Latvijas Universitāte
Fizikas un matemātikas fakultāte
Datorikas nodaļa

TEKSTISKAS INFORMĀCIJAS SASPIEŠANA AR HAFFMANA ALGORITMU

Bakalaura darbs

Autors
Jurijs Fjodorovs
Stud. apl. Nr. DatZ 020061

Vadītājs
Guntis Arnicāns
docents, Dr. sc. comp.

Rīga, 2006.

Anotācija

Šī bakalaura darba mērķis ir izpētīt jau eksistējošas informācijas saspiešanas metodes, kā arī, balstoties uz iegūto informāciju, izstrādāt savu arhivātoru, kurš pielietos vienkāršāku no tām – Haffmana algoritmu. Pētījuma sākumā detalizēti ir aplūkota viena no iespējamajām Haffmana algoritma realizācijas metodēm, pēc kuras autors izstrādājis savu arhivātoru JVC6. Darba turpinājumā ir aprakstītas arī citu saspiešanas metožu specifiskās īpašības, raksturīgās iezīmes, sniedzot nelielu ieskatu to realizācijām. Pētījuma noslēgumā ir veikta jaunizveidota arhivātora analīze un salīdzinājums ar mūsdienas populārākām saspiešanas programmām, nosakot katras stiprās puses, pozitīvās iezīmes. Veikta algoritmu salīdzināšana dažādiem tekstiskiem failiem. Darba rezultātā ir izstrādāta konkrēta saspiešanas programma – JFC6 arhivātors, kā arī piedāvāti iespējamie šīs programmas uzlabojumi tuvākajā nākotnē.

Summary

The main target of this bachelor's paper is to learn modern compression algorithms, to produce a compressor for textual data. It will implement the easiest and simplest algorithm – Huffman code for textual data compression. In the beginning of research is gone into a detail one of the possible methods of Huffman algorithm realization. After that an author produced his own compressor JVC6. In this work are also described specific characteristics of other compression methods, and their realization possibility. In the end of the research are done some analyses of new compression program, its comparison with modern, popular and wildly used programs-compressors. Positive and good features are finalized for each of them. Comparison of algorithms for some textual files is also done. During the work is written a concrete compression program JFC6 and are given some advices for its future improvements.

Аннотация

Целью данной работы является исследование уже существующих методов сжатия информации, а также, опираясь на полученную информацию, разработать свой архиватор, который будет основан на реализации одного из самых простых и эффективных алгоритмов – на основе кода Хаффмана. В начале исследования детально рассмотрен один из возможных методов реализации алгоритма Хаффмана, после рассмотрения которого автор применил его при написании своего архиватора JVC6. В продолжение работы описаны также специфические свойства других методов сжатия, их характерные признаки, возможные реализации. В конце исследований произведен анализ и сравнение нового архиватора с современными популярными программами сжатия данных. Произведен анализ алгоритмов для некоторых текстовых файлов. В результате работы написан новый архиватор и представлены способы его улучшения.

Autoreferāts

Šajā bakalaura darbā ir studēti jau eksistējošie saspiešanas algoritmi un arhivēšanas programmas, kā arī, izmantojot šo studiju laikā uzkrātās zināšanas, ir izstrādāts arhivātors JFC6, kas ir balstīts uz Haffmana kodēšanas algoritmu. Autors interesējas par to, kādi saspiešanas algoritmi un modelēšanas metodes ir visefektīvākie tekstisku failu saspiešanas ziņā.

No bakalaura darba apskatītā autors ir paveicis sekojošo:

- Izstudēja literatūru, saistītu ar informācijas saspiešanas jēdzienu.
- Izveidoja pārskatu par eksistējošiem saspiešanas algoritmiem un modelēšanas metodēm.
- Izmantojot uzkrātas zināšanas, Delphi 7 vidē uzrakstīja programmu JFC6, kas ir paredzēta tekstiskas informācijas saspiešanai.
- Izmantojot jaunizveidotu arhivātoru un mūsdienas populārākas saspiešanas programmas tika veikta algoritmu, metožu un arhivātoru salīdzināšana.
- Apkopojot iegūtus rezultātus tika izdarīti secinājumi par datu modelēšanas nepieciešamību, algoritmu ātrdarbību, saspiešanas efektivitāti.

Satura radītājs

Ievads **8**

1. Vispārīgā informācija **10**

1.1 Izmantoti jēdzieni un definīcijas 10

1.2 Ideāls saspiešanas algoritms 11

1.3 Modelēšana un kodēšana 11

1.4 Adaptīvie un neadaptīvie modeļi 12

1.5 Saspiešanas metožu klasifikācija 13

1.6 Saspiešanas metožu novērtējuma kritēriji 14

2. Mūsdienu saspiešanas metodes un algoritmi **16**

2.1 Statistiskas modelēšanas algoritmi 16

2.1.1 PPM dzimtas algoritmi 16

2.1.2 DMC dzimtas algoritmi 17

2.2 Vārdnīcu saspiešanas algoritmi 18

2.2.1 LZ77 dzimtas algoritmi 19

2.2.2 Tekstu saspiešana ar vārdu aizstāšanu 21

2.3 Saspiešanas algoritmi ar bloku šķirošanu 22

2.3.1 BWT algoritms 22

2.4 Entropijas kodēšanas metodes 24

2.4.1 Prefiksu kodi 24

2.4.1.1 Haffmana kods 26

2.4.1.2 Kanoniskais Haffmana kods 29

2.4.1.3 Kodu garuma aprēķināšana 33

2.4.1.4 Maksimālais koda garums 33

2.4.1.5 Kanonisku kodu aprēķināšana 37

2.4.1.6 Kodu koka saglabāšana 38

2.4.2 Aritmētiskie kodi 39

2.5 Lokāli-adaptīva kodēšana 41

2.5.1 Intervāla kodēšana 41

2.5.2 Grāmatu kaudzes metode (MTF) 42

2.6 Sēriju garumu vai grupveida kodēšana (RLE) 42

3. Veiktie eksperimenti un iegūtie rezultāti **43**

3.1 Problemātikas apraksts 43

3.2 Uzdevuma nostādne 45

3.3 Izmantotā metode 46
3.4 Arhivātors JFC6 46
3.4.1 JFC6 interfeisa apraksts 47
3.5 Arhivātoru un algoritmu testēšana un salīdzināšana 49
3.5.1 Arhivātoru saskarnes salīdzināšana 49
3.5.1.1 Kopsavilkums par saskarnēm 55
3.5.2 Vispārīgā testēšana 56
3.5.2.1 Testu kopsavilkums. 58
3.5.3 Speciālu gadījumu testēšana 62
3.5.3.1 XML failu saspiešana 62
3.5.3.2 Ciparisku failu saspiešana 65
3.5.3.3 Saspiešanas efektivitātes atkarība no izvelētas valodas 68
Noslēgums **72**
Izmantotas literatūras saraksts **74**

Ievads

Mūsdienās datortehnoloģijas strauji attīstās un informācijas apjomi, kurus mēs apstrādājam, ikdienā kļūst apjomīgāki un aizņem vairāk vietas. Mūsdienu esošā datortehnoloģija dod iespēju rast šādu risinājumu: saarhivēt nepieciešamo informāciju saspiestos arhīvos un tad to uzglabāšana aizņems daudz mazāk vietas datu nesējos.

Kas ir "Datu arhivēšana"? Tā ir datu saspiešana un būtiska faila izmēra samazināšana. Ar arhivēšanu mums ir nācies sastapties ļoti bieži, piemēram, kad bija nepieciešamība aiznest kādu ļoti svarīgu dokumentu uz mājām, lai vēlāk to apskatītu vai palabotu, vai nosūtītu liela apjoma failu draugam pa elektronisko pastu. Bet bija zināmas grūtības izdarīt minēto, jo disketes neļāva ierakstīt tajās liela izmēra failu, kā arī, faila izmērs elektroniskās vēstules pielikumam ir ierobežots. Šajos gadījumos palīgā nāk arhivātori, jeb failu saspiedēji. Tas ir visvienkāršākais piemērs, bet dzīvē nākas sastapties ar ievērojami būtiskākām lietām – informācijas apmaiņa starp attālinātām datu bāzēm, informācijas saspiešana lokālajās datu bāzēs.

Datu arhivēšana ir dotās informācijas saglabāšanai vai pārsūtīšanai nepieciešamā datu daudzuma samazināšanas process, izmantojot kodēšanas tehnoloģijas.

Eksistē daudzas tehnoloģijas, ko lieto ciparu skaitļotāji un sakaru ierīces, lai arhivētu (saspiestu) bināros datus. Binārajā sistēmā katru alfabēta burtu vai ciparu pārstāv astoņu bināro ciparu virkne. Elementārā datu saspiešanas sistēma ir atslēgas vārdu kodēšana, ar kuru bieži lietoti vārdi kā - "bet" tiek pārvērsti divu baitu kodā. Uzlabotākas tehnikas analīzē, identificē un tad aizvieto biežāk parādošos teksta paraugus ar vienu simbolu, piemēram, "šana". Tādos vārdos kā "skriešana" varētu tikt pārvērsta par "$", tādējādi ievērojami samazinot liela teksta bloka izmēru. Šīs tehnikas var arī pārstāvēt simbolus ar virknēm, kas īsākas par astoņiem bitiem. Tad simbolus, ko lieto biežāk kodē ar mazāku bitu skaitu. Prasība veiksmīgai atkodēšanai shēmās, kas lieto bitu virknes ar dažādiem garumiem ir, bitiem, kas apzīmē simbolu beigas, jābūt viennozīmīgi identificētiem. Haffmana kodēšana ir šīs tehnikas plaši izmantota forma. Plūstošā garuma (Run-length) kodēšana tiek lietota datiem, kas satur atkārtojošos simbolus, tā glabā atkārtotā simbola kodu vienreiz un norāda parādīšanos reižu skaitu.

Dažreiz priekš datu saspiešanas izmanto aparatūru ierīces. Tādām ierīcēm jābūt gan no sūtīšanas puses ,gan no saņemšanas puses. Rezultāta tie dod labus saspiešanas koeficientus un nelielus aizkavējumus. Tādas ierīces var būt gan ārējas gan iebūvētas, parādījās arī speciālas integrālās shēmas, kuras risina saspiešanas un dekompresijas problēmas.

Galvenie ieguvumi no datu saspiešanas ir: lielāks datu uzglabāšanas blīvums, efektīvāka informācijas pārraide ar faksa mašīnām un modemiem, un šifrēšana jeb informācijas slēpšana. Visprogresīvākās datu saspiešanas metodes raksturo kompromiss starp laiku un ātrumu. Parasti,

jo ilgāku laiku arhivēšanas programmai atļauj analizēt datus, jo lielāka būs saspiešana.

Informācijas saspiešana gan priekš glabāšanas gan priekš sūtīšanas ir aktuāls uzdevums. Tieši tāpēc datu saspiešana droši vien ir, un paliks aktuāla aizvien pieaugošo glabājamo un pārsūtāmo datu apjomu dēļ.

Saspiešanas tēma ir tik neaptverama, ka viena darba ietvaros nav iespējams aprakstīt visas saspiešanas metodes un algoritmus. Visām saspiešanas metodēm un to modifikācijām ir veltīti dažreiz pat grāmatu sējumi. Tāpēc darbā tiek apskatītas tikai bāzes idejas un koncepcijas, kas tiek pielietotas informācijas saspiešanas procesā.

Darbs ir sadalīts trijās galvenās daļās:

- Darba pirmajā nodaļā ir aprakstīti pamatjēdzieni, kas saistīti ar datu saspiešanas procesu. Dota algoritmu un metožu klasifikācija. Aprakstīti datu saspiešanas metožu novērtējuma kritēriji.
- Otrajā nodaļā tiek aprakstīti mūsdienas populārākie saspiešanas algoritmi un modelēšanas metodes.
- Trešā nodaļa ir veltīta eksistējošu algoritmu, metožu, arhivātoru salīdzināšanai un analīzei. Tiek salīdzināti mūsdienās populārākie arhivātori un tajos izmantotie algoritmi. Visvairāk uzmanības ir veltīts Haffmana algoritmam, jo darba mērķis ir izpētīt vai tiešam ir vajadzīgas visādas viltīgas modelēšanas metodes, lai panāktu labāku saspiešanu. Varbūt to izdosies izdarīt, neizmantojot modelēšanu, bet pielietojot datiem uzreiz Haffmana kodēšanas algoritmu.

1. Vispārīgā informācija

1.1 Izmantoti jēdzieni un definīcijas

Pieņemsim, ka eksistē galīgs alfabēts $\sum$ un kaut kāds avots *S*, kas ģenerē alfabēta $\sum$ simbolu virknes (rindas) kuras saucās par īsziņām.

Avots *S* saucas par Bernulli avotu, ja simbola *a* parādīšanas varbūtība ir vienāda ar $p_s(a)$ un nav atkarīga no iepriekšējiem simboliem.

Ja simbola *a*, kuru ģenerē avots *S*, parādīšanas varbūtība ir atkarīga tikai no viena iepriekšēja simbola, tad tādu avotu sauc par Markova pirmās kārtas avotu.

Markova pirmās kārtas avotu *S* var noteikt ar $|\Sigma|\times|\Sigma|$ matricu $P_s=p_{ij}$, kur p_{ij} – varbūtība ka simbols a_j parādīsies uzreiz pēc a_i . Varbūtība, ka *S* uzģenerēs rindu a_i ... a_{in} ir vienāda ar $q_{i_1} p_{i_1 i_2} \ldots p_{i_{n-1} i_n}$, kur *q* ir tāds vektors ,ka $P_{sq}=q$.

Analoģiska veidā var noteikt Markova k kārtas avotu, kur katra simbola parādīšanas varbūtība ir atkarīga no iepriekšējiem k simboliem.

Par kodu sauc atspoguļošanu $f : \sum^* \rightarrow \{0,1\}^*$, kur A^* - alfabēta *A* visu iespējamu galīgu rindu kopa.

Turpmāk alfabētu $\{0,1\}$ sauksim par bināru, bet alfabēta simbolus par bitiem.

Par rindas $S \in \sum^*$ koda vārdu sauc atbilstošu tai nuļļu un vieninieku virkni $f(s)$, bet par koda vārda garumu $/ f(s)/$ rindas $f(s)$ garumu.

Par koda *f*, avota *S* kodēšanas cenu $C_s(f)$ sauc koda vārda vidēju garumu:

$$C_s(f) = \sum_{s \in \Sigma^*} p_s(s) \mid f(\mathrm{s}) \mid.$$

Bitu rindu $a \in \{0,1\}^*$ sauc par viennozīmīgi dekodējamu, ja eksistē ne vairāk par vienu rindu $s \in \sum^*$ tādu, ka $f(s)=a$. Kodu *f* sauc par viennozīmīgi dekodējamu, ja jebkura bitu rinda $a \in \{0,1\}^*$ viennozīmīgi dekodējama.

Viennozīmīgas dekodēšanas vai nesagrozošas kodēšanas uzdevums ir sameklēt avotam *S* viennozīmīgi dekodējamu kodu *f*, kura koda vārda garums ir minimāls vai ir tuvs minimālam.

Par Bernulli avota *S* entropiju (vai 0 kārtas entropiju) sauc:

$$\varepsilon(S) = \sum_{a \in \Sigma} p_s(a) \log_2 \frac{1}{p_s(a)}$$

Par Markova pirmās kārtas avota *S* entropiju sauc:

$$\varepsilon_1(S) = \sum_{i,j=1}^{|\Sigma|} p_{ij} \log_2 \frac{1}{p_{ij}}$$

Šennons pierādīja, ka Markova avotiem vidējais koda vārda garums(viennozīmīgi

dekodējamam kodam) nevar būt mazāks par avota entropiju.

Lieko datu apjoms rindā S ir L(S)-H(S), kur L(S) – rindas garums bitos, H(S) – entropija (entropija – informācijas teorijas pamatjēdziens, kas kvantitatīvi raksturo gadījuma lieluma nenoteiktību. Praktiski informācijas avota entropija ir mazākais iespējamais bitu skaits, ko aizņem informācijas avots pēc saspiešanas (veikta ideāla kodēšana), nezaudējot datus. (dažos avotos entropiju mēra nevis bitos, bet bitos uz vienu nokodēto simbolu (vidēji)). Algoritmu, kuri varētu bez datu zaudēšanas saspiest tos līdz mazākam bitu skaitam kā to entropija, neeksistē.)

Par avota S koda f redundanci sauc starpību starp kodēšanas cenu un entropiju:

.
$$R_S(f) = C_S(f) - \varepsilon_S$$

Kodus f_1 un f_2 sauc par ekvivalentiem, ja visu kodu vārdu garumi sakrīt:

.
$$|f_1(s)| = |f_2(s)| \; \forall s \in \Sigma^*$$

Markova avota jēdziens ir ļoti svarīgs jo gandrīz visus tipiskus datus (piemēram tekstus) var viegli aprakstīt ar Markova modeļiem. Bet dažos gadījumos precīzs avota apraksts un tā uzbūve nav iespējama:

- nav avota, kas var uzģenerēt visas jau uzrakstītas grāmatas vai tekstus.
- nav avota, kas var uzģenerēt visas vēl neuzrakstītas grāmatas, tekstus, runas kas vēl nav pateiktas. [5].

1.2 Ideāls saspiešanas algoritms

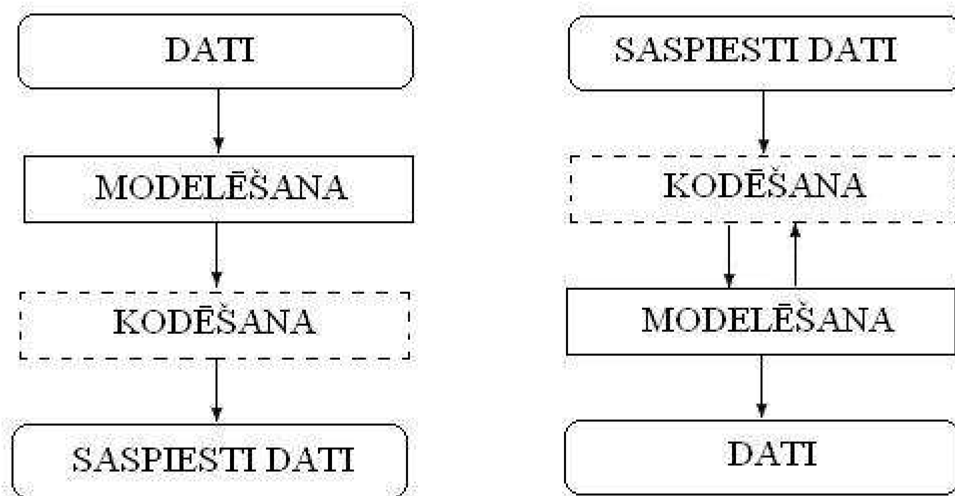
Nepieciešams uzreiz pateikt, ka neeksistē ideāla nesagrozoša saspiešanas algoritma, t.i. tāda, kurš saīsina jebkuru simbolu virknes garumu kaut kādam alfabētam(piem. binārам), pie tam garantējot viennozīmīgas dekodēšanas iespēju. Jebkurš nesagrozošs saspiešanas algoritms, kurš ir spējīgs samazināt dažu īsziņu garumus, obligāti palielinās citu īsziņu garumus.

1.3 Modelēšana un kodēšana

1981. gadā Rissanens un Langadons piedāvāja interesantu koncepciju saspiešanas procesa sadalīšanai uz divām neatkarīgam stadijām: uz modelēšanu un kodēšanu [5,6]. Modelēšanas etapā ar ieejas datiem ir izdarāmi daži pārveidojumi, kas samazina datu izmēru, padara datus labākus priekš saspiešanas.

Modelim nav obligāti jāapraksta saspiešanas process, viņam ir tikai jāpareģo datu uzvedība, un jo labāks un precīzāks pareģojums, jo labāka būs saspiešana. Vienīgais

ierobežojums, kas attiecas uz modeli ir tas , ka kodējot datus var lietot tikai to informāciju, kura būs pieejama kodēšanas laikā, nekādi citi ierobežojumi neeksistē. Vēl, modelis var izmainīties kodēšanas laikā, pielāgojoties kodējamajiem datiem. Tādi modeļi tiek saukti par adaptīviem.



Attēls 1. Modelēšana un kodēšana.

Modelēšana un kodēšana — divi pietiekami neatkarīgi un patstāvīgi informācijas teorijas apgabali. Saspiešanas kvalitāte tiek noteikta ar izmantotu modeli, tomēr ar labas kodēšanas paņēmienu saspiešanas kvalitāte var būt būtiski uzlabota, tāpēc gan modelēšanas gan kodēšana rada lielu praktisku interesi, dažreiz modelēšanas metožu pielietošana bez efektīvas kodēšanas ir bezjēdzīga.

Turpmāk, modelēšanas rezultātu kodēšana tiks saukta par entropijas kodēšanu, un ar kodēšanu sauksim modelēšanas posmu vai saspiešanu.

1.4 Adaptīvie un neadaptīvie modeļi

Lai viennozīmīgi dekodēt datus, dekodēšanas laikā ir jālieto tāds pats modelis, kas tika izmantots priekš kodēšanas. Lai to sasniegt eksistē trīs modelēšanas veidi: statiska, pusadaptēta un adaptēta modelēšana [5].

Statiska modelēšana izmanto visiem tekstiem vienu un to pašu modeli. Tā tiek palaista, palaižot kodēšanu, iespējams, izmantojot gaidāmā teksta tipa paraugu. Tāda modeļa kopija glabājas kopā ar dekodētāju. Galvenais trūkums- shēma dos neierobežoti sliktu saspiešanu katru reizi, kad kodējamais teksts neatbilst izvēlētajām modelim, tādēļ statisku modelēšanu izmanto tikai tad, kad ir svarīgi pirmkārt ātrums un realizācijas vienkāršība.

Pusadaptēta modelēšana risina problēmu, izmantojot katram tekstam savējo modeli, kurš tiek būvēts līdz saspiešanai uz iepriekšēja teksta (vai viņa parauga) caurskatīšanas rezultātu pamata. Pirms ir pabeigta saspiestā teksta noformēšana, modelis ir jānodod kodētājam. Neskatoties uz to ka jaunais modelis ir jānodod arī dekodētājām, šī stratēģija vispārējā gadījumā atmaksājas pateicoties labākajai modeļa atbilstībai tekstam.

Adaptēta (dinamiska) modelēšana nerāda problēmas ar modeļa nodošanu dekodētājam. No sākuma kodētājs un dekodētājs piešķir sev kaut kādu ne tukšu modeli, tādu kur visi simboli kodējamajā teksta būtu vienvērtīgi. Kodētājs izmanto šo modeli kārtēja simbola saspiešanai, bet dekodētājs – dekodēšanai (atspiešanai). Pēc tam tie izmanto eksistējošo modeli vienādā veidā (piemēram palielinot eksistējoša simbola paradīšanas varbūtību). Nākamais simbols tiek kodēts ar jaunizveidotu modeli, un pats pēc tam šo modeli izmaina. Dekodēšana notiek analoģiskā veidā, tā uztur identisku modeli, pateicoties tam, ka izmanto to pašu algoritmu. Izmantotais modelis, kuru nevajag nodod dekodētājam pa tiešo, būs labi piemērots kodējamām tekstam. Adaptētie modeļi salīdzinājumā ar neadaptētiem ir elegantāki un efektīvāki, jo tie netaisa teksta iepriekšējo analīzi un nodrošina labu saspiešanu (dažreiz labāku par neadaptētiem modeļiem). Turpretim, aprakstītai shēmai ir arī vāja vieta: modelis nekad nav nododams pa tiešo, līdz ar to kļūda var rasties tad, kad nepietiks atmiņas modeļa dinamiskai veidošanai.

Ir svarīgi, lai varbūtību vērtības, ko piešķir modelis, nebūtu vienādas ar nulli, jo simboli tiek kodēti ar log_p bitiem. Ja simbola varbūtība ir tuvu nullei, koda garums tiecas uz bezgalību. Nulles varbūtība parādās tikai tad, ja tekstā simbols nav sastopams. Šī situācija ir raksturīga adaptīviem modeļiem sākuma stadijai. Tā ir pazīstama kā nultas varbūtības problēma, kuru var atrisināt dažādos veidos. Viens piegājiens ir piešķirt katra simbola skaitītājam vieninieku.

1.5 Saspiešanas metožu klasifikācija

Saspiešanas metodes dalās uz dažām klasēm [5]:

- nesagrozošās (loseless) saspiešanas metodes garantē, ka dekodētie dati precīzi sakritīs ar ieejas datiem, kas bija saspiesti;
- sagrozošās (lossy) saspiešanas metodes(saucamie arī par saspiešanas metodēm ar zudumiem) var sagrozīt izejošos datus, piemēram, ar trokšņa signāla noraidīšanu, viņa spektra sašaurinājumu un t. t.

Bez tam, var izcelt

- vispārēji pielietojamās (general-purpose) saspiešanas metodes, kuras nav atkarīgas no ieeju datu fiziskas dabas un, kā likums, orientētas uz tekstu,

izpildāmu programmu, objektu moduļu un bibliotēku saspiešanu, t.i. datu, kas galvenokārt glabājas uz cietā diska.

- speciālas (special) saspiešanas metodes, kuras ir orientētas uz noteikta datu tipa saspiešanu, piemēram, skaņa, attēli un t. t. Tādēļ ka ir zināma šo datu specifiskas īpatnības ir iespējam sasniedz būtiski labāku saspiešanas kvalitāti, ka arī saspiešanas ātrumu, nekā izmantojot kopīgas iecelšanas saspiešanas metodes.

Pēc definīcijas , vispārēji pielietojamās metodes ir nesagrozošās; sagrozošas var būt tikai speciālas saspiešanas metodes. Sagrozījumi ir pieļaujami tikai apstrādājot dažādus signālus (skaņas, attēlus un t. t.), kad ir zināms kāda veidā un līdz kuram brīdim var izmainīt datus nepazaudējot to kvalitāti.

1.6 Saspiešanas metožu novērtējuma kritēriji

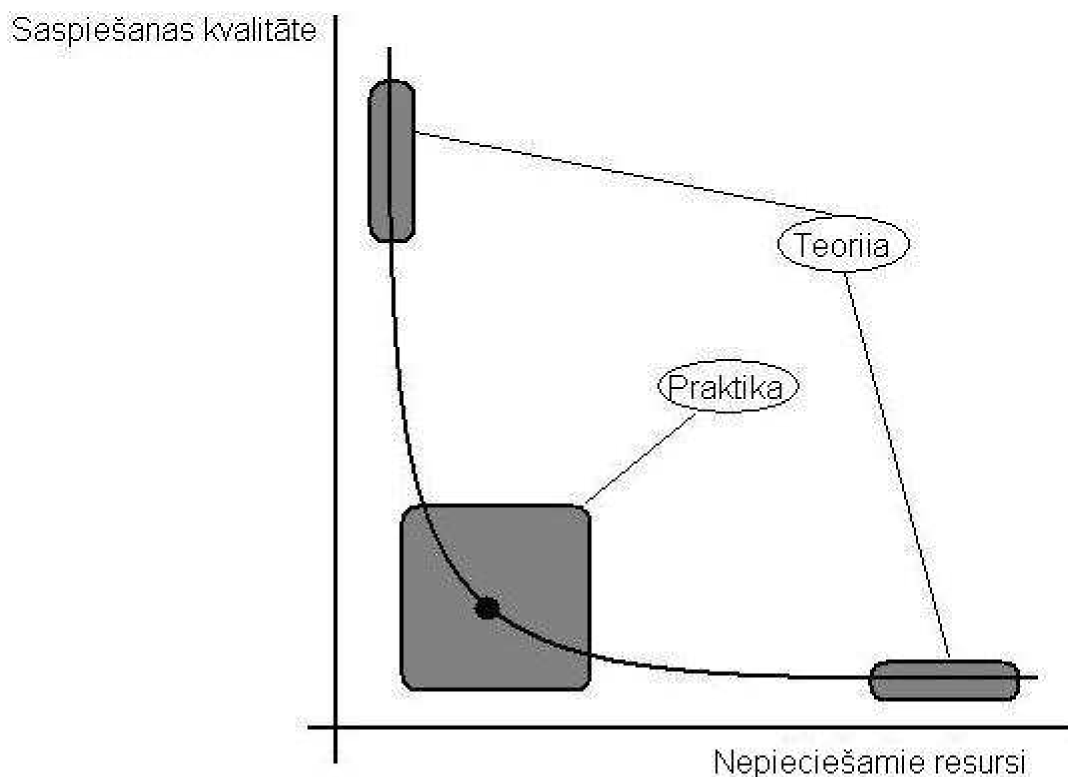
Datu saspiešanas algoritma galvenās īpašības ir:

- saspiešanas kvalitāte, t.i. saspiestu datu garuma attiecība pret sākuma datu garumu. Garums tiek ņemts bitos.
- kodēšanas un dekodēšanas ātrums;
- pieprasāmās atmiņas apjoms.

Datu saspiešanas jomā, kā tas bieži gadās, darbojas sviras likums (attēls 2): algoritmi, kas izmanto vairāk resursu (laika un atmiņas), parasti sasniedz labāko saspiešanas kvalitāti, un pretēji: algoritmi kas izmanto mazāk resursu parasti strādā ātrāk, bet nevar sasniegt labāku saspiešanas kvalitāti.

Tādējādi, datu saspiešanas algoritma, kurš ir optimāls no praktiska redzes viedokļa, uzbūve ir pietiekami netriviāls uzdevums, tā kā nepieciešams panākt pietiekami augstu saspiešanas kvalitāti (nav obligāti optimālu no teorētiska redzes viedokļa) pie neliela izmantojamu resursu apjoma.

Ir skaidrs, ka saspiešanas metožu novērtējuma kritēriji no praktiska redzes viedokļa ir atkarīgi no paredzama pielietojuma apgabala. Piemēram, izmantojot saspiešanu sistēmās, kas strādā reālajā laikā, ir nepieciešams nodrošināt kodēšanas un dekodēšanas augstu ātrumu. Iebūvētajām sistēmām kritisks parametrs ir pieprasāmās atmiņas apjoms. Ilglaicīgas datu glabāšanas sistēmām — saspiešanas kvalitāte un/vai dekodēšanas ātrums un t. t.



Attēls 2. Saspiešanas kvalitātes atkarība no resursiem.

Saspiešanas algoritmu novērtējuma kritērijos eksistē dažas atšķirības. Ja no teorētiska redzes viedokļa vislielāko interesi veido saspiešanas metožu uzbūve, kas ir optimāli pēc kvalitātes vai ātruma, tad praktiskas intereses apgabals ir ievērojami šaurāks un atrodas pārliekuma punkta apkārtnē 2. attēlā, kas atspoguļo saspiesto datu garumu atkarībā no pieprasāmo resursu apjoma.

2. Mūsdienu saspiešanas metodes un algoritmi

Atšķirīgu datu saspiešanas metožu skaitu var mērīt tūkstošos, tomēr daudzas no tām nav interesantas. Palikušas metodes var sadalīt uz trijām lielām klasēm.

2.1 Statistiskas modelēšanas algoritmi

Pēc saspiešanas kvalitātes vislabākie ir statistiskas modelēšanas algoritmi [5]:

- PPM (no angl. Prediction by Partial Matching).
- DMC (no angl. Dynamic Markov Compression).
- ACB (no angl. Associative Coding by Buyanovski).

Algoritmi veic analīzi ar kādu varbūtību un secību simboli parādās iepriekš iekodētās īsziņas. Ar šis analīzes palīdzību tie var noteikt nākamu simbolu.

Šiem algoritmiem piemīt ļoti lēns saspiešanas ātrums (2-20 Kb/s) un tie pieprasa lielu operatīvas atmiņas apjomu. Dekodēšanas ātrums praktiski neatšķiras no kodēšanas ātruma.

Neskatoties uz labu saspiešanas kvalitāti, izmantot statistiskas modelēšanas algoritmus praksē bieži vien ir grūti vai ir neiespējami lēna saspiešanas ātruma dēļ.

2.1.1 PPM dzimtas algoritmi

PPM dzimtas algoritmi (no angl. Prediction by Partial Matching).
Saņemot kārtējo simbolu, kuru vajag iekodēt, skatās konteksta statistikas aprakstu iepriekšējiem k simboliem, kas dod paradīšanas varbūtību vērtējumu simboliem, kas nāks pēc šī konteksta. Tā kā kontekstu skaits ir liels (bieži tas ir daži miljoni) tiek pielietota adaptīva aritmētiska kodēšana. Ja simbols i jau parādījās konkrētā kontekstā, viņš tiek kodēts saskaņā ar savu parādīšanas varbūtības vērtējumu:

$$p_i(a) = \frac{w_i(a)}{W_i}$$

kur $w_i(a)$ - simbola a parādīšanas skaits, bet

$$W_i = \sum_{a \in \Sigma} w_i(a)$$

Ja šīs simbols parādījās pirmo reizi, tad rodas nulles varbūtības problēma. Simbols ar nulles varbūtību nevar būt iekodēts. Parasti tas tiek atrisināts sekojoši:

visiem iespējamiem simboliem tiek piešķirta kaut kāda nenulles varbūtība:

$$p_i(a) = \frac{w_i + \varepsilon}{W_i + \varepsilon\left|\Sigma\right|}$$

Cits risinājums ir īpaša simbola izmantošana, to sauc par speciālu skrejošu simbolu (escape symbol): ja kodējamā simbola nav kontekstā ar kārtu A, tiek sūtīts speciāls simbols. Trūkstošs simbols tiek iekļauts tekošajā kontekstā un notiek pāreja pie (k-1) kārtas konteksta, pēc tam (ja nepieciešams – pie (k-2) un t. t.

Eksistē daudz atšķirīgu modifikācijas variantu priekš galvenā algoritma. Piemēram, nav acīmredzams kā simbolu svaru vajag obligāti palielināt par vieninieku. Vairākas PPM algoritma modifikācijas (PPMA, PPMB, PPMC, PPMD) tikai ar to arī atšķiras.

Visi simboli, kas atrodas k kārtas kontekstā, obligāti ir arī ar mazāku kārtu kontekstos. Kodējot speciālu simbolu (escape simbolu, t. i. kodējot simbolu, kas nav kontekstā) var uz laiku izslēgt simbolus no tekoša konteksta, kas atrodas kontekstos ar lielāku kārtu. Tas var ievērojami palielināt saspiešanas kvalitāti.

Kad tekošajā kontekstā nav kodējamā simbola nav neobligāti pāriet pie iepriekšējās kārtas konteksta. Ja tekošā konteksta kārta pietiekoši liela, tad ir iespējams, ka iepriekšējā kontekstā arī nav kodējamā simbola. Dažos gadījumos ir jēga pāriet uzreiz pie kontekstiem ar būtiski mazāku kārtu.

Nesen PPM autori piedāvāja jaunu variantu, nosauktu par PPM*, kurā tie piedāvāja izmantot neierobežota garuma kontekstus ar noteikumu, ka tāds konteksts ir unikāls. Viņi ievēroja, ka gari unikāli konteksti, kā likums, viennozīmīgi noteic nākošus simbolus. Neunikālu kontekstu glabāšana nepasliktina saspiešanas kvalitāti. Tomēr vairākos gadījumos pieprasāmās atmiņas apjoms priekš 9. kārtas PPM algoritma ir simti megabaitu. Tajā paša laikā 4. kārtas PPM* izmantojot 10-30 Mb ļauj panākt apmēram tādu pašu saspiešanas kvalitāti.

2.1.2 DMC dzimtas algoritmi

DMC dzimtas algoritmi (no angl. Dynamic Markov Compression). Paņemot kādu galīgu determinētu automātu, kodēšana notiek ar pāreju pie nākama stāvokļa pa loku, kas iezīmēts ar kodējamu simbolu, skatoties uz svaru, kas ir raksturīgs šim lokam. Dažos gadījumos notiek stāvokļu dublēšana(klonēšana) ar visiem lokiem, tā ka jaunajā stāvoklī var nokļūt tikai pa unikālu ceļu. Tāda veidā automāts adaptējas kodējamai simbolu

virknei.Oriģinālā variantā ir kodējami bināra alfabēta simboli un no katra stāvokļa iziet tieši divi loki. Tādēļ klonēšana nesagādā problēmas. Tomēr pieprasa daudz atmiņas (parasti ir nepieciešams 10-20 Mb).

DMC dzimtas algoritmu pieprasāmas atmiņas apjoms un saspiešanas laiks ir 2-4 reizēs mazāks nekā PPM algoritmam, bet saspiešanas kvalitāte ir sliktāka par 10-15%.

2.2 Vārdnīcu saspiešanas algoritmi

Vārdnīcu saspiešanas algoritmi aizstāj simbolu virknes ar atsaucēm uz identiskiem simbolu virknēm, kas atrodas vārdnīcā.

No praktiska redzes viedokļa šis klases vislabākie algoritmi ir LZ77 dzimtas algoritmi [5,6] (kurus pirmoreiz piedāvāja Zivs un Lempeļis 1997 g.), tas aizstāj kodējamās īsziņas sākumu ar atsauci uz pašu garāko identisku simbolu virkni, kas jau bija iekodēta iepriekš.

LZ77 dzimtas algoritmi ir 1.3-1.7 reizes sliktāki par statistiskas modelēšanas metodēm, vērtējot saspiešanas kvalitāti. Tomēr tiem ir raksturīgs liels saspiešanas ātrums (50-150 Kb/s) pie salīdzinoši neliela atmiņas apjoma (300-800 Kb). Ar saspiešanas kvalitātes pasliktināšanu 1.5 reizēs var panākt ātrumu 300-700 Kb/s.

LZ77 dzimtas algoritmu milzīga priekšrocība ir augsts dekodēšanas ātrums (1-2 Mb/s izmantojot otrreizēju saspiešanu ar Haffmana algoritmu, un 10-20 Mb/s bez tā). Tas ļauj pielietot tos gadījumos, kad dekodēšana notiek biežāk nekā kodēšana vai dekodēšanas ātrums ir ļoti svarīgs (piemēram, glabājot datus uz CD vai DVD diskiem, failu sistēmās ar saspiešanu).

Lielāka daļa no mūsdienas datu saspiešanas sistēmām ir uzbūvētas uz LZ77 algoritma bāzes, kas ilgus gadus bija visslābākais algoritms ar optimālu saspiešanas kvalitātes un ātruma attiecību. 1998 g. Storers savējā monogrāfijā „Data compression: Methods and Theory" (kas praktiski pilnīgi veltīta Ziva-Lempeļa dzimtas algoritmiem) deva vispārēja vārdnīcu saspiešanas algoritma aprakstu, kurš arī tiek aprakstīts zemāk.

Pieņemsim, ka eksistē rinda S, alfabēts $\sum$ un kaut kāda vārdnīca D, kas satur alfabēta $\sum$ rindas, pie kam rindu garumi ir vienādi. Pārlūkojot kodējamu simbolu virkni S, kodētājs atrod kaut kādu simbolu virkni ar garumu n, kas pieder vārdnīcai un ir sākums virknei S, un aizstāj to ar norādi uz identisku simbolu virkni, kas atrodas vārdnīcā. Tālāk notiek nobīde pa labi uz n simboliem, un varētu būt ka modificējas vārdnīca(tāda veidā, ka tā joprojām saturēs simbolu virknes ar vienādu garumu).

Ja kodētājs ir sinhronizēts ar dekodētāju t. i. tiem ir vienādas sākumu vārdnīcas un

modificē to ar vienādu metodi, dekodēšanas procesā notiek atsauču aizstāšanu ar atbilstošam simbolu virknēm no vārdnīcas. Ir skaidrs, ka eksistē milzīgs daudzums atšķirīgu veidu vārdnīcas organizēšanai, to modifikācijas, rindu meklēšanas un atsauču kodēšanas algoritmu .

Vārdnīcu saspiešanas algoritmiem ar vienkāršu piemērotu rindu izvēli no vārdnīcas (t.i. vai nu paša garāka vai pirmā piemērota) var konstruēt modeli, kas ir analoģisks PPM un DMC un nodrošina tādu pašu vai pat labāku saspiešanas rezultātu. Tādus vārdnīcu saspiešanas algoritmus dēvē par statistiskas modelēšanas metožu apakšklasi.

Tādējādi, var uzskatīt, ka jebkuram vārdnīcu saspiešanas algoritmam var uzbūvēt statistiskas modelēšanas modeli, kas nodrošina tādu pašu vai labāku saspiešanas kvalitāti, tādēļ vārdnīcu saspiešanas algoritmi nav (un nekad nebūs) līderi pēc saspiešanas kvalitātes. Tomēr tiem raksturīgas vērtīgas īpašības pēc kurām tie apsteidz gandrīz visus citus zināmus saspiešanas algoritmus :

- pietiekami augsts kodēšanas ātrums (5-20 reizēs augstāk, nekā statistiskas modelēšanas metodēm);
- ārkārtīgi augsts dekodēšanas ātrums, kas ir praktiski vienāds ar rindu kopēšanas ātrumu (100 tūkstošs reizes augstāk, nekā statistiskas modelēšanas metodēm);
- nelielais pieprasāmās atmiņas apjoms (10-100 reizēs mazāk par to, kas nepieciešams statistiskas modelēšanas metodēm);
- apmierinoša saspiešanas kvalitāte (parasti 1.2-1.7 reizes sliktāka, nekā statistiskas modelēšanas metodēm);

Tieši spriežot pēc šiem parametriem vairākās datu saspiešanas sistēmās pamatā ir kaut kāda vārdnīcu saspiešanas algoritma modifikācija.

2.2.1 LZ77 dzimtas algoritmi

LZ77 dzimtas algoritmi atšķiras no citiem vārdnīcu saspiešanas algoritmiem ar to, ka vārdnīcā šajā gadījumā ir apskatītas simbolu virknes visas apakšvirknes kopa (t.i. pēc divu simbolu apskatīšanas no virknes abcd, vārdnīca jau saturēs virknes a, ab, abc, b, bc, bcd), bet norāde tiek ierakstīta vārdnīcā ka (nobīde, garums), kur nobīde ir starpība starp eksistējošu virknes pozīciju un to pozīciju kur šī virkne parādījās iepriekšējā reizē. Tā kā vārdnīca nav obligāti satur visas rindas ar garumu viens, tad vārdnīcā var nebūt rindu, kas sākās no tekošā simbola. Tādos gadījumos kodētājs atstāj tekošu simbolu un nobīdās pie nākama.

Lai paaugstinātu apakšvirknes meklēšanas efektivitāti maksimāla nobīde tiek

ierobežota ar kādu parametru W. Apakšvirkni, kas ir kodējamas virknes vēl neiekodētas daļas sākums, ir jāmeklē tikai starp W iepriekšējiem simboliem (pa kreisi no tekošas pozīcijas). Tādas LZ77 algoritma modifikācijas saucas par slīdoša loga algoritmiem(LZ77 with sliding window). Tā kā iespējamas apakšvirknes atrodas logā ar platumu W, kas savukārt atrodas pirms tekošas pozīcijas, nav grūti pamanīt, ka LZ77 shēmas, bez ierobežojumiem uz nobīdi, ir apakšgadījumi LZ77 shēmām ar slīdošu logu, kur $W=\infty$.

Ja eksistē vairākas apakšvirknes, kas ir sākums tekošai(kas pašlaik tiek kodēta), parasti ņem visgarāko un vistuvāko. Šī metode saucās par LMH (Longest Match Heuristic).

Apzīmēsim ar $S = s_0 \ldots s_{L-1}$ kodējamu rindu ar garumu $L>0$, rindas simboli pieder alfabētam $\sum$, slīdoša loga platums $W>1$.

Par literāli sausas kodējamā alfabēta $|\sum|$ simbols, par norādi- naturālu skaitļu pāris (l,p) tāds, ka $l,p>0$ un $l,p<W$.

Par netukšas rindas $S \in \sum^*$ $(|S|>0)$ LZ77 kodu sauc netukšu virkni S' $(|S'|>0)$ ar norādēm un literāļiem, ja dekodējot S' ka rezultātu iegūsim S, dekodēšana notiek pēc sekojošā algoritma:

1) Uzstādīt i un j pozīcijā 0.
2) Palielināt j par 1:
   - ja virknes S' simbols j ir literālis a – pariet uz soļi 3.
   - citādi tā ir norāde(l,p)- pariet uz soļi 4.
3) Palielināt i par 1, uzstādīt $s_i \rightarrow a$, pariet uz soļi 6.
4) Ja $p> i$, tad S' nav korekts LZ77 kods, pabeigt dekodēšanu atgriezt tukšu rindu.
5) Priekš $k = 1,..., l$ uzstādīt $s_{i+k} \rightarrow s_{i+k-p}$, pēc tam palielināt i par l.
6) Ja $j < |S'|$, tad pariet uz soļi 2, citādi dekodēšanu pabeigta un rezultāts ir rinda $s_1 \ldots s_i$.

Tādējādi, kodēšanas rezultāts ir norādes un literāļu virkne, ko sauc par LZ kodu. Piemēram, rindas aaaaa LZ-kods var būt virkne a(1,1)(2,2)a vai arī a(1,4).

Vajadzētu pateikt, ka pašas garākās apakšvirknes kodēšana tekošajā pozīcijā ne vienmēr ir optimāla. Dažreiz ir jēga nokodēt literāli pat ja eksistē piemērota apakšvirkne. Piemēram, rinda abcbcababcab var būt iekodēta ka :

- abc(2,2)(2,5)(2,2)(3,7)(2,3)
- abc(2,2)(2,5)a(4,5)

Ir skaidrs ka norādes kodēšanas cena ir lielāka par literāļa kodēšanas cenu, līdz ar to otrs LZ kods ir īsāks par pirmo.

2.2.2 Tekstu saspiešana ar vārdu aizstāšanu

Viens no pašiem vienkāršākajiem vārdnīcu saspiešanas algoritmiem balstās uz sekojošas metodes: no visiem vārdiem, kas ir izmantoti tekstā, veidojam sarakstu(vārdnīcu) un kodēšanas laikā aizstājam vārdus ar norādēm uz identiskiem vārdiem, kas atrodas vārdnīcā. Tāds saspiešanas veids bija zināms kopš tiem laikiem, kad parādījās pirmie datori, pirmais publikācijas par šo metodi parādījās 1963 gadā. Kopš tā brīža tika piedāvāti simti atšķirīgi (atšķiras tikai ar detaļām) tekstu saspiešanas algoritmi ar vārdu aizstāšanu, izmantojot statistiskas vārdnīcas (tā kā saspiešana notiek divos posmos-vārdnīcas uzbūve un kodēšana), adaptīvas vārdnīcas, kad vārdnīcas uzbūve notiek kodēšanas laikā un saspiešanas metodes ar vārdnīcu papildināšanu (adaptīva kodēšana ar netukšu sākuma vārdnīcu).

Mūsdienās interese pēc tādām saspiešanas metodēm būtiski izauga, t. k. ievērojami palielinājās glabājamu un nododamu datu apjomi, kas bieži ir teksta informācija.

Saspiešanas efektivitāte ir atkarīga no tā, cik efektīvi kodējami:

- vārdi vārdnīcā;
- atsauces vārdnīcā;
- atdalītāji (atstarpes, komati un t. t.).

Parasti vārdu numuri vārdnīcā ir kodējami ar Haffmana kodiem ievērojot vārdu izmantošanas biežumu. Aritmētiski kodi ir mazāk noderīgi šim mērķim, t. k. pieprasa glabāt kopā ar saspiestajiem datiem nevis garumus koda vārdiem, bet vārdu paradīšanas biežumus (kas ir mazāk efektīvs un pasliktina saspiešanas kvalitāti) un, pats galvenais, pie lielas vārdnīcas apjoma aritmētisku kodu dekodēšanas ātrums uz kārtu ir zemāks nekā Haffmana kodu dekodēšanas ātrums.

Tā kā pie liela saspiežamu datu apjoma vārdnīcas izmērs var būt pārāk liels, Haffmana koda būvēšanas un vārdnīcas saglabāšanas uzdevums nemaz nav triviāls. Bieži izmanto vienkāršas pieejas: katru vārdnīcas simbolu saspiež ar Haffmana kodu. Vārdi vārdnīcā tiek sakārtoti tādā kartībā kā tie parādās tekstā.

2.3 Saspiešanas algoritmi ar bloku šķirošanu

Saspiešanas algoritmi ar bloku šķirošanu pieder pie BWT/BS dzimtas, kurus izstrādāja Barrouzs un Uilers 1994 g. [5,6]. Tie sadala kodējamu virkni uz blokiem no N ~ 10^6 simboliem, pārliek (noteiktā veidā) katra bloka simbolus tā, kā rodas viena un tā paša simbola atkārtošanas. Pēc tam saspiež pārveidotus datus ar jebkādu pietiekami vienkāršu saspiešanas algoritmu.

Pēc saspiešanas kvalitātes tie atgādina statistiskas modelēšanas metodes, bet pēc ātrdarbības(saspiešanas ātrums 50-150 Kb/s) – LZ77. Pie neliela atmiņas pieprasījuma, dekodēšanas ātrums sasniedz 300-500 Kb/s.

Bet algoritmam ir arī trūkumi. Norādītu ātrumu var sasniegt tikai strādājot ar teksta informāciju. Sliktākajos gadījumos ātrums var krietni samazināties, kas nav pieļaujams drošu sistēmu izstrādāšanā.

2.3.1 BWT algoritms

Saspiešanas algoritms ar bloku šķirošanu (no angl. BSLDCA — Block Sorting Lossless Data Compression Algorithm) [5]. Pirmo reizi tādu vienkāršu un elegantu algoritmu aprakstīja Barrouzs un Uilers 1994. gadā .

Lai saspiest rindu S ar garumu N, izveidojam matricu M ar izmēru N × N, kas sastāv no rindas S cikliskiem pagriezieniem pa vienam simbolam, tā kā matricas M rinda i satur simbolus:

$$M_i = S_i S_{i+1} \ldots S_{N-2} S_{N-1} S_0 S_1 \ldots S_{i-2} S_{i-1} ,$$

```
     matrica  M                          matrica  M'
a b c b c a b a b c a b .   0       a b a b c a b . a b c b c
b c b c a b a b c a b . a   1       a b c a b . a b c b c a b
c b c a b a b c a b . a b   2  ==>  a b c b c a b a b c a b .
b c a b a b c a b . a b c   3       a b . a b c b c a b a b c
a b a b c a b . a b c b c   4       b a b c a b . a b c b c a
b a b c a b . a b c b c a   5       b c a b a b c a b . a b c
a b c a b . a b c b c a b   6       b c a b . a b c b c a b a
b c a b . a b c b c a b a   7       b c b c a b a b c a b . a
c a b . a b c b c a b a b   8       c a b . a b c b c a b a b
a b . a b c b c a b a b c   9       c b c a b a b c a b . a b
. a b c b c a b a b c a b   1       . a b c b c a b a b c a b
```

Rezultātā saņemsim matricu M' , sakārtojot matricas M rindas leksikogrāfiskā kārtībā.

Barrouzs un Uilers parādīja, ka zinot matricas M’ pēdēju kolonnu (cb.cacabbb) un kodējamas rindas indeksu (dotajā piemērā — 2) var ar O(N) operācijām un O(N) papildus atmiņas vārdiem dekodēt iekodētu rindu.

Tāda datu pārveide saucas par Barrouza-Uilera modelēšanu (Burrows-Wheeler Transformation, BWT), pats par sevi algoritms nevis samazina kodējamo datu apjomu, bet palielina. Tomēr pārveidoti dati labāk piemēroti saspiešanai dažādu simbolu atkārošanas dēļ(kas labi redzams augšēja piemēra).

To var paskaidrot sekojoši: divām matricas M kaimiņ rindām, visticamāk, ir pietiekami garš kopīgs sākums, kuru var apskatīt kā kontekstu iepriekšējam simbolam, kas atrodas pēdējā M‘ kolonā. Ja konteksti ir līdzīgi, tad visticamāk iepriekšējie simboli ir vienādi.

Praksē rindu sakārtošana notiek ar rindu salīdzināšanu nevis virzoties no kreisas puses uz labu, bet otrādi, tā kā rezultāts būs nevis pēdēja matricas M’ kolona, bet pirmā.

Tas nedaudz uzlabo saspiešanas kvalitāti, t. k. ir pareģojams simbols, kurš nāks aiz konteksta, nevis atrodas pirms tā. Labāko saspiešanas kvalitāti var paskaidrot ar to, ka vairumā gadījumu dati tiek rakstīti no kreisas puses uz labo un nākamie simboli ir atkarīgi no iepriekšējiem, nevis pretēji .

Uilers izveidoja BWT realizāciju ar grāmatu kaudzes algoritmu(kuru izgudroja B. J. Rjabko). Uz šīs realizācijas piemēra tika parādīts, ka tā saspiešanas ātrums apmēram pusotrās reizēs sliktāks nekā LZ77 algoritmam, un 3-5 reizēs ātrāks par PPM un DMC algoritmiem. Tomēr vissliktākajā gadījumā izmantotas šķirošanas metodes darba laiks var pārsniegt $O(N\sqrt{N}\log_2 N)$. Saspiesto datu izmērs(t. i. saspiešanas kvalitāte) aizņem vidēju pozīciju starp LZ77 un PPM,DMC algoritmiem.

Bloku šķirošanas algoritma priekšrocības: augsts saspiešanas ātrums (tuvs LZ77), laba saspiešanas kvalitāte (tuva PPM un DMC).

Pastāv sekojošie trūkumi: diezgan liels, pieprasāmās atmiņas apjoms, kas ir lineārs priekš bloka garuma. Bloka izmēru parasti izvēlas pietiekami lielu - no simtiem kilobaitu līdz dažiem megabaitiem. Tad pieprasāmās atmiņas apjoms var būt nozīmīgs.

Saspiešanas realizācija ar bloku šķirošanu algoritmu sastāv no diviem gandrīz neatkarīgiem posmiem:

- Barrouza- Uilera pārveides efektīva realizācija.
- pārveides rezultāta efektīva kodēšana.

2.4 Entropijas kodēšanas metodes

Kā likums, augstāk uzskaitītas saspiešanas metodes tiek pielietotas nevis patstāvīgi, bet gan savienojumā ar kaut kādu entropijas kodēšanas metodi, kas aizstāj simbolus ar kodu vārdiem — nuļļu un vieninieku rindām — tā, kā biežāk sastopamajiem simboliem atbilst īsāki kodu vārdi.

Tādas kodēšanas metodes ir zināmas no 40 gadu beigām un labi izpētītas. Tos var sadalīt uz divām lielām klasēm: prefiksu kodi (Haffmana, Šennona, Šennona — Fano kodi) un aritmētiskie kodi.

2.4.1 Prefiksu kodi

Prefiksu kodi saucas tā tāpēc, ka nevienam kodu vārdam nav pilns sākums (t.i. prefikss) nekādam citam vārdam, kas garantē dekodēšanas viennozīmīgumu.

Ir zināmi daudzi prefiksu kodu būvēšanas veidi: Šennona un Šennona-Fano kodi ir gandrīz optimāli, bet Haffmana kods — ir optimāls visu prefiksu kodu starpā.

Tā kā katra koda vārda garums ir izpaužams ar veselu bitu skaitu, tad prefiksu kodi ir neefektīvi mazam alfabētam(2-8 simboli), vai tad kad eksistē simboli ar lielu paradīšanas varbūtību (t. i. lielāk par 30-50%) pēc saspiešanas kvalitātes tie var būt sliktāki par aritmētiskiem kodiem.

Par rindas $b=b_1 .. b_k$ prefiksu sauc tādu rindu $a= a_1 ...a_n$ $(n \leq k)$, ka $a_1=b_1, \dots , a_n = b_n$.

Kods tiek saukts par prefiksu kodu, ja neviens koda vārds nav prefikss kādam citam koda vārdam.

Prefiksa īpašība ir ļoti svarīga dažu iemeslu dēļ. Piemēram kodu (0,01,011) var viennozīmīgi dekodēt, bet dekodējot, piemēram, virkni 0110 ir jāskatās vai 0 ir kods simbolam 1, vai 01 – kods simbolam 2 un t. t. Prefiksu kodiem tas nedraud, jo dekodējamais simbols ir nodefinēts viennozīmīgi.

Otrkārt, Krafts un Makmillians parādīja, ka katram viennozīmīgi dekodējamam kodam eksistē ekvivalents viņam prefiksu kods (piemēram, prefiksu kods (0,10,110) ir ekvivalents augstāk pieminētam viennozīmīgi dekodējamam kodam).

Krafts norādīja prefiksu kodam vēl vienu svarīgu īpašību: katram prefiksu kodam eksistē binārs koks, kura lapas ir iezīmētas ar E alfabēta simboliem, bet šķautnes — ar nullēm un vieniniekiem, un ceļš no saknes pie lapas ir koda vārds simbolam kas apzīmē šo lapu.

Par koda koku sauc iezīmētu koku, kuram lapas ir iezīmētas ar E alfabēta simboliem, bet šķautnes- ar nullēm un vieniniekiem.

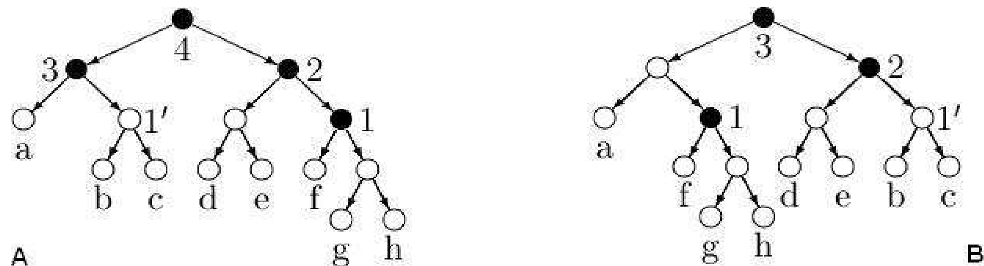
Turpmāk uzskatīsim, ka koda koka kreisas šķautnes iezīmētas ar nullēm, bet labas — ar vieniniekiem. Līdz ar to var uzstādīt viennozīmīgu atbilstību starp prefiksa kodu un attiecīgu koda koku.

Prefiksu kods tiek saukts par pilnu, ja jebkurai bitu virknei s, kas ir nekas cits ka prefikss kādam koda vārdam f(a), kas nav s, bitu virknes s0 un s1 (iegūti ar s konkatenāciju ar rindām 0 un 1 attiecīgi) — arī ir kaut kādu vārdu prefiksi.

Koks tiek saukts par kanonisku vai secīgu, ja tas ir pilns prefiksu kods un atbilst nosacījumiem:

- Ja koda vārds $f(a_1)$ ir īsāks par citu koda vārdu $f(a_2)$, tad $f(a_1)$ ir leksikogrāfiski mazāks par $f(a_2)$.
- Ja koda vārdu $f(a_1)$ un $f(a_2)$ garumi ir vienādi, tad $f(a_1)$ leksikogrāfiski ir mazāks par $f(a_2)$ tad un tikai tad, ja a_1 ir mazāks par a_2.

3. attēlā koks A ir kanonisks, bet B nav.



Attēls 3. Kanoniskais koks.

Par bitu virknes $a= a_1 \dots a_n$ vērtību sauc ne negatīvu skaitli v , kuram pieraksts binārajā sistēmā (skaitļošanas sistēma ar pamatu 2) sakrīt ar a:

$$v=a_1 2^{n-1}+a_2 2^{n-2}+ . . +a_n 2^0.$$

2.4.1.1 Haffmana kods

Šajā nodaļā mēs aplūkosim vienu no pašām izplatītākajām datu saspiešanas metodēm. Runa ir par Haffmana (Huffman code) vai prefiksu kodu ar minimālu redundanci (minimum-redundancy prefix code). Mēs sāksim ar Haffmana koda pamatidejām, izpētīsim dažas svarīgas īpašības un darba beigās ilustrēsim kodēšanas un dekodēšanas pilnu realizāciju, kas ir uzbūvēti uz idejām, izklāstītajiem šajā nodaļā.

Ideja, kas ir Haffmana koda pamatā, ir pietiekami vienkārša. Tā vietā, lai kodētu visus simbolus ar vienādu bitu skaitu (kā tas ir izdarīts, piemēram, ASCII kodēšanai, kur uz katru simbolu ir veltīti tieši 8 biti), kodēsim simbolus, kuri ir sastopami biežāk, ar mazāku bitu skaitu, nekā tos, kuri ir sastopami retāk. Mēģināsim to izdarīt tā, lai kods būtu optimāls vai, citiem vārdiem, ar minimālu redundanci.

Pirmo tādu algoritmu nopublicēja Devids Haffmans [1] (David Huffman)1952.gadā. Haffmana algoritms ir ar diviem posmiem. Pirmā posmā tiek būvēta biežumu vārdnīca un ir ģenerējami kodi. Otrajā posmā notiek tieša kodēšana.

Ir vērts pateiks, ka par 50 gadiem kopš publicēšanas dienas, Haffmana kods nemaz nav pazaudējis savu aktualitāti un vērtību. Var teikt, ka mēs sastopamies ar to, katru dienu (svarīgākais ir, ka Haffmana kods reti tiek izmantots atsevišķi, biežāk strādājot kopā ar citiem algoritmiem), praktiski ikreiz, kad arhivējam failus, skatāmies fotogrāfijai, filmas, nosūtām faksu vai klausāmies mūziku.

Pieņemsim ka $A=\{a_1,a_2\ldots a_n\}$ ir alfabēts no n dažādiem simboliem, $W=\{w_1,w_2\ldots w_n\}$ atbilstošu svaru kopa. Tad bināru kodu kopa $C=(c_1,c_2\ldots c_n)$ ar īpašībām:

- c_i nav prefikss jebkuram c_j , pie i != j
- $\sum_{i=1}^{n} w_i \mid c_i \mid$ ir minimāla ($|c_i|$ ir koda c_i garums)

saucās par prefiksu kodu ar minimālu redundanci, jeb Haffmana kodu.

Piezīmes:

1. Īpašība (1) saucas par prefiksu īpašību. Tā ļauj viennozīmīgi dekodēt mainīga garuma kodus.
2. Summu īpašībā (2) var definēt kā nokodēto datu izmēru bitos. Praksē tas ļoti ērti ļauj novērtēt saspiešanas kvalitāti bez tiešas kodēšanas.
3. Turpmāk, lai izbēgt pārpratumus, ar kodu apzīmēsim noteikta garuma bitu rindu. Bet ar minimāli - redundantu kodu, vai Haffmana kodu - kodu (bitu rindu) daudzumu, kas atbilst noteiktiem simboliem un apveltīts ar noteiktām īpašībām.

Zināms, ka jebkuram binārām prefiksu kodam atbilst noteikts binārs koks.

Bināru koku, kas atbilst Haffmana kodam, sauksim par Haffmana koku.

Haffmana koda būvēšanas uzdevums ir vienvērtīgs koka būvēšanas uzdevumam.

Aprakstīsim kopīgu shēmu Haffmana koka būvēšanai:

1. Sastādīsim kodējamo simbolu sarakstu (pie tam aplūkosim katru simbolu kā vien elementu bināru koku, kura svars ir vienāds simbola svaram).
2. No saraksta izvēlēsimies 2 mezglus ar vismazāko svaru.
3. Izveidosim jaunu mezglu un pievienosim pie viņa, divus bērnus(divi mezgli, kas ir izvēlēti no saraksta). Pie kam, izveidota mezgla svars būs vienāds ar bērnu svaru summu.
4. Pieliksim izveidotu mezglu pie saraksta.
5. Ja sarakstā ir vairāk nekā viens mezgls, tad atkārtot 2-5.

Apskatīsim piemēru: uzbūvēsim Haffmana koku rindai:

S="A H F B H C E H E H C E A H D C E E H H H C H H H D E G H G G E H C H H ".

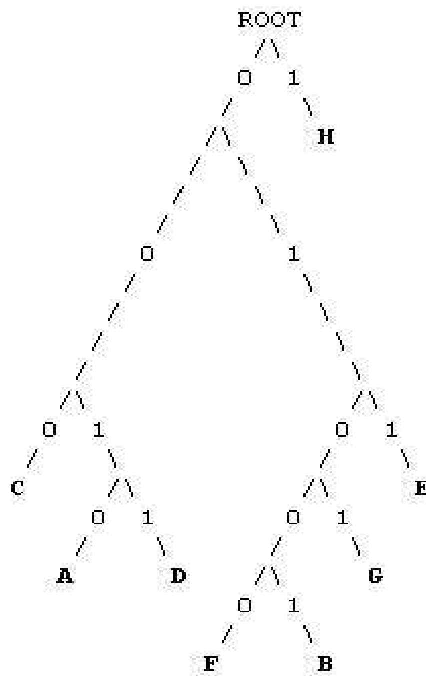
No sākumam ievedīsim dažus apzīmējumus:

1. Mezglu svara apzīmēsim ar apakšējiem indeksiem: A_5, B_3, C_7.
2. Saliktus mezglus liksim iekavās: $((A_5+B_3)_8+C_7)_{15}$.

Tātad, mūsu gadījumā A={A, B, C, D, E, F, G, H}, W={2, 1, 5, 2, 7, 1, 3, 15}.

1. $A_2\ B_1\ C_5\ D_2\ E_7\ F_1\ G_3\ H_{15}$
2. $A_2\ C_5\ D_2\ E_7\ G_3\ H_{15}\ (F_1+B_1)_2$
3. $C_5\ E_7\ G_3\ H_{15}\ (F_1+B_1)_2\ (A_2+D_2)_4$
4. $C_5\ E_7\ H_{15}\ (A_2+D_2)_4\ ((F_1+B_1)_2+G_3)_5$
5. $E_7\ H_{15}\ ((F_1+B_1)_2+G_3)_5\ (C_5+(A_2+D_2)_4)_9$
6. $H_{15}\ (C_5+(A_2+D_2)_4)_9\ (((F_1+B_1)_2+G_3)_5+E_7)_{12}$
7. $H_{15}\ ((C_5+(A_2+D_2)_4)_9+(((F_1+B_1)_2+G_3)_5+E_7)_{12})_{21}$
8. $(((C_5+(A_2+D_2)_4)_9+(((F_1+B_1)_2+G_3)_5+E_7)_{12})_{21}+H_{15})_{36}$

Sarakstā, kā arī bija uzdots, palika tikai viens mezgls. Haffmana koks ir uzbūvēts. Tagad pierakstīsim to koka veidā .



Attēls 4. Haffmana koks.

Haffmana koka lapu mezgli atbilst kodējamā alfabēta simboliem. Lapu mezglu dziļums ir vienāds atbilstošo simbolu kodu garumam.

Ceļu no koka saknes uz lapu mezglu var iedomāties bitu rindas veidā, kurā "0" atbilst kreisa apakškoka izvēlei, bet "1" - laba. Izmantojot šo mehānismu, mēs bez piepūles varam atrast kodus visiem kodējamā alfabēta simboliem. Izrakstīsim, piemēram, kodus visiem simboliem mūsu piemērā:

$A=0010_{bin}$	$C=000_{bin}$	$E=011_{bin}$	$G=0101_{bin}$
$B=01001_{bin}$	$D=0011_{bin}$	$F=01000_{bin}$	$H=1_{bin}$

Tagad mums ir viss nepieciešamais, lai nokodētu rindu S. Diezgan vienkārši aizstāt katru simbolu ar atbilstošu kodu:

S’="00101010000100110000111011100001100101001100001101111100011100110110101101 010101011100011".

Novērtēsim tagad saspiešanas pakāpi. Sākuma rindā S bija 36 simboli, uz katru no kuriem bija pa [$\log_2$|A|]=3 bitiem (šeit un tālāk kvadrātiekavas [] ir vesela daļa ar apaļošanu uz augšu, t.i. [3,018]=4). Tādējādi, S izmērs ir 36*3=108 baiti.

Nokodētās S’ rindas izmēru var iegūt izmantojot 1. definīcijas otro piezīmi, vai saskaitot bitu daudzumu rindā S’. Abos gadījumos rezultāts būs viens un tas pats - 89 biti.

Tātad, mums izdevās saspiest tekstu kas aizņem 108 bitus - 89 bitos.

Tagad dekodējam rindu S'. Sākot ar koka sakni kustēsimies uz leju, izvēloties kreisu apakškoku, ja kārtējais bits rindā ir vienāds ar "0", un labo - ja "1". Nonākot līdz lapu mezglam mēs dekodējam simbolu.

Sekojot šim algoritmam mēs precizitātē saņemsim sākuma rindu S.

2.4.1.2 Kanoniskais Haffmana kods

No iepriekšējas nodaļas var secināt, ka Haffmana kodam var būt vairākas modifikācijas un veidi. Mēs varam to pakļaut jebkurām transformācijām bez efektivitātes zaudējuma ievērojot tikai divus nosacījumus: koda garumiem nedrīkst mainīties; tiem jābūt prefiksa kodiem.

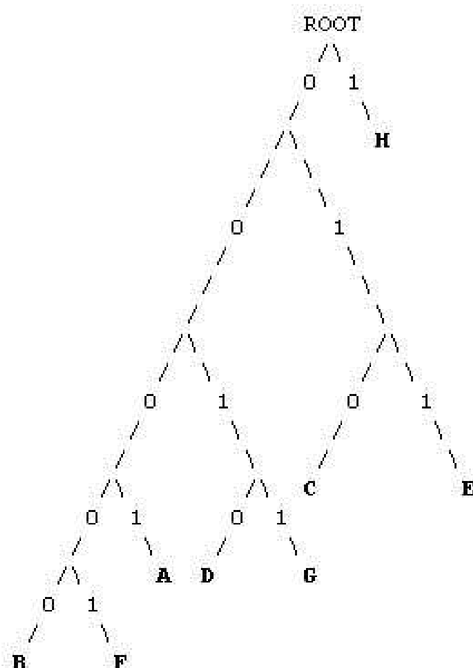
Haffmana kodu $D=\{d_1,d_2,\ldots,d_n\}$ sauc par kanonisku [2], ja:

- Īsu kodu (ja tos papildināt ar nullēm no labas puses) ir skaitlisks vairākums
- Vienāda garuma kodu skaits palielinās kopā ar alfabētu.

Tālāk, sauksim kanonisku Haffmana kodu vienkārši par kanonisku kodu.

Bināru koku, kas atbilst kanoniskam Haffmana kodam, sauksim par kanonisku Haffmana koku.

Kā piemēru ilustrēsim kanonisku Haffmana koku rindai S, un salīdzināsim to ar parastu Haffmana koku.



Attēls 5. Kanoniskais Haffmana koks.

Izrakstīsim tagad kanoniskus kodus visiem mūsu alfabēta simboliem bināra un decimālā pierakstā. Pie tam sagrupēsim simbolus pēc koda garuma.

$B=00000_{bin}=0_{dec}$ $A=0001_{bin}=1_{dec}$ $C=010_{bin}=2_{dec}$ $H=1_{bin}=1_{dec}$

$F=00001_{bin}=1_{dec}$ $D=0010_{bin}=2_{dec}$ $E=011_{bin}=3_{dec}$

$G=0011_{bin}=3_{dec}$

Pārliecināsimies, ka izpildās divas īpašības no definīcijas 3:

Aplūkosim, piemēram, divus simbolus: E un G. Papildināsim $E=011_{bin}=3_{dec}$ simbola kodu ar divām nullēm no labas puses(maksimālais koda garums ir 5, tātad 5-3=2): $E'=01100_{bin}=12_{dec}$, analoģiski saņemsim $G'=00110_{bin}=6_{dec}$. Ir redzams ka E'>G'.

Aplūkosim tagad trīs simbolus: A, D, G. Tiem visiem ir viena garuma kodi. Leksikogrāfiski A<D<G. Tādas pašas attiecības ir arī to kodiem: 1<2<3.

Tālāk ievērosim, ka jebkura lapu mezgla kārtas numurs(tajā līmeni, kur viņš atrodas), ir vienāds ar atbilstoša simbola kodu. Tādu kanonisku kodu īpašību sauc par skaitlisku (Numerical property).

Paskaidrosim iepriekšteikto uz piemēra. Aplūkosim simbolu C. Tas atrodas trešajā līmenī (koda garums ir 3). Viņa kārtas numurs ir vienāds ar 2 (ņemot vērā divus mezglus pa kreisi, kas nav lapas), t.i. vienāds simbola C kodam. Tagad ierakstīsim šo numuru binārajā formā un papildināsim viņu ar nulles bitu no kreisas puses (t. k. 2 aizņem divus bitus, bet C simbola kods trīs): 2dec=10bin=>010bin. Mēs saņēmām precīzu simbola C kodu.

Tādējādi, mēs atnācām pie ļoti svarīga secinājuma: kanoniski kodi var būt pilnīgi noteikti ar saviem garumiem. Šī kanonisku kodu īpašība ļoti plaši tiek izmantota praksē.

Tagad atkal iekodēsim rindu S, bet jau ar kanonisku kodu palīdzību:

Z'="00011000010000010100111011101001100011001001001101111101011100100110011100
110011011101011"

Sakarā ar to, ka mēs neesam izmainījuši kodu garumus, nokodētās rindas izmērs nav izmainījies: |S'|=|Z'|=89 biti.

Tagad dekodēsim rindu Z', izmantojot kanonisku kodu īpašības.

Uzbūvēsim trīs masīvus: base[], symb[], offs[].

- base[i] – mezglu, kas nav lapas, daudzums līmenī i;
- symb[] - alfabēta simbolu permutācija, kas ir sakārtota ievērojot divus nosacījumus:
  - ✓ koda garums
  - ✓ leksikogrāfiskā nozīme
- offs[i] - symb[] masīva indekss, tāds ka symb[offs[i]] ir pirmais lapu mezgls (simbols) līmenī i.

Mūsu gadījumā:

- base[0..5]={?,1,2,2,1,0},
- symb[0..7]={B,F,A,D,G,C,E,H},

- offs[0..5]={?,7,?,5,2,0}.

Tagad paskaidrosim dažas lietas. base[0]=? un offs[0]=? netiek izmantoti sakarā ar to, ka nav kodu ar garumu 0, bet offs[2]=? jo otrajā līmenī nav lapu mezglu.

Apskatīsim dekodēšanas algoritmu (CANONICAL_DECODE) [3] :

- code = nākamais ieejas bits, length = 1
- Kamēr code < base[length]

  code = code << 1

  code = code + nākamais bits

  length = length + 1
- symbol = symb[offs[length]+ code - base[length]]

Tādejādi, lasīsim katru bitu mainīgajā code līdz tam laikam, kamēr code < base[length]. Pie tam, katrai iterācijai palielināsim mainīgo length par 1 (t.i. bīdīsimies pa koku uz leju). Cikls apstāsies pozīcija 2 kad mēs noteiksim koda garumu (t. i. līmenis kokā, kur atrodas meklētais simbols). Paliek tikai noteikt kāds tieši simbols šajā līmenī mums ir vajadzīgs.

Pieņemsim, ka pēc dažam iterācijām cikls apstājās pozīcijā 2. Šajā gadījumā izteiksme (code - base[length]) ir meklētā mezgla (simbola) kārtas numurs līmenī length. Pirmais mezgls (simbols), kas atrodas kokā līmenī length, ir masīva symb[] simbols ar indeksu offs[length]. Bet mūs interesē nevis pirmais simbols, bet gan simbols ar numuru (code - base[length]). Tādēļ meklētā simbola indekss masīvā symb[ ] vienāds ar (offs[length]+ (code - base[length])). Citiem vārdiem, meklētais simbols symbol=symb[offs[length]+ code - base[length]].

Apskatīsim konkrēto piemēru. Pielietojot apskatīto algoritmu dekodējam rindu Z'.

Z'="0001100001000001010011101110100110001100100100110111110101110010011001110 0110011011101011"

1. code = 0, length = 1
2. code = 0 < base[length]= 1

   code = 0 << 1 = 0

   code = 0 + 0 = 0

   length = 1 + 1 = 2

   code = 0 < base[length]= 2

   code = 0 << 1 = 0

   code = 0 + 0 = 0

   length = 2 + 1 = 3

   code = 0 < base[length]= 2

   code = 0 << 1 = 0

   code = 0 + 1 = 1

length = 3 + 1 = 4

code = 1 = base[length]= 1

3. symbol = symb[offs[length]= 2 + code = 1 - base[length]= 1]= symb[2]= A

1. code = 1, length = 1

2. code = 1 = base[length]= 1

3. symbol = symb[offs[length]= 7 + code = 1 - base[length]= 1]= symb[7]= H

1. code = 0, length = 1

2. code = 0 < base[length]= 1

code = 0 << 1 = 0

code = 0 + 0 = 0

length = 1 + 1 = 2

code = 0 < base[length]= 2

code = 0 << 1 = 0

code = 0 + 0 = 0

length = 2 + 1 = 3

code = 0 < base[length]= 2

code = 0 << 1 = 0

code = 0 + 0 = 0

length = 3 + 1 = 4

code = 0 < base[length]= 1

code = 0 << 1 = 0

code = 0 + 1 = 1

length = 4 + 1 = 5

code = 1 > base[length]= 0

3. symbol = symb[offs[length]= 0 + code = 1 - base[length]= 0]= symb[1]= F

Tātad, mēs dekodējām 3 pirmos simbolus: A, H, F. Tādejādi, sekojot šim algoritmam mēs rezultātā saņemsim rindu S.

Tas, droši vien, pats vienkāršākais algoritms, kanonisko kodu dekodēšanai. Tam var izdomāt vairākas modifikācijas. [3]

2.4.1.3 Kodu garuma aprēķināšana

Lai nokodētu rindu, mums nepieciešams zināt simbolu kodus un to garumus. Kā jau bija pateikts iepriekšējā nodaļā, kanoniski kodi tiek pilnīgi noteikti ar saviem garumiem. Tādējādi, mūsu galvenais uzdevums ir kodu garumu aprēķināšana.
Izrādās, ka šī uzdevuma risināšanai, nav obligāti vajadzīga Haffmana koka būvēšana. Algoritmi kuri izmanto kaut kādu savu struktūru Haffmana koka glabāšanai izrādās ir daudz efektīvāki, salīdzinot ātrdarbību un atmiņas izmaksas.
Uz šo brīdi eksistē diezgan liels skaits efektīvu kodu garumu aprēķināšanas algoritmu [2]. Mēs apskatīsim tikai vienu no tiem. Šis algoritms ir pietiekami vienkāršs, bet neskatoties uz to ļoti populārs. Viņš tiek izmantots tādās programmās kā zip, gzip, pkzip, bzip2 un daudzās citās.
Atgriezīsimies pie Haffmana koka būvēšanas algoritma. Katras iterācijas laikā, mēs izpildījām divu mezglu(ar vismazāko svaru) lineāru meklēšanu. Šim mērķim vairāk ir piemērota prioritāšu rinda, tāda kā piramīda (minimāla). Mezglam ar vismazāko svaru būs visaugstākā prioritāte un tas atradīsies uz piramīdas virsotnes. Apskatīsim šo algoritmu.

1. Iekļausim visus kodējamos simbolus piramīdā.
2. Konsekventi izvilksim no piramīdas 2 mezglus (tie būs divi mezgli ar vismazāko svaru).
3. Izveidosim jaunu mezglu un pievienosim pie tā divus bērnus (mezgli paņemti no piramīdas. Izveidota mezgla svars būs vienāds ar bērnu svaru summu.
4. Ieliksim izveidotu mezglu piramīdā.
5. Ja piramīdā ir vairāk nekā viens mezgls, tad atkārtot 2-5.

Pieņemsim, ka katram mezglam ir saglabāta norāde uz vecākiem mezgliem. Koka saknei šī norādē ir vienāda ar NULL. Izvēlēsimies tagad lapas mezglu (simbolu) un sekojot saglabātajām norādēm celsimies augšā pa koku, kamēr norāde nebūs vienāda ar NULL. Pēdējais nosacījums nozīmē, ka mēs nonācām līdz koka saknei. Skaidrs, ka soļu skaits ko mēs izdarījām ejot no mezgla līdz saknei ir mezgla(simbola) dziļums, kā arī viņa koda garums. Apejot tādējādi visus mezglus, mēs saņemsim to kodu garumus.

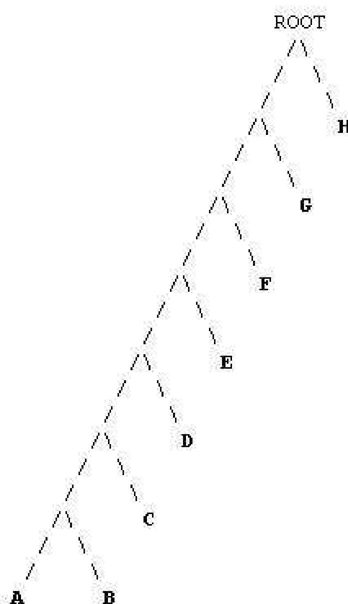
2.4.1.4 Maksimālais koda garums

Kodēšanā tiek izmantota tā saucamā kodu grāmata (CodeBook), vienkārša datu struktūra, praksē - divi masīvi: viens ar garumiem, cits ar kodiem. Citiem vārdiem, kods (kā bitu rinda) glabājas atmiņas šūnā vai fiksētā izmēra reģistrā (biežāk 16, 32 vai 64). Tādēļ, lai nenotiktu pārpildīšana, mums ir jāpārliecinās, ka kods nokļūs reģistrā.

Izrādās, ka N-simbolu alfabētam maksimāls koda izmērs var sasniegt (N-1) bitus. Citiem vārdiem runājot, pie N=256 (klasiskais variants) mēs varam saņemt 255 bitu garu kodu (failam vajag būt ļoti lielam: $2.292654130570773*10^{53}$~=$2^{177.259}$)! Ir skaidrs, ka tāds kods reģistrā nenovietosies un ar viņu vajag kaut ko darīt.

No sākuma noskaidrosim pie kādiem nosacījumiem rodas pārpildīšana. Pieņemsim, ka simbola i biežums ir vienāds ar Fibonači *i-to* skaitli.

Piemēram: A-1, B-1, C-2, D-3, E-5, F-8, G-13, H-21. Uzbūvēsim atbilstošo Haffmana koku (6. attēls).



Attēls 6. Haffmana kods.

Tādu koku sauc par deģenerētu. Lai saņemtu tādu koku, simbolu biežumiem vajadzētu augt kā Fibonači skaitļiem vai vēl ātrāk. Kaut arī praksē, ar reāliem datiem, tādu koku saņemt praktiski nav iespējams. To ir ļoti viegli saģenerēt mākslīgi. Jebkurā gadījumā šo bīstamu situāciju vajadzētu ievērot.

Šo problēmu var atrisināt ar diviem iespējamiem veidiem. Pirmais no tiem balstās uz vienas īpašības, kas ir raksturīga kanoniskiem kodiem. Kanoniskajā kodā (bitu rindai) var būt ne vairāk, ka [log2N] ne nulles bitu. Visus pārējus bitus var vispār nesaglabāt. Tie vienmēr ir vienādi ar nulli. N=256 gadījumā mums pietiek no katra koda saglabāt tikai jaunākus 8 bitus, uzskatot ka visi pārējie biti vienādi ar nulli. Tas atrisina problēmu, bet tikai daļēji. Tas ievērojami palēninās gan kodēšanu, gan dekodēšanu. Tāpēc šī metode reti tiek pielietota praksē.

Otrais veids ir koda garuma mākslīgais ierobežojums (būvēšanas laikā, vai arī pēc tā). Šis veids ir visizplatītākais, tāpēc mēs apskatīsim to detalizētāk.

Eksistē dīvu tipu algoritmi, kas ierobežo kodu vārdu garumus: tuvinātie un optimālie. Optimālie algoritmi ļoti sarežģīti no realizācijas viedokļa - prasa daudz laika un atmiņas. Tuvināta koda

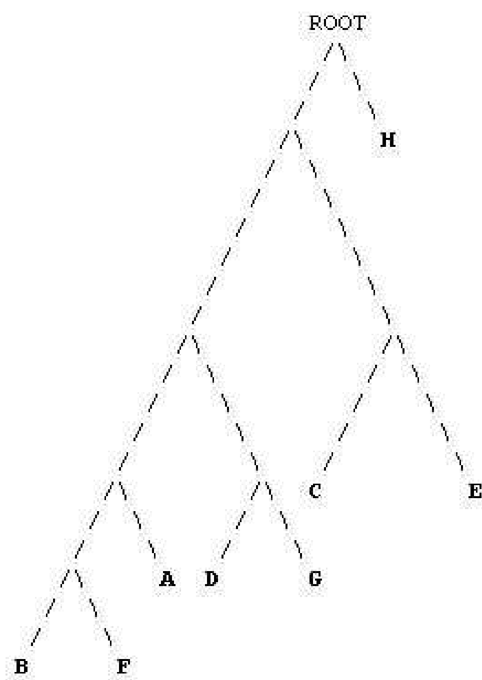
efektivitāte ir atkarīga no tā, cik tuvu viņš ir optimālam kodam. Jo mazāk šī starpība, jo labāk. Ir vērts pateikt, ka dažiem tuvinātiem algoritmiem šī starpība ir niecīga [4]), pie kam tie bieži ģenerē optimālu kodu (kaut arī negarantē, ka tas būs vienmēr). Tā kā praksē pārpildīšana notiek ārkārtīgi reti(ja nav uzlikti stingri ierobežojumi uz maksimālu koda garumu), mazam alfabētam ir lietderīgi pielietot vienkāršas un ātras tuvinātas metodes.

Mēs aplūkosim vienu pietiekami vienkāršu un ļoti populāru tuvināšanas algoritmu. Tas ir pielietots tādās programmās kā zip, gzip, pkzip, bzip2 un daudzās citās.

Maksimāla koda garuma ierobežošanas uzdevums ir ekvivalents Haffmana koka augstuma ierobežošanai. Ievērosim, ka būvējot Haffmana koku jebkuram ne lapas mezglam ir tieši divi pēcnācēji. Katras iterācijas laikā samazināsim koka augstumu par 1. Pieņemsim, ka L ir maksimāls koda garums (koka augstums) un ir nepieciešams to ierobežot līdz L' < L. LN_i - pats kreisais lapu mezgls līmenī i, bet RN_i pats labais .

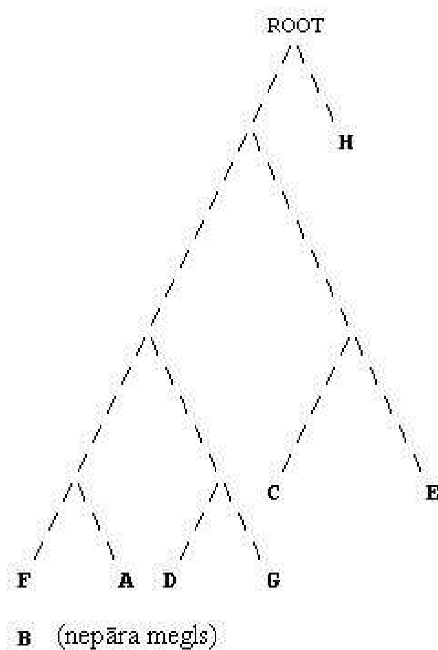
Sāksim darbu ar līmeni L. Pārvietosim RN_L mezglu uz savu vecāko vietu. Sakarā ar to, ka mezgli iet pa pāriem mums nepieciešams atrast vietu arī RN_L kaimiņ mezglam. Sameklēsim līmeni j tādu, ka j < (L-1), j atrodas vistuvāk L un satur lapu mezglus. LN_j vietā izveidosim mezglu un pievienosim pie tā „bērnus"- LN_j un mezglu no līmeņa L, kurš ir palicis bez pāra. Visiem palikušiem mezglu pāriem līmenī L pielietosim to pašu operāciju. Ir skaidrs, ka izkārtojot mezglus šādā veidā mēs samazinājām mūsu koka dziļumu par 1. Tagad dziļums ir vienāds ar (L-1) . Šo operāciju var atkārtot, kamēr vajadzīgais ierobežojums nebūs panākts.

Atgriezīsimies pie mūsu piemēra, kur L=5. Ierobežosim maksimālu koda garumu līdz L'=4 (7. attēls).



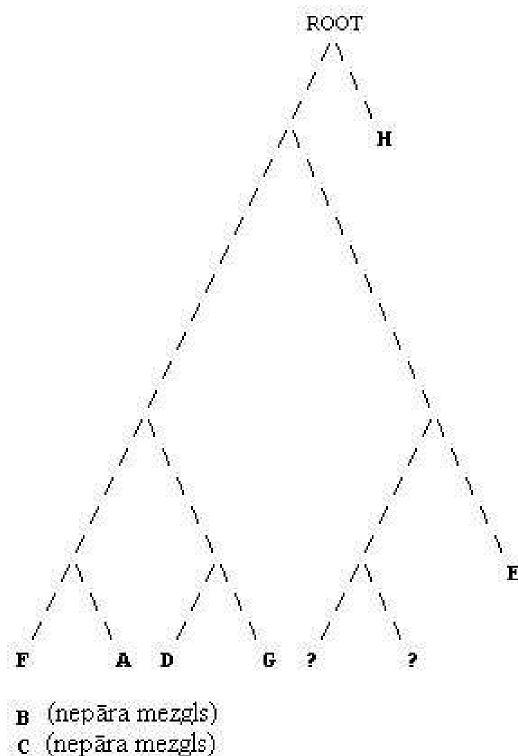
Attēls 7. Ierobežojums maksimālam koda garumam.

Redzams, ka mūsu gadījumā RN_L=F, j=3, LN_j=C. No sākumā pārvietosim mezglu RN_L=F uz savu „vecāku" (parent) vietu (8. attēls).



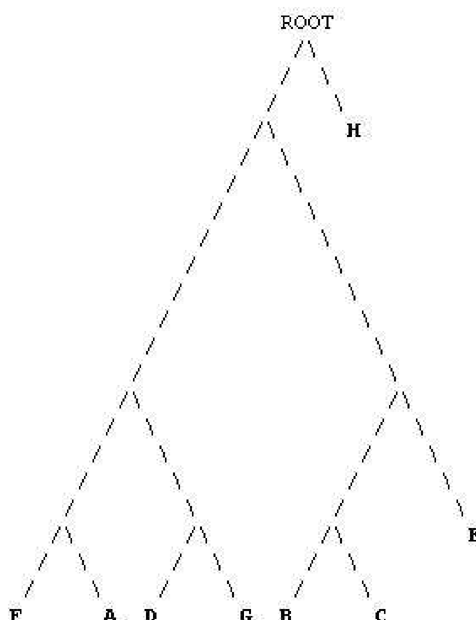
Attēls 8. Koka pārveidošana.

Tagad $LN_j=C$ vietā izveidosim jaunu mezglu (9. attēls).



Attēls 9. Jauna mezgla izveidošana.

Pievienosim pie izveidotā mezgla divus nepāra mezglus: B un C (10. attēls).



Attēls 10. Jauna mezgla veidošana.

Tādējādi, mēs ierobežojām maksimālu koda garumu līdz 4. Izmainot kodu garumus, mēs nedaudz pazaudējām efektivitāti. Rinda S izmērs, kas tiks nokodēta ar tādu kodu palīdzību, būs 92 biti, t.i. uz 3 vairāk salīdzinot ar minimāli-redundantu kodu.

Tātad, jo spēcīgāk mēs ierobežosim maksimālu koda garumu, jo mazāk efektīvs būs kods. Noskaidrosim uz cik bitiem var ierobežot maksimālu koda garumu. Acīmredzami ka ne īsāk, ka par $[\log_2 N]$.

2.4.1.5 Kanonisku kodu aprēķināšana

Kā jau tika atzīmēts iepriekš, pietiek ar kodu garumiem, lai saģenerēt pašus kodus. Tālāk tiks parādīts, kā to var izdarīt. Pieņemsim, ka mēs jau esam aprēķinājuši kodu garumus un zinām cik daudz kodu ar noteiktu garumu mums ir. L ir maksimāls koda garums, bet T_i - i garuma kodu skaits.

Aprēķināsim S_i – sākuma vērtība kodam ar garumu i, visiem i no [1..L].

$S_L = 0$ (vienmēr)

$S_{L-1} = (S_L + T_L) >> 1$

$S_{L-2} = (S_{L-1} + T_{L-1}) >> 1$

...

$S_1 = 1$ (vienmēr)

Mūsu piemērā: L = 5, $T_{1..5} = \{1, 0, 2, 3, 2\}$.

$S_5 = 00000_{bin} = 0_{dec}$

$S_4 = (S_5{=}0 + T_5{=}2) >> 1 = (00010_{bin} >> 1) = 0001_{bin} = 1_{dec}$

$S_3 = (S_4{=}1 + T_4{=}3) >> 1 = (0100_{bin} >> 1) = 010_{bin} = 2_{dec}$

$S_2 = (S_3{=}2 + T_3{=}2) >> 1 = (100_{bin} >> 1) = 10_{bin} = 2_{dec}$

$S_1 = (S_2{=}2 + T_2{=}0) >> 1 = (10_{bin} >> 1) = 1_{bin} = 1_{dec}$

Redzams, ka S_5, S_4, S_3, S_1 - ir simbolu B, A, C, H kodi. Šos simbolus apvieno tas, ka tie visi atrodas pirmajā vietā, katrs savējā līmenī. Citiem vārdiem, mēs atradām sākuma kodu priekš katra garuma (vai līmeņa).

Tagad sameklēsim kodus pārējiem simboliem. Pirmā simbola kods līmenī i ir vienāds ar S_i, otrā $S_i + 1$, trešā $S_i + 2$ un t. t.

Izrakstīsim palikušus kodus mūsu piemēram:

B $= S_5 = 00000_{bin}$	**A** $= S_4 = 0001_{bin}$	**C** $= S_3 = 010_{bin}$	**H** $= S_1 = 1_{bin}$
F $= S_5 + 1 = 00001_{bin}$	**D** $= S_4 + 1 = 0010_{bin}$	**E** $= S_3 + 1 = 011_{bin}$	
	G $= S_4 + 2 = 0011_{bin}$		

Mēs saņēmām precīzi tādus pašus kodus, kā uzbūvējot kanonisku Haffmana koku.

2.4.1.6 Kodu koka saglabāšana

Lai nokodēto īsziņu izdevās dekodēt, nepieciešams izmantot tādu pašu kodu koku, kāds tika izmantots kodēšanas laikā. Tādēļ, kopā ar iekodētajiem datiem mēs esam spiesti saglabāt atbilstošo kodu koku. Ir skaidri, ka jo kompaktāks viņš būs, jo labāk.

Atrisināt šo uzdevumu var dažādos veidos. Pats acīmredzamākais lēmums - saglabāt koku (tādu ka viņš ir, ar visiem mezgliem un norādēm). Tas ir pats sliktākais un neefektīvs veids. Praksē viņš netiek izmantots.

Var saglabāt sarakstu ar simbolu biežumiem (t.i. biežumu vārdnīca). Ar to palīdzību dekodēšanas laikā bez piepūles var rekonstruēt kodu koku. Kaut arī šis veids ir labāks par iepriekšējo, tas nav vislabākais.

Var izmantot vienu no kanonisku kodu īpašībām. Kā jau tika atzīmēts agrāk, kanoniski kodi tiek pilnīgi noteikti ar saviem garumiem. Citiem vārdiem, viss kas ir nepieciešams dekodēšanai - tas ir simbolu kodu garumu saraksts. Ņemot vērā, ka vidēji viena koda garumu N-simbolu alfabētam var nokodēt ar $[(\log_2(\log_2 N))]$ bitiem, saņemsim ļoti efektīvu algoritmu.

Pieņemsim, ka alfabēta izmērs N=256, un mēs saspiežam parastu teksta failu (ASCII). Visticamāk mūsu failā mēs neatradīsim visus alfabēta N simbolus. Tādā situācijā kodu garumi simboliem, kas nav tekstā, ir vienādi ar 0. Sarakstā, kurš saturēs visus kodu garumus, būs

pietiekami liels vienādu simbolu skaits(daudziem simboliem var būt viens un tas pats koda garums). Katru tādu grupu var saspiest ar tā saucamās grupās kodēšanas palīdzību - RLE (Run - Length - Encoding). Šis algoritms ir ārkārtīgi vienkāršs .

Šo metodi var arī paplašināt. Mēs varam pielietot RLE algoritmu ne tikai nulles garuma grupām, bet arī visām pārējam. Tāds kodu koka pārraides veids ir plaši pielietots mūsdienās.

2.4.2 Aritmētiskie kodi

Aritmētiski kodi parasti tiek pielietoti savienojumā ar statistiskas modelēšanas metodēm [5]. Simboli tiek kodēti saskaņā ar noteiktām varbūtībām.

Tā kā prefiksu kodi katru simbolu kodē ar atbilstošu bitu virkni, katra koda vārda garums ir vesels skaitlis. Tādēļ dažos gadījumos prefiksu kodi nav efektīvi (piemēram nelielam alfabētam vai kad varbūtības sadalījums nav vienmērīgs).

50 gadu beigās Gilberts un Murs piedāvāja alfabēta kodu ar redundanci: $R_{HM} < \frac{2}{N},$

kur N- iekodētu simbolu skaits, bet 60. gados tika piedāvāta aritmētiskas kodēšanas ideja ar redundanci: $R_{AC} < \frac{c}{N}$

kur N- iekodētu simbolu skaits, bet c- kaut kāda konstante.

Tāda veidā redundance tiecas uz 0, kad N→∞, tāpēc aritmētiski kodi ir optimāli no kodēšanas cenas viedokļa. Žēl, bet aritmētiski kodi klasiskajā veidā nevar būt realizēti praksē, jo tas prasa aritmētikas pielietošanu ar bezgalīgu precizitāti un neierobežotu atmiņas apjomu. Bieži tiek izmantotas metožu modifikācijas ar tuvinātu izrēķināšanu.

Aritmētiskai kodēšanai ir veltīts liels publikāciju skaits, ņemot vērā pietiekoši augstu algoritma būvēšanas sarežģītību un diezgan lielu visādu modifikāciju skaitu, šajā darbā tie netiek aprakstīti.

Izmantojot statistisku aritmētisku kodēšanu, no sākuma tiek izrēķināts biežums ar kuru parādās katrs simbols un pēc tām notiek kodēšana, kuras redundance apmierina sekojošu nevienādību:

$$\frac{2}{N} < R_{SA} < \frac{2 + |\Sigma| \log_2 N}{N}.$$

Izmantojot adaptīvu aritmētisku kodēšanu, no sākuma visiem simboliem tiek piešķirts svars w(a) = ε. Kārtējais simbols ir kodējams ar viņa pāradīšanās varbūtības novērtējuma palīdzību, vienādu ar

$$p'(a) = \frac{w(a)}{\sum w(\beta)}$$

pēc tam *w(a)* palielinās par vieninieku. Tādas kodēšanas redundance pie $\varepsilon \to 0$ apmierina nevienādību:

$$\frac{|\Sigma|}{N}(\varepsilon - \log_2 |\Sigma| - 1) < R_{AA} < \frac{|\Sigma|}{N}(\varepsilon - \log_2 |\Sigma| + \log_2 N).$$

Tā kā statiska kodēšana ar prefiksu kodiem ir novedama pie bitu rindu konkatenācijas, kuru nepieciešams veikt arī aritmētiskiem kodiem (kopā ar citām visai netriviālām darbībām), pie pietiekami lielā kodējamās īsziņas garuma aritmētiska kodēšana strādā lēnāk nekā kodēšana ar Haffmana kodu (pat ievērojot laiku, kas ir nepieciešams kodu koka uzbūvei).

Pēc autora un citu speciālistu domām, aritmētiskas kodēšanas pielietojums ne vienmēr attaisnots, tā kā pie pietiekami liela alfabēta izmēra |Å| > 100, saspiešanas kvalitāte parasti tikai uz 1-2% labāk, nekā prefiksu kodiem, tomēr ātrums ievērojami lēnāks.

Teksts, ko iegūst, saspiežot aritmētiskos datus, tiek aplūkots kā daļa, kurā katrs alfabēta burts tiek saistīts ar kādu no labās puses vaļēja intervāla [0;1) apakšintervālu. Avota tekstu var uzskatīt par burtisku daļas, kas izmanto skaitļošanas sistēmu, kurā katrs alfabēta burts tiek izmantots kā skaitlis, attēlojumu, bet vērtību, kas ar to saistītas, intervāls atkarīgs no šī burta sastopamības. Pirmais saspiestā teksta burts ("visnozīmīgākais" skaitlis) var tikt dekodēts atrodot burtu, kura pusintervāls ietver tekstu pārstāvošās daļas vērtību. Pēc kārtējā izejas teksta burta noteikšanas, daļa pārrēķinās, lai noteiktu nākošo. Tas notiek, atskaitot no daļas ar atrasto burtu saistīto apakšapgabalu, un dalot rezultātu ar tā pusintervāla platumu. Pēc šīs operācijas izdarīšanas var dekodēt nākošo burtu.

Kā aritmētiskās kodēšanas piemēru apskatīsim četru burtu alfabētu (A, B, C, D) ar varbūtībām (0,125; 0,125; 0,25; 0,5). Intervāls [0;1) var tikt sadalīts sekojoši:

A = [0, 0.125), B = [0.125, 0.25), C = [0.25, 0.5), D = [0.5, 1).

Intervāla dalīšana ir viegli veicama, izmantojot katra alfabēta burta un tā priekšgājēju varbūtības. Dots saspiests teksts 0,6 (desmitdaļas veidā), tad pirmajam tā burtam jābūt D, tāpēc ka šis skaitlis atrodas intervālā [0,5; 1). Pārrēķins dod rezultātu:

(0.6 - 0.5) / 0.5 = 0.2

Otrais burts būs B, tā kā jaunā daļa atrodas intervālā [0.125, 0.25).

Pārrēķins dod:

(0.2 - 0.125) / 0.125 = 0.6.

Tas nozīmē, ka trešais burts ir D, un sākotnējais teksts, ja nav dota informācija par tā garumu,

būs rinda DBDBDB...

Pati svarīgākā problēma šeit ir augstā aritmētiskas precizitāte saprašanai un operēšanai ar nepārtrauktu bitu plūsmu, kāds izskatās saspiestais teksts, ko apskata kā skaitli. Šī problēma tika atrisināta 1979. gadā. Saspiešanas efektivitāte, izmantojot statisko aritmētiskās kodēšanas metodi, būs vienāda ar H tikai izmantojot neierobežotas precizitātes aritmētiku. Bet arī ar galīgo vairuma mašīnu precizitāti ir pietiekoši, lai varētu veikt ļoti labu saspiešanu. Ar veseliem 16 bitu gariem mainīgajiem, 32-bitu reizināmajiem un dalāmajiem ir pietiekams, lai adaptīvas aritmētiskās saspiešanas rezultāts būtu dažu procentu attālumā no robežas un būtu gandrīz vienmēr nedaudz labāks, nekā optimālajam adaptīvajam Haffmana kodam.

2.5 Lokāli-adaptīva kodēšana

Kad avota statistika ļoti ātri mainās, tradicionālas entropijas kodēšanas metodes izmantošana nav efektīva. Tādos gadījumos tiek izmantotas lokāli-adaptīvas kodēšanas metodes [6] : pašorganizējošie koki˙ (selfadjusting) vai dažādi intervālu kodēšanas algoritmi: intervālu kodi, grāmatu kaudzes metode, modificētā intervālu kodēšana un citas metodes.

2.5.1 Intervāla kodēšana

Izmantojot intervāla kodēšanu, simbols tiek aizvietots ar ciparu ,kas ir attālums līdz tai vietai kodējamā simbolu virknē, kur šis simbols parādījās iepriekšējo reizi. Piemēram simbolu virkne (cba)aaabbbccca tiks pārveidota par (???)0004008006, dekodējot tādu rindu vajadzēs katru dekodētu ciparu i aizstāt ar simbolu, kas bija dekodēts i soļus atpakaļ.

Var ievērot, ka intervāla kodēšana (un citi tās varianti, kas ir aprakstīti tālāk) nevis samazina kodējamās rindas garumu, bet gan izmaina tās saturu, tātad precīzāk būtu saukt tādas kodēšanas metodes par pārveidojumiem.

Intervāla kodēšanas metodes tiek vienmēr izmantotas savienojumā ar entropijas kodēšanas metodēm.

2.5.2 Grāmatu kaudzes metode (MTF)

Grāmatu kaudzes metode ir viena no intervāla kodēšanas modifikācijām. Algoritma pamatā arī ir simbola aizvietojums ar ciparu. Katrs simbols kodējamā virknē tiek aizstāts ar ciparu i , kur i ir dažādu simbolu skaits, kas atrodas intervālā no ieejas simbola līdz tai vietai kodējamā simbolu virknē, kur šis simbols parādījās iepriekšējo reizi. Šis metodes autors ir B.J. Rjabko, kurš arī nosauca šo algoritmu par grāmatu kaudzes metodi, jo tā atgādina procesu kad cilvēks izņem grāmatu no kaudzes un liek virsu. Vienā brīdī sanāks tā, ka bieži izmantotas grāmatas būs augšā.

Vajadzētu pateikt, ka algoritms, kas ir funkcionāli ekvivalents grāmatu kaudzes metodei, sen ir zināms programmēšanā zem nosaukuma LRU (Last Recently Used) un iepriekš tika izmantots lai paātrināt datu meklēšanas procesu.

LFU (Last Frequently Used), Move-One-Up un Move - Half - Up, kuri pārvieto elementus secīgā masīvā, aizstājot simbolu ar viņa tekošā stāvokļa indeksu masīvā. Tāpat var tikt izmantotas priekš lokāli-adaptīvas kodēšanas, jo algoritmiem raksturīga laba saspiešanas kvalitāte.

2.6 Sēriju garumu vai grupveida kodēšana (RLE)

Lokāli-adaptīvu intervālu metožu pielietojums rāda virknes, kur viens un tas pats simbols atkārtojas vairākas reizes pēc kārtas. Kā secinājums, šīs metodes tiek pielietotas savienojumā ar sēriju garumu $\underbrace{a...a}_{N}$ kodēšanu (Run Length Encoding - RLE), kas aizstāj simbolu virkni ar pāri (a,N). [5].

3. Veiktie eksperimenti un iegūtie rezultāti

3.1 Problemātikas apraksts

Mūsdienās ir gan CD/DVD-rakstītāji, gan cietie diski ar lielu apjomu. Atmiņas apjoms jau izmērās ar desmitiem un pat ar simtiem gigabaitiem. Tomēr mazizmantojamas informācijas saspiešanas problēma paliek aktuāla. Arī pie lielas arhivātoru izvēles, ne visi no tiem strādā patiešām efektīvi. Dotajā nodaļā tiks aplūkotas un notestētas īpaši populāras arhivēšanas programmas un algoritmi. Nodaļas beigās būs aprakstītas rekomendācijas par to izvēli.

Arhivātori- tās ir programmas, kas ļauj būtiski samazināt faila izmērus, tādejādi atstājot arvien vairāk brīvās vietas cietajā diskā. Katrs no tiem darbojas savādāk nekā citi, tomēr, darbības principi visiem kopumā sakrīt: failos (to struktūrā) ir kaut kādi elementi, kas ik pa laikam atkārtojas, tādejādi veidojot līdzīgu elementu jeb simbolu virknes. Nav nekādas vajadzības turēt uz nodalījuma visus vienlaicīgi. Arhivātora uzdevums ir atrast visus šos elementus (fragmentus), kuri atkārtojas, un ierakstīt to vietā kāda cita rakstura informāciju, pēc kuras vēlāk varētu visus šos datus 100% atjaunot - atgriezt tiem to "izskatu", kāds bijis pirms uzsākot arhivēšanu.

Cik aktuāla ir arhivēšanas ideja un cik bieži ir pielietojami arhivēšanas algoritmi šodien? Cieto disku un operatīvas atmiņas apjomi parastiem lietotājiem pašlaik ir tik diženi, ka daudzi vienkārši nezin, kā izmantot tādu lielu datu glabātavas potenciālu. Vai Ir vērts tādā situācijā aizdomāties par arhivēšanas būtību un vispār to izmantot?

Kādus desmit gadus atpakaļ vidējais cietā diska izmērs bija 270Mb, lietotāji ļoti aktīvi pielietoja arhivēšanas programmas brīvo vietu palielināšanai, bet tas ne vienmēr palīdzēja. Tie bija smagi laiki. Bet arī pašlaik visi pielieto arhivēšanu, kaut arī problēmas ar atmiņas apjomu jau sen ir pagātnē. Tam ir daudz paskaidrojumu. Pārskaitīšu dažus no tiem:

- kad notiek failu pārraide pa pastu vai ar terminālu programmu palīdzību katrs kilobaits ir svarīgs. Ja pārsūtāmu failu skaits ir liels (īpaši ar sarežģīto katalogu struktūru), pārsūtīšanas vēstulē ir vienkāršāk ielikt tikai vienu failu – arhīvu.
- lokālu tīklu caurlaides spēja ir ierobežota. Liela informācijas apjoma pārsūtīšanai pa tīklu rekomendē izmantot arhivēšanu ne tikai trafika samazināšanai, bet arī lai vienkāršot galīgu salīdzināšanu. Pēc kopēšanas vienkāršāk ir pārbaudīt arhīva atbilstību, nekā pārbaudīt simtus failu, izskatot to saturu.
- rakstāmais CD-ROM kļuva gandrīz par standartu vidējam datoram. Daudziem no mums radās problēma, kad uz kompaktdisku nepieciešams pārnest ļoti sarežģītu katalogu struktūru, bet kompaktdiska failu sistēma neatļauj ierakstīt mapes, kurām faili atrodas

astoņkārtīgā ielikšanas līmenī. Tā ir ļoti nopietna problēma, ja nebūtu arhivēšanas, kas ļauj ielikt visu mapes struktūru vienā failā (tādiem mērķiem daži arhivātori izmanto saspiešanas metodi „bez saspiešanas”.

- daudzie arhivātori ir ļoti efektīvi kodētāji, kas ļauj noslēpt konfidenciālu informāciju no svešām acīm. Iepakojot failus tie pieliek arhīvam paroli.

No augstāk uzrakstītā var izdarīt vēl vienu svarīgu secinājumu: arhivēšana pašlaik tiek izmantota pārraides ātruma palielināšanai pie ierobežotas caurlaides spējas. Tas nesamazina saspiešanas algoritmu svarīgumu, bet palielina. Aprakstīti piemēri demonstrē šo kopīgu tendenci:

- modemu datu pārraidīšanas protokoli izmanto vienkāršus saspiešanas algoritmus(kas aizstāj vienādu baitu virknes „aaaaaabbbbbbbb” ar „6a8b”). Tas palielina failu pārraides ātrumu, nemainot sakara kanāla caurlaides spējas. Lai norādītu datu pārraidīšanas ātrumu tiek izmantoti divi neekvivalenti raksturojumi: fizisks kanāla raksturojums (signāla izmaiņu biežums) un loģisks raksturojums (bitu daudzums ko pārraida viena laika vienībā, b/s).
- šodien daudziem programmu izstrādātājiem rodas jautājums, kādā veidā nodrošināt pieklājīgu informācijas saspiešanu datu bāzēs. Ir acīmredzama nepieciešamība glabāt visu bāzi kompaktā, saspiestā variantā. Taču tai ļoti bieži ir diezgan liels apjoms, kas ir izmērāms ar simtiem megabaitiem. Pie tam vajag nodrošināt operatīvu piekļūšanu datiem un lielāku saspiešanas koeficentu. Optimālākais variants ir izstrādāt personīgo arhivēšanas sistēmu. Tādā gadījumā nepieciešams ņemt vērā dažas svarīgas datu īpatnības un iepriekš pārdomāt visus problēmas aspektus.
- spēles ir tādas programmas, kuram vajag visvairāk resursu. Īpaši 3D-spēlem. Mūsdienu videoatmiņas apjoms videokartēs ir no 32 līdz 256 megabaitiem, kas ir vajadzīgs, lai pa šūnām sūtīt lielu grafiskas informācijas apjomu. Ja tekstūram pielietot saspiešanu, pie viena un tā paša videoatmiņas apjoma var sasniegt daudz augstāku ražotspēju. Tekstūras saspiežas diezgan efektīvi. Tātad, palielinās to pārraides ātrums, un pieejamā atmiņa tiek izmantota efektīvāk.
- mūsdienās datora arhitektūrā ir divās vājas vietas. Tieši tas bremze datoru produktivitātes augšanu. Šo divu komponenšu dēļ, dubultojot procesora frekvenci nevar iegūt divkāršu visa datora produktivitātes palielināšanu. Divas komponentes par kurām iet runa – cietie diski un atmiņa. Procesoru produktivitāte piecu gadu laikā palielinājās 20 reizēs, ber atmiņas frekvence tikai piecās(no 66Mhz DIMM SDRAM PC66 līdz 333Mhz DDR PC2700). Tas nav tik slikti, bet sinhronizēšanas frekvence ar sistēmas šūnu neko nenosaka. Izmantojot papildus gaidīšanas ciklus, ražotājiem izdevās palielināt atmiņas

frekvenci, bet produktivitāte nav palielinājusies (par to liecina vairāki testi). Ar cietiem diskiem ir vēl sliktāk. Apjoms pieaug, bet ātrums gandrīz nemainās, pie visa ir vainīga diska mehāniska struktūra. Tāds svarīgs parametrs kā RPM (Rotate Per Minute – izpildāmu pagriezienu skaits viena laika vienībā) mainījies neievērojami. No 5400 tas palielinājās līdz 7200. Mūsdienās failu sistēmas atbalsta failu saspiešanu ar operētājsistēmas līdzekļiem. Datora produktivitāte mainījusies kardinālā veidā. Saspiešanas izmantošana failu sistēmas līmenī ne tikai nebremzē visu sistēmu, bet gluži otrādi, paātrina to.

Visi augstāk apskatītie piemēri secina par saspiešanas algoritmu aplūkošanas aktualitāti. Pēc autora domām, saspiešanas un informācijas kodēšanas algoritmu zināšana ir viens no elementiem, kas nosaka, ka cilvēks ir kompetents datorikas sfērā.

3.2 Uzdevuma nostādne

Gandrīz visi mūsdienas arhivātori un tajos izmantotie algoritmi prasa lielus datoru resursus. Algoritmu struktūra ir ļoti sarežģīta. Tas ir galvenokārt tāpēc, ka tie tiek pielietoti jebkuram datu tipam. Lielāka daļa no visiem algoritmiem un arhivātoriem ir nopatentēta un par to izmantošanu jāmaksā noteikta naudas summa. Visbiežāk saspiešana tiek pielietota arhivējot tekstisku informāciju: dati datu bāzes pārsvarā nesatur skaņas un attēlus, elektroniskajā pastā arī tiek izmantota tekstiska informācija īsziņu pārsūtīšanai. Vai ir vērts šādam nolūkam izmantot kaut kādu sarežģītu algoritmu, kas prasa daudz resursu un kurš dažreiz nav bezmaksas?

Tātad, uzdevuma prasības ir sekojošās:

- izvēlieties vienkāršu un viegli realizējamu saspiešanas algoritmu.
- uzrakstīt programmu-arhivātoru, kas realizē šo algoritmu vienā no mūsdienas programmēšanas valodām.
- pārbaudīt, vai jaunizveidotais arhivātors varēs konkurēt ar mūsdienas populārākiem arhivātoriem.
- veikt saspiešanas metodes analīzi, pārbaudīt cik labi un ātri ar to palīdzību var saspiest tekstisku informāciju:
    - dažādas grāmatas elektroniskajā formātā.
    - cipariskus failus (tie arī ir tekstieskie faili, kuru saturs pārsvarā sastāv no cipariem).
    - failus ar noteiktu struktūru, piemēram XML failus.

- pārbaudīt, kā tekstiskas informācijas alfabēta simbolu skaits var ietekmēt saspiešanas kvalitāti.
- pēc apkopotiem rezultātiem izdarīt secinājumus, izvēlieties un nodefinēt nākamu pētīšanas soļi un virzienu.

3.3 Izmantotā metode

Aplūkosim arhivēšanas (kodēšanas) metodi, kura tiek izmantota programmā-arhivātorā, ko izstrādāja darba autors:

Kodēšana strādā ar datu plūsmu kaut kādā alfabētā, tajā laikā simbolu frekvence ir dažāda. Arhivēšanas mērķis ir datu plūsmas pārveidošana uz bitu plūsmu ar minimālu garumu.To mēs varam panākt, samazinot datu plūsmas entropiju, izmantojot simbolu frekvenci: kodu garumam jābūt proporcionālam informācijai, kura ir ieejas plūsmā. Ja mēs zinām frekvences varbūtības sadalījumu, tad mēs varam iegūt optimālu kodēšanu. Uzdevums ir sarežģītāks, ja simbolu frekvences sadalījums iepriekš nav zināms. Šajā gadījumā tiks izmantota sekojoša pieeja: aplūkojam ieejas datu plūsmu un uzbūvējam kodēšanu pamatojoties uz savākto statistiku (mums ir jālasa dati, kuri atrodas failā, divas reizes, kas ierobežo pielietošanas sfēru šādiem algoritmiem). Tādā gadījumā kodēšanās shēmu, kas tika izmantota priekš saspiešanas, jāieraksta izejas datu plūsmā. Tā ir Haffmana statiska kodēšana. Vairāk par šo metodi var izlasīt 2. nodaļā.

Haffmana metodes priekšrocības ir pietekami augsts ātrums un laba saspiešanas kvalitāte. Šīs algoritms jau sen ir zināms un plaši pielietojams, piemēram tā ir programma Compress OS UNIX (programmas realizācija) un kodēšanas standarts faksiem (Hunter) (aparatūras realizācija).

Haffmana kodēšanai ir minimāla redundance, katrs simbols tiek kodēts ar atsevišķu simbolu virkni (koda virkne katram simbolam var saturēt tikai 0 un 1).

3.4 Arhivātors JFC6

Šīs arhivātors ar nosaukumu JFC6 ir programma, kuru izstrādāja darba autors. Programma paredzēta failu saspiešanai, izmantojot Haffmana algoritmu,

- Nosaukuma atšifrējums: Jurijs Fjodorovs Compression 2006(JFC6).
- Arhīva dokumenta paplašinājums: jfc.
- Programmas izstrādes vide ir Delphi7.
- Izpildāmais fails JFC6.exe aizņem 464Kb.

- Programmas izpildes laiks ir atkarīgs no datora resursiem (procesora un cietā diska ātrums, operatīvas atmiņas daudzums).
- Arhivātors paredzēts lietošanai Windows vidē.

3.4.1 JFC6 interfeisa apraksts

Ja faila izmērs ir ļoti mazs un viņš sastāv no dažādiem simboliem, tad var iznākt, ka kodēta faila izmērs ir lielāks pār sākotnēja faila izmēru. Šādas situācijas dēļ nav rekomendēts izmantot arhivātoru JFC6 failiem kuru izmērs iz mazāks par 50Kb.

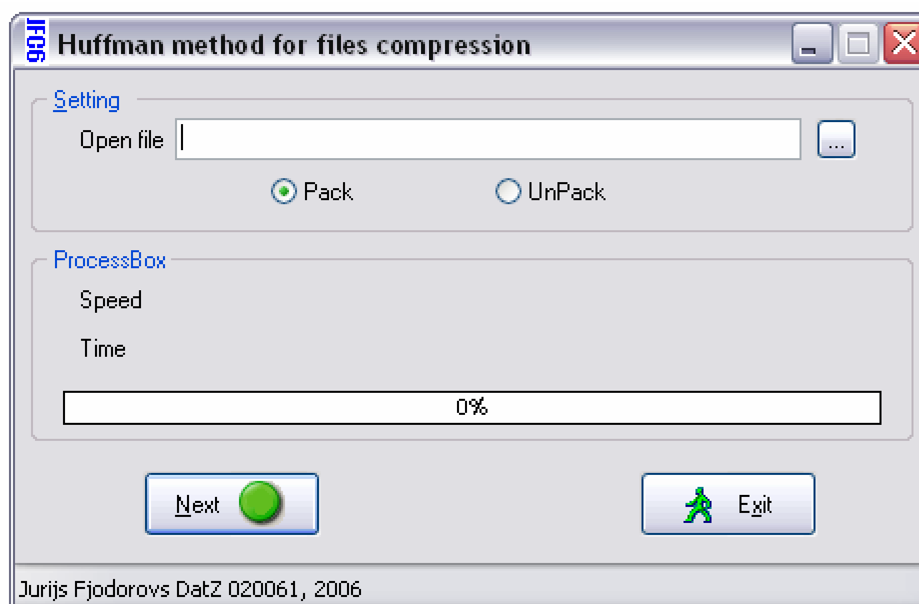
Programmas izpilde notiek ar faila JFC6.exe palaišanu. Atvērsies programmas pamatlogs (attēls 11). Pamatlogs ir sadalīts divās daļās :

- parametru konfigurēšana (Settings).
- algoritma darbības atspoguļošana (Process box).

Laukā „Open file" jāieraksta faila vārds, vai jāizvēlas fails nospiežot attiecīgu pogu (attēls 12). Nospiežot uz vienu no radiopogām var izvelēties darbības veidu:

- faila saspiešana (Pack).
- faila atspiešana (Unpack).

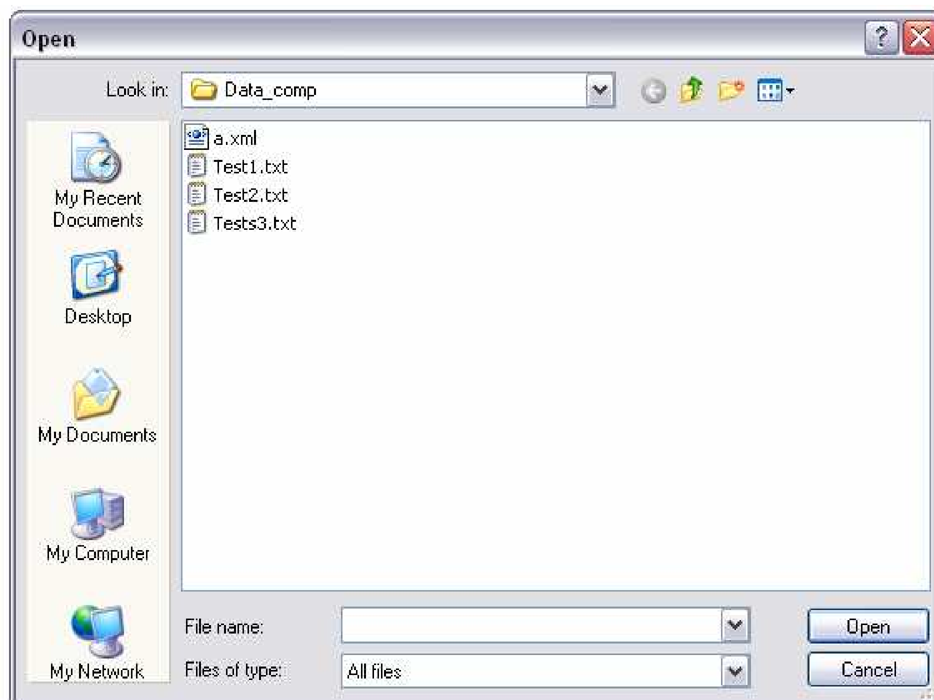
Pēc noklusēšanas ir izvēlēta saspiešanas darbība.



Attēls 11. Arhivātors JFC6.

Lai izpildīt darbību ir jānospiež „Next" poga. „Exit" - lai pabeigt darbu. Programmas darbības laikā saspiestais vai atspiestais fails tiks izveidots tajā pašā mapē, kur atrodas ieejas

fails. Saspiestam failam programma pievienos paplašinājumu ".jfc".

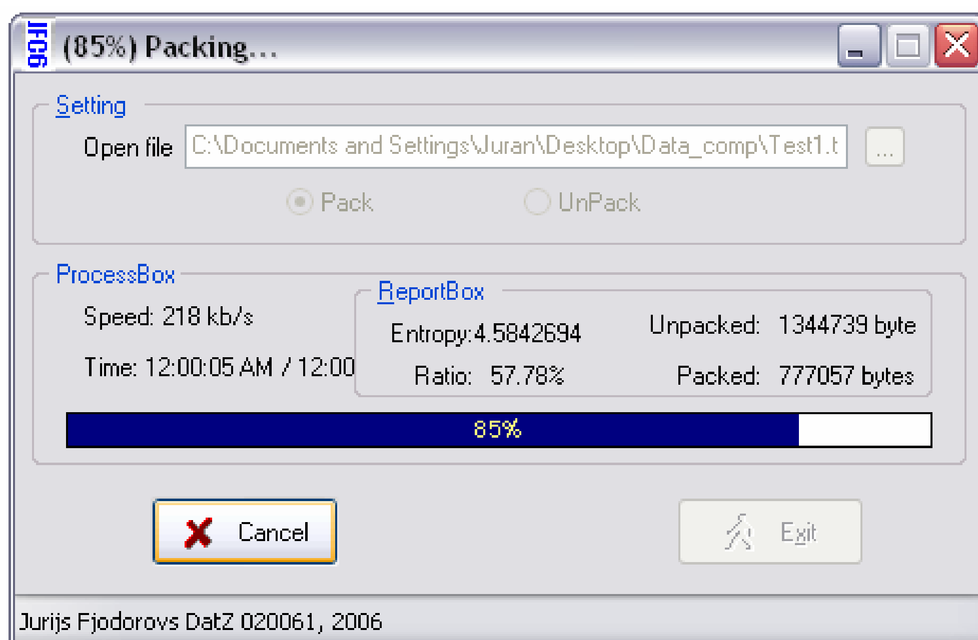


Attēls 12. JFC6 faila izvēle.

Ja programma tiks palaista bez parametra(faila vārds nav norādīts), vai no tastatūras tiks ievadīta neeksistējoša direktorija, parādīsies kļūdas paziņojums (13. attēls).

Attēls 13. JFC6 kļūdas paziņojums.

Ja ievaddati ir korekti, tad nospiežot pogu „Next" programma uzsāks faila saspiešanas vai atspiešanas darbību. Parādīsies atskaites lodziņš „Report box", kurā var apskatīt informāciju par faila izmēru pirms un pēc saspiešanas, ka arī entropiju un saspiešanas koeficientu (14. attēls).



Attēls 14. JFC6 arhivēšanas process.

Ja darbības laika ir nepieciešams apstādināt arhivātora darbību ir jānospiež poga „Cancel”. Pēc darbības pabeigšanas arhivēšanas procesa logs paliks aktīvs , lai varētu apskatīt saspiešanas rezultātus. Lai iziet no programmas ir jānospiež poga „Exit”

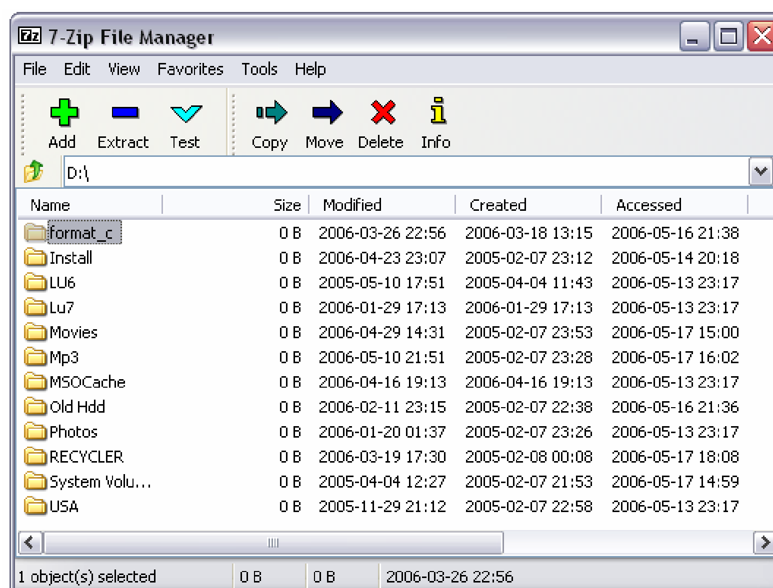
3.5 Arhivātoru un algoritmu testēšana un salīdzināšana

3.5.1 Arhivātoru saskarnes salīdzināšana

Testiem tika izvēlēti 6 arhivātori. Mūsdienās 4 populārākie: 7-Zip, WinAce, WinRar, WinZip. Dgca, kas salīdzinājumā ar pārējiem arhivātoriem nav tik populārs, tas izmanto BWT un aritmētisku kodu saspiešanas metodes. JFC6 ir autora izveidotais arhivātors, kas ir balstīts uz Haffmana algoritma. Tika salīdzināti arhivātoru saskarsmes plusi un mīnusi, kā arī arhivēšanas spējas. Ir veikta algoritmu analīze.

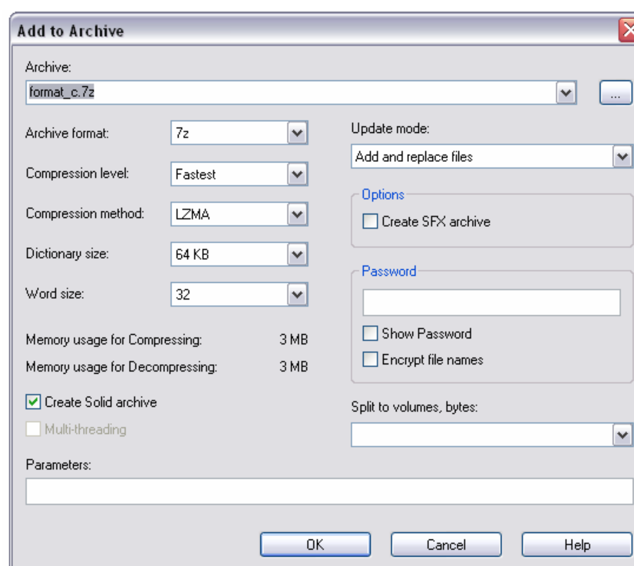
Sāksim ar nelielu saskarnes plusu un mīnusu uzskaiti katram no arhivātoriem.

7-Zip



Attēls 15. 7-Zip pamatlogs.

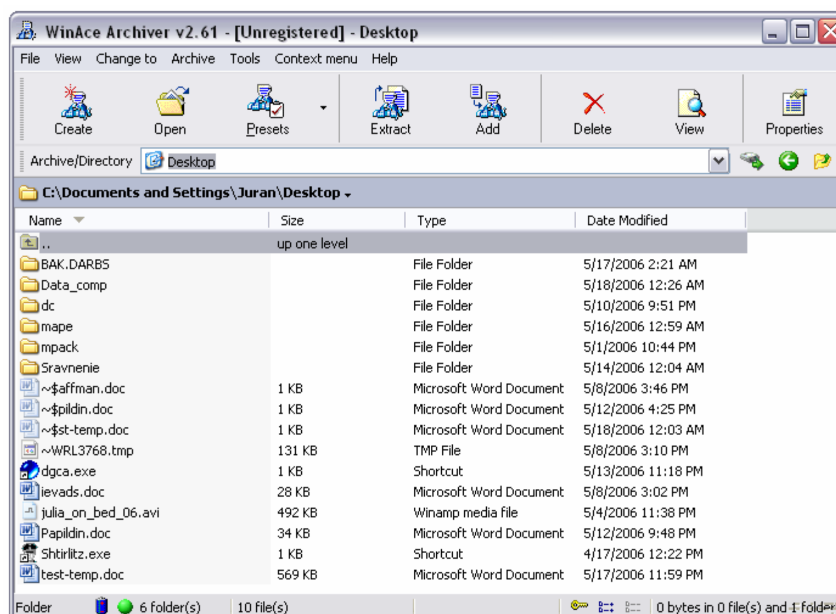
7-Zip [7] saskarne ir ļoti vienkārša un viegli uztverama. Uzreiz pieejamas galvenās funkcijas. Ja ieskatīties detalizētāk un 7-Zip saskarni salīdzināt ar WinRar saskarni, tad var secināt, ka tās ir ļoti līdzīgas. 7-Zip izstrādātājs nav centies izgudrot kaut ko jaunu, bet izmanto pārbaudītas lietas.



Attēls 16. 7-Zip pirmsarhivēšanas logs.

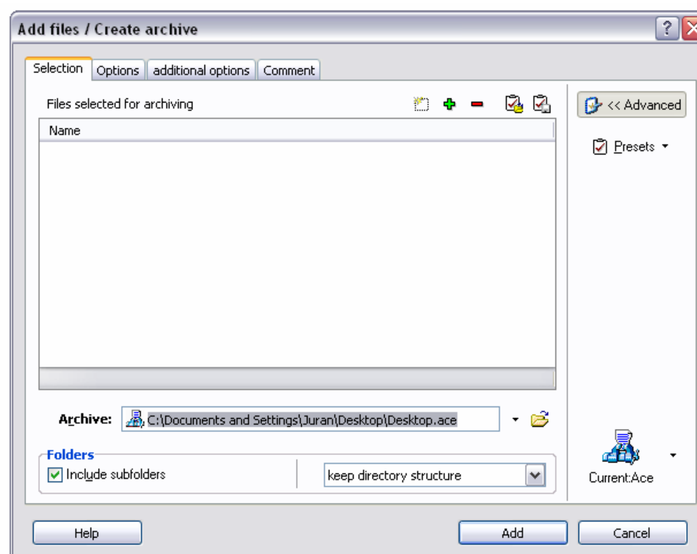
"Pirmsarhivēšanas logs" arī atgādina WinRar, tikai ar ievērojami mazāk opcijām. Par "pirmsarhivēšanas logu" tiek dēvēts logs, kurš parādās izvēlējoties arhivēt kādu konkrētu failu vai failu grupu. Lietotāja ērtumam pirmsarhivēšanas logā ir pieejamas galvenās opcijas: arhivēšanas metode, faila nosaukuma izvēle, iespēja veidot SFX arhīvu un citas.

WinAce



Attēls 17. WinAce pamatlogs.

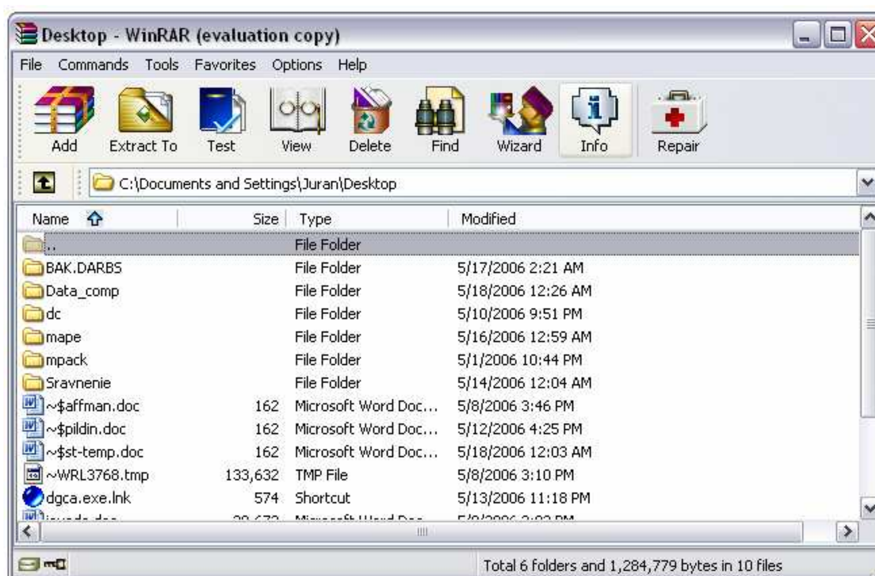
Līdzīgi, kā jau apskatītajam 7-Zip arī WinAce galvenais logs ir viegli uztverams un pārskatāms. Tam ir viegla navigācija. [9].



Attēls 18. WinAce pirmsarhivēšanas logs.

Pirmsarhivēšanas loga arhitektūra nav no veiksmīgākajām. Lai gan tā sevī apvieno daudzas labas īpašības. Iespēja izvēlēties faila saspiešanas metodi un citus parametrus ir pluss, bet ir arī vairāki mīnusi. Loga pirmajā sadaļā pieejamās opcijas ir pārāk mazā skaitā. Bieži nākas pārslēgties uz sadaļu „Options”, tātad - liekas darbības. Loga izmēri ir pārāk lieli.

WinRar



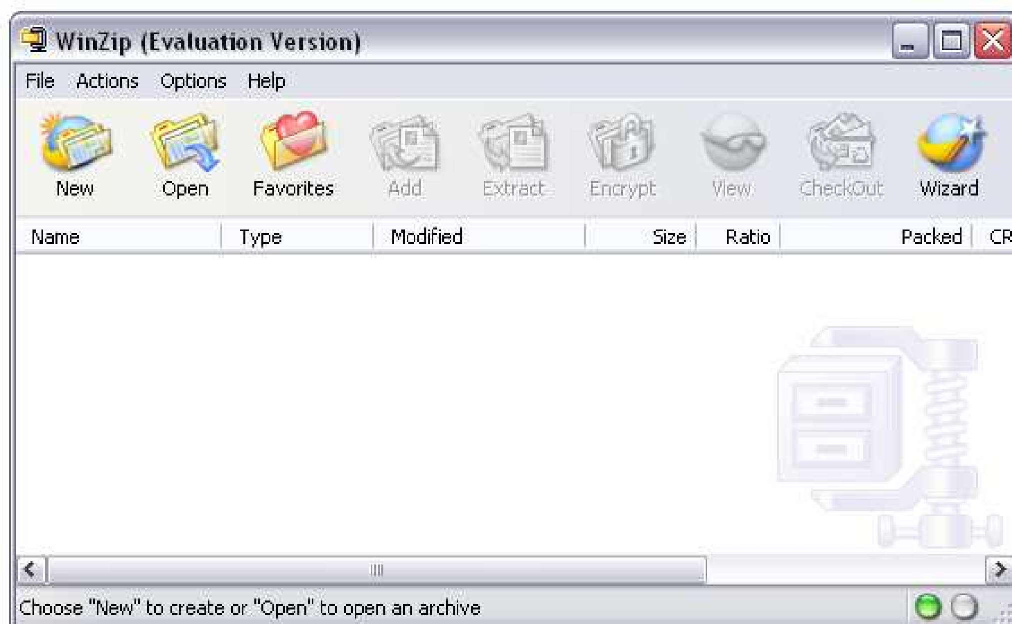
Attēls 19. WinRar pamatlogs.

Galvenais logs tāpat kā augstāk apskatītajiem arhivātoriem arī ir viegli uztverams. Ir pieejamas galvenās funkcijas un viegla navigācija. [10].

Attēls 20. WinRar pirmsarhivēšanas logs.

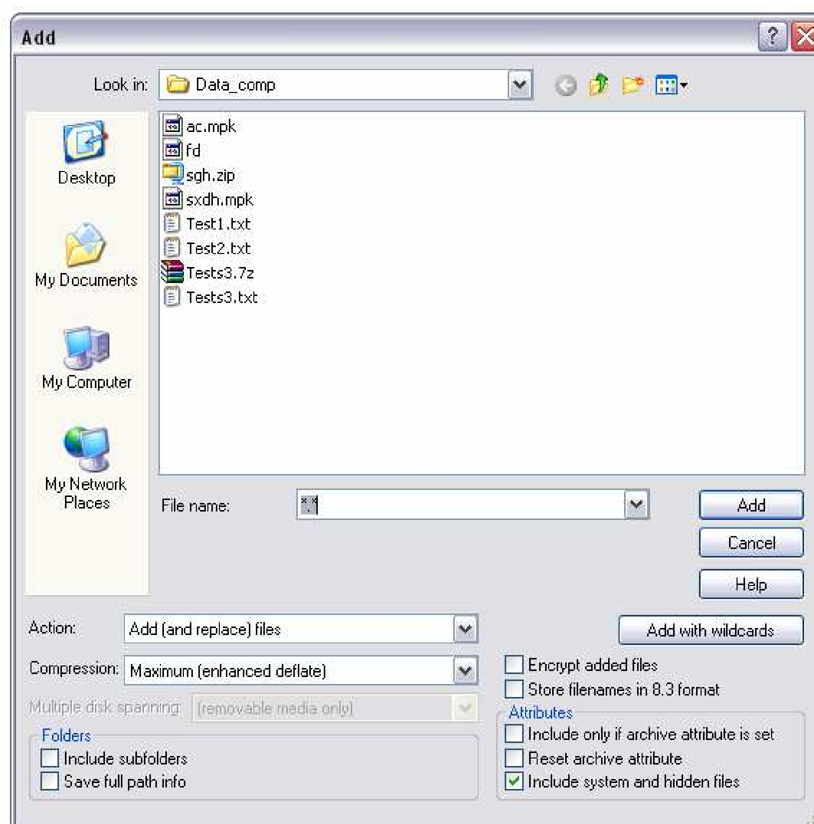
Pirmsarhivēšanas loga arhitektūra visveiksmīgākā. Tā sevī apvieno kompaktus izmērus, iespējas izvēlēties vairākas papildus opcijas. Turklāt, visbiežāk maināmie parametri ir pieejami loga pirmajā sadaļā.

WinZip



Attēls 21. WinZip pamatlogs.

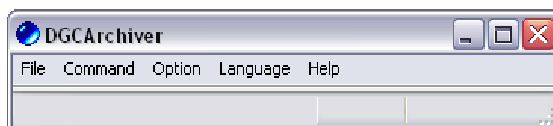
Pēc WinZip [11] startēšanas pirmajā mirklī nav saprotams kā un ko ir iespējams izdarīt. Nav failu koka un citu navigācijai svarīgu iespēju.



Attēls 22. WinZip pirmsarhivēšanas logs.

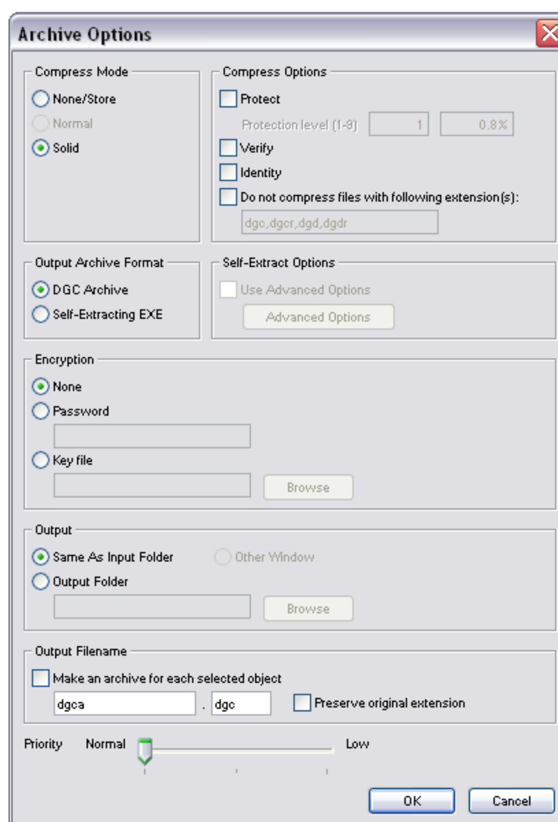
Pirmsarhivēšanas logs ir samērā viegli uztverams, taču neiekļauj sevī vairākas būtiskas īpašības, piemēram, SFX arhīva veidošanas iespēju.

Dgca



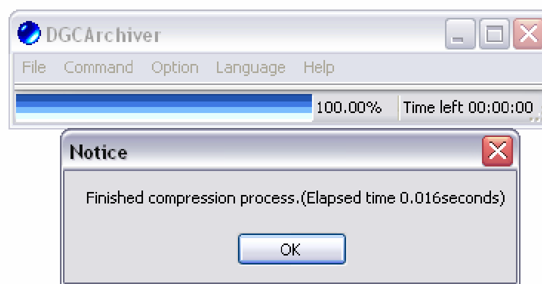
Attēls 23. Dgca pamatlogs.

Līdzīgi, kā WinZip gadījumā, palaižot Dgca [8] nevar uzreiz saprast kā un ko ir iespējams izdarīt ar šo arhivātoru.



Attēls 24. Dgca pirmsarhivēšanas logs.

Izvēloties failu priekš saspiešanas parādās konfigurāciju logs. Taču tas neiekļauj sevī vairākas būtiskas īpašības, piemēram, SFX arhīva veidošanas, ātruma un saspiešanas kvalitātes konfigurācijas iespējas.



Attēls 25. Dgca darbības beigšanas paziņojums.

Vienīgais no visiem arhivātoriem, kas pabeidzot darbību parāda precīzu saspiešanas un atspiešanas laiku līdz pat mīļi sekundēm.

3.5.1.1 Kopsavilkums par saskarnēm

Vislabāk izstrādāta saskarne priekš WinRar. WinZip, WinRar, WinAce un 7-Zip saskarņu arhitektūrā ir vairāki līdzīgi elementi, kuri sevi ir pierādījuši, kā ļoti viegli uztverami un izprotami pat lietotājiem, kuriem ar arhivātoriem vispār nav bijusi nekāda saskarsme. Uzsākot darbu ar WinZip ir jātērē ievērojams laiks, lai apgūtu arhīvu veidošanu. Kā iemesls ir grūti uztveramā saskarne.

Nr.	Nosaukums	Versija	Vide	Izmērs	Iespējas	Izplatīšana	Algoritmi
1	WinAce	2.61i	Win32	3.61 Mb	sl	bezmaksas	LZ77+H
2	7Zip	4.32-x64	Win32	1.08 Mb	a, vl, vg, sl	bezmaksas	LZMA+H PPM BWT+H
3	dgca	v1.09	Win32	1.19 Mb	ls, sk	bezmaksas	BWT+AK
4	WinZip	9.0	Win32	2.31 Mb	sl, sk	bezmaksas uz laiku, cena~29$	filtri un visādu algoritmu apvienojumi
5	WinRar	3.40	Win32	1.10 Mb	sl, sk	bezmaksas uz laiku, cena~29$	filtri un visādu algoritmu apvienojumi
6	JFC6	1.0	Win32	220 Kb	sk	bezmaksas	H

Tabula 1. Arhivātoru parametri.

a	Algoritma izvēle
vl	Vārdnīcas lieluma izvēle
vg	Vārda garuma izvēle
ls	Laika skaitītājs
sl	Saspiešanas līmeņa izvēle(Maximum, Norma, Fast, Super Fast)
sk	Saspiešanas koeficienta radītājs
H	Haffmana algoritms
AK	Aritmētiskie kodi
	Vislabākā saspiešanas kvalitāte
	Optimālākais algoritms
	Vissliktākais rādītais

Tabula 2. Izmantotu apzīmējumu paskaidrojums.

3.5.2 Vispārīgā testēšana

Vispārīgas testēšanas mērķis ir izpētīt, kā jaunizveidotais arhivātors JFC6 spēj konkurēt ar mūsdienas arhivātoriem, kā arī pārbaudīt Haffmana metodes efektivitāti tekstisku failu saspiešanai. Priekš šīs testēšanas tika izvelēti trīs patvaļīgi tekstiski faili. Failu apraksts ir apkopots 3. tabulā.

Nr.	Nosaukums	Formāts	Izmērs	Apraksts
1	Tests1	txt	1.28 Mb(1347584 b)	Grāmata „Trīs musketieri" A.Duma. Teksts angļu valodā.
2.	Tests2	doc	353Kb(360448 b)	Kursa darbs informātikā. Teksts latviešu valodā.
3.	Tests3	txt	2.86Mb(3006484 b)	Referāts. Teksts angļu valodā.

Tabula 3. Vispārīgas testēšanas failu apraksts.

Testēšana tika veikta ar sekojošu datoru:

- procesors - Intel Celeron 2700 Mhz
- operatīva atmiņa - DDR 512 Mb
- cietā ciska apjoms - HDD 160 Gb
- video karte - nVidia GeForce 128 Mb
- operētājsistēma – Windows XP

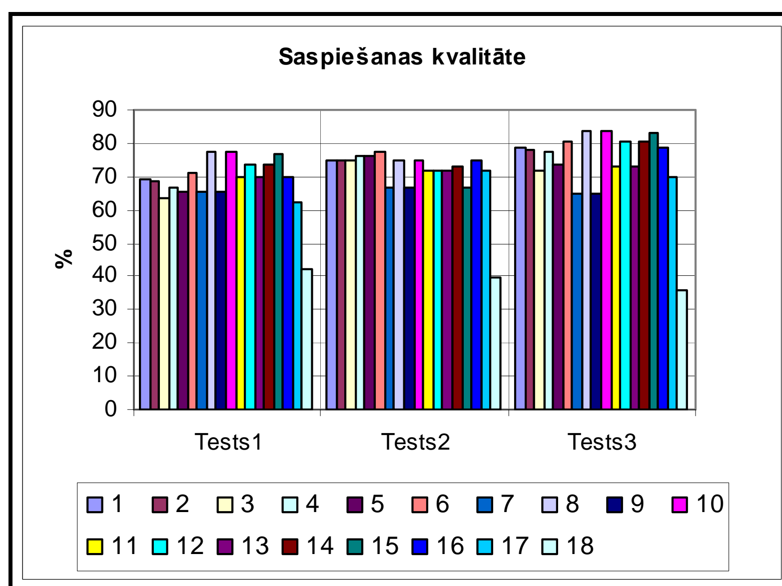
Iegūti testēšanas rezultāti ir apkopoti 4. tabulā.

Algoritmu un arhivātoru vispārīgas testēšanas rezultātu tabula												
Nr.	Arhivātors	Konfigurācija	Izmantota metode	Saspiešanas laiks (s)			Atspiešanas laiks (s)			Saspiešanas koeficients (%)		
Testētā faila numurs:				Tests1	Tests2	Tests3	Tests1	Tests2	Tests3	Tests1	Tests2	Tests3
1	WinAce	maximum	LZ77+H	3	1	4	0.5	0.5	0.5	69.3	75	78.6
2	WinAce	super fast	LZ77+H	1.5	0.8	3	0.5	0.5	0.5	68.7	75	78
3	7Zip	fastest LZMA Dictionary 64kb Word size-8	LZMA+H	1	0.5	1	0.5	0.5	0.5	63.5	75	71.8
4	7Zip	fastest LZMA Dictionary 32Mb Word size 64	LZMA+H	1	0.5	3	0.5	0.5	0.2	66.6	76	77.7
5	7Zip	maximumLZMA Dictionary 64kb Word size 8	LZMA+H	2	0.5	2	0.5	0.5	0.5	65.3	76	73.6
6	7Zip	maximumLZMA Dictionary 32Mb Word size 64	LZMA+H	3.5	1	6	0.5	0.5	1	71.1	77.3	80.8
7	7Zip	fastest PPMd Dictionary 1Mb Word size 2	PPM	1	0.5	1	0.5	0.5	1	65.7	67	64.8
8	7Zip	fastest PPMd Dictionary 32Mb Word size 16	PPM	1.5	0.5	2.5	1.5	0.5	2.5	77.2	75	84
9	7Zip	maximumPPMd Dictionary 1Mb Word size-2	PPM	1	0.5	1	1	0.5	1	65.7	67	64.8
10	7Zip	MaximumPPMd Dictionary 32Mb Word size-16	PPM	2.5	0.5	2.5	2.5	0.5	2.5	77.2	75	84
11	7Zip	fastest Bzip2 Dictionary 100kb	BWT+H	1.5	0.5	2	1	0.5	1	69.9	71.6	73
12	7Zip	fastest Bzip2 Dictionary 900kb	BWT+H	2.5	0.5	4	1	0.5	1	73.9	71.6	80.7
13	7Zip	maximumBzip2 Dictionary 100kb	BWT+H	2.5 *	1	5	1	0.5	1	69.9	71.6	73
14	7Zip	fastest Bzip2 Dictionary 900kb	BWT+H	3.5	1	9	1	0.5	1	73.9	73	80.7
15	dgca	standart	BWT+AK	1.7	0.4	4.8	0.6	0.1	1.1	76.6	67	83
16	WinRar	normal		2	0.2	3	0.5	0.2	0.2	69.9	75	78.9
17	WinZip	normal		1	0.2	1	5	0.2	0.2	62.6	71.6	70
18	JFC6	standart	H	5	1	13	43	10	95	42.3	39.7	35.8

Tabula 4. Vispārīgās testēšanas rezultāti.

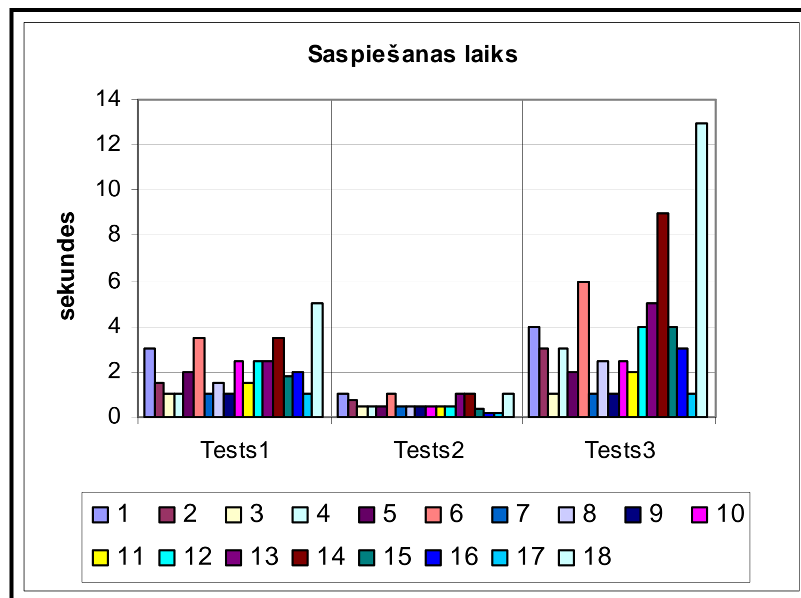
3.5.2.1 Testu kopsavilkums.

Veicot arhivātoru testus tika novērots, ka mainot arhivēšanas vārdnīcas (Dictionary) un vārda (Word) lielumus var ietekmēt arhivēšanas kvalitāti. Palielinot vārdnīcas izmēru un vārda lielumu nepieciešams lielāks atmiņas apjoms un vairāk laika arhivēšanas veikšanai, taču tiek iegūti labāki rezultāti. Samainot vārda lielumu no 8 uz 64 un vārdnīcas izmēru no 64Kb uz 32Mb arhivēšanai nepieciešamais laiks var palielināties pat trīs reizes un vairāk. Jo lielāks ir datu apjoms, jo krietni palielināsies arhivēšanas laiks un pieprasāmas atmiņas daudzums. Tests tika veikts ar 7-Zip un arhivēšanas metodēm LZMA, PPM kuros varēja novērot vislielāko vārdnīcas un vārda lieluma ietekmi. Līdzīgu situāciju var ievērot apskatot metodi Bzip2, kur mēs maināmtikai vārdnīcas lielumu. Saspiešanas kvalitātes rezultātus var apskatīt zemāk (26. attēlā).



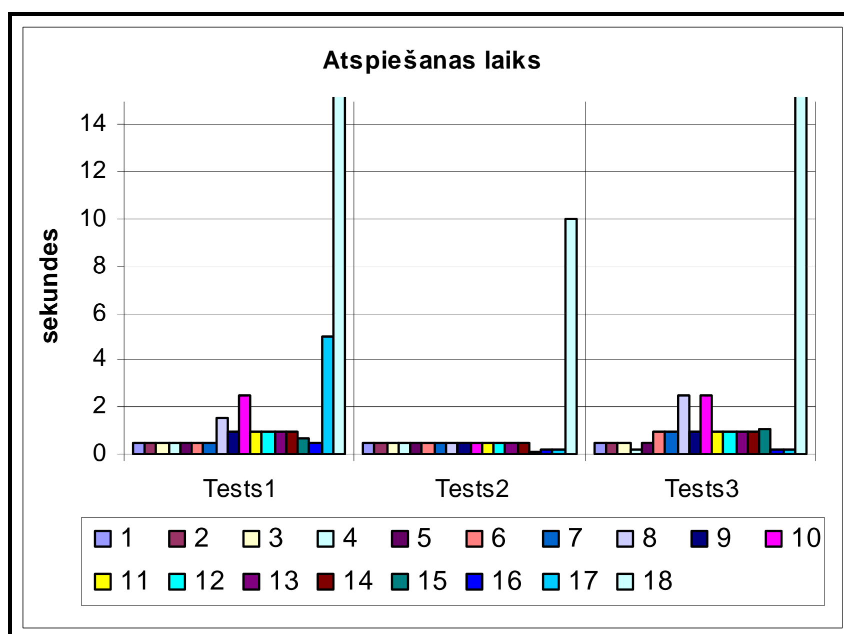
Attēls 26. Saspiešanas koeficients.

Attēlā varat redzēt, kā mainās noarhivētā faila lielums mainot vārdnīcas izmēru un vārda lielumu. Aplūkojot kolonas 3 un 5, 6 un 8, 9 un 11 (attēls 26) var ieraudzīt, ka pirmās kolonas ir mazākās par otrām. Tas nozīme, ka saspiešana ir labāka tad, kad vārdnīcas un vārda lielumi ir lielāki. Mainot arhivēšanas ātrumu, mainās arī saspiešanas kvalitāte. Aplūkojot kolonas 3un 4, 5 un 6, 8un 9 (attēls 26) var ieraudzīt, ka otras kolonas ir lielākas par pirmām. Tas nozīme, ka tērējot vairāk laika tiek panākta labāka saspiešanas kvalitāte. Nepieciešama laika izmaiņas var apskatīt 27. attēlā, salīdzinot kolonas līdzīgā veidā.



Attēls 27. Saspiešanas laiks.

28. attēlā ir parādīts, ka mainās atspiešanas laiks, mainoties datu apjomam. Salīdzinot attiecīgas kolonas no 27. un 28. attēliem var ievērot interesantu faktu, ka saspiešanas laiks ir gandrīz piecas reizes lielāks nekā atspiešanas laiks. To var paskaidrot sekojoši: saspiešana laika notiek kodējamas informācijas analīze un modelēšana, lai datu saspiešana būtu vislabākā. Bet atspiešanas laika modelis jau ir zināms un ir skaidrs, kā dekodēt datus. Līdz ar to atspiešanai ir nepieciešams mazāk laika.



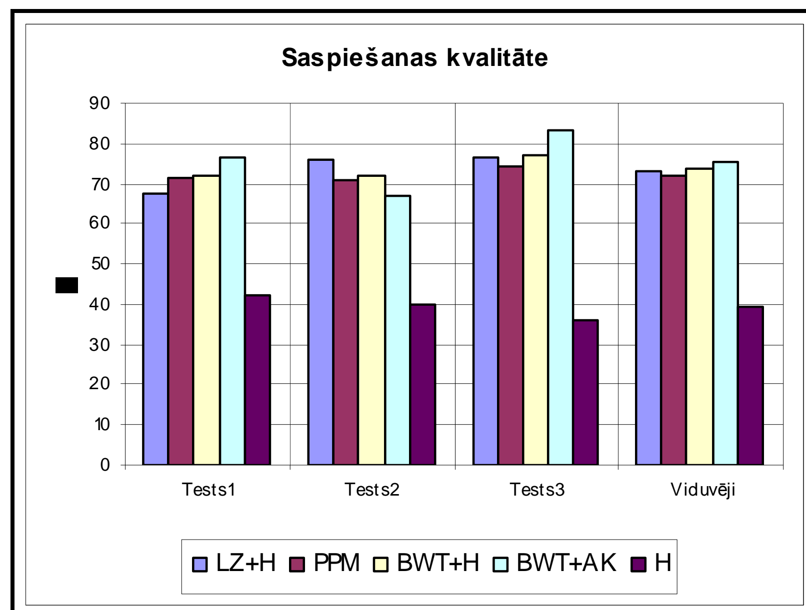
Attēls 28. Atspiešanas laiks.

Labākais arhivātors

Ir grūti pēc trīs testiem secināt, kāds no izmantotiem arhivātoriem ir pats labākais. Bet testi parādīja ka vislabāk un visātrāk dokumentus saspiež 7Zip arhivātors. Jo lielāks datu apjoms, jo labāku saspiešanu rāda 7zip. Šis ir viens no labākiem arhivātoriem, kurš ir pieejams bezmaksas. Ir vērts atzīmēt, ka mūsdienās vispopulārākie un visizplatītākie arhivātori WinZip un WinRar nevienā no testiem neparādīja labākus rezultātus. Pie kam tie ir maksas programmas, kuru cenas ir apmēram 29$.

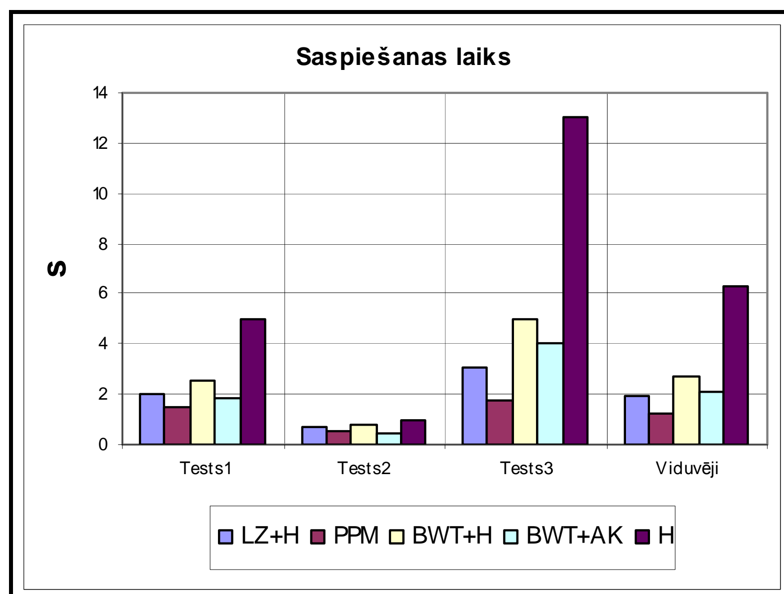
Saspiešanas algoritmi

29. attēlā ir parādīts, kā atšķiras saspiešanas kvalitāte katram no algoritmiem. Aplūkojot testu rezultātus var konstatēt, ka pirmā un otrā testa BWT algoritms ir labākais. Tomēr ja paņemt un salīdzināt vidējus rezultātus no visiem testiem var secināt, ka BWT, LZ un PPM algoritmiem ir gandrīz vienāda saspiešanas kvalitāte – virs 70%. Haffmana algoritma gadījumā tā ir mazāka par 40%.



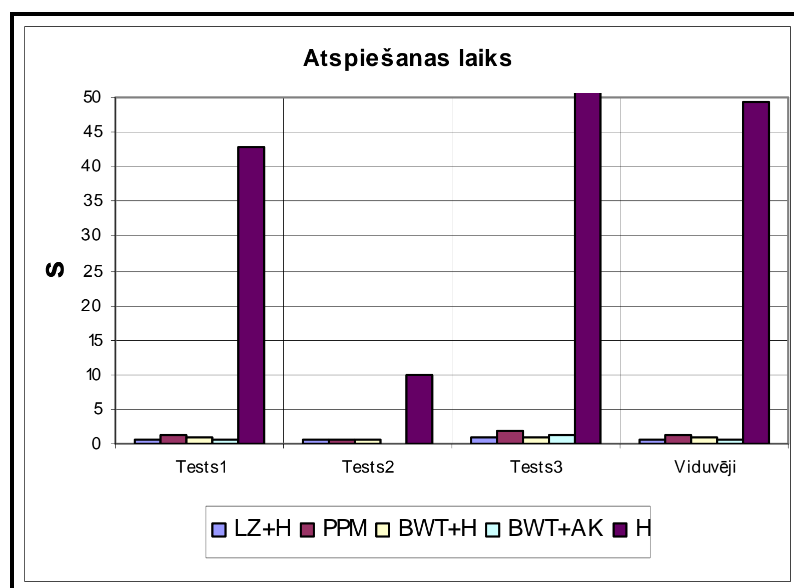
Attēls 29. Saspiešanas kvalitāte.

Saspiešanas ātrums ir atkarīgs no datu apjoma. Jo lielāks datu apjoms, jo ilgāks laiks nepieciešanas saspiešanas veikšanai. Aplūkojot Grafiku 30. attēlu, var pateikt, ka visātrākais ir PPM algoritms – vidējais saspiešanas laiks ir mazāks par 2 sekundēm. Laikietilpīgākais algoritms ir Haffmana kods. Vidējais rezultāts tam ir apmēram 6 sekundes. Tas aptuveni trīs reizes sliktāk nekā pārējiem algoritmiem.



Attēls 30. Saspiešanas laiks.

31. attēlā ir atspoguļota atspiešana laika salīdzināšana. Neskatoties uz to, ka saspiešanas laiks visiem algoritmiem ir dažāds, atspiešanas laiks LZ, BWT, PPM ir gandrīz līdzīgs un ir mazāks par vienu sekundi. Situācija ar Haffmana algoritmu ir gandrīz dramatiska, atspiešanas laiks ir gandrīz astoņas reizēs lielāks par saspiešanas laiku. Tas savukārt ir ievērojami lielāks par citu algoritmu laikiem.



Attēls 31. Atspiešanas laiks.

Kā secinājums, Haffmana algoritma saspiešanas kvalitāte un ātrums krietni atšķiras no citiem apskatītājiem algoritmiem un tie ir pusotras vai pat divas reizēs sliktāki, nekā pārējiem algoritmiem. Ir vērts atzīmēt, jo vairāk ir vienādu simbolu kodējamā tekstā, jo lielāku

saspiešanas kvalitāti un ātrumu izdosies panākt. To var paskaidrot sekojoši: vairākus vienādus simbolus izdosies nokodēt ar īsākiem kodiem un kodu vārdnīca nebūs tik liela. Ka arī kodu koks būs īsāks, un koka apstaigāšana neaizņems tik daudz laika. Ir nepieciešams atzīmēt, ka neviens no salīdzinātiem algoritmiem netika izmantots vienveidīgi, izņemot Haffmana algoritmu. T.i. viņi visi saturēja vienu algoritmu priekš datu modelēšanas un vienu papildus algoritmu priekš datu saspiešanas. Daži no tiem priekš modelētiem datiem izmantoja tieši Haffmana algoritmu. Tādejādi var secināt, ka arhivēšanas procesa sadalīšana divos etapos, t.i. modelēšana un kodēšana, ir ļoti nozīmīga un dod iespēju daudz labāk un pat ātrāk saspiest datus. Gadījumos, kad priekš informācijas saspiešanas (pēc tās modelēšanas) tika izvēlēts Haffmana algoritms, tas parādīja labus rezultātus.

3.5.3 Speciālu gadījumu testēšana

Iepriekšējā nodaļā tika apskatīta vispārīga testēšana. Tika salīdzināts, kā jaunizveidotais arhivātors JFC6 spēj konkurēt ar mūsdienas populārākiem arhivātoriem. Šīs nodaļas mērķis ir izpētīt, kā ar Haffmana algoritmu var saspiest dažādu tekstisku informāciju, kādiem datiem šis algoritms parādīs efektīvāku saspiešanu. Sadalīsim šo eksperimentu uz trīs pamatblokiem:

- XML failu saspiešana.
- ciparisku failu saspiešana .
- saspiešanas efektivitātes atkarība no izvelētas valodas.

Līdzīgi, kā iepriekšējā nodaļā, testēšana tika veikta ar tādiem pašiem arhivātoriem, mainot vārda garumu un vārdnīcas lielumu. Tika izrēķināti vidēji testēšanas rezultāti priekš katra algoritma, uz šis informācijas pamata izveidoti grafiki un tabulas.

3.5.3.1 XML failu saspiešana

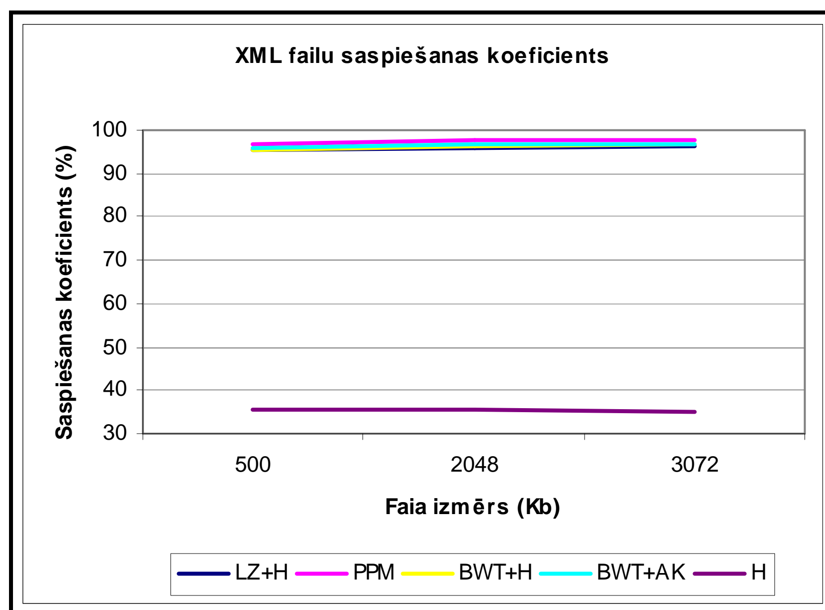
XML failu saspiešana ir ļoti aktuāla mūsdienas, jo informācijas pārmaiņa starp attālinātiem datu bāzēm notiek tieši ar šī tipa failu palīdzību. Priekš šī testa tika izveidoti trīs XML faili ar izmēriem 500Kb, 2Mb un 3Mb. 5.tabulā ir apkopoti iegūti saspiešanas rezultāti

XML failu saspiešanas rezultāti.									
Parametrs	Saspiešanas koeficients (%)			Saspiešanas laiks (s)			Atspiešanas laiks(s)		
Faila izmērs(Kb) / Algoritms	500	2048	3072	500	2048	3072	500	2048	3072
LZ+H	95.2	95.9	96.1	0.1	1	1.5	0.1	0.2	0.2
PPM	96.8	97.5	97.5	0.1	1	1.4	0.1	0.2	1
BWT+H	95.2	96.3	96.9	1	4	6	0.2	0.2	0.2
BWT+AK	96	96.7	96.9	0.3	1	2	0.1	0.2	0.2
H	35.3	35.5	35.27	3	8	14	12	53	110

.Tabula 5. XML failu saspiešanas rezultāti.

No tabulas ir redzams, ka algoritmiem LZ, PPM un BWT saspiešanas koeficients un atspiešanas laiks ir gandrīz vienāds. BWT algoritms tērē gandrīz trīs reizēs vairāk laika faila saspiešanai, nekā LZ un PPM algoritmi. Haffmana algoritms joprojām rāda nelielu saspiešanas koeficientu ~ 35%, kā arī sliktu saspiešanas un atspiešanas laiku, kas lineāri aug, mainoties faila izmēram. Interesants fakts, ka mainoties faila izmēram saspiešanas koeficients gandrīz nemainās. Tas ir tāpēc, ka XML failam ir noteikta struktūra, kas nemainās, palielinot faila izmēru. Tāpēc saspiešanas koeficients ir stabils, bet joprojām neliels. Tabulas 5 informācija ir atspoguļota attēlos 32, 33, 34 un 35, kas ilustrē saspiešanas koeficientu un laiku salīdzināšanu XML failiem.

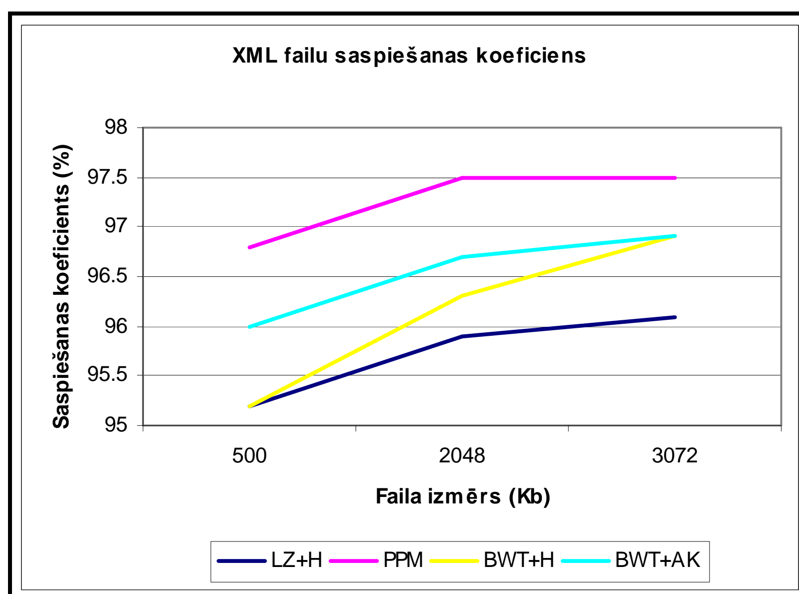
Saspiešanas koeficients algoritmiem LZ, PPM un BWT krietni atšķiras no Haffmana algoritma. Attēlā 32 var ieraudzīt, cik tālu ir Haffmana algoritms no mūsdienas saspiešanas metodēm.



Attēls 32. XML failu saspiešanas koeficients.

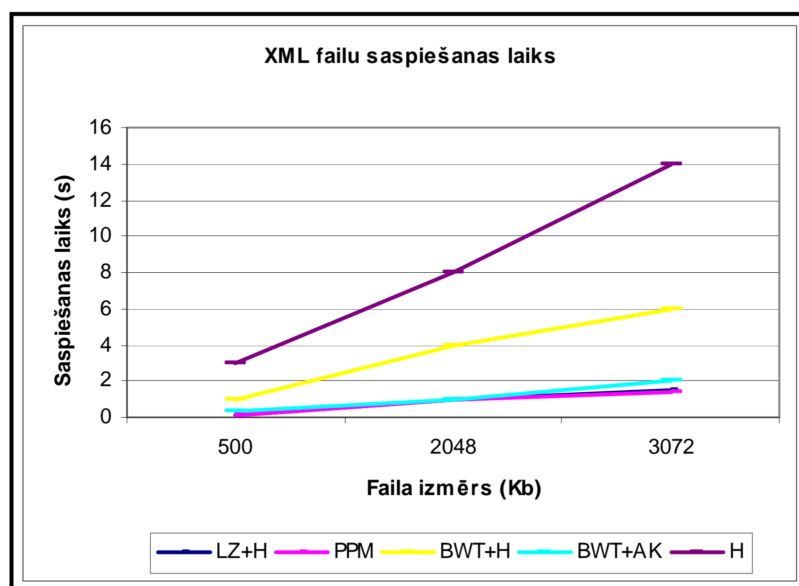
Attēlā 33 var apskatīt, kuram algoritmam ir labākais saspiešanas koeficients un kā tas

mainās, palielinot faila izmēru. Visiem algoritmiem, kas ir atspoguļoti attēlā 33 ir labs saspiešanas koeficients. Tas ir lielāks par 95%. PPM algoritms parādīja labāku rezultātu, pie kam šis rezultāts tika sasniegts jau failam ar izmēru 2Mb un pēc tam nemainījās.



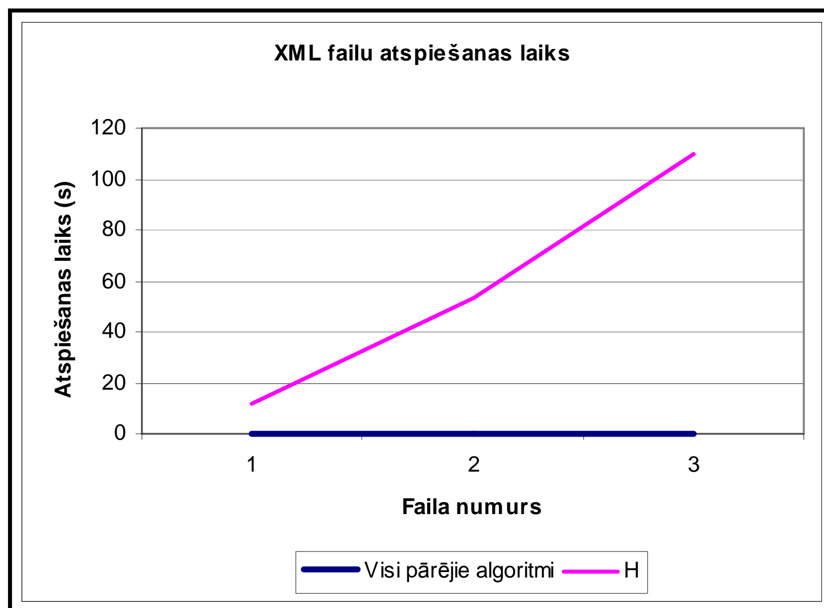
Attēls 33. XML failu saspiešanas koeficients.

LZ algoritma saspiešanas koeficients ir par 1% mazāks, nekā citiem algoritmiem. Tomēr laikā ziņā tas ir tik pat labs, kā PPM algoritms. Attēlā 34 ir atspoguļots, cik daudz laika tērē katrs algoritms XML faila saspiešanai. Visātrākā saspiešana ir raksturīga PPM un LZ algoritmiem. Nedaudz vairāk laika nepieciešamas BWT algoritmam, kurš izmanto aritmētiskus kodus. Visiem algoritmiem saspiešanas laiks lineāri pieaug, mainoties faila izmēram.



Attēls 34. XML failu saspiešanas laiks.

Atspiešanas laikā ziņā LZ, PPM un BWT algoritmi paradīja vienādus rezultātus(attēls 35). Atspiešanas laiks ir mazāks par vienu sekundi, tas tiešam ir labs rezultāts. Haffmana algoritma gadījumā atspiešanas laiks pieaug straujāk, nekā saspiešanas laiks.



Attēls 35. XML failu atspiešanas laiks.

Apkopojot iegūtos rezultātus var izdarīt secinājumu, ka PPM un LZ dzimtas algoritmi ir labāk piemēroti XML faila saspiešanai. Tiem raksturīgs labs saspiešanas koeficients, kas ir lielāks par 95%, labi saspiešanas un atspiešanas radītāji.

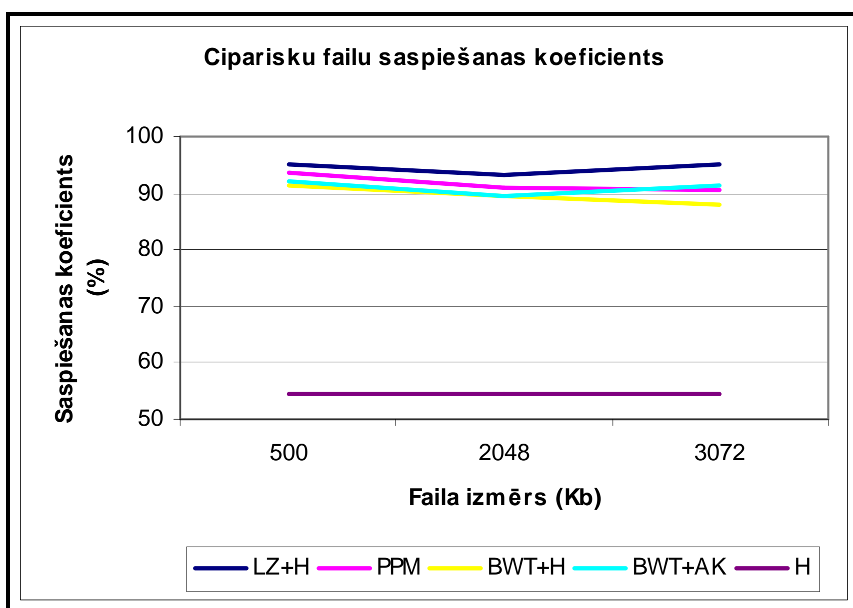
3.5.3.2 Ciparisku failu saspiešana

Cipariskas informācijas saspiešanas ir tik pat aktuāla, kā XML failu gadījumā. Lielajās datu bāzēs ir pietiekami daudz cipariskas informācijas, visi dati pārsvarā tiek glabāti pēc „ID” atslēgas, kas daudzos gadījumos ir „integer” tipa lauks. Kā arī personu apliecinoši kodi, tālruņu numuru, visādas naudas summas un t. t. Priekš šī testa tika noģenerēti trīs cipariskie faili ar izmēriem 500Kb, 2Mb un 3Mb. 6. tabulā ir apkopoti iegūtie saspiešanas rezultāti.

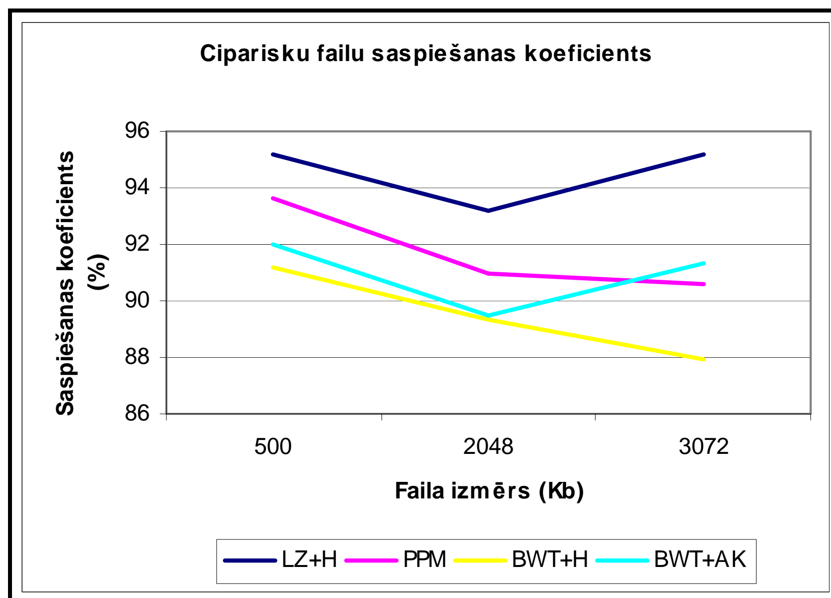
Cipariskie faili									
Parametrs	Saspiešanas koeficients (%)			Saspiešanas laiks(s)			Atspiešanas laiks (s)		
Faila izmērs (Kb) / Algoritms	500	2048	3072	500	2048	3072	500	2048	3072
LZ+H	95.2	93.2	95.2	0.1	1.3	1.5	0.1	0.5	0.5
PPM	93.6	91	90.6	0.1	1.3	1.5	0.1	1.5	1.5
BWT+H	91.2	89.3	87.9	0.5	3	5	0.1	1	1
BWT+AK	92	89.5	91.3	0.5	1.7	7	0.1	0.3	0.5
H	54.4	54.5	54.5	1.5	8	10	15	55	160

Tabula 6. Ciparisku failu saspiešanas rezultāti.

36. attēlā ir parādīts, kā mainās saspiešanas koeficients cipariskiem failiem, manot faila izmēru. Ir vērts atzīmēt, ka Haffmana algoritms šoreiz parādīja labāku rezultātu, nekā iepriekšējos testos ~55%. Vislabākā un efektīvāka saspiešana tika veikta ar LZ algoritmu ~95%.

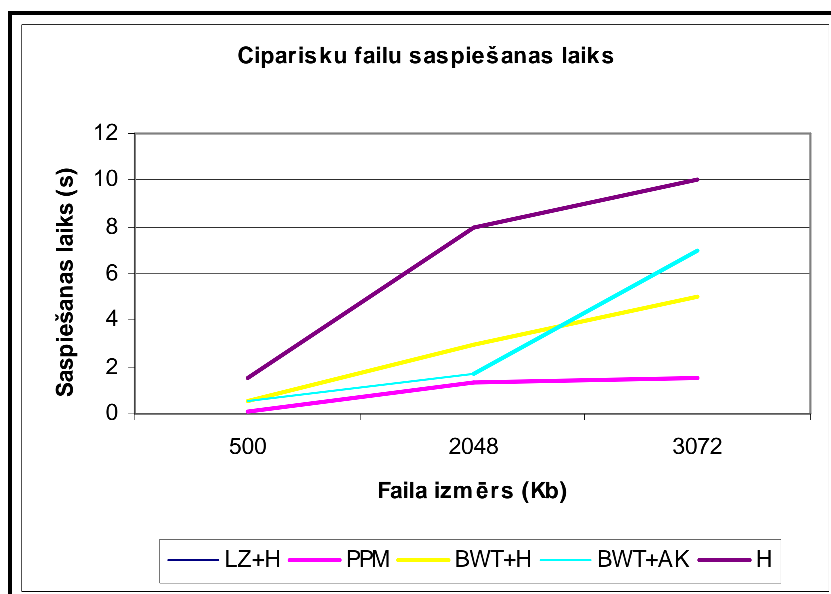


Attēls 36. Ciparisku failu saspiešanas koeficients.



Attēls 37. Ciparisku failu saspiešanas koeficients.

37. attēlā ir atspoguļota saspiešanas efektivitāte LZ, PPM, BWT algoritmiem. Ir skaidri redzams, ka LZ algoritms ir līderis šāda tipa failu saspiešanā. Vidējais saspiešanas koeficients ir 95%. Nedaudz sliktāki ir BWT+AK un PPM algoritmi, saspiešanas koeficients šiem algoritmiem ir par 5% sliktāks, nekā LZ algoritmam. BWT+H algoritma saspiešanas kvalitāte ir labāka nekā Haffmana algoritmam, bet tā lineāri samazinās, palielinoties faila izmēram.

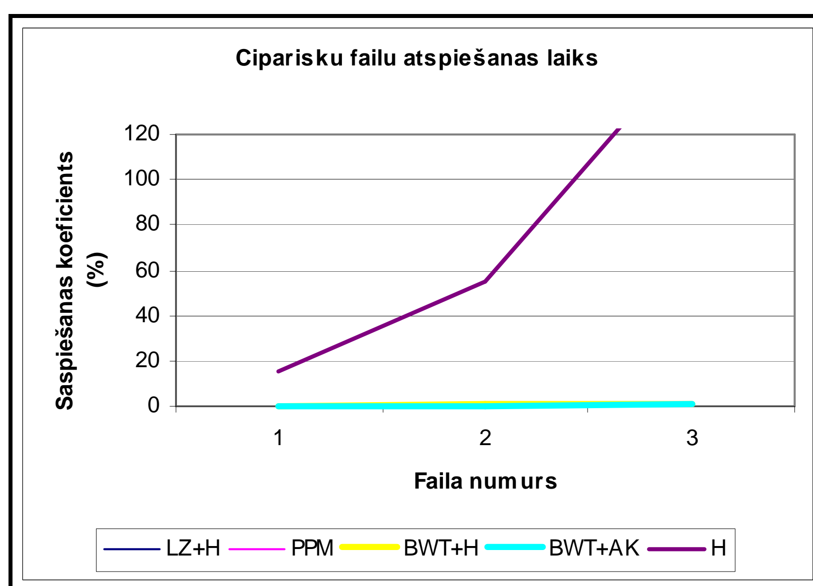


Attēls 38. Ciparisku failu saspiešanas laiks.

38. attēlā ir atspoguļots saspiešanas laiks visiem izmantotiem algoritmiem. Attēla nevar redzēt LZ algoritmu, šī algoritma saspiešanas ātruma rezultāti precīzi sakrīt ar PPM algoritma rezultātiem (38. attēlā ir rozā krasā). Saspiešanas laiks algoritmiem LZ un PPM nepārsniedz

divas sekundes. Citiem algoritmiem tas strauji aug, mainoties faila izmēram. Šajā testa BWT algoritmi parādīja sliktākus rezultātus laikā ziņā, nekā visos parējos testos. Kā arī saspiešanas koeficients nav pat tuvu līderim-LZ algoritmam. Pie kam tas samazinās, palielinot faila izmēru (attēls 37). Cipariskiem failiem BWT algoritmi ir tuvu Haffmana algoritmam saspiešanas laika ziņā.

Ciparisku failu atspiešanas ātruma rezultāti ir atspoguļoti 39. attēlā. LZ, PPM, BWT algoritmi paradīja tādus pašus rezultātus, kā visos iepriekšējos testos. Atspiešanas laiks nepārsniedz vienu sekundi. Neskatoties uz to, kā BWT algoritmam vajadzēja daudz vairāk laika lai saspiest datus, tas neietekmēja atspiešanas laiku. Haffmana algoritma gadījumā atspiešanas laiks ir līdzīgs, kā visos iepriekšējos testos ~2 minūtes.



Attēls 39. Ciparisku failu atspiešanas laiks.

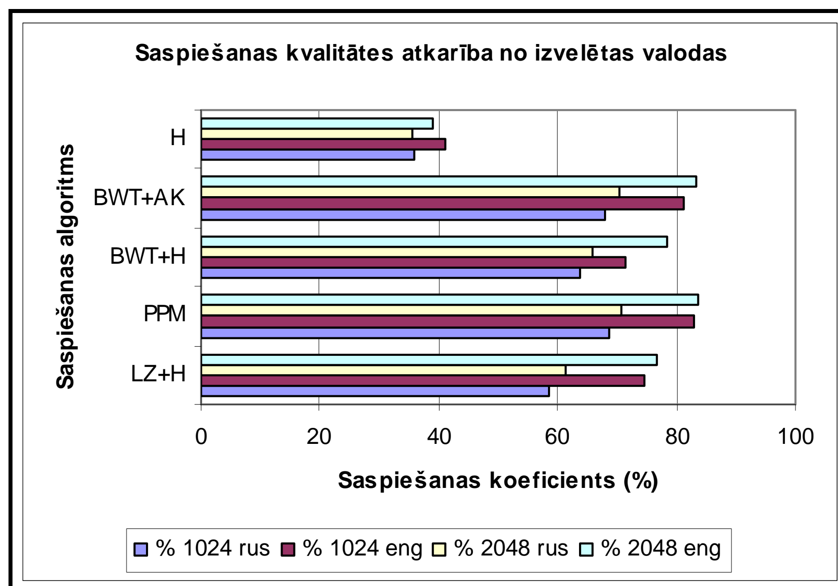
3.5.3.3 Saspiešanas efektivitātes atkarība no izvelētas valodas

Šī testa mērķis ir pārbaudīt, kā izvelētas valodas alfabēta lielums var ietekmēt saspiešanas rezultātus. Šim mērķim tika izvelētas divas valodas: krievu un angļu. Izveidoti faili ar izmēriem 1Mb un 2 Mb. 7. tabulā ir apkopoti iegūti testēšanas rezultāti.

Teksta faili												
Parametrs	Saspiešanas koeficients (%)				Saspiešanas laiks (s)				Atspiešanas laiks (s)			
Faila izmērs (Kb)	1024		2048		1024		2048		1024		2048	
Valoda / Algoritms	rus	eng	rus	eng	rus	eng	rus	eng	rus	eng	rus	eng
LZ+H	58.6	74.4	61.3	76.7	2.5	1.5	4	2.5	0.5	0.5	0.5	0.5
PPM	68.8	82.8	70.7	83.7	2	1.5	3	2.5	2	1.5	3.5	2
BWT+H	63.7	71.5	66	78.3	2.5	2	3.5	3	1	1	1	1
BWT+AK	67.9	81.2	70.3	83.2	1.5	1.5	3.5	3.5	0.5	0.5	1	1
H	36	41	35.5	38.9	6	5	11	9	42	35	85	70

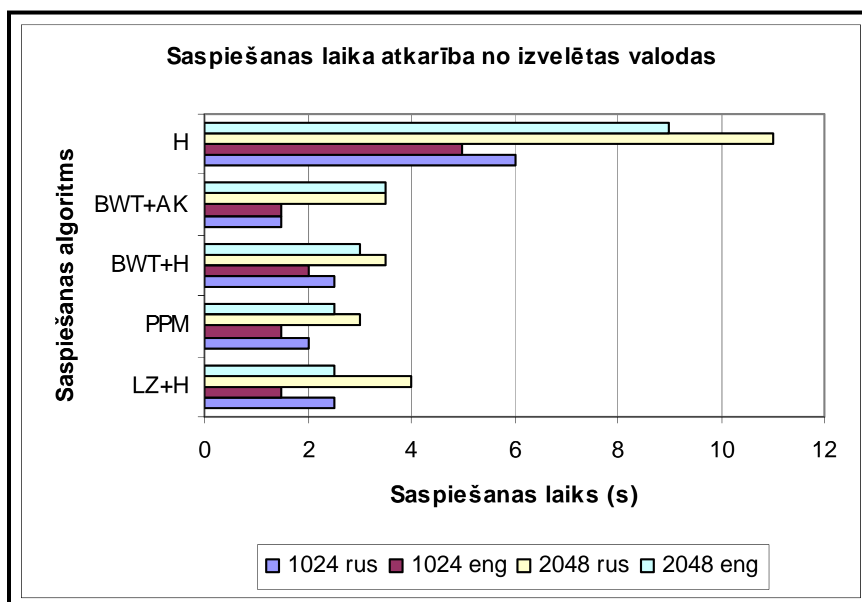
Tabula 7. Teksta failu saspiešanas rezultāti.

Saspiešanas kvalitātes rezultāti ir atspoguļoti 40. attēlā. Var droši pateikt ka valodas izvēle var krietni ietekmēt saspiešanas procesu. Jo mazāks ir valodas alfabēts jo labāku saspiešanas koeficientu izdosies panākt. 40. attēlā var ieraudzīt ka algoritmiem LZ, BWT un PPM saspiešanas koeficients ir par ~15% lielāks spiežot angļu valodas tekstiskus failus. Haffmana algoritma gadījumā šis izmaiņas nav tik būtiskas, bet tomēr angliskiem tekstiem saspiešanas koeficients ir par ~5% labāks.



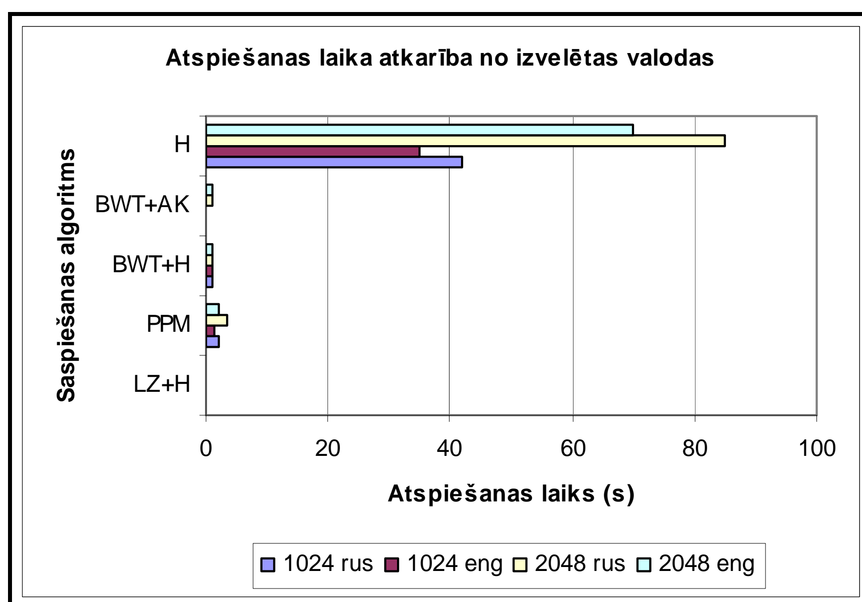
Attēls 40. Saspiešanas kvalitātes atkarība no izvelētas valodas.

41. attēlā ir atspoguļota saspiešanas ātrdarbība failiem ar dažādam valodām. Saspiešanas laiks ir mazāks tiem failiem kuriem valodas alfabēts ir mazāks. LZ algoritmam tas ir par 30% mazāks angļu valodas failiem. PPM, BWT+H un Haffmana algoritmiem par 15%. BWT+AK algoritma gadījumā, valodas izvēle neietekmē saspiešanas ātrdarbību.



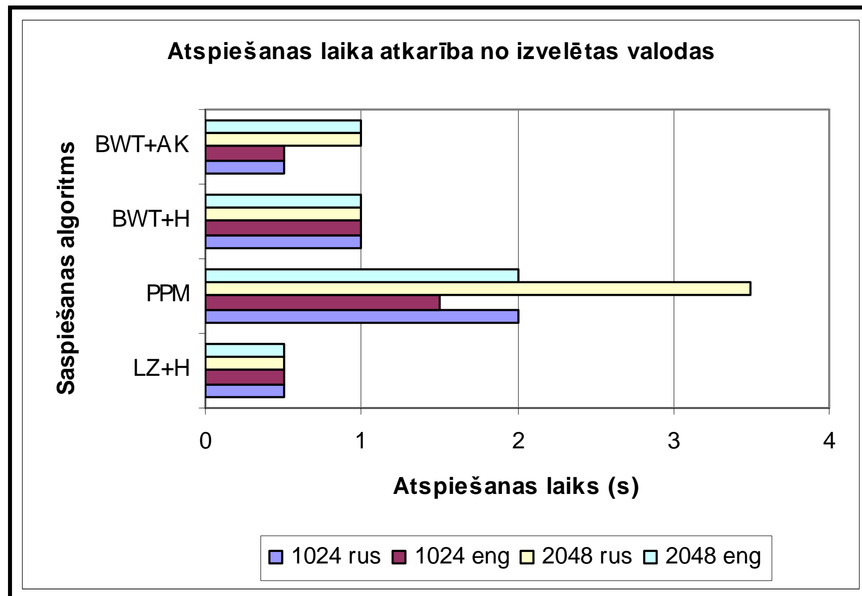
Attēls 41. Saspiešanas laika atkarība no izvelētas valodas.

Attēlos 42 un 43 ir atspoguļots atspiešanas process un tā izmaiņas atkarība no izvelētas valodas. 42. attēlā var ieraudzīt, kā Haffmana algoritma gadījumā valodas izvēle var ietekmēt atspiešanas procesu. Jo mazāks ir valodas alfabēts jo ātrāka ir atspiešana - līdzīgi ka saspiešanas gadījumā.



Attēls 42. Atspiešanas laika atkarība no izvelētas valodas.

LZ un BWT algoritmiem atspiešanas laiks ir vienāds, neatkarīgi no tā, kāda valoda tika izmantota veidojot failus. PPM algoritma gadījuma, tāpat ka Haffmana algoritmam atspiešanas laiks ir mazāks angļu valodas failiem.



Attēls 43. Atspiešanas laika atkarība no izvelētas valodas.

Noslēgums

Bakalaura darba izpildes gaitā tika paveikta salīdzinošā analīze mūsdienas populārākajiem saspiešanas algoritmiem, modelēšanas metodēm un arhivātoriem.

Modelēšana un kodēšana ir divi pietiekami neatkarīgi un patstāvīgi informācijas saspiešanas teorijas apgabali. Saspiešanas kvalitāte tiek noteikta ar izmantotu modeli, tomēr ar labas kodēšanas paņēmienu saspiešanas kvalitāte var būt būtiski uzlabota, tāpēc gan modelēšana gan kodēšana rada lielu praktisku interesi. Dažreiz modelēšanas metožu pielietošana bez efektīvas kodēšanas ir bezjēdzīga. Un arī otrādi, laba saspiešana netiek panākta, izmantojot tikai efektīvas kodēšanas metodes un nepielietojot datu modelēšanu.

Visbeidzot, apkopojot datus par izpētītajām metodēm un arhivātoriem, autors secināja, ka vislabāk un visātrāk dokumentus saspiež 7Zip arhivātors. Jo lielāks datu apjoms, jo labāku saspiešanu rāda 7zip. Šis ir viens no labākajiem arhivātoriem, kas ir pieejams bezmaksas. No pētītiem saspiešanas algoritmiem, labākus rezultātus paradīja PPM un LZ algoritmi. Saspiešanas koeficients tiem atšķīrās tikai par dažiem procentiem un failu saspiešanas/atspiešanas laiks ir gandrīz vienāds. XML failu saspiešanā labāku un ātrāku rezultātu parādīja PPM algoritms, cipariskiem failiem –LZ algoritms.

Visas darbā pētītās metodes var ērti pielietot praksē. Autora izveidotais arhivātors JFC6 ir Haffmana algoritma realizācija programmēšanas valodā Delphi7. Šī programma palīdz labāk izjust algoritma īpašības un saprast pirmsarhivēšanas modelēšanas metožu nepieciešamību, tekstiskas informācijas saspiešanai. Kas arī tika konkretizēts testos un izdarītos secinājumos. Modificējot Haffmana algoritmu, kas tika izmantots arhivātorā JFC6 var panākt ātrākus saspiešanas un atspiešanas rezultātus, bet būtiski uzlabot saspiešanas koeficientu neizdosies. Šis algoritms, tīrā veidā, neder tekstiskai informācijai, ir jāpielieto modelēšanas metodes, tekstiskas informācijas pārveidošanai, pirms datu kodēšanas. Turklāt, pielietojot Haffmana kodu jau nomodelētiem datiem, saspiešana ir ātrākā un efektīvākā, nekā citiem algoritmu apvienojumiem (testēšanas laikā izmantotais LZMA+Huffman algoritms to ļoti labi demonstrē).

Efektīvu saspiešanas realizāciju autors ieteic veikt ar LZ dzimtas algoritmiem, jo tiem ir raksturīgs labs ātrums un datu saspiešanas kvalitāte. Nomodelētiem datiem var pielietot Haffmana algoritmu, lai panāktu labāku saspiešanas kvalitāti. Pārsvarā visiem LZ grupas pārstāvjiem ir vajadzīgs mazāk datora resursu, nekā PPM dzimtas algoritmiem, bet saspiešanas kvalitāte ir tik pat laba. Lietojot vārdnīcu saspiešanas metodes, ieteicams vairāk laika veltīt efektīvas un ātras meklēšanas metodes izstrādāšanai. Saspiešanas ātrums un efektivitāte šiem algoritmiem ir atkarīgi tieši no meklēšanas algoritma, kuram pēc iespējas ātrāk ir jānomaina konkrēts konteksts, vai vārds, ar norādi. Pēc autora domām, LZ dzimtas algoritmi ir vienīgi, kas

varētu būt pielietoti efektīvai datu bāzes saspiešanā. Ar šiem algoritmiem var efektīvi saspiest visu datu bāzi, bet kad rādīsies nepieciešamība atrast kaut kādus datus ,var efektīvi atspiest tikai konkrētu datu bāzes daļu. Citiem algoritmiem ir jāatspiež visu, lai dekodētu tikai daļu no datiem. Būtu labi, ja algoritms varētu atpazīt faila kontekstu un pielietot efektīvākus algoritmus konkrētam gadījumam (t.i. atpazīt lielus kontekstus ar cipariskiem datiem, izšķirs valodas un t.t.). Bet LZ algoritma gadījumā tas nav tik būtiski, tas efektīvi saspiež jebkurus tekstiskus datus.

Bakalaura darba ietvaros bija cerēts iegūt ieskatu par tekstiskas informācijas saspiešanas algoritmiem, modelēšanas metodēm un to pielietojumiem. Veicot bakalaura darbu, izdevies iedziļināties saspiešanas metožu teorētiskajās nostādnēs un to praktiskajās realizācijās.

Veiktā algoritmu analīze var kalpot par tālāku darba attīstību. Nākamais pētīšanas solis, kas, iespējams, būs aprakstīts autora nākamajos darbos, ir efektīva datu bāzes saspiešana un atspiešana ar LZ dzimtas algoritmiem. To var izdarīt, detalizēti izpētot algoritmus, kas izmanto datu modelēšanu saspiešanas procesā, kā arī veidojot dažāda veida modifikācijas jau aprakstītajām metodēm.

Izmantotas literatūras saraksts

1. D.A. Huffman, "A method for the construction of minimum-redundancy codes", Proc. Inst. Radio Engineers, vol. 40, no. 9, pp. 1098-1101, Sep. 1952. [tiešsaite]. [atsauce 20.04.2006]. Pieejams internetā: http://compression.graphicon.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf
2. J. Katajainen, A. Moffat, "In-place calculation of minimum-redundancy codes", Proc. Workshop on Algorithms and Data Structures, pp. 393-402, Aug. 1995. . [tiešsaite]. [atsauce 15.04.2006]. Pieejams internetā: http://www.cs.mu.oz.au/~alistair/abstracts/inplace.html
3. A. Moffat, A. Turpin, "On the implementation of minimum-redundancy prefix codes", IEEE Transactions on Communications, vol. 45, no. 10, pp. 1200-1207, June 1997. [tiešsaite]. [atsauce 10.04.2006]. Pieejams internetā: http://compression.graphicon.ru/download/articles/huff/moffat_turpin_1997_huff_impl_pdf.rar
4. R.L. Milidiu, E.S. Laber,"The warm-up algorithm: A Lagrangean construction of length restricted Huffman codes", TR-15, Departmento de Informatica, PUC-RJ, 1997. [tiešsaite]. [atsauce 21.04.2006]. Pieejams internetā: http://citeseer.nj.nec.com/milidiu96warmup.html
5. Ватолин Д., Ратушняк А., Смирнов М., Юкин В. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео. - М.: ДИАЛОГ-МИФИ, 2002. - 384 стр. [tiešsaite]. [atsauce 15.05.2006]. Pieejams internetā: http://compression.graphicon.ru/book.
6. Д. Сэломон. Сжатие данных, изображений и звука. - М.: Техносфера, 2004, 386 стр.
7. 7-Zip Homepage. [tiešsaite]. [atsauce 15.05.2006]. Pieejams internetā: http://www.7-zip.org
8. Dgca Homepage. [tiešsaite]. [atsauce 10.05.2006]. Pieejams internetā: http://www.emit.jp
9. WinAce - your archiving companion. [tiešsaite]. [atsauce 20.05.2006]. Pieejams internetā: http://www.winace.com
10. WinRAR archiver, a powerful tool to process RAR and ZIP files. [tiešsaite]. [atsauce 19.05.2006]. Pieejams internetā: http://www.rarlab.com
11. WinZip Upgrade Information. [tiešsaite]. [atsauce 17.05.2006]. Pieejams internetā: http://www.winzip.com

Apliecinājums

Ar šo es apliecinu, ka šodien iesniegto bakalaura darbu es esmu veicis pašrocīgi un esmu izmantojis tikai tajā norādītos palīglīdzekļus.

Rīgā, Paraksts:

Bakalaura darbs izstrādāts
LU Datorikas nodaļā

Autors:
Fizikas un matemātikas ……………………………
fakultātes students Jurijs Fjodorovs
St. apl. Nr. DatZ 020061 2006.g………….jūnijā.

Darba vadītājs:
docents, Dr. sc. comp
Guntis Arnicāns …………………………..
LU Fizikas un matemātikas fakultāte

Recenzents:
Mg. sc. comp.
Jānis Zuters ……………………………
LU Fizikas un matemātikas fakultāte

Darbs iesniegs Datorikas nodaļā 2006.g…….jūnijā.

Pieņēma sekretāre ..……………………………

Aizstāvēts datorzinātņu bakalaura pārbaudījumu komisijas sēdē

2006.g…………………………ar atzīmi……………………………………………

Protokols Nr.____

Bakalaura pārbaudījumu komisijas sekretārs:
Mg. sc. comp. Uldis Straujums ………………………….