Example: %s\n", ZAR_HELP_EXAMPLE1);

        // Print commands:
        printf("\n<Commands>\n");
        for (i=0; i<g_pzarInit->nNumCommands; i++)
                printf(" %s: %s\n", g_pzarInit->pCommands[i].szCmdArg, g_pzarInit
->pCommands[i].pName);


        // Print general switches (if any):
        if (g_pzarInit->nNumGenSwitch > 0)
        {
                printf("\n<General switches>\n");

                for (i=0; i<g_pzarInit->nNumGenSwitch; i++)
                {
                        sprintf(szBuf, "%s:", g_pzarInit->pGenSwitch[i].szSwitch
Arg);
```



```
                        printf(" %-3s %s\n", szBuf, g_pzarInit->pGenSwitch[i].pNa
me);
                }
        }

        // Print switches for compression (if any):
        if (g_pzarInit->nNumComSwitch > 0)
        {
                printf("\n<Switches for compression>\n");

                for (i=0; i<g_pzarInit->nNumComSwitch; i++)
                {
                        sprintf(szBuf, "%s:", g_pzarInit->pComSwitch[i].szSwitch
Arg);
                        printf(" %-3s %s\n", szBuf, g_pzarInit->pComSwitch[i].pNa
me);
                }
        }
}



//*****************************************************************************
**
// zarPrintHeader()

void zarPrintHeader(void)
{
        char  szBuildDate[25];
        char* pPos;

        // Append a zero to the beginning of the date, if it´s below 10:
        strcpy(szBuildDate, ZAR_BUILD_DATE);

        if (NULL != (pPos = strstr(szBuildDate, " ")))          // Search for do
uble-space
                pPos[1] = '0';
        // Append zero

        printf("%s %i.%02i created by %s %s\n", ZAR_PROGRAM_NAME, ZAR_VERSION_MAJOR, ZA
R_VERSION_MINOR, ZAR_CREATOR, szBuildDate);
        printf("This program is freeware and may only be used for none commercial purposes.\n");
        printf("We are not taking responsibility for any damage this program might cause.\n\n");
}
```

```
//*****************************************************************************
**
// FILE:                zarIO.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
***/
// Headers:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "zarLocal.h"


//*****************************************************************************
**
// Typedefs:

typedef struct
{
        _dword   dwFileTag;             //      The tag for this file (== ZAR_FI
LE_TAG)
        _ushort  usVer;                 //  The version of the program
        _ushort  usInfo;                //  User data, can be used for any purpo
se
        _ushort  cxFileName;    //  The size of the file name in bytes:
} ZAR_FILE_HEADER;


// The file tag:
#define ZAR_FILE_TAG    0x72617A        // == 'zar';
#define BUFFER_SIZE             16384           // == 16 KB


//*****************************************************************************
**
// Globals:

_dword g_dwUpdateChunck = 0xffffffff;


//*****************************************************************************
**
// zarLoadFile

ZAR_FILE* zarLoadFile(_pcstr pFileName, _pcstr pReadMode)
{
        ZAR_FILE* pzarFile;

        // Open the file:
        FILE* pFile = fopen(pFileName, pReadMode);

        if (pFile == NULL)
                return NULL;


        // Allocate memory for file:
        pzarFile = (ZAR_FILE*) malloc(sizeof(ZAR_FILE));        // Cast required
 for C++
        if (pzarFile  == NULL)
        {
                fclose(pFile);
                return NULL;
        }

        // Init the data for the file:
        memset(pzarFile, 0, sizeof(ZAR_FILE));

        setvbuf(pFile, NULL, _IOFBF, BUFFER_SIZE);
```



```
        pzarFile->pFile   = pFile;
        zarFileSize(pzarFile);


        // Return the opened file:
        return pzarFile;
}


//*****************************************************************************
**
// zarLoadInputFile()

ZAR_FILE* zarLoadInputFile(_pcstr pFileName)
{
        ZAR_FILE* pzFile = zarLoadFile(pFileName, "rb");

        if (pzFile == NULL)
        {
                zarError(ERR_INPUT_FILE_NOT_READABLE, pFileName);
                return NULL;
        }


        // Read the fist byte:
        pzFile->nCurByte = 0;
        pzFile->nBitMask = 0;

        return pzFile;
}


//*****************************************************************************
**
// zarLoadOutputFile()

ZAR_FILE* zarLoadOutputFile(_pcstr pFileName, int nFlags)
{
        ZAR_FILE* pzFile;

        if (nFlags & ZAR_ASK_IF_EXISTS)
        {
                // See if the file already exists:
                FILE* pFile = fopen(pFileName, "rb");

                if (pFile != NULL)
                {
                        // Close the file:
                        fclose(pFile);


                        // Ask user if it´s ok to overwite the output file:
                        printf("The file \"%s\" does already exist, do you want to overwrite it (y/n)? "
, pFileName);
                        if ('y' != tolower(getc(stdin)))
                                return NULL;

                        // Print space (looks better):
                        printf("\n");
                }
        }


        // Load the file:
        pzFile = zarLoadFile(pFileName, "wb");

        if (pzFile == NULL)
        {
                zarError(ERR_OUTPUT_FILE_NOT_CREATEABLE, pFileName);
                return NULL;
        }
```

```
        pzFile->nBitMask = 0x80;

        return pzFile;
}


//***************************************************************************
**
// zarCloseFile()

void zarCloseFile(ZAR_FILE* pzarFile)
{
        if (pzarFile != NULL)
        {
                // Close file:
                if (pzarFile->pFile != NULL)    // Required, because fclose() ca
n´t handle NULL pointers.
                        fclose(pzarFile->pFile);

                // Free memory used by the structure:
                free(pzarFile);
        }
}


//***************************************************************************
**
// zarFlushData()

void zarFlushFileData(ZAR_FILE* pzFile)
{
        // Should some data be flushed:
        if (pzFile->nBitMask != 0x80)
        {
                fputc(pzFile->nCurByte, pzFile->pFile);
                pzFile->nBitMask = 0x80;
        }
}


//***************************************************************************
**
// getFileSize()

_dword zarFileSize(ZAR_FILE* pzFile)
{
        long lPos, lEof;

        // Save current position:
        lPos = ftell(pzFile->pFile);

        // Seek to the end of the file, so we can determinate the filesize:
        fseek(pzFile->pFile, 0L, SEEK_END);
        lEof = ftell(pzFile->pFile);

        // Save the size:
        pzFile->dwFileSize = (_dword) lEof;

        // Reset the file-pos, so we don´t mess things up!
        fseek(pzFile->pFile, lPos, SEEK_SET);

        return (_dword) lEof;
}


//***************************************************************************
**
// zarFileSizeFromPath()
```



```
_dword zarFileSizeFromPath(_pcstr pszFile)
{
        long  lEof;
        FILE* pFile = fopen(pszFile, "rb");

        if (pFile == NULL)
                return 0;

        // Find end of file:
        fseek(pFile, 0, SEEK_END);
        lEof = ftell(pFile);

        fclose(pFile);

        return (_dword) lEof;
}


//***************************************************************************
**
// zarGetFileNameInPath()

int zarGetFileNameInPath(_pcstr pszSrc, _pstr pszDst)
{
        _pcstr   pszFileName;

        // First try if this is a microsoft-path?
        pszFileName = strrchr(pszSrc, '\\');

        if (pszFileName == NULL)
        {
                // Then try a unix-path:
                pszFileName = strrchr(pszSrc, '/');

                if (pszFileName == NULL)
                {
                        // Then assume this is a drive:
                        pszFileName = strrchr(pszSrc, ':');

                        if (pszFileName != NULL)
                                pszFileName++;
                }
        }

        // Then it must be a filename without a path:
        if (pszFileName == NULL)
                pszFileName = pszSrc;

        // Copy the buffer:
        strcpy(pszDst, pszFileName);

        return 1;
}


//***************************************************************************
**
// zarSetFileExtension()

void zarSetFileExtension(_pstr pszSrc)
{
        _pstr pExt;

        // Find last '.':
        pExt = strrchr(pszSrc, '.');

        // Remove the old extension (if any):
        if (pExt != NULL)
                *pExt = '\0';
```

```
        // Add the new one:
        strcat(pszSrc, ZAR_FILE_EXTENSION);
}


//***************************************************************************
**
// zarSetFileHeader()

int zarSaveFileHeader(const ZAR_FILE* pzFile, _ushort usInfo, _pcstr pFileName)
{
        ZAR_FILE_HEADER zfHdr;

        // Fill in the header:
        zfHdr.dwFileTag  = ZAR_FILE_TAG;
        zfHdr.usVer              = (_ushort) ZAR_VERSION;
        zfHdr.usInfo     = usInfo;
        zfHdr.cxFileName = (_ushort) ((pFileName == NULL) ? 0 : strlen(pFileName
));

        // Find begining of file:
        if (0 != fseek(pzFile->pFile, 0, SEEK_SET))
                return zarError(ZAR_ERR_FILE_SEEK, pzFile);

        // Save header:
        if (1 != fwrite(&zfHdr, sizeof(ZAR_FILE_HEADER), 1, pzFile->pFile))
                return zarError(ERR_OUTPUT_FILE_NOT_WRITEABLE, NULL);

        // Save the file name of the source file (if any):
        if (pFileName != NULL)
                if (zfHdr.cxFileName != fwrite(pFileName, sizeof(char), zfHdr.cx
FileName, pzFile->pFile))
                        return zarError(ERR_OUTPUT_FILE_NOT_WRITEABLE, NULL);

        return 1;
}


//***************************************************************************
**
// zarLoadFileHeader()

int zarLoadFileHeader(const ZAR_FILE* pzFile, _ushort* pusInfo, _pstr pFileName)
{
        ZAR_FILE_HEADER zfHdr;

        // Read the header:
        if (1 != fread(&zfHdr, sizeof(ZAR_FILE_HEADER), 1, pzFile->pFile))
                return zarError(ERR_INPUT_FILE_NOT_READABLE, NULL);

        // Make sure this is a .zar file:
        if (zfHdr.dwFileTag != ZAR_FILE_TAG)
                return zarError(ERR_INPUT_FILE_NOT_VALID_ZAR_FILE, NULL);

        // Verify version:
        if (zfHdr.usVer > ZAR_VERSION)
                return zarError(ZAR_ERR_WRONG_ZAR_VERSION, NULL);

        // Save the info-member:
        if (pusInfo != NULL)
                *pusInfo = zfHdr.usInfo;

        // Save the filename:
        if (pFileName == NULL)
                fseek(pzFile->pFile, zfHdr.cxFileName, SEEK_CUR);       // Skip
filename
        else
        {
                // Get the fileName:
                if (zfHdr.cxFileName == 0)
```



```
                        strcpy(pFileName, "");          // No filename, se retur
n an empty string.
                else
                {
                        // Copy filename:
                        if (zfHdr.cxFileName != fread(pFileName, sizeof(char), z
fHdr.cxFileName, pzFile->pFile))
                                return zarError(ERR_INPUT_FILE_NOT_READABLE, NUL
L);

                        pFileName[zfHdr.cxFileName] = '\0';      // Append zero-t
erminator.
                }
        }

        return 1;
}

//***************************************************************************
**
// zarGetBitsF()

_dword zarGetBitsF(ZAR_FILE* pzFile, int nNumBits)
{
        _dword dwRet = 0;
        _dword dwMask = 1L << (nNumBits -1);

        while (dwMask != 0)
        {
                // Should we get a new byte?
                if (pzFile->nBitMask == 0)
                {
                        // Read a byte:
                        pzFile->nCurByte = fgetc(pzFile->pFile);
                        pzFile->dwByteCount++;

                        if ((pzFile->dwByteCount % g_dwUpdateChunck) == 0)
                                printf("*");

                        // Update mask to extract the first bit:
                        pzFile->nBitMask = 0x80;
                }

                if (pzFile->nCurByte & pzFile->nBitMask)
                        dwRet |= dwMask;

                // Update masks:
                dwMask >>= 1;
                pzFile->nBitMask >>= 1;
        }

        return dwRet;
}


//***************************************************************************
**
// zarGetBitF()

int zarGetBitF(ZAR_FILE* pzFile)
{
        int nVal;

        // Should we read a new byte?
        if (pzFile->nBitMask == 0)
        {
                // Read a byte:
                pzFile->nCurByte = fgetc(pzFile->pFile);
                pzFile->dwByteCount++;
```

```
                if ((pzFile->dwByteCount % g_dwUpdateChunck) == 0)
                        printf("*");

                // Update mask to extract the first bit:
                pzFile->nBitMask = 0x80;
        }

        // Update the mask to extract the next bit:
        nVal = pzFile->nBitMask;
        pzFile->nBitMask >>= 1;

        return (pzFile->nCurByte & nVal);
}


//***************************************************************************
**
// zarSetBits()

void zarSetBitsF(ZAR_FILE* pzFile, _dword dwBits, int nBitCount)
{
        _dword dwMask = 1L << (nBitCount -1);

        while (dwMask != 0)
        {
                // Add a one?
                if (dwMask & dwBits)
                        pzFile->nCurByte |= pzFile->nBitMask;

                // Update mask:
                pzFile->nBitMask >>= 1;

                // Should we store the data:
                if (pzFile->nBitMask == 0)
                {
                        fputc(pzFile->nCurByte, pzFile->pFile);

                        // Save the byte:
                        pzFile->dwByteCount++;

                        if ((pzFile->dwByteCount % g_dwUpdateChunck) == 0)
                                printf("*");

                        // Update:
                        pzFile->nBitMask = 0x80;
                        pzFile->nCurByte = 0;
                }

                dwMask >>= 1;
        }
}


//***************************************************************************
**
// zarSetBitF()

void zarSetBitF(ZAR_FILE* pzFile, int bit)
{
        if (bit)
                pzFile->nCurByte |= pzFile->nBitMask;


        // Update mask to next bit:
        pzFile->nBitMask >>= 1;


        // Should we store the data:
        if (pzFile->nBitMask == 0)
        {
```



```
                fputc(pzFile->nCurByte, pzFile->pFile);

                // Save the byte:
                pzFile->dwByteCount++;

                if ((pzFile->dwByteCount % g_dwUpdateChunck) == 0)
                        printf("*");

                // Update:
                pzFile->nBitMask = 0x80;
                pzFile->nCurByte = 0;
        }
}


//***************************************************************************
**
// zarGetByteF()

int zarGetByteF(const ZAR_FILE* pzFile)
{
        return fgetc(pzFile->pFile);
}


//***************************************************************************
**
// zarSetByteF()

void zarSetByteF(const ZAR_FILE* pzDst, int nByte)
{
        fputc(nByte, pzDst->pFile);
}


//***************************************************************************
**
// zarSetUpdateChunckF()

void zarSetUpdateChunckF(_dword dwChunckSize)
{
        g_dwUpdateChunck = dwChunckSize;

        if (g_dwUpdateChunck == 0)      // the library could crash, if g_dwUpdat
eChunck == 0!
                g_dwUpdateChunck = 0xffffffff;
}


//***************************************************************************
**
// zarSetBytesF()

void zarSetBytesF(const ZAR_FILE* pzDst, _pstr pBytes, int nNumBytes)
{
        fwrite(pBytes, sizeof(char), nNumBytes, pzDst->pFile);
}


//***************************************************************************
****
// zarGetBytesF()

_dword zarGetBytesF(const ZAR_FILE* pzSrc, void* pBuffer, int nNumBytes)
{
        return fread(pBuffer, sizeof(_byte), nNumBytes, pzSrc->pFile);
}
```

```
//****************************************************************************
**
// FILE:                zarMain.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//****************************************************************************
***/
// Headers:
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "zarLocal.h"


//****************************************************************************
**
// Globals:

const ZAR_INIT*  g_pzarInit;


//****************************************************************************
**
// zarInit()

void zarInit(const ZAR_INIT* pzarInit)
{
        // Validate parameters:
        assert(pzarInit != NULL);
        assert(pzarInit->nNumCommands > 0);
        assert(pzarInit->compressMain != NULL);

        g_pzarInit = pzarInit;
}


//****************************************************************************
**
// zarMain()

int zarMain(_pcstr* ppArgs)
{
        ZAR_TASK  zTask;

        // Validate     parameters:
        assert(ppArgs != NULL);

        // Extract the task:
        if (!processArgs(ppArgs, &zTask))
                return 0;

        return g_pzarInit->compressMain(&zTask);
}


//****************************************************************************
**
// zarClose()

void zarClose(void)
{
        g_pzarInit = NULL;
}
```

```
//*****************************************************************************
**
// FILE:                zarLocal.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
***/

// Headers:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "zarLocal.h"

//*****************************************************************************
**
// Globals:

extern const ZAR_INIT* g_pzarInit;


//*****************************************************************************
**
// freeInitData()


//*****************************************************************************
**
// verifyArgs()

int processArgs(_pcstr* ppArgs, ZAR_TASK* pzTask)
{
        int i, j;

        // Zero data:
        memset(pzTask, 0, sizeof(ZAR_TASK));


        //---------------------------------//
        // Printing:
        //---------------------------------//

        // Should we print help?
        if (ppArgs[1] == NULL || (0 == strcmp(ppArgs[1], "/?")) || (0 == strcmp(
ppArgs[1], "-?")))
        {
                zarPrintHelp();
                return 0;
        }

        // Print the header:
        zarPrintHeader();


        //---------------------------------//
        // Extract switches and commands:
        //---------------------------------//

        // Get commands:
        for (j=0; j<g_pzarInit->nNumCommands; j++)
        {
                // Search through the given commands and try to find a match:
                if (0 == strcmp(g_pzarInit->pCommands[j].szCmdArg, ppArgs[1]))
                {
                        // Add the command:
                        pzTask->dwCommand |= (1 << j);  // 2^j
                        break;
                }
        }
```



```
        // Did we find any valid commands:
        if (pzTask->dwCommand == 0)
                return zarError(ERR_ARGS_WRONG_PARAMETER, ppArgs[1]);


        // Get switches:
        for (i=2; ppArgs[i]!=NULL; i++)
        {
                // Is this a switch?
                if (ppArgs[i][0] == '-')
                {
                        // Go through all compression switches:
                        for (j=0; j<g_pzarInit->nNumComSwitch; j++)
                        {
                                // Try to find a switch:
                                if (0 == strcmp(g_pzarInit->pComSwitch[j].szSwit
chArg, ppArgs[i] +1))
                                {
                                        // Add the switch:
                                        pzTask->dwCompressSwitch |= (1 << j);
// 2^j
                                        break;
                                }
                        }

                        // Go through all general switches:
                        for (j=0; j<g_pzarInit->nNumGenSwitch; j++)
                        {
                                // Try to find a switch:
                                if (0 == strcmp(g_pzarInit->pGenSwitch[j].szSwit
chArg, ppArgs[i] +1))
                                {
                                        // Add the switch:
                                        pzTask->dwGeneralSwitch |= (1 << j);
// 2^j
                                        break;
                                }
                        }
                }

                // Then it must be a path:
                else
                {
                        // Is it the source-path?
                        if (pzTask->szSrc[0] == 0)
                                strcpy(pzTask->szSrc, ppArgs[i]);
                        else
                        if (pzTask->szDst[0] == 0)
                                strcpy(pzTask->szDst, ppArgs[i]);
                        else
                                return zarError(ERR_ARGS_WRONG_PARAMETER, ppArgs
[i]);
                }
        }

        // Did we find any source files?
        if (pzTask->szSrc[0] == 0)
                return zarError(ERR_ARGS_NO_SOURCE_FILE, NULL);

        return 1;
}
```

```
//*****************************************************************************
**
// FILE:                zarUtl.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//*****************************************************************************
***/
// Headers:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "zarLocal.h"


//*****************************************************************************
**
// Constants:

#define ZAR_SORT_CUTOFF         8


//*****************************************************************************
**
// Globals:

static clock_t g_clock;


//*****************************************************************************
**
// Inline functions or macros:

#if _MSC_VER && _MSC_EXTENSIONS

__inline int comp(ZAR_FREQ* pA, ZAR_FREQ* pB)
{
        return (pB->cxChar -pA->cxChar);
}

__inline void swap(ZAR_FREQ* pA, ZAR_FREQ* pB)
{
    ZAR_FREQ tmp;

    tmp = *pA;
    *pA = *pB;
    *pB = tmp;
}

#else   // Use macros to emulate inline functions ): ):

// Global needed for macro:
static ZAR_FREQ g_tmp;

// Macros:
#define swap(pA, pB)    \
        g_tmp = *pA;            \
        *pA   = *pB;            \
        *pB   = g_tmp;

#define comp(pA, pB)    \
        (int)(pB->cxChar -pA->cxChar)

#endif  // _MSC_VER && _MSC_EXTENSIONS


//*****************************************************************************
**
// Function prototypes:
static int  charFreqCompare(const void* pA, const void* pB);
```



```
static void shortSort(ZAR_FREQ* lo, ZAR_FREQ* hi);


//*****************************************************************************
**
// zarInitTime()

void zarInitTime()
{
        g_clock = clock();
}

//*****************************************************************************
**
// zarDumpTime()

long zarDumpTime()
{
        return (clock() -g_clock);
}


//*****************************************************************************
**
// zarGetNumSwitches()

int zarGetNumSwitches(_dword dwSwitches)
{
        int             nFound = 0;
        _dword  dwMask = 0x10000000;

        while (dwMask)
        {
                if (dwSwitches & dwMask)
                        nFound++;

                dwMask >>= 1;
        }

        return nFound;
}



//*****************************************************************************
**
// zarInitCharFreq()

void zarInitCharFreq(ZAR_CHAR_FREQ* pzCharFreq)
{
        int i;
        ZAR_FREQ* pFreq = pzCharFreq->freq;

        // Zero counts:
        memset(pzCharFreq, 0, sizeof(ZAR_CHAR_FREQ));

        // Fill in the charecters:
        for (i=0; i<256; i++)
                pFreq[i].ubChar = (_ubyte) i;
}


//*****************************************************************************
**
// zarCountCharFreq()

void zarCountCharFreq(const ZAR_FILE* pzFile, ZAR_CHAR_FREQ* pzCharFreq)
{
        int             nByte, i;
        long    lPos;
```

```
        _dword    dwMax, dwScale;
        FILE*     pFile = pzFile->pFile;        // Could be faster on some stupi
d compilers!
        ZAR_FREQ* pFreq = pzCharFreq->freq;     // Could be faster on some stupi
d compilers!

        lPos = ftell(pFile);

        while (EOF != (nByte = fgetc(pFile)))
                pFreq[nByte].cxChar++;


        // Get the maximum count, so we know if scaling is required!
        dwMax = pFreq->cxChar;

        for (i=1; i<256; i++)
        {
                if (pFreq[i].cxChar > dwMax)
                        dwMax = pFreq[i].cxChar;
        }

        if (dwMax > 256)
        {
                // The scale factor:
                dwScale = 1 + dwMax /256;

                // Scale all values:
                for (i=0; i<256; i++)
                {
                        if (pFreq->cxChar != 0) // Don´t scale zero values:
                        {
                                pFreq->cxChar /= dwScale;

                                if (pFreq->cxChar == 0)
                                        pFreq->cxChar = 1;
                        }

                        pFreq++;
                }
        }

        // Seek to old pos of file:
        fseek(pFile, lPos, SEEK_SET);
}


//*************************************************************************
**
// zarSortCharFreq()

void zarSortCharFreq(ZAR_CHAR_FREQ* pzCharFreq)
{
        ZAR_FREQ        *lo, *hi;                       /* ends of sub-array curr
ently sorting */
    ZAR_FREQ    *mid;                   /* points to middle of subarray */
    ZAR_FREQ    *loguy, *higuy;         /* traveling pointers for partition step
*/
    _uint               size;
        ZAR_FREQ        *lostk[30], *histk[30];
    int stkptr;                                 /* stack for saving sub-array to
be processed */

    stkptr = 0;                                             /* initialize stack *
/

    lo = pzCharFreq->freq;
    hi = pzCharFreq->freq +255;

lbl_recurse:
```



```
    size = (hi -lo) +1;        /* number of el's to sort */

    /* below a certain size, it is faster to use a O(n^2) sorting method */
    if (size <= ZAR_SORT_CUTOFF)
            shortSort(lo, hi);
    else
        {
        mid = lo +(size /2);       /* find middle element */
        swap(mid, lo);             /* swap it to beginning of array */

        loguy = lo;
        higuy = hi +1;

        for (;;)
            {
                    do  {
            loguy++;
        } while (loguy <= hi && comp(loguy, lo) <= 0);

            do  {
            higuy--;
        } while (higuy > lo && comp(higuy, lo) >= 0);

            if (higuy < loguy)
            break;

            swap(loguy, higuy);
            }

        swap(lo, higuy);     /* put partition element in place */

        if (higuy - 1 - lo >= hi - loguy )
            {
            if (lo +1 < higuy)
                    {
                lostk[stkptr] = lo;
                histk[stkptr] = higuy -1;
                ++stkptr;
            }                           /* save big recursion for later */

            if (loguy < hi)
                    {
                lo = loguy;
                goto lbl_recurse;           /* do small recursion */
            }
        }
        else
            {
            if (loguy < hi)
                    {
                lostk[stkptr] = loguy;
                histk[stkptr] = hi;
                ++stkptr;               /* save big recursion for later */
            }

            if (lo +1 < higuy)
                    {
                hi = higuy -1;
                goto lbl_recurse;           /* do small recursion */
            }
        }
    }

    /* We have sorted the array, except for any pending sorts on the stack.
       Check if there are any, and do them. */

    --stkptr;
    if (stkptr >= 0)
        {
        lo = lostk[stkptr];
```

```
        hi = histk[stkptr];
        goto lbl_recurse;           /* pop subarray from stack */
    }
}


//*************************************************************************
**
// shortSort()

static void shortSort(ZAR_FREQ* lo, ZAR_FREQ* hi)
{
    ZAR_FREQ    *p;
        ZAR_FREQ        *max;

    while (hi > lo)
        {
        max = lo;

                for (p = lo +1; p <= hi; p++)
                {
                        if (comp(p, max) > 0)
                max = p;
        }

        swap(max, hi);
        hi--;
    }
}
```