Komprimering av HTML-dokument.

(HS-IDA-EA-98-118)

Mattias Runsten (a95matru@ida.his.se)

Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Examensarbete på programmet för systemprogrammering under vårterminen 1998.

Handledare: Henrik Gustavsson

Komprimering av HTML-dokument

Examensrapport inlämnad av Mattias Runsten till Högskolan i Skövde, för Kandidatsexamen (BSc) vid Institutionen för Datavetenskap.

1998-06-11

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Komprimering av HTML-dokument.

Mattias Runsten (a95matru@ida.his.se)

Key words: Compression, Data-transfer, HTML-documents

Abstract

This report describes why transmission of digital data over the PSTN is problematic, and how data compression can be used to increase the speed of the transmission. Data compression is already used to reduce the size of data to be transmitted, e.g. pictures are often in JPEG-format or some other compressed picture format. This report is focused on compression of HTML-documents, since these are not currently compressed before transmission. First a number of compression methods are examined and then the algorithm that appears to be most suitable for compression of HTML-documents is selected to be implemented and tested. The compression algorithm is modified to perform better compression on HTML-documents by using a special combination of lossy compression and "unused" ASCII-codes.

Data is currently compressed by the modem before it is sent, but the hypothesis of this report is that compression is more effective if performed by software on the processor of the computer to which the modem is connected. Therefore the algorithm is compared to the MNP-5 and V.42bis protocols used for data compression in modems. Since the algorithm appears to compress HTML-documents better than both of the protocols mentioned above, it could be used as a complement to them. It can not replace the protocols since it cn only compress HTML-documents and nothing else.

Innehållsförteckning

Sammanfattning **1**
1 Introduktion **2**
2 Bakgrund **4**
2.1 Bakgrund till komprimering 4
2.2 Datakommunikation över telenätet 5
2.2.1 Modem 5
2.2.2 Modulering 5
2.2.3 Komprimering i modem 8
2.3 Datarepresentation 8
2.3.1 Textkodning 9
2.3.2 Teckenkodning 10
2.3.3 Binär kodning 10
2.4 Grundläggande komprimeringsteori 10
2.5 Metoder 11
2.5.1 Packed decimal 12
2.5.2 Relativ komprimering 12
2.5.3 Run-length komprimering 12
2.5.4 Ziv-Lempel komprimering 13
2.5.5 Huffmankomprimering 14
2.5.6 Dynamisk Huffmankomprimering 15
2.5.7 Aritmetisk komprimering 15
3 Problemdefinition **16**
3.1 Problemspecificering 16
3.2 Avgränsning 17
3.3 Förväntat resultat 17
4 Metod **18**
4.1 Analys av HTML 18
4.2 Övergripande metoder 21
4.2.1 Generell algoritm 22
4.2.2 Speciell algoritm för HTML 22
4.2.3 Generell kombinerad med generell 22
4.2.4 Specifik algoritm kombinerad med generell 22

4.2.5 Specifik algoritm kombinerad med specifik....23
4.3 Val av övergripande metod....23
4.4 Algoritmer....24
4.4.1 Förlustkomprimering....24
4.4.2 Run-length komprimering....24
4.4.3 Ziv-Lempel komprimering....24
4.4.4 Huffmankomprimering....25
4.4.5 Dynamisk Huffmankomprimering....28
4.4.6 Aritmetisk Komprimering....29
4.5 Val av algoritmer....31
5 Genomförande....33
5.1 Implementering av HTML-anpassad Huffman-algoritm....33
5.1.1 Generell Huffman-algoritm med balanserat träd....33
5.1.2 HTML-anpassad Huffman-algoritm....34
5.2 LZSS-algoritmen....39
5.3 Komprimeringseffektivitet hos algoritmen....39
5.4 Komprimering av taggar genom överskottsbitar från ASCII....40
5.5 Jämförelse med modem....43
5.6 Analys....45
6 Sammanfattning....48
6.1 Slutsats....48
6.1.1 Problembeskrivningen....48
6.1.2 Resultat....48
6.2 Diskussion....49
6.3 Fortsatt arbete....50
Referenser....52
Bilaga 1 – Http-adresser
Bilaga 2 – Huffmankod
Bilaga 3 – tabell.dat
Bilaga 4 – tabell2.dat
Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

Sammanfattning

Denna rapport beskriver varför transmission av data över telenätet är problematiskt och hur datakomprimering kan användas för att öka hastigheten på transmissionen. Datakomprimering används redan nu för att hålla nere storleken på den data som ska skickas, t ex är ofta bilder i JPEG-format eller något annat komprimerat bildformat. Rapporten inriktar sig på komprimering av HTML-dokument då dessa i nuläget inte komprimeras före transmissionen. Först beskrivs ett antal vanliga komprimeringsalgoritmer och sedan väljs en metod ut som verkar lämplig för komprimering av HTML-dokument. Algoritmen modifieras till att utföra bättre komprimering på HTML-dokument genom att använda en kombination av förlustkomprimering och ”överblivna” ASCII-koder. Algoritmen implementeras och testas med avseende på komprimeringseffektivitet och komprimeringshastighet.

All data komprimeras av modemet innan den skickas iväg, men hypotesen i denna rapport är att en implementerad algoritm som exekveras på processorn på den dator modemet sitter kopplat till komprimerar mera effektivt. Därför jämförs algoritmen med MNP-5 och V.42bis protokollen som används för datakomprimering i moderna modem. Då algoritmen tycks komprimera bättre än båda de protokoll som nämns ovan, bör den kunna användas som komplement till komprimeringen i modemen. Den kan dock inte helt ersätta komprimeringen i modemen då den enda komprimering som kan utföras av algoritmen är den av HTML-dokument.

1 Introduktion

Ända sedan människan först började kommunicera över långa avstånd har det funnits ett behov av att komprimera meddelandens storlek. När de gamla grekerna skickade meddelanden på papyrusrullar, gjorde den höga kostnaden på papyrus att de skrev utan mellanrum mellan orden för att tjäna in lite utrymme [Wil93]. De gamla romarna använde ofta förkortningar på gravstenar eftersom graveringen av dessa var mycket dyr [Wil93]. Förkortningar är en vanlig form av komprimering även idag. Som exempel kan nämnas armén, som är känd för sitt flitiga användande av komplicerade förkortningar. Många senare transmissionsmetoder bygger på att man skickar en bokstav i taget, t ex morse-kod. Sådana metoder är oerhört tidskrävande, vilket också Morse insåg, och han tilldelade därför bokstäver som förekommer ofta i det engelska språket korta koder medan ovanliga tecken fick långa koder. Senare har det också tillkommit olika specialkoder för att ange vanliga teckenkombinationer och liknande. I vår tid används ofta modem för att skicka data över telenätet. Modem arbetar också genom att sända ett tecken i taget, vilket gör att det är lämpligt att försöka hitta metoder för att ge de vanligaste tecknen så korta representationer som möjligt.

Idén om att komprimera data uppstod när man gick över från hålkort till text som lagrades på band eller disketter. De gamla hålkorten hade en fast radlängd på 80 tecken och därmed hade också de första texteditorerna en fast radlängd på 80 tecken. Detta gjorde att texten tog upp mycket mer dyrbar plats än vad som var nödvändigt och ganska snart började man använda editorer med dynamisk radlängd. Detta kan sägas vara den första enkla formen av komprimering i datorsammanhang.

Då komprimering innebär att överföringstider blir mindre och att mindre lagringsutrymme krävs för att lagra datan ligger det mycket pengar i att ta fram bättre komprimeringsmetoder. Därför har mycket forskning utförts inom området ända sedan början av femtiotalet. Länge rörde sig forskningen huvudsakligen om textkomprimering. Det är först på senare tid som bilder, ljud och rörliga bilder nått ett stort användningsområde och blivit intressanta att skapa kommersiella komprimeringsprogram för, och det är först när något blir kommersiellt intressant som företagen börjar lägga pengar i forskning om det.

Det första stora genombrottet gjordes 1952 av Huffman, som visade på ett optimalt sätt att byta ut datavärden mot koder av varierbar längd. Kodens längd svarar mot datavärdets sannolikhet att uppträda i datamängden. Då Huffman bevisade att hans metod var optimal kom forskningen av sig i några år eftersom ingen trodde att det gick att göra något bättre [Wil93]. Så småningom bevisades dock att Huffman hade fel i vissa antaganden och så var forskningen i full gång igen. Många olika varianter av Huffman-kodning har presenterats fram till idag, men ordet Huffman-kodning används som ett samlingsnamn för dem alla.

Senare forskning rör utveckling av anpassningsbara komprimeringsmetoder, dvs metoder som anpassar sig efter den data som ska komprimeras. Forskningen på detta område är uppdelad på två grenar, en gren som studerar hur en modell av datan lämpligast kan byggas upp och en gren som studerar hur modellerna ska användas för att lämpligast komprimera data.

Komprimering används framförallt vid hantering av grafik, t ex bilder, filmer, bildtelefoni och liknande. Det skulle vara omöjligt att använda t ex en bildtelefon på det vanliga telenätet utan att komprimera bilderna [Hal95]. Vid kommunikation över

1 Introduktion

telenätet används ett sk modem (se kap. 2.2.1) och av anledningar som beskrivs längre fram i denna rapport är det svårt att komma upp i höga hastigheter när digitala signaler ska skickas över detta medium. För att skicka datan använder man sig av ett modem. För att få upp överföringshastigheten använder sig de flesta modem av komprimering. En bra komprimeringsalgoritm i modemet kan spara avsevärda summor pengar för både företag och privatpersoner. Ett vanligt användningsområde för modem är att ”surfa” runt på Internet, och det finns därför ett behov av att komprimera innehållet på de sidor som finns på nätet. Då en sida innehåller bilder är dessa vanligtvis komprimerade med JPEG eller liknande. Eftersom komprimering av bilder är så vanlig finns många bra verktyg för att titta på dessa bilder utan att de behövs packas upp först. Moderna webbläsare har inbyggda verktyg för att packa upp och visa bilder komprimerade med JPEG. Vad gäller exempelvis text är komprimering av denna så ovanlig att det inte finns några bra verktyg för att hantera den i komprimerad form. Texten måste packas upp varje gång den ska editeras eller läsas och det finns inget stöd för textpackning i webbläsarna. Den enda komprimering som sker av text är den som modemet utför.

Avsikten med denna rapport är att undersöka huruvida den textkomprimering som utförs av modemet skulle kunna göras bättre, med avseende på komprimeringseffektivitet, med mjukvara som körs på processorn på den dator modemet är kopplat till. Komprimeringseffektiviteten är en jämförelse mellan storleken på den komprimerade och den okomprimerade datan. Arbetet kommer att koncentreras på komprimering av HTML-kod av den anledning att en mycket stor del av den text som skickas över modem är just HTML-kod. Rapporten kommer att beskriva några av de vanligaste komprimeringsmetoderna som förekommer idag och några algoritmer kommer att implementeras och testas. Testerna kommer att leda fram till en slutsats om vilken av metoderna som är bäst för HTML-komprimering och om det över huvud taget leder till någon förbättring jämfört med den komprimering som finns i modemen.

2 Bakgrund

2.1 Bakgrund till komprimering

Det finns flera skäl till varför data bör komprimeras. Det är till stor del en fråga om kostnader för dataöverföring, men också om bekvämlighet och tidsätgång. För många är telenätet det enda alternativ som finns för att komma ut på nätet. En nackdel med telenätet är att det är designat för ljudöverföring medelst analoga signaler medan datorer kommunicerar med digitala ettor och nollor. De digitala signalerna måste alltså omvandlas till analoga signaler innan de kan sändas. Pga telenätets utformning är det dock svårt att sända analoga signaler som representerar mer är 9600 digitala bitar per sekund. Denna hastighet är fullt tillräcklig för att tydligt uppfatta en människoröst, vilket ju var hela meningen med telenätet, men det är mycket långsamt när stora datamängder ska överföras. Som exempel kan nämnas en standard för bildtelefoner (H.320) som arbetar med bilder på 176 bildelement *144 linjer *12 bitar per bildelement * 10 bilder per sekund. Okomprimerad skulle denna data kräva 3.04 Mbps, vilket ligger långt över vad som är möjligt att sända över telenätet. För bildtelefonen är komprimering alltså en nödvändighet för att den över huvud taget ska kunna användas [Hal95].

Andra applikationer kräver inte komprimering, men av rena kostnads och bekvämlighetsskäl vill vi ändå komprimera datan. Ett exempel på detta är en vanlig privatperson som vill ta hem en sida från nätet. Personen måste då betala normal samtalstaxa för lokalsamtal under hela den tid det tar att föra över filerna som ingår i sidan. Ju snabbare det går att föra över filerna desto gladare kommer personen att bli. I längden kan det alltså löna sig för personen att betala för att få upp hastigheten på överföringen, t ex genom komprimering. Även om personen skulle vara mycket förmögen är det troligt att han/hon gärna skulle betala en hel del bara för att slippa vänta så länge. En undersökning av [GaTech] står följande:

> ”Speed continues to be the number one problem facing Web users, with 66.31% of the users reporting that it takes too long to download pages.”

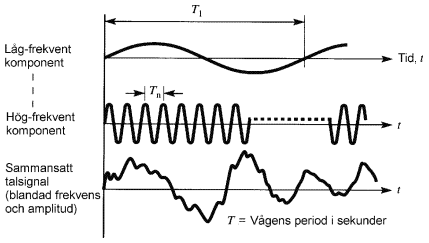
Det vill säga att det största problemet de flesta internetanvändare upplever är att överföringshastigheten är för långsam.

Datakomprimering har fler fördelar än att bara minska lagringsutrymme och överföringstider. Genom att datan består av ett mindre antal bitar och överföringstiden blir mindre vid datatransmission blir sannolikheten för att det uppstår fel i överföringen mindre.

Om man komprimerar många små filer är det vanligt att de slås samman till en stor fil. I och med detta behövs inte ’starta överföring’ och ’överföring slut’ sändas för varje fil vid filöverföring, och vid lagring går inget utrymme till spillo pga till stor del outnyttjade block på t ex en hårddisk. En stor fil istället för flera små kan i och för sig också ses som en nackdel eftersom det kan räcka med att en enda bit blir fel för att hela datamängden ska bli omöjlig att dekomprimera, och då förloras alla små filer samtidigt.

2.2 Datakommunikation över telenätet

Telenätet är, som nämnts tidigare, designat för röstkommunikation som antas ligga mellan 400 – 3400 Hz [Hal95]. Figur 1 visar ett exempel på hur en sådan analog signal kan se ut.



Figur 1: Exempel på analog signal för överföring på telenätet

Antag att man vill skicka en digital signal med en spänning för att representera en etta och en annan spänning för att representera en nolla. Om bitströmen enbart består av ettor eller enbart nollor kommer strömmen att representera en frekvens som är för låg för att kunna skickas över telenätet. För att undvika detta problem använder man sig av ett sk modem.

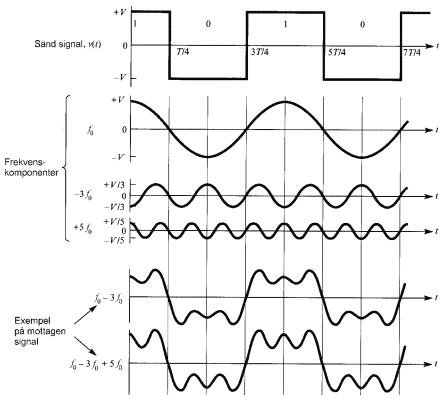
2.2.1 Modem

Ordet 'modem' är en förkortning för modulator/demodulator. Modem används för att omvandla digital data till analoga signaler som kan skickas över telenätet. Den traditionella definitionen av termen 'modulator' är "en apparat som tar emot en seriell ström av binära signaler från en datakälla och modulerar någon egenskap (såsom amplitud, frekvens eller fas) hos en analog signal för att skapa en signal som lämpar sig för överföring på ett analogt medium" [Bin88]. En demodulator definieras likartat som en komplementerande mottagningsenhet. Man brukar dock i termen modem även inkludera alla andra funktioner som krävs för dataöverföring över telenätet, såsom uppringningsenhet, felkorrigering, osv [Bin88].

2.2.2 Modulering

För att förstå hur moduleringen fungerar behövs lite bakgrundsinformation om bandvidd. Bandvidden hos ett transmissionsmedium specificerar vilket frekvensomfång av sinuskurvor som kan sändas genom mediet utan att försvagas. En periodisk signal, dvs en signal som upprepar sig själv med en visst tidsintervall (period), är uppbyggd av ett oändligt antal sinuskomponenter. Signalens period bestämmer den sk fundamentala frekvensen. Andra frekvenser som är multiplar av denna frekvens kallas harmonier (se figur 2). Det går att visa [Hal95] att alla periodiska binära sekvenser består av en oändlig serie sinussignaler inkluderande den fundamentala frekvensen, f_0, en tredje harmonisk komponent, $3f_0$, en femte harmonisk komponent, $5f_0$, och så vidare. Märk att för binära sekvenser används bara udda harmoniska komponenter.

2 Bakgrund

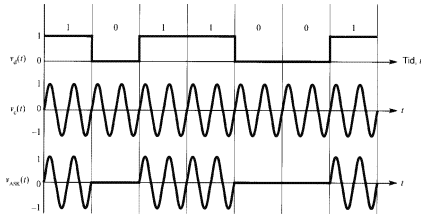


Figur 2: Hur harmoniska frekvenser bygger upp en binär signal

När man sänder data kommer bara de harmoniska komponenter som ligger inom bandvidden att användas [Hal95]. Som syns i bilden blir signalen tydligare ju fler harmonier vi använder. Det räcker med den fundamentala frekvensen för att skicka en signal, men oftast vill man använda även den tredje harmoniska komponenten för att få högre kvalitet på signalen. Problemet med detta är att ju fler harmoniska frekvenser vi använder desto mer bandvidd tas upp av signalen.

Vi ska nu titta lite närmare på tre vanliga moduleringsmetoder, amplitud-, frekvens-, och fasskiftning.

Amplitudskiftning går ut på att amplituden hos en enkelfrekvent ljudton ändras mellan två nivåer [Hal95] (se figur 3). Den enkelfrekventa tonen brukar kallas bärton eftersom den bär den digitala signalen genom kanalen.

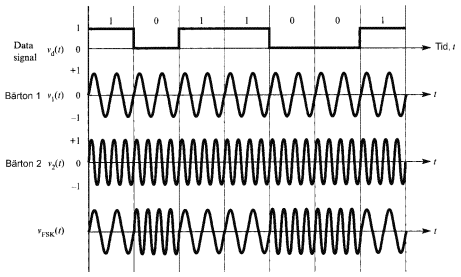


Figur 3: Amplitudskiftning

2 Bakgrund

Bärtonen väljs inom det frekvensband, bandvidden, som finns tillgängligt i telenätet. Ju högre frekvens bärtonen har desto fler bitar kan sändas per sekund. Så länge bitströmmen 101010… kan sändas utan problem kan alla andra bitströmmar sändas eftersom frekvensen för alla andra sekvenser är lägre [Hal95]. Den maximala överföringshastigheten med denna metod är ca 1000 bps med den tredje harmoniska komponenten eller 3000 bps med enbart den fundamentala frekvenskomponenten.

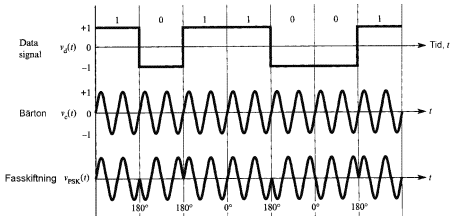
Frekvensskiftning var den första metoden som användes i modem [Hal95]. Metoden går ut på att det finns en frekvens som representerar en etta och en annan frekvens som representerar en nolla. Genom att växla mellan frekvenserna kan en bitström skickas (se figur 4).



Figur 4: Frekvensskiftning

För frekvensskiftning är det den mest högfrekventa bärtonen som anger hur stor bandvidd signalen upptar [Hal95]. Den maximala överföringshastigheten är samma som för amplitudskiftning

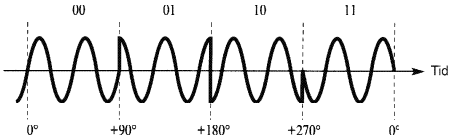
Fasskiftning går ut på att man förskjuter fasen med ett visst gradantal för varje bit som sänds [Hal95]. Ett exempel illustreras i figur 5. Fasskiftning använder sig av ungefär samma bandbredd som de bäda andra metoderna.



Figur 5: Fasskiftning

Som slutsats kan vi alltså komma fram till att genom att kombinera alla de ovanstående moduleringsmetoderna blir den maximala överföringshastigheten över telenätet ca 9600 bps om endast den fundamentala frekvenskomponenten används. En mera exakt

siffra är 9600 bps och är den siffra som brukar anges som en fysisk gräns för dataöverföring på telenätet. Nyare modem använder sig av mera komplicerade moduleringsmetoder som bygger på att fasen skiftas i flera steg och att det finns flera olika signalnivåer [Hal95]. Som framgår i figur 6 kan man på detta sätt representera fler än en bit med varje signal. Genom att skicka flera bitar per signal kan hastigheten då komma upp i ett antal multiplar av 9600 bps, t ex 28.8 kbps som varit den högsta hastigheten fram tills helt nyligen. En ny standard för modem är i skrivande stund under utveckling, och med denna nya standard ska överföringen komma upp i hastigheter på 56 kbps.



Figur 6: Fasskiftning i fyra steg gör att fyra olika kombinationer eller två bitar av information kan överföras med varje signal.

2.2.3 Komprimering i modem

De nya snabbaste modemen klarar alltså snart 56 kbps, men då modem är dyra kommer många använda sig av 28.8 kbps modem även i framtiden. Dessa hastigheter kan ses som mycket låga i jämförelse med storleken på de filer som skickas över nätet. Stora datafiler, textdokument, bilder, filmer, ljudfiler och allt annat kommer lätt upp i storlekar på flera megabytes. En megabyte tar över 140 sekunder att föra över med ett 56 kbps modem eller 280 sekunder med ett 28.8 kbps modem. För att öka överföringshastigheten använder moderna modem datakomprimering. Med en bra komprimeringsalgoritm kan kanske en fil komprimeras till hälften av sin ursprungliga storlek, och då kommer överföringen bara att ta hälften så lång tid. V.42bis är det mest effektiva protokollet för datakomprimering i modem och sägs ha en maximal komprimeringseffektivitet på 4:1. Det är på grund av detta som man kan se företag göra reklam för V.32 modem med en maximal överföringshastighet på 38.4 kbps trots att V.32 modem arbetar med 9600 bps [Diac].

2.3 Datarepresentation

Som nämndes i inledningen kan data komprimeras genom att den representation som används för olika datavärden ändras. För att förstå hur detta går till förklaras först vad som menas med datarepresentation och vi tittar på några olika modeller för att representera data.

All data måste tilldelas någon form av representation för att vi ska kunna tolka och förstå den. Denna representation kan bestå i t ex text, bilder eller diagram. Data som lagras i datorer måste tilldelas en binär representation då datorer arbetar enbart med ettor och nollor. Varje etta eller nolla i en dator kallas en ’bit’. Ett set av åtta bitar kallas en ’byte’. Ett godtyckligt antal bitar kan kombineras till att representera en mängd data. ’n’ bitar kan kombineras till att representera 2^n distinkta värden. Det finns en mängd scheman för att tilldela binära nummer till datavärden. Dessa scheman delas upp i tre generella kategorier.

- Textkodning
- Teckenkodning
- Binär kodning

Dessa tre scheman beskrivs närmare nedan tillsammans med exempel på hur de används.

2.3.1 Textkodning

Textkodning innebär att man tilldelar olika binärkombinationer en bokstav ur ett alfabet. Det finns standardtilldelningar för många av världens språk. En standard för alla världens språk är ISO 10646, som bygger på industristandarden Unicode. Den standard som används mest är dock ASCII (American Standard Code for Information Interchange) som används för det engelska alfabetet. ASCII använder 7 bitar för att representera 128 olika bokstäver, tecken och specialfunktioner. I tabell 1 visas de tecken som finns i ASCII-alfabetet och vilket nummer som associeras med de olika tecknen.

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

The C0 Set The G0 set

Tabell 1: ASCII-koder

De första två kolumnerna 'C0' är kontrolltecken, t ex CR – carriage return, ESC – escape, och HT – horisontal tab.

Exempel på applikationer som använder ASCII för att representera data är en uppsjö av texteditorer och även postscript som använder sig av text för att rita upp bilder.

Nackdelen med ASCII är att om man ska representera data som innehåller mycket siffror blir filerna mycket stora. T ex blir textsträngen '123' 3 bytes istället för 1 om den hade lagrats som ett heltal.

2.3.2 Teckenkodning

Teckenkodning använder en teckenmängd som ASCII, men istället för att tilldela bitsträngarna bokstäver tilldelas de olika datavärden. Genom att representera data med en specifik kod istället för med ett antal ASCII-tecken kan filstorleken reduceras avsevärt. Till skillnad från Textkodning finns ingen definierad standard för hur datan representeras, utan olika applikationer använder en egen representation efter vilken typ av data som ska representeras.

2.3.3 Binär kodning

Binär kodning är maskinberoende, dvs beroende på vilken processor man arbetar på kan representationen variera. Binär kodning innebär att datan kan kommas åt direkt, utan översättning eller manipulation, vilket ökar användarens åtkomsthastighet till datan. Nackdelen blir inkompatibilitet, dvs att det är svårt att flytta data från en plattform till en annan. Ett exempel på binär kodning är att operationer i PostScript-formatet kan ges koder mellan 0 och 255, vilket gör att t ex operationen moveto tar upp en byte istället för 6 bytes om den lagrats med textkodning.

Som framgår av ovanstående stycken finns många och skiftande sätt att representera data. Olika komprimeringsmetoder lämpar sig bäst för olika representationer, vilket gör det oerhört svårt, om inte helt omöjligt, att hitta en komprimeringsalgoritm som fungerar bra för alla sorters data. Detta faktum har medfört att det finns en uppsjö av olika komprimeringsmetoder, alla specialiserade på en viss typ av data. I kommande kapitel beskrivs några av de vanligaste metoderna.

2.4 Grundläggande komprimeringsteori

Många applikationer idag arbetar med mycket stora datamängder. Ta till exempel bildtelefonen som nämndes under rubrik 2.1. Denna videotelefon skulle behöva skicka 3.04 Mbps i varje riktning, vilket i och för sig är möjligt att genomföra, men som knappast skulle bli praktiskt användbart eftersom man då skulle behöva en egen 10 Mbps linje för varje enskilt samtal. För att kunna använda videotelefonen måste alltså datan på något sätt minskas rejält, helst så mycket att det skulle gå att sända över det vanliga telefonnätet. Detta kan göras på flera olika sätt, t ex genom att minska upplösningen, minska antalet bitar för att representera ett enskilt bildelement (t ex genom att minska antalet färger i bilden), eller minska bildfrekvensen. Alla dessa metoder gör att vi förlorar en hel del i bildkvalitet, och det vore istället önskvärt att hitta en metod att minska datamängden utan att bildkvaliteten påverkas allt för mycket. Ett sätt att göra detta är genom datakomprimering.

Datakomprimering reducerar antalet bitar som behövs för att representera en datamängd, dvs minskar mängden lagringsutrymme som åtgår för att lagra datan (ex hårddisk, magnetband) och minskar den tid som åtgår för att sända data. För att förstå hur datakomprimering fungerar definierar vi först ett antal viktiga begrepp tagna ur [BS95] om ej annat anges.

Komprimeringseffektivitet – Storleken på den komprimerade datan jämfört med den okomprimerade. Beräknas som

$$\frac{\textit{Antal Bytes i originalfil}}{\textit{Antal bytes i komprimerad fil}}$$

Komprimeringseffektivitet anges ofta som två tal åtskilda med ett kolon. Komprimeringen 10:1 innebär att 10 bitar i originaldatan har komprimerats till i bit i den komprimerade datan.

Förlustfri komprimering – Komprimering som inte resulterar i förlust av data. Denna typ av komprimering är nödvändig för text och annan data där varje tecken/datavärde är nödvändigt för förståelsen av datan.

Förlustkomprimering – Komprimering som resulterar i förlust av data. Denna metod ger oftast en högre komprimeringseffektivitet, men det är inte alltid acceptabelt att data går förlorad. Förlustkomprimering används främst vid komprimering av bild och ljud, där en viss dataförlust endast förorsakar en liten, acceptabel kvalitetsförlust.

Komprimerare – Process som tar in data och skickar ut komprimerad data.

Dekomprimerare – Process som tar komprimerad indata och skickar ut originaldata.

Realtidskomprimering – Process som komprimerar data tillräckligt snabbt för att tillgodose sin applikation.

Realtidsdekomprimering – Process som dekomprimerar data tillräckligt snabbt för att tillgodose sin applikation.

Entropi – Ett mätt på slumpmässighet. Helt slumpmässig data kan inte komprimeras. Enligt Howard och Witter kan data bara komprimeras när några datasymboler är mer troliga än andra [Sto92]. Ju högre ordningen är i datan desto mer kan den komprimeras. Komprimeringen sker genom att man tar bort redundant information, dvs tar bort ordningen. Den komprimerade filen kommer alltså att likna slumpmässig data. Entropin H(p) hos en datamängd kan beräknas som

$$H(p) = -\sum_{i=1}^{n} P_i \log_2 P_i$$

Där n är antalet datavärden i datamängden och P_i är sannolikheten för att bokstaven 'i' dyker upp i datamängden.

Modeller – En del komprimeringsalgoritmer använder modeller för att förutsäga sannolikheten för att nästa datavärde ska vara av en viss typ. Det finns detaljerade modeller för t ex det engelska språket som anger sannolikheten för varje bokstav i alfabetet, och även sannolikheten för en bokstav med avseende på ett antal tidigare bokstäver [BCW90].

Anpassningsbar modell – En sannolikhetsmodell som byggs upp allteftersom man går igenom en datamängd. Sannolikheterna kompenseras då efter varje inläst tecken [BCW90].

Vi ska nu titta på några av de komprimeringsmetoder som finns, till vilken data de passar bäst och hur de fungerar.

2.5 Metoder

Det finns många olika metoder för att komprimera data. Man brukar dela in dessa metoder i de tre övergripande metoderna transformering, reducerad precision, och minimering av antalet bitar [BS95].

Transformering gör om en datamängd till en annan ekvivalent datamängd som är mindre än originalet. En del transformationer reducerar antalet dataenheter i

datamängden, medan andra reducerar den numeriska storleken på datavärden (eng data items) så att de kan representeras med ett mindre antal bitar.

Reducerad precision minskar precisionen på enskilda datavärden (eng data values) i en mängd, vilket gör att det krävs ett mindre antal bitar för att representera varje värde. Reducerad precision kan också minska slumpmässigheten i en dataström. Reducerad precision kan endast användas till att komprimera datafiler där noggrannheten på enskilda datavärden inte spelar så stor roll, t ex alla datavärden representeras med heltal istället för med flyttal. Då metoden inte lämpar sig för komprimering av text kommer inga algoritmer som utnyttjar metoden att tas upp.

Minimering av antalet bitar innebär att varje datavärde får en representation som minimerar det totala antalet bitar som behövs för att representera hela dataströmen.

2.5.1 Packed decimal

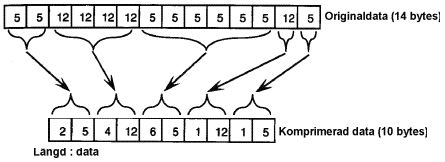
Denna metod används när man vill komprimera ASCII-filer som enbart innehåller siffror, t ex logg-filer från någon sorts datainsamling. Istället för att representera siffrorna 0-9 med ASCII-tecken kan de representeras med BCD (Binary Coded Decimal). BCD använder bara fyra bitar per tecken istället för ASCIIs sju. Metoden kan placeras under gruppen transformering eftersom den går ut på att man byter ut hela representationsformen mot en annan. Då denna komprimeringsmetod är helt värdelös till alla filer utom rena sifferfiler i ASCII-kod, kommer den inte att betraktas närmare.

2.5.2 Relativ komprimering

Istället för att lagra alla datavärden för sig, lagrar man endast skillnaden mellan föregående och nuvarande datavärden. Denna typ av komprimering lämpar sig för värdetabeller med långsam variation mellan olika värden, t ex en logg-fil för vattenståndet i en sjö. Då värdeförändringen troligen är mycket mindre än själva värdet kan ett mindre antal bitar användas till att representera datan. Då metoden går ut på att välja ett storleksmässigt mindre värde än urspungsvärdet kan den placeras under metoden transformering.

2.5.3 Run-length komprimering

Run-length komprimering är en mycket enkel och vanligt förekommande form av komprimering. Den går ut på att flera identiska datavärden på rad byts ut mot ett datavärde och en siffra som anger det ursprungliga antalet värden (se figur 7) och hamnar därmed under metoden transformation. Run-length komprimering är mycket effektiv när man vill komprimera data med många identiska värden på rad.



Figur 7: Exempel på run-length komprimering

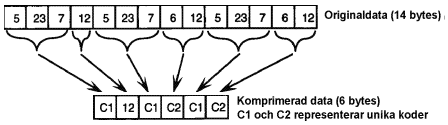
Lägg märke till att de två sista värdena i figuren leder till fyra tecken i den komprimerade datan. Filer med helt slumpmässiga värden kommer i och med detta att bli större efter komprimering än de var innan, och filer med vissa delar slumpmässiga tal kommer inte att komprimeras optimalt. Detta kan man komma runt genom att inte komprimera 'ensamma' värden. Man måste då sätta någon form av identifierare framför värdena för att tala om ifall de är komprimerade eller inte.

Komprimeringseffektiviteten hos run-length komprimering varierar beroende på strukturen hos den data som ska komprimeras. En del data kan reduceras avsevärt medan annan data kan förstoras av komprimeringen. Den typiska komprimeringseffektiviteten i de applikationer där run-length komprimering används ligger från 2:1 upp till 5:1 [BS95].

Exempel på applikationer som använder run-length komprimering är faxar, modem, och vissa grafiska representationer (TARGA, TIFF, PCX, PostScript) [BS95].

2.5.4 Ziv-Lempel komprimering

Ziv-Lempel komprimering (även kallad LZ-komprimering) är en familj av likartade komprimeringsmetoder som bygger på att sekvenser av datavärden som förekommer flera gånger i datamängden byts ut mot referenser till den första förekomsten av sekvensen. En unik kod tilldelas sekvensen och sedan lagras koden i den komprimerade datan. Denna metod hör således till transformeringsmetoderna. Figur 8 visar ett exempel på hur det kan gå till.



Figur 8: Exempel på kodtilldelning vid en typ av LZ-komprimering

Som framgår i figuren är det inte alla sekvenser som blir tilldelade en kod. Sekvensen (12, 5) repeteras två gånger men tilldelas ingen kod då den överlappas av andra sekvenser som redan tilldelats koder. Sekvensen (5, 23, 7, 6, 12) repeteras också två gånger men blir inte heller tilldelad någon kod. Om denna sekvens blivit tilldelad en kod hade komprimeringen varit mera effektiv, vilket visar på problemet med att välja ut vilka sekvenser som är mest lämpliga att tilldela koder. Det existerar en bästa gruppering för varje datamängd, men att hitta denna gruppering innebär att man måste gå igenom varje möjlig kombination i hela datamängden, vilket är en oerhört tidskrävande process. Man går istället igenom datamängden byte för byte och tilldelar koder till de första sekvenser som upprepar sig. Det är ändå inte säkert att den längsta sekvensen ger den bästa komprimeringen, då en kortare sekvens kanske förekommer fler gånger.

Koderna och de associerade sekvenserna lagras i en 'kodbok' som måste lagras tillsammans med den komprimerade datan, annars vet inte dekomprimeraren vad koderna står för. Om kodboken blir stor minskas effektiviteten hos komprimeringen. Ett komprimeringsschema framtaget av Lempel, Ziv och Welch (LZW-komprimering), gör att man inte behöver komma ihåg kodboken. LZW-komprimering bygger på att kodboken byggs upp allteftersom datan gås igenom. En sekvens tilldelas en kod först

efter att den förekommit två gånger och första gången den lagras så lagras den okomprimerad. När sedan dekomprimeraren läser in filen ser den att sekvensen är okomprimerad och lägger till den i sin kodbok. Förutsatt att komprimeraren och dekomprimeraren bygger upp sina kodböcker likadant kan dekomprimeraren känna igen koden för sekvensen nästa gång den dyker upp.

LZ-komprimering fungerar bra för många olika datatyper och används av bla PostScript, Unix *compress*-kommando, och grafiktyperna GIF och TIFF. Komprimeringseffektiviteten är databeroende och ligger vanligtvis mellan 2:1 till 9:1 [BS95].

2.5.5 Huffmankomprimering

Huffmankomprimering innebär att man byter ut representationer av datavärden mot andra representationer. Genom att välja korta representationer för vanligt förekommande datavärden och längre representationer för ovanliga datavärden kan komprimering uppnås. Detta förutsätter att vissa datavärden förekommer oftare än andra, annars blir resultatet dåligt och kan i värsta fall resultera i att datamängden blir större efter komprimeringen. Huffmankomprimering hör till metoden minimering av antalet bitar.

För att kunna utföra Huffmankomprimering måste det finnas en modell över vilka datavärden som är mest förekommande. Detta kan göras på tre sätt.

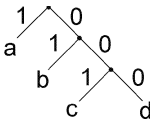
Det lättaste är att det redan finns en fördefinierad modell. Sådana modeller finns väl definierade över t ex engelska språket, där man kan få tillgång till information över sannolikheten för varje enskild bokstav och även för varje bokstav med avseende på ett antal tidigare bokstäver. Nackdelen med denna metod är att de fördefinierade modellerna inte alltid är optimerade för den data man vill komprimera. En modell över det engelska språket är, oavsett hur väl definierad den är för just den tillämpningen, helt värdelös om man skulle vilja komprimera t ex en datafil med enbart siffror. En variant på denna metod är att ha ett större antal fördefinierade modeller och komprimera med den som ger bäst resultat. Då behövs bara informationen om vilken modell det är som använts lagras med den komprimerade filen, men det blir ändå ingen optimal komprimering för alla typer av filer om man inte har oändligt många modeller att välja på.

Ett annat sätt är att innan själva komprimeringen utförs så går man igenom hela datamängden och räknar hur många gånger varje datavärde förekommer. På detta sätt kommer man alltid att utföra komprimeringen med en optimal modell. Dock finns två nackdelar. Det tar mycket längre tid att utföra komprimeringen eftersom man måste gå igenom datamängden två gånger, och dessutom har mottagaren ingen aning om hur modellen ser ut så man måste lagra modellen också, vilket tar upp en massa onödigt lagringsutrymme eller bandbredd.

Ett tredje sätt är att bygga upp modellen dynamiskt, vilket beskrivs under nästa rubrik.

Modellerna är uppbyggda i form av ett sk Huffmanträd (eng Huffman code tree). Det är ett binärt träd med grenar som tilldelas värde 0 eller 1. Trädbasen kallas rotnod (eng root node) och noder där trädet förgrenar sig kallas förgreningsnod (eng branch node). Ändnoderna kallas lövnoder (eng leaf node). Varje gren delar upp sig i en 1-gren (höger) och en 0-gren (vänster). Man får fram hela kodordet genom att följa en stig från rotnoden till det datavärde man vill ha kodordet för. Kodordet byggs då upp av de 1:or och 0:or man passerar på vägen. Ju ovanligare bokstaven är, desto längre ner i

trädet ligger den. Figur 9 visar ett exempel på ett Huffmanträd. I trädet kommer ’a’ att tilldelas koden ’1’, ’b’ tilldelas ’01’, och så vidare.



Figur 9: Huffmanträd

Om man har ett stort alfabet med ungefär lika stor förekomst av alla datavärden är Huffman en mycket dålig metod. Den är bäst när några få tecken återkommer ofta. Den används främst vid komprimering av text på naturligt språk, t ex för lexikon och uppslagsverk.

2.5.6 Dynamisk Huffmankomprimering

Istället för att ha en fast modell kan både sändaren och mottagaren bygga upp ett Huffmanträd under tiden som komprimeringen/dekomprimeringen pågår. Detta kallas dynamisk Huffmankomprimering. Om ett datavärde redan finns i trädet komprimeras det som vanligt, men om datavärdet förekommer för första gången lagras det okomprimerat. Komprimeraren/dekomprimeraren uppdaterar sedan sitt Huffmanträd genom att antingen uppdatera frekvensen för datavärdet eller sätta in ett nytt datavärde i trädet.

Dynamisk Huffmankomprimering har fördelen att modellen den jobbar med kan lagras med den komprimerade datan med minimal förlust av komprimeringseffektivitet. En liten nackdel är att det tar tid att uppdatera Huffmanträdet hela tiden. Metoden används i modem av standarden V.32, vilket är en standard för modem som arbetar med överföringshastigheter på 9600 bps [Hal95].

2.5.7 Aritmetisk komprimering

Aritmetisk komprimering använder sig också av en dynamisk modell. Tekniken bygger på att istället för att tilldela ett datavärde en kod som ligger så nära logaritmen av sannolikheten som möjligt så tilldelas det en kod som är exakt lika lång. Dvs att om sannolikheten för x är 0,25 ska x representeras med en 2 bitars kod ($-\log_2 0{,}25$ bitar). Likaså ska en symbol med sannolikheten 273/374 ges en kod som ligger så nära $-\log_2 273/374$ bitar som möjligt [Wil93]. Detta kan uppnås genom att inte se varje enskilt tecken som ett värde som ska tilldelas en kod som i Huffmankomprimering, utan snarare som att en kod ska tilldelas hela datamängden. Om ett tecken tilldelas en kod på 1.5 bitar och ett annat tecken får en kod på 0.5 bitar kan de två bräkdelarna slås samman så att de båda tecknen tillsammans tilldelas en kod på 2 bitar.

3 Problemdefinition

Många privatpersoner som hämtar hem sidor från internet använder sig av ett modem för att ta emot och skicka filer via telenätet. Problemet med detta är att vissa fysiska begränsningar gör att data inte kan skickas särskilt snabbt över telenätet (se kap. 2.2.2). En undersökning har visat att de flesta internetanvändarna anser att just den långsamma överföringshastigheten är det största problemet med internet idag [GaTech]. Problemet blir värre av att ju längre tid det tar att skicka eller ta emot en fil desto dyrare blir det i och med att man betalar samtalstaxa under hela överföringstiden. Det finns alltså ett behov av att utföra en så effektiv komprimering som möjligt av all data innan den skickas iväg.

3.1 Problemspecificering

Texten, både dokumenttexten och HTML-koden, skickas i allmänhet i ASCII-format. Ett skäl till detta skulle kanske kunna vara att idén med att bygga nätverk av datorer började i USA och att ASCII är den vanligaste standarden för ren text där. Det kan därmed verka logiskt att bygga en textkomprimerare på statistik över det engelska språket. Dock liknar HTML-kod knappast vanligt engelskt talspråk och den övriga texten i dokumentet kan vara på vilket språk som helst (även om engelska är vanligast på nätet). Existerande metoder för komprimering av text är därför inte optimala.

HTML-kod är väl strukturerad och upprepar ofta samma sekvens om och om igen. Dessa sekvenser kallas ”taggar” och de talar om för webbläsaren var textstycken börjar och slutar, vilket teckensnitt som ska användas, när det kommer en bild osv. Då taggarna är lätta att känna igen bör en modell kunna byggas så att den utnyttjar det faktum att taggar förekommer ofta i HTML-text.

Nya modem innehåller algoritmer för att komprimera data, men jag anser att en implementerad algoritm kan få en bättre överblick över datan och fatta bättre beslut om hur den ska komprimeras. Detta baserar jag på att en implementerad algoritm har större tillgång till både minne och processorkraft än vad som finns i modemet. Om det skulle komma fram nya och bättre komprimeringsalgoritmer vore det dessutom enklare att byta ut ett komprimeringsprogram än att byta ut hela komprimeringsenheten i modemet.

Problemet som studeras i denna rapport är om överföring av www-dokument med modem går snabbare (färre antal bytes skickas) om datan komprimeras av en implementerad algoritm innan den skickas till modemet än om modemet självt komprimerar datan. Olika sidor på nätet som kan anses representera typisk www-data kommer att studeras, t ex sidor med mycket text, mycket länkar, mycket bilder, och en kombination av alla dessa. Statistik över hur vanligt förekommande olika bokstäver är på de olika sidorna kommer sedan att sammanställas och skillnaden med och utan taggar kommer att jämföras. Detta för att visa att taggarna gör att texten i HTML-dokumenten skiljer sig avsevärt från vanlig text. En metod för att komprimera HTML-dokument kommer att väljas och sidorna kommer att komprimeras med denna metod. En jämförelse kommer att göras med de algoritmer som sitter i modem för att visa att den implementerade algoritmen är bättre. Två jämförelser måste göras, dels avseende komprimeringseffektivitet, dvs vilken algoritm som ger minst utdata, och dels avseende komprimeringshastighet eftersom algoritmen måste generera utdata lika snabbt som modemet klarar att skicka datan.

Sammanfattningsvis består problemet av två punkter:

- Ta fram en algoritm för komprimering av HTML-dokument.
- Jämför den implementerade algoritmen med den algoritm som finns i modem. Jämförelsen ska ske med avseende på både komprimeringseffektivitet och komprimeringshastighet.

3.2 Avgränsning

Www-dokument består i huvudsak av två saker, bilder och text. Bilderna är i det närmaste alltid redan komprimerade med metodena GIF eller JPEG, vilket gör att komprimering av dessa bilder inte ger någon fördel. Det går dessutom att stänga av bilder i webbläsaren så att dessa inte skickas över huvud taget, och om man väljer att se bilderna så lagras de lokalt i en cache så att de inte behövs laddas hem mer än en gång. Texten är däremot okomprimerad, och det är just komprimering av text som denna rapport inriktar sig mot. Texten i ett www-dokument består dels av den text som är synlig i dokumentet och dels av den HTML-kod som talar om för webbläsaren hur sidan ske se ut, t ex fontstorlekar, placering av text och bilder och så vidare. Komprimering av bilder kommer inte att tas upp.

Ett färdigt program måste kunna komprimera data i realtid om det ska kunna innebära någon tidsvinst att använda det. Programmet måste också arbeta helt transparent, dvs användaren ska inte behöva tala om att datan ska komprimeras utan det ska gå automatiskt. Om algoritmen fungerar i realtid kan grovt kontrolleras genom att mäta tiden som åtgår för att komprimera en fil och sedan jämföra denna med storleken på den komprimerade filen. Att göra en fullständig implementation innebär att man måste hålla reda på hur olika webb-läsare hanterar data, hur det fungerar när data skickas mellan olika plattformar, hur programmet kommunicerar med modemet, osv. Dock kommer inte någon implementation av själva programmet göras.

3.3 Förväntat resultat

Resultatet av denna rapport kommer att bli en utvärdering av om den implementerade algoritmen är bättre vad avser komprimeringseffektivitet och komprimeringshastighet än den algoritm som används av modemet. Rapporten kommer också att visa på att HTML-kod kan komprimeras mera effektivt om hänsyn tas till taggar än om en vanlig generell algoritm används. Ett möjligt fortsättningsarbete på denna rapport skulle kunna vara att utveckla algoritmen till en färdig programvara.

4 Metod

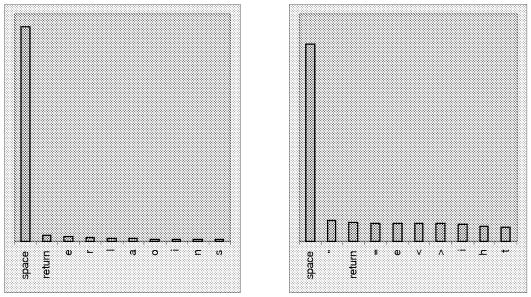
4.1 Analys av HTML

HTML-dokument är textfiler på ASCII-kod som innehåller dels den text som ska synas i dokumentet, och dels olika formateringskommandon för hur texten ska se ut, vart bilder ska vara placerade, osv. Dessa formateringskommandon kallas taggar, och de känns igen på att de börjar med ’<’ och slutar med ’>’. En typisk länk i ett dokument kan se ut som följer:

<A HREF = ”http://www.a.b”>Beskrivande text till länken</A>

Där ’<A’ innebär att det är en tagg som beskriver en hyperlänk, dvs en text som refererar till en annan text som man kan komma till genom att klicka på länken. ’HREF=”xxx”’ är den data länken refererar till, vilket kan vara en plats i dokumentet, ett helt annat dokument, en bild, etc. ’>’ visar att taggen är slut och att eventuell vidare text är vanlig text som ska skrivas ut på skärmen (om det inte är en ny tagg förstås). Efter den beskrivande texten kommer taggen ’</A>’ som talar om att länktexten är slut och att eventuell vidare text ska skrivas ut som normaltext.

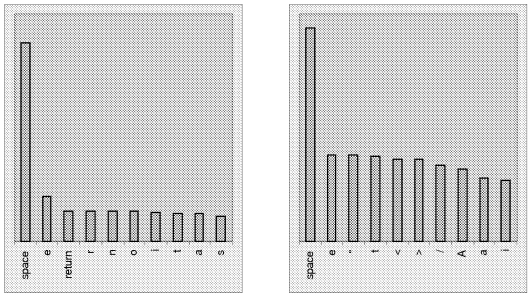
Komprimering skulle kunna uppnås genom att utnyttja den upprepning som förekommer i HTML-dokument. De flesta sidor som finns på nätet innehåller en hel del taggar. Undantaget är sidor som innehåller stora textmassor utan några bilder eller länkar. För att undersöka hur stor del av den text som skickas över nätet som är taggar gjordes en sammanställning över ett antal slumpvis utvalda sidor. Sidorna är uppdelade i grupperna ’bilder’, ’länkar’, ’text’, och ’blandat’. ’Bilder’ är sidor med enbart bilder med korta beskrivande texter. I vissa fall var också varje bild en länk. ’Länkar’ är sidor med enbart länkar. ’Text’ är sidor med mycket text och ’blandat’ är sidor som innehåller lite av varje, dvs lite text, lite bilder och lite länkar. Låt oss börja med att titta på hur en sida med mycket bilder ser ut. Figur 10a visar statistiken för vilka tecken som är vanligast i texten på sidan och förhållandet mellan dem. Att andelen mellanslag är så hög beror på att bilderna och texten till dem inte ligger vid vänstermarginalen. Området till vänster om bilderna och texten fylls då ut med mellanslag. Om man bortser från detta är diagrammet ganska representativt för vanlig text, de bokstäver som är vanligast ligger högt även i övrig statistik över engelska språket [Sal97]. Figur 10b visar vad som händer om vi räknar med taggarna i texten. Plötsligt ligger tecknen ”, =, <, och > högt upp på listan. Även h och t ligger högt upp på grund av att många av bilderna också var länkar, vilka innehåller ordet ’http’. Totalt innehåller sidorna med bilder hela 72% taggar. Detta är dock en missvisande siffra då t ex namnet på den bild som ska visas och även adressen som en länk pekar på inte är specifik HTML-kod och det är inte säkert att mer än kanske en liten del av dem upprepas över huvud taget. Siffran är dock så hög att det ändå kan inses att en stor del av texten också är HTML-kod som upprepas flera gånger.



Figur 10: a) Bilder, synlig text b) Bilder, taggar medräknade

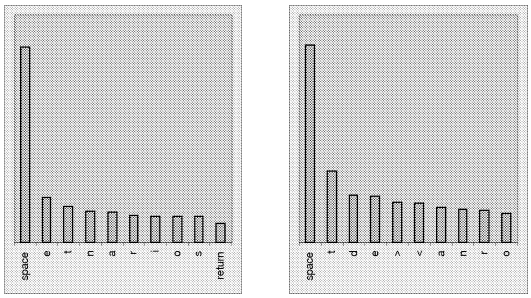
Om vi nu övergår till att titta på sidorna med länkar ser vi endast små skillnader i grafernas utseende jämfört med graferna för bild-sidorna. Figur 11a visar statistiken över bokstäverna i den synliga texten och som vi kan se är den ganska lik figur 10a vad avser vilka bokstäver som är vanligast. Likaså kan vi se ungefär samma förändringar mellan 11a och 11b som mellan 10a och 10b. Andelen mellanslag är inte lika stor då sidorna innehåller text som ligger utmed vänstermarginalen, utan ser mera ut som andelen mellanslag i normal text. Den totala mängden taggar i dokumenten är 66%, vilket alltså inte är riktigt lika mycket som för bild-sidorna men som ändå är mycket om det skulle gå att komprimera effektivt. Även denna siffra bör tas med en nypa salt då alla länk-taggarna innehåller en adress som inte går att komprimera med en speciell HTML-komprimeringsalgoritm.

För att visa på att siffrorna är lite missvisande valdes en av sidorna (http://albie.wcupa.edu/schmus.mue/515/menu6.htm), och alla webbadresser togs bort. Tecknen '<A HREF="' och '' som då blir kvar utgör 14% av hela texten, vilket innebär att dessa två teckenkombinationer är nästan lika vanligt förekommande i denna text som de två vanligaste bokstäverna, 'e' och 't', är i vanlig engelsk text [BCW90]. Hela dokumentet består till drygt 21% av taggar exklusive webbadresserna, vilket visar att siffran 66% taggar som nämns ovan är missvisande. Dock är siffran så hög att det ändå finns en hel del att tjäna på att komprimera taggarna.



Figur 11: a) Länkar, synlig textb) Länkar, taggar medräknade

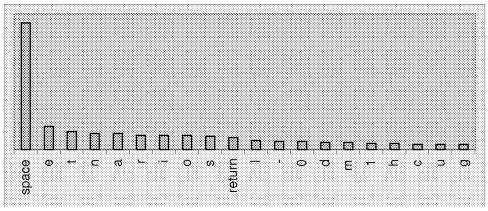
Om vi nu tittar på statistiken för sidor med större textmängder ser vi fortfarande ungefär samma tendenser. Diagrammen finns i figur 12. Det är ungefär samma bokstäver som är vanligast som i de andra fallen och om vi räknar med taggarna kommer tecknen < och > med högt upp i statistiken. Detta är lite olyckligt och motsäger att HTML-dokument med mycket taggar skulle skilja sig från dokument med mycket text. Det visar sig att det beror på att ett av HTML-dokumenten som ingår i textfilen (http://www.his.se/neverlost/schema2.idc) innehåller text som är formaterad i kolumner. Denna formatering görs med taggarna <td> och </td>. Som vi kan se i diagrammen är det just bokstäverna 't' och 'd' som ligger i topp tillsammans med '<' och '>'. Om sidan med den formaterade texten tas bort från dokumentet kommer diagrammet i figur 12b att se ut ungefär som det i figur 12a. Som bevis på detta kan vi se att tecknet ' " ' inte finns med i diagrammet, och att det alltså inte är samma taggar i textfilen som i de övriga filerna.



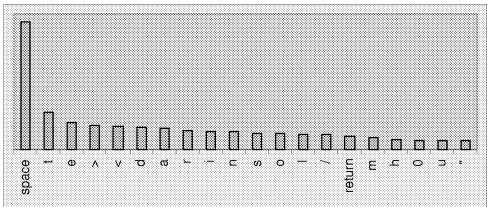
Figur 12: a) Text, synlig text b) Text, taggar medräknade

Det är svårt att dra några slutsatser i och med att den formaterade texten gör att textfilen inte skiljer sig speciellt mycket från de övriga filerna. Om den formaterade texten räknas med kan vi dra slutsatsen att det inte spelar någon roll vilka typer av sidor som överförs, det är ändå ungefär samma tecken som är vanligast. Om den däremot inte räknas med kan slutsatsen dras att det är skillnad mellan HTML-dokument med många länkar och bilder och sådana med mycket text. Det skulle dock kunna ses som ”fusk” att ta bort en sida bara för att den inte ger önskat resultat.

Som en sorts bekräftelse på att texterna faktiskt skiljer sig åt kan vi titta på diagrammen i figur 13 och 14, som är en sammanställning av alla sidorna i de andra diagrammen samt de sidor som står som ’blandat’. Dessa diagram ser till stor del ut som de ovanstående och för att tydligare visa på vilka tecken som finns med bland de vanligaste har de 20 vanligaste tagits med. I ser dock att tecknet ’ ” ’ ligger långt ner på listan trots att det varit med bland de tre vanligaste i både länkar- och bilderfilerna. Detta kan ses som ett tecken på att det faktiskt är skillnad i utseendet på HTML-dokument och textdokument.



Figur 13: Alla sidor, synlig text



Figur 14: Alla sidor, taggar medräknade

4.2 Övergripande metoder

Vi ska nu titta lite närmare på hur en komprimeringsalgoritm borde vara uppbyggd för att ge bra komprimering av HTML-dokument på någorlunda kort tid. Det första problemet som uppkommer är om komprimeringen ska ske i ett eller två pass, dvs

räcker det med att använda en algoritm eller lönar det sig att köra två algoritmer i följd. Oavsett vilken metod som väljs blir nästa problem att välja vilken/vilka algoritm(er) som ska användas. Det finns fem huvudtyper av övergripande metoder som är intressanta att titta närmare på:

- Endast generell algoritm
- Endast speciell algoritm för HTML
- Generell algoritm + generell algoritm
- Speciell algoritm för HTML + generell algoritm
- Speciell algoritm för HTML + speciell algoritm för HTML

De generella algoritmer som ska övervägas är: run-length komprimering, Ziv-Lempel komprimering, Huffmankomprimering, Dynamisk Huffmankomprimering, samt Aritmetisk komprimering. De speciella algoritmerna är varianter av de generella som ändras lite för att passa just HTML-dokument.

4.2.1 Generell algoritm

Med en generell algoritm avses helt enkelt någon av de algoritmer som nämns ovan. Fördelen med denna metod är att en generell metod inte kräver så mycket arbete, utan det går att ta en färdig implementation från nätet eller någon bok. En annan fördel är att själva komprimeringen går ganska snabbt eftersom den endast sker i ett steg. Den stora nackdelen är dock att en ensam algoritm sällan komprimerar lika effektivt som två på varandra följande algoritmer.

4.2.2 Speciell algoritm för HTML

Med en specifik HTML-algoritm menas en algoritm som designats för att hitta mönster i HTML-dokument. I övrigt ser den speciella algoritmen ut ungefär som den generella. Fördelen med denna metod är att den komprimerar HTML-dokument mera effektivt än en generell algoritm. Nackdelen är att metoden, liksom metoden i 4.2.1, troligen inte klarar att komprimera data lika effektivt som två på varandra följande algoritmer.

4.2.3 Generell kombinerad med generell

Med denna metod avses alltså att två generella algoritmer följer på varandra. Anledningen till att göra så är att det blir lättare att uppnå effektiv komprimering med två algoritmer. Det visas inte minst av att nästan alla kommersiella komprimeringsprogram, och även MNP-5 protokollet för modem, använder sig av två olika på varandra följande algoritmer. Nackdelen blir att komprimeringen tar längre tid att genomföra.

4.2.4 Specifik algoritm kombinerad med generell

Metoden med en specifik algoritm följd av en generell kommer att bli ganska lik metoden i 4.2.3, men med ett par skillnader. En extra fördel blir att den specifika algoritmen komprimerar HTML-koder lite mera effektivt än en generell algoritm. Nackdelen blir att en specifik algoritm kräver en hel del implementering vilket tar mycket tid i anspråk.

4.2.5 Specifik algoritm kombinerad med specifik

Denna metod innebär att två algoritmer specialanpassas till att fungera bra för HTML-dokument. Nyttan med detta kan dock vara tvivelaktig. Efter det att den första algoritmen komprimerat dokumentet finns det inget i den komprimerade datan som kan visa att det varit just ett HTML-dokument. Det vore därför ingen idé att bygga den andra algoritmen för att passa HTML-dokument utan snarare för att passa utdatan från den första algoritmen. Det skulle ta mycket tid att implementera den andra algoritmen och det vore dessutom tveksamt om den skulle jobba bättre än någon generell algoritm i alla fall.

4.3 Val av övergripande metod

Vi ska nu välja ut den metod i kapitel 4.2 som verkar vara bäst för problemet, dvs för komprimering av HTML-dokument.

Att bara använda en generell algoritm ger i allmänhet inte lika hög komprimeringseffektivitet som att använda två olika på varandra följande algoritmer [Gus98]. Detta är ett välkänt faktum som kan testas genom att helt enkelt köra två algoritmer på datan och se på den slutliga storleken. Det krävs naturligtvis att algoritmerna inte använder sig av samma komprimeringsmetod. Att köra en LZW-algoritm efter en LZSS-algoritm ger ingen förbättring. Alltså kommer en metod med två algoritmer att väljas. Risken att två algoritmer kräver för mycket processorkraft borde inte vara alltför överhängande, det gäller bara att välja ”rätt” algoritmer. Det är dock viktigt att inte så okomplicerade och snabba algoritmer väljs att komprimeringseffektiviteten blir alltför låg. Detta är en avvägning som får göras.

Att välja två algoritmer som är speciellt avsedda för HTML-kod är den metod som kanske låter som om den vore bäst. Vid närmare eftertanke kan man dock diskutera nyttan med den andra algoritmen. Efter det att den första algoritmen utfört den första delkomprimeringen är filen inte längre särskilt speciell i det avseende att man kan se att det varit en HTML-fil. Därför borde det gå lika bra med vilken generell algoritm som helst i det andra steget så länge den komprimerar vanlig data med bra resultat. Även om en speciell algoritm skulle ge en smula högre komprimering vore det knappast lönt det extra arbete som krävs att implementera den speciella algoritmen.

Kvar återstår då att välja mellan två generella algoritmer eller en speciell HTML-algoritm följd av en generell algoritm. Det är svårt att säga vilken av dessa metoder som är bäst utan att implementera båda och testa dem mot varandra. Det är mycket troligt att den specialanpassade algoritmen skulle vara bättre på HTML-dokument, och dessutom finns det redan ett flertal program på marknaden som använder sig av två generella algoritmer så att implementera en till vore knappast värt besväret. Alltså kommer den metod väljas som först kör en speciell HTML-anpassad algoritm och sedan en generell algoritm.

Slutligen måste vi då välja vilka algoritmer som ska användas i första respektive andra steget. Vi ska nu titta lite närmare på de algoritmer som omnämns i kapitel 4.2, hur de fungerar, till vilken typ av data de lämpar sig, hur snabba de är, osv.

4.4 Algoritmer

4.4.1 Förlustkomprimering

Vid komprimering av text gäller att komprimeringen måste vara förlustfri, dvs att texten måste kunna dekomprimeras till att se exakt likadan ut som originaltexten. Html-kod ska inte läsas av människor så för denna text gäller att den måste följa de regler som finns i webbläsaren. Dessa regler är lite flytande så samma HTML-kommando kan se ut på olika sätt beroende på vem som skrivit den. Dessa variationer skulle kunna rättas till av komprimeraren så att samma kommando alltid ser likadant ut. Det spelar till exempel ingen roll om koderna står med stora eller små bokstäver. Genom att alltid konvertera HTML-koderna till stora bokstäver kan antalet kombinationer minskas och komprimeringen göras mera effektiv. Det går också att ta bort onödiga mellanslag. Som exempel kan vi titta på följande text:

<A HREF = ”http://www.a.b”>Beskrivande text till länken</A>

I exemplet står …HREF•=•”ht… (mellanslag kommer i fortsättningen alltid att skrivas som • när de behöver framträda tydligt), men det kunde lika gärna stå …HREF=”ht… I många fall används också onödiga citationstecken. Det går precis lika bra att skriva <FONT SIZE=1> istället för <FONT SIZE=”1”>. Detta gäller dock inte alltid, så om komprimeraren ska kunna ta bort onödiga citationstecken, och även vid borttag av onödiga mellanslag, krävs att algoritmen har kunskap om alla olika typer av HTML-kommandon och i vilka fall citationstecken och mellanslag måste användas eller inte.

4.4.2 Run-length komprimering

Implementationen av run-length komprimering är enkel och komprimeringen går mycket snabbt. Det behövs inga komplicerade beräkningar, utan bara en kontroll om nästa tecken är samma som föregående tecken eller inte. Problemet är att komprimeringseffektiviteten inte är speciellt hög om inte datan är enformig i den mening att samma tecken upprepas många gånger i följd. Detta kommer av att run-length komprimering jobbar genom att byta ut långa sekvenser av samma tecken mot endast ett tecken samt en siffra som anger hur många tecken som fanns innan (se figur 7). Filer med helt slumpmässiga tecken kan bli större efter run-length komprimering än utan någon komprimering alls.

Vid komprimering av naturligt språk och HTML-kod är run-length komprimering troligen inte speciellt bra då det är sällan som ett tecken upprepas fler än två gånger i följd i dessa texter. Det finns faktiskt en stor risk att komprimering av HTML-dokument med run-length komprimering resulterar i en större fil än den ursprungliga.

En implementation av run-length komprimering skulle gå snabbt då den inte är komplicerad. Den finns också väl beskriven i böcker och på nätet. Den kan inte användas för sig självt, eftersom komprimeringseffektiviteten på ASCII-filer inte är särskilt bra, men den är bra att använda i kombination med någon annan metod då det inte krävs så mycket extra processorkraft att köra en run-length komprimering också.

4.4.3 Ziv-Lempel komprimering

Ziv-Lempel komprimering (LZ-komprimering) bygger på en sk ordliste-teknik. Det går ut på att alla olika teckenkombinationer som dyker upp tilldelas en kod som lagras i en ordlista. Nästa gång samma teckenkombination dyker upp kommer den att hittas i

ordlistan och tilldelas samma kod som den tidigare kombinationen. Figur 8 visar ett exempel på en typ av LZ-komprimering.

Det svåra med LZ-komprimering är att välja vilka sekvenser som ska tilldelas en kod. Det existerar en bästa gruppering för varje datamängd, men att hitta denna gruppering innebär att man måste gå igenom varje möjlig kombination i hela datamängden, vilket är en oerhört tidskrävande process. Man går istället igenom datamängden byte för byte och tilldelar koder till de första sekvenser som upprepar sig. För att få ut så mycket som möjligt av komprimeringen måste varje kod representera strängar med så många bokstäver som möjligt, samtidigt är sannolikheten att en längre sträng ska upprepas flera gånger lägre än sannolikheten för att en kortare sträng upprepas.

I vanlig text finns många ord som förekommer ganska ofta, men vilka dessa ord är varierar beroende på vilket språk texten är på. HTML-koderna är däremot alltid desamma oavsett språk i resten av texten, och det är ofta samma ord som förekommer om och om igen. Man skulle därför kunna komprimera taggarna genom att tilldela en kod till varje textsträng som står avskild med ett mellanslag eller ett citationstecken. Problemet med denna strategi är att i taggen <A HREF=”http://www.his.se”> så kan inte själva webbadressen komprimeras med denna metod. Det bästa borde vara att välja en algoritm som letar efter mönster på vanligt sätt men som samtidigt uppmuntras att välja ord som är vanligt förekommande i HTML-kod.

LZ-komprimering är inte en specifik algoritm utan snarare en hel familj av algoritmer. De varianter som beskrivs i [BCW90] och [Sal97] är LZ77, LZR, LZSS, LZB, LZH, LZ78, LZW, LZC, LZT, LZMW, LZJ', LZFG, LZRW1, LZRW4, LZAP, LZY, och LZP1-4. Alla dessa algoritmer har det gemensamt att ju bättre de är på att komprimera, desto svårare är de att implementera. LZW är den algoritm som används av V.42bis protokollet som sitter i de flesta moderna modem.

4.4.4 Huffmankomprimering

Huffmankomprimering innebär att man byter ut representationer av tecken mot andra representationer. Genom att välja korta representationer för vanligt förekommande tecken och längre representationer för ovanliga tecken kan komprimering uppnås. Metoden fungerar bara om det finns tecken som förekommer oftare än andra, annars blir resultatet dåligt och kan i värsta fall resultera i att filen blir större efter komprimeringen. Vilken representation som ska tilldelas varje tecken bestäms av ett sk Huffmanträd (se figur 9). Det största problemet vid Huffmankomprimering är att bygga upp Huffmanträdet. Trädet kan skapas på tre olika sätt.

Vill man göra det lätt för sig så använder man ett statiskt träd, dvs ett träd som förblir detsamma under hela komprimeringen och som används varje gång någon fil ska komprimeras. Statiska träd har två fördelar:

- Implementationen blir enkel eftersom det enda man behöver göra är att hitta ett tecken i trädet och se vilken kod det ska få.
- Komprimeringen blir snabb eftersom man inte behöver ändra i trädet under exekveringen.

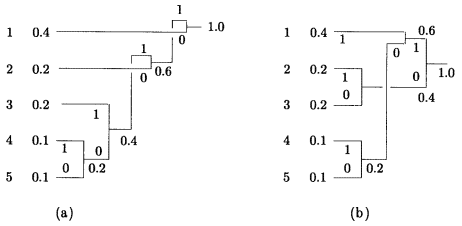
Nackdelen med statiska träd är att komprimeringen sällan blir optimal. Om trädet t ex är byggt för engelskt skriftspråk kommer det att fungera ganska dåligt på en svensk text. HTML-kod är ju i och för sig den samma hela tiden, men det är stor skillnad i

statistik över taggarna på en sida med mycket text och bilder och en sida med nästan enbart länkar.

Om det finns gott om tid att utföra komprimeringen kan ett träd byggas genom att hela datan gås igenom innan själva komprimeringen så att alla förekomster av olika tecken kan räknas. Datan kommer då att komprimeras med ett optimalt träd, och då kommer även komprimeringen att bli optimal [BCW90]. Tyvärr tar detta alldeles för lång tid för att vara användbart i realtidssammanhang, och dessutom måste trädet skickas med datan om dekodaren ska ha en chans att veta hur datan ska dekomprimeras.

Ett tredje alternativ är att bygga upp trädet samtidigt som komprimeringen utförs. Detta kallas dynamisk Huffmankomprimering och beskrivs i detalj under nästa rubrik.

Trädet byggs genom att utgå från en lista med sannolikheter för varje tecken i alfabetet. Ett exempel på hur det kan se ut finns i figur 15. Vissa regler måste följas när trädet byggs upp. En vital del av Huffmankoderna är att inga av de kortare koderna får kunna kombineras till att bilda en längre kod, då skulle inte dekodaren veta hur de skulle dekomprimeras. Trädet byggs upp genom att kombinera de två tecken som har lägst sannolikhet till en temporär symbol som har en sannolikhet som är summan av de två tidigare. Figur 15 visar hur det går till.



Figur 15: Huffmanträd.

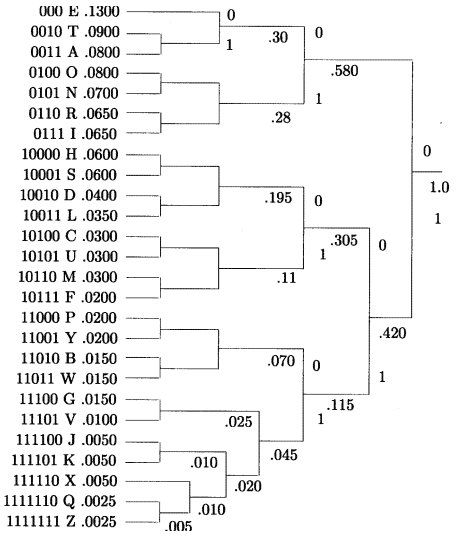
Trädet i figur 15a byggs nedifrån och upp på följande sätt:

1. a_4 kombineras med a_5 och båda byts ut mot den kombinerade symbolen a_{45} med sannolikheten 0.2.
2. Vi kan nu kombinera ihop vilka två som helst av a_2, a_3, eller a_{45} då de alla har samma sannolikhet. Eftersom a_3 och a_{45} står längst ned väljer vi att kombinera dem till symbolen a_{345} med sannolikheten 0.4.
3. Vi kan nu välja att kombinera a_1 med a_2 eller a_2 med a_{345}. För att vara konsekventa väljer vi att kombinera a_2 med a_{345} och bildar på så sätt symbolen a_{2345} med sannolikheten 0.6.
4. Slutligen kombineras a_1 med a_{2345} till symbolen a_{12345} med sannolikheten 1.

Nu återstår bara att sätta in ettor och nollor vid förgreningarna. Vilken gren som tilldelas vad spelar ingen roll så länge vi är konsekventa och alltid tilldelar samma siffra till den övre respektive undre delen genom hela trädet. Trädet är nu komplett och vi får ut Huffmankoderna för varje symbol genom att följa alla grenar från rotnoden till den nod vi vill ha koden för.

4 Metod

Vad händer då om vi inte väljer att kombinera symbolerna med den närmast nedanstående hela tiden? Antag att vi istället kombinerar dem som i figur 15b. Istället för koderna 0, 10, 111, 1101, och 1100 som vi får i 15a får vi koderna 11, 01, 00, 101, och 100. Detta ger en medelkodlängd för 10a på 0.4×1+0.2×2+0.2×3+0.1×4+0.1×4=2.2 bitar per symbol och en medelkodlängd för 10b på 0.4×2+0.2×2+0.2×2+0.1×3+0.1×3=2.2 bitar per symbol. Vilken modell vi än väljer kommer alltså medelkodlängden att bli densamma. Vilken modell är dä bäst? Svaret är enligt [Sal97] att modellen med minst varians är bäst. Variansen är ett mätt på hur mycket storleken på den individuella koden avviker från medelstorleken. Det visar sig att modellen i 15b har minst varians [Sal97]. För att få fram det bästa trädet måste vi alltså följa regeln: när det finns fler än två minsta noder att välja på så para ihop de två som är längst ned och högst upp i trädet. Figur 16 visar på ett större exempel med ett alfabet med 26 bokstäver.



Figur 16: Huffmanträd för alfabet med 26 bokstäver.

Dekomprimering av Huffmankod sker genom att dekomprimeraren läser in en bit i taget och följer vägen i trädet tills den kommer till en lövnod. Den skriver ut det tecken som står vid lövnoden och börjar sedan om vid rotnoden igen.

4.4.5 Dynamisk Huffmankomprimering

Med dynamisk Huffmankomprimering menas att Huffmanträdet byggs upp under exekveringen. Detta har fördelen att man kan arbeta med ett optimalt träd samtidigt som man slipper att räkna igenom hela texten i förväg. Dessutom behöver trädet inte skickas med den komprimerade filen, utan den följer med automatiskt genom att dekomprimeraren känner till hur komprimeraren bygger upp sitt träd. En närmare förklaring av hur det går till kan kanske vara på sin plats.

När komprimeringen börjar finns bara ett tomt träd. Komprimeraren läser in det första tecknet och skickar iväg det okomprimerat. Den sätter sedan in tecknet i Huffmanträdet och tilldelar det en sannolikhet. Nästa gång tecknet dyker upp komprimeras det med den aktuella koden och skickas iväg. Efter det uppdateras sannolikheten för tecknet i trädet och om det behövs så omstruktureras trädet och tecknet tilldelas en ny kod som kommer att användas nästa gång det komprimeras. Anledningen till att trädet uppdateras först efter det att tecknet komprimerats är att vid dekomprimering vet inte dekomprimeraren vilket tecken det är förän efter det är dekomprimerat. Först därefter kan alltså dekomprimeraren uppdatera sitt träd, och eftersom det är nödvändigt att både komprimeraren och dekomprimeraren uppdaterar sina träd på exakt samma sätt kan inte komprimeraren uppdatera sitt träd förän efter tecknet är komprimerat.

Problemet med denna metod är att dekomprimeraren måste kunna skilja på ett komprimerat och ett okomprimerat tecken. Detta görs genom att komprimeraren innan den skickar ett okomprimerat tecken sätter in ett sk ‘escapetecken’. Dekomprimeraren vet då att nästa 8 bitar är en okomprimerad ASCII-kod. Escapetecknet får inte ha samma kod som något annat tecken, och eftersom koden för alla andra tecken ändras hela tiden måste koden för escapetecknet också ändras. Detta löses genom att escapetecknet sätts in i trädet som ett tecken med sannolikheten noll, vilket gör att det alltid kommer att bestå av en rad nollor. Eftersom escapetecknet finns med i trädet kommer det att tilldelas en ny kod när trädet ändras.

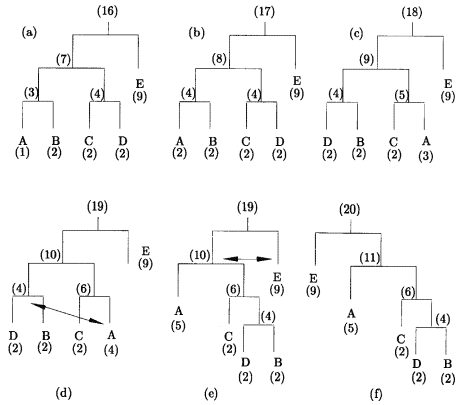
Det svåra med dynamisk Huffmankomprimering är själva uppdateringen av trädet efter att en ny symbol lästs in. Det skulle gå att räkna om hela trädet efter varje inläst tecken, men det skulle ta mycket lång tid och vore inte praktiskt möjligt. Istället uppdateras endast den nod i trädet som påverkas av tecknet och sedan kontrolleras närliggande noder och förgreningar för att se om även de behöver uppdateras. Om ingen av de närliggande noderna eller förgreningarna behöver uppdateras behöver inte någon annan del av trädet kontrolleras, och på så sätt sparas en hel del beräkning in.

För att visa på hur uppdateringen av trädet fungerar kommer här ett exempel ur [Sal97]. Trädet i figur 17a innehåller fem tecken: A, B, C, D, och E. Siffran inom parantes visar hur många gånger varje symbol förekommit efter det att 16 tecken lästs in. Trädet är ett Huffmanträd eftersom siffrorna står sorterade in stigande ordning om de läses från vänster till höger, nedifrån och upp. Om vi nu läser in en bokstav X med frekvens F kommer trädet att uppdateras i följande steg:

1. Jämför X med de intilliggande tecknen. Om den närmaste intilliggande noden och förgreningarna i det närmast överliggande lagret har frekvenserna F+1 eller mer är noderna fortfarande sorterade och ingen åtgärd behövs. Om så inte är fallet har någon eller några intilliggande nod(er) frekvensen F eller lägre. X ska då byta plats med den sista noden i denna grupp, men inte med någon överliggande förgrening.

2. Öka frekvensen för X från F till F+1.
3. Om X är rotnod så stanna, annars fortsätt utföra steg 1 med den närmast överliggande förgreningen till X.

Figur 17b visar trädet efter att ytterligare ett A har lästs in. Eftersom de intilliggande noderna inte har lägre frekvenser behöver inget göras, men vi kan se att om det skulle komma ytterligare ett A, som i figur 17c, så måste A flyttas längst åt höger, förbi alla med frekvensen 2. Om vi nu läser in ytterligare ett A uppstår situationen att A har samma frekvens som en överliggande nod. Detta illustreras i figur 17d. Nästa gång det kommer ett A måste vi alltså byta plats på A och förgreningen till DB. I figur 17e ser vi hur trädet ser ut efter det att ännu ett A läses in. Problemet nu är att den överliggande förgreningen till A har en högre frekvens än E, och alltså måste även dessa byta plats. Figur 17f visar det slutgiltiga trädet.



Figur 17: Uppdatering av Huffmanträd

En enklare variant av dynamisk Huffmankomprimering används av protokollet MNP-5 som sitter i de flesta moderna modem. Istället för ett Huffmanträd använder MNP-5 en tabell för att representera sannolikheten för olika tecken, vilket är snabbare och enklare att implementera men som förlorar en del i komprimeringseffektivitet. Den algoritm som beskrivs ovan är alltså långsammare och svårare att implementera, men den komprimerar avsevärt mycket bättre. Risken är att ovan beskrivna algoritm kräver allt för mycket processorkraft för att kunna användas i realtid.

4.4.6 Aritmetisk Komprimering

Huffmankomprimering jobbar genom att tilldela koder till ett tecken i taget. Ett tecken med sannolikheten 0.4 kommer således att representeras med en eller två bitar eftersom $-\log_2 0.4 \approx 1.32$. I idealfallet skulle detta tecken tilldelas en representation på 1.32 bitar, men detta är naturligtvis omöjligt. Aritmetisk komprimering kommer

runt detta problem genom att tilldela en enda kod till hela teckenmängden istället för till ett tecken i taget. Den resulterande koden kan ses som ett nummer i intervallet [0, 1). (Notationen [a, b) representerar det halvöppna intervallet $a \leq x < b$.) Det första som måste göras är att tilldela sannolikheter för varje tecken. Detta är inte nödvändigt om en dynamisk modell används, men det kommer vi till senare.

För att lättare kunna förklara algoritmen visas ett exempel ur [Sal97]. Antag att vi har tre symboler a_1, a_2, och a_3 med sannolikheterna P_1, P_2, och P_3. Intervallet [0, 1) delas då upp mellan symbolerna genom att tilldela ett subintervall med en storlek som är proportionell mot sannolikheten. Symbolerna kommer alltså att tilldelas subintervallen [0, 0.4), [0.4, 0.9), och [0.9, 1). Om vi vill komprimera strängen ”$a_2a_2a_2a_3$” börjar vi med intervallet [0, 1). Den första symbolen, a_2, reducerar detta intervall till subintervallet från dess 40% punkt till dess 90% punkt. Resultatet blir [0.4, 0.9). Nästa a_2 reducerar [0.4, 0.9) på samma sätt till [0.6, 0.85). (Beräknas som $0.4 + (0.9 - 0.4) \times 0.4 = 0.6$ och $0.4 + (0.9 - 0.4) \times 0.9 = 0.85$.) Det tredje a_2 reducerar detta till [0.7, 0.825) och a_3 reducerar slutligen detta till att gå från 90% till 100% av [0.7, 0.825), vilket ger [0.8125, 0.825). Den slutgiltiga koden kan vara ett godtyckligt tal i detta intervall. Istället för ”$a_2a_2a_2a_3$” kan vi alltså skicka t ex 0.813.

Som slutsats kan aritmetisk komprimering delas upp i följande steg:

1. Börja med att definiera nuvarande intervall som [0, 1).
2. Utför följande två steg för varje tecken i filen:

2.1 Dela upp det nuvarande intervallet i subintervall vars storlek är proportionella mot tecknens sannolikheter

2.2 Välj subintervallet för s och definiera det som nytt nuvarande intervall.

3. När hela filen gåtts igenom är ut-koden ett tal i det nuvarande intervallet.

För varje tecken som processas kommer intervallet att bli mindre, och det kommer att gå åt fler bitar för att representera det. Meningen med aritmetisk komprimering är att utdatan består av ett enda stort tal som representerar hela filen och inte ett tal för varje kodat tecken.

Det går också att använda en dynamisk modell som hela tiden uppdaterar sannolikheterna för de olika tecknen. Detta innebär en mera komplicerad modell som kräver en hel del beräkningar, men det påverkar inte själva kodtilldelningen. Kodaren bryr sig inte om vilket tal den får av modellen, den bara gör det den ska.

Det är dock inte riktigt så lätt att implementera aritmetisk komprimering som det kanske kan se ut ovan. Det finns några problem som måste lösas. Antag att en 1Mb textfil komprimeras. Resultatet av detta skulle kunna bli ett flyttal med en halv miljon decimaler, och även om man skulle hitta en representationsform för ett så stort tal skulle det ta mycket lång tid att utföra beräkningar på talet. Istället används heltal på 16 eller 32 bitar som representerar [låg, hög). Låg initieras till 0000 och hög till 9999. När komprimeringen utförs kommer intervallet mellan [låg, hög) att hela tiden minskas. När intervallet är så litet att den första siffran är lika, t ex låg=7000 och hög=7499, skrivs den första siffran till utfilen och de övriga siffrorna skiftas ett steg åt vänster. Efter skiftningen sätts en nolla in längst till höger på låg och en nia sätts in på hög. Om låg=7000 och hög=7499 kommer alltså en sjua att skrivas till utfilen och talen kommer efter skiftning att se ut som låg=0000 och hög=4999.

Dekomprimering sker genom att titta på storleken på det tal som blir resultatet av komprimeringen. Om talet är 0.6xxx och intervallet för Y är [0.5, 0.8) vet man att det första tecknet är Y. Man tar sedan bort effekterna av Y på koden som kod=(kod-LågIntervall(X))/Intervall, där LågIntervall(Y) är 0.5 och Intervall är 0.8 - 0.5 = 0.3.

Aritmetisk komprimering är den mest beräkningsintensiva komprimeringsalgoritmen av alla de som finns nämnda här. Det finns alltså en viss risk att den inte går att använda i en realtidsapplikation. Om den å andra sidan skulle gå att använda är den en kraftfull algoritm som når goda resultat med många typer av data.

Algoritmen för aritmetisk komprimering är optimal, dvs den tilldelar alltid varje tecken ett minimalt antal bitar. Det är mycket svårt om inte omöjligt att ta fram någon metod som komprimerar bättre. För att förbättra komprimeringseffektiviteten försöker man istället att ta fram bättre modeller för att beskriva sannolikheten för olika tecken. Till exempel kan man istället för att titta på sannolikheten för varje enskilt tecken titta på sannolikheten för ett tecken med hänsyn till vilka de föregående x tecknen varit. Som ett exempel på hur mycket effektivare denna metod kan vara kan sägas att sannolikheten för ett u i det engelska språket är ca 2%, men sannolikheten för ett u och föregående bokstav var ett q är 99% [Sal97].

4.5 Val av algoritmer

När komprimeringsmetod ska väljas finns enligt Brown och Shepherd ett antal begränsningar man bör ha i åtanke [BS95].

Begränsning	**Val**
Tid	Både realtidskomprimering och dekomprimering nödvändig. Endast realtidsdekomprimering nödvändig. Långsammare än realtid acceptabelt.
Kostnad	Kräver enbart mjukvara (låg kostnad) Kräver flyttalsprocessor (viss kostnad) Kräver speciell hårdvara (dyrt)
Dataförlust	Ingen data får gå förlorad. Viss data får förloras så länge det inte syns. Data får förloras så länge avsedd information når mottagaren.
Komprimerings-effektivitet	Låg komprimering tillräcklig (2:1 eller 3:1). Medelhög komprimering tillräcklig (5:1 eller 10:1). Hög komprimering krävs (20:1 till 100:1 eller högre).

Låt oss titta lite på vart och ett av de olika valen ovan.

Vad gäller komprimeringstiden måste algoritmen klara både realtidskomprimering och dekomprimering. Det finns ingen mening med att skicka över filer snabbt om man ändå måste sitta och vänta på att de ska dekomprimeras. Detta innebär att aritmetisk komprimering inte kan väljas. Aritmetisk komprimering är oerhört beräkningsintensiv och används snarare i de fall man vill uppnå hög komprimeringseffektivitet och tiden det tar att uppnå detta inte är viktig.

Kostnaden kommer inte att vara något problem eftersom det är meningen att programmet ska köras på den dator modemet sitter kopplat till, det krävs alltså enbart mjukvara. Om algoritmen är så resurskrävande att speciell hårdvara krävs är den inte intressant att använda.

4 Metod

Vad gäller dataförlust så får ingen data gå förlorad. Text skulle se mycket underlig ut om den inte dekomprimerades till exakt originalskick och HTML-koden måste vara syntaktiskt korrekt om webbläsaren ska kunna tolka den. Den förlustkomprimering som beskrivs i kapitel 4.4.1 innebär dock inte att någon data går förlorad. Den tar bara bort onödiga tecken som inte ska läsas av människor och som inte påverkar webbläsarens tolkning av datan. Då text inte kan förlustkomprimeras har inga förlustalgoritmer tagits upp i rapporten.

Komprimeringseffektiviteten finns det inget specificerat krav på. Beroende på hur datan ser ut kommer den att komprimeras olika och det enda kravet som finns är att komprimeringen ska vara mera effektiv än den som sitter i modem, dvs bättre än protokollen MNP-5 och V.42bis. Ingen av dessa algoritmer komprimerar speciellt bra. Enligt [Sal97] kan V.42bis, den bästa av de båda, uppnå en maximal komprimeringseffektivitet på 4:1, och detta endast i sällsynta fall. Av de algoritmer som beskrivs i kapitel 4.4 kan dock run-length komprimering väljas bort då denna ger mycket dåliga resultat vid textkomprimering.

Kvar att välja på finns då LZ-komprimering och Huffman. Innan vi går vidare och bestämmer oss för någon av algoritmerna kan det vara intressant att titta på hur protokollen i modemen fungerar. Genom att utgå från dessa protokoll kan kanske några slutsatser dras om vilka algoritmer som ger bäst resultat. MNP-5 använder sig av en tvåstegs-process med först en run-length komprimering och sedan en dynamisk frekvenskomprimering. Run-length algoritmen komprimerar endast i de fall ett tecken förekommer fem gånger i rad eller mer och är därmed i de allra flesta fall inte speciellt effektiv. Det andra steget tilldelar koder till olika tecken på 4 – 11 bitar på ett sätt som påminner om Huffman. Data som innehåller extremt få tecken (max 4 stycken) komprimeras till 2:1 medan data med många tecken med ungefär samma sannolikhet blir större efter komprimering. V.42bis använder en variant av LZW komprimering med dynamisk ordlista. V.42bis är bättre än MNP-5 i den mening att protokollet upptäcker när en fil blir större efter komprimeringen och istället skickas den okomprimerad. Om filen går att komprimera är V.42bis i de flesta fall bättre än MNP-5, den maximala komprimeringen sägs ligga på 4:1.

En bra idé vore att försöka kombinera det bästa från båda protokollen. Exempelvis kan en kombination av en Huffmanliknande algoritm och en LZ-algoritm användas. Då Huffman och LZ-algoritmer fungerar på helt olika sätt borde de kunna komplettera varandra bra. Då de två kvarvarande algoritmerna efter tidigare diskussion är just Huffman och LZ, är detta val lämpligt.

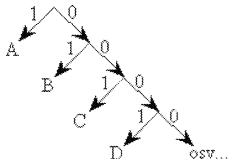
Den första av de båda algoritmerna, dvs den som ska specialanpassas för HTML-dokument, är en variant av Huffman. Denna valdes dels för att Huffman är enklast i sin funktion och därmed enklare att manipulera, dels för att vi troligen uppnår högre komprimeringseffektivitet genom att ha Huffmanalgoritmen först [Gus98].

Den andra algoritmen ska då vara en algoritm för generell data. I detta arbete kommer en LZSS-algoritm att användas. Det skulle fungera med vilken typ av LZ-komprimering som helst, och då LZSS-algoritmen är färdigimplementerad är det mycket lämpligt att välja denna, då det innebär en kraftig tidsbesparing. Algoritmen är också ganska snabb, vilket är ytterligare ett plus då algoritmen bör kunna komprimera data i realtid.

5 Genomförande

5.1 Implementering av HTML-anpassad Huffman-algoritm

Det mest komplicerade med Huffman-komprimering är att bygga upp Huffmanträdet. Man måste hålla reda på vart alla noder sitter i trädet, vilka noder som ligger ovanför dem och vilka tecken som går ihop i samma nod. I ett programmeringsspråk som C är detta mycket svårt att göra. För att underlätta kan man låt bli att göra trädet dynamiskt. Då slipper man bygga om trädet hela tiden, men det måste fortfarande byggas upp en gång. Det finns dock ett sätt att komma runt detta problem. Istället för att bygga upp trädet som i figur 15 kan man använda ett träd som aldrig förgrenar sig på grenar som går åt vänster. Ett sådant träd visas i figur 18.



Figur 18: Huffmanträd som endast förgrenar sig åt höger.

Ett sådant träd innebär att alla tecken kommer att tilldelas en kod som består av ett antal nollor och avslutas med en etta. Ju större sannolikhet ett tecken har desto färre nollor kommer det att ha innan ettan. Fördelen med ett sådant träd är att implementationen blir mycket enkel. Allt som behövs ar en lista med alla tecken sorterade efter fallande sannolikhet. När ett tecken ska tilldelas en kod gås listan igenom tills tecknet hittas. Om tecknet hittas som nummer x i listan skrivs x-1 nollor och en etta till den komprimerade filen. Nackdelen med metoden är att koderna blir långa. Det ovanligaste tecknet i en ASCII-text kommer att tilldelas hela 127 nollor! De enda tecknen som faktiskt komprimeras är de sju vanligaste. Detta vore naturligtvis helt oaccepterbart om enbart Huffman används eftersom i stort sett alla filer skulle bli mycket större än de var innan komprimeringen. Om man däremot ser Huffmankomprimeringen endast som ett sätt att förbereda filen inför ännu en komprimering med en annan algoritm kommer saken i ett helt annat läge. I vårt fall är den andra algoritmen en LZSS-algoritm. De sju vanligaste tecknen kommer att komprimeras av Huffmanalgoritmen och de andra tecknen kommer sedan att komprimeras mycket effektivt av LZSS-algoritmen då det blir lätt att hitta mönster med så många nollor på en rad. Det är egentligen ingen riktigt lyckad lösning att använda denna metod, men en Huffmanalgoritm som använder ett träd som i figur 15 är avsevärt komplexare att implementera.

5.1.1 Generell Huffman-algoritm med balanserat träd

Om vi tittar på komprimeringseffektiviteten hos Huffman-algoritmen på de HTML-dokument vars statistik beskrivs i diagrammen i figur 10-14 kommer vi till följande resultat (storleksangivelserna är i bytes och siffrorna inom parentes motsvarar komprimeringseffektiviteten):

Typ av text	Storlek	Komprimerad Huffman	Komprimerad Huffman+LZSS	Komprimerad enbart LZSS
Bilder	68055	114087 (1:1.68)	24625 (2.76:1)	12492 (5.43:1)
Länkar	157016	370863 (1:2.36)	103676 (1.51:1)	56138 (2.79:1)
Text	619865	1107379 (1:1.79)	363495 (1.71:1)	207914 (2.99:1)
Blandat	193901	430075 (1:2.22)	109241 (1.77:1)	61567 (3.14:1)
Alla	1038846	2223573 (1:2.14)	630415 (1.65:1)	337913 (3.08:1)

Tabell 2: Komprimeringseffektivitet hos generell Huffmanalgoritm

Som framgår av tabellen dubblerar Huffman-koden alla texttyperna varefter LZSS-koden komprimerar ner filen till ca halva ursprungsstorleken. Den resulterande filen blir i alla fallen ungefär dubbelt så stor efter komprimeringen än om endast LZSS-komprimeringen använts. Ändå räknades hela filen igenom innan Huffmankomprimeringen, så att en optimal sannolikhetsmodell användes. Vi kan med andra ord dra slutsatsen att denna metod inte var särskilt lyckad. Det är dock viktigt att notera att det endast är denna implementation av Huffman som inte fungerar särskilt bra. Om ett Huffmanträd så som det beskrivs i kapitel 4.4.4 används skulle ett bättre resultat uppnås [Gus98].

5.1.2 HTML-anpassad Huffman-algoritm

Det var ju dock meningen att en HTML-anpassad algoritm skulle användas, så hur ska algoritmen ovan ändras för att kunna komprimera HTML-koder? Lösningen är mycket enkel: Istället för att bara tilldela koder till olika bokstäver kan koder också tilldelas de vanligaste HTML-koderna. I och med att hela strängar representeras med en kod kan då storleken på den Huffman-komprimerade filen hållas nere och den totala komprimeringseffektiviteten ökas. I detta exempel valdes de 127 vanligaste taggarna ut genom att räkna dem manuellt, men det vore möjligt att lägga upp en databas med alla existerande HTML-koder och leta igenom dokumentet efter alla koderna. I detta exempel kommer också alla 127 koder alltid att tilldelas koder, vilket kanske inte alltid är helt lyckat. Antag t ex att vi har den situation som visas i diagrammet i figur 14. Tecknet ' " ' är det 20:e vanligast tecknet och '>' är det 4:e vanligaste. Det skulle då krävas att taggen ' ">' är minst det 23:e vanligaste tecknet om den inte ska tilldelas en längre kod än om varje bokstav i koden tilldelades en egen kod. Man skulle kunna tänka sig att komprimeringsprogrammet självt skulle kunna kontrollera om så är fallet, och om man inte tjänar något på att komprimera taggen så kommer det inte heller att ske. Detta skulle dock bli en mycket komplicerad kod, eftersom om taggen istället ska komprimeras tecken för tecken måste sannolikheterna för de tecknen ändras.

Vi ska nu titta på hur implementationen av Huffmankoden ser ut. Hela koden finns i bilaga 2. Beskrivningen kommer att bestå av pseudokod och börja med 'main'-funktionen och sedan fortsätta med varje funktion i den ordning de anropas av 'main'-funktionen.

Först definieras arrayerna 'teckenarray' och 'nummerarray'. Dessa kommer att innehålla alla de strängar som kommer att tilldelas koder och hur många gånger varje sträng förekommer i texten.

```
     int nummerarray[266];
     int teckenarray[256];

int main()
{
```

5 Genomförande

```
Nollställ nummerarray;
Läs in datafilen 'tabell.dat' till teckenarryen;
Fråga efter infilnamn;
Öppna infilen och skapa utfilen 'workfile';
```

Vi börjar med att nollställa 'nummerarray'. Det är denna array som håller reda på hur många förekomster det finns av varje tecken eller sträng.

'nummerarry' är kopplad till en teckenarray som innehåller alla de strängar som ska testas. Om en sträng hittas på plats 156 i teckenarrayen kommer nummerarray[156] att räknas upp ett steg. För att slippa en massa tilldelningar i koden och för att skapa en lite mer förändringsbar lista med strängar läses de strängar som ska användas in från en fil. Vill man ändra i sträng-listan behöver man inte gå in i koden och kompilera om, utan bara ändra i filen "tabell.dat". Denna fil finns som bilaga 3. Inläsningen till 'teckenarray' börjar på plats 128 eftersom plats 0-127 upptas av ASCII-koder. Inläsningen är här begränsad till att alltid vara 127 rader, men det skulle lätt kunna göras dynamiskt genom att man läser in till filslut påträffas. Man får då en mera dynamisk struktur, dvs man kan lägga till eller ta bort så många rader man vill i tabell.dat. 'teckenarray' är dynamiskt allokerad för att spara in lite minne och för att inte låsa maxlängden på strängarna.

En temporär fil, "workfile", skapas då komprimeringen sker i två steg. Andra steget kommer att använda sig av 'workfile' som infil.

Nu kommer själva huvudloopen i det första steget. Huffmanalgoritmen är indelad i två steg för att underlätta implementering. Den första delen räknar igenom hela texten och bygger upp statistik över förekomsten av olika tecken och strängar. Denna del utför också en sorts komprimering, vilket kräver att ut-datan lagras i en temporär fil som används som infil i nästa steg. Vad detta är för komprimering kommer vi till senare. Loopen går igenom filen tecken för tecken och räknar upp nummerarray för varje tecken som hittas. Om ett '<'-tecken läses in vet vi att vi kommit till en tagg och funktionen 'istag' anropas. 'istag' går sedan igenom taggen och letar efter de strängar som finns listade i 'tabell.dat'. 'istag' utgår alltid ifrån att ett '<'-tecken innebär att en tagg hittats. Så är emellertid inte alltid fallet i verkligheten, det skulle kunna vara ett vanligt 'mindre än'-tecken i texten. Detta problem kan lösas genom att man tittar framåt i filen och om det finns ett nytt '<'-tecken innan det kommer ett '>'-tecken vet vi att det inte var någon tagg.

```
Läs in ett tecken från infilen;
Så länge inte filslut {
     Om tecken == <
          Anropa 'istag';
     Annars {
          Räkna upp tecknets förekomst i nummerarryen;
          Skriv ut tecknet till den temporära filen 'workfile';
     }
     läs in nästa tecken;
}
Sortera nummerarrayen;
Öppna 'workfile' som infil;
Skapa filen 'utfil.huff' som ny utfil;
Läs in datafilen 'tabell2.dat' till teckenarrayen;
```

'nummerarray' innehåller nu antalet förekomster av alla tecken och strängar i texten. Nu måste 'nummerarray' sorteras så att vi vet hur många nollor som ska tilldelas varje värde. 'nummerarray' sorteras av funktionen 'sort()'. 'sort' är en vanlig bubblesort och kommer inte att beskrivas närmare. Efter sorteringen innehåller 'nummerarray' istället

för antalet förekomster av ett tecken eller en sträng en siffra som anger hur många nollor tecknet/strängen ska kodas med.

'workfile' är nu infil och den slutliga komprimerade datan lagras i filen 'utfil.huff'. Innan vi kan börja med det andra steget måste vi lägga in nya strängar i teckenarrayen. Detta beror på vissa omändringar som gjorts i texten av det första steget. Vilka dessa förändringar är kommer vi till om en stund. De nya strängarna finns i filen 'tabell2.dat' som finns i bilaga 4.

Huvudloopen i den andra delen är mycket lik den i den första delen. Dokumentet letas igenom på precis samma sätt, men istället för att räkna upp 'nummerarray' för det tecken eller den sträng som hittats kommer så många nollor att skrivas till utfilen som anges av tecknets/strängens plats i 'nummerarray'.

```
Läs in tecken från infilen;
Så länge inte filslut {
     Om tecken == <
          Anropa funktionen huffman();
     Annars
          Anropa skriv_nollor() med nummerarray;
     Läs in nästa tecken från infilen;
}
```

Funktionen 'skriv_nollor' skriver ut så många nollor till utfilen som anges av nummerarrayen. Om 'skriv_nollor' skriver ut tre nollor kommer dessa att sparas i en variabel tills de fyller ut en hel byte (8 bitar). Byten fylls på varje gång 'skriv_nollor' anropas.

Vi ska nu titta lite närmare på den första funktionen som anropas av 'main', nämligen 'istag'. 'istag' anropas när 'main'-funktionen läser in ett '<'-tecken, och läser då vidare i filen tills den stöter på ett '>'-tecken, vilket innebär att taggen är slut. Taggen får inte vara längre än 255 tecken om den ska få plats i arrayen. Det finns ingen felkontroll om taggen skulle överskrida detta värde, men man skulle kunna tänka sig att den då betraktas som vanliga tecken.

```
int istag()
{
     Läs in tecken till en array till ett '>'-tecken hittas
     Så länge det finns tecken kvar i arrayen {
          Anropa jamfor();
          Skifta arrayen åt vänster lika många tecken som storleken
          på den match som hittades av jamfor();
          Om storleken på matchen var större än ett
               Anropa funktionen space_comp();
     }
```

När hela taggen är inläst anropas funktionen 'jamfor'. Om de x första tecknen i taggen motsvarar någon sträng av längd x i teckenarrayen kommer 'jamfor' att returnera detta x. Taggen skiftas sedan åt vänster x steg så att den matchade strängen försvinner och sedan anropas 'jamfor' igen. x kan vara 1 om ingen sträng hittades.

Om storleken på matchen är större än ett innebär det att den kan vara en HTML-kod, och då anropas funktionen 'space_comp'. Vad denna funktion gör beskrivs senare.

Funktionen 'jamfor' går igenom taggen och letar efter en match med någon teckenkombination i teckenarrayen.

```
int jamfor()
{
```

```
Kopiera taggen till en temporär array;
Omvandla den temporära arrayen till versaler;
Leta igenom den temporära arrayen efter strängar som matchar
någon sträng i teckenarrayen;
Om match {
     Räkna upp antalet förekomster för strängen i nummerarray;
     Skriv ut strängen till 'workfile';
}
```

'jamfor' utför egentligen två jämförelser. Först konverteras de tolv första tecknen i strängen till stora bokstäver och jämförs, sedan jämförs strängen som den såg ut från början. Anledningen till detta är att alla HTML-koder kan skrivas med stora eller små bokstäver utan att det spelar någon roll. Genom att konvertera alla HTML-koder till stora bokstäver kan antalet möjliga kombinationer hållas nere och det behövs inte två olika koder till taggarna <html> och <HTML>.

Det är endast tolv tecken som konverteras åt gången eftersom den längsta strängen i teckenarrayen är tolv tecken lång. Man skulle kunna konvertera hela strängen, men det skulle bara ta tid eftersom det måste göras om varje gång 'jamfor' anropas.

Om en sträng hittas skrivs den ut till 'workfile' och nummerarrayen räknas upp ett steg.

Även de små bokstäverna måste sedan jämföras, då det finns strängar med små bokstäver i teckenarrayen. Denna jämförelse fungerar precis på samma sätt som den tidigare.

```
     Annars {
          Leta igenom originalarrayen efter strängar som matchar
          någon sträng i teckenarrayen;
          Om match {
               Räkna upp förekomster för strängen i nummerarray;
               Skriv ut strängen till 'workfile';
          }
          Annars {
              Räkna upp förekomsten för det första tecknet i arrayen;
              Skriv ut tecknet till 'workfile';
          }
     }
     Returnera storleken på matchen;
}
```

Om ingen match hittas räknas det första tecknet som ett ensamt tecken, det skrivs ut till 'workfile' och förekomsten för det räknas upp.

'jamfor' returnerar storleken på den matchade strängen.

Som nämns under rubrik 4.4.1 kan mellanslag ibland tas bort i taggarna. T ex kan strängen href•=•" bytas ut mot href=". Funktionen 'space_comp' gör just detta. I funktionen 'istag' kontrolleras om 'jamfor' hittat någon match som är längre än ett tecken. Om så är fallet finns en möjlighet att matchen är en HTML-kod och att det finns mellanslag som kan tas bort. Alltså anropas 'space_comp'. Funktionen är mycket enkel. Om tecknet direkt efter matchen är ett mellanslag följt av ett likamed-tecken kan mellanslaget tas bort. Är det ett likamed-tecken följt av ett mellanslag kan även detta mellanslag tas bort. Om något mellanslag tas bort skrivs likamed-tecknet ut till 'workfile'. Sedan skiftas arrayen så många steg åt vänster som togs bort och storleksanvisningen på arrayen miskas och returneras. Det är denna förlustkomprimering som gör att en ny teckenarray (tabell2.dat i bilaga 4) kan användas. Istället för att tilldela 'href' en kod i nästa steg kan 'href="' tilldelas en kod,

vilket innebär att vi tjänar in två bytes. Dessutom tjänar vi in en byte eller två på att ta bort mellanslagen.

```
int space_comp()
{
     Skapa variabler;
     Om arrayen börjar med '•=' {
          Skriv '=' till utfilen;
          Om nästa tecken är ett '•'
               Skifta arrayen tre steg åt vänster för att bli av med
               strängen '•=•';
          Annars
               Skifta arrayen två steg åt vänster för att bli av med
               strängen '•=';
     }
     Annars om arrayen börjar med '=•' {
          Skriv '=' till utfilen;
          Skifta arrayen två steg åt vänster för att bli av med
          strängen '=•';
     }
     Returnera nuvarande storlek på arrayen;
}
```

Nu är det dags att titta närmare på det andra steget i Huffmankomprimeringen, dvs själva tilldelningen av koderna. Först anropas 'sort' som sorterar 'nummerarray' och istället för antal förekomster av varje tecken eller sträng kommer 'nummerarray' efter sorteringen att innehålla antal nollor som varje tecken/sträng ska kodas med. En sortering av 'nummerarray' skulle dock innebära att den koppling som finns mellan värdena i 'nummerarray' och de som finns i 'teckenarray' förloras. Därför måste vi på något sätt hålla reda på vilken plats de ursprungliga värdena i 'nummerarray' låg. Detta löses med en temporär array med värdena 1, 2, …, n som ändras om samtidigt som 'nummerarray'. Efter sorteringen kan vi då se att om det 3:e värdet i den temporära arrayen har värdet 7 var det 7:e tecknet i den ursprungliga nummerarrayen det 3:e vanligaste och ska därför tilldelas två nollor. På så sätt kan 'nummerarray' återskapas efter sorteringen.

Innan det andra steget påbörjas läses den nya tabellen med strängar in. Som det ser ut nu är det bara tillagt ett likamed-tecken på några av HTML-koderna, men det finns fler mönster som kan utnyttjas. Vissa koder har alltid ett citationstecken efter likamed-tecknet och andra har alltid eller ofta ett #-tecken efter citationstecknet osv. Genom att sätta sig in mera noggrant i HTML-koder kan säkert tabell2.dat förbättras.

Nu påbörjas själva Huffmankomprimeringen. Denna andra del är mycket lik den första där texten räknades igenom. Enda skillnaden är att istället för att 'nummerarray' räknas upp när ett tecken eller en sträng hittats så skrivs ett antal nollor till den slutliga filen 'utfil.huff'. Funktionerna 'huffman' och 'hitta_nollor' är så lika 'istag' och 'jamfor' att de inte behöver förklaras närmare.

En liten skillnad i 'hitta_nollor' jämfört med 'jamfor' är att strängen inte behöver konverteras till stora bokstäver då detta redan är gjort från det första steget.

Då antalet nollor som ska skrivas till utfilen har hittats anropas funktionen 'skriv_nollor'. Denna funktion tar som invärde ett nummer som specificerar hur många nollor som ska skrivas ut. 'skriv_nollor' fyller på en variabel med nollor och när variabeln blir 'full' skrivs den till fil.

Nu är filen utfil.huff klar och vi kan påbörja komprimering med LZSS-algoritmen.

5.2 LZSS-algoritmen

LZSS-algoritmen är skriven av Haruhiko Okumura. Koden är inte belagd med något upphovsrättsskydd. Då det är Haruhiko som skrivit koden kommer bara grundtanken bakom LZSS-komprimering att gås igenom.

Namnet LZSS kommer av att algoritmen togs fram som en förbättring av LZ77-algoritmen av Storer och Szymanski [Sal97]. Alla LZ-algoritmer går ut på att strängar lagras i en 'ordlista' och om en inläst sträng finns med i ordlistan kommer en referens till denna sträng att lagras istället för strängen själv. För att uppnå komprimering krävs att referensen är kortare än strängen, och ju längre strängar som finns i ordlistan desto högre komprimering kan uppnås.

LZSS lagrar sin ordlista i ett sk binärt sökträd. Ett binärt sökträd är ett träd där det vänstra subträdet till varje nod N innehåller noder som är mindre än N och det högra subträdet innehåller noder som är större än N. Då noderna innehåller strängar jämförs först den första bokstaven i varje sträng. Ju längre bak i alfabetet en bokstav finns desto större räknas den att vara. Om den första bokstaven är lika i de båda strängarna går man vidare till den andra bokstaven, osv. Då strängarna ligger i ett träd istället för i en lista kan söktiden för att hitta en sträng minskas avsevärt. Den LZSS-algoritm som används här använder 256 olika träd, ett för varje möjligt första tecken.

Det svåra med LZ-komprimering är att uppdatera trädet. Om alla strängar som någonsin lästs in skulle sparas hela tiden skulle algoritmen använda oerhört mycket minne, därför kastas gamla strängar bort när en ny sträng läses in och det maximala antalet strängar har uppnåtts. LZSS löser detta genom att helt enkelt alltid kasta bort de äldsta strängarna, även om det finns nyare strängar som inte förekommit alls lika ofta. Detta gör implementationen enklare, men ger inte lika bra komprimering som om den sträng som har lägst frekvens kastas bort.

De referenser som lagras istället för strängarna är i LZSS två bytes långa. Alla matcher som är endast en eller två tecken långa kommer inte att komprimeras. De skrivs istället till utfilen i okomprimerad form tillsammans med en flagga som talar om att de inte är komprimerade. Referenserna är uppdelade i två delar, en del som talar om vart i trädet vi refererar och en del som talar om hur lång matchen är.

Funktionerna för LZSS-komprimeringen finns som InitTree, InsertNode, DeleteNode, Encode, samt Decode i koden i bilaga 5.

5.3 Komprimeringseffektivitet hos algoritmen

Nu är algoritmen implementerad och vi kan testa hur bra den fungerar. I testet har samma filer använts som i tabell 2 och resultatet visas här nedan i tabell 3. Storleksangivelserna är i bytes och siffrorna inom parentes är komprimeringseffektiviteten.

Typ av text	Storlek	Html-Huffman	Generell Huffman	Html-Huffman +LZSS	Enbart LZSS
Bilder	68055	79120 (1:1.16)	114087 (1:1.68)	16584 (4.10:1)	12492 (5.43:1)
Länkar	157016	327627 (1:2.09)	370863 (1:2.36)	90399 (1.74:1)	56138 (2.79:1)
Text	619865	1019652 (1:1.64)	1107379 (1:1.79)	345490 (1.79:1)	207914 (2.99:1)
Blandat	193901	339955 (1:1.75)	430075 (1:2.22)	89743 (2.16:1)	61567 (3.14:1)
Alla	1038846	1993303 (1:1.92)	2223573 (1:2.14)	566184 (1.83:1)	337913 (3.08:1)

Tabell 3: Komprimeringseffektivitet hos HTML-anpassad Huffmanalgoritm med LZSS

Som vi kan se i tabellen ger den HTML-anpassade algoritmen en bättre komprimering än den generella algoritmen, men efter LZSS-algoritmen är den komprimerade filen fortfarande större än om enbart LZSS-algoritmen använts. Detta beror förmodligen på att mängden taggar helt enkelt inte är tillräckligt stort för att Huffmankoden ska kunna komprimera den ordentligt. Dessutom beror det på att LZSS-algoritmen är så bra på att hitta mönster att Huffmankoden helt enkelt är helt onödig. Det skulle alltså ge bättre komprimeringseffektivitet med enbart LZSS samtidigt som det skulle gå bra mycket fortare! Det skulle gå att få upp komprimeringseffektiviteten hos Huffman-algoritmen genom lite manipulationer med tabellerna (tabell.dat och tabell2.dat) men det skulle knappast göra tillräckligt för att komma ikapp enbart LZSS-algoritmen och definitivt inte tillräckligt för att det skulle vara värt den extra komprimeringstiden eller det extra arbetet med att genomföra förändringen.

5.4 Komprimering av taggar genom överskottsbitar från ASCII

Det verkade alltså inte som om Huffman-algoritmen var någon bra idé. Istället testar vi en annan metod, som bygger på ungefär samma princip, men som endast komprimerar HTML-koder i det första steget istället för alla bokstäver. Alla HTML-dokument är skrivna i ASCII-kod, och även om varje tecken tar upp 8 bitar minnesutrymme används egentligen bara 7 av bitarna, eftersom ASCII bara består av 128 tecken (se tabell 1). För varje byte som skrivs till filen finns således 128 outnyttjade kombinationer som kan användas för att representera taggarna. Fördelen med denna metod är att de 128 strängar som tilldelas koder mellan 128 och 255 alltid kommer att komprimeras till en byte, oavsett hur stora de är. Genom att använda samma 127 strängar som finns i tabell.dat behövs inget extra arbete utföras. Även största delen av koden från Huffmankomprimeringen kan återanvändas.

Istället för att räkna upp nummerarray i koden räcker det med att vi hittar strängen i teckenarrayen. Om strängen hittas på plats p skrivs talet p till utfilen. Detta fungerar eftersom strängarna bara ligger på platser över 128 i teckenarrayen. Metoden innebär att det finns en begränsning på 128 strängar som kan tilldelas koder, men i de exempel som används här förekommer det över huvud taget inte att fler än 128 HTML-koder finns med flera gånger i samma dokument. Även om det är mycket sällsynt att fler än 128 HTML-koder används i ett dokument är det mycket möjligt att det används sådana som inte finns med i listan. Det bästa vore alltså att ha en större lista med alla existerande koder och sedan välja ut de 128 vanligaste att sätta in i teckenarrayen. För att slippa räkna alla taggarna, vilket skulle ta upp dyrbar extratid, kan man helt enkelt tilldela koder till de första 128 koderna man stöter på. Detta skulle troligen inte medföra någon allvarlig begränsning då dokument med fler än 128 olika HTML-koder är mycket sällsynta.

5 Genomförande

Nackdelen med denna metod är att alla existerande taggar måste lagras med programmet, men i våra dagar tar en vanlig ordbehandlare lätt upp 30 Mb hårddiskutrymme, så om komprimeringsprogrammet tar upp ett par kilobytes extra är det ändå ingen som märker. Då HTML-koderna alltid är desamma behöver de bara inkluderas en gång och sedan aldrig bytas ut. De 128 koder som väljs ut måste inkluderas i den komprimerade filen så att dekomprimeraren vet hur filen ska dekomprimeras, men om de taggar som ska finnas med skickas okomprimerade första gången de dyker upp, på samma sätt som andra dynamiska metoder fungerar, behöver det inte ta upp så mycket extra plats.

Koden är fortfarande uppdelad i två delar. Detta kan tyckas onödigt eftersom texten inte behöver räknas igenom, då det inte behövs någon statistik över vilka tecken som är vanligast. Dock behöver taggarna fortfarande göras om till stora bokstäver och utrymme kan tjänas in om mellanslagen tas bort på samma sätt som vid Huffmankomprimeringen.

Om vi tittar lite närmare på koden för denna komprimeringsalgoritm ser vi att den är mycket lik koden för Huffmankomprimeringen. Hela koden, med koden för LZSS-komprimeringen inkluderad, finns i bilaga 5. Dä algoritmerna för denna metod och Huffman är så lika kommer denna kod att gås igenom mera ytligt.

I början finns deklarationer av globala variabler som är i stort sett desamma som för Huffmankoden samt några som behövs för LZSS-komprimeringen. De första funktionerna InitTree, InsertNode, DeleteNode, Encode, och Decode hör även de till LZSS-komprimeringen.

En av de största förändringarna av ’main’-funktionen är att det finns ett val för komprimering eller dekomprimering. Det kan ju vara bra att kunna dekomprimera sin fil efter att den har komprimerats. Hur själva dekomprimeringen fungerar kommer vi till om en liten stund.

Om komprimering väljs kommer tabellen med strängar att läsas in på samma sätt som tidigare, och sedan läses texten igenom tecken för tecken, även det på samma sätt. Skillnaden är att om ett vanligt tecken läses in kommer detta tecken att skrivas ut till den temporära filen utan att något annat behöver göras. Om tecknet var ett ’<’ anropas som tidigare funktionen ’istag’.

```
int main()
{
     ...
     Välj komprimering eller dekomprimering;
     Om dekomprimering {
          ...
          Anropa funktionen decompress();
     }
     Annars {
          ...
          Läs in ett tecken från infilen;
          Så länge inte filslut påträffas {
               Om tecknet är ett '<'
                    Anropa funktionen istag();
               Annars
                    Skriv ut tecknet till utfilen;
               Läs in nästa tecken från infilen;
          }
          ...
```

Sedan läses 'tabell2.dat' in och andra fasen i komprimeringen påbörjas. Denna andra fas fungerar som andra fasen i Huffmankoden, men istället för att tilldela koder anropas 'compress' som helt enkelt skriver ut en kod till utfilen istället för ett antal nollor.

```
...
Läs in ett tecken från infilen;
Så länge inte filslut påträffas {
     Om tecknet är ett '<'
          Anropa funktionen compress();
     Annars
          Skriv ut tecknet till utfilen;
     Läs in nästa tecken från infilen;
}
...
```

Slutligen utförs LZSS-komprimeringen som vanligt.

'istag' ser ut precis som den gjorde i Huffmankoden, och även 'space_comp' fungerar precis som tidigare.

Skillnaderna i 'jamfor' är att om ingen match hittas skrivs första tecknet i taggen ut på utfilen utan att någon sannolikhet behöver räknas upp. Annars fungerar 'jamfor' på samma sätt med konverteringen till stora bokstäver och att storleken på matchen returneras.

I andra fasen anropar 'main' först funktionen 'compress'. 'compress' fungerar som funktionen 'huffman' i Huffmankoden. Enda skillnaden är att den anropar 'hitta_strang' istället för 'hitta_nollor'.

'hitta_strang' fungerar precis som 'hitta_nollor'. Skillnaden blir när en match har hittats. Istället för att kalla på funktionen 'skriv_nollor' kommer strängens plats i teckenarrayen att skrivas till utfilen. Om ingen sträng hittas skrivs det första tecknet i taggen ut till filen omanipulerat.

```
int hitta_strang()
{
     ...
     Leta igenom arrayen efter strängar som matchar någon sträng i
     teckenarrayen;
     Om match
          Skriv ut den siffra till utfilen som motsvarar på vilken
          plats i teckenarrayen strängen hittades;
     Annars
          Skriv ut det första tecknet i arrayen till utfilen;
     returnera storleken på matchen;
}
```

Låt oss nu titta på hur väl algoritmen fungerar på samma texter som testats tidigare. Talen inom parantes anger komprimeringseffektiviteten, dvs den komprimerade filens storlek dividerat med den ursprungliga filens storlek. Storleksangivelserna är i bytes och talet inom parentes är komprimeringseffektiviteten. Talet inom hakparenteser vid "ASCII-algoritm+LZSS" anger komprimeringseffektiviteten hos LZSS algoritmen om man ser den ASCII-komprimerade filen som ursprunglig fil.

Typ av text	Storlek	ASCII-algoritm	ASCII-algoritm +LZSS	Enbart LZSS
Bilder	68055	46872 (0.689)	9815 (0.144) [0.209]	12492 (0.184)
Länkar	157016	111550 (0.710)	50197 (0.320) [0.450]	56138 (0.358)
Text	619865	530754 (0.856)	195440 (0.315) [0.368]	207914 (0.335)
Blandat	193901	140988 (0.727)	53480 (0.276) [0.379]	61567 (0.318)
Alla	1038846	830173 (0.799)	308782 (0.297) [0.372]	337913 (0.325)

Tabell 4: Komprimeringseffektivitet hos HTML-anpassad Huffmanalgoritm med LZSS

Som framgår av tabellen verkar det alltså som om denna algoritm fungerar riktigt bra! Som syns i tabellen är algoritmen klart bäst på ”Bilder”-filen, och även LZSS är klart bäst på denna fil. Det beror på att denna text innehåller en mycket hög procentandel taggar, och att både ASCII-algoritmen och LZSS-algoritmen är bra på att komprimera sådana texter. LZSS-algoritmen arbetar genom att hitta mönster i datan, och HTML-koderna ger just sådana mönster då samma koder, och även kombinationer av koder, förekommer ofta i texten.

5.5 Jämförelse med modem

Nästa steg är nu att jämföra algoritmen med MNP-5 och V.42bis som sitter i modem. Enligt [Sal97] kan den maximala komprimeringseffektiviteten hos MNP-5 sägas ligga på 2:1 och hos V.42bis kan den sägas vara 4:1. I tabell 4 kan vi utläsa att den sämsta komprimeringen, den för ”Länkar”, ligger på ungefär 1:3, vilket är bättre än den maximala förmågan hos MNP-5. Då V.42bis kan ses som den klart bättre av de båda protokollen och då ASCII-algoritmen tycks komprimera bättre än MNP-5 även i det sämsta testfallet är det inte speciellt intressant att göra någon jämförelse med MNP-5, utan endast en jämförelse med V.42bis kommer att utföras.

I tabell 4 kan vi också se att den bästa komprimeringen, den för ”Bilder”, nästan kommer upp i hela 1:7, vilket ligger en bra bit över V.42bis bästa. Det verkar alltså som om ASCII-algoritmen slår de protokoll som sitter i modemen, men det går naturligtvis inte att säga säkert förrän efter ett test.

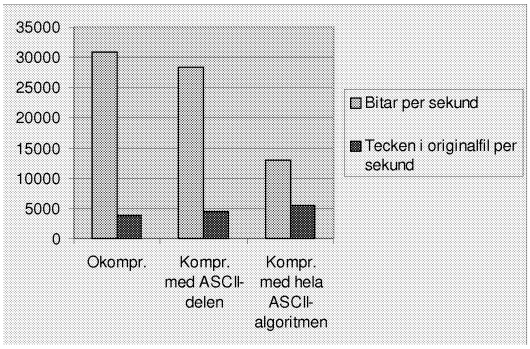
Då själva specifikationen till V.42bis-algoritmen inte finns att tillgå på ett enkelt sätt har inte några exakta tester kunnat utföras. Testet har därför utförts genom att en fil skickats dels okomprimerad, dels komprimerad genom ett V.32bis modem. V32bis är en standard för modem som arbetar med 14400 bps. Modemet använder V.42bis-protokollet för datakomprimering. Genom att titta på överföringstiden kan vi då dra slutsatser om hur effektiv ASCII-algoritmen är i jämförelse med V.42bis-protokollet. Resultatet kommer inte att bli lika exakt som om filerna kunnat testkomprimerats med de båda olika algoritmerna, men det räcker till för att ge en grov uppskattning.

Den fil som överförts är en sammanslagning av alla de filer som använts i tidigare test. Detta motsvarar posten ’Alla’ i de tabeller och diagram som förekommit tidigare i rapporten. Filen har först skickats okomprimerad, sedan komprimerats med enbart ASCII-delen av algoritmen, och slutligen komprimerats med hela ASCII-algoritmen. Då överföringstiderna varierat en smula har ett medelvärde av tre överföringar använts. Denna variation i överföringstiderna kan t ex bero på brus i uppkopplingen som gör att några bytes måste skickas igen eller liknande. Spridningen i överföringstiderna var låg, som mest knappt 2%. Denna låga siffra kan knappast ogiltigförklara resultatets trovärdighet. Resultaten från överföringstesterna kan utläsas i tabell 5 och figur 19.

Cps i tabellen innebär 'characters per second', eller tecken per sekund. Denna siffra ger ett mått på hur snabbt överföringen av hela filen går. Högre bps-värde innebär bättre komprimering i modemet och högre cps-värde innebär snabbare total överföringstid, dvs mera effektiv komprimering totalt sett.

Fil	Storlek (bytes)	Överföringstid (sekunder)	Motsvarar överföringshastighet (bps) [cps]
Okomprimerad	1038846	269	(30895) [3862]
Komprimerad med enbart ASCII-delen	830704	234	(28400) [4440]
Komprimerad med hela ASCII-algoritmen	309031	191	(12944) [5439]

Tabell 5: Överföringshastigheter med och utan ASCII-komprimering.



Figur 19: Tabell 5 i diagramform.

I tabellen och diagrammet kan vi utläsa följande: Överföringshastigheten för den okomprimerade filen ligger på nästan 31000 bps trots att V.32bis modemet jobbar med 14400 bps. Detta innebär att V.42bis-protokollet kan komprimera filen till ungefär 2.1:1. Överföringshastigheten motsvarar nästan 3900 tecken per sekund i originalfilen. Utan någon form av komprimering kan ett V.32bis modem skicka 1800 tecken per sekund. Den fil som är komprimerd med enbart ASCII-delen av ASCII-komprimeringen (dvs inte LZ-algoritmen) skickas med ungefär 28000 bps, vilket visar att modemet även här använder V.42bis-protokollet för att komprimera datan. Komprimeringen är inte lika effektiv, men den hjälper helt klart till. Det intressanta är att trots att själva överföringshastigheten har minskat en aning har antalet tecken i originalfilen som överförs per sekund ökat. Detta innebär att den totala komprimeringseffektiviteten ökat och att en tidsvinst gjorts vid överföringen. Om vi

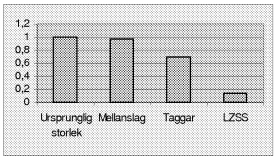
slutligen tittar på den fil som är komprimerad med hela ASCII-algoritmen kommer vi till det verkligt intressanta. Överföringshastigheten för filen är ungefär 13000 bps, vilket ligger lite under de 14400 bps som modemet jobbar med. Anledningen till detta kan vara dels att en del extra bitar måste skickas för att synkronisera modemen, dels att en del bytes måste skickas om vid överföringsfel, t ex pga brus i uppkopplingen. Att överföringshastigheten är så låg är ett bevis på att V.42bis-protokollet inte använts för komprimering, det stängs nämligen av om inte en viss komprimeringsnivå uppnås. Trots att protokollet alltså inte använts och att överföringshastigheten därmed tycks vara inte ens hälften så hög som i de båda övriga fallen, kan vi se att den totala överföringstiden är betydligt lägre, dvs antalet tecken i originalfilen som överförs per sekund är högre. Antalet överförda tecken per sekund är över 5400, eller drygt 40% högre än om endast V.42bis-protokollet används.

5.6 Analys

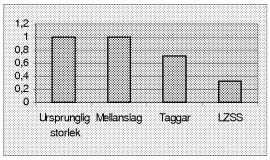
Som vi kan se i tabell 4 komprimerar ASCII-algoritmen HTML-dokumenten relativt bra. ”Text”-filen komprimeras inte så mycket av ASCII-delen men detta vägs upp av LZSS-delen i det andra steget. Det verkar alltså som om de båda algoritmerna jobbar bra tillsammans. ”Länkar”-filen är den som komprimeras klart sämst av LZSS-delen, men samtidigt komprimeras den mycket bra av ASCII-delen vilket gör att komprimeringseffektiviteten för denna fil inte längre är lika tydligt sämst som tidigare.

En mera detaljerad bild över hur mycket filerna komprimeras i varje steg i algoritmen visas i figur 20. I figuren ser vi att det första steget egentligen inte utför någon komprimering alls. Detta första steg förbereder egentligen bara filen inför steg 2. Hela steg 1 kan ersättas av att tabellen i steg 2 utökas till att innehålla alla olika kombinationer av mellanslag som kan korrigeras av steg 1. En sådan förändring skulle innebära att steg 2 tar betydligt längre tid att genomföra, men det vore intressant att se om det verkligen tar så lång tid att det inte uppväger steg 1.

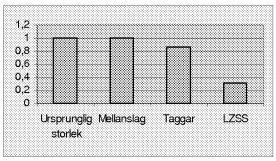
I figuren kan vi också se hur väl taggkomprimeringen fungerar. Av naturliga skäl är komprimeringseffektiviteten på textfilen inte speciellt hög, medan bild- och länk-filerna komprimerats bra. Intressant är att även de blandade filerna komprimerats bra. ”Bilder” och ”Länkar” är speciellt utvalda för att de innehåller många taggar, men ”Blandat”-filerna representerar ”vanliga” sidor på nätet. I och med att ”Blandat”-filerna komprimeras bra kan vi dra slutsatsen att algoritmen kan tillföra förbättrad komprimering vid normal nätanvändning och därmed kan vara användbar för alla nätanvändare. Hade endast ”Bilder” och ”Länkar” komprimerats bra hade algoritmen inte tillfört något förutom vid enstaka tillfällen.



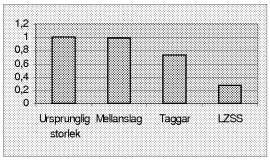
Bilder



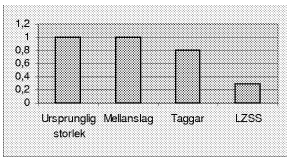
Länkar



Text



Blandat



Alla

Figur 20: Komprimeringseffektivitet hos de olika stegen i algoritmen.

Hittills fungerar allt tillfredsställande. ASCII-algoritmen ger en klar fördel gentemot att använda enbart LZSS-komprimeringen och enligt testet i kapitel 5.5 komprimerar algoritmen bättre än V.42bis protokollet. Så långt uppfylls hypoteserna i kapitel 3 bra. Dock är den stora nackdelen med algoritmen komprimeringshastigheten. Den tid det tar att komprimera en viss mängd data är för läng för att algoritmen ska kunna förse modemet med komprimerad data i samma takt som modemet skickar iväg datan. Ett mål var att komprimeringen skulle kunna göras i realtid, dvs att komprimeringshastigheten minst skulle motsvara överföringshastigheten hos de snabbaste modemen, vilket i nuläget innebär 56 kbps. LZSS-delen är mycket snabb och skulle utan problem kunna klara att komprimera data i realtid för att tillgodose modemet. LZSS-delen genererar utdata med en hastighet motsvarande ca 260 kbps, vilket ger en trygg marginal till de 56 kbps som krävs för modemet. ASCII-delen är däremot mycket långsam och skulle troligen inte klara kraven om den inte körs på en mycket snabb dator.

Komprimeringshastigheten hos ASCII-delen är mycket beroende av hur datan ser ut. ”Text”-filen innehåller mycket få taggar och därför kommer nästan alla tecken att skickas vidare till LZSS-delen utan att behöva passera varken jämförelsen med tabellen eller mellanslagskomprimeringen. Detta innebär att komprimeringen går någorlunda snabbt. På en SUN Sparc Station 4 genereras utdata med motsvarande 130 kbps, vilket är tillräckligt för att förse modemet med data i realtid och ger tid över till att utföra LZSS-delen. Om däremot ”Länkar”-filen komprimeras kommer nästan varenda textrad att vara en tagg som måste jämföras med tabellen och genomgå mellanslagskomprimeringen. Dessutom innehåller taggarna mycket ofta en adress till någon sida på nätet (det är ju länksidor så den enda texten på sidorna är den förklarande texten till länkarna). I dessa adresser är det bara strängen ”http://www.” som tilldelas koder. För alla andra bokstäver i adressen kommer ingen match att hittas, men ändå kommer hela tabellen med strängar att gås igenom för varje bokstav och hela

arrayen med taggen måste skiftas för varje bokstav som tas bort. Detta tar oerhört lång tid och gör att komprimeringstiden blir mycket lång. På samma maskin som ovan genererades utdata med en hastighet motsvarande 9660 bps, vilket inte ens kommer i närheten av att vara tillräckligt.

Den genomsnittliga komprimeringshastigheten för alla olika typer av HTML-dokument motsvarar 25500 bps, och det är altså nödvändigt att snabba upp algoritmen om den över huvud taget ska kunna användas. Även om den genomsnittliga hastigheten hade legat över 56 kbps är det den lägsta hastigheten som måste tas hänsyn till. Anledningarna till varför algoritmen är så långsam är att koden inte är optimerad på något sätt. Till exempel läser och skriver algoritmen på disken oerhört ofta, så det är möjligt att prestandan skulle kunna ökas avsevärt genom att buffra datan och läsa och skriva större datastycken i taget. Störst tidsvinst borde kunna uppnås om hanteringen av taggarna snabbas upp. Detta då det är just dokument med många taggar som tar längst tid att komprimera.

Då algoritmen verkar fungera bra kan det vara lämpligt att göra en dekomprimeringsalgoritm till den. Dekomprimeringen är mycket enklare och framför allt snabbare än komprimeringen. Både ASCII-algoritmen och LZSS-algoritmen dekomprimerar data snabbare än de komprimerar, och den kumulativa effekten blir ganska stor. En fil som tar fem minuter att komprimera tar inte ens 30 sekunder att dekomprimera. Dekomprimeringsalgoritmen för LZSS fanns redan implementerad. Den fungerar genom att läsa in de två referenstecknen och hitta rätt ord i trädet, och om den flagga som talar om att nästa sträng är okomprimerad dyker upp läser den in det ordet och sätter in det i trädet. Dekomprimeringen för den ASCII-komprimerade datan är också mycket enkel. Först läses tabell2.dat in till en teckenarray så att algoritmen vet vilka strängar som motsvaras av koderna 128-255, och sedan läses filen in tecken för tecken. Är det inlästa tecknet ett tal mellan 0-127 är det ett okomprimerat ASCII-tecken och skickas vidare till utfilen. Är tecknet större än 127 är det en sträng i teckenarrayen och den sträng som sitter på motsvarande plats i 'teckenarray' skrivs till utfilen. Det verkar inte som om det skulle vara några problem med att dekomprimera data i realtid med nuvarande implementation.

6 Sammanfattning

6.1 Slutsats

Det är nu dags att sammanfatta vilka resultat som framkommit i denna rapport, samt hur väl dessa resultat stämmer överens med de mål som sattes upp i problembeskrivningen. Först kommer en kort sammanfattning av problembeskrivningen och sedan tittar vi på hur de olika punkterna uppfyllts i rapporten.

6.1.1 Problembeskrivningen

De teorier som framfördes i problembeskrivningen kan kort sammanfattas till följande punkter:

1. Taggarna i HTML-dokument borde kunna utnyttjas till att göra en algoritm som komprimerar dokumenten mera effektivt än en generell algoritm.
2. Algoritmen komprimerar HTML-dokument mera effektivt än den algoritm som finns i modem, och därför borde denna algoritm användas istället för att få ner överföringstiderna vid dataöverföring över telenätet.

För att testa dessa teorier sattes följande mål upp:

1. En algoritm ska väljas och implementeras. Algoritmen ska ta hänsyn till taggar i HTML-dokumentet och på detta sätt uppnå en högre komprimeringsgrad än en generell algoritm.
2. Ett antal HTML-dokument ska komprimeras med algoritmen för att bekräfta ovanstående punkt. Dokumenten ska representera alla typer av HTML-dokument som kan finnas på nätet för att få en jämförelse om dokument med mycket taggar komprimeras bättre än vanliga textdokument.
3. En jämförelse ska göras mellan algoritmen som tar hänsyn till taggarna och de algoritmer som finns i moderna modem. Jämförelser bör göras både vad avser komprimeringseffektivitet och komprimeringshastighet. Den implementerade algoritmen måste generera utdata minst lika snabbt som modemet arbetar.

Det förväntade resultatet av ovanstående punkter var dels en komprimeringsalgoritm med hög komprimeringseffektivitet på HTML-dokument, dels en jämförelse mellan denna algoritm och de algoritmer som finns i modem som visar att den implementerade algoritmen är bättre främst vad avser komprimeringseffektivitet men också avseende komprimeringshastighet.

6.1.2 Resultat

Metoden att använda en modifierad Huffmanalgoritm följd av en LZ-algoritm (LZSS) verkade lovande då den var förhållandevis enkel att implementera och borde enligt [Gus98] kunna ge ett bra resultat. Efter implementation i kapitel 5.1.2 visade sig dock metoden med ett träd som enbart förgrenar sig åt höger vara ett misslyckande, så en annan liknande metod valdes i kapitel 5.4. Denna andra metod, kallad ASCII-komprimering då den utnyttjar överblivna kombinationer hos ASCII-koden, gav ett bättre resultat än LZSS-algoritmen.

6 Sammanfattning

För att undersöka om algoritmen komprimerar lika bra som den algoritm som sitter i modem utfördes ett test i kapitel 5.5. Av resultatet i testet kan slutsatsen dras att ASCII-algoritmen komprimerar bättre än både MNP-5 och V.42bis.

Det sista målet var att komprimeringshastigheten minst skulle motsvara överföringshastigheten hos de snabbaste modemen. Detta mål har tyvärr inte kunnat uppfyllas. Komprimeringen av mellanslag och taggar i ASCII-komprimeringen går oerhört långsamt och kräver en rejäl dator för att komma upp i de 56 kbps som krävs. Med en del förbättringar i implementationen borde även detta mål kunna uppfyllas.

6.2 Diskussion

Motiveringen till detta arbete är att det finns ett behov av snabbare överföring av data över telenätet. I undersökningen av [GaTech] menar en majoritet av alla internetanvändare att just hastigheten är det största problemet med internet idag.

Problemet ligger inte bara i att telenätet är för långsamt. Även om man använder ett 28.8 kbps modem är det inte alls säkert att man alltid kommer upp i den hastigheten. Det kan mycket väl vara så att trafiken på nätet är så stor att hastighetsbegränsningen uppstår i den server man är uppkopplad mot. I framtiden när modemen klarar högre hastigheter och allt fler personer skaffar uppkoppling är det troligt att just detta problem växer.

Problemet har två lösningar. Antingen bygger man ut alla nät i takt med att internetanvändningen ökar eller så ser man till att skaffa bättre komprimeringsalgoritmer som gör att hastigheten kan ökas genom att mindre data skickas. Det råder ingen tvekan om att den senare lösningen är den klart billigaste och enklaste att genomföra. Det finns därför ett behov av att undersöka vilka möjligheter som finns för att bättre komprimera datan, och denna rapport är ett försök att göra just detta.

Det test med modem som genomfördes visar att det bästa sättet att bättra på komprimeringen är att utföra den med mjukvara i datorn istället för i modemet. Detta bekräftas i en undersökning av [Diac]. I undersökningen skickades några olika bild- och textfiler okomprimerade med ett modem med V.42bis protokollet, vilket ger den bästa komprimeringen av de båda olika protokollen. Samma filer komprimerades sedan med ARC, vilket är ett komprimeringsprogram för Amiga, innan de skickades. Det visade sig då att de mjukvarukomprimerade filerna kunde överföras med i genomsnitt 30% högre hastighet än när de var okomprimerade.

I ett annat exempel jämförs överföringshastigheten av en bild i TIFF-format. Filen kan laddas ner på 56 sekunder med ett V.32/V.42bis modem, vilket motsvarar en överföringshastighet på ungefär 30 kbps. Detta låter kanske inte så illa med tanke på att ett V.32 modem arbetar med 14,4 kbps, men om filen komprimerats med t ex Stuffit för Macintosh hade överföringen tagit mindre än 10 sekunder!

Dessa båda exempel visar tydligt att komprimering med mjukvara är klart överlägsen V.42bis-protokollet i modemen och därmed borde de också vara bättre än MNP-5 protokollet. Det är dock viktigt att påpeka att den komprimeringsalgoritm som beskrivs i denna rapport inte kan liknas vid de algoritmer som använts i testerna av [Diac]. I testerna nämns ZIP som en komprimeringsalgoritm som ger bättre komprimering än V.42bis. ASCII-algoritmen ger dock inte lika bra komprimering som ZIP på någon av de filer som finns med i testerna i tidigare kapitel. Jämförelsen mellan ASCII-algoritmen och ZIP är dock inte helt rättvis, då ASCII-algoritmen mycket väl

skulle kunna göra bättre ifrån sig om mera arbete lades ner på den. Det är möjligt att om LZSS-algoritmen i ASCII-komprimeringen byts ut mot samma algoritm som används i ZIP så skulle ASCII-komprimeringen vara bättre. Värt att påpeka är också att det inte är så att mjukvara alltid är bättre än hårdvara, utan att de implementerade algoritmerna är bättre än de som sitter i modemens hårdvara just nu. Om modemens hårdvara uppgraderas kan helt andra resultat uppkomma.

Anledningen till att mjukvarukomprimering inte används är enligt [Diac] att det är smidigare att komprimera automatiskt i modemet än att först komprimera datan 'för hand' med något komprimeringsprogram, sedan skicka den till modemet, och slutligen dekomprimera manuellt.

Enligt [Diac] är en av de främsta anledningarna till att komprimeringen sker i modemen istället för med mjukvara i datorn att det blir mycket smidigare så. Användaren slipper att själv komprimera datan innan den skickas och att sedan dekomprimera den för att kunna titta på den. Om användaren ligger uppkopplad på telenätet under hela den tid det tar att själv dekomprimera datan och titta på den kommer uppkopplingstiden troligen att bli minst lika lång som om modemet hade skött allt arbete, även om det skulle innebära att själva överföringstiden blir längre. Anledningen till detta är att det tar en liten stund att först starta dekomprimeringsprogrammet, sedan tala om för detta program vilken fil som ska dekomprimeras och vart man vill ha den dekomprimerade filen, och slutligen starta något program för att titta på filen. Små filer som bara tar några sekunder att överföra skulle då ta kortare tid om allt sköttes automatiskt. Särskilt jobbigt blir det då med just HTML-dokument, eftersom dessa ofta inte är större än ett par tusen tecken. Ofta surfar man också mellan så många sidor att det vore oerhört frustrerande om man var tvungen att sitta och dekomprimera HTML-koden till varenda sida man besökte.

Om däremot ett komprimeringsprogram jobbade transparent mot det webbläsarprogram man använder skulle allt kunna skötas av mjukvaran utan att användaren skulle märka något. Precis som att webbläsaren kan dekomprimera en JPEG-bild och visa den skulle den kunna dekomprimera HTML-koden automatiskt. Allra smidigast vore det om HTML-dokumenten låg komprimerade på ägarens hårddisk. Det skulle dels spara in hårddiskutrymme och dessutom skulle inte ägarens dator belastas av att den måste komprimera all data som skickas ut. För en stor server vore det kanske till och med omöjligt för processorn att hinna med att komprimera all data som ska skickas ut om den inte redan låg komprimerad på hårddisken. Man skulla kunna tänka sig att när ett dokument ska editeras öppnar man det bara med sin webbläsare som då dekomprimerar filen automatiskt. När filen editerats klart och sparas kan webbläsaren komprimera filen samtidigt som den sparas.

6.3 Fortsatt arbete

Det finns mycket kvar att göra och många tester som vore intressanta att göra. Det vore intressant göra en mera exakt jämförelse mellan ASCII-algoritmen och V.42bis protokollet. MNP-5 protokollet är egentligen inte nödvändigt att testa eftersom detta komprimerar sämre än V.42bis. En jämförelse borde göras med en implementation av V.42bis-protokollet för att få mera tillförlitliga siffror på komprimeringseffektiviteten.

Ett större projekt är att försöka få ordning på själva algoritmen. Det viktigaste är att hastigheten ökas. Detta kan säkert åstakommas på flera sätt, men vad jag kan se borde störst vinster kunna göras genom att buffra data i minnet istället för att läsa och skriva

på disken hela tiden och genom att göra hanteringen av taggarna mera effektiv vad gäller skiftning av arrayen och förlustkomprimeringen av mellanslagen. Det vore också intressant att se om det går fortare om tabellen utökas så att alla kombinationer av HTML-koder och mellanslag finns representerade där. Mellanslagskomprimeringen skulle då kunna skötas i samma steg som konverteringen av HTML-koderna till stora bokstäver. Det är också möjligt att ännu fler mönster i taggarna kan utnyttjas. Till exempel verkar det som om färgangivelser alltid börjar med ="# och inte bara med =" som det ser ut nu.

En fortsättning på detta arbete vore alltså att utveckla en programvara som samarbetar med webbläsaren och komprimerar/dekomprimerar all data som skickas till och från modemet. Det är viktigt att den algoritm som används kan både komprimera och dekomprimera alla typer av data i realtid, dvs att utdatan från algoritmen kan skickas till modemet med minst samma hastighet som modemet kan skicka data över telenätet. Det är också ett krav att algoritmen kan upptäcka om datan inte komprimeras så att komprimeringen då kan avbrytas och datan skickas okomprimerad.

Kravet på programvaran är att den ligger integrerad i webbläsaren och jobbar helt transparent. Användaren ska över huvud taget inte behöva veta om att den finns där. Ett annat krav är naturligtvis att alla webbläsare börjar använda samma algoritm, annars kommer vissa sidor bara att kunna läsas med en viss webbläsare.

Referenser

[Wil93] Ross N. Williams (1993), Adaptive Data Compression, Kluwer Academic Publishers, Andra upplagan.

[Hal95] Fred Halsall (1995), Data Communications, Computer Networks and Open Systems, Addison-Wesley Publishers Ltd. Och Addison-Wesley Publishing Company Inc., Fjärde upplagan.

[BS95] C. Wayne Brown och Barry J. Shepherd (1995), Graphics File Formats, reference and guide, Manning Publications Co.

[BCW90] Timothy C. Bell, John G. Cleary och Ian H. Witten (1990), Text Compression, Prentice-Hall, Inc.

[Sto92] Editerad av James A. Storer (1992), Image and Text Compression, Kluwer Academic Publishers.

[Bin88] John A. C. Bingham (1988), The theory and practice of modem design, John Wiley & Sons, Inc.

[GaTech] Georgia Institute of Technology, Datum: 1998-02-20, Adress: http://www.gvu.gatech.edu/user_surveys/

[Sal97] David Salomon (1997), Data Compression, the complete reference, Springer-Verlag New York Inc.

[Diac] Denver Internet Access Corp., Datum 1998-04-15, Adress: http://www.diac.com/DOCS/MT-Compression.html

[Gus98] Muntlig kommunikation med Henrik Gustavsson, handledare vid datainstitutionen Högskolan i Skövde.

Bilaga 1 – Http-adresser

Text 1: http://www.riksdagen.se/debatt/9798/utskott/ku/ku4/ku400003.asp
Text 2: http://anacin.nsc.vcu.edu/~jim/mvax/mvax_faq_text.html
Text 3: http://www.his.se/neverlost/schema2.idc
Text 4: http://anubis.science.unitn.it/services/perlman/perlxs.html

Bilder 1: http://threshold.premierweb.com/realm/gallery/
Bilder 2: http://isar-art.com/c_vraq_d.htm
Bilder 3: http://www.erinet.com/jelane/families/allthumb.html
Bilder 4: http://hobbes.jct.ac.il/~naiman/c++-oop/thumbs.html

Länkar 1: http://www.jmk.su.se/dig/guide/digbookm.html
Länkar 2: http://www.sics.se/isl/sicstus/sicstus_toc.html
Länkar 3: http://www.flashback.se/archive/zines.html
Länkar 4: http://albie.wcupa.edu/schmus.mue/515/menu6.htm

Blandat 1: http://www.his.se/neverlost/schema.idc
Blandat 2: http://www.mp3.com
Blandat 3: http://www.ara-animation.com/
Blandat 4: http://www.unitedmedia.com/comics/dilbert/

Bilaga 2 – Huffmankod

```
# include <ctype.h>
# include <stdio.h>
# include <string.h>

unsigned char nolltecken=0;
char temparr[255], temptemparr[255];
char *teckenarray[256];
int nummerarray[256];
int position=0;

int sort()
{
     int i,j,temp,arr[256];
     for(i=0;i<256;i++)
          arr[i]=i;
     for(i=0;i<255;i++) {
          for(j=0;j<254;j++) {
               if(nummerarray[j]<nummerarray[j+1]) {
                    temp=nummerarray[j];
                    nummerarray[j]=nummerarray[j+1];
                    nummerarray[j+1]=temp;
                    temp=arr[j];
                    arr[j]=arr[j+1];
                    arr[j+1]=temp;
               }
          }
     }
     for(i=0;i<255;i++)
          nummerarray[arr[i]]=i;
}

int space_comp(FILE *utfil, int storlek)
{
     int i;
     long steg;
     char utarr[3];
     if(temparr[0]==' ' && temparr[1]=='=') {
          utarr[0]='=';
          if(temparr[2]==' ') {
               fprintf(utfil,"%c",utarr[0]);
               strcpy(temptemparr,temparr);
               for(i=0;i<255;i++)
                    temparr[i]=temptemparr[i+3];
               storlek=storlek-3;
          }
          else {
               fprintf(utfil,"%c",utarr[0]);
               strcpy(temptemparr, temparr);
               for(i=0;i<255;i++)
                    temparr[i]=temptemparr[i+2];
               storlek=storlek-2;
          }
     }
     else if(temparr[0]=='=' && temparr[1]==' ') {
          utarr[0]='=';
          fprintf(utfil,"%c",utarr[0]);
          strcpy(temptemparr, temparr);
          for(i=0;i<255;i++)
               temparr[i]=temptemparr[i+2];
          storlek=storlek-2;
     }
     return storlek;
```

Bilaga 2 – Huffmankod

```
}

int jamfor(FILE *utfil)
{
     int samma=0, j, n;
     unsigned char tecken;
     char arr[255], arr_temp[255], temparr_temp[255];
     strcpy(arr, temparr);
     for(j=0;j<12;j++) {
          tecken=arr[j];
          tecken=toupper(tecken);
          arr[j]=tecken;
     }
     j=128;
     while(!samma && j<255) {
          n=strlen(teckenarray[j]);
          strcpy(arr_temp, arr);
          arr_temp[n]=0;
          if(!strcmp(teckenarray[j], arr_temp))
               samma=1;
          else
               j++;
     }
     if(samma==1) {
          nummerarray[j]++;
          for(j=0;j<n;j++)
               fprintf(utfil,"%c",arr[j]);
     }
     else {
          j=128;
          while(!samma && j<255) {
               n=strlen(teckenarray[j]);
               strcpy(temparr_temp, temparr);
               temparr_temp[n]=0;
               if(!strcmp(teckenarray[j], temparr_temp))
                    samma=1;
               else
                    j++;
          }
          if(samma==1) {
               nummerarray[j]++;
               for(j=0;j<n;j++)
                    fprintf(utfil,"%c",temparr[j]);
          }

          else {
               n=1;
               tecken=temparr[0];
               nummerarray[tecken]++;
               fprintf(utfil,"%c",temparr[0]);
          }
     }
     return n;
}

int skriv_nollor(FILE *utfil, int nollor)
{
     int i, bytes;
     unsigned char temptecken;
     if (nollor>(8+(7-position))) {
          nollor=nollor-(8-position);
          fprintf(utfil,"%c",nolltecken);
          bytes=nollor/8;
          nolltecken=0;
          for (i=0; i<bytes; i++)
```

```
                fprintf(utfil,"%c",nolltecken);
            nollor=nollor-(8*bytes);
            nolltecken=128;
            nolltecken=nolltecken >> nollor;
            position=nollor+1;
        }
        else if (nollor<(8-position)) {
            temptecken=128;
            temptecken=temptecken >> (position+nollor);
            nolltecken=nolltecken|temptecken;
            position=position+nollor+1;
        }
        else {
            fprintf(utfil,"%c",nolltecken);
            nollor=nollor-(8-position);
            nolltecken=128;
            nolltecken=nolltecken >> nollor;
            position=nollor+1;
        }

        if (position==8) {
            fprintf(utfil,"%c",nolltecken);
            nolltecken=0;
            position=0;
        }
}

int hitta nollor(FILE *utfil)
{
        int samma=0, j=128, n;
        unsigned char tecken;
        char temparr_temp[255];
        while(!samma && j<255) {
            n=strlen(teckenarray[j]);
            strcpy(temparr_temp, temparr);
            temparr_temp[n]=0;
            if(!strcmp(teckenarray[j], temparr_temp))
                samma=1;
            else
                j++;
        }
        if(samma==1)
            skriv_nollor(utfil, nummerarray[j]);
        else {
            n=1;
            tecken=temparr[0];
            skriv_nollor(utfil, nummerarray[tecken]);
        }
        return n;
}

int huffman(FILE *infil, FILE *utfil)
{
        int i,j,sammastorlek;
        temparr[0]='<';
        temparr[1]=0;
        for(i=2;i<150;i++) {
            temptemparr[0]=getc(infil);
            temptemparr[1]=0;
            strcat(temparr, temptemparr);
            if(temptemparr[0]=='>') {
                while(i>0) {
                    sammastorlek=hitta_nollor(utfil);
                    i-=sammastorlek;
                    strcpy(temptemparr,temparr);
```

```
                    for(j=0;j<150;j++)
                         temparr[j]=temptemparr[j+sammastorlek];
               }
               return 0;
          }
     }
     return 0;
}

int istag(FILE *infil, FILE *utfil)
{
     int i,j,sammastorlek;
     temparr[0]='<';
     temparr[1]=0;
     for(i=2;i<150;i++) {
          temptemparr[0]=getc(infil);
          temptemparr[1]=0;
          strcat(temparr, temptemparr);
          if(temptemparr[0]=='>') {
               while(i>0) {
                    sammastorlek=jamfor(utfil);
                    i-=sammastorlek;
                    strcpy(temptemparr,temparr);
                    for(j=0;j<150;j++)
                         temparr[j]=temptemparr[j+sammastorlek];
                    if(sammastorlek>1)
                         i=space_comp(utfil, i);
               }
               return 0;
          }
     }
     return 0;
}

int main()
{
     FILE *datafil,*infil,*utfil;
     int i,j,k;
     char temptecken,infilnamn[12];
     unsigned char kladd;
     for(i=0;i<256;i++)
          nummerarray[i]=0;
     datafil=fopen("tabell.dat","r");
     for(i=128;i<255;i++) {
          j=0;
          temparr[0]=0;
          temptemparr[0]=getc(datafil);
          temptemparr[1]=0;
          while(temptemparr[0]!='\n') {
               j++;
               strcat(temparr, temptemparr);
               temptemparr[0]=getc(datafil);
          }
          teckenarray[i]=(char *)malloc(sizeof(char)*j);
          for(k=0;k<=j;k++)
               teckenarray[i][k]=temparr[k];
     }
     fclose(datafil);
     printf("\nAnge fil att räkna : ");
     scanf("%s",infilnamn);
     infil=fopen(infilnamn,"r");
     utfil=fopen("workfile","w");
     temptecken=getc(infil);
     while(!feof(infil)) {
          if (temptecken=='<')
```

Bilaga 2 – Huffmankod

```
            istaq(infil,utfil);
        else {
            kladd=temptecken;
            nummerarray[kladd]++;
            fprintf(utfil,"%c",temptecken);
        }
        temptecken=getc(infil);
    }
    fclose(infil);
    fclose(utfil);
    sort();
    infil=fopen("workfile","r");
    utfil=fopen("nollor.dat","w");
    datafil=fopen("tabell2.dat","r");
    for(i=128;i<255;i++) {
        j=0;
        temparr[0]='\0';
        temptemparr[0]=getc(datafil);
        temptemparr[1]=0;
        while(temptemparr[0]!='\n') {
            j++;
            strcat(temparr, temptemparr);
            temptemparr[0]=getc(datafil);
        }
        free(teckenarray[i]);
        teckenarray[i]=(char *)malloc(sizeof(char)*j);
        for(k=0;k<=j;k++)
            teckenarray[i][k]=temparr[k];
    }
    fclose(datafil);
    temptecken=getc(infil);
    while(!feof(infil)) {
        if (temptecken=='<')
            huffman(infil,utfil);
        else {
            kladd=temptecken;
            skriv_nollor(utfil,nummerarray[kladd]);
        }
        temptecken=getc(infil);
    }
    fprintf(utfil,"%c",nolltecken);
    fclose(infil);
    fclose(utfil);
    for(i=0;i<256;i++)
        free(teckenarray[i]);
    return 0;
}
```

Bilaga 3 – tabell.dat

```
</A>
</B>
</BODY>
</CENTER>
</CODE>
</DL>
</EM>
</FONT>
</FORM>
</H1>
</H2>
</H3>
</HEAD>
</HTML>
</I>
</P>
</PRE>
</SELECT>
</STRONG>
</TABLE>
</TD>
</TH>
</TITLE>
</TR>
</UL>
</div>
</SMALL>
</
<AREA
<AT>
<A•
<B>
<BODY•
<BR>
<CENTER>
<CODE>
<DD>
<DL>
<DT>
<EM>
<FONT•
<FORM•
<H1>
<H2>
<H3•
<HEAD>
<HR>
<HR•
<HTML>
<I>
<IMG•
<INPUT•
<LI>
<LINK
<META•
<OPTION•
<P>
<PRE>
<P•
<SELECT•
<STRONG>
<TABLE•
<TD>
<TD•
<TH•
<TITLE>
<TR>
<TR•
<UL>
<DIV•
<SMALL>
ACTION
ADD DATE
ALIGN
ALINK
ALT
BACKGROUND
BGCOLOR
BORDER
CELLPADDING
CELLSPACING
COLOR
COLSPAN
CONTENT
COORDS
FACE
FOLDED
HEIGHT
HREF
HSPACE
HTTP-EQUIV
LAST VISIT
LINK
METHOD
NAME
NOWRAP
OnMouseOver
REL
ROWSPAN
SELECTED
SHAPE
SIZE
SRC
TARGET
TEXT
TYPE
VALIGN
VALUE
VLINK
VSPACE
WIDTH
nobr
http://
www.
MARGIN
LEFT
RIGHT
.jpg
.html
.gif
ABSMIDDLE
CENTER
TOP
MIDDLE
">
cgi
<>
```

Bilaga 4 – tabell2.dat

```
</A>
</B>
</BODY>
</CENTER>
</CODE>
</DL>
</EM>
</FONT>
</FORM>
</H1>
</H2>
</H3>
</HEAD>
</HTML>
</I>
</P>
</PRE>
</SELECT>
</STRONG>
</TABLE>
</TD>
</TH>
</TITLE>
</TR>
</UL>
</DIV>
</SMALL>
</
<AREA
<AT>
<A•
<B>
<BODY•
<BR>
<CENTER>
<CODE>
<DD>
<DL>
<DT>
<EM>
<FONT•
<FORM•
<H1>
<H2>
<H3•
<HEAD>
<HR>
<HR•
<HTML>
<I>
<IMG•
<INPUT•
<LI>
<LINK
<META•
<OPTION•
<P>
<PRE>
<P•
<SELECT•
<STRONG>
<TABLE•
<TD>
<TD•
<TH•
<TITLE>
<TR>
<TR•
<UL>
<DIV•
<SMALL>
ACTION=
ADD DATE=
ALIGN=
ALINK=
ALT=
BACKGROUND=
BGCOLOR=
BORDER=
CELLPADDING=
CELLSPACING=
COLOR=
COLSPAN=
CONTENT=
COORDS=
FACE=
FOLDED
HEIGHT=
HREF="
HSPACE=
HTTP-EQUIV=
LAST VISIT=
LINK=
METHOD=
NAME=
NOWRAP
OnMouseOver=
REL=
ROWSPAN=
SELECTED
SHAPE=
SIZE=
SRC="
TARGET=
TEXT=
TYPE=
VALIGN=
VALUE=
VLINK=
VSPACE=
WIDTH=
NOBR
http://
www.
MARGIN
LEFT
RIGHT
.jpg
.html
.gif
ABSMIDDLE
CENTER
TOP
MIDDLE
">
cgi
<>
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
# include <ctype.h>
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

#define N                4096
#define F                  18
#define THRESHOLD       2
#define NIL N

unsigned long int textsize = 0;
unsigned char text_buf[N+F-1];
int match_position,match_length,lson[N+1],rson[N+257],dad[N+1];
char temptecken;
unsigned char nolltecken=0;
char temparr[255], temptemparr[255];
char *teckenarray[256];
int nummerarray[256];

void InitTree(void)
{
     int  i;
     for (i=N+1;i<=N+256;i++) rson[i] = NIL;
     for (i=0;i<N;i++) dad[i] = NIL;
}

void InsertNode(int r)
{
     int  i, p, cmp;
     unsigned char  *key;

     cmp=1;
     key=&text_buf[r];
     p=N+1+key[0];
     rson[r]=lson[r]=NIL;
     match_length=0;
     for ( ; ; ) {
          if (cmp>=0) {
               if (rson[p]!=NIL)
                    p=rson[p];
               else {
                    rson[p]=r;
                    dad[r]=p;
                    return;
               }
          }else   {
                    if (lson[p]!=NIL) p=lson[p];
                    else {
                         lson[p]=r;
                         dad[r]=p;
                         return;
                    }
               }
          for (i=1;i<F;i++)
               if ((cmp=key[i]-text_buf[p+i])!=0)  break;
          if (i>match_length){
               match_position=p;
               if((match_length=i)>=F) break;
          }
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
     }
     dad[r]=dad[p];
     lson[r]=lson[p];
     rson[r]=rson[p];
     dad[lson[p]]=r;
     dad[rson[p]]=r;
     if (rson[dad[p]] == p) rson[dad[p]] = r;
     else                   lson[dad[p]] = r;
     dad[p] = NIL;
}

void DeleteNode(int p)
{
     int  q;
     if (dad[p]==NIL) return;
     if (rson[p]==NIL) q=lson[p];
     else if (lson[p]==NIL) q=rson[p];
     else {
          q = lson[p];
          if (rson[q] != NIL) {
               do {  q = rson[q];  } while (rson[q] != NIL);
                    rson[dad[q]]=lson[q];
                    dad[lson[q]]=dad[q];
                    lson[q]=lson[p];
                    dad[lson[p]]=q;
          }
          rson[q]=rson[p];
          dad[rson[p]]=q;
     }
     dad[q]=dad[p];
     if (rson[dad[p]] == p) rson[dad[p]] = q;  else lson[dad[p]] = q;
     dad[p] = NIL;
}

int Encode(FILE *infile, FILE *outfile)
{
     int  i, c, len, r, s, last_match_length, code_buf_ptr;
     unsigned char  code_buf[17], mask;

     InitTree();
     code_buf[0] = 0;
     code_buf_ptr = mask = 1;
     s = 0;
     r = N - F;
     for (i = s; i < r; i++) text_buf[i] = ' ';
     for (len = 0; len < F && (c = getc(infile)) != EOF; len++)
          text_buf[r + len] = c;
     if ((textsize = len) == 0) return;
     for (i = 1; i <= F; i++) InsertNode(r - i);
     InsertNode(r);
     do {
          if (match_length > len) match_length = len;
          if (match_length<=THRESHOLD) {
               match_length=1;
               code_buf[0]|=mask;
               code_buf[code_buf_ptr++]=text_buf[r];
          } else {
              code_buf[code_buf_ptr++]=(unsigned char)match_position;
              code_buf[code_buf_ptr++]=(unsigned char)
                      (((match_position >> 4) & 0xf0)
               | (match_length - (THRESHOLD + 1)));
          }
          if ((mask <<= 1) == 0) {
               for (i = 0; i < code_buf_ptr; i++)
                    putc(code_buf[i], outfile);
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
                code_buf[0] = 0;  code_buf_ptr = mask = 1;
            }
            last_match_length = match_length;
            for (i = 0; i < last_match_length &&
                    (c = getc(infile)) != EOF; i++) {
                DeleteNode(s);
                text_buf[s] = c;
                if (s < F - 1) text_buf[s + N] = c;
                s=(s+1)&(N-1);
                r=(r+1)&(N - 1);
                InsertNode(r);
            }
            while (i++ < last_match_length) {
                DeleteNode(s);
                s = (s + 1) & (N - 1);  r = (r + 1) & (N - 1);
                if (--len) InsertNode(r);
            }
        } while (len > 0);
        if (code_buf_ptr > 1) {
            for (i=0;i<code_buf_ptr;i++)
                putc(code_buf[i], outfile);
        }
        return 0;
}

int Decode(FILE *infile, FILE *outfile)
{
        int  i, j, k, r, c;
        unsigned int  flags;
        for (i = 0; i < N - F; i++) text_buf[i] = ' ';
        r=N-F;
        flags=0;
        for ( ; ; ) {
            if (((flags >>= 1) & 256) == 0) {
                if ((c = getc(infile)) == EOF) break;
                flags = c | 0xff00;
            }
            if (flags & 1) {
                if ((c = getc(infile)) == EOF) break;
                putc(c, outfile);
                text_buf[r++]=c;
                r&=(N - 1);
            } else {
                if ((i = getc(infile)) == EOF) break;
                if ((j = getc(infile)) == EOF) break;
                i |= ((j & 0xf0) << 4);  j = (j & 0x0f) + THRESHOLD;
                for (k = 0; k <= j; k++) {
                    c = text_buf[(i + k) & (N - 1)];
                    putc(c, outfile);
                    text_buf[r++] = c;
                    r&=(N-1);
                }
            }
        }
        return 0;
}

int space_comp(FILE *utfil, int storlek)
{
        int i;
        long steg;
        char utarr[2];

        if(temparr[0]==' ' && temparr[1]=='=') {
            utarr[0]='=';
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
          if(temparr[2]==' ') {
               fprintf(utfil,"%c",utarr[0]);
               strcpy(temptemparr,temparr);
               for(i=0;i<255;i++)
                    temparr[i]=temptemparr[i+3];
               storlek=storlek-3;
          }
          else {
               fprintf(utfil,"%c",utarr[0]);
               strcpy(temptemparr, temparr);
               for(i=0;i<255;i++)
                    temparr[i]=temptemparr[i+2];
               storlek=storlek-2;
          }
     }
     else if(temparr[0]=='=' && temparr[1]==' ') {
          utarr[0]='=';
          fprintf(utfil,"%c",utarr[0]);
          strcpy(temptemparr, temparr);
          for(i=0;i<255;i++)
               temparr[i]=temptemparr[i+2];
          storlek=storlek-2;
     }
     return storlek;
}

int jamfor(FILE *utfil)
{
     int samma=0, j, n;
     unsigned char tecken;
     char arr[255], char arr_temp[255], temparr_temp[255];

     strcpy(arr, temparr);
     for(j=0;j<12;j++) {
          tecken=arr[j];
          tecken=toupper(tecken);
          arr[j]=tecken;
     }
     j=128;
     while(!samma && j<255) {
          n=strlen(teckenarray[j]);
          strcpy(arr_temp, arr);
          arr_temp[n]=0;
          if(!strcmp(teckenarray[j], arr_temp))
               samma=1;
          else
               j++;
     }
     if(samma==1) {
          for(j=0;j<n;j++)
               fprintf(utfil,"%c",arr[j]);
     }
     else {
          j=128;
          while(!samma && j<255) {
               n=strlen(teckenarray[j]);
               strcpy(temparr_temp, temparr);
               temparr_temp[n]=0;
               if(!strcmp(teckenarray[j], temparr_temp))
                    samma=1;
               else
                    j++;
          }
          if(samma==1) {
               for(j=0;j<n;j++)
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
                    fprintf(utfil,"%c",temparr[j]);
          }
          else {
               n=1;
               fprintf(utfil,"%c",temparr[0]);
          }
     }
     return n;
}

int hitta_strang(FILE *utfil)
{
     int samma=0, n;
     unsigned char tecken, j=128;
     char temparr_temp[255];

     while(!samma && j<255) {
          n=strlen(teckenarray[j]);
          strcpy(temparr_temp, temparr);
          temparr_temp[n]=0;
          if(!strcmp(teckenarray[j], temparr_temp))
               samma=1;
          else
               j++;
     }
     if(samma==1)
          fprintf(utfil, "%c", j);
     else {
          n=1;
          tecken=temparr[0];
          fprintf(utfil, "%c", tecken);
     }
     return n;
}

int compress(FILE *infil, FILE *utfil)
{
     int i,j,sammastorlek;
     temparr[0]='<';
     temparr[1]=0;

     for(i=2;i<255;i++) {
          temptemparr[0]=getc(infil);
          temptemparr[1]=0;
          strcat(temparr, temptemparr);
          if(temptemparr[0]=='>') {
               while(i>0) {
                    sammastorlek=hitta_strang(utfil);
                    i-=sammastorlek;
                    strcpy(temptemparr,temparr);
                    for(j=0;j<(255-sammastorlek);j++)
                         temparr[j]=temptemparr[j+sammastorlek];
               }
               return 0;
          }
     }
     return 0;
}

int istag(FILE *infil, FILE *utfil)
{
     int i,j,sammastorlek;
     temparr[0]='<';
     temparr[1]=0;
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
     for(i=2;i<255;i++) {
          temptemparr[0]=getc(infil);
          temptemparr[1]=0;
          strcat(temparr, temptemparr);
          if(temptemparr[0]=='>') {
               while(i>0) {
                    sammastorlek=jamfor(utfil);
                    i-=sammastorlek;
                    strcpy(temptemparr,temparr);
                    for(j=0;j<(255-sammastorlek);j++)
                         temparr[j]=temptemparr[j+sammastorlek];
                    if(sammastorlek>1)
                         i=space_comp(utfil, i);
               }
               return 0;
          }
     }
     return 0;
}

int decompress(char infilnamn[])
{
     FILE *infil, *utfil, *datafil;
     int i, j, k;
     unsigned char tecken;

     datafil=fopen("tabell2.dat","r");
     for(i=128;i<255;i++) {
          j=0;
          temparr[0]='\0';
          temptemparr[0]=getc(datafil);
          temptemparr[1]=0;
          while(temptemparr[0]!='\n') {
               j++;
               strcat(temparr, temptemparr);
               temptemparr[0]=getc(datafil);
          }
          free(teckenarray[i]);
          teckenarray[i]=(char *)malloc(sizeof(char)*j);
          for(k=0;k<=j;k++)
               teckenarray[i][k]=temparr[k];
     }
     fclose(datafil);

     infil=fopen(infilnamn,"r");
     utfil=fopen("workfile","w");

     Decode(infil, utfil);

     fclose(infil);
     fclose(utfil);

     infil=fopen("workfile","r");
     utfil=fopen("decompressed.txt","w");

     tecken=getc(infil);
     while(!feof(infil)) {
          if(tecken<128)
               fprintf(utfil, "%c", tecken);
          else
               fprintf(utfil, "%s", teckenarray[tecken]);
          tecken=getc(infil);
     }

     fclose(infil);
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
     fclose(utfil);
     return 0;
}

int main()
{
     FILE *datafil,*infil,*utfil;
     int i,j,k;
     char infilnamn[12], val[2];

     printf("\nKomprimera/Dekomprimera (K/D) : ");
     scanf("%c",val);

     if(val[0]=='d' || val[0]=='D') {
          printf("\nAnge filnamn : ");
          scanf("%s",infilnamn);
          decompress(infilnamn);
     }

     else {
          for(i=0;i<256;i++)
               nummerarray[i]=0;
          datafil=fopen("tabell.dat","r");
          for(i=128;i<255;i++) {
               j=0;
               temparr[0]=0;
               temptemparr[0]=getc(datafil);
               temptemparr[1]=0;
               while(temptemparr[0]!='\n') {
                    j++;
                    strcat(temparr, temptemparr);
                    temptemparr[0]=getc(datafil);
               }
               teckenarray[i]=(char *)malloc(sizeof(char)*j);
               for(k=0;k<=j;k++)
                    teckenarray[i][k]=temparr[k];
          }
          fclose(datafil);

          printf("\nAnge filnamn : ");
          scanf("%s",infilnamn);
          infil=fopen(infilnamn,"r");
          utfil=fopen("steg1.temp","w");

          temptecken=getc(infil);
          while(!feof(infil)) {
               if (temptecken=='<')
                    istag(infil,utfil);
               else
                    fprintf(utfil,"%c",temptecken);
               temptecken=getc(infil);
          }

          fclose(infil);
          fclose(utfil);

          infil=fopen("steg1.temp","r");
          utfil=fopen("steg2.temp","w");
          datafil=fopen("tabell2.dat","r");

          for(i=128;i<255;i++) {
               j=0;
               temparr[0]=0;
               temptemparr[0]=getc(datafil);
               temptemparr[1]=0;
```

Bilaga 5 – Kod för ASCII-baserad tagg-komprimering med påföljande LZSS-komprimering

```
                while(temptemparr[0]!='\n') {
                     j++;
                     strcat(temparr, temptemparr);
                     temptemparr[0]=getc(datafil);
                }
                free(teckenarray[i]);
                teckenarray[i]=(char *)malloc(sizeof(char)*j);
                for(k=0;k<=j;k++)
                     teckenarray[i][k]=temparr[k];
           }
           fclose(datafil);

           temptecken=getc(infil);
           while(!feof(infil)) {
                if (temptecken=='<')
                     compress(infil,utfil);
                else
                     fprintf(utfil,"%c",temptecken);
                temptecken=getc(infil);
           }

           fclose(infil);
           fclose(utfil);

           for(i=0;i<256;i++)
                free(teckenarray[i]);

           infil=fopen("steg2.temp","r");
           utfil=fopen("compressed.file","w");

           Encode(infil, utfil);

           fclose(infil);
           fclose(utfil);
      }
      return 0;
}
```