HTML-KOMPRESSION

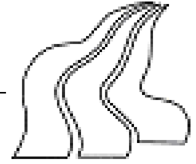
P1

Anders Rune Jensen
Jasper Kjersgaard Juhl
Ester Mungure
Michael Knudsen
Ole Laursen
Martin Qvist
Rasmus Jørgensen

December 2001

AALBORG UNIVERSITET

Det Teknisk-Naturvidenskabelige Fakultet
Aalborg Universitet



Basisuddannelsen

Titel:
HTML-kompression

Projektperiode:
P1,
9. oktober–18. december, 2001

Projektgruppe:
B333

Gruppemedlemmer:
Anders Rune Jensen
Jasper Kjersgaard Juhl
Ole Laursen
Martin Qvist
Ester Mungure
Rasmus Jørgensen
Michael Knudsen

Vejleder:
Thomas Nielsen

Bivejleder:
Claus Monrad Spliid

Antal kopier: 14

Rapport – sideantal: 64

Appendiks – sideantal: 11

Totalt sideantal: 79

Synopsis:

I denne rapport beskæftiger vi os med kompression af HTML. Dels fra en teoretisk synsvinkel i form af informationsteori, dels fra en mere konkret synsvinkel i form af en gennemgang af generelle algoritmer der kan bruges til at komprimere HTML-sider. Vi undersøger desuden om man kan udnytte nogle af egenskaberne ved HTML til at skabe en mere effektiv, specifik algoritme.

Grunden til at HTML-kompression er interessant, skyldes at mængden af internettrafik vokser støt mens udbyderne ønsker at spare penge på stadigt voksende forbindelser, og brugerne ønsker en bedre internetoplevelse. Kompression af HTML har potentialet til at hjælpe på dette fordi HTML-sider kan komprimeres effektivt, ligesom billeder, lyd, video og software der allerede komprimeres rutinemæssigt. Spørgsmålet er om potentialet er stort nok. Og om behovet er stort nok.

Forord

Denne rapport er udarbejdet som P1-opgave på Ålborg Universitets basisuddannelse. Den er resultat af 3 måneders arbejde med emnet kompression.

Det forudsættes at læseren har grundlæggende kenskab til visse tekniske aspekter af internettet, som f.eks. router, proxyserver, webserver og browser.

Rapporten må frit distribueres i en hvilken som helst form.

En særlig tak til vores vejledere Thomas Nielsen og Claus Spliid, som har været en stor hjælp gennem projektperioden.

God fornøjelse med rapporten

Jasper, Martin, Ole, Ester, Rasmus, Michael og Anders

Indhold

1 Indledning **5**
1.1 Problembeskrivelse 5
1.2 Problemformulering 6
1.3 Oversigt over projektet 6

2 Introduktion til kompression **7**

3 Behovsanalyse **9**
3.1 Brugerne 9
3.1.1 Undersøgelse af datasammensætning 10
3.1.2 Undersøgelse af overførselshastighed 11
3.1.3 Kompression af forbindelser 13
3.2 Udbyderne 13
3.2.1 Undersøgelse af datasammensætning 14
3.3 Sammenfatning 15

4 Informationsteori **17**
4.1 Entropi 17
4.2 Egenskaber ved entropi 18
4.2.1 Optimal kodning 19
4.2.2 Maksimal entropi ved ligelig fordeling 19
4.2.3 Afhængigheder reducerer entropi 19
4.3 Komprimering og entropi 20
4.3.1 HTML, informationsteoretisk set 21

5 Analyse af kompressionsmetoder **23**
5.1 Statistiske metoder 24
5.1.1 Præfikskoder 24
5.1.2 Huffman 25
5.2 Ordbogsbaserede metoder 30
5.2.1 Lempel-Ziv 77 30
5.2.2 Lempel-Ziv 78 34
5.2.3 Konklusion 36
5.3 Transformationer 37

5.3.1 Burrows-Wheeler-transformationen 37
5.3.2 Move-to-front 40
5.3.3 Konklusion 41

6 Udvælgelse af algoritme 43
6.1 Hvad kan komprimeres i HTML? 43
6.2 Dynamisk ordbog 44
6.2.1 Fremgangsmåde 44
6.2.2 Konklusion 45
6.3 Ændring af alfabetet 45
6.4 Opdeling af filerne 46
6.5 Vores algoritme 46

7 Implementering 48
7.1 Gennemgang af algoritme i pseudokode 48
7.2 Øvrige funktioner 50
7.3 Programmets grænseflade 51
7.4 Afvikling af programmet 51
7.5 Test 51
7.5.1 Komprimeringsevne 53
7.5.2 Tidsaspektet 54
7.5.3 Sammenfatning 54
7.5.4 Algoritmens tidskompleksitet 55
7.5.5 Konklusion 56

8 Praktiske overvejelser 57
8.1 Fordele og ulemper for udbyder 57
8.2 Brugerne 58
8.3 Eksisterende løsninger 59
8.4 Anvendelse af HTTP 1.1-kompression 59

9 Konklusion 61
9.1 Resultater fra rapporten 61
9.2 Perspektivering 62

A Eksempler 65
A.1 Eksempler fra statistiske metoder 65
A.1.1 Fortsat eksempel fra Huffman-kodning 65
A.2 Eksempler fra ordbogsbaserede metoder 67
A.2.1 Eksempel fra Lempel-Ziv 78 67
A.3 Eksempler fra transformationer 67
A.3.1 Eksempel fra Burrows-Wheeler 67

B Beskrivelse af undersøgelser **70**
B.1 Testsiderne 70
B.2 Datasammensætning i logfiler 71
B.2.1 Proxyserveren granit.but.auc.dk 71
B.2.2 IT-Avisen 72
B.3 Overførselstid for forskellige forbindelser 72
B.4 Test af kompressionshastighed 73

C Kompressionstest **74**

D Udbredelse af HTML-kompression **75**

E Telefoninterviews **76**
E.1 www.tv2.dk 76
E.2 www.dating.dk 77
E.3 www.ofir.dk 77

Kapitel 1

Indledning

1.1 Problembeskrivelse

Internet er verdens hurtigst voksende medie med et udbud af mange milliarder websider. Men forbindelserne til internet er begrænset i båndbredde, dvs. der er grænse for hvor meget data der kan sendes mellem to computere pr. sekund. At købe ekstra båndbredde er relativt dyrt – dels er det dyrt for virksomheder der driver websites, og dels er det dyrt for brugerne. Derfor er der behov for at udnytte disse forbindelser bedre, og det er her kompression kommer ind som en mulig løsning.

Kompression handler om at kode data for derved at formindske mængden af dem på en måde så de kan gendannes når de skal bruges. Denne teknik er allerede meget udbredt når det gælder billeder, lyd og film, men ikke for HTML-sider (der er de filer man læser når man surfer på internettet) på trods af at disse kan komprimeres meget effektivt. Men hvorfor bliver siderne så ikke komprimeret? Det kunne være fordi HTML-filerne udgør en for lille del af trafikken til at det kan betale sig at bruge kompression på dem.

Hvis man skal komprimere HTML-filer, hvilken algoritme skal man så bruge? Det kan ikke være en af de ellers meget effektive algoritmer med tab, som indebærer at filerne ikke kan genskabes til nøjagtigt det samme som før kompression; i modsætning til billeder og lyd kan man nemlig ved HTML-sider ikke tillade ændringer, da dette kunne lave om på layout og mening for udbydernes websider.

Men der findes stadig mange algoritmer der er tabsfri. Hvilken skal vi vælge, og hvis vi endelig finder en algoritme, findes der så de tekniske forudsætninger til at løsningen kan implementeres? Kan man overhovedet få folk til at bruge det? Dette afhænger selvfølgelig af hvor stor en evt. besparelse ville være, hvor godt teknologien kunne integreres med eksisterende praksis og hvorvidt den nødvendige viden er til stede hos udbyderne af disse websider, men dette er alle spørgsmål der bliver diskuteret i rapporten.

1.2 Problemformulering

Problemet er altså om det er en fordel at indføre kompression af HTML-sider over internettet for brugere og udbydere, og hvilken metode man i givet fald skulle bruge.

1.3 Oversigt over projektet

Vi starter med at undersøge hvad kompression af data er for noget for at få opbygget en fast referenceramme til resten af rapporten. Herefter behandler vi spørgsmålet om det overhovedet kan betale sig at indføre kompression af websider – bl.a. undersøger vi for de to hovedparter, brugerne og udbyderne, hvor stor en mængde data HTML-siderne udgør i forhold til den totale mængde data der overføres, dette inkluderer bla. billeder og lyd. Dette kan, ud over at indikere i hvor stor udstrækning det er relevant at komprimere HTML-filer, give en baggrund for at vurdere de algoritmer vi senere analyserer.

Herefter kommer vi ind på informationsteori som giver en teoretisk forståelse for og et overblik over hvorfor kompressionsalgoritmer virker.

Denne forståelse sætter os i stand til at foretage en undersøgelse af forskellige algoritmer – dette resulterer i en analyse og sammenligning af forskellige måder at komprimere på og en afvejning af deres styrker og svagheder i forbindelse med websidekompression. Sammen med de resultater vi har opnået ved behovsanalysen, forsøger vi derefter at vurdere en alternativ metode til at opnå bedre komprimering af websider. Dette følges op af en implementation der tillader os at foretage en test der kan holdes op mod gængse komprimeringsmetoder som `gzip` [10].

Vores praktiske test giver desuden et grundlag for at besvare spørgsmålet om de krav som brugerne og udbyderne måtte stille til en kompressionsløsning, kan opfyldes, og kan derfor hjælpe med at afklare om en kompressionsløsning overhovedet vil have nogen chance for blive anvendt hos henholdsvis brugerne og udbyderne. Sidste del af rapporten behandler dette emne.

Men vi starter med en grundlæggende undersøgelse af kompression.

Kapitel 2

Introduktion til kompression

Der er forskel på størrelsen af en datamængde og størrelsen af informationsindholdet i dataene. At *komprimere* handler om at formindske størrelsen af data uden at ødelægge informationsindholdet.

I rapporten opfatter vi data som bestående af en ordnet række af *symboler*, eller *tegn*, der via en kompressionsalgoritme gennemgår en *kodning* så de bliver til en ordnet række af *kodeord*, eller bare koder, der gerne samlet skulle være mindre end den oprindelige række.

Grunden til at man kan komprimere, hænger dels sammen med at man ofte har data liggende i en form som gør at det er hurtigere og/eller lettere at bruge dem – f.eks. bruger man som regel 8 bit til at repræsentere tegn, også selvom man reelt kun bruger måske 128 forskellige tegn som kunne lagres på 7 bit. Dels hænger det sammen med at de data vi almindeligvis beskæftiger os med, f.eks. tekst, normalt fylder mere end den information de viderebringer – der er *redundans* i dem.

Den første type af overflødigt data som beskrevet i ovenstående er relativt let at skære væk da det blot kræver at man i stedet for at optimere mht. hastighed, går over til at opbevare dataene mere kompakt. F.eks. kunne modtager og afsender enes om at bruge en *ordbog*, en fortegnelse over symboler med deres tilhørende kodeord, hvor der kun optræder de tegn der faktisk bruges så man kun behøvede f.eks. 5 bit til at repræsentere et tegn.

Det er imidlertid sværere at komme redundansen i dataene til livs – man kan grundlæggende gå to veje: enten kan man vælge en generel metode, som en af de der er beskrevet i kapitel 5, eller en metode som er optimeret til netop den type data man arbejder med. En undersøgelse af dataene giver naturligvis mulighed for bedre kompression og oftest sker der også en analyse af de data der ønskes komprimeret, inden den reelle kompression igangsættes, også ved de generelle metoder.

En uddybning og forklaring af redundansfænomenet gives i kapitel 4, men et kort eksempel med sekvenslængdekodning kan illustrere det.

Sekvenslængdekodning eller „run length encoding“, RLE, går ud på at erstatte gentagelser af samme symbol med et enkelt symbol samt antallet af forekomster – f.eks. kan „muuuuuuuuh“ erstattes med „m*8uh“ hvor stjernen repræsenterer et

kontroltegn som bruges ved afkodningen til at afgøre at næste tegn skal duplikeres et antal gange. Dette nedbringer datamængden fra 10 til 5 tegn, dvs. giver *kompressionsforholdet* 50% (kompressionsforhold er defineret som forholdet mellem størrelsen af de komprimerede data og størrelsen af de oprindelige data, f.eks. 1:2). Dette forhold er naturligvis ikke generelt for RLE som ikke er særligt velegnet til at komprimere almindelig tekst, men kan være effektiv i andre sammenhænge.

Det skal i øvrigt bemærkes at de generelle algoritmer ikke er så generelle at de kan komprimere *alle typer* data – kompression kan jo betragtes som en funktion der afbilder en given mængde data over i en værdimængde. Funktionen skal imidlertid være en bijektion hvis man altid skal kunne komme tilbage til de samme data, hvorfor værdimængden skal være lige så stor som definitionsmængden hvis alle elementer i definitionsmængden kan forekomme i dataene. Altså kan man ikke generelt gå fra f.eks. 8 bit til 7, ikke alle data kan komprimeres.

I nogle tilfælde af kompression kan man tillade et tab af data så kompression fulgt af dekompression ikke giver præcist de samme data. Dette sker ofte ved kompression af lyd eller billeder som man som regel kan beskrive med færre detaljer og nuancer uden at det behøver at være mærkbart. Derved reducerer man antallet af forskellige tegn så der kan opnås en bedre kompression; JPEG og MP3 er eksempler på dette.

Som tidligere nævnt vil vi dog i denne rapport beskæftige os med tabsfri kompression fordi udseendet og indholdet af siderne ikke må blive forstyrret. Der er dog to sider af HTML – det indhold som f.eks. forfatteren af dokumentet ser, koderne og teksten, og den formaterede fremstilling som brugerne af siderne ser i deres browser. Man kan argumentere for at små ændringer i HTML-koderne, f.eks. skift fra små bogstaver til store, der ikke giver nogen forskel i udseendet i browseren, ikke medfører noget tab. Vi overvejer dette i kapitel 6.

Efter denne korte introduktion til komprimeringsbegrebet følger en analyse af behovet for en kompression af websider.

Kapitel 3

Behovsanalyse

Udgangspunktet for denne rapport er at kompression af websider har et potentiale for udbredelse. Men hvad består potentialet egentligt af, og hvem gælder det? Formålet med behovsanalysen er at få klarlagt disse spørgsmål – især er det interessant om et evt. behov er stort nok til at kunne motivere en almen brug.

Man kan skelne mellem to parter: de almindelige brugere af websider og udbyderne, dem der ejer og driver de foreskellige websites. Den følgende analyse fokuserer på tekniske problemer – problemer som brugervenlighed og økonomiske forhold i forbindelse med at indføre en løsning ude i erhvervslivet bliver behandlet i kapitel 8.

3.1 Brugerne

For brugerne er det mest af alt et spørgsmål om at komprimerede sider kan give en hastighedsgevinst – også rent økonomisk er selve trafikmængden ikke så interessant fordi man som oftest betaler pr. tid og ikke pr. kb hentet data[1]. Hastighedsforbedringen kan dels komme direkte ved at mindre filer tager kortere tid at hente, dels indirekte ved at en generelt mindre trafikmængde på internettet alt andet lige burde give kortere responstider for alle. Sidstnævnte er dog i sagens natur svært at undersøge, og vi beskæftiger os i det følgende kun med den umiddelbare gevinst.

Hastighedsgevinsten afhænger naturligvis af komprimeringsalgoritmen – dels hvor effektivt den komprimerer, dels hvor hurtig den er. Gevinsten afhænger imidlertid også af HTML-dataenes andel af den samlede overførte mængde data – hvis f.eks. kun 1% af dataene er HTML, vil en kompressionsløsning, ligegyldigt hvor effektiv den er, næppe være interessant fordi den mulige gevinst er for lille.

Desuden skal en hastighedsgevinst i forbindelse med overførslerne sættes i forhold til hvor lang tid det tager at overføre dataene nu. Selvom den relative gevinst kan være stor, f.eks. 50%, ville det ikke betyde noget hvis en given bruger sammenlagt kun sparede en hundredel sekund.

[1] Dog med enkelte undtagelser, heriblandt StofaNet

I begge tilfælde ville der ikke være behov for en løsning. Vi er derfor nødt til at foretage en undersøgelse af dels HTML-filernes andel af den samlede datamængde, dels af de nuværende overførselstider.

3.1.1 Undersøgelse af datasammensætning

Til analysen betragter vi en logfil for proxyserveren `granit.but.auc.dk`, en maskine der sidder mellem de lokale brugere på basisuddannelsen på Aalborg Universitet, og resten af internettet og gemmer ofte benyttede filer midlertidigt.

Logfilen indeholder oplysninger om hvilke filer på internettet brugere på basisuddannelsen har besøgt, og en analyse af den kan derfor give et overblik over omkring 900 menneskers dataforbrug, se tabel 3.1 (selve analyseprogrammet og sorteringskriterierne er beskrevet i afsnit B.2). Der er dog ingen garanti for at dataforbruget fra maskiner på universitetet modsvarer forbrugsmønstret når folk sidder hjemme. Dette antager vi dog.

datatype	mængde (Mb)	andel (%)	i cache (%)	stør. (kb)
HTML	1879,37	11,69	15,16	7,79
billeder	1622,76	10,09	61,6	6,62
andet	12580,49	78,22	2,19	27,02
statisk	364,45	19,39	41,18	9,71
dynamisk	1514,92	80,61	8,9	7,44

Tabel 3.1: Tabel over en uges trafik fra basisuddannelsens proxyserver

Fordelingen af datamængden

Tabel 3.1 giver oplysninger om tre forskellige kategoriers andel af trafikken, HTML som er de websider vi kunne være interesseret i at komprimere, billeder der ofte indgår som en del af siderne, men ligger særskilt og så godt som altid er komprimerede, og andet der dækker over 65% programdata (næsten udelukkende fra `distributed.net`-klienter[2]), 4% lyd, 7% video og 24% andre ting som ISO-filer (til cd'ere), programmer og andre pakkede ting.

Umiddelbart giver fordelingen med kun omtrent 12% på HTML-sider ikke plads til megen forbedring. I midlertid dækker de 78% over data som ikke er en typisk del af navigationsmønstret på websider – hentning af lyd, video og programmer foregår ofte parallelt med selve surfningen hvorfor en bruger ikke sidder og venter på de data før vedkommende kan se siden, som det ellers er tilfældet med billederne der typisk vises indlejret.

Hvis vi ser bort fra dem, er fordelingen på omtrent halvdelen på websider og halvdelen på billeder – her er der altså mulighed for at effektivisere. I øvrigt kan

[2]Program der arbejder i baggrunden og løser svære problemer fra en central server

fordelingen være noget skæv i retning af de 78% andre data fordi universitetets internetforbindelse er meget hurtigere end de flestes brugers linjer, 100 Mbit/s, og dermed bedre er i stand til at klare de tunge datatyper.

En sidste pointe er at de datamængder der er angivet i tabellen, er for de faktiske overførsler mellem proxyserveren og internettet – kolonnen med „i cache“ viser at f.eks. billeder ofte kan hentes fra proxyserveren i stedet for direkte fra webserveren, og dette er altså indregnet i tallene for datamængderne. Hvis dette ikke var tilfældet, ville billeddatamængden f.eks. være en del højere. En almindelig bruger har selvfølgelig næppe adgang til en proxyserver lokalt, men på den anden side udfører browsere i forvejen samme funktion.

Men i hvert fald ses det på fordelingen af datamængden at en kompression af HTML ville kunne give brugeren en hastighedsgevinst.

Statisk og dynamisk indhold

Tabellen angiver også fordelingen mellem statiske og dynamiske websider. En statisk webside er en almindelig HTML-fil der ligger i sin færdige version direkte på serveren, mens en dynamisk side er en side som sættes sammen af serveren lige før den sendes, f.eks. ud fra oplysninger i en database.

Grunden til at det er interessant at undersøge, er at det kan vanskeliggøre komprimeringen af siderne. Statiske sider kan komprimeres på forhånd så hastigheden af komprimeringsalgoritmen er relativt ligegyldig – dynamiske sider skal derimod komprimeres løbende hvad der stiller større krav til hastigheden. Måske noget overraskende kan man i tabellen se at 80% af siderne er dynamiske, altså skal en effektiv løsning også kunne bruges på dynamiske sider. Vi kan derfor konstatere at hastighed spiller en rolle.

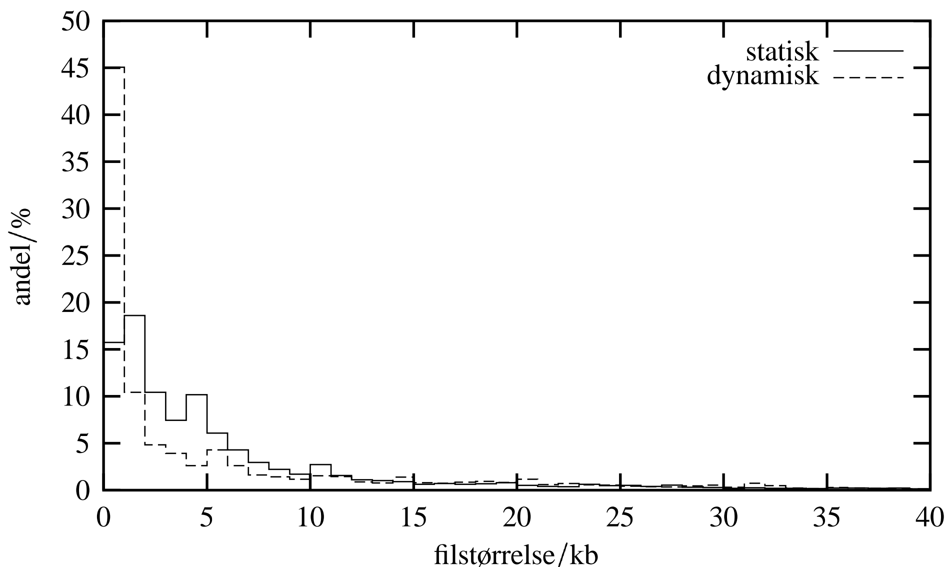
Filstørrelser

Den gennemsnitlige filstørrelse fremgår i den sidste kolonne i tabellen. Den gennemsnitlige størrelse siger imidlertid ikke så meget om fordelingen hvorfor et diagram over de forskellige filstørrelser for datatyperne fremgår i figur 3.1 på den følgende side. Filerne er inddelt i 1 kb’s intervaller hvis andele af det samlede antal filer derefter er plottet op til 40 kb (over denne størrelse var antallet af filer så lille at det ikke kan ses på diagrammet).

Diagrammet viser at størstedelen af filerne er mindre end 10 kb. Så komprimeringsmetoden skulle gerne være effektiv på små filer. Selvom det ikke er så stort et problem fordi de små filer jo i sagens natur ikke fylder så meget og derfor kan hentes hurtigere.

3.1.2 Undersøgelse af overførselshastighed

For at få et indtryk af hvor meget der er at spare for brugerne har vi gennemført en test af overførselstiderne for en række websider (de er beskrevet i afsnit B.1) for



Figur 3.1: Fordelingen af filstørrelser for granit.but.auc.dk.

henholdsvis et almindeligt analogt modem med en 56k-forbindelse, en enkeltkanals ISDN-linje på 64 kbit/s og en ADSL-forbindelse på 256 kbit/s. Resultatet kan ses i tabel 3.2, selve testen er dokumenteret i bilag B.3 på side 72.

HTML-side	modem (s)	ISDN (s)	ADSL (s)	filstør. (kb)
Carlsberg	2,0	**0,6**	**0,6**	7,0
The Voice	5,6	**2,2**	**2,2**	28,0
IT-Avisen	6,9	**2,3**	2,4	30,8
ITL	4,7	**1,5**	**1,5**	18,4
Jyllandsposten	15,3	5,7	**5,1**	66,2
Jubii	10,7	3,6	**3,4**	43,5

Tabel 3.2: Overførselstiderne for forskellige forbindelser

Det fremgår af tabel 3.2 at der faktisk er en vis ventetid selv for de hurtigere forbindelser; allerede omkring 2 sekunders ventetid kan mærkes. I hvert fald er 5-10 sekunders ventetid for de middelstore sider med det analoge modem nok til at kunne forstyrre, og ifølge [19] er der 2,8 mill. danskere med abonnement til analogt modem og kun ca. 400.000 med ISDN og 70.000 med ADSL (svarende til andelene 86%, 12% og 2%). Altså er der grundlag for en forbedring.

Det kan virke mærkeligt at ISDN-linjen er nogenlunde lige så hurtig som ADSL-forbindelsen, men det kan skyldes at der over ISDN-forbindelsen allerede bruges komprimering, se afsnit 3.1.3.

3.1.3 Kompression af forbindelser

Noget, der kan undergrave ideen i at komprimere HTML-filerne, er, hvis forbindelserne allerede bliver komprimeret.

På større forbindelser – f.eks. over Atlanterhavet og på ruter internetudbydere imellem – benyttes der ingen kompression. Årsagen til dette er formodentlig at hardwarekravene til at komprimere så store mængder data tilstrækkeligt hurtigt gør det for dyrt.

I teknologien for analoge modemer er der imidlertid allerede indbygget kompression [3] i form af V.42bis som benytter en variant af LZW (se evt. afsnit 5.2.2). Kompressionen foregår løbende og slås fra hvis den ikke er effektiv nok [16]. Kompressionen er dog ikke specielt kraftig, og det er ikke klart defineret, hvad „effektivt nok“ betyder.

Vores undersøgelser har ikke vist, at der bruges nogen form for hardwarekompression på ISDN-forbindelser, men der kan dog benyttes softwarekompression [8]. Der er foreslået nogle standarder, bl.a. LZ-Stac[3], omkring det [14], og disse har ved test vist sig at give en temmelig effektiv kompression [9] som det også fremgår af tabel 3.2 på forrige side.

Vores undersøgelser har heller ikke vist, at ADSL-opkoblinger benytter hardwarekompression. Der er dog foreslået en standard [15], som medfører at ADSL-forbindelser kan benytte LZ-Stac. En del routere og lignende udstyr kan benytte denne standard og at stadigt mere hardware understøtter det.

Som nævnt er LZ-Stac kun en *foreslået* standard. Det betyder også, at der ved oprettelse af opkaldsforbindelser forhandles, om der skal benyttes kompression. Det vil sige, at det kræver, at internetudbyderne understøtter og benytter LZ-Stac, for at man kan bruge det.

3.2 Udbyderne

Mens besparelser på trafikmængden i sig selv ikke er så interessant for brugerne, er det anderledes for udbyderne, da en mindre trafikmængde direkte kan oversættes til en billigere forbindelse med mindre båndbredde (eller evt. en udskydelse af en planlagt udvidelse). Hastigheden af hele processen i forbindelse med overførslen betyder naturligvis stadig også meget fordi den er afgørende for hvor mange brugere en server kan betjene. Både båndbredden og behandlingstiden for en betjening kan være flaskehalse.

Som ved brugersiden er det nødvendigt at få fastslået om andelen af HTML er tilstrækkelig stor i forhold til andre data. Desuden er forholdet mellem statiske og dynamiske sider yderligere interessant fordi det kan have stor betydning for hvor meget regnekraft udbyderen skal have til rådighed hvis komprimeringsalgoritmen viser sig at være relativ langsom.

[3] Den kaldes også MS-Stac, da den angiveligt er udviklet af Microsoft

3.2.1 Undersøgelse af datasammensætning

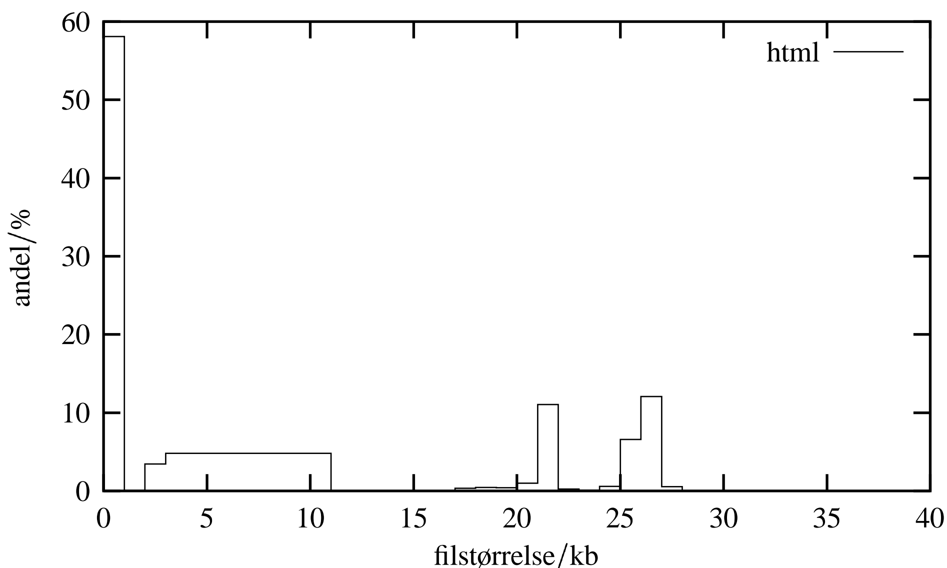
Analysen for udbydersiden bygger på en logfil fra serverprogrammet på `www.it-avisen.dk`, se tabel 3.3, der repræsenterer et typisk nyhedssted.

datatype	mængde (Mb)	andel (%)
HTML	2424.77	63,4
billeder	1399.53	36,6

Tabel 3.3: 11 dages logfil fra www.it-avisen.dk

Det ses at andelen af websider i forhold til billeder er 63%, dvs. at der kunne være en del at hente ved at komprimere. Det viste sig at IT-Avisen udelukkende har dynamiske sider. Til gengæld er alt indholdet dynamiske sider så man kan ikke komme uden om at komprimere løbende.

Fordelingen af filstørrelser for IT-Avisen fremgår på figur 3.2. Det ses at en stor del af websiderne som ved brugersiden er mindre end 10 kb, men til gengæld er der to toppe mellem 20 og 30 kb der samlet kan ses at stå for omtrent 25% af filerne.



Figur 3.2: Fordelingen af filstørrelser for IT-Avisen.

Spørgsmålet er om tallene fra IT-Avisen er typiske for alle udbydere. Det kan kun afklares ved yderligere undersøgelser af andre udbydere, men desværre er det ikke helt let at få udleveret logfilerne, og vi har derfor været afskåret fra det. Umiddelbart *ser* siderne hos IT-Avisen typiske ud; Men er altså ikke nødvendigvis repræsentative.

3.3 Sammenfatning

Fra brugernes synspunkt skal en gevinst komme fra en hastighedsforbedring – brugeren skal kunne skære ned på den tid vedkommende skal vente før en side dukker op i browseren. At der faktisk er mulighed for en gevinst forudsat en passende kompressionsmetode, dokumenteres af undersøgelsen af datasammensætningen der viser at en betragtelig del af de overførte data er websider. Samtidig viser hastighedstesten at der faktisk er noget tid at afkorte idet analoge modemer, som flertallet i danmark har, kan give ventetider på 5-10 sekunder for middelstore sider.

Til gengæld udhuler testen af eksisterende kompression i modemer og i forbindelse med ISDN-linjer dog dette noget, også selvom den ikke er nær så effektiv som det et almindeligt program kan klare.

For udbydersiden er der penge at spare i at begrænse mængden af overført data; tabel 3.4 viser et eksempel på priserne for diverse hastigheder for en fast internetforbindelse hos Cybercity [5]. Hvis man antager at kompression kan reducere datamængden for HTML til $\frac{1}{5}$, kan halvdelen af webtrafikken spares, hvad der, som det ses i tabellen, næsten kan halvere udgifterne til internetforbindelsen.

hastighed	årspris (kr.)
256 kbit	30.000
512 kbit	35.000
1 Mbit	40.000
2 Mbit	70.000
4 Mbit	110.000
8 Mbit	180.000

Tabel 3.4: Priseksempel fra Cybercity Erhverv

Med hensyn til krav til kompressionsmetoden har undersøgelserne af datasammensætningen vist at andelen af dynamiske sider i forhold til statiske er så stor at der er behov for en metode der også kan bruges på dynamiske sider – for udbyderne kan dette betyde større krav til servernes beregningskraft fordi denne må foregå løbende, statiske sider ville bare kunne komprimeres en gang for alle, men når de i forvejen har investeret en del i den for at kunne omsætte de dynamiske sider til HTML, er det muligvis relativt billigt at foretage den fornødne udvidelse.

Desuden er en stor del af websiderne ret små, dvs. under 10 kb, så medmindre metoden også er effektiv på små filer, kan en del af gevinsten forsvinde (mindre filer fylder dog ikke så meget så de udgør forholdsmæssigt mindre af den samlede datamængde).

Kravene til en kompressionsalgoritme er altså at den

- er effektiv på HTML-sider
- kan bruges i forbindelse med dynamiske sider

- ikke er for beregningskrævende, dvs. er hurtig til at komprimere og dekomprimere
- er god til små såvel som store filer

Nogle af disse krav kan være i konflikt med hinanden så en given algoritmes egenskaber må afhænge af en afvejning.

Når potentialet for en kompressionløsning og nogle krav til den således er blevet fastslået, kan vi gå over til at definere selve kompressionsbegrebet, først fra en teoretisk synspunkt i form af begrebet *informationsteori*.

Kapitel 4

Informationsteori

Informationsindholdet er som nævnt i indledningen ikke nødvendigvis lig med størrelsen af data. Hvordan finder man da ud af hvor meget information der er i en given mængde data?

En måde at gribe problemet an på er at stille en række ja/nej-spørgsmål om dataene – er det næste tegn et bogstav? Er det et stort bogstav? Er det et 'a'? Er det et 'b'?

Et ja eller et nej kan opfattes som enten et 0 eller et 1 så rækken af svar bliver til en binær streng. Hvis vi opfatter dataene som adskilte tegn, kan målet for information være det gennemsnitlige antal af spørgsmål, svarende til den gennemsnitlige længde af de binære strenge, der er nødvendige for at kunne identificere hvert tegn.

Antallet af nødvendige spørgsmål afhænger imidlertid af sandsynlighederne for de enkelte udfald; hvis et givent udfald var meget sandsynligt kunne det jo f.eks. godt betale sig at starte med at spørge direkte efter netop det udfald, mens hvis modsat alle udfald var omtrent lige sandsynlige, kunne det bedst betale sig at spørge om det konkrete udfald var et fra den ene halvdel af udfaldene – så fik man hurtigst muligt indkredset det.

4.1 Entropi

Antallet af nødvendige spørgsmål til et meget sandsynligt udfald er altså mindre end for et med en lille sandsynlighed, hvilket kan opfattes som at informationsindholdet for det første udfald er mindre end for det andet. Dette forhold kan opnås matematisk ved at definere informationsindholdet for et givent udfald som [17, s. 14]

$$i(x) = -\log_2 P(X = x)$$

hvor x er et udfald for en diskret stokastisk variabel X og $P(X = x)$ sandsynligheden for at udfaldet optræder (vi benytter herefter en kortere notation hvor $p(x) = P(X = x)$). Her er benyttet totalslogaritmen hvad der giver resultatet i bit,

andre logaritmer giver en anden enhed der dog kun afviger fra bit-enheden med en konstant faktor [4, s. 14].

Før vi kan fortsætte, skal vi have definitionen af et *alfabet* på plads. Et alfabet $\mathbb{A}$ er mængden af forskellige symboler som en algoritme kan finde i en given mængde data; f.eks. kan „abccba“ repræsenteres vha. alfabetet $\mathbb{A} = \{a, b, c\}$, men det kan også repræsenteres vha. $\mathbb{B} = \{abc, cba\}$.

For en given diskret stokastisk variabel X hvis udfaldsrum er alfabetet $\mathbb{A}$, og hvis udfald er bestemt af sandsynlighedsfordelingsfunktionen $p(x)$, definerer man så *entropien* $H(X)$ som den forventede middelværdi af informationsindholdene for alle udfaldene [4, s. 13]:

$$H(X) = \sum_{x \in \mathbb{A}} p(x) i(x) = -\sum_{x \in \mathbb{A}} p(x) \log_2 p(x)$$

Tag som et eksempel den stokastiske variabel i tabel 4.1 som bygger på tegnfølgen „aaabbcbcbcabababaa“ med alfabetet $\{a, b, c\}$. Ud fra sandsynlighedsfordelingen kan entropien beregnes til

$$H = -\tfrac{7}{16} \cdot \log_2 \tfrac{7}{16} - \tfrac{6}{16} \cdot \log_2 \tfrac{6}{16} - \tfrac{3}{16} \cdot \log_2 \tfrac{3}{16} = 1{,}5052$$

X	a	b	c
$p(x)$	$\frac{7}{16}$	$\frac{6}{16}$	$\frac{3}{16}$

Tabel 4.1: Frekvensfordeling med alfabetet $\{a, b, c\}$

Hvis man betragter tegnfølgen, ses det at vi også kan bruge alfabetet $\{aa, ab, bc\}$ da a, b og c kun optræder i denne rækkefølge. Hvis man gør det, bliver entropien i stedet

$$H = -\tfrac{2}{8} \cdot \log_2 \tfrac{2}{8} - \tfrac{3}{8} \cdot \log_2 \tfrac{3}{8} - \tfrac{3}{8} \cdot \log_2 \tfrac{3}{8} = 1{,}5613$$

Entropien afhænger altså hvordan man vælger sit alfabet. Ved at vælge et med blokke af to tegn, har vi næsten halveret entropien – 1,505 bit pr. symbol mod 1,561 bit pr. to datategn svarende til 0,7807 bit pr. datategn.

4.2 Egenskaber ved entropi

Man kan bevise nogle interessante træk ved $H(X)$ – det er faktisk disse egenskaber der gør at entropi er et nyttigt begreb at beskæftige sig med. Sætningerne vil ikke blive bevist her, da en udtømmende bevisførelse ikke er relevant for projektet som helhed.

4.2.1 Optimal kodning

Når man skal vælge et alfabet og dets repræsentation i form af kodeord, gælder det naturligvis om at kodeordene resulterer i den kortest mulige tekst når de bliver sat sammen til den komprimerede tekst. Der er eksisterer altså *optimale* koder der giver et bedre kompressionsforhold end alle andre koder på en given mængde data.

Hvis $l_1, l_2 \ldots l_n$ er længderne af de optimale kodeord for en stokastisk variabel X med sandsynlighedsfordelingen $p(x)$, og L er den forventede længde af den optimale kode ($L = \sum p(x_i) l_i$), gælder så [4, s. 86–88]

$$H(X) \leq L \leq H(X) + 1$$

Det vil sige at den mindste mængde data man kan bruge til at repræsentere et tegn, ligger inden for én bit af entropien af dataene. Vi kan presse dataene ned til $H(X)$, men så heller ikke længere.

At det optimale ligger inden for én bit og ikke er mindre, hænger sammen med at der er et vist spild hvis entropien ikke giver et helt tal – vi kan ikke sende halve bit. Dette problem kan formindskes ved at kode større blokke af gangen så spildet bliver spredt ud på flere tegn og dermed samlet bliver mindre.

4.2.2 Maksimal entropi ved ligelig fordeling

Hvis n er antallet af udfald i fordelingen for X, opnås den maksimale entropi hvis sandsynlighederne for udfaldene er ens ($p(x) = \frac{1}{n}$), og har værdien $\log_2 n$ [4, s. 27].

Hvis f.eks. X kan antage 256 værdier med lige stor sandsynlighed, er entropien $H(X) = \log_2(256) = 8$ bit. Hvis der er forskel på sandsynlighederne, bliver entropien mindre end 8 bit.

Denne egenskab fortæller noget om hvorfor almindelige data normalt kan komprimeres. Hvis en tegnfølge er genereret helt tilfældigt, vil de forskellige værdier naturligvis optræde med nogenlunde samme sandsynlighed hvis man betragter en tilstrækkeligt stor datamængde – det vil sige at entropien for sådan en tegnfølge vil ligge lige omkring en almindelig repræsentation af tegnene hvor man bruger et fast antal bit, altså kan disse ikke komprimeres.

For data der ikke er genereret helt tilfældigt, vil der derimod som regel være en struktur der gør at nogle ting går igen mange gange. F.eks. optræder 'e' meget hyppigere end 'z' i en almindelig dansk tekst, og dermed bliver entropien mindre.

Det kan virke mærkeligt at helt tilfældige data indeholder mere information end omhyggeligt udformede, menneskeskabte data, men den menneskelige bearbejdningsproces er jo netop gået ud på at sortere og ordne dataene. En større orden giver færre muligheder, og gør det muligt at forudsige data.

4.2.3 Afhængigheder reducerer entropi

I almindelig tekst optræder tegnene med en vis indbyrdes afhængighed. Hvis f.eks. bogstaverne „bæld“ står i en dansk tekst, er der en meget stor sandsynlighed for

at x enten er et ‘o’ eller et ‘y’. Det viser sig at indbyrdes afhængigheder mellem tegnene forårsager at entropien reduceres. Først er der dog brug for at få defineret fælles entropi og betinget entropi.

Den *fælles entropi* $H(X,Y)$ for to diskrete stokastiske variable med den fælles sandsynlighedsfordeling $p(X,Y)$ er defineret som [4, s. 15]

$$H(X,Y) = -\sum_i \sum_j p(x_i, y_j) \log p(x_i, y_j)$$

mens den *betingede entropi* $H(X|Y)$ er defineret som

$$\begin{aligned} H(X|Y) &= \sum_j p(y_j) H(X|Y = y_j) \\ &= -\sum_j p(y_j) \sum_i p(x_i|y_j) \log p(x_i|y_j) \\ &= -\sum_i \sum_j p(x_i, y_j) \log p(x_i|y_j) \end{aligned}$$

hvor vi har benyttet definitionen for entropi og skrevet lidt om. Den fælles entropi kan vha. en kæderegel udvides til at opfatte flere variable med fælles fordeling.

Man kan da bevise [4, s. 27–28] at for to afhængige stokastiske variable X og Y med en fælles sandsynlighedsfordeling $p(X,Y)$, vil entropien for X givet at udfaldet af Y er kendt, være mindre end entropien for blot X:

$$H(X|Y) < H(X)$$

Det kan bl.a. bevises at have den følge at entropien for en blok af tegn altid vil være mindre end (eller evt. lig med) de sammenlagte entropier for de enkelte tegn

$$H(X_1, X_2, \dots, X_n) \leq H(X_1) + H(X_2) + \dots + H(X_n)$$

Eksemplet fra afsnit 4.1 hvor entropien blev halveret ved at vælge et alfabet med blokke af to tegn, illustrerer dette.

Ud fra denne egenskab fås yderligere en begrundelse for at f.eks. almindelig tekst kan komprimeres. Den struktur, der er i menneskeskabte data, gør at sandsynligheden for at et tegn optræder, afhænger meget af konteksten. Denne afhængighed formindsker entropien hvis ens model af dataene er i stand til at tage hensyn til den, f.eks. ved at arbejde på blokke af tegn i stedet for kun enkelte.

4.3 Komprimering og entropi

Man kan faktisk vise at en optimal kode der ligger inden for grænserne $H(X)$ og $H(X) + 1$ er mulig at konstruere (endda ret simpelt, en algoritme til at gøre det præsenteres i afnit 5.1.2), men som det også ses er der forskellige måder man kan

formindske datamængden yderligere på. Alt afhængig af hvordan man vælger sit alfabet, repræsenteret ved den stokastiske variabel X, kan $H(X)$ nemlig variere.

For at optimere kompressionsforholdet gælder det altså om at vælge en model der passer til dataene. Hvis modellen har som en grundlæggende antagelse at hvert tegn er helt uafhængigt af det foregående, mister man en mulig gevinst hvis hvert tegn faktisk afhænger til en vis udstrækning af sin forgænger i de data man benytter modellen på.

Med de ovennævnte egenskaber kan vi allerede se at det kan være en fordel at slå flere tegn sammen fordi man så dels kan udnytte afhængigheder mellem tegnene, afhængigheder der kommer fra den underliggende datastruktur; dels kan reducere det spild der opstår i forbindelse med generering af kodeordene.

For at få en fornemmelse af hvordan informationsindholdet i HTML-filer udmønter sig, beskæftiger vi os i det følgende lidt med deres struktur.

4.3.1 HTML, informationsteoretisk set

HTML består af bidder af almindelig tekst omgivet af opmærkninger, *tags*, af hvilke kun et vist antal er gyldige, f.eks. <P> eller <TABLE>. Simple tags som de nævnte er hyppige og indeholder kun lidt information. Imidlertid kan de fleste tags også tage parametre, f.eks. <a href="fil.html">, og da indholdet af parametrene ikke er begrænset på nogen måde, kan den form for opmærkninger besidde et større informationsindhold.

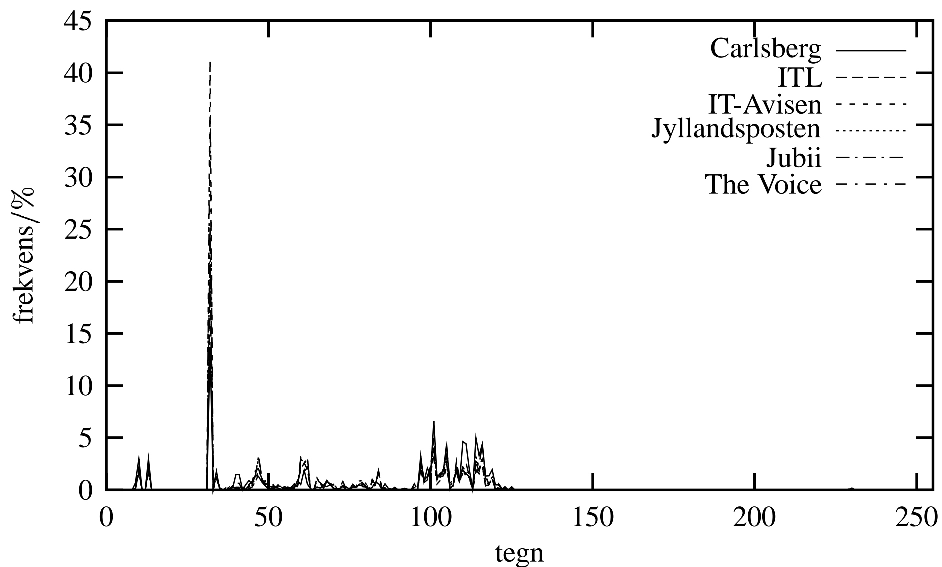
Tags uden parametre kan altså teoretisk komprimeres rigtigt meget, mens vi ikke kan forvente at kunne komprimere de andre tags så meget medmindre de optræder flere gange i HTML-filen.

Ud over opmærkningen er der også den almindelige tekst. Kompressionen af denne kan være meget effektiv fordi bogstaverne i almindelig tekst som før nævnt afhænger meget af hinanden. Desuden kan vi spare på alfabetet fordi f.eks. almindelig dansk tekst kun består af typisk omkring 26 bogstaver samt nogle få symboler som bindestreg, komma og punktum så alfabetet med stor sandsynlighed vil holde sig på under 64 forskellige tegn. Dvs. at vi umiddelbart, uden at være overdrevent sofistikerede, kan spare 2 bit for hvert tegn.

Et af de teoretiske udledte resultater er at en „skæv“ frekvensfordeling giver en mindre entropi end en ligelig, hvorfor vi har undersøgt frekvensfordelingen for tegnene på vores udvalgte sider (beskrevet i afsnit B.1), se figur 4.1 på næste side.

Figuren viser at fordelingen faktisk ikke er ligelig. Men derimod er den overraskende ens i de forskellige filer. Den eneste tydelige forskel er højden på toppen omkring 32 som svarer til ASCII-koden for mellemrum, hvilket kan skyldes forskellige måder at indrykke HTML-filerne på.

Med et almindeligt kompressionsprogram i hånden kan man hurtigt overbevise sig om at HTML kan komprimeres meget, og dette støttes af de ovenstående observationer. Frekvensfordelingernes ensartethed antyder at HTML-filerne vil ligge meget lige mht. kompressionsforholdet og tyder desuden på at der kan spares data. Hvor meget der kan komprimeres varierer naturligvis lidt med skrivestil, opsæt-



Figur 4.1: Frekvensfordeling for tegnene (repræsenteret ved deres kode ifølge kodningen ISO-8859-1) på vores udvalgte sider

ning, mængden af opmærkninger, den valgte algoritmes styrker osv., men alt i alt viser vores overvejelser et stort potentiale for kompression.

På baggrund af dette undersøger vi herefter nogle algoritmer der kan bruges til at komprimere HTML-filerne.

Kapitel 5

Analyse af kompressionsmetoder

Der findes to grundlæggende forskellige metoder at komprimere på. Den statistiske, der baserer sin kompression på statistisk analyse af data, og den ordbogsbaserede, der opretholder en ordbog over gentagne mønstre i data.

Under de statistiske metoder undersøger vi Huffman-koder og aritmetiske koder som er klassiske metoder indenfor kompression. Grundideen i statistiske kompressionsmetoder er at symboler der bruges tit, ikke skal beskrives med lige så meget data som symboler der bruges sjældent. Altså at de skal beskrives med værdien af deres entropi.

Første eksempel på denne form for kompression finder vi i midten af 1800-tallet takket være Samuel Morses alfabet [17]. Han opdagede nemlig at man kunne spare tid i transmission af beskeder, og dermed også kunne opnå en form for kompression, ved at tildele hyppigt anvendte bogstaver som 'e' korte kodeord.

For at kunne udtale sig om hyppighed bliver man nødt til at finde eller estimere frekvenser for disse symboler. Jo tættere disse symbolsandsynligheder ligger på de faktiske forhold i dataene, jo bedre kan der komprimeres – man kan vha. informationsteori bevise at man så kommer tættere på entropien [4].

Under de ordbogsbaserede metoder undersøger vi de klassiske Lempel-Ziv 77 og Lempel-Ziv 78, samt populære varianter her af. Disse kompressionsmetoder blev udviklet i 1970erne af de to forskere Jacob Ziv og Abraham Lempel [16] (navnene fik de efter de respektive årstal de blev udviklet i).

De to metoder varierer på den måde de laver deres henvisninger til ordbogen, og på deres måde at opretholde denne. Ordbogen i de ordbogsbaserede komprimeringsmetoder er en liste over ofte forekomne tegnfølger i dataene; metoderne arbejder nemlig ikke kun med enkelte symboler, men med *mønstre* som dækker flere tegn. Ideen er så at erstatte mønstre af variable længder med kodeord af faste længder, omvendt statistiske metoder hvor symbolerne har faste størrelser og koderne variable længder.

Ud over disse metoder undersøger vi også en transformation der kan gøre metoderne mere effektive. Den transformation vi ser på, går under navnet Burrows-Wheeler-transformationen og bruges ofte sammen med kodningsprincippet move-

to-front. Ideen er at transformationen ændrer dataene på en sådan måde at de slutteligt kan komprimeres bedre – også selvom der kræves lidt mere data undervejs til at holde styr på transformationen for at gøre den reversibel.

Burrows-Wheeler-transformationen kan anvendes på alle former for data, men den stiller dog et krav om at de skal kunne deles op i blokke. Move-to-front-kodning er kun effektiv at bruge på specielle data hvor ens symboler ligger i nærheden af hinanden. Sådanne data er netop det Burrows-Wheeler-transformationen producerer.

Alle disse komprimeringsmetoder kan bruges på HTML-filer. I vores gennemgang af metoderne tager vi udgangspunkt i det samme eksempel:

```
<HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML>
```

Dette er ikke for at kunne sammenligne metodernes effektivitet, men for at gøre det lettere at se forskellen i deres virkemåde.

Eksemplet er på 53 tegn, inklusive mellemrum, og består af symbolerne {<, >, -, /, ␣, a, b, d, e, f, H, i, L, M, n, o, p, s, T, t, y} som altså udgør vores alfabet $\mathbb{A}$. Ukomprimeret vil dette eksempel sædvanligvis optage 8 bit pr. symbol, eller $8 \cdot 53 = 424$ bit i alt. Med den antagelse at hvert tegn er uafhængigt af de andre, er entropien på 4,18 bit pr. symbol – så der er plads til forbedringer. Vi vil nu undersøge hvor godt Huffman klarer det.

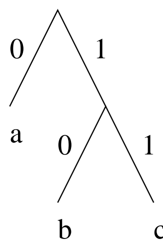
5.1 Statistiske metoder

5.1.1 Præfikskoder

Før beskrivelsen af Huffman-kodning introducerer vi præfikskoder da Huffman-kodning netop danner præfikskoder.

En præfikskode er en kode som har den egenskab at inget kodeord er et præfiks i noget andet kodeord. Det vil sige at ingen lange kodeord begynder med samme talsekvens som nogle af de andre kortere kodeord.

Antag at ‘a’ har koden 0, ‘b’ koden 10 og ‘c’ koden 11. Så ses det at man uden at kende længden af koden umiddelbart kan se hvilken kode man skal vælge. Hvis man f.eks. skriver 00, ved vi at dette betyder „aa“.



Figur 5.1: Kodeordene illustreret ved et binært træ

Koderne kan vises som et binært træ som i figur 5.1 på forrige side hvor bladene svarer til symbolerne – det kan man gøre med alle præfikskoder. Hvis vi så har fået sendt bitrækken „00100111001111“, kan vi ved hjælp af træet entydigt afkode den oprindelige tekst:

0	0	10	0	11	10	0	11	11
a	a	b	a	c	b	a	c	c

Den afkodede tekst fylder $9 \cdot 8 = 72$ bit, mens bitrækken kun optager 14 bit. Det hænger sammen med at vi kun har 3 symboler i vores alfabet, og vi kan dermed nøjes med 2 bit til hver for at beskrive dem entydigt. Præfikskoderne gør det muligt at bruge forskellige længder kode til at beskrive vores symboler med, og i vores eksempel beskrives hvert symbol med $\frac{5}{3} \approx 1{,}66$ bit pr. symbol i gennemsnit. Men hvordan laver vi vores træ optimalt? Dette svar ligger i Huffman-kodning.

5.1.2 Huffman

Huffman-kodning er udviklet af David Huffman i 1950'erne [16], deraf navnet, og efterfulgte Shannon-Fano-kodning som var udviklet af Shannon, Weaver og Fano. Shannon-Fano-algoritmen komprimerer ikke så godt som Huffman, selvom dens operationer ligner. Huffmans metode går ud på at finde frekvenser for symbolerne i dataene og derefter oprette en træstruktur hvori man indsætter disse symboler ud fra deres frekvens. Herefter kan man ved hjælp af træet let udlede et kodeord med forskellige antal bit til hvert symbol.

For at lave dette træ starter vi med at tage de to symboler med lavest frekvens og erstatte dem med et fælles, kombineret symbol C_1. Dette symbol C_1 tildeles frekvensen svarende til summen af frekvenserne for de to symboler. Herefter indgår C_1 igen i mængden af symboler til næste iteration af algoritmen hvor C_2 bliver fundet. Dette fortsætter til der kun er ét symbol tilbage. Vha. C-metasymbolerne kan vi så generere træet.

Eksempel

Vi illustrerer Huffman-metoden ved hjælp af vores eksempel.

```
<HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML>
```

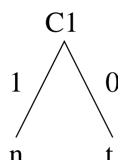
Først skal vi finde $p(x)$, hvor x er et symbol fra $\mathbb{A}$, altså hvert symbols frekvens. Den findes ved at tælle forekomster af de enkelte symboler og dividere dem med længden af teksten. Resultatet ses i tabel 5.1 på næste side.
Her har vi samtidig sorteret tabellen efter frekvens.

Huffmantræet

Vi kan nu gå i gang med at danne Huffman-træet ved at finde de to symboler med mindste frekvens, altså 'n' og 't'. Disse to symboler beskrives samlet som C_1, og dette danner så den første del af træet, se figur 5.2 på den følgende side.

Symbol	<	>	T	/	␣	d	e
Forekomster	6	6	4	3	3	3	3
$p(x)$	0,1132	0,1132	0,0754	0,0566	0,056	0,0566	0,0566
Symbol	H	L	M	b	o	p	s
Forekomster	3	3	3	2	2	2	2
$p(x)$	0,0566	0,0566	0,0566	0,0377	0,0377	0,0377	0,0377
Symbol	y	-	a	f	i	n	t
Forekomster	2	1	1	1	1	1	1
$p(x)$	0,0377	0,0188	0,0188	0,0188	0,0188	0,0188	0,0188

Tabel 5.1: Tabel over frekvenser



Figur 5.2: C_1-delen af Huffman-træet

C_1 optræder nu med den sammenlagte frekvens for ‘n’ og ‘t’ og anbringes i skemaet i stedet for de to. Bemærk at C_1 placeres så højt i skemaet som dens frekvens tillader, det vil i dette tilfælde sige lige efter ‘M’. Se tabel 5.2

Symbol	<	>	T	/	␣	d	e
Forekomster	6	6	4	3	3	3	3
$p(x)$	0,1132	0,1132	0,0754	0,0566	0,0566	0,0566	0,0566
Symbol	H	L	M	$\mathbf{C_1}$	b	o	p
Forekomster	3	3	3	2	2	2	2
$p(x)$	0,0566	0,0566	0,0566	0,0377	0,0377	0,0377	0,0377
Symbol	s	y	-	a	f	i	
Forekomster	2	2	1	1	1	1	
$p(x)$	0,0377	0,0377	0,0188	0,0188	0,0188	0,0188	

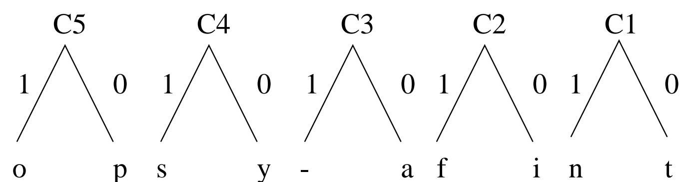
Tabel 5.2: Første skridt i Huffman-algoritmen

Herefter finder vi igen de to mindst sandsynlige tegn i skemaet. På denne måde danner vi C_2 op til C_5, se figur 5.3 på den følgende side. Det nye skema ses i tabel 5.3 på næste side.

I næste skridt skal vi kombinere C_1 og ‘b’. Dette foregår på samme måde som hvis C_1 havde været et enkelt symbol. Frekvenserne lægges sammen og det nye symbol C_6 skal anbringes i skemaet.

C_6 består af ‘b’ og C_1 og dette gør træet dybere, som det ses på figur 5.4.

Vi fortsætter nu på samme måde indtil der kun er é kombineret symbol til-



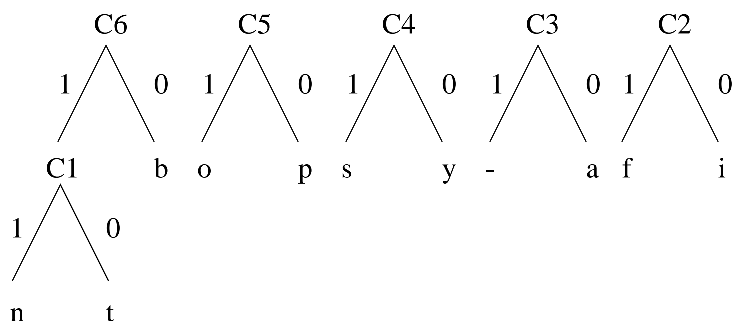
Figur 5.3: Huffman-træet med C_1, C_2, C_3, C_4 og C_5

Symbol	<	>	**C_5**	**C_4**	`T`	`/`	␣
Forekomster	6	6	4	4	4	3	3
$p(x)$	0,1132	0,1132	0,0754	0,0754	0,0754	0,0566	0,0566
Symbol	`d`	`e`	`H`	`L`	`M`	**C_3**	**C_2**
Forekomster	3	3	3	3	3	2	2
$p(x)$	0,0566	0,0566	0,0566	0,0566	0,0566	0,0377	0,0377
Symbol	**C_1**	`b`					
Forekomster	2	2					
$p(x)$	0,0377	0,0377					

Tabel 5.3: Andet skridt i Huffman-algoritmen

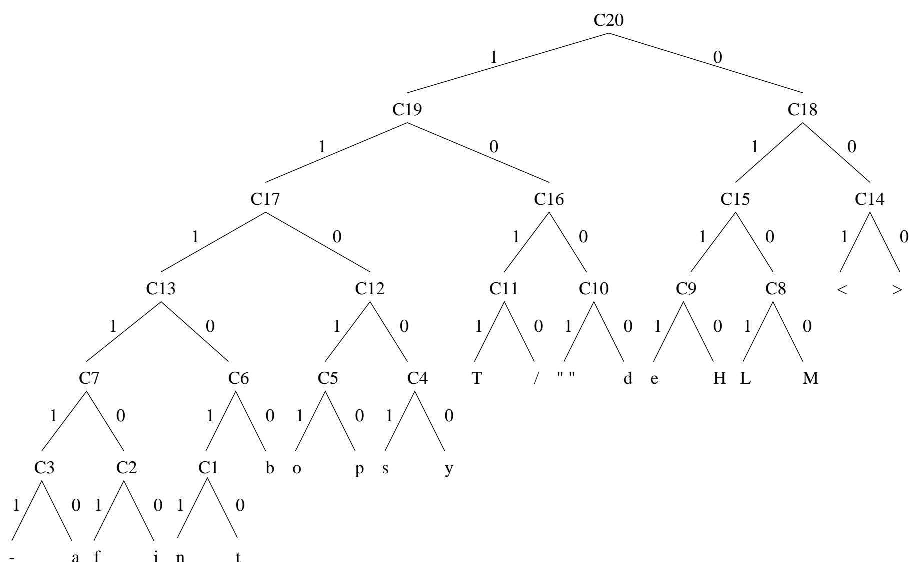
Symbol	<	>	**C_6**	**C_5**	**C_4**	`T`	`/`
Forekomster	6	6	4	4	4	4	3
$p(x)$	0,1132	0,1132	0,0754	0,0754	0,0754	0,0754	0,0566
Symbol	␣	`d`	`e`	`H`	`L`	`M`	**C_3**
Forekomster	3	3	3	3	3	3	2
$p(x)$	0,0566	0,0566	0,0566	0,0566	0,0566	0,0566	0,0377
Symbol	**C_2**						
Forekomster	2						
$p(x)$	0,0377						

Tabel 5.4: Tredje skridt i Huffman-algoritmen



Figur 5.4: Træet bliver dybere med C_6

bage[1]. Dette symbol vil have frekvensen 1 da det består af alle symbolerne i dataene. Det færdige træ kommer således til syne i figur 5.5 (hvor et mellemrum skrives ␣).



Figur 5.5: Det færdige Huffman-træ

Nu kan vi danne koderne for hvert symbol ud fra træet. For at finde koden skal man blot følge grenene ud til symbolet og nedskrive de enkelte nuller eller et-taller på vejen. Eksempelvis bliver koden for > til 000 og koden for `t` bliver 111010. Bemærk at vores kombinerede symboler C_1-C_{20} ikke længere har nogen betydning.

Symbol	Kode	Symbol	Kode
`>`	`000`	`<`	`001`
`M`	`0100`	`L`	`0101`
`H`	`0110`	`e`	`0111`
`d`	`1000`	␣	`1001`
`/`	`1010`	`T`	`1011`
`y`	`11000`	`s`	`11001`
`p`	`11010`	`o`	`11011`
`b`	`11100`	`t`	`111010`
`n`	`111011`	`i`	`111100`
`f`	`111101`	`a`	`111110`
`-`	`111111`		

Nu kan vi skrive vores oprindelige tekst som følgende bitstreng:

[1]Se eventuelt disse skridt i bilag A.1.1 på side 65

```
001 0110 1011 0100 0101 000 001 11100 11011 1000 11000 000
001 11010 000 1011 0111 11001 111010 1001 111110 111101 1001
0111 111011 1001 0110 1011 0100 0101 111111 11001 111100 1000
0111 001 1010 11010 000 001 1010 11100 11011 1000 11000 000
001 1010 0110 1011 0100 0101 000
```

Vores data bliver således komprimeret til en bitstreng på 222 bit i forhold til 424 bit før kompression, eller $\frac{222}{53} \approx 4{,}19$ bit pr. symbol i gennemsnit, meget tæt på entropien.

Men man skal have information med således at træet kan regenereres. Dennes størrelse kan estimeres ud fra alfabetet. Vi skal bruge et tal der kan beskrive frekvensen, samt 8 bit til at beskrive symbolet. Tallet der beskriver frekvensen, kan f.eks. være af en kommatalstype som optager 4 byte. Det vil sige at vi bruger 5 byte pr. symbol i alfabetet, og da alfabetet består af 21 symboler, bliver den totale størrelse $5 \cdot 21 = 105$ byte eller $8 \cdot 105 = 840$ bit. Så vi kan faktisk ikke opnå kompression i vores lille eksempel.

Mellemrummene i bitstrengen er kun med af hensyn til læsbarhed for læseren da man vha. træet kan dekode sekvensen uden, eftersom Huffman-koderne er præfikskoder. Hvorfor man kan se bort fra mellemrummere, blev forklaret i afsnit 5.1.1 på side 24.

Reversibilitet

Når man skal gendanne data der er blevet kodet med Huffman, skal man først have regenereret Huffman-træet [16]. Derefter er dekodning triviel. Man starter med at læse den første bit af de komprimerede data. Hvis det er et 0, skal man tage grenen til højre i træet, og modsat hvis det er et 1-tal. Dette fortsættes til man møder et bogstav, og så starter man igen fra toppen.

Hvordan man vælger at gemme information om træet varierer fra implementation til implementation. Normalt gemmer man blot det sorterede skema med frekvenserne. Dermed kan Huffman-træet regenereres som ved kodningen.

Optimale data

Hvor meget Huffman-kodning kan komprimere ens data afhænger selvfølgelig af dataene og deres tilknyttede entropi. Der eksisterer data som Huffman-kodning slet ikke kan komprimere, f.eks. eksemplet fra afsnit 4.2.2 hvor alfabetet består af 256 forskellige symboler med ens frekvens. Huffman-træet bliver så helt fladt, og dermed har vi samme længde kode for alle symbolerne. Da der var 256 forskellige symboler, skal vi bruge en 8 bits kode til hvert symbol, og det er hvad vi normalt skal bruge. Derudover skal vi jo kunne regenerere Huffman-træet, og vi har dermed brugt mere plads end før kodningen.

Hvis Huffman-kodning komprimerer dårligt når træet er fladt, hvordan komprimerer Huffman så når træet er højt, dvs. frekvenserne er meget forskellige? Entropien for sådanne data er lav, hvilket betyder en god kompression idet Huffman-metoden matematisk kan vises at generere optimale, kortest mulige koder der ligger

inden for á bit af entropien [4, s. 101] (forsendelsen af træet skal dog lægges til separat).

Vi kan altså direkte ud fra entropien afgøre om vi kan spare plads ved at bruge Huffman-kodning. Denne egenskab ved Huffman-kodning har gjort den meget benyttet i kombination med andre metoder. Eksempelvis bruger `gzip` og `bzip2` Huffman-kodning som sidste skridt i deres algoritme.

Konklusion

Hvis vi bruger Huffman-kodning på HTML-filer vil vi opnå den optimale kode for denne fil, men vi benytter os ikke af de mønstre som optræder i HTML-filerne. Her ville det være bedre hvis vi optog noget som HTML i alfabetet og så medtog frekvensen for netop dette mønster.

Mønstre optræder ikke kun i HTML-filer, og dette har skabt behovet for andre kompressionsalgoritmer, og dermed måske de mest berømte: Lempel og Zivs algoritmer.

5.2 Ordbogsbaserede metoder

5.2.1 Lempel-Ziv 77

Lempel-Ziv 77 er en ordbogsbaseret algoritme, men den har faktisk ikke en separat ordbog [16]. I stedet består ordbogen blot af de symboler der er forud for det punkt algoritmen er nået til i dataene. Dette gør LZ77 til en adaptiv kompressionsmetode der tilpasser sig dataene og kan skifte mellem forskellige hyppigheder for mønstrene. dette gør den ideel til f.eks. kompression over modemforbindelser da man i disse kan opleve at mønstrenes frekvensfordeling skifter meget (desuden holder ordbogen sig på en konstant størrelse, hvor man f.eks. i LZ78 kan komme ud for at ordbogen bliver for stor).

Algoritmen betragter hele tiden en del af dataene gennem et *vindue* der flytter sig efterhånden som algoritmen går frem. Vinduet spænder over en portion læst data, *søgedelen*, og en portion data som skal til at læses, *indlæsningsdelen*. Søgedelen er så vores ordbog, og er i programmet `gzip` maksimalt på 32 tusinde tegn. Størrelsen af indlæsningsdelen afgør hvor stort et mønster man potentielt kan fange og er i `gzip` på 258 tegn [16, s. 149].

De kodede symboler

LZ77 erstatter tegnene med kodede symboler der består af tre dele: afstanden fra det nuværende punkt til et mønster der passer længere tilbage i dataene, længden af mønstret og endelig det næste tegn, dvs. det tegn der kommer lige efter det mønster der er blevet erstattet. Afstanden er begrænset af størrelsen af søgedelen, mens længden er begrænset af størrelsen af indlæsningsdelen. Man kan opfatte et kodet symbol som en tre-tuppel i form af (afstand, længde, næste tegn).

Grunden til at det næste tegn skal med i det kodede symbol er at man ellers på en eller anden måde skulle angive at et tegn er ved at blive beskrevet i stedet for en tuppel når der dukker et tegn op der ikke forekommer i ordbogen, hvilket sker hyppigt i starten af dataene. Hvis tegnet t ikke forekommer i ordbogen, erstattes det nemlig med (0,0,t).

Der findes dog en variant af LZ77 ved navn LZSS [16, s. 107] (efter opfinderne, Storer og Szymanski) hvor det ekstra tegn ikke tages med. I stedet benytter man tilstandsskift hvor man reserverer en ekstra bit til hver byte for at angive om det er en tuppel eller et tegn.

Gennemgang af eksempel

I vores eksempel

```
<HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML>
```

findes der ikke mønstre større end 6 tegn, men mønstret `HTML>` findes både i slutningen og i begyndelsen af filen, så vi vælger størrelsen af søgedelen til at være på 63 tegn og indlæsningsdelen til 7 tegn. Disse kan beskrives med henholdsvis $\log_2 64 = 6$ bit og $\log_2 8 = 3$ bit. En tre-tuppel vil derfor være på $6 + 3 + 8 = 17$ bit så hver tuppel skal i gennemsnit erstatte 3 eller flere tegn (24 bit) for at vores kompression skal kunne betale sig. Se tabel 5.5 på næste side for skridtene i algoritmen.

De første seks tupler består blot af enkelttegn, og det første interessante sker i linje 7 når algoritmen læser et mønster der kan findes i søgedelen. Her findes ‘<’ seks pladser fra det aktuelle tegn, men da der ikke er flere tegn der passer med mønstret bliver længden kun 1 og næste tegn ‘b’. Herefter hopper algoritmen så over ‘b’ da det allerede er blevet beskrevet.

Hvis der er flere mønstre der passer, vælges det længst mulige naturligvis, og hvis der er flere af samme længde, vælges det længst væk for at forenkle algoritmen.

Vores eksempel ender med at blive til 29 tupler. Da hver tuppel består af 17 bit, får vi i alt 493 bit data ud af LZ77, mod 424 bit før kompression. Dette hænger sammen med at vores eksempel består af små mønstre der kun gentages få gange. Tuplerne optager simpelthen for meget plads, 17 bit, i forhold til de oprindelige 8 bit når der ikke bliver sparet mere på mønstre der gentager sig selv.

LZSS – tilstandsskift

Hvis vi i stedet havde benyttet tilstandsskift i form af LZSS, kunne vi have reduceret tuplernes størrelse, men det kræver til gengæld at vi lader dem bestå af et helt antal byte for at kunne angive om en given byte er en tuppel eller ej. Da vi ikke skal have næste tegn med, kan vi klare os med $6 + 3 = 9$ bit som oprundet giver 16 bit pr. tuppel, mens enkelttegn stadig vil fylde 8 bit.

søgedel		indlæsningsdel	symbol
	<	HTML><b...	(0, 0, <)
<	H	TML><bo...	(0, 0, H)
<H	T	ML><bod...	(0, 0, T)
<HT	M	L><body...	(0, 0, M)
<HTM	L	><body>...	(0, 0, L)
<HTML	>	<body><...	(0, 0, >)
<HTML>	<	body><p...	(6, 1, b)
<HTML><b	o	dy><p>T...	(0, 0, o)
<HTML><bo	d	y><p>Te...	(0, 0, d)
<HTML><bod	y	><p>Tes...	(0, 0, y)
<HTML><body	>	<p>Test...	(6, 2, p)
<HTML><body><p	>	Test␣af...	(9, 1, T)
<HTML><body><p>T	e	st␣af␣e...	(0, 0, e)
<HTML><body><p>Te	s	t␣af␣en...	(0, 0, s)
<HTML><body><p>Tes	t	␣af␣en␣...	(0, 0, t)
<HTML><body><p>Test	␣	af␣en␣H...	(0, 0, ␣)
<HTML><body><p>Test␣	a	f␣en␣HT...	(0, 0, a)
<HTML><body><p>Test␣a	f	␣en␣HTM...	(0, 0, f)
<HTML><body><p>Test␣af	␣	en␣HTML...	(3, 1, e)
<HTML><body><p>Test␣af␣e	n	␣HTML-s...	(0, 0, n)
...HTML><body><p>Test␣af␣en	␣	HTML-si...	(6, 1, H)
...ML><body><p>Test␣af␣en␣H	T	ML-side...	(25, 3, -)
...body><p>Test␣af␣en␣HTML-	s	ide</p>...	(14, 1, i)
...dy><p>Test␣af␣en␣HTML-si	d	e</p></...	(24, 1, e)
...><p>Test␣af␣en␣HTML-side	<	/p></bo...	(35, 1, /)
...p>Test␣af␣en␣HTML-side</	p	></body...	(24, 2, <)
...est␣af␣en␣HTML-side</p><	/	body></...	(4, 1, b)
...t␣af␣en␣HTML-side</p></b	o	dy></HT...	(34, 5, /)
...n␣HTML-side</p></body></	H	TML>	(47, 4, >)

Tabel 5.5: Skridtene i LZ77-algoritmen virkende på eksemplet

Hvis vi igen arbejder vores eksempel igennem, se tabel 5.6, bliver enkelttegn bare oversat til almindelige tegn, og disse er således ikke taget med i tabellen, mens mønstre som genkendes i søgedelen, erstattes med tupler som før bortset fra at næste tegn som nævnt ikke tages med.

søgedel	mønster	symbol
`<HTML>`	`<`	(6, 1)
`<HTML>(6,1)body`	`><`	(6, 2)
`<HTML>(6,1)body(6,2)p`	`>`	(9, 1)
`<HTML>(6,1)body(6,2)p(9,1)`	`T`	(13, 1)
...	...	...
...`(35,1)/(24,2)(4,2)`	`body(6,2)`	(34, 6)
...`/(24,2)(4,2)(34,6)`	`/`	(11, 1)

Tabel 5.6: Udvalgte skridt for LZSS virkende på eksemplet

Så vi får nu 21 tegn og 17 tupler, svarende til $21 \cdot 8 + 17 \cdot 16 = 440$ bit. Ud over dette skal der medtages en ekstra tilstandsbit for hver byte, hvilket i praksis omsættes til en byte mere pr. 8 byte komprimeret data – i eksemplet bliver dette til $7 \cdot 8 = 56$ bit. Altså en total størrelse på 496 bit, hvilket faktisk er en forøgelse i forhold til LZ77. Dette skyldes dog at vi ikke udnyttede de 16 bit pr. tuppel idet de 7 bit af dem blot blev brugt som fyld.

På trods af spildet kom dette eksempel alligevel tæt på LZ77 hvilket ud over de mindre resulterende symboler også skyldes at måden mønstre erstattes på i LZSS ikke er så tilbøjeligt til at bryde mønstre, som det f.eks. kunne ses i de sidste linjer ved tabellen for LZ77. Her blev algoritmen nødt til at tage ‘b’ med i den tredjesidste tuppel hvorved vi mistede muligheden for at erstatte det 6 tegn lange mønster fra begyndelsen af eksemplet og måtte nøjes med 5 tegn.

Reversibilitet

Kompression med LZ77 giver en række tupler som blot kan indlæses enkeltvis og konverteres til almindelige data. Hvis en tuppel består af $(0, 0, t)$, skrives blot tegnet t, ellers hentes først det angivne mønster og indsættes hvorefter det ekstra tegn i tuplen kan tilføjes. Ved LZSS foregår det tilsvarende idet man dog skal holde styr på om en byte er en del af en tuppel eller et tegn for sig. Ordbogen er så de data man regenererer og bliver ved komprimeringen opbygget undervejs.

Optimale data

Eksemplet indeholder mange tupler der kun erstatter é tegn, hvad der er uheldigt for kompressionens effektivitet. Ved lidt større datamængder bliver disse tupler igen erstattet med større tupler hvorved dette problem mindskes. Dette fremgår også i slutningen af eksemplerne.

Størrelsen af søgedelen og indlæsningsdelen hænger nøje sammen med hvor godt LZ77 komprimerer, og er typisk det man vælger som parameter til sit kompressionsprogram. Større søgedel giver mulighed for bedre kompression da ordbogen er større og der dermed er flere muligheder for at finde gentagne mønstre, men samtidig skal den søges igennem fra ende til anden, og dette har en effekt på hastigheden af algoritmen. Desuden skal vi bruge større tal til at referere til ordbogen med, og større tal betyder mere data. Ved forøgelse af indlæsningsdelens størrelse forøger vi også datamængden, men søgetiden bliver faktisk mindre fordi algoritmen i værste fald stopper mønstersøgningen på samme tid og i bedste fald kan fange længere mønstre. Et langt mønster der bliver delt op, betyder jo at hele søgedelen skal søges igennem en ekstra gang.

Man kan dog finde optimale størrelser for hver del, men dette kræver foranalyse af dataene, og en sådan foranalyse tager længere tid end at komprimere dem fordi den kræver at man først skal komprimere dataene uden begrænsning på hver del for så bagefter at finde de maksimale værdier for afstand og længde.

Det giver sig selv at hvis data kun består af et enkelt mønster, gentaget mange gange, så komprimerer LZ77 rigtigt godt, men hvordan hænger dette sammen med entropien for dataene?

Vi så i afsnittet om Huffman-kodning at data hvor hvert symbol har samme frekvens komprimeres dårligt med Huffman-kodning. Hvis vi nu igen antager at vores data har denne egenskab, men at dataene danner mønstre. Hvis f.eks. dataene består af $x_1x_2x_3x_4x_5$ gentaget 10 gange, så har disse data førnævnte egenskab, men alligevel kan dette komprimeres med LZ77. Vi får nemlig (0, 0, x_1) op til (0, 0, x_5) og derefter (5, 44, x_5), dog afhængigt af hvor stor indlæsningsdelen er. Grunden til at dette kan lade sig gøre, er at hvis længdefeltet er større end afstandsfeltet så gentages symbolerne fra søgedelen indtil man når op på længdefeltets værdi. Vi har altså hermed komprimeret 50 tegn med ligelig frekvensfordeling ned til 6 tupler der hver kan beskrives med $3 + 6 + 8 = 17$ bit. Det bliver altså 102 bit mod 400 bit.

Dette kan lade sig gøre fordi LZ77 arbejder på mønstre, altså strenge af symboler, hvor Huffman arbejder på enkelte symboler. Mønstret $x_1x_2x_3x_4x_5$ bliver jo gentaget ofte, frekvensen er faktisk 1, og har dermed et lavt informationsindhold så det kan komprimeres.

Det skal dog nævnes at man kan opnå det samme med Huffman blot ved at lade ens alfabet bestå af dette mønster, se f.eks. eksemplet i afsnit 4.1 på side 17. Men generelt kræver dette igen en foranalyse af dataene. I stedet er det populært med en kombination af de to metoder. Man lader først dataene gennemgå LZ77-kompression og dernæst Huffman-kodning, netop for at opnå den optimale kodning. Eksempler på dette er `gzip` og `zip` [16, s. 149].

5.2.2 Lempel-Ziv 78

LZ78 benytter sig af en seperat ordbog [16], dvs. at der ikke er noget der slettes i ordbogen efterhånden som man komprimerer filen. Dette bevirker at mønstre fra starten af filen stadig kan genkendes i slutningen af filen, men gør at ordbogen som

regel bliver meget stor og dermed gør søgningstiden og kompressionstiden større. Desuden tilføjer LZ78 ikke mønstre til ordbogen på helt samme måde som LZ77.

Ligesom LZ77 erstatter LZ78 mønstre med tupler, dog drejer det sig om to-tupler der indeholder et indeks til ordbogen og det næste tegn. Sidstnævnte har samme formål som i LZ77 og tilsvarende findes der også varianter der ikke arbejder på denne måde. Umiddelbart kunne man tro at man sparer plads i forhold til LZ77 når der kun er 2 felter, men da ordbogen som nævnt tit bliver meget stor, bliver indeksnumrene tilsvarende store og kræver dermed også flere bit pr. mønster. Så en LZ78-tuppel kan godt blive større end en LZ77-tuppel.

LZ78 fungerer ved at et tegn c_1 indlæses og ledes efter i ordbogen. Hvis tegnet ikke findes, indsættes der – analogt med LZ77 – en tuppel med 0 i indeks og c_1 i andet felt, hvorefter tegnet optages i ordbogen. Hvis tegnet c_1 findes i ordbogen, indlæses det næste tegn c_2 og der søges efter mønstret c_1c_2 i ordbogen; hvis det også findes, så $c_1c_2c_3$ osv. Dette fortsætter indtil der opnås et mønster som ikke er i ordbogen, hvorefter der indsættes en tuppel med indekset til det genkendte mønster og det næste tegn c_n. Endelig indsættes dette nye mønster i ordbogen.

Det ses at ordbogen hurtigt kan blive meget stor, men da man ikke behøver at sende den med, se afsnit 5.2.2 på den følgende side, er dette i sig selv ikke et problem, bortset fra effekten på komprimeringstiden og RAM forbruget.

Gennemgang af eksempel

Til at illustrere LZ78 bruger vi igen vores eksempel; de første skridt er vist i tabel 5.2.2. Resten af tabellen kan ses i bilag A.2.1 på side 67.

ordbog	tuppel	mønster
1: <	(0,<)	<
2: H	(0,H)	H
3: T	(0,T)	T
4: M	(0,M)	M
5: L	(0,L)	L
6: >	(0,>)	>
7: <b	(1,b)	<b
8: o	(0,o)	o
9: d	(0,d)	d
10:y	(0,y)	y
11:><	(6,<)	><
12:p	(0,p)	p
13:>T	(6,T)	>T
14:e	(0,e)	e

Tabel 5.7: De første skridt i LZ78 virkende på eksemplet

De første tegn bliver blot erstattet med tupler af formen (0, t), men anden gang

‘<’ bliver fundet, i linje 7, kan de to tegn ‘<’ og ‘b’ erstattes med tuplen (1,b) og indsættes som et mønster på to tegn i ordbogen.

Reversibilitet

Ud fra tuplerne er det muligt at genskabe ordbogen undervejs og dermed få den oprindelige tekst tilbage uden at sende ordbogen med. For hver tuppel læses først indekset – hvis det er et nul, indsættes tegnet i tuplen direkte i ordbogen og indgår desuden som en del af den færdige tekst. Hvis det er forskelligt fra nul, findes først det mønster som indekset svarer til, og sættes sammen med tegnet, hvorefter den færdige streng indsættes i ordbogen og i den færdige tekst. Hvis man nummererer felterne i ordbogen på samme måde som ved komprimeringen, opnås så den oprindelige tekst.

Der er dog en vigtig forskel på kompression og dekompression. Man skal nemlig ikke søge i ordbogen ved dekompression, blot foretage et opslag, og dette gør dekompression væsentligt hurtigere end kompression.

Optimale data

Ligesom LZ77 arbejder LZ78 med mønstre, men ordbogen er anderledes indrettet idet nye mønstre kun tilføjes et bogstav ad gangen, hvad der forårsager at LZ78 er langsommere til at tilpasse sig sandsynligheder for dataene. Så LZ78 virker bedst på større mængder ensartede data hvor mønstrene kan nå at komme i ordbogen. Dette kan være problematisk i forbindelse med HTML-filer der som vi har set i behovsanalysen, kan være forholdsvist små – desuden kan der optræde små blokke, f.eks. lister, med meget ensartet materiale i HTML-filerne, og her vil LZ78 så muligvis ikke kunne nå at tilpasse sig.

Ligesom ved LZ77 er der blevet opfundet, endog endnu flere, varianter på LZ78. Variationerne ændrer bl.a. på hvor mange tegn der tilføjes til genkendte mønstre ved hver forekomst og hvordan ordbogen opretholdes og søges i. Desuden er der varianter med kun ét felt i hver tuppel.

Et eksempel på denne variant kaldes LZW (efter T. Welch [16, s. 123]) og bruger en speciel type ordbog som gør at det kun er nødvendigt med indekset til ordbogen i hver tuppel. Når ordbogen oprettes, starter den med at indeholde alle tegn i alfabetet (f.eks. ‘a’, ‘b’, ‘c’ osv.) – på den måde kan man undgå (0, t)-tupler så det ikke er nødvendigt at tage tegndelen af LZ78-tuplerne med. LZW har dog også en del andre ændringer, f.eks. sendes ordbogen med.

5.2.3 Konklusion

De ordbogsbaserede metoder fanger altså mønstre i data, men dette betyder ikke nødvendigvis at vi har opnået den bedste kompression. F.eks. så burde vi jo ikke bruge lige så meget data på at gemme et mønster der kun bliver brugt én gang som et mønster der bliver mange gange. Derfor benytter man ofte Huffman-kodning på de LZ77/78 komprimerede data, og dette er præcis hvad `gzip` gør.

Andre programmer bruger også kombinationsløsninger med varianter af Lempel-Ziv 77/78, eksempelvis `compress`, `pkzip` og `arj`. Tilmed bruges specialiserede varianter også i modemer og i grafikformater (som GIF og PNG). Men inden for de seneste år er en metode der kan komprimere bedre, dukket op i form af *Burrows-Wheeler-transformationen*.

5.3 Transformationer

5.3.1 Burrows-Wheeler-transformationen

I 1994 opfandt Michael Burrows og D. J. Wheeler en ny metode til at transformere data så de kan komprimeres mere effektivt [16]. Burrows-Wheeler-transformationen sorterer dataene således at der er stor sandsynlighed for at ens symboler optræder i nærheden af hinanden – det vil senere fremgå hvorfor dette resulterer i bedre kompression.

Gennemgang af eksempel

Første punkt er at dele dataene op i blokke. Disse bliver herefter roteret trinvist så man danner en tabel som vist i tabel 5.8 på næste side. Det ses at hver linje i tabellen er den samme som den forrige linje, blot har vi fjernet det første bogstav og sat det på til sidst.

Når den roterede tabel er blevet dannet, skal rækkerne sorteres alfabetisk (leksikografisk). Dette resulterer i tabel 5.9 på side 39 (en komplet udgave fremgår i tabel A.1) på side 68.

Vi skal selvfølgelig have noget information til at gendanne de oprindelige data, men det er faktisk nok blot at gemme den sidste kolonne i tabel 5.9 samt linjenummeret på den linje hvor den oprindelige tekst optræder i den sorterede tabel. I vores tilfælde er dette linje 11 (hele tabellen fremgår i appendiks A.3.1 på side 67). Den sidste kolonne er resultatet af transformationen og er i dette tilfælde:

```
ntfL<<<>>e>>>ypLLyp␣</MMMTTTHHH>␣/<iood␣Tasebb/<-esdd
```

Bemærk at denne streng har samme længde som vores oprindelige eksempel, så vi har altså ikke opnået kompression endnu. Men det ses at de fleste tegn optræder i grupper, hvilket kan udnyttes til at komprimere dem med. Vi har dog ikke gentagelser på mere end 3 bogstaver, så RLE kan ikke betale sig i dette tilfælde. Derimod kan man bruge et princip ved navn *move-to-front* (se på side 40).

Reversibilitet

At dataene kan regenereres er måske ikke umiddelbart klart, men dette hænger sammen med en bestemt egenskab ved de to tabeller. I alle linjerne optræder alle bogstaver fra den oprindelige tekst, blot i en anden rækkefølge. Det samme gør

```
<HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML>
HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML><
TML><body><p>Test␣af␣en␣HTML-side</p></body></HTML><H
ML><body><p>Test␣af␣en␣HTML-side</p></body></HTML><HT
L><body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTM
><body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML
<body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML>
body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><
ody><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><b
                          ⋮
y></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></bod
></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body
</HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body>
/HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body><
HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body></
TML><HTML><body><p>Test␣af␣en␣HTML-side</p></body></H
ML><HTML><body><p>Test␣af␣en␣HTML-side</p></body></HT
L><HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTM
><HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML
```

Tabel 5.8: Eksemplet efter rotation i Burrows-Wheeler-transformationen

sig faktisk gældende for hver kolonne (det ses bedst i tabel 5.8), og selv om man sorterer tabellen vil dette stadig gælde, også for den sidste.

Den konkrete måde man kommer tilbage til den oprindelige tekst på, er ved først at sortere den kodede tekst lexikografisk, således at man får den oprindelige første lodrette række i den sorterede tabel. Under denne proces gemmes for hvert tegn et tal der beskriver hvor bogstavet i den kodede tekst hører til i den lexikografiske tekst. Dvs. '␣' får tildelt værdien 1, det næste '␣' værdien 2, '-' værdien 4 osv. (se tabel 5.9 igen). Det interessante ved den beregnede værdi er at den angiver den række som starter med tegnet i den roterede tabel.

Man kan da danne en tabel som tabel 5.10 på side 40 hvor rækkenummeret, det kodede tegn, placeringen af tegnet i den sorterede tekst og tegnene fra den sorterede kolonne fremgår.

Ved kodningen huskede vi på rækkenummeret hvor den oprindelige besked stod, linje 11. Dermed ved vi også at bogstav nr. 11 i de kodede data er det sidste bogstav i vores oprindelige besked (fordi de kodede data blev dannet af den sidste kolonne), altså '>'. Tallet 16 ud for '>' angiver at den 16. række i den roterede tabel startede med '>'. Husk på at vi ikke lavede om på linjerne, vi flyttede dem blot. Det sidste bogstav i denne række må pga. roteringen imidlertid være det bogstav der stod lige før '>', og da vi i de kodede data har gemt alle de sidste bogstaver i rækkerne, finder vi ved række 16 i tabellen bogstavet før '>', nemlig 'L'. For 'L'

```
␣HTML-side</p></body></HTML><HTML><body><p>Test␣af␣en
␣af␣en␣HTML-side</p></body></HTML><HTML><body><p>Test
␣en␣HTML-side</p></body></HTML><HTML><body><p>Test␣af
-side</p></body></HTML><HTML><body><p>Test␣af␣en␣HTML
/HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body><
/body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p><
/p></body></HTML><HTML><body><p>Test␣af␣en␣HTML-side<
</HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body>
</body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p>
                          ⋮
ody></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></b
ody><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><b
p></body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</
p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><body><
side</p></body></HTML><HTML><body><p>Test␣af␣en␣HTML-
st␣af␣en␣HTML-side</p></body></HTML><HTML><body><p>Te
t␣af␣en␣HTML-side</p></body></HTML><HTML><body><p>Tes
y></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></bod
y><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><bod
```

Tabel 5.9: Eksemplet efter det er blevet sorteret

kan vi på tilsvarende vis finde det bogstav der stod lige før, og dette fortsættes indtil vi har været igennem alle tallene.

Optimale data

Siden Burrows-Wheeler-transformationens styrke er at samle bogstaver i grupper, giver det sig selv at jo mere data der er at samle, dvs. jo større blokke, jo større grupper, og dermed jo bedre kompression, men dette gør selvfølgelig kompressionstiden større.

I eksemplet viser det sig at f.eks. '/' ikke optræder i grupper. Grunden til dette er at Burrows-Wheeler-transformationen grupperer ud fra konteksten for det pågældende symbol, men kun til den højre side. Hvis vi betragter et bogstav x, så vil dette optræde i en gruppering hvis der på højre side af x er høj frekvens af et andet bogstav y, fordi y'erne bliver sorteret efter hinanden og derfor forårsager at x'erne i den sidste kolonne også ligger ved siden af hinanden. Hvis det derimod var på venstre side, betød afhængigheden ikke noget fordi x'erne så er grupperet kolonne nr. 2 som vi ikke gemmer. Dvs. grunden til at '/' er spredt er at til højre for '/' optræder 'p', 'b' og 'H'. Disse tegn er kontekst for '/' og når disse sorteres spredes '/' naturligvis. Dette forklarer også hvorfor 'M' ses i en stor gruppe. Den eneste kontekst 'M' har, er 'L'.

De optimale data for Burrows-Wheeler-transformationen er altså symboler der

række	tegn	værdi	række	tegn	værdi	række	tegn	værdi
1:	n	44	19:	p	48	37:	o	45
2:	t	51	20:	␣	1	38:	o	46
3:	f	42	21:	<	11	39:	d	36
4:	L	23	22:	/	5	40:	␣	3
5:	<	8	23:	M	26	41:	T	32
6:	<	9	24:	M	27	42:	a	33
7:	<	10	25:	M	28	43:	s	49
8:	>	14	26:	T	29	44:	e	40
9:	>	15	27:	T	30	45:	b	34
10:	e	39	28:	T	31	46:	b	35
11:	>	16	29:	H	20	47:	/	7
12:	>	17	30:	H	21	48:	<	13
13:	>	18	31:	H	22	49:	-	4
14:	y	52	32:	>	19	50:	e	41
15:	p	47	33:	␣	2	51:	s	50
16:	L	24	34:	/	6	52:	d	37
17:	L	25	35:	<	12	53:	d	38
18:	y	53	36:	i	43			

Tabel 5.10: Dekodningstabellen for Burrows-Wheeler for eksemplet

hyppigt har samme kontekst til højre. Dette kan også fange mønstre ligesom Lempel-Ziv fordi hyppigt gentagne sekvenser resulterer i ens passager i den kodede tekst – som f.eks. „MMMTTTHHH“ i eksemplet – som derfor kan komprimeres meget.

Der er dog det problem at Burrows-Wheeler-transformationen kræver relativt store blokke for at være effektiv [16, s. 256], hvilket dels betyder langsommere kodning pga. de større datamængder der skal sorteres, dels kan betyde at HTML-filerne er for små til at Burrows-Wheeler-transformationen kan sortere dem effektivt.

5.3.2 Move-to-front

Move-to-front-metoden beskriver tegnene ud fra hvornår de sidst blev set i dataene [16] og er som skræddersyet til Burrows-Wheeler. Den virker ved at vi først opskriver vores alfabet. Hvis vi bruger det samme alfabet som ved eksemplet fra Burrows-Wheeler-transformationen, fås

$$\{<, >, -, /, \sqcup, a, b, d, e, f, H, i, L, M, n, o, p, s, T, t, y\}$$

Vi beskriver nu hvert tegn med dets position i alfabetet. ‘<’ er nr. 1, ‘>’ er nr. 2 osv. Når man så møder et tegn i dataene, bliver dette tegn flyttet op på forreste position i alfabetet. Dette betyder at hvis tegnene ligger i grupper får vi med stor sandsynlighed en masse små tal efter kodning.

alfabet	tegn	kode
<,>,-,/,␣,a,...	n	15
n,<,>,-,/,␣,...	t	20
t,n,<,>,-,/,...	f	12
f,t,n,<,>,-,...	L	15
L,f,t,n,<,>,...	<	5
<,L,f,t,n,>,...	<	0
<,L,f,t,n,>,...	<	0
<,L,f,t,n,>,...	>	5
>,<,L,f,t,n,...	>	0
>,<,L,f,t,n,...	e	13
e,>,<,L,f,t,...	>	1
>,e,<,L,f,t,...	>	0

Tabel 5.11: Move-to-front-kodning på eksemplet fra Burrows-Wheeler-transformationen

Eksempel

Grundet det store alfabet har vi kun vist det første udsnit af alfabetet. Tabel 5.11 viser de første 12 skridt i en move-to-front-kodning på de data vi fik ved Burrows-Wheeler-transformationen, altså den sidste kolonne i tabel 5.9.

Som det ses til sidst, giver move-to-front små værdier når symbolerne optræder i grupper, også selv om der kommer et ‘e’ ind mellem gruppen af ‘>’. Dette er hvad der skaber den skæve sandsynlighedsfordeling som Huffman-kodning kan udnytte.

Reversibilitet

Vi får altså en talrække ved move-to-front-kodning, men vi skal desuden gemme det oprindelige alfabet for at kunne gendanne de oprindelige data. Dette kan blot gøres som en streng af symboler, og det kommer dermed til at fylde det samme som længden af alfabetet.

Vi starter med at opskrive det samme alfabet som vi startede med ved kodning. Dernæst henter vi de første nummer fra koden, henter symbolet der er på denne position i alfabetet og nedskriver dette symbol. Symbolet skal så ligesom ved kodning, flyttes frem i alfabetet. Dernæst gør vi det samme med næste tal i koden. Man kan faktisk gøre nøjagtigt dette ved at betragte tabel 5.11, og så blot opfatte koden som den givne.

5.3.3 Konklusion

Burrows-Wheeler-transformationen resulterer i mange ens tegn tæt på hinanden, og move-to-front transformerer dette til en sekvens hvor forekomsten af små værdier er stor i forhold til store værdier. Høj sandsynlighed for nogle værdier, lille

for andre er så de ideelle data til Huffman-kodning, og denne form for komprimering med Burrows-Wheeler-transformationen, dernæst move-to-front og til sidst Huffman-kodning benyttes i programmet `bzip2` [18].

Man opnår ikke ret meget ved at bruge Burrows-Wheeler-transformationen og move-to-front hver for sig. Burrows-Wheeler-transformationen flytter kun rundt på bogstaverne i dataene og tilføjer et tal, og move-to-front er kun nyttig at bruge på meget specielle typer af data. Men tilsammen er de anvendelige.

Det er dog umiddelbart sværere at udtale sig om hvor meget plads det er nødvendigt at bruge for et givent mønster i modsætning til de ordbogsbaserede algoritmer. For at få en ide om dette skal man finde ud af hvordan den pågældende gruppering ser ud med hensyn til hvor mange indskudte bogstaver der optræder i grupperingen. Desuden skal man så vide hvor tit de enkelte tal fra move-to-front-kodningen optræder, og dermed hvor langt et kodeord de bliver tildelt under Huffman-kodningen.

Med hensyn til kompression af HTML så kan Burrows-Wheeler-transformationen fange de mønstre, uanset størrelse, der er i HTML-filerne. Det kræver dog at de bliver gentaget et vist antal gange, men dette gælder jo også for Lempel-Ziv-algoritmerne. Der kan dog desuden være et problem med den typiske størrelse af en HTML-fil – hvis de er for små, kan Burrows-Wheeler-transformationen vise sig at være for dårlig i forhold til en ordbogsbaseret algoritme.

Kapitel 6

Udvælgelse af algoritme

Vi har i de foregående kapitler foretaget undersøgelser der kan bruges til at opstille krav til den algoritme som bruges til at komprimere HTML-filerne, samt at belyse hvordan eksisterende algoritmer fungerer. Ud fra dette kan vi forsøge at sammensætte en – teoretisk set – bedre løsning end blot de eksisterende generelle algoritmer. En efterfølgende implementering og test vil så vise om vi har gjort de rigtige slutninger.

Vi indleder kapitlet med en gennemgang af hvad man kan komprimere i HTML. Herefter diskuterer vi hvorvidt, og hvorledes dette kan implementeres. Vi fandt desuden i kapitel 3 ud af at vores algoritme ikke kun skal kunne komprimere statisk HTML, men også dynamisk HTML, og dette stiller nogle ekstra krav til den algoritme vi vælger. Algoritmen skal være så hurtig at den i værste tilfælde kan komprimerere dynamiske HTML-filer hver gang de bliver hentet.

6.1 Hvad kan komprimeres i HTML?

I HTML findes redundans og en forudsigelig struktur – begge kan udnyttes til kompression. F.eks. skrives HTML mindst to gange i en HTML-fil, i det første tag og i det sidste. Derudover findes der ting der kan forudsiges ved brug af mindre information. Et eksempel på dette er ‘<’ og ‘>’, der er overflødige hvis man ved at man har at gøre med et tag – dette kunne gøres ved at indføre et flag foran hvert skift. Desuden findes der tags som kun er gyldige hvis de står indenfor andre tags. Et eksempel på dette er „<td>“ som kun kan stå inden for „<table>“.

Der er derfor flere mønstre man bør kunne udnytte i HTML til at frembringe en bedre kompression. Dette kunne til dels være ved at forarbejde HTML-filen inden kompression eller ved at lave en ny kompressionsalgoritme, som er optimeret til mønstre i HTML filer.

Vi har på den baggrund valgt at belyse tre metoder som umiddelbart kan være oplagte til kompression af HTML:

- Dynamisk ordbogsskift

- Ændring af alfabetet så det indeholder tags
- Opdeling af filerne så at tags og tekst komprimeres hver for sig

Vi vil i det følgende uddybe de tre ovenstående metoder og designe en algoritme til komprimering af HTML-filer.

6.2 Dynamisk ordbog

En måde at udnytte forekomster af ord og bogstaver på er ved at benytte en ordbogsbaseret algoritme hvor man dog skifter ordbog dynamisk. Vi forestiller os at man definerer en række nøgleord som forårsager et ordbogsskift hver gang de mødes. Dette kunne være ord som „table“, „font“ og „script“, hvor de parametre som optræder i tag’et har forskellige sandsynligheder. Eksempelvis er der meget lille sandsynlighed for at finde ordet „size“ i forbindelse med nøgleordet „table“, mens det derimod er meget sandsynligt at møde i forbindelse med nøgleordet „font“.

6.2.1 Fremgangsmåde

Algoritmen virker ved at man starter med at bruge en primær ordbog som kommer til at indeholde de tegn og ord der ikke kommer efter et nøgleord. Når algoritmen møder et nøgleord – f.eks. „table“ – skifter den til en „table“-ordbog som indeholder de ord den møder i forbindelse med nøgleordet. Når algoritmen møder „>“ (tag-slut) skifter den tilbage til den primære ordbog. Ordbogen arbejder således med hele ord, således at ordet „size“ vil indgå i „font“-ordbogen.

Fordelen ved dette ville være at de parametre som ofte bruges i et bestemt tag vil få lavere indeksnumre, end hvis de lå i en stor ordbog. Man kunne forestille sig at en ordbog for nøgleordet „table“ ville indeholde ord som „border“, „bgcolor“ og „cellpadding“ der med stor sandsynlighed vil indgå i det pågældende tag. Derved vil der i de komprimerede data indgå et større antal lave indeksnumre hvilket medfører en skævere sandsynlighedsfordeling der kan komprimeres mere effektivt med f.eks. Huffman-kodning.

Endvidere kan en række af de dynamiske ordbøger være indbygget i algoritmen. Ved at gennemgå et stort antal hjemmesider og at gennemgå HTML-standarden kan ordbøgernes grundlæggende indhold bestemmes. Derved sparer man at overføre en vis mængde tekst for hver HTML-fil der overføres.

Der er dog et problem i forbindelse med store og små bogstaver. Hvis algoritmen skal kende alle standard-tags og deres parametre bliver det i sig selv en stor mængde data, men hvis der derudover f.eks. skal tages højde for at ordet „size“ kan optræde i alle mulige kombinationer af store og små bogstaver, f.eks. „size“, „Size“, „sIze“ osv., bliver ordbøgerne for store.

Hvis denne algoritme skal bruges, er det altså nødvendigt at opgive kravet om en totalt tabsfri algoritme fordi man ville være nødt til at vælge én bestemt af kombinationerne. Til sidernes primære formål, at blive fremvist i en browser, ville

der dog ikke ske nogen ændring fordi HTML ikke gør forskel på store eller små bogstaver.

Dette stiller dog nogle større krav til selve implementeringen, fordi man ikke kan tillade at noget af den almindelige tekst bliver opfattet som HTML og dermed bliver ændret. Hvis der f.eks. blev ændret i Javascript, kunne resultatet være fatalt. Derfor skal algoritmen kunne tage højde for alle de fejlkilder der kan være i forbindelse start og slut af HTML-tags.

6.2.2 Konklusion

De ovenstående overvejelser viser at vi ved at udnytte den viden man i forvejen har om HTML-formatet efter alt at dømme, kan opnå en bedre kompression end de generelle metoder. Men samtidig kan vi konkludere at der en lang række situationer der bør overvejes inden en egentlig standard kan defineres og implementeres. Dels kræver den en grundig undersøgelse af HTML-standarden og af hvilke tags der bruges oftest, men der skal også gøres nogle overvejelser med hensyn til at sikre at metoden er robust nok til f.eks. ikke at give problemer hvis der er små fejl i HTML-filerne. Endelig kunne man overveje at bruge forskellige algoritmer til at komprimere de forskellige dele.

Alt i alt vil det blive for omfattende at implementere ovenstående i dette projekt, idet vi ikke har den fornødne tid til at lave benarbejdet inden den egentlige implementering. Derudover kunne man forestille sig at man ved simplere metoder kunne opnå næsten ligeså god kompression, så vi har ikke arbejdet mere med denne algoritme.

6.3 Ændring af alfabetet

En anden metode til at udnytte den viden vi i forvejen har om HTML, er ved at lave en variant af Huffman-kodning. Varianten skal udover at analysere hyppigheden af enkelte bogstaver også analysere hyppigheden af anvendte HTML-tags, dvs. at den skal medtage en foruddefineret mængde mønstre i alfabetet.

Man kunne på den måde forestille sig at mønstret „table“ indgår lige så hyppigt som bogstavet „q“, og at man dermed kunne opnå en meget effektiv kompression af sider der bruger de samme HTML-tags meget, hvad de fleste jo gør.

Problemet med denne idé er at det er tvilvsomt hvor effektiv den vil være i forhold til f.eks. LZ77 og derefter Huffman. LZ77 opdager jo mønstre og på den måde vil et ord der optræder flere gange blive komprimeret betydeligt. Derudover opfanger LZ77 også længere mønstre end blot enkelte ord. Vores forslag ville f.eks. komprimere „size“ godt, men hvis „size=2 face=Verdana“ optræder mere end én gang vil LZ77 komprimere hele dette stykke. På den måde vil den også komprimerere mønstre der opstår i forbindelse med ofte brugte almindelige ord. Vi har på den baggrund valgt ikke at arbejde mere med denne algoritme.

6.4 Opdeling af filerne

Ved at dele HTML-filen op i to dele, hvor den ene del består af HTML-koderne og den anden af tekst, kan vi udnytte vores viden om komprimeringsalgoritmer til at bruge de mest optimale algoritmer på hver del.

Vi kan bruge LZ77 på HTML-delen, da denne er hurtig og vil fange alle mønstre, forudsat at vi lader søgedelen være på størrelse med filen. Dermed har vi reduceret redundansen i HTML-filen. Men Lempel-Ziv 77 kan ikke komprimere almindeligt tekst nær så godt som en algoritme der bruger Burrows-Wheeler-transformationen (dette hænger sammen med at almindelig tekst har flere afhængigheder tegnene indbyrdes end direkte mønstre).

Man kunne bruge Burrows-Wheeler-transformationen på hele filen da den fanger de samme mønstre som LZ77, dog eventuelt undtaget det sidste '>' idet der kan opstå forskellige kontekster til højre for dette tegn fordi et tag kan efterfølges af tekst eller et andet tag. Men en algoritme der benytter Burrows-Wheeler-transformationen er noget langsommere end LZ77, og af denne grund er det ønskværdigt at begrænse brugen til der hvor det gør mest gavn.

Da `bzip2` komprimerer vha. Burrows-Wheeler-transformationen og `gzip` vha. af LZ77, behøver vi ikke at implementere disse algoritmer og kan nøjes med selve opdelingen og sammensætningen af filerne.

6.5 Vores algoritme

Vores algoritme skal altså dele HTML-filer op. Derved forventer vi at kunne opnå bedre kompression end ren brug af `gzip` da vi bruger `bzip2` på tekstdelen, og `gzip` på HTML-tags, da den er hurtigere og bedre til at fange mønstre i filen.

Vores algoritme vil producere 3 dele:

1. En sekvens med HTML-tags
2. En sekvens med tekst
3. Information til samling af de to dele

Informationen til samling af de to dele er nødvendig at have for at kunne gendanne dataene. Da der kun er to dele, kan vi nøjes med en række tal der angiver skiftevist hvor stor den næste del af tekst eller HTML er. Hvis der står to tags ved siden af hinanden, kan tekstdelen i mellem blot angives til at have størrelsen 0. På denne måde kan vi desuden undlade at gemme de omgivende '<' og '>'. Et eksempel hvor vi ved at vi starter med et HTML-tag kan ses i tabel 6.1 på næste side.

Da taldelen består af tal med samme størrelse, er det nærliggende at forsøge at komprimere den ved hjælp af Huffman-kodning – de mange små tags i filerne burde i hvert fald give en skæv fordeling.

Tekstdel	HTML-del	Samlingsdel	Læst tekst
			<HTML><body> Test af en HTML-side </body></HTML>
	HTML	4	<body> Test af en HTML-side </body></HTML>
	HTML	4,0	<body> Test af en HTML-side </body></HTML>
	HTMLbody	4,0,4	Test af en HTML-side </body></HTML>
	HTMLbody	4,0,4,0	Test af en HTML-side </body></HTML>
	HTMLbodyp	4,0,4,0,1	Test af en HTML-side</p></body></HTML>
Test af en HTML-side	HTMLbodyp	4,0,4,0,1,20	</p></body></HTML>
Test af en HTML-side	HTMLbodyp/p	4,0,4,0,1,20,2	</body></HTML>
Test af en HTML-side	HTMLbodyp/p	4,0,4,0,1,20,2,0	</body></HTML>
Test af en HTML-side	HTMLbodyp/p/body	4,0,4,0,1,20,2,0,5	</HTML>
Test af en HTML-side	HTMLbodyp/p/body	4,0,4,0,1,20,2,0,5,0	</HTML>
Test af en HTML-side	HTMLbodyp/p/body/HTML	4,0,4,0,1,20,2,0,5,0,5	

Tabel 6.1: Et eksempel på vores algoritmes ønskede virkemåde

Kapitel 7

Implementering

Implementationen af algoritmen skal kunne opdele en HTML-fil således at vi kan efterprøve om det kan betale sig at komprimere tekst for sig og HTML for sig. Derudover skal den give en talliste til gendannelse af HTML-filen.

Til selve programkoden er brugt SML fordi gruppen sideløbende med projektet har modtaget undervisning i dette sprog så udgangspunktet er fælles. For at få et overblik over vores programkode starter vi ved med at gennemgå de enkelte SML-funktioner.

7.1 Gennemgang af algoritme i pseudokode

Programmet bygger på to funktioner – en til at opdele en streng indeholdende en webside til almindelig tekst, HTML og en liste med sekvenslængderne, og en til at samle de tre elementer til en streng igen.

De to funktioner arbejder på lister af enkelte bogstaver. Man kunne godt implementere funktionerne således at de arbejdede direkte på strenge, men dette er enten noget besværligt i SML eller også så langsomt at en konvertering fra en streng til en liste af bogstaver foregår meget hurtigere.

Funktionerne ser således ud:

INITIALISERING:
    sæt en lokal variabel *ErTekst* = sand
    sæt en lokal variabel *AntalBogstaver* = 0
OPSPLITNING(DATA):
    hvis næste tegn er ‘<’ og *ErTekst* = sand
        sæt *ErTekst* = falsk
        tilføj *AntalBogstaver* til *Talliste*
        sæt *AntalBogstaver* = 0
        kald OPSPLITNING(MED TEGNET FJERNET FRA DATA)
    ellers hvis næste tegn er ‘>’ og *ErTekst* = falsk

```
          sæt ErTekst = sand
          tilføj AntalBogstaver til Talliste
          sæt AntalBogstaver = 0
          kald OPSPLITNING(MED TEGNET FJERNET FRA DATA)
     ellers hvis ErTekst = sand
          tilføj næste tegn til Tekstliste
          forøg AntalBogstaver med 1
          kald OPSPLITNING(MED TEGNET FJERNET FRA DATA)
     ellers
          tilføj næste tegn til HTML-liste
          forøg AntalBogstaver med 1
          kald OPSPLITNING(MED TEGNET FJERNET FRA DATA)
     når vi er løbet tør for Data
          returner HTML-liste, Tekstliste og Talliste

INITIALISERING:
     find antallet af elementer i Talliste
     hvis det er et lige antal
          sæt ErTekst = falsk
     ellers
          sæt ErTekst = sand
SAMLING(HTML-LISTE, TEKSTLISTE, TAL::TALLISTE):
     hvis ErTekst = sand og Tal = 0
          sæt ErTekst = falsk
          tilføj et '>' til Output
          kald SAMLING(HTML-LISTE, TEKSTLISTE, TALLISTE)
     ellers hvis ErTekst = sand
          tilføj det nuværende tegn fra Tekstliste til Output
          kald SAMLING(HTML-LISTE, TEKSTLISTE, (TAL−1)::TALLISTE)
     ellers hvis ErTekst = falsk og Tal = 0
          sæt ErTekst = sand
          tilføj et '<' til Output
          kald SAMLING(HTML-LISTE, TEKSTLISTE, TALLISTE)
     ellers hvis ErTekst = falsk
          tilføj det nuværende tegn fra HTML-liste til Output
          kald SAMLING(HTML-LISTE, TEKSTLISTE, (TAL−1)::TALLISTE)
     når både HTML-liste og Tekstliste er tomme
          returner Output
```

Bemærk at opsplitningen og gendannelsen returnerer lister i modsat rækkefølge af de data som bliver givet til funktionen som parameter. Dette er gjort af hastighedsmæssige årsager; man kan nemlig hurtigt tilføje et element til en liste i starten af listen, men hvis man skal tilføje et element til slutningen af listen skal

listen faktisk gendannes helt fra ny af. Med hensyn til kompression har dette ingen betydning da alle mønstre bliver transformeret ens.

7.2 Øvrige funktioner

Da vores funktioner arbejder på lister, skal vi desuden have funktioner der varetager forbindelsen fra filer over tekststrenge til lister og tilbage igen, for at kunne hente og gemme dataene.

Af disse funktioner vil vi kun beskrive funktionerne der indlæser tallisten fra talfilen og gemmer den, da bitoperationer er noget indviklede i SML – tallene gemmes med 16 bits præcision (hvilket svarer til at der må være 65536 tegn mellem et ’<’ og et ’>’ eller omvendt).

```
INITIALISERING(FILNAVN):
    lav en binær Instream til Filnavn
HENT TAL FRA FIL(INSTREAM):
    hvis vi er ved slutningen af Instream
        returner Liste
    ellers
        sæt Tal1 = de næste 8 bit fra Instream
        konverter Tal1 til et heltal
        sæt Tal1 = Tal1·256
        sæt Tal2 = de næste 8 bit fra Instream
        konverter Tal2 til et heltal
        sæt Tal2 = Tal1+Tal2
        cons Tal2 på Liste
    HENT TAL FRA FIL(INSTREAM)

INITIALISERING(FILNAVN):
    lav en binær Outstream til Filnavn
GEM TALLISTE I FIL(TAL::TALLISTE):
    hvis Talliste er tom
        luk og skriv Outstream
    ellers
        sæt NytTal = Tal heltalsdivideret med 256
        lav NytTal til 8 bit
        gem NytTal i Outstream
        sæt NytTal = Tal modulo 256
        lav NytTal til 8 bit
        gem NytTal i Outstream
    GEM TALLISTE I FIL(TALLISTE)
```

Grunden til at vi foretager en heltalsdivion med 256 når vi gemmer tallet, er at dette svarer til at skifte de enkelte bit i tallet 8 pladser til højre (et højreskift svarer til at heltalsdividere med 2, altså $\frac{x}{256} = \frac{x}{2^8} = \frac{x}{2\cdot2\cdot2\cdot2\cdot2\cdot2\cdot2\cdot2}$). Hvis man har et tal på 16 bit eller derunder, svarer de 8 skift til at fjerne de 8 sidste bit og gøre de andre 8 tilgængelige som et tal mellem 0 og 255 der kan gemmes som en enkelt byte – når vi så henter tallet igen, skal vi naturligvis gange med 256 for at få det oprindelige tal igen. Tilsvarende henter modulo-operationen de første 8 bit information ud.

7.3 Programmets grænseflade

Grænsefladen fremgår i tabel 7.3 på den følgende side.

7.4 Afvikling af programmet

Vores program gemmer tre filer bestående af henholdsvis HTML, tekst og tal. Tallene er gemt med 16 bit og har dermed en maksimal størrelse på $2^{16} = 65536$. Dette vil sige at der maksimalt må være 2^{16} tegn mellem et tag og et stykke tekst i de HTML-filer vi kører programmet på.

Programmet kan køres med kommandoen „./split -s filnavn" og „./split -u filnavn" hvor „-s" betyder split og „-u" betyder unsplit. Filnavnet man skal give programmet, er filnavnet på den oprindelige HTML-fil, eksempelvis „jubii.html". Programmet laver ved „-s" tre filer: HTML-delen bliver gemt i „filnavn.1", tekst-delen bliver gemt i „filnavn.2" og tallisten bliver gemt i „filnavn.3". Når man kalder programmet med „-u", skriver det „filnavn.orig". Dette er valgt således vi nemt kunne verificere at den gendannede fil var lig med den oprindelige fil.

7.5 Test

Efter implementation af algoritmen skal den naturligvis testes og holdes op med simplere løsninger som blot komprimerer med `gzip` eller `bzip2` hver for sig. For at algoritmen kan betale sig at bruge, er der visse krav.

Den skal som minimum give et bedre kompressionsforhold end `gzip`, ellers ville `gzip` være bedre end vores algoritme både med hensyn til hastighed og kompressionsforhold. Desuden skal den være hurtigere end `bzip2` da denne isoleret sandsynligvis vil give et bedre kompressionsforhold.

På dette grundlag undersøger vi hvorvidt vores forarbejdning af HTML-filer vil forbedre eksisterende algoritmers pakkeevne og hvor tidskrævende pakkearbejdet er. Som testfiler bruger vi de seks hjemmesider vi valgte i behovsanalysen, se appendiks B.1 på side 70.

Vores implementation forarbejder kun dataene, og de skal så derefter komprimeres. Tekstdelen af HTML-filen komprimeres med `bzip2`, mens HTML-delen komprimeres med `gzip`.

Funktion	Specifikation	Kommentar
stringToList:	string -> char list	Laver en streng om til en liste af chars.
listToString:	char list -> string	Laver en liste af chars om til en streng.
listsToStrings:	char list * char list * ’a -> string * string * ’a	Laver de to første lister af chars om til strenge.
stringsToLists:	string * string * ’a -> char list * char list * ’a	Laver de to første strenge om til lister af chars.
loadFile:	string -> string	Henter en fil ind som en streng.
saveFile:	string * string -> unit	Gemmer en streng til en fil.
loadNum:	string -> int list	Henter en fil ind 16 bit ad gangen, til en int list. Dette er funktionen HENT TAL FRA FIL fra pseudokoden.
saveNum:	string * int list -> unit	Gemmer en liste af heltal som 16 bit værdier i en fil. Dette er funktionen GEM TALLISTE I FIL fra pseudokoden.
splitList:	char list * char list * char list * int list * int * bool -> char list * char list * int list	Opdeler en char liste til 2 lister af chars og en liste af heltal. Dette er funktionen OPDELING fra pseudokoden.
concatList:	char list * (char list * char list * int list) * bool -> char list	Samler 2 char lister og en liste af heltal til en char liste. Dette er funktionen SAMLING fra pseudokoden.
splitFile:	string -> unit	Opdeler en fil og skriver de 2 filer med hhv. HTML og tekst og en talfil.
concatFile:	string * string * string -> unit	Samler de 3 filer og skriver den oprindelige fil.
mainprogram:	unit -> unit	Funktionen der udføres ved start af programmet fra kommandolinien.

Tabel 7.1: Liste over funktioner, deres typer og beskrivelse som de ser ud i kildekoden

7.5.1 Komprimeringsevne

Test af komprimeringsevnen afhænger kun af komprimeringsprogrammet og testfilerne og ikke af testmaskinen i modsætning til en test af kompressionshastigheden. Komprimeringsprogrammerne der er brugt i testen, er `gzip` 1.2.4 og `bzip2` 0.9.0c. Taldelen komprimeres med et SML-program gruppen har udviklet der giver størrelsen af en fil efter Huffman-kodning (det var nødvendigt med et specialindrettet program da størrelsen af symbolerne i alfabetet er på to byte). Resultatet ses i tabel 7.2.

HTML-side	stør. (byte)	`gzip` (%)	`bzip2` (%)	egen (%)
Jubii	43452	18,33	**17,85**	23,40
Carlsberg	6976	**36,91**	39,51	44,14
ITL	18350	**22,82**	24,38	28,00
IT-Avisen	30795	14,97	**14,78**	18,77
Jyllandsposten	66159	18,14	**16,22**	21.17
The Voice	27995	17,03	**16,94**	21,38

Tabel 7.2: Kompressionsforholdet for de forskellige algoritmer

Det ses at `bzip2` i 4 ud af 6 tilfælde, ved de 4 største filer, var mere effektiv til at komprimere end `gzip`. Det fremgår dog at der ikke er stor forskel mellem kompressionsforholdene ved `gzip` og `bzip2` – formodentlig på grund af testfilernes størrelse. Vores egen algoritme er imidlertid klart dårligst i alle test. En del af grunden til dette finder vi i tabel 7.3.

HTML-side	HTML (%)	tekst (%)	tal (%)
Jubii	17,6	27,1	37,5
Carlsberg	44,7	41,7	100,0
ITL	25,7	28,4	51,8
IT-Avisen	13,8	17,4	48,6
Jyllandsposten	14,3	25,6	36,3
The Voice	19,7	14,2	45,9

Tabel 7.3: Kompressionsforholdene for de forskellige dele ved egen algoritme

I tabel 7.3 ses at talfilen ikke kan komprimeres forholdsmæssigt nær så meget som de andre dele. Dette kan hænge sammen med at Huffman-kodning i sig selv måske ikke er tilstrækkeligt – en nærmere undersøgelse som vi dog ikke har foretaget, ville sandsynligvis kunne finde en bedre algoritme eller kombination af algoritmer. Uanset hvor lille filen bliver, er det dog ikke nok til at vores algoritme er mere effektiv end `bzip2`.

Et andet problem kunne være at noget af alfabetet til siden overføres både i `bzip2`- og `gzip`-delen hvilket skaber redundans. Trods dette viser det sig dog at

tekstfilen komprimeret med `bzip2` og HTML-filen komprimeret med `gzip` tilsammen i nogle tilfælde er mindre end den oprindelige fil komprimeret med `gzip`.

Endelig skal det bemærkes at `gzip` i nogle tilfælde pakker tekstdelen bedre end `bzip2` – det undergraver ideen bag vores algoritme med at `bzip2` skulle være mere effektiv til tekstkompression end `gzip`. At `bzip2` ikke er det i alle tilfælde, skyldes sandsynligvis filernes størrelse – for at `bzip2` skal være effektiv, skal andelen af helt små tal som move-to-front returnerer, være tilpas stor så entropien bliver lille. Dette sker kun ved store blokke af ens bogstaver hvilket er sjældent i små filer.

7.5.2 Tidsaspektet

I modsætning til komprimeringsevnen afhænger testen af tidsaspektet også af omgivelserne den er udført i, f.eks. styresystemet, processoren, osv. Testen af tidsaspektet blev udført på en maskine som vi forestiller os kunne være en realistisk webserver, se B.4 på side 73 for nærmere detaljer.

test nr.	egen	`gzip`	`bzip2`
1	4,89	65.15	14,22
2	4,90	65,19	14,19
3	4,90	65,32	14,23
gnm.	4,90	**65,22**	14,21

Tabel 7.4: Test af kompressionshastighederne (i filer/s)

Tiden det tager at komprimere med vores algoritme, er afhængig af dels forarbejdningen som vores SML-program foretager, og dels den egentlige komprimering af de enkelte elementer. Resultatet fremgår i tabel 7.4. Det skal bemærkes at vi i testen ikke har komprimeret selve taldelen fordi vores SML-program kun kunne give størrelsen af den resulterende fil og ikke skrive den (pakning af bit er ikke helt let i SML). Men det ses at det er uden betydning idet vores algoritme i forvejen er langsommere.

Af testen kan vi se at `gzip` er klart hurtigere end `bzip2` som yderligere er hurtigere end vores egen algoritme. Spørgsmålet er hvad det er ved vores algoritme som gør den så meget langsommere – tabel 7.5 på næste side viser andelen af udførselstiden for de forskellige dele af algoritmen. Det ses at akilleshælen i vores algoritme er opdelingsdelen som tager 83,24% af den samlede tid.

7.5.3 Sammenfatning

Testen viser at vores algoritme ikke komprimerer bedre end eksisterende løsninger. Problemet ligger i talfilen; vores algoritme kan sandsynligvis forbedres på dette område, enten bruge end mere sofistikeret metode end Huffman-kodning alene eller ved at ændre algoritmen helt så talfilen ikke er nødvendig. Man kunne f.eks. klare

test nr.	opdeling (%)	`gzip` (%)	`bzip2` (%)
1	83,25	5,16	11,59
2	83,26	5,14	11,60
3	83,22	5,13	11,56
gennemsnit	**83,24**	5,14	11,58

Tabel 7.5: Andelen af udførselstiden for de enkelte dele i vores algoritme, `gzip` *er for komprimeringen af HTML-delen,* `bzip2` *er for komprimeringen af tekstdelen*

skiftene mellem tekst og HTML ved at reservere en ASCII-værdi som ikke bliver brugt i HTML-filerne, og indsætte en ekstra byte med denne værdi alle de steder hvor der skal skiftes i filerne.

Hvis vi ser helt bort fra talfilen, komprimerer vores algoritme en smule bedre. Det kan hænge sammen med at `bzip2` bedre kan komprimere de mønstre som findes i tekstdelen end `gzip` kan, men det kan også være fordi vi fjerner alle '>' og '<'.

Der er dog en ulempe ved at bruge to forskellige algoritmer; alfabetet skal overføres to gange. Specielt ved små filer udgør alfabetet en betydelig del af filstørrelsen. Derudover kan man overveje om der er nogen grund til at opdele filerne, når vi i afsnit 7.5.1 bemærkede at `gzip` i nogle tilfælde komprimerer bedre på begge filer end `bzip2`. I behovsanalysen konkluderede vi at HTML-filerne generelt er forholdsvis små, hvilket igen taler for `gzip`.

Udførselstiden for vores algoritme er også problematisk. Den langsomme udførselstid kommer fra vores opdelingsprogram i SML, og spørgsmålet er om det skyldes en dårlig implementation eller om det er en dårlig algortime.

7.5.4 Algoritmens tidskompleksitet

For at få klarlagt hvorfor vores implementation var så langsom under testen, undersøger vi om det er selve algoritmens struktur der gør at det tager lang tid.

Dette kan teoretisk tidskompleksitet svare på [1] – tidskompleksiteten for en algoritme beskriver forholdet mellem mængden af data og udførselstiden for algoritmen. Man beskriver dette vha. af O-notationen, f.eks. $O(n^2)$. Dette betyder at hvis $T(n)$ er tiden det tager at udføre en funktion når størrelsen af dataene er n, så eksisterer der et k og et n_0 så for $n \geq n_0$ gælder:

$$T(n) \leq kf(n),$$

hvor $f(n)$ er det vi skriver i O-notationen, altså $O(f(n))$.

En operation der ikke afhænger størrelsen af dataene angives med $O(1)$, dvs. den tager 1 tidsenhed. Hvis man har flere operationer, kan man slå dem sammen, og resultatet bliver blot den største. Eksempelvis hvis man har en operation der

tager $O(1)$ og $O(n)$, så er summen blot $O(n)$. Desuden gælder at $O(kf(n))$ er ækvivalent med $O(f(n))$ – en konstant faktor har ingen betydning.

På denne baggrund kan vi nu analysere vores algoritme. Da vi blot ønsker en øvre grænse for hvor lang tid vores algoritme vil tage, undersøger vi den med hensyn til værste tilfælde. De første funktioner laver lister om til strenge og strenge om til lister, og det er som sagt ligegyldigt om vi gør dette ved é eller flere lister, så dette har tidskompleksiteten $O(n)$. Det samme gælder det at hente dataene og gemme dem i filerne.

Det virkelige spørgsmål er om vores opdelingsfunktion også opfører sig lineært. Pseudokoden (på side 48) viser at algoritmen består en betingelskonstruktion hvor hver forgrening udgøres af to eller tre operationer plus et kald til funktionen selv med et element mindre i dataene. Operationerne består af tildeling og tilføjelse af en nyt element til en liste, og begge dele er $O(1)$ (idet sidstnævnte er en tilføjelse til begyndelsen af listen).

OPSPLITNING er altså en rekursiv funktion som gentages med datamængden reduceret med é for hver iteration, og da vi kun foretager $O(1)$-operationer inde i funktionen, betyder dette at tidskompleksiteten for OPSPLITNING er en konstant gange antallet af operationer, dvs. $O(n)$.

Nuvel, dette må betyde at hele vores programs kompleksitet er $O(n)$.

7.5.5 Konklusion

Årsagen til at vores metode er langsom, er altså ikke algoritmen i sig selv da den kan udføres i overskuelig tid. Derfor må problemet ligge et sted i implementationen. Sandsynligvis er de listeoperationer som programmet er bygget op ved hjælp af – eftersom det er det mest naturlige i et funktionelt sprog som SML – for ineffektive i forhold til de datastrukturer man ville bruge ved en implementation i et imperativt sprog som C eller C++.

Hvis vores algoritme blev optimeret med hensyn til hastighed, så burde den som det ses, ikke være beregningsmæssig tungere end de andre programmer som vi kører i testen, faktisk en del lettere da Lempel-Ziv-algoritmerne involverer en del søgning og Burrows-Wheeler-transformationen sortering som er $O(n \log n)$.

Hvorvidt vores algoritme tilsammen med de kompressionsalgoritmer vi bruger, er hurtigere end ren brug af `bzip2` er sværere at sige, og det afhænger selvfølgelig af mængden af tekst i forhold til HTML efter opdelingen. Hvis der f.eks. kun er é tegn HTML og resten er tekst, vil vores kombinerede metode ikke være hurtigere end ren brug af `bzip2`, men dette er selvfølgelig usandsynligt.

Men vi skal i alt bruge under hvad `bzip2` bruger af tid, altså $\frac{1}{14{,}21/\mathrm{s}} \approx 0{,}0704$ s ifølge tabel 7.4 på side 54. Hvis fordelingen bliver 50% til hver del, vil dette altså sige at vi har $0{,}0704\ \mathrm{s} - (\frac{1}{2} \cdot \frac{1}{65{,}22/\mathrm{s}} + \frac{1}{2} \cdot 0{,}0704\ \mathrm{s}) = 0{,}0275$ s til opdelingen. Dette svarer til at vores opsplitter skal kunne tage 36,33 filer pr. sek, eller omkring halvdelen af hvad `gzip` viste sig at kunne klare. Så ved denne fordeling synes det rimeligt at dette kan lade sig gøre.

Kapitel 8

Praktiske overvejelser

I dette kapitel gør vi os nogle overvejelser om den praktiske anvendelighed af vores resultater – er der overhovedet nogen der vil bruge kompression af HTML-sider?

Det viser sig at det er der, for der findes faktisk allerede løsninger på markedet, HTTP 1.1-standarden indeholder bl.a. et afsnit om emnet (mere om det i afsnit 8.3). Men det er langt fra alle der udnytter muligheden for kompression, se f.eks. en mindre undersøgelse af udbredelse af HTTP 1.1-kompression i Danmark i appendiks D på side 75 – det viser sig at ingen af de 20 største steder i Danmark benytter kompression. Hvorfor bliver det ikke brugt noget mere?

For at klarlægge dette spørgsmål gennemgår vi først fordele og ulemper for HTML-kompression generelt, både for bruger og for udbyder, dernæst for de specifikke fordele og ulemper ved brug af muligheden i HTTP 1.1-standarden.

8.1 Fordele og ulemper for udbyder

Udbyderne vil naturligvis kun bruge løsningen hvis det kan betale sig økonomisk. Det kan det hvis udbyderen kan spare penge ved at indføre HTML-kompression i stedet for en internetforbindelse med større båndbredde.

Det kan dog være at de skal ansætte en tekniker for at installere løsningen, og dette koster ekstra penge. Det kan også vise sig at kompression af HTML sætter så store beregningsmæssige krav til udbyderens maskinpark at udgifterne til nye computere bliver større end udgifterne til en ny forbindelse.

En større forbindelse koster dog flere penge hvert kvartal, mens prisen for en systemopgraderin afhænger af hvor hurtigt hardwaren bliver forældet. Så rentabiliteten afhænger af skæringspunktet mellem de to.

Vores hastighedstest har vist at det er muligt at komprimere 65 sider pr. sekund, mens f.eks. `jubii.dk` har omkring 26 mill. sidevisninger pr. uge (for uge 48, 2001 [6]) svarende til netop gennemsnitligt ca. 65 sidevisninger pr. sekund[1], og `jp.dk` har omkring 2,7 mill. svarende til 6-7 pr. sekund. For Jubii er en enkelt

[1]Hvis man regner med at besøgstiden kun er 16 timer om dagen: $\frac{26.000.000/\mathrm{uge}}{7 \cdot 16 \cdot 60 \cdot 60\mathrm{s}/\mathrm{uge}} \approx 65/\mathrm{s}$

maskine af samme kaliber som ved testen altså ikke nok (spidsbelastningen må være en hel del højere end den gennemsnitlige), men det ses at ydelsen er inden for rækkevidde, selv ved Jubii som leverer langt flest sider i Danmark pr. uge. Priseksemplet for Cybercity Erhverv i tabel 3.4 på side 15 viser at anskaffelse af f.eks. en enkelt server sagtens ville kunne tjenes ind på en besparelse med hensyn til linjen.

Endeligt kræver løsningen i øvrigt at brugerne kan dekomprimere HTML-filerne, og udbyderen skal være sikker på at dette er tilfældet for at undgå at miste besøgende.

8.2 Brugerne

For at få brugerne til at acceptere kompression skal de motiveres ved enten at kunne opleve en hastighedsmæssig gevinst ved brugen eller simpelthen ikke at kunne se visse sider. Eller evt. blot ikke opdage at det sker.

Kompression af HTML nedbringer trafikken til udbyderen så brugeren vil opleve at det går hurtigere at hente siden ned. Vi har udført en test, se tabel 8.1 (selve undersøgelsen er dokumenteret i appendiks B.3 på side 72), som viser at tiden kan nedbringes væsentligt ved at hente HTML-filerne komprimeret, også selv om der allerede bruges kompression på den forbindelse man bruger. Især for modem hvor vi f.eks. har reduceret tiden for Jyllandspostens side fra 15,3 sekunder til 3,41 sekunder. Denne tid er dog uden tiden for kompression/dekompression, men denne vil være lille pga. af filernes størrelse, se evt. hvor mange filer man kan komprimere pr. sekund i tabel 7.4 på side 54.

HTML-side	modem (s)	ISDN (s)	ADSL (s)	filstør. (kb)
Carlsberg	0,47	0,05	0,06	2,5
The Voice	1,14	0,12	0,14	4,7
IT-Avisen	1,23	0,12	0,13	4,6
ITL	0,96	0,10	0,12	4,2
Jyllandsposten	3,41	0,48	0,41	12,0
Jubii	2,24	0,24	0,25	7,7

Tabel 8.1: Overførselstiderne for de komprimerede HTML-filer

Man kunne mene at reduktionerne for de hurtigere forbindelser er så små at de er uden betydning, men faktisk skal svartiderne i programmer generelt være på under 0,2 sekunder for at de virker som om de reagerer flydende [2].

Problemet med at brugerne skal kunne dekomprimere HTML-filerne kan løses ved f.eks. et plugin til en browser – dette giver en næsten transparent løsning da plugin automatisk kan installeres og således ikke kræver meget viden af brugeren. Under alle omstændigheder skal det være meget let at installere løsningen – man kan næppe forvente meget mere end et klik på „OK“ af den jævne bruger.

Selve udførselstiden for dekomprimeringen betyder ikke noget da den for de undersøgte algoritmer er så kort at det ikke kan mærkes med en almindelig størrelse HTML-side.

8.3 Eksisterende løsninger

Med HTTP 1.1 [7] blev der indført en mulighed for kompression mellem webservere og browsere. Standarden blev færdig i juni 1999, og stort set alle browsere og HTTP-servere siden den dato understøtter den [11] og de komprimeringsmetoder den definerer: `gzip`, `compress` og `deflate`, alle baseret på Lempel-Ziv-algoritmerne.

Idet webserver og browser nu har en standard at knytte sig til når de skal udveksle komprimeret data, er det blevet nemt at komprimere HTML. Kompressionsløsninger findes til de to mest brugte webservere, Microsoft Internet Information Server (IIS) og Apache, som udgør tæt på 90% af samtlige webservere på internettet [12].

Da både servere og browsere understøtter denne standard, skal vi ikke længere bekymre os om omkostninger og andre problemer i forbindelse med installation hos udbyderne og brugerne. Det vil sige at hvis man benytter HTTP 1.1-kompression, er den eneste tilbageværende ulempe at der stadig stilles større krav til serverne hos udbyderne.

Men hvis dette er den eneste ulempe hvorfor benytter alle udbyderne det så ikke?

8.4 Anvendelse af HTTP 1.1-kompression

Til at besvare dette spørgsmål kan vi benytte en teknologianalytisk model. Problemet kan ligge inden for fire forskellige aspekter: organisationen, teknikken, produktet eller viden. Et organisatorisk problem kan ligge i rækkefølgen i arbejdsprocessens faser og koordineringen af arbejdsdelingen. Et vidensmæssigt problem kan være mangel på viden og teoretisk indsigt i en teknologi. Et teknisk problem kan være mangel på arbejdskraft eller arbejdsmidler f.eks. værktøj og maskiner. Og endelig kan et produktmæssigt problem ligge i et kvalitativt problem eller i de tre førnævnte da produktet er resultatet af disse.

Vi mener ikke at det kan være et teknisk problem da udbydere allerede har folk til at administrere deres server, og da løsningen allerede er lavet, så de skal blot installere et modul til webserveren. Beregningskraft er desuden blot et spørgsmål om penge, så et direkte problem ligger heller ikke i arbejdsmidlerne, blot kan disse være for dyre.

Organisatorisk kan der være et kommunikationsproblem hvis ledelsen er dårlig til at følge arbejdsprocessen. Der er måske blevet ansat for få folk og de ikke har tid til at undersøge løsningerne, og dermed ikke ved at de eksisterer.

Produktmæssigt mener vi heller ikke at der er et problem da kompression i HTTP 1.1 kan foregå med en anerkendt og udbredt standard, nemlig `gzip` som vi faktisk argumenterede for brugen af i afsnit 6.4 på side 46.

Det kan dog være at udbyderne enten ikke ved at de kan spare penge ved at komprimere deres sider, eller som førnævnt, at der overhovedet eksisterer en løsning. Altså direkte eller indirekte et vidensmæssigt problem – ideen om at komprimere HTML og HTTP 1.1's understøttelse af det synes at være en velbevaret hemmelighed. Yderligere kan det være at udbyderne tror at den største del af deres trafik er billeder og lyd, og dermed ikke tror at HTML-kompression vil være økonomisk gunstigt, selvom vi tidligere i rapporten har vist at dette ikke behøver at være tilfældet.

Vi har på bagrund af denne hypotese spurgt udbydere af nogle få udvalgte større websites om hvorfor de ikke bruger kompression (undersøgelsen er beskrevet i appendiks E på side 76). Ud af de tre adspurgte udbydere var der kun én som gav indtryk af at have undersøgt muligheden nøjere. Vi tolker svarene således at denne faktisk havde undersøgt potentialet nærmere og foretaget en test, mens de to andre var helt eller delvist uvidende om muligheden.

Med mere viden burde der være en stor chance for at udbyderne vil bruge kompression af HTML når det er del af en allerede accepteret standard, ved at sætte et kryds i et indstillingsvindue i Microsofts IIS eller installerede `mod_gzip` til Apache. Eftersom alle nyere browsere understøtter standarden, er det i hvert fald klart at initiativet ligger hos udbyderne.

Med mere viden hviler anvendelsen af kompression af HTML udelukkende på at en evt. investering i regnekraft er billigere end at investere i ekstra båndbredde. En sådan analyse afhænger naturligvis af både af tid og sted og må derfor foretages ved den enkelte udbyder.

Kapitel 9

Konklusion

9.1 Resultater fra rapporten

Efter den korte introduktion til kompression har vi foretaget en analyse af behovet for kompression af HTML-sider. Den viser at der for brugerne kan være en gevinst i form af en tidsbesparelse – altså en oplevelse af et hurtigere internet. For udbyderne af siderne kan der være en betydelig økonomisk gevinst hvis komprimeringen er effektiv nok, både med hensyn til at mindske størrelsen af store og små sider, men også med hensyn til hastighed.

For at få et grundlag for forståelse af hvor meget og hvorfor HTML kan komprimeres, har vi undersøgt emnet informationsteori, og vi kan konstatere at den mængde data man skal bruge til at repræsentere et tegn med, er indenfor *entropien* af dataene plus [illegible]. Vi har også konstateret at entropien kan reduceres yderligere ved at tage større blokke af tegn i betragtning, hvis der i dataene forefindes afhængigheder imellem de enkelte tegn. Da vi desuden har konstateret at der er afhængigheder og redundans i HTML, kan vi konkludere at HTML kan komprimeres.

På denne baggrund har vi undersøgt forskellige algoritmer for at klarlægge deres styrker og svagheder i forbindelse med kompression af HTML. Vi har undersøgt en algoritme, Huffman-kodning, der kan bruges til at kode dataene så størrelsen når ned i det optimale område.

Vi undersøgt ordbogsbaserede algoritmer og har fundet ud af at de udmærker sig ved kompression af ens blokke bestående af flere tegn, *mønstre*, og derfor er gode til HTML-tags. Derudover kan de med fordel kombineres med Huffman sådan at styrkerne ved de to principper forenes. LZ77 er en af disse algoritmer og i kombination med Huffman implementeret i det udbredte program `gzip`.

Derudover har vi undersøgt en transformation, Burrows-wheeler-transformationen, der efterfulgt af et princip ved navn move-to-front kan resultere i data der kan komprimeres meget effektivt med Huffman-kodning. De tre algoritmer er kombineret i programmet `bzip2`.

Efter undersøgelse af de forskellige kompressionsalgoritmer har vi beskrevet tre mulige løsninger som kunne tænke sig at være brugbare. Ud af disse har vi

valgt at implementere den sidste, nemlig opdeling af HTML-filen i tre, tags, tekst og en kontrolfil, for at se om det kan betale sig at komprimere de forskellige dele af filen hver for sig. På baggrund af vores undersøgelse af algoritmer har vi valgt at bruge `gzip` på HTML-delen og `bzip2` på tekstdelen. Kontrolfilen består af en række tal som vi komprimerer med Huffman-kodning.

Efter implementering af vores løsning har vi fundet ud af at vores algoritme ikke komprimerer bedre end eksisterende løsninger. Man kan dog muligvis løse dette problem ved at skære taldelen fra, for i stedet at markere overgangen fra HTML til tekst med et flag. Desværre vidste det sig at vores implementering var meget langsom, ved at undersøge vores implementering med teoretisk tidskompleksitet fandt vi ud af at problemet skyldtes implementeringen af splitteren.

For at finde ud af om vores løsning vil blive brugt har vi gjort os nogle praktiske overvejelser, hvor vi har fundet ud af at med mere viden burde der være en stor chance for at udbyderne vil bruge HTML-komprimering. Specielt når det er en del af en allerede accepteret standard som HTTP 1.1 som fjerner de fleste ulemper udbyderne og brugerne kan komme ud for. Det eneste tilbageværende problem er de investeringer i beregningskraft som udbyderne muligvis skal foretage. Det virker dog som om det ville være væsentligt billigere end at investere i ekstra båndbredde.

9.2 Perspektivering

Der er for os ingen tvivl om at komprimering af HTML pt. er en god idé Der er penge at spare for udbyderne og siderne vil indlæses hurtigere for brugerne.

Man kan naturligvis argumentere for at man i takt med udviklingen kan få hurtigere internetforbindelser til den samme pris. Men det samme gælder processorer, og rentabiliteten i HTML-kompression afhænger derfor af udviklingen i forholdet mellem de to.

Et helt andet problem er at fordelingen i datamængder måske i fremtiden bliver ændret mere i retning af andre, tungere formater end blot tekstbaserede HTML-sider. Men så vil det til gengæld være relevant at forsøge at komprimere de formater, som det også allerede sker med video og lyd.

Det ville kræve en nærmere undersøgelse at kunne fastslå med sikkerhed hvorfor udbyderne ikke i højere grad benytter HTTP 1.1-kompression. Hvis de større udbydere begynder at udnytte muligheden, virker det dog sandsynligt at kendskabet også vil brede sig til mindre.

Litteratur

[1] John E. Hopcroft og Jeffrey D. Ullman Alfred V. Aho. *Data structures and algorithms*. Addison-Wesley Publishing Company, 1987.

[2] Peter Bickford. Human interface online: Worth the wait?, 1999. http://developer.netscape.com/viewsource/bickford_wait.htm.

[3] Patrick Chen. Modem tutorial - data compression protocols, 1991. http://www.ics.mq.edu.au/docs/general/technical/modem/MT-Compression.ht%ml.

[4] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[5] Cybercity erhverv fastnet internetforbindelser, nov 2001. http://erhverv.cybercity.dk/produkter/fastnet/internetforbindelse/.

[6] Gallup website index. http://www.fdim.dk/?vis=ind.

[7] R. Fielding, J. Gettys, J. Mandul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.txt.

[8] Geocrawler.com - linux-isdn - isdn compression. http://www.geocrawler.com/mail/msg.php3?msg_id=1373637&list=33.

[9] Acc systems custom test - isdn - compression test. http://www.etestinglabs.com/main/reports/snci/reports/accrept/accrep03.%htm.

[10] Jean loup Gailly. The gzip home page. http://www.gzip.org/.

[11] mod_gzip for the apache web server. http://www.remotecommunications.com/apache/mod_gzip/.

[12] Netcraft web server survey. http://www.netcraft.com/survey/.

[13] Netcraft web site finder. http://www.netcraft.com/.

[14] RFC 1974, aug 1996. ftp://ftp.isi.edu/in-notes/rfc1974.txt.

[15] RFC 2516, feb 1999. ftp://ftp.isi.edu/in-notes/rfc2516.txt.

[16] David Salomon. *Data Compression – The complete reference*. Springer-Verlag, 1997.

[17] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann Publishers, Inc., 1996.

[18] Julian Seward. The bzip2 and libbzip2 home page. http://sources.redhat.com/bzip2/.

[19] Telestyrelsen. Data til offentliggørelse, 2001. http://www.telestyrelsen.dk/dk/Data_til_offentliggorelse12001.xls.

Bilag A

Eksempler

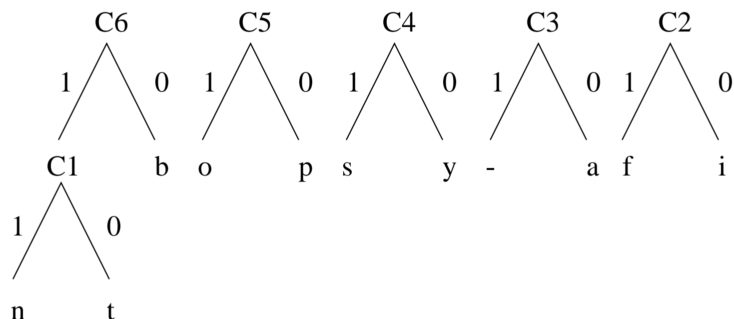
A.1 Eksempler fra statistiske metoder

A.1.1 Fortsat eksempel fra Huffman-kodning

Her fortsættes eksemplet fra afsnittet om Huffman-kodning. Vi sluttede ved:

Symbol	<	>	C6	C5	C4	T	/
Forekomster	6	6	4	4	4	4	3
$p(x)$	0,1132	0,1132	0,0754	0,0754	0,0754	0,0754	0,0566
Symbol	␣	d	e	H	L	M	C3
Forekomster	3	3	3	3	3	3	2
$p(x)$	0,0566	0,0566	0,0566	0,0566	0,0566	0,0566	0,0377
Symbol	C2						
Forekomster	2						
$p(x)$	0,0377						

Og træet ses på figur A.1:

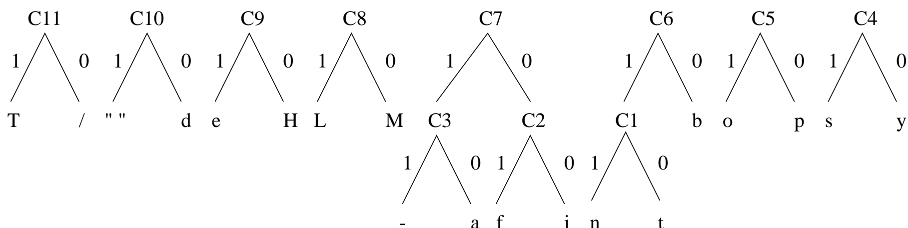


Figur A.1: Træet som det så ud efter C6

De næste 5 skridt giver en ny tabel.

Symbol	C11	C10	C9	C8	<	>	C7
Forekomster	7	6	6	6	6	6	4
$p(x)$	0,1320	0,1132	0,1132	0,1132	0,1132	0,1132	0,0754
Symbol	C6	C5	C4				
Forekomster	4	4	4				
$p(x)$	0,0754	0,0754	0,0754				

Træet ses nu på figur A.2. Bemærk at " " svarer til ␣.

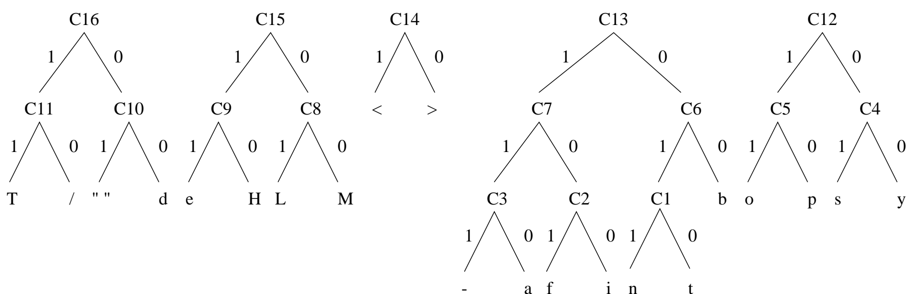


Figur A.2: Træet som det ser ud efter C11

Vi tager endnu 5 iterationer og får en ny tabel.

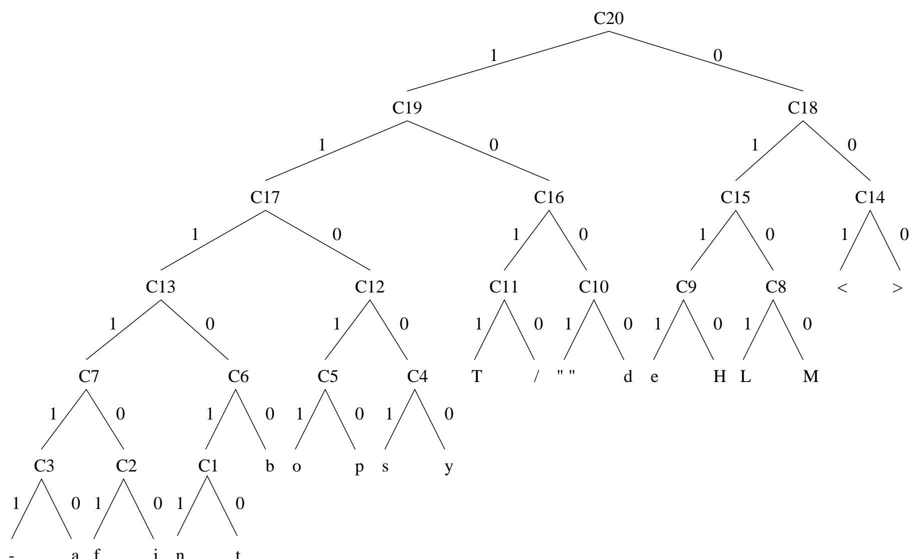
Symbol	C16	C15	C14	C13	C12
Forekomster	13	12	12	8	8
$p(x)$	0,2452	0,2264	0,2264	0,1509	0,1509

Træet ses nu på figur A.3.



Figur A.3: Træet som det ser ud efter C16

Vi mangler nu blot 4 iterationer for at få det fulde træ. Det sidste kombinerede symbol C20 har dermed frekvensen 1, dvs. at alle symboler ligge i dette træ. Dette ses på figur A.4.



Figur A.4: Det færdige Huffman-træ

A.2 Eksempler fra ordbogsbaserede metoder

A.2.1 Eksempel fra Lempel-Ziv 78

Det fulde skema fra LZ78. Læg mærke til at der ikke er forskel på tuplerne, også selvom det er et større mønster man genkender.

ordbog	tuppel	x	ordbog	tuppel	x	ordbog	tuppel	x
1:<	(0,<)	<	13:>T	(6,T)	>T	25:si	(15,i)	si
2:H	(0,H)	H	14:e	(0,e)	e	26:de	(9,e)	de
3:T	(0,T)	T	15:s	(0,s)	s	27:</	(1,/)	</
4:M	(0,M)	M	16:t	(0,t)	t	28:p>	(12,>)	p>
5:L	(0,L)	L	17:␣	(0,␣)	␣	29:</b	(27,b)	</b
6:>	(0,>)	>	18:a	(0,a)	a	30:od	(8,d)	od
7:<b	(1,b)	<b	19:t	(0,t)	f	31:y>	(10,>)	y>
8:o	(0,o)	o	20:␣e	(17,e)	␣e	32:</H	(27,H)	</H
9:d	(0,d)	d	21:n	(0,n)	n	33:TML	(23,L)	TML
10:y	(0,y)	y	22:␣H	(17,H)	␣H	34:>	(0,>)	>
11:><	(6,<)	><	23:TM	(3,M)	TM			
12:p	(0,p)	p	24:L-	(5,-)	L-			

A.3 Eksempler fra transformationer

A.3.1 Eksempel fra Burrows-Wheeler

Her bringes den fulde sorterede tabel delt op i to dele, tabel A.1 og tabel A.2.

```
1   ␣HTML-side</p></body></HTML><HTML><body><p>Test␣af␣en
2   ␣af␣en␣HTML-side</p></body></HTML><HTML><body><p>Test
3   ␣en␣HTML-side</p></body></HTML><HTML><body><p>Test␣af
4   -side</p></body></HTML><HTML><body><p>Test␣af␣en␣HTML
5   /HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body><
6   /body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p><
7   /p></body></HTML><HTML><body><p>Test␣af␣en␣HTML-side<
8   </HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body>
9   </body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p>
10  </p></body></HTML><HTML><body><p>Test␣af␣en␣HTML-side
11  <HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML>
12  <body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML>
13  <p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><body>
14  ></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body
15  ></body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</p
16  ><HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML
17  ><body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML
18  ><p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><body
19  >Test␣af␣en␣HTML-side</p></body></HTML><HTML><body><p
20  HTML-side</p></body></HTML><HTML><body><p>Test␣af␣en␣
21  HTML><HTML><body><p>Test␣af␣en␣HTML-side</p></body></
22  HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTML><
23  L-side</p></body></HTML><HTML><body><p>Test␣af␣en␣HTM
24  L><HTML><body><p>Test␣af␣en␣HTML-side</p></body></HTM
25  L><body><p>Test␣af␣en␣HTML-side</p></body></HTML><HTM
26  ML-side</p></body></HTML><HTML><body><p>Test␣af␣en␣HT
27  ML><HTML><body><p>Test␣af␣en␣HTML-side</p></body></HT
28  ML><body><p>Test␣af␣en␣HTML-side</p></body></HTML><HT
29  TML-side</p></body></HTML><HTML><body><p>Test␣af␣en␣H
30  TML><HTML><body><p>Test␣af␣en␣HTML-side</p></body></H
31  TML><body><p>Test␣af␣en␣HTML-side</p></body></HTML><H
32  Test␣af␣en␣HTML-side</p></body></HTML><HTML><body><p>
                              ⋮
```

Tabel A.1: Første del af den fulde sorterede tabel fra eksemplet

	⋮
33	af␣en␣HTML-side </body></HTML><HTML><body> Test␣
34	body></HTML><HTML><body> Test␣af␣en␣HTML-side </
35	body> Test␣af␣en␣HTML-side </body></HTML><HTML><
36	de </body></HTML><HTML><body> Test␣af␣en␣HTML-si
37	dy></HTML><HTML><body> Test␣af␣en␣HTML-side </bo
38	dy> Test␣af␣en␣HTML-side </body></HTML><HTML><bo
39	e </body></HTML><HTML><body> Test␣af␣en␣HTML-sid
30	en␣HTML-side </body></HTML><HTML><body> Test␣af␣
41	est␣af␣en␣HTML-side </body></HTML><HTML><body> T
42	f␣en␣HTML-side </body></HTML><HTML><body> Test␣a
43	ide </body></HTML><HTML><body> Test␣af␣en␣HTML-s
44	n␣HTML-side </body></HTML><HTML><body> Test␣af␣e
45	ody></HTML><HTML><body> Test␣af␣en␣HTML-side </b
46	ody> Test␣af␣en␣HTML-side </body></HTML><HTML><b
47	p></body></HTML><HTML><body><p>Test␣af␣en␣HTML-side</
48	p>Test␣af␣en␣HTML-side</p></body></HTML><HTML><body><
49	side </body></HTML><HTML><body> Test␣af␣en␣HTML-
50	st␣af␣en␣HTML-side </body></HTML><HTML><body> Te
51	t␣af␣en␣HTML-side </body></HTML><HTML><body> Tes
52	y></HTML><HTML><body> Test␣af␣en␣HTML-side </bod
53	y> Test␣af␣en␣HTML-side </body></HTML><HTML><bod

Tabel A.2: Anden del af den fulde sorterede tabel fra eksemplet

Bilag B

Beskrivelse af undersøgelser

I dette bilag er der nærmere beskrivelser af de forskellige undersøgelser, alle programmer skrevet af gruppen kan findes på den vedlagte cd.

B.1 Testsiderne

Til de undersøgelser hvor der er brug for testsider har vi valgt forsider fra carlsberg.dk, it-avisen.dk, itl.dk, jp.dk, thevoice.dk samt en søgeside fra Jubii.dk.

Disse sider dækker et meget bredt udvalg af indhold, og vi mener derfor at de er repræsentative for det generelle udbud af websider. I udvalget er der både sider med mange billeder, f.eks. the Voice, mens andre består mere af tekst, som f.eks. JP.

Jubii.dk er en søgeside på linje med yahoo.com (eller .dk) og altavista.com. Den har overvejende meget tekst, men en del af siden består også af reklamer i form af billeder. Siden vi har valgt er forsiden idet den er rimelig repræsentativ for hele sitet. Reklamebillederne går igen over hele siden og forsiden har forholdsvis meget tekst.

JP.dk er Jyllandspostens hjemmeside og består mere af tekst end af billeder. Der er stadig reklamer på siden, og JP har også gjort brug af billeder til artiklerne på siden som blikfang. Siden der er valgt her, er igen forsiden, da vi mener den er repræsentativ for resten af sitet. Siden ligner i øvrigt andre avisers hjemmesider og kan derfor også til en hvis grad stå som repræsentant.

ITL.dk – fra denne side har vi ikke valgt forsiden fordi den ikke repræsenterer de underliggende sider så godt. De andre sider har en smule tekst med beskrivelse af firmaet og hvad det kan tilbyde af webløsninger. Som testside har vi her valgt at bruge siden „totalløsninger“ under „IT-løsninger“.

IT-Avisen.dk er en nyhedstjeneste og siden består derfor mest af tekst. Antallet af billeder er begrænset til et par reklamer og små thumbnails. Den udvalgte side er forsiden.

TheVoice.dk ligger i den anden ende af skalaen med hensyn til mængden af billeder kontra mængden af tekst. Forsiden er blevet valgt som testside, og den har en overvældende mængde af billeder. Siden kan siges at være repræsentativ for mange af de sider som skal henvende sig til unge mennesker. Den indeholder en masse flotte farver og effekter som kan hjælpe til med at tiltrække folk fra denne aldersgruppe.

Carlsberg.dk – Carlsbergs danske hjemmeside består hovedsageligt af et Flash-dokument. Der er en HTML-startside der leder besøgende hen til hovedsiden, eller fortæller dem hvor de kan hente et Flash-plugin til deres browser. Siden repræsenterer derfor bedst af alt de Flash-sider som man kan finde på nættet idag (hvilket er en stadig voksende del).

B.2 Datasammensætning i logfiler

For at finde sammensætningen af dataene i de udvalgte logfiler skrev vi et program i Perl. Logfilerne bestod af en evt. beskrivelse i toppen og ellers et stort antal linjer med samme struktur. Programmerne er bygget ens op, de består af først indsamling af data fra logfiler og beregninger på dem, derefter formatering og til sidst en visning af resultatet.

B.2.1 Proxyserveren granit.but.auc.dk

Idet proxyserveren står mellem netværket og internettet, registrerer den alt webtrafik.

Først sorteres alt data fra som ikke er blevet sendt med status „ok" (serverkode 200), desuden sorteres data fra med størrelsen 0, idet det må antages at være en fejl. Dernæst har vi valgt at sorteres dataene i tre kategorier: billeder, HTML, som igen bliver delt op i dynamisk og statisk, og andet data. Analysekriterierne er valgt med den antagelse at proxyserveren kan finde ud af at inddele filerne efter de rigtige typer.

Billederne bliver matchet ud fra et felt i logfilen som specificerer typen. Desuden viste det sig at typen „x-httpd-html" også var billeder som var blevet benævnt forkert.

HTML var straks lidt vanskeligere. Filer med endelsen „.css" eller „.xml" bliver i første omgang identificeret som statisk HTML, mens filer med typen „javascript" bliver sorteret som dynamisk HTML. Resten af HTML-filerne bliver først matchet på at de har typen HTML. Da dynamiske sider ofte bliver sendt med parametre, er de sværere at matche direkte på navne-endelsen, selv med Perls regulære udtryk. Derfor matchede vi derefter statiske sider på HTML-lignende endelser (f.eks. „.htm", „.html" osv.) og antog at resten var dynamisk HTML (selvfølgelig stadig inden for typen HTML). Dette er ikke helt korrekt idet vi f.eks. ikke kan vide om „http://www.somesite.com/" er dynamisk eller statisk, men pga. de om-

talte problemer med at matche dynamiske sider på endelsen, bliver eksemplet talt som en dynamisk side.

Resten af dataene antages at være andet data.

Efter at have sorteret dataene i typer skal procentdelen af data som proxyserverens cache fanger, bestemmes. Dette kunne ske direkte efter om serveren kunne finde filerne i lageret.

B.2.2 IT-Avisen

Indholdet på IT-Avisens sider viste sig at være rent dynamisk, og da websitet desuden hverken bruger lyd eller video, skulle filerne kun opdeles i billeder og dynamisk HTML.

Først sorteres alt fra som ikke blev sendt med statuskode „ok“, ligesom ved proxyserveren. Desuden har IT-Avisen et samarbejde med `msn.dk` som gør at de får ekstra trafik. Denne trafik bliver også sorteret fra. Billederne bliver matchet på endelserne „.jpg“, „.gif“ og „.swf“ og dynamisk HTML bliver matchet på endelsen „.asp“.

B.3 Overførselstid for forskellige forbindelser

Til at teste overførselstiden brugte vi programmet `GNU wget` (det kan findes på `www.gnu.org/software/`) som er et kommandolinjeprogram til at hente filer ned over FTP og HTTP. `Wget` benytter ingen form for lokal cache, imodsætning til alm. browsere. Når en fil er hentet, bliver overførselshastigheden og størrelsen af filen udskrevet hvorfra man så kan beregne overførselstiden.

Selve proceduren for forsøget var som følger: vi anbragte de seks testfiler i `pub_html`-kataloget hos et gruppemedlem og desuden på `sourceforge.net` (for at se om der var nogen forskel i hvor dataene fysisk var placeret, Sourceforge har sæde i Amerika), og gik derefter hjem og hentede hver fil over HTTP vha. `wget` tre gange efter hinanden fra hver server med de forskellige forbindelser uden anden belastning af linjerne.

Den gennemsnitlige hastighed for hver overførsel som rapporteret af `wget` blev noteret hvorefter vi beregnede gennemsnittene for de tre forsøg med hver fil. Det viste sig så at værdierne for Sourceforge og for basisuddannelsens webserver næsten ikke adskilte sig fra hinanden (og i hvert fald ikke i nogen entydig retning), hvorfor de værdier som fremgår i tabel 3.2 på side 12 blot er gennemsnittene af de to.

Selve forsøgene blev foretaget om aftenen – der kan naturligvis være et vist udsving i hastigheden afhængigt af tidspunktet da den til dels afhænger af hvor mange andre der bruger linjerne, men vi har anset en mere kontrolleret, grundigere test (f.eks. på forskellige tidspunkter) for at være for omfatttende i betragtning af at de konklusioner vi drager fra dem ikke afhænger af at de er helt præcise.

Vi har desuden undersøgt tiderne for komprimerede HTML-filer på nøjagtigt samme måde som ved de ukomprimerede sider, bortset fra at filerne er komprimeret med „`gzip` filnavn“ før overførslerne.

B.4 Test af kompressionshastighed

Maskinen til testen havde følgende relevante hardware- og softwarespecifikationer:

- Pentium 3, 750 MHz (100 MHz bus), 256 Mb ram
- Linux, v. 2.4.16 af kernen
- `gzip` 1.3.2, `bzip2` 1.0.1

For at få et gennemsnitsoverblik udførte vi tidstagningerne på 10 tilfældigt valgte HTML-filer fra de udvalgte testsider 100 gange vha. et Perl-program der startede kompressionsprocesserne op parallelt vha. systemkaldet FORK. Dette blev gjort for at give et billede af hvor mange komprimeringer en webserver vil kunne udføre pr. sekund. Testen blev udført tre gange, og resultatet kan ses i tabel 7.4.

Bilag C

Kompressionstest

HTML-side	ukomprimeret	`gzip`	`gzip -9`	`bzip2`	Huffman
Jubii	43452	7963	7937	**7757**	
– HTML	30495	5360	**5312**	5441	
– tekst	10267	2783	**2726**	2752	
– tal	5382	1395	1388	**1266**	2017
Carlsberg	6976	**2575**	**2575**	2756	
– HTML	4181	**1867**	**1867**	2002	
– tekst	2727	1065	**1064**	1136	
– tal	138	125	125	**115**	138
ITL	18350	4188	**4166**	4473	
– HTML	12872	3042	**3031**	3315	
– tekst	4762	1200	**1195**	1353	
– tal	1434	466	466	**454**	743
IT-Avisen	30795	4615	4543	**4542**	
– HTML	18536	2563	**2534**	2717	
– tekst	10497	1967	**1918**	1942	
– tal	2626	777	777	**747**	1275
Jyllandsposten	66159	12004	11888	**10728**	
– HTML	42523	6085	**5997**	6019	
– tekst	19662	5368	5274	**5038**	
– tal	7950	2182	2155	**2002**	2887
The Voice	27995	4767	**4686**	4741	
– HTML	17396	3422	**3407**	3581	
– tekst	9249	1303	**1277**	1312	
– tal	2702	810	807	**758**	1240

Tabel C.1: Størrelsen (i byte) af forskellige dele af siderne, ukomprimeret og komprimeret

Bilag D

Udbredelse af HTML-kompression

Tabel D.1 indeholder resultatet af en undersøgelse af de 20 mest besøgte sider fra `www.fdim.dk`'s indeks i uge 49 for om de benytter kompression.

website	komprimerer	website	komprimerer
jubii.dk	nej	infopaq.dk	nej
sol.dk	nej	jp.dk	nej
opasia.dk	nej	dsb.dk	nej
tv2.dk	nej	politiken.dk	nej
metropol.dk	nej	scor.dk	nej
krak.dk	nej	dba.dk	nej
ofir.dk	nej	tiscali.dk	nej
dr.dk	nej	edbpriser.dk	nej
degulesider.dk	nej	borsen.dk	nej
ekstrabladet.dk	nej	dmi.dk	nej

Tabel D.1: Undersøgelse af udbredelsen af kompression

Vi har benyttet kommandoen

wget -S –header='Accept-encoding: gzip, deflate' http://www.adresse.dk

til at foretage undersøgelsen. Parametrene fortæller, at vi kan modtage komprimeret data, og at vi vil se svarene fra serveren. Hvis serveren svarer med "Content-encoding: gzip", betyder det, at den vil sende dataene komprimeret.

Bilag E

Telefoninterviews

Vi har foretaget en række telefoninterviews med forskellige udbydere. Alle interviews er foretaget d. 17. december 2001.

E.1 www.tv2.dk

Deltagere: Jasper Juhl (studerende, AAU) og Peter Stoffer (webserveransvarlig, TV2 Interactive)

Forundersøgelse: Forud for samtalen har vi undersøgt hvilken platform TV2.dk bruger og om deres forside bruger kompression. TV2.dk bruger Apache/1.3.22 (Red-Hat/Linux) [13] og benytter ikke kompression af deres forside som det ses i tabel D.1 på forrige side).

Formål: At undersøge hvorfor der ikke bruges kompression på TV2.dk.

Referat: Samtalen indledes af Jasper Juhl som præsenterer sig og redegør for den forundersøgelse der er foretaget. Indledende spørgsmål fra Jasper er om Peter/TV2 har overvejet at benytte kompression. Peter redegør for at TV2 har været og er i overvejelse omkring brugen af kompressionen. TV2 har foretaget interne test på en lille del af deres webside, som viste at softwaren i den Apache de benytter var for ustabilt. Den fik iflg. Peter deres Java Server Pages (jsp) til at gå ned. Derfor blev kompression ikke implementeret. Peter afslutter dog med at fortælle at de venter på Apache 2.0, som efter hvad han havde hørt skulle indeholde et mere stabilt forhold til kompression. Herefter ville de foretage en ny test og tage kompression op til fornyet overvejelse.

E.2 www.dating.dk

Deltagere: Jasper Juhl (studerende, AAU) og Henrik Back (webserveransvarlig, Freeway/Dating.dk)

Forundersøgelse: Forud for samtalen har vi undersøgt hvilken platform dating.dk bruger og om deres forside bruger kompression. Dating.dk bruger Microsoft-IIS/5.0 på Windows 2000 [13] og benytter ikke kompression af deres forside.

Formål: At undersøge hvorfor der ikke bruges kompression på dating.dk.

Referat: Samtalen indledes af Jasper Juhl som præsenterer sig og redegør for den forundersøgelse der er foretaget. Indledende spørgsmål fra Jasper er om Henrik/Freeway har overvejet at benytte kompression. Det har Henrik ikke, men han hævder dog at have kendskab til kompression. Han virker dog ikke særlig overbevisende; mest af alt virker det som om han ikke har kendskab til kompressionsmuligheden. Derfor takker Jasper for svarene, idet han vurder at han kan risikere at træde Henrik over tæerne ved at spørge mere til kompressionsmuligheden.

E.3 www.ofir.dk

Deltagere: Jasper Juhl (studerende, AAU) og Per Christiansen (webserveransvarlig, Ofir.dk)

Forundersøgelse: Forud for samtalen har vi undersøgt hvilken platform ofir.dk bruger og om deres forside bruger kompression. Ofir.dk bruger Microsoft-IIS/5.0 on Windows 2000 [13] og benytter ikke kompression af deres forside.

Formål: At undersøge hvorfor der ikke bruges kompression på ofir.dk.

Referat: Samtalen indledes af Jasper Juhl som præsenterer sig og redegør for den forundersøgelse der er foretaget. Indledende spørgsmål fra Jasper er om Per/Ofir har overvejet at benytte kompression. Per svarer "både ja og nej". Han mener der skulle en relativ stor investering i form af arbejdstimer til for at implementere en egentlig kompression. Denne investering har de umiddelbart ikke lyst til at foretage, da han mener at den samlede udgift ville være større end den besparelse der ville være på internetforbindelsen. Han vil dog ikke udelukke at man i fremtiden ville gå over til at benytte kompression, men det skulle være for at tilgodese brugerne, altså for at give et bedre indtryk af Ofir.dk.