UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Mežik

Stiskanje podatkov s pomočjo kontekstno neodvisnih gramatik

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00532/2010
Datum: 01.10.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MATEJ MEŽIK**

Naslov: **STISKANJE PODATKOV S POMOČJO KONTEKSTNO NEODVISNIH GRAMATIK**

COMPRESSION USING CONTEXT FREE GRAMMARS

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Preučite glavne metode stiskanja podatkov z uporabo kontekstno neodvisnih gramatik in podajte razloge za njihovo uporabo. Realizirajte nekaj najpomembnejših metod te vrste in primerjajte njihovo učinkovitost med seboj. Primerjavo izbranih metod opravite na različnih vrstah vhodnih podatkov.

Mentor:
B. Slivnik
doc. dr. Boštjan Slivnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Vstavite original izdane teme diplomskega dela (priloženo).....

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Matej Mežik,

z vpisno številko 63060216,

sem avtor/-ica diplomskega dela z naslovom:

Stiskanje podatkov s pomočjo kontekstno neodvisnih gramatik

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. Boštjan Slivnik

  in somentorstvom (naziv, ime in priimek)

  _____

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

Zahvala

Zahvaljujem se mentorju doc. dr. Boštjanu Slivniku ter asistentu dr. Urošu Čibeju za ves trud in vso pomoč, ki sta mi jo nudila pri izdelavi diplomskega dela.

Prav tako gre zahvala tudi moji družini, punci, sodelavcem in vsem tistim, ki so mi stali ob strani v času študija in pri izdelavi diplomske naloge.

Iskrena hvala.

Kazalo

Povzetek

Pojav računalnikov in operacijskih sistemov, ki tečejo na njih, je s seboj prinesel tudi prenašanje in shranjevanje ogromnih količin podatkov, s katerimi zna operacijski sistem upravljati. Tako kot smo pri shranjevanju oz. pomnjenju omejeni prostorsko, smo tudi pri prenosu podatkov na nek način omejeni in sicer časovno. To sta bila vzroka za nastajanje prvih zasnov algoritmov, za stiskanje (kompresijo) podatkov v manjše zaključene celote, namenjene shranjevanju in prenašanju. Takšne pakete je moč kasneje seveda ponovno razširiti v izvorne podatke, kar pomeni, da imajo namen manjše porabe razpoložljivih virov, bodisi prostora ali časa.

Osrednja tema je primerjava peščice obstoječih stiskalnih algoritmov, ki temeljijo na kontekstno neodvisnih gramatikah. Namen je ugotoviti, kateri izmed njih se na izbranih formatih datotek obnašajo optimalno. Za primerjavo sem vzel osnovno različico algoritma LZW, algoritem Re-Pair ter algoritem za stiskanje podatkov s pomočjo bisekcije. Vsi algoritmi izhajajo iz skupine algoritmov, ki za jedro stiskanja uporabljajo kontekstno neodvisne gramatike in slovar, kot pripomoček pri stiskanju. Algoritme sem tudi implementiral in sicer v programskem jeziku Pascal, s pomočjo razvojnega okolja Delphi in uredil ustrezno obliko vmesnika za testiranje. Za primerjanje sem izbral časovno potratnost (hitrost) in razmerje med izvorno in stisnjeno datoteko. Glavni namen je bil ugotoviti obnašanje algoritmov na izbranih datotekah, različnih formatov, in sicer grafičnem BMP (bitmap) formatu, tekstovnem TXT formatu, besedilno oblikovnem PS (postscript) formatu ter programski kodi, napisani v Javi.

Ključne besede:

Stiskanje podatkov, kontekstno neodvisne gramatike, algoritem LZW, algoritem Re-Pair, metoda za stiskanje s pomočjo bisekcije.

Abstract

The emergence of computers and their operating systems caused the transmition and storing huge amounts of data, which operating system can manage. We are limited with memory space and also with the transfer data time. This was the reason for invention of first algorithms for compression data to small packages, intended for the storage and transmission. Such packages can be later extended in the source data, which means that they are intended to lower consumption of available resources, either space or time.

The main theme is the comparison of a handful of existing compression algorithms based on context-free grammar. It will identify one of them in selected file formats behave optimally. For comparison I took a basic version of the LZW algorithm, the Re-Pair algorithm and the bisection algorithm. All algorithms are used for core compression context-free grammar and dictionary. I also implemented algorithms in the Pascal programming language, with Delphi development environment and made the user interface for testing. I chose to compare the time (speed) and compression ratio between the source and the compressed file. The main purpose was to determine behavior of algorithms on the selected files of different formats: graphic BMP (bitmap) format, TXT format, PostScript format, and Java source.

Keywords:

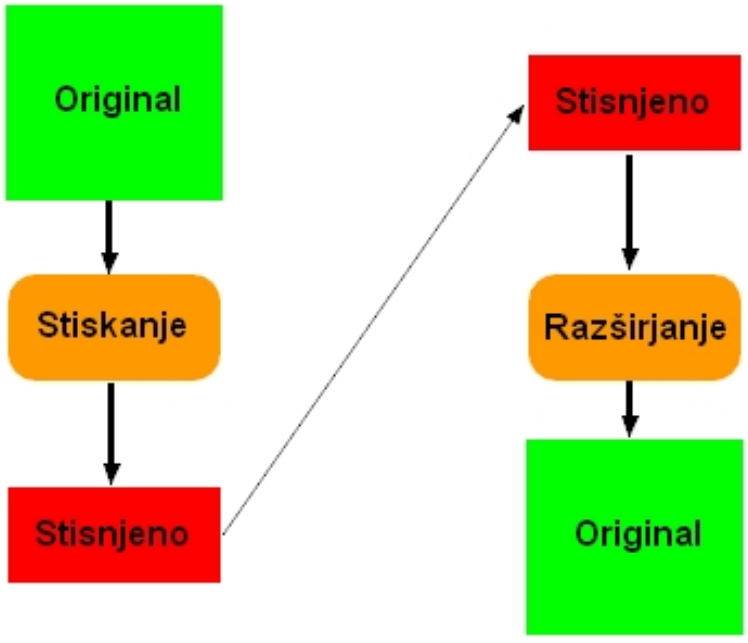
Data compression, context-free grammars, LZW algorithm, Re-Pair algorithm, bisection method.

1. Uvod

Razvoj računalnikov temelji na čimhitrejši obdelavi čimvečih podatkov, ne glede na njihov pomen. Pri teh procesih pa smo vedno nekako omejeni, saj imamo končno število virov, ki jih lahko uporabimo. V primeru, da so ti viri namenjeni hranjenju, pomnenju podatkov, potem smo prostorsko omejeni. Na voljo pa imamo tudi določen čas, zato smo omejeni tudi časovno. Ljudje zelo neradi dolgo čakamo za dosego cilja. Primer je brskanje po medmrežju, kjer je potrebno počakati na prikaz spletne strani. Poleg čakanja, pa bi radi imeli večino informacij nekje shranjenih, za kar je potreben prostor. Takšne omejitve lahko rešimo začasno, z dodajanjem novih virov ali pa dolgoročno, s stiskanjem podatkov.

Začetki stiskanja podatkov segajo v leto 1838 [9], ko je bila odkrita Morsejeva abeceda, namenjena telegrafiji. To je bil prvi primer stiskanja podatkov, zasnovanega na spremenljivih dolžinah kod. Modernejši postopki stiskanja so se začeli pojavljati v 40. letih 19. stoletja, ko se je začela razvijati tudi informatika. Leta 1949 sta Claude Shannon in Robert Fano razvila sistematično določevanje kod, ki je temeljilo na verjetnostih pojavitev posameznih blokov. Optimizirano različico pa je leta 1951 razvil David Huffman (poglavje 2.2.7). Starejše metode se še danes uporabljajo, predvsem v strojni opremi, kjer je potreben kompromis med učinkom stiskanja in izogibanju napakam.

Cilj stiskanja podatkov je določeno količino podatkov predstaviti s podatki, ki zavzamejo manj prostora in jih je kasneje moč razširiti v izvorne podatke. Tak proces (slika 1) zahteva definicijo stiskanja in definicijo razširjanja podatkov, saj sta stiskanje in razširjanje med seboj odvisna. Osnovni pojmi, ki se pojavljajo pri takšnih procesih so izvorni podatki (original), stiskanje, stisnjeni podatki in razširjanje.



Slika 1: Proces stiskanja podatkov

Začetna ideja stiskanja podatkov je bila, kako bi lahko s čim manj simboli poslali neko sporočilo (besedilo), ki bi ga lahko ponovno »razvozlali«. Tako sta leta 1977 Abraham Lempel in Jacob Ziv razvila prvi stiskalni algoritem LZ77, ki se je uveljavil v računalnišvu, imel pa je tudi zelo dobro lastnost, omogočal je brezizgubno stiskanje, kar pomeni, da je bilo iz stisnjenih podakov mogoče ponovno pridobiti izvorne, ne da bi pri tem katerega izgubili. LZ77 je bil temelj kasnejšim različicam [2,5], kot so LZ78, LZW, LZSS, LZMA, LZO (zelo hitra različica) in LZX. Danes v računalništvu najbolj razpoznaven ZIP kompresijski format uporablja v osnovi LZMA različico.

Zaradi kasnejših performančnih potreb so se razvile tudi danes malo manj uveljavljene verzije stiskalnih algoritmov, med katere sodijo algoritem za stiskanje podatkov s pomočjo bisekcije (Bisection), algoritem za stiskanje podatkov s pomočjo sekvenc (Sequential), algoritem z najdaljšim ujemanjem (Longest Match), razni t.i. požrešni algoritmi (Greedy), algoritem Re-Pair, itd. Vse te metode lahko delimo glede na:

- izgube pri pridobivanju izvornih podatkov
  (izgubne, brezizgubne),
- uporabo slovarja (preslikovalna tabela) med samim potekom stiskanja
  (dictionary-based compression)
- uporabo gramatik (kontekstno neodvisne), kot osnovo za stiskanje
  (grammer-based compressions).

Izgubne metode (slika 5) je moč uporabiti v primerih, ko lahko del izvornih podatkov tudi izgubimo, brez da bi jih kasneje pogrešali. Primer takšne uporabe je ogled video posnetka preko spleta. Metode, ki uporabljajo slovar, nimajo zapletenega sistema stiskanja znakov, temveč za to uporabljajo preslikovalno tabelo (hashmap), imenovano slovar.

Uporaba kontekstno neodvisnih gramatik ima pri stiskanju podatkov velik vpliv pri postopku razširitve, saj lahko zelo enostavno in zato tudi zelo hitro pridobimo izvorne podatke. Lastnost takšnih metod je, da običajno uporabljajo slovarje, kar pa velja omeniti je to, da lahko po stisnjenih podatkih iščemo obstoječe vzorce. To si predstavljamo tako, da brez razširjanja v njih ugotovimo, ali se med njimi nahaja iskana vsebina (beseda, znak, vzorec podatkov, itd.). Ravno zato bomo metodam, ki uporabljajo kontekstno neodvisne gramatike, namenili posebno pozornost, jih podrobnejše preučili in preizkusili kako se obnašajo v praksi. V diplomski nalogi smo si zadali cilj, da ugotovimo, kateri izmed prej navedenih algoritmov je najbolj učinkovit za stiskanje izbranega formata datoteke.

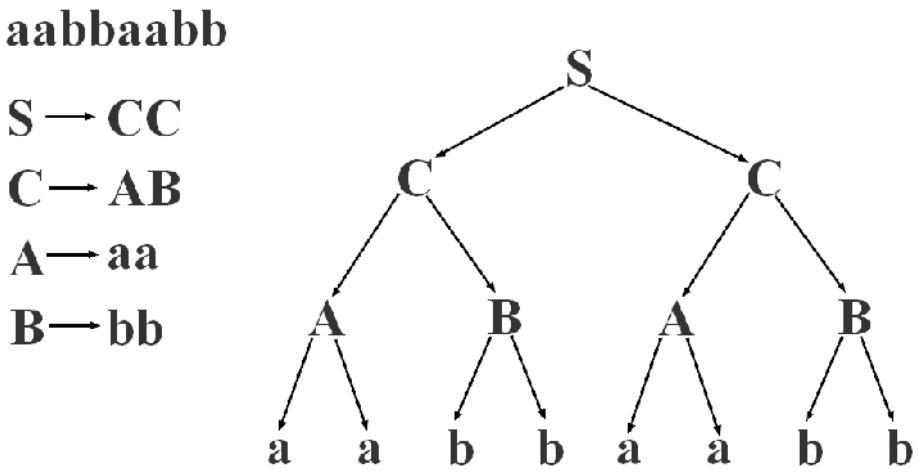
1.1. Kontekstno neodvisne gramatike

Gramatika je zapis pravil, ki jih je potrebno upoštevati v samem kontekstu uporabe [7]. Skupek takšnih pravil lahko najdemo v slovnici, kjer nam predpisujejo, na kakšen način lahko gradimo posamezne stavke, besede, ipd., saj bi bilo drugače nemogoče prepoznati njihovo vsebino. Množična uporaba gramatik v samem računalnišvu je pri definiranju programskih jezikov, ki imajo točno določeno strukturo in zaporedja.

V kontekstno neodvisnih gramatikah ni dovoljena uporaba pravil, ki se prekrivajo, kar pomeni, da ni mogoče, da bi dve pravili modificirali enako območje uporabe. V našem primeru bomo potrebovali takšna pravila za definiranje preslikav med novo uvedenimi znaki in njihovimi pomeni.

Gramatika je opisno sestavljena iz:

- množice končnih simbolov, najmanjše enote izvornih podatkov,
- množice spremenljivk, novonastale enote,
- začetnega simbola, ena izmed spremenljivk,
- množice produkcij, ki se generirajo po formuli $X \rightarrow \gamma$, kjer je X spremenljivka, γ pa zaporedje končnih simbolov ali spremenljivk.



Slika 2: Zapis množice produkcij in izpeljava niza »*aabbaabb*« v gramatiki

Na zgornji sliki (slika 2) je z uporabo drevesne strukture prikazana izpeljava kontekstno neodvisne gramatike na primeru niza *aabbaabb*, ki ima končna simbola *a* in *b* in spremenljivke *S*, *C*, *A*, *B*, pri čemer je *S* začetni simbol. Na levi strani slike je zapisana množica produkcij, ki določajo sestavo niza.

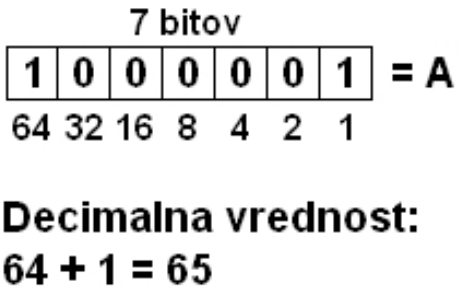
2. Metode stiskanja podatkov

2.1. Stiskanje in redundanca

Kot je bilo že omenjeno, poznamo več metod stiskanja podatkov, ki se uporabljajo za različne namene. Vse so zasnovane na različnih idejah, namenjene različnim tipom podatkov, njihovi rezultati se razlikujejo, v principu pa je njihov končni namen odstraniti redundante podatke iz izvora (originala) in pridobiti na prostoru, posledično tudi na času. Katerakoli nenaključna zbirka podatkov ima točno določeno strukturo, ki se jo da predstaviti s podatki manjše velikosti, kjer izvorna struktura podatkov na prvi pogled ni opazna.

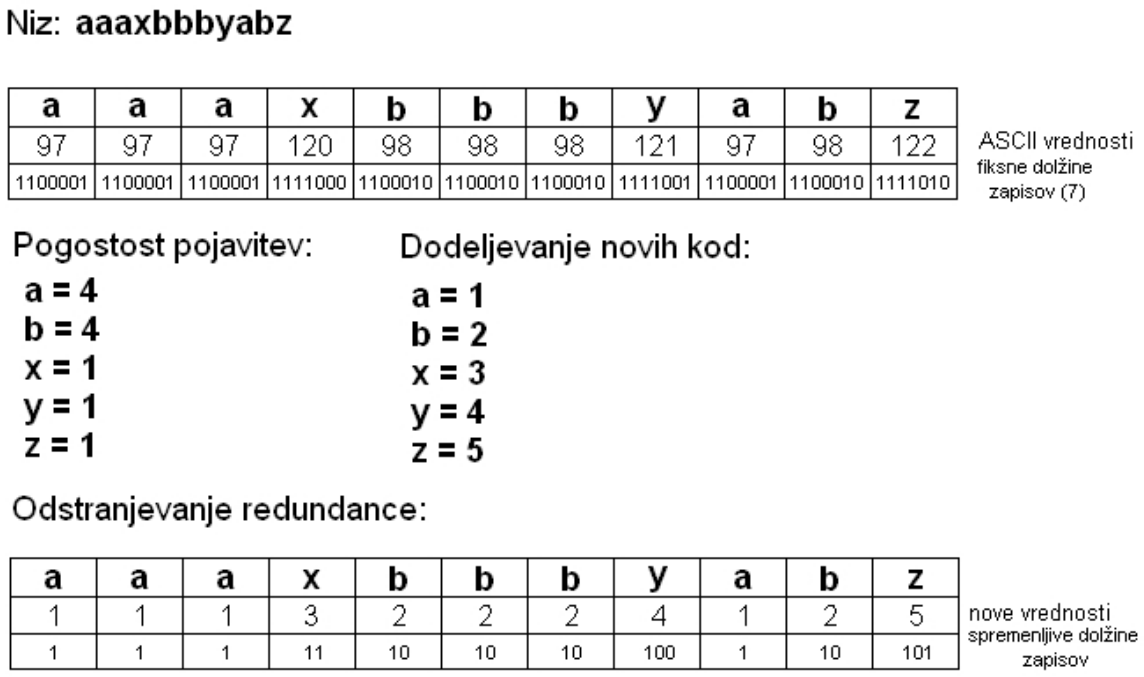
Pojem redundanca je pri stiskanju podatkov zelo uveljavljen in hkrati tudi pomemben. V besedilu tipične slovenščine zelo pogosto najdemo znak *A*, zelo redko pa *X*. To je primer znakovne redundance. Poznamo tudi kontekstno redundanco, kjer je vnaprej znano, kateri so možni sosedi določenega znaka. Takšen primer je v angleščini, in sicer znaku *Q* skoraj vedno sledi znak *U*. Podobno se dogaja tudi pri slikah, le da se tu pogovarjamo o barvah in slikovnih točkah, saj je velika verjetnost da sta sosednji točki enake barve. Kakorkoli, znano je, da so znaki spremenljivih dolžin manj redundatni kot znaki fiksnih dolžin. Ta ugotovitev je povzročila splošno pravilo o prirejanju krajših zapisov znakom, ki se pojavijo pogosteje in daljših zapisov znakom, ki se pojavijo redkeje, kot to počne metoda Huffman (poglavje 2.2.7).

Večina obstoječih metod temelji na predstavitvi neučinkovitih (dolgih) podatkov z učinkovitimi (krajšimi), gledano prostorsko. V računalništvu je takšno kodiranje mogoče, ker je večina podatkov predstavljena z daljšim zapisom, kot bi bil potreben. Vzrok tiči v tem, da se takšni podatki mnogo lažje in zato tudi hitreje obdelujejo (prikaz slike), to pa je večkrat bolj pomembno kot stiskanje podatkov. ASCII sistem kodiranja znakov je primer takšne uporabe, saj so znaki zapisani daljše kot bi bilo potrebno. Uporablja fiksne 7-bitne zapise (slika 3), nad katerimi je lažje izvajati operacije, kot če bi bili spremenljivih dolžin, čeprav bi bili ti bolj učinkoviti. V svetu, kjer so podatki predstavljeni z najkrajšim možnim zapisom, stiskanje seveda ne pride v poštev, saj z njim ni moč doseči ciljnega učinka, to je zmanjšanje obsega podatkov.



Slika 3: Predstavitev znaka A z ASCII kodo fiksne dolžine

Princip odstranjevanja redundatnih podatkov (slika 4) poskrbi za to, da stiskanje že stisnjenih podatkov nima smisla, saj je redundanca že minimalna ali pa je sploh ni več. V primeru, ko imamo popolnoma naključno besedilo, kjer ima vsak izmed znakov skoraj enako verjetnost pojavitve, fiksna dolžina zapisa znakov ne predstavlja redundance. Nobenih podatkov ni moč tolikokrat stistniti, da bi bili na koncu lahko predstavljeni kot byte ali pa celo bit (poglavje 3.2), saj tako majhen podatek ne more vsebovati preostalih informacij, ki jih potrebujemo ob postopku razširitve.



Slika 4: Prikaz odstranjevanja redundantnih podatkov

2.2. Obstoječe metode

Obstaja mnogo različnih metod za stiskanje podatkov [2]. Nekatere so namenjene besedilom, druge grafičnim formatom (slike in video). Večina metod je klasificiranih v štiri različne kategorije, in sicer kodiranje znakov s pomočjo zaporedij (RLE), statistične metode, metode s slovarjem (tudi LZ metode) in transformacije.

2.2.1. Osnovni pojmi

Pri stiskanju poznamo nekaj specifičnih izrazov, katere je pametno pojasniti, kaj pomenijo oz. kaj predstavljajo. Že večkrat omenjen *stiskalnik* ali *kodirnik* je program, namenjem stiskanju podatkov na vhodu in zapisovanju izhodnega toka podatkov, ki so nizko redundančni. *Razširjevalec* ali *dekoder* je program, ki pretvarja podatke v obratni smeri, ponovno pridobi izvorne podatke (original). Izraz *tok podatkov*, omenjen zgoraj, delimo na *vhodni* in *izhodni tok* (poglavje 3.1), ki sta namenjena črpanju podatkov iz datoteke in shranjevanju le-teh v datoteko. *Tok podatkov* je bolj splošen izraz za pretok podatkov, saj jih je možno uporabiti tudi brez vmesnega shranjevanja v datoteko. Za *vhodni tok izvornih podatkov* uporabljamo tudi izraz *nestisnjeni podatki* ali *originalni podatki*. Vsebino končnega, *izhodnega toka podatkov* pa imenujeno tudi *kodirani* ali *stisnjeni podatki*. Razmerje med velikostjo izhodnih podatkov in velikostjo vhodnih podatkov imenujemo *stiskalno razmerje* (1). Pove nam koliko % velikosti originalnih podatkov zavzemajo izhodni podatki. V primeru razmerja, večjega od 100%, govorimo o *negativnem stiskanju*.

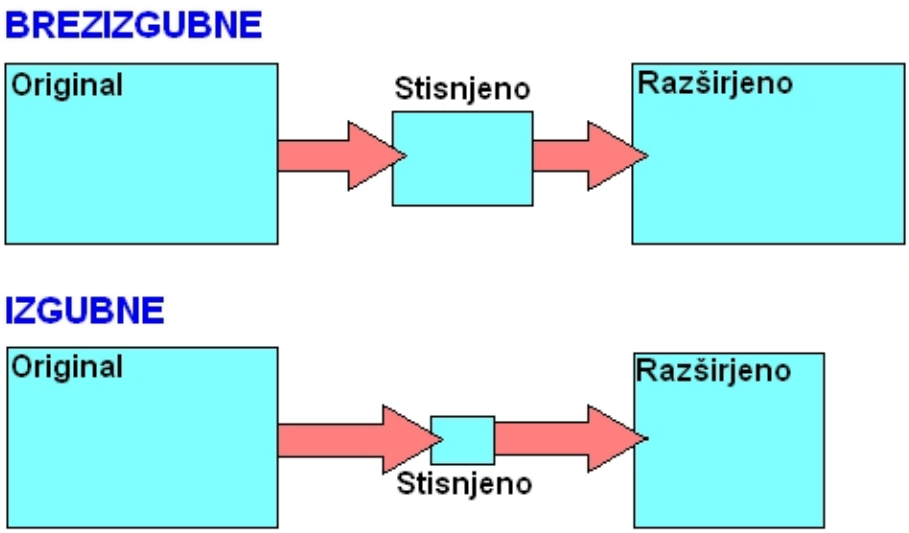
$$Stiskalno\ razmerje(\%) = \frac{Velikost\ izhodnih\ podatkov}{Velikost\ vhodnih\ podakov} \cdot 100 \qquad (1)$$

2.2.2. Vrste metod stiskanja

Poznamo več vrst metod glede na način stiskanja podatkov, ki ga uporabljajo, in njihove lastnosti. Ena izmed njihovih lastnosti je prilagodljivost.

Tako poznamo ***neprilagodljive*** stiskalne metode, ki so zelo toge, saj med samim procesom ne spreminjajo nobenih operacij, parametrov ali tabel. Takšne vrste metode so najbolj uporabne, ko na vhodu obstajajo podatki ene same vrste. V nasprotju z neprilagodljivimi metodami pa poznamo tudi ***prilagodljive***, ki med samim procesom prilagajajo parametre, operacije, tabele, ... Primer takšne metode je *metoda Huffman* (poglavje 2.2.7), namenjena kodiranju podatkov. Nekatere metode izvajajo 2-kratni pregled vhodnih podatkov, med prvim zbirajo statistične podatke, med drugim pa izvajajo stiskanje s pomočjo prednastavljenih parametrov iz prvega pregleda. Takim metodam pravimo ***delno-prilagodljive*** metode. Obstajajo pa tudi ***lokalno-prilagodljive*** metode, za katere je značilno, da prilgajajo same sebe glede na lokalne pogoje v vhodnih podatkih. Primer slednje metode je *metoda pomikanja v ospredje* (move to front method).

Kot smo že v uvodu omenili, pa ločimo metode tudi glede na možnost razširitve stisnjenih podatkov brez izgub. Obstaja nekaj metod, ki imajo lastnost ***izgube informacij***, saj tako pridobijo boljše stiskalno razmerje. Pri ***izgubnih metodah*** (slika 5) ob razširjanju stisnjenih podatkov ne dobimo takšnih, ki bi bili identični izvornim podatkom. Take metode lahko uporabljamo za stiskanje slik, filmov ali glasbe, saj pri teh formatih ne bomo zaznali delno izgubljenih podatkov. Na drugi strani pa imamo formate, kjer lahko samo en izgubljen bit povzroči nedelovanje programa ali nemogoče branje tekstovnih datotek. Za te vrste formatov moramo uporabiti ***brezizgubne metode***, med katere spada večina obstoječih metod.



Slika 5: Metode z izgubami in brez izgub

Kaskadno stiskanje je značilnost, kjer podatke *A* stisnemo z metodo *X* in dobimo stisnjene podatke *B*, ki jih nato ponovno stisnemo z metodo *Y*, ki nam vrne podatke *C*. Kot vidimo, imamo dve različni metodi *X* in *Y*, za katere pa je odvisno, ali sta izgubni oz. brezizgubni. V primeru, da sta obe metodi breizgubni, potem lahko v obratnem vrstnem redu pridobimo izvorne (začetne) podatke *A*, če pa je katerakoli izmed *X* in *Y* metod izgubna, pa je tovrstno dejanje nemogoče.

Zaznavno stiskanje nastopi pri metodah z izgubami, saj morajo znati poskrbeti za to, da ljudje psihofizično ne zaznamo sprememb, kljub izgubi informacij. Da bi takšne informacije metoda našla, potrebuje svojevrsten algoritem, ki ima to sposobnost, da zazna kateri del podatkov lahko zavrže, da na koncu ljudje ne bomo zaznali večjih sprememb, bodisi vizualnih ali slušnih. Pri tovrsntih metodah ima kodirnik v osnovi nalogo zadrževanja konstantnega stiskalnega razmerja, kar pomeni da za *X* bitov vedno poskuša najti pretvorbo, ki je dolga *Y* bitov.

Simetrično stiskanje je lastnost metod, kjer procesa stiskanje in razširjanje uporabljata enak algoritem, le v obratnem vrstnem redu. To ima posledično enako kompleksnost in zahtevnost pri stiskanju in razširjevanju. Pri ***asimetričnem stiskanju*** pa eden izmed procesov opravlja težje delo, kar ni nujno slabo. Obstajajo primeri uporabe asimetričnih metod pri arhiviranju, kjer se proces stiskanja podatkov izvaja enkrat, zato je lahko počasnejši, razširjanje pa večkrat in je zato smiselno, da je hitrejše. Pri varnostnem kopiranju podatkov pa je zaželjeno hitro stiskanje, razširjanje pa je lahko počasnejše, saj je verjetnost le-tega manjša. Večina današnjih modernih metod uporablja lastnost asimetričnega stiskanja.

2.2.3. Kodiranje s pomočjo zaporedij (RLE)

Ideja pri metodi RLE [1] je zasnovana na ponavljajočih se zaporedjih znakov, katere zamenjamo z zapisom para, ki vsebuje število znakov v zaporedju in znak. Če se znak *x* v zaporedju podatkov pojavi *n*-krat, potem to zaporedje zamenjamo z zapisom *nx*. Takšno kodiranje je mogoče uporabiti tako na nizih simbolov, kot tudi na slikovnih formatih. Pri obeh opcijah je znanih nekaj težav, katere pa se da rešiti na različne načine, le z določenimi omejitvami.

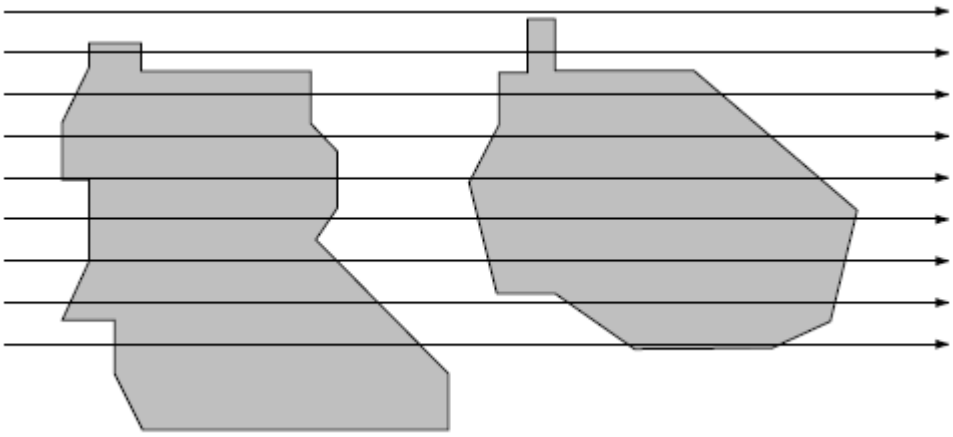
Pri kodiranju nizov s pomočjo zaporedij ni dovolj, da zamenjujemo zaporedja enakih simbolov samo s pari (dolžina, simbol), saj iz takšnih zapisov ni moč ugotoviti, ali je število v paru predstavljeno kot število v besedilu, ali kot število pojavitev simbola. Zato je potrebno uvesti nekakšen predznak, ki najavlja, da se za njim nahaja zapis para (dolžina, znak), v tem primeru pa potrebujemo že tri simbole za zapis, kar pomeni, da je smiselno zamenjati samo tista zaporedja, ki dosegajo minimalno tri pojavitve enakega simbola. Pri tem načinu pa se pojavijo tudi problemi:

1) V običajnem besedilu (angleščina) najdemo največ zaporedij z maksimalno 2-kratnim ponavljanjem simbola, kar pomeni, da takšno besedilo nima smisla stiskati z metodo, ki si pomaga z zaporedji.

2) Predznak, ki smo si ga zamislili za najavljanje zamenjave, je lahko tudi del vsebine podatkov, saj obstajajo vsebine, ki uporabljajo simbole iz celotnega nabora obstoječih znakov. Ta problem uspešno rešuje, sicer z določenimi omejitvami, metoda MNP5 (Microcom Network Protocol), ki se uporablja pri modemskih komunikacijah, saj zaporedja, ki so daljša od dveh pojavitev nadomesti z zapisom naslednjega formata: trikratna ponovitev simbola, nato število ponovitev (lahko tudi 0). Slabost takšnega zapisa je, da stiskanje dosežemo šele takrat, ko se simboli ponavljajo več kot 4-krat.

3) V primeru, ko število pojavitev zapisujemo kot byte (8 bitov), smo s številom ponovitev navzgor omejeni do števila 255. Ponavadi je takšno ponavljanje simbolov zelo neobičajno, je pa mogoče pridobiti še dodatna tri mesta tako, da začnemo z 0, ki pomeni 3-kratno ponavljanje in tako pridemo do števila 258.

Kodiranje s pomočjo zaporedij se uporablja tudi pri slikovnih formatih, kjer imamo opravka s slikovnimi točkami. Običajno so te točke predstavljane s tabelo (t.i. bitmap), kjer so zapisane vse vrednosti obstoječih slikovnih točk. Slikovne točke so lahko predstavljene z enim bitom (črne in bele), lahko pa z več biti (barvne in sivinske). Pregled točk običajno poteka po vrsticah, in sicer od zgornje proti spodnji, kar pomeni, da je prva točka tista, ki je v zgornjem levem kotu polja, zadnja pa tista, ki je v spodnjem desnem kotu. Pri slikovnih formatih obstaja velika verjetnost, da so sosednje točke enake barve, zato je stiskanje (RLE) slikovnih datotek skorajda vedno zagotovljeno, odvisno od kompleksnosti slike. Primer stiskanja slikovnih točk, ki si sledijo na način: 15 belih, 34 črnih, 28 belih, itd. na koncu izgleda nekako takole: 15, 34, 28, ..., pri čemer se vedno začne s številom belih točk, v primeru da jih ni (na začetku), zapišemo število 0.



Slika 6: Potek branja slikovnih točk

Kodiranje s pomočjo zaporedij pa lahko uporabimo tudi na sivinskih slikah, in sicer na enak način pri nizih. Algoritem RLE bere intenzitete slikovnih točk in šteje njihove pojavitve v zaporedju, ki jih nadomesti z zapisom para (število ponavljanj, intenziteta), pri čemer običajno število ponavljanj zavzame byte, intenziteta točke pa od 4 do 8 bitov, odvisno od sivinske stopnje. Nastane problem ločevanja števcev pojavitev od intenzitet, za kar obstaja nekaj rešitev:

1) V primeru, da slikovne točke obsegajo maksimalno 128 sivinskih stopenj, lahko en bit v byte-u rezerviramo kot oznako (flag), ki ločuje števce od vrednosti točke.

2) Če je število sivinskih stopenj 256, potem jih lahko zmanjšamo na 255 z rezervirano vrednostjo »predznaka«, ki označuje vsak števec. Če se pojavi predznak, recimo 255, potem sledi števec.

3) Izberemo bit, ki bo ločeval števec od vrednosti točke, in sicer na način, da vsak posamičen bit zapišemo kot skupino osmih byte-ov pred vsak števec.

2.2.4. Statistične metode

Posebnost statističnih metod je ta, da za kodiranje uporabljajo spremenljive dolžine zapisov, to pomeni, da priredijo krajše zapise tistim simbolom, ki se pojavljajo večkrat oz. je njihova verjetnost za pojav večja. Razvijalci takšnih algoritmov se srečujejo z dvema glavnima problemoma:

1) Kako novonastale simbole kasneje nedvoumno prepoznati?

2) Kako novonastale simbole zapisati z minimalno povprečno dolžino?

Eden prvih, ki se je ukvarjal z reševanjem te problematike, je bil Samuel Morse, ki ga bolj poznamo kot izumitelja Morsejeve abecede, pri kateri je uporabljal zapise simbolov spremenljivih dolžin. Znano je, da stiskalne metode strmijo k čimvečji odpravi redundatnih podatkov. Eden najpopularnejših načinov odprave je uporaba zapisov simbolov spremenljivih dolžin, določenih glede na verjetnost pojavitve, ki jo ugotovimo kot del statistike vhodnih podatkov. Metodam s predhodnim izračunavanjem pravimo statistične metode.

Med tovrstne metode uvrščamo še kodiranje Shannon-Fano, kodiranje Huffman (poglavje 2.2.7), MNP5 ter MNP7.

2.2.5. Metode s slovarjem

Ocenjevanje modela za stiskanje podatkov se izraža s kvaliteto stiskanja, ki jo lahko model ustvari. Metode s slovarjem ne uporabljajo statističnih modelov, niti zapisov spremenljivih dolžin (lahko kot nadgradnja metode), ampak kodirajo nize simbolov na podlagi že obstoječih nizov v slovarju. Slovarji se lahko med samim procesom sproti prilagajajo, dovoljujejo dodajanje nizov, ne pa tudi brisanja le-teh (rezen praznjenje slovarja), lahko pa so statični in ne dovoljujejo sprememb. Algoritmi tovrstnih metod se zelo dobro obnesejo tudi na slikovnih in audio formatih, saj temeljijo na ponavljanju določenih nizov simbolov.

Preprost primer statičnega slovarja, je slovar slovenskega jezika, ki vsebujejo približno 100.000 nizov, katerega lahko uporabimo pri stiskanju slovenskega besedila. Niz znakov (ločen s presledkom) preberemo iz vhodnih podatkov in ga skušamo poiskati v slovarju. Če ga najdemo, zapišemo na izhod zaporedno številko niza v slovarju, v nasprotnem primeru pa nespremenjen niz simbolov. V rezultatu (izhodni podatki) imamo vsebovane zaporedne številke nizov in t.i. gole nize, za katere je pomembno, da jih med seboj znamo ločiti. Ena izmed rešitev je, da si zamislimo poseben bit, ki bo določeval ali je podatek zaporedna številka ali niz. Takšen bit je potrebno izbrati glede na število možnih nizov v slovarju.

Z uporabo statičnega slovarja bomo tako dosegli optimalno stiskanje le, če bodo vhodni podatki oblikovani jezikovno slovensko. Ob preostalih nizih, ki so bolj specifični, npr. programska koda, bi bili rezultati negativni, saj takšni podatki vsebujejo nize simbolov, ki niso v slovarju slovenskega jezika. Kot boljša izbira se izkaže slovar, ki dovoljuje sprotno prilagajanje med samim procesom stiskanja.

Slovar, ki omogoča kasnejše modificiranje, je na začetku procesa lahko prazen ali pa vsebuje privzet (majhen) nabor nizov. Sprotno dodajanje novih nizov in brisanje starih pomeni počasno iskanje nizov (gledano performančno). Metoda z vgrajeno iteracijo naj bi za vsak obstoječ niz simbolov v vhodnih podatkih izvedla iskanje le-tega v slovarju in v primeru uspeha na izhod zapisala zaporedno število, v primeru neuspeha pa niz in ga hkrati dodala tudi v slovar (za kasnejša iskanja). Ob koncu vsake takšne iteracije mora preveriti, kateri izmed starejših nizov je najbolj primerem za odstranitev iz slovarja. Zadeva se sliši komplicirano, ima pa dve prednosti:

1) Gre za iskanje niza s primerjanjem in ne numeričnimi operacijami, ki so performančno zahtevnejše.

2) Razširjevalec je preprostejši (asimetrična metoda), saj ne potrebuje nobenega primerjanja, ugotoviti mora le ali gre za niz ali zaporedno število, v primeru slednjega, z njegovo pomočjo najde niz, s katerim ga nadomesti na izhodu. Ne potrebuje nikakršnih primerjanj ali iskanj.

V letih 1977 in 1978 je raziskovalcema Jacobu Zivu in Abrahamu Lampelu uspelo izumiti prvi metodi, ki sta uporabljali slovar, to sta LZ77 in LZ78. Njune ideje so bile kasneje izvor inspiracij za mnogo raziskovalcev, ki so metode generalizirali in kombinirali z RLE in statističnimi metodami. Njihov namen je razviti čimbolj uporabno brezizgubno metodo, namenjeno besedilom, slikam in glasbi.

2.2.6. Ostale metode

Zgoraj smo opisali metode kategorij RLE, statističnih metod in metod s slovarjem. Obstajajo pa tudi metode, ki jih ne moremo preprosto klasificirati, kot so:

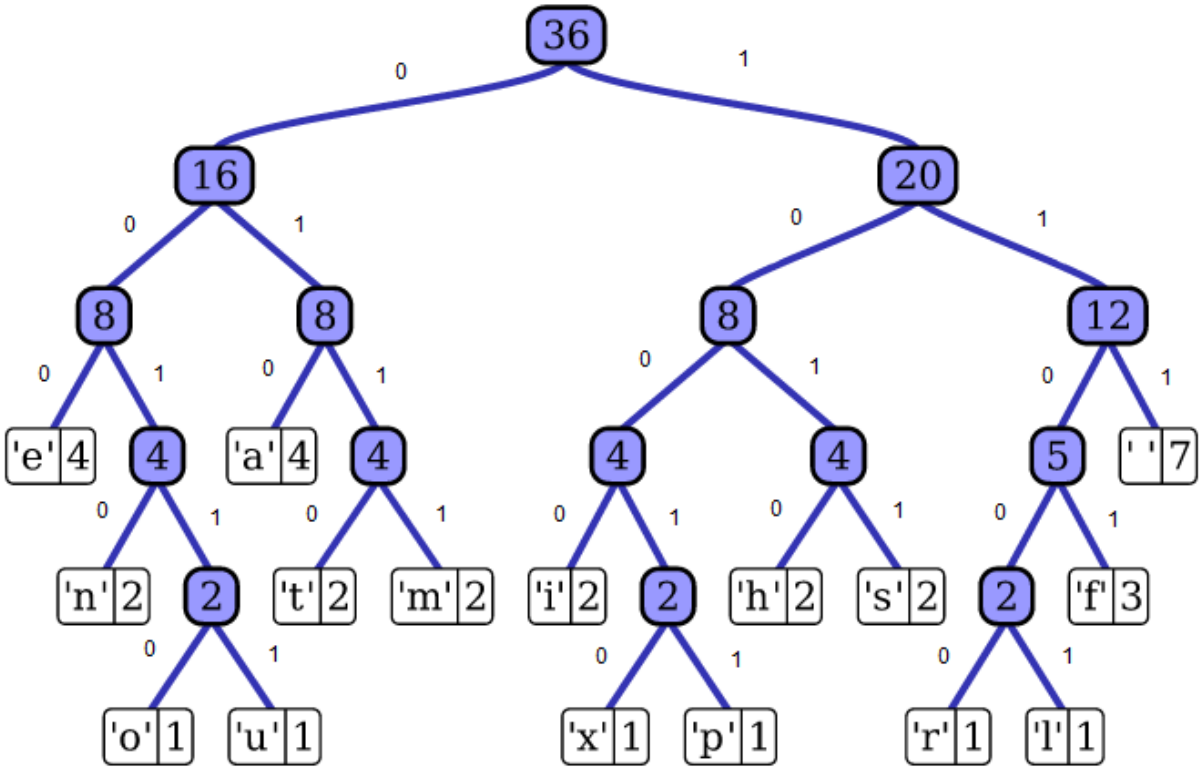
a) metoda Burrows-Wheeler,
b) tehnika simbolnega rangiranja,
c) metoda ACB,
d) metoda redkih binarnih nizov,
e) metoda FHM (Fibonacci, Huffman in Markov),
f) dinamično Markovo kodiranje.

2.2.7. Metoda Huffman

Metoda Huffman [1,8] je namenjena predstavitvi podatkov s čim krajšim zapisom. Mi smo jo uporabili pri algoritmih Re-Pair in pri metodi stiskanja s pomočjo bisekcije, kjer nam je koristila pri iskanju najkrajšega zapisa preostalih simbolov in/ali slovarja.

Metoda deluje na nivoju 8-bitnih znakov, in sicer poišče njihove frekvence pojavitev v izvornih podatkih ter jih predstavi s pomočjo drevesa, ki ga gradi sproti. Drevo ima korenski člen, ki vsebuje vsoto vseh frekvenc pojavitev (količino podatkov). Sami znaki so vedno predstavljeni kot zadnji členi drevesa (listi), in si nivojsko sledijo od vrha proti dnu, razporejeni od najpogostejših do najredkejših. Dva člena skupaj tvorita člen, ki je vsota njunih pojavitev, ti pa se rekurzivno seštevajo do korenskega.

Pri iskanju zapisov posameznih simbolov vsaka povezava, ki je na poti do simbola, predstavlja en bit, običajno leve povezave predstavljajo bit 0 in desne bit 1. Na ta način lahko predstavimo podatke z zelo kratkim zapisom, saj je zagotovljeno, da imajo pogostejši znaki krajši zapis (višje v drevesu), redkejši pa daljšega (nižje v drevesu) in vedno enolično določenega. Na koncu je na izhod potrebno zapisati drevo in zaporedje novonastalih zapisov simbolov, da jih kasneje lahko ponovno razširimo.



Slika 7: Primer Huffman-ovega drevesa

2.3. Algoritem LZW

Algoritem LZW je znana različica metode LZ78, razvita s strani Terrya Welcha leta 1984 [6,2]. Njena glavna novost je bila odstranitev zapisa obravnavanega simbola iz izhodnih podatkov. Izhodni podatki vsebujejo samo kazalce (zaporedna števila) zapisov v slovarju. Metoda se začne z inicializacijo slovarja z vsemi možni alfanumeričnimi simboli, ki zasedajo 8 bitov, intervala od 0 do 255 (prvih 256 zapisov v slovarju). To se zgodi še preden se začne postopek branja vhodnih podatkov. Zaradi že inicializiranega slovarja bo prvi vhodni znak zagotovo najden v slovarju. To je razlog, zakaj metoda LZW lahko uporablja kazalce brez zapisov znakov, kot to počneta metodi LZ77 in LZ78.

Algoritem LZW v osnovi ni algoritem, ki bi si pomagal s kontekstno neodvisnimi gramatikami. Zasnovan je tako, da si ga lahko predstavljamo kot takšnega, saj nizi, ki jih shranjuje v slovar, nastajajo po principu produkcij kontekstno neodvisnih gramatik. To se dogaja zaradi dejstva, da shranjuje samo novonastale nize, kar pomeni, da se niz, ki še ne obstaja v slovarju, zagotovo razlikuje samo v zadnjem dodanem simbolu od niza, ki je že v slovarju, upoštevajoč da je novi niz dolžine najmanj dveh simbolov.

2.3.1. Stiskanje

Princip metode LZW je takšen, da na vhodu bere simbole enega za drugim in jih združuje v niz *I*. Po vsakem prebranem simbolu in združitvi k nizu *I*, v slovarju poišče trenutni niz *I*. Dokler niz *I* že obstaja v slovarju, se postopek ponavlja vse do trenutka, ko prebran simbol *x* povzroči neuspešno iskanje niza v slovarju (niz *I* obstaja, ampak niz *Ix* ne obstaja). Na tem mestu algoritem na izhod zapiše zaporedno število niza *I* v slovarju, shrani niz *Ix* v slovar za nadaljna iskanja in nizu *I* priredi vrednost *x*.

Psevdokoda za stiskanje LZW:

```
STRING = get input character
WHILE there are still input characters DO
    CHARACTER = get input character
    IF STRING+CHARACTER is in the string table then
        STRING = STRING+character
    ELSE
        output the code for STRING
        add STRING+CHARACTER to the string table
        STRING = CHARACTER
    END of IF
END of WHILE
output the code for STRING
```

Opis postopka na primeru niza *mama*:

0) Slovar vsebuje vseh 256 8-bitnih simbolov (od 0 do 255).

1) Prvi prebran znak je *m*, ki že obstaja v slovarju (z zaporednim številom 109, ki je njegova ASCII vrednost). Naslednji prebran znak je *a*, ampak niza *ma* ni v slovarju, zato algoritem stori naslednje: (1) na izhod izpiše število 109, (2) v slovar doda niz *ma* (z zaporednim številom 256) in (3) nizu *I* priredi znak *a*.

2) Znak *m*, kot naslednji, povzroči nastanek niza *am*, ki ga ni v slovarju, zato ponovi korake iz točke 2), le da tokrat na izhod izpiše število 97 (ASCII vrednost znaka *a*), in v slovar doda niz *am* (zaporedno število 257), ter *I*-ju priredi znak *m*.

Začetnih 256 vnosov v slovar uporabi celotno območje 8-bitnih zapisov, kar pomeni, da je od samega začetka potrebno uporabljati več kot 8-bitov za zapisovanje zaporednih števil. Običajno LZW uporablja 16-bitno zapisovanje, se pravi ima na razpolago $2^{16} = 65536$ števil. Še vedno je to velika omejitev, saj se slovar zelo hitro polni. To je bil že problem metode LZ78 in njene rešitve uporablja tudi LZW:

1) Najpreprostejša rešitev je takšna, da slovar v točki, ko je popolnoma zapolnjen, postane statičen, kar pomeni, da se ne more dodajati nobenih nizov več. Od tu naprej stiskalno razmerje pada.

2) Običajna rešitev pa je, da na tej točki slovar izpraznimo, vstavimo začetnih 256 vnosov in nadaljujemo s postopkom stiskanja. Ta rešitev vhodne podatke v bistvu razdeli v več blokov, za vsakega svoj slovar. Če se vsebina vhodnih podatkov razlikuje od bloka do bloka, potem ta način stiskanja povzroči še večji učinek, saj se iz slovarja izbrišejo nizi, ki se najverjetne sploh ne bodo pojavili več oz. imajo manjšo verjetnost pojavitve. Lahko rečemo, da ta rešitev implicitno predpostavlja, da bodo prihajajočim simbolom bolj koristili novi nizi kot stari (enako predpostavlja metoda LZ77).

3) Ena izmed bolj kompleksnih rešitev je tudi uporaba stiskanja UNIX.

4) V trenutku, ko je slovar zapolnjen, lahko algoritem namenjen določevanju najmanj uporabljenih nizov, izbere seznam tistih, ki se lahko odstranijo ter prepusijo prostor novim. V praksi ne obstaja algoritem, ki bi znal po pravilih določiti katere in koliko nizov odstraniti.

Katerokoli izmed zgoraj naštetih rešitev je uporabljena pri stiskanju, je enako potrebno uporabiti pri razširjanju. To pomeni večjo kompleksnost razširjevalca.

2.3.2. Razširjanje

Razširjanje pri metodi LZW poteka skorajda enako kot stiskanje. Razširjevalec začne z začetnimi vnosi simbolov v slovar (običajno jih je 256), nato bere vhodne podatke, ki so zaporedna števila (kazalci) zapisov v slovarju, na podlagi katerih pridobi razširjen niz, ki jim pripada in ga zapiše na izhod. Sproti gradi slovar na enak način kot to poteka pri stiskanju, tako da včasih lahko rečemo, da sta proces stiskanja in proces razširjanja sinhronizirana.

V prvem koraku razširjevalec z vhoda prebere število in ga uporabi za razširitev v niz *I*. *I* je niz simbolov, ki jih nato zapiše na svoj izhod. Niz *Ix* je potrebno dodati v slovar, ampak simbol *x* je v tem trenutku še neznan. To bo prvi simbol naslednjega razširjenega niza, ki bo najden v slovarju.

V vsakem naslednjem koraku (po prvem), pa razširjevalec prebere naslednje število, v slovarju poišče niz *J*, ga izpiše na izhod, izbere prvi simbol niza *J* kot simbol *x* in doda niz *Ix* v slovar (če še ni dodan). Nazadnje priredi nizu *I* niz *J* in ponovi iteracijo.

Psevdokoda za razširjanje pri metodi LZW:

```
Read OLD_CODE
output OLD_CODE
CHARACTER = OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    IF NEW_CODE is not in the translation table THEN
        STRING = get translation of OLD_CODE
        STRING = STRING+CHARACTER
    ELSE
        STRING = get translation of NEW_CODE
    END of IF
    output STRING
    CHARACTER = first character in STRING
    add OLD_CODE + CHARACTER to the translation table
    OLD_CODE = NEW_CODE
END of WHILE
```

Obstaja tudi možnost, da prebranega števila še ni v slovarju (tabela 1, »!«). Temu dogodku sledita naslednja dva koraka, ki rešita ta problem: (1) razširjevalec poišče v slovarju niz, ki pripada zadnjemu uspešno najdenemu številu, (2) in mu doda še prvi simbol najdenega niza.

STISKANJE			
NIZ: mamamamammamaama			
niz I	**ali je niz I v slovarju?**	**nov vnos**	**izhod**
m	DA		
ma	NE	256-ma	77 (m)
a	DA		
am	NE	257-am	65 (a)
m	DA		
ma	DA		
mam	NE	258-mam	256 (ma)
m	DA		
ma	DA		
mam	DA		
mama	NE	259-mama	258 (mam)
am	DA		
amm	NE	260-amm	257 (am)
m	DA		
ma	DA		
mam	DA		
mama	DA		
mamaa	NE	261-mamaa	259 (mama)
a	DA		
am	DA		
ama	NE	262-ama	257 (am)
a	DA		
a			65 (a)

REZULTAT: 77 65 256 258 257 259 257 65

RAZŠIRJANJE			
NIZ KOD: 77 65 256 258 257 259 257 65			
prebrana koda	**ali je število v slovarju?**	**nov vnos**	**izhod**
77	DA		m
65	DA	256-ma	a
256	DA	257-am	ma
258	**NE**	**258-mam**	**mam**
257	DA	259-mama	am
259	DA	260-amm	mama
257	DA	261-mamaa	am
65	DA	262-ama	a

REZULTAT: mamamamammamaama

!

Tabela 1: Primer stiskanja in razširjanja z metodo LZW

Struktura slovarja je lahko definirana na več načinov. Najbolj uporabljeni so slovarji s tabelo nizov, kjer je niz shranjen v polju tabele, njegovo število pa je zaporedno število polja. Najbolj učinkovita struktura slovarja, je struktura drevesa, kjer zapis vsebuje zaporedno število in posamezni simbol, nizi pa se sestavljajo od korenskega zapisa navzdol. Problem takšnih dreves je, da niso binarna ampak večkrat razvejana. »Starševski« člen ima lahko več »otroških«, poleg tega, da se lahko členi posameznih simbolov ponavljajo (za en simbol več členov). To lahko rešimo s tabelo kazalcev (poglavje 3.3.) na zapise (nabor vseh možnih simbolov) in zgradimo virtualno drevo le s kazalci na zapise v tabeli. Pomagamo si lahko tudi s t.i. »hash« vrednostmi, ki omejujejo območje iskanja.

Metoda LZW je bila uradno sicer izdana leta 1984, namenjena stiskanju podatkov, mnogim raziskovalcem pa je dala navdih za nadaljne različice, kot so: LZMW, LZAP, LZY, LZP. Metoda LZMW rešuje npr. težavo, ki jo ima LZW, da v primeru niza, sestavljenega iz milijona znakov *a* (velika redundanca), zgradi slovar, ki vsebuje najdaljši niz z »le« 1414 znakov *a*.

Pojavila se je tudi ideja o relativnem stiskanju, ki je še posebej uporabna pri LZW grafičnem stiskanju, če se slikovne točke na sliki ne razlikujejo preveč, saj takšna metoda temelji na razlikah v vrednostih med posameznimi točkami.

2.4. Algoritem Re-Pair

Algoritem Re-Pair (Recursive-Pairing) sta zasnovala Larsson in Moffat [3]. Osnovni princip metode je nadomestitev najbolj frekventnih parov simbolov v izvornem nizu z novimi simboli. To ponovi tolikokrat, da v samem nizu ne obstaja več para, ki bi se pojavil vsaj dvakrat. Sam potek korakov je takšen:

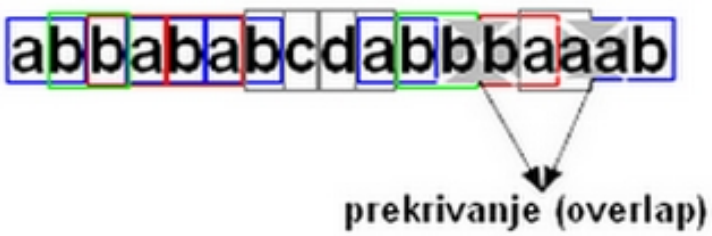
1) Identifikacija simbolov *a* in *b*, kot *ab*, ki se največkrat pojavita v nizu kot soseda (par). V primeru da ne obstaja več para, ki bi se pojavil več kot enkrat, potem konča.

2) Uvede nov simbol *A* in zamenja vse pojavitve para *ab* z novonastalim *A*-jem, nato ponovi korak iz prejšnje točke.

Izvorni niz se zmanjša na zaporedje simbolov, kjer vsak simbol predstavlja znak ali rekurzivno definirane simbole parov (zamenjave). Ničelna entropija končnega niza je zadnji korak v procesu stiskanja, predzadnji pa je prenos slovarja s frazami, ki predstavljajo pravila zamenjav.

V primeru nastopa večih parov z enako frekvenco pojavitev, ni točno določeno po katerem vrstnem redu naj bi se pari nadomestili. Sama definicija algoritma tega ne predvideva, saj ta odločitev skorajda nima vpliva na končni rezultat. Običajna implementacija rešuje ta problem tako, da v tem primeru izbere zadnjega izmed najdenih parov.

Posebno pozornost je potrebno nameniti zaporedjem enakih simbolov v nizu s tremi ali več pojavitvami. Sama definicija algoritma prepoveduje prerkrivanje parov (slika 8), tako da zaporedje *aaaa* ne vsebuje treh parov, temveč le dva.



Slika 8: Identifikacija parov v algoritmu Re-Pair

Odstranjevanje redundance bi bilo pri tem algoritmu zelo težko, saj bi moral v takem primeru preverjati načine pojavitev simbolov, v kakšnem vrstnem redu se pojavljajo, kolikokrat se pojavi posamezen simbol, kakorkoli, simbole bi bilo treba preverjati kot kontekstno odvisne. Obstajajo teorije, ki dokazujejo maksimalno stiskanje. Guttman in Bell sta ugotovila, da je verjetnost vsake fraze odvisna od zadnjega simbola prejšnje fraze.

STISKANJE PODATKOV Z ALGORITMOM RE-PAIR

ZAČETNI NIZ: **mamamamammmamaama**

1. pregled		
NIZ: mamamamammmamaama		
SLOVAR:	ma(A)	**2**
	am	1

2. zamenjava **(ma)**		
NIZ: AAmamammmamaama		
SLOVAR:	ma(A)	0
	am	0
	AA	1
	Am	1

3. pregled		
NIZ: AAmamammmamaama		
SLOVAR:	ma(A)	**2**
	am	1
	AA	1
	Am	1

4. zamenjava **(ma)**		
NIZ: AAAAmmamaama		
SLOVAR:	ma(A)	0
	am	0
	AA(B)	**2**
	Am	1

5. zamenjava **(AA)**		
NIZ: BBmmamaama		
SLOVAR:	ma(A)	0
	am	0
	AA(B)	0
	Am	0
	BB	1
	Bm	1

6. pregled		
NIZ: BBmmamaama		
SLOVAR:	ma(A)	**2**
	am	1
	AA(B)	0
	Am	0
	BB	1
	Bm	1
	mm	1

7. zamenjava **(ma)**		
NIZ: BBmAAama		
SLOVAR:	ma(A)	0
	am	0
	AA(B)	1
	Am	1
	BB	1
	Bm	1
	mm	0

8. pregled		
NIZ: BBmAAama		
SLOVAR:	ma(A)	1
	am	1
	AA(B)	1
	Am	1
	BB	1
	Bm	1
	mm	0

ZAPIS: **ma(A),AA(B),BBmAAama**

Tabela 2: Stiskanje podatkov z algoritmom Re-Pair

Naslednja možnost, ki bi dosegala boljše rezultate je, da končno zaporedje simbolov ne vsebuje ponavljajočih se parov simbolov, niti ne kateregakoli para, ki je že v slovarju.

Oba pogoja sta zelo kompleksna za implementacijo, zato jih je smiselno uporabiti le, če je naš končni cilj čimboljše stiskalno razmerje. V takšnem primeru je odločitev za kontekstni mehanizem boljša za algoritem.

Na koncu je potrebno slovar s frazami (preslikavami) zapisati s karseda majhno količino podatkov. Za to obstaja več modelov, po katerih lahko zakodiramo slovar in ga kasneje tudi dekodiramo:

- model Bernoulli,
- direktno oštevilčenje parov,
- kodiranje s pomočjo interpolacije.

Za prihranek časa pri stiskanju niza lahko opravimo manjšo spremembo opisanega, ki pa nima bistvenega vpliva na sam potek razširjanja. Zamenjevanje parov lahko opravljamo kar med branjem simbolov in to počnemo toliko časa, dokler obstaja par, ki se v tistem trenutku pojavi več kot enkrat.

Tudi proces razširjanja omogoča nekaj modifikacij, ki imajo predvsem performančne prednosti. Najlažja strategija razširjanja je, da za vsak novonastali simbol, zavezujoč vrstnega reda prečkanja hierahije fraz, na izhod zapišemo vsak list hierarhije, ki ga najdemo. Alternativa temu pa je, da vse fraze razpakiramo v nize, ki jih direktno zapišemo na izhod, kar prispeva k pospešitvi procesa.

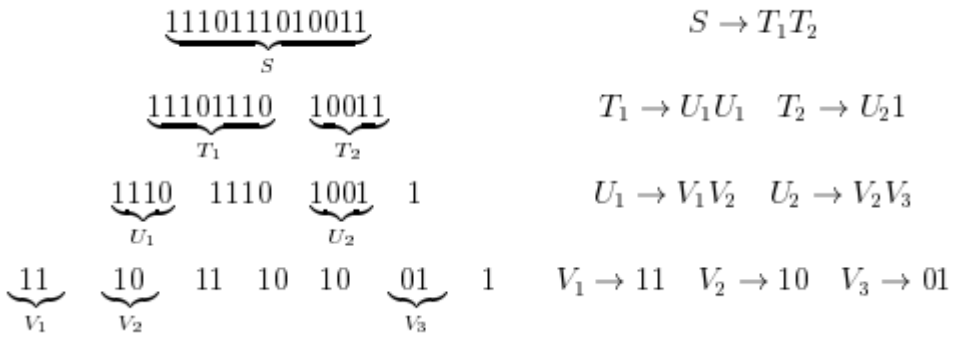
RAZŠIRJANJE PODATKOV Z ALGORITMOM RE-PAIR			
ZAČETNI NIZ: **BBmAAama**			
SLOVAR:	**ma(A)**		
	AA(B)		
1. zamenjava (B)		2. zamenjava (A)	
NIZ: AAAAmAAama		NIZ: mamamamammamaama	
SLOVAR:	ma(A)	SLOVAR:	ma(A)
	AA(B)		AA(B)
REZULTAT: **mamamamammamaama**			

Tabela 3: Razširjanje podatkov z algoritmom Re-Pair

2.5. Metoda za stiskanje podatkov z bisekcijo

Metoda z bisekcijo je nastala s strani štirih raziskovalcev: Kieffer, Yang, Nelson in Cosman [4]. Namenjena je za stiskanje binarnih podatkov, in sicer dolžine 2^k bitov. Vsak byte (znak) je takšne dolžine, saj obsega 8 bitov ($2^3 = 8$), zato lahko stiskamo tudi besedila.

Stiskalec izvede iteracijo natanko k-krat, in sicer tako, da izbere največje število j, da velja enačba $2^j < l$, kjer je l dolžina vhodnega niza. Nato vhodni niz razdeli na dva podniza, prvi je dolg 2^j bitov, in drugi preostanek dolžine. To ponavlja toliko časa dokler je novo nastali podniz daljši od 1 bita. Ob vsaki delitvi niza, ustvari za vsak različen niz, ki je daljši od 1 bita, nov simbol, ki ga kasneje uporabi za generiranje produkcij (slika 9).

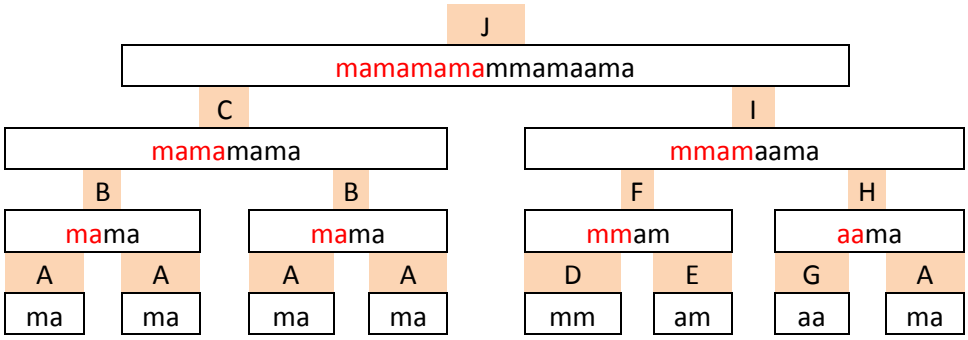


Slika 9: Prikaz delitve niza z bisekcijo

Pri metodi z bisekcijo je stiskalno razmerje odvisno od dolžine vhodnega niza ter kolikokrat se posamezne spremenljivke v gramatiki ponovijo.

STISKANJE PODATKOV S POMOČJO BISEKCIJE

NIZ: **mamamamammamaama**



SLOVAR:		
	1.	ma (A)
	2.	AA(B)
	3.	BB(C)
	4.	mm(D)
	5.	am(E)
	6.	DE(F)
	7.	aa(G)
	8.	GA(H)
	9.	FH(I)
	10.	CI(J)

ZAPIS: **ma(A),AA(B),BB(C),mm(D),am(E),DE(F),aa(G),GA(H),FH(I),CI(J)**

Tabela 4: Drevo stiskanja in razširjanja podatkov s pomočjo bisekcije

Proces razširjanja pri tej metodi ni kompliciran, saj je potrebno le prebrati vsa pravila in jih razširiti, pri čemer mora biti znano pravilo, ki je prvo (najvišje, korensko).

3. Implementacija algoritmov

Sama implementacija algoritma je predvsem odvisna od programerja, ki se bo odločil kakšen pristop bo uporabil, saj na različnih področjih obstaja več možnosti implementacije z enakim rezultatom. Izbira je odvisna od cilja, ki ga želimo doseči, od performančnega vpliva, navsezadnje pa tudi od programskega jezika, ki ga izbere programer.

Pri programiranju lahko naletimo na mnogo težav, nekatere so predvidene, druge ne. Za predvidene težave imamo običajno že vnaprej pripravljene načrte, kako se jim bomo izognili, ob nepredvidenih pa si je potrebno zamisliti načrt, ki bo težavo odpravil, a obenem obdržal preostale začrtane poti.

Naša aplikacija, ki je namenjena testiranju algoritmov za stiskanje, vsebuje določena področja, brez katerih ne more funkcionirati. Vemo, da običajno stiskamo podatke, zapisane v datoteki, ki jo je potrebno odpreti, prebrati, lahko vanjo tudi kaj zapisati. Zato je jasno, da bo to neka zaključena celota oz. modul, ki bo vse to znal opraviti.

Potem imamo v sami implementaciji opravka s hranjenjem in prenašanjem mnogih informacij, za kar je potrebno definirati strukture in objekte, ki morajo vsebovati vnaprej začrtane parametre (procedure, funkcije, ...). Naloga načrtovanja struktur in objektov ni pretirano lahka, saj je potrebno imeti v glavi vsaj približen potek prenosa in obdelovanja podatkov, informacij.

Poleg vsega pa je potrebno skrbeti tudi za performance aplikacije. Aplikacija s počasnim odzivanjem ni uporabna. Tudi aplikacija, ki je omejena s količino podatkov, za katero še lahko zagotovi varno obdelavo, v večini primerov ne pride v poštev. V računalništvu je količina podatkov skorajda vedno neomejena, zato je potrebno predvideti rešitev za takšen problem. V današnjem času ni težav s spominom (RAM), pa vendar lahko omenimo, da ni pametno pretirano »razmetavati« z njim, saj se nam takšna »naložba« skorajda ne obrestuje. Zato je potrebno uporabiti pravilne pristope, ki bodo poskrbeli tako za spomin, časovno odzivnost, kot tudi za uporabnika (namigi, pomoč).

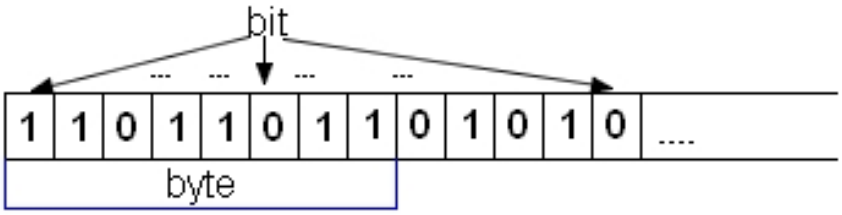
3.1. Branje in pisanje datotek

Branje in pisanje datotek je eden izmed osnovnih modulov komunikacije med datoteko in aplikacijo. Tako pri branju kot tudi pisanju je potrebno zagotoviti, da se noben podatek ne izgubi, da poskrbimo za zadostno količino prostora v datotečnem sistemu in ne dovolimo brisanja podatkov brez dovoljenja uporabnika. Branje in pisanje datotek ima tako določene omejitve, ki jih je potrebno upoštevati.

3.1.1. Branje datoteke

Branje datotek je možno izvesti na več načinov, lahko celotno datoteko naložimo v RAM ali pa jo beremo sproti (streaming). Večkrat je smiselno datoteko odpreti za branje z opombo (flag), imenovano »Samo za branje« (Read only), kar pomeni, da med samim procesom uporabe vanjo ne moremo pisati.

Nekateri že obstoječi moduli za komunikacijo z datotekami imajo omejitev pri branju, in sicer je najmanjša možna enota za branje byte (poglavje 3.2), v primeru ko potrebujemo bite, pa je potrebno implementirati medpomnilnik (buffer) (slika 10), kjer shranjujemo byte kot bite, ki so začasno na voljo. Točno to smo morali storiti mi, saj programski jezik Pascal nima direktnega dostopa do bitov preko obstoječih modulov. Medpomnilnik mora imeti velikost najmanj enega byte-a (8 bitov) in je pametno, da je pomikajoč, saj se s tem izognemo težavam pri branju različnih dolžin podatkov.



Slika 10: Medpomnilnik

3.1.2. Pisanje v datoteko

Pisanje v datoteko je moč izvajati samo, če imamo v datotečni strukturi na mestu kamor želimo pisati, pisalne pravice. Pisanje datotek je namenjeno predvsem hranjenju informacij, zato je potrebno zagotoviti konsistenčnost podatkov in si rezervirati prostor na disku, ki je ekskluzivno zaklenjen samo za izbrano aplikacijo.

Tudi pri pisanju sta možna dva načina zapisa, lahko vse podatke zapišemo naenkrat ali pa sprotno zapisovanje, bodisi po byte-ih ali bit-ih, le da slednja rešitev ni povsod mogoča, zato je potrebno prilagoditi prvo, in sicer z medpomnilnikom (slika 10). Medpomnilnik smo uporabili tudi v našem primeru, saj smo potrebovali zapisovanje podatkov po bitih.

3.2. Nizi, znaki, bajti, biti

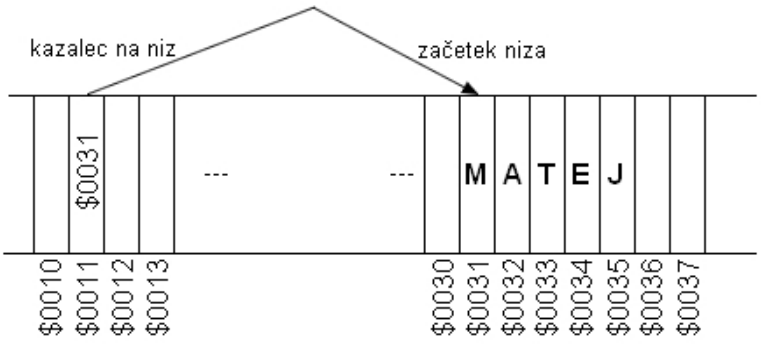
Nizi, znaki, bajti, biti so enote podatkov, ki lahko hranijo različno količino informacij. Znano je, da je bit najmanjša enota v računalništvu, saj ima lahko samo dve stanji, 0 ali 1. Byte je enota, v kateri se ponavadi merijo velikosti datotek. Byte je sestavljen iz 8 bitov (slika 10), kar pomeni da ima lahko $2^8 = 256$ vrednosti (stanj) na intervalu od 0 do 255. Z byte-i so predstavljeni tudi znaki (ASCII vrednosti), ki se lahko združujejo v nize. Znak je osnovna enota v besedilu, poznamo pa vidne (črke) in nevidne znake (prehod v novo vrstico, presledek). Nizi dolžinsko niso omejeni, lahko si jih predstavljamo kot tabele, ki vsebujejo znake. Nizi se običajno končajo s posebnim znakom NULL, ki označuje konec niza.

3.3. Kazalci

Kazalci so namenjeni naslavljanju določenih struktur, z namenom varčevanja pomnilnika. Tako lahko enako informacijo hranimo v pomnilniku kot strukturo le na enem mestu, do nje pa dostopamo z večih mest. Kazalec zavzame v pomnilniku samo en naslov, dočim struktura lahko tudi do 10 ali več (slika 11), to je bistvo kazalcev. Takšen primer smo imeli pri implementaciji slovarja z drevesom, kjer so bili simboli shranjeni v tabeli, do njih pa smo dostopali z različnih mest drevesa, preko kazalcev.

Kazalec ima določen tip podatka na katerega lahko kaže. Če želimo, da bo kazalec kazal na strukturo *X*, potem mu moramo to tudi povedati v sami deklaraciji. Tako vemo, da od kazalca ki kaže na število, ne moremo pričakovati niza in obratno. Preko kazalcev pa lahko izvemo tudi vsebino (število, niz), ne le naslova strukture.

V nekaterih programskih jezikih poznamo tudi večnivojske kazalce (programski jezik C), kar pomeni, da kazalec kaže na kazalec, ki lahko ponovno kaže na naslednji kazalec, ni nujno enakega tipa. Običajno je smiselno nivo kazalcev omejiti, saj lahko pride do velikih zmešnjav in posledično tudi do težavnega programiranja in nepreglednosti.

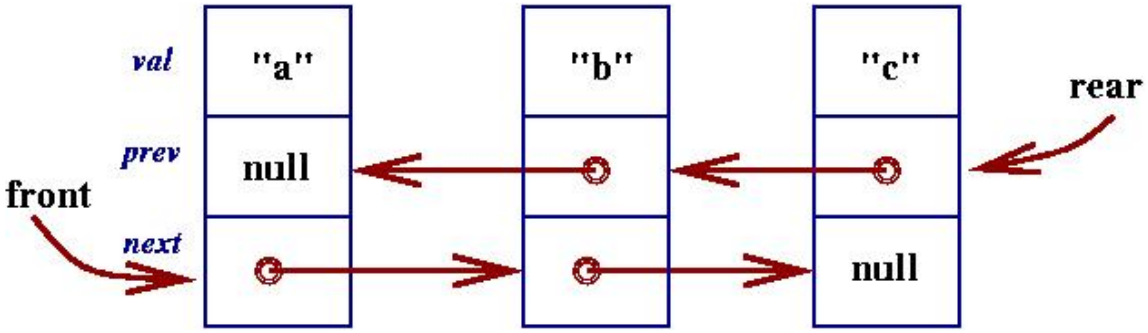


Slika 11: Kazalec v pomnilniku

3.4. Dvosmerni seznami

Dvosmerne sezname (linked lists) uporabljamo v aplikacijah kot strukturo za hitro pomikanje po zapisih. Primer parov v algoritmu Re-Pair, kjer je hitrost ključna, saj ne želimo par vsakokrat iskati po nizu. Ključ do uspeha so dvojne povezave med zapisi (slika 12), ki vsebujejo kazalec na naslednji zapis in kazalec na prejšnji zapis, tako je zagotovljeno pomikanje brez iskanja.

Poznamo tudi enosmerne sezname, kar pomeni, da se lahko pomikamo samo v eno smer, takšni seznami so redkeje uporabljeni.



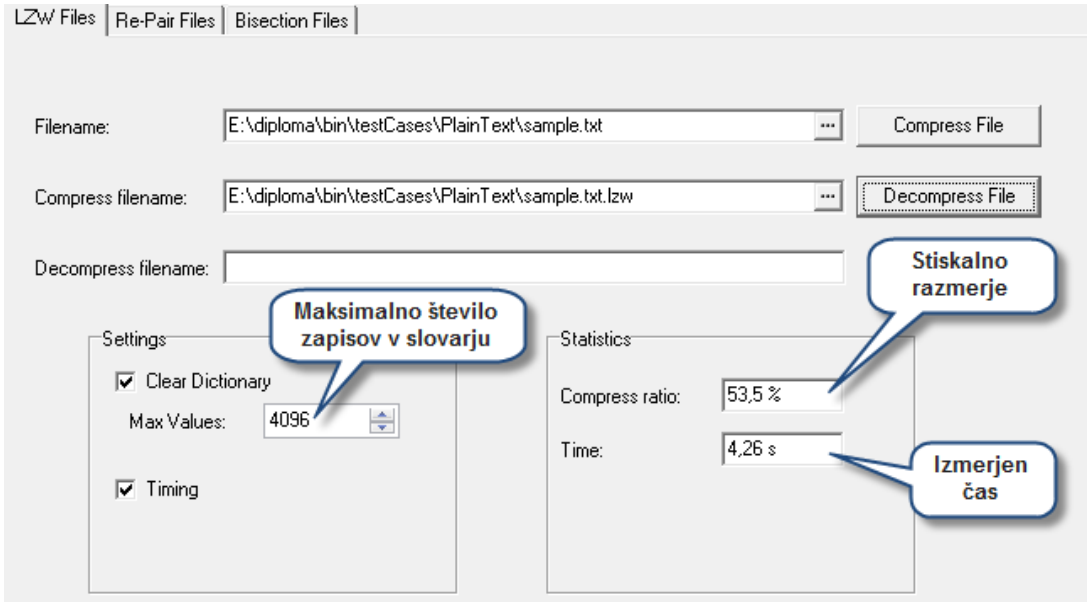
Slika 12: Dvosmerni seznam

Previdnost pri seznamih ni odveč, še posebej takrat, ko želimo kakšnega izmed zapisov odstraniti ali dodati, saj je potrebno popraviti vse kazalce, ki so na kakršenkoli način vpleteni. Na tem mestu si je potrebno seznam kot strukturo zelo dobro predstavljati. Možnost napake je velika, še težje pa jo je najti (ugotoviti vzrok napake) in nato odpraviti.

Seznami so namenjeni tudi hranjenju in prenašanju podatkov v obliki struktur ali objektov. S tem lažje vzpostavimo reference, saj so na tak način podatki vedno pri roki, kar pa bi bilo brez kazalcev (poglavje 3.3) veliko težje.

3.5. Algoritmi

Kot sem že v uvodu omenil, sem vse tri algoritme implementiral v jeziku Pascal v razvojnem okolju Delphi. Za vsakega posebej pa sem pripravil tudi grafični vmesnik (slika 13), ki služi lažjemu testiranju, saj omogoča enostavno izbiranje datotek, ki jih potrebujem za izvajanje operacij. Na njem se na desni strani po končanem procesu stiskanja/razširjanja pokažejo tudi statistični podatki, kot so stiskalno razmerje, ki pomeni, koliko odstotka prostora zaseda novonastala datoteka. Če na levi strani nastavitev vključimo merjenje časa, se le-ta ob končanem procesu tudi pokaže, kot približek časa trajanja v sekundah. Pri vmesniku za testiranje algoritma LZW je še posebnost, dodatna možnost vključitve in omejitve maksimalnega števila zapisov v slovarju.



Slika 13: Uporabniški vmesnik aplikacije

Pri vseh algoritmih sem moral uporabiti strukture, ki se običajno uporabljajo pri takšnih aplikacijah, in sicer kazalce, tabele, dvosmerne sezname, medpomnilnike, ipd. Pri vsakemu posebej sem definiral tudi interne strukture za prenos podatkov. Pri nekaterih sem to storil na način večnamenske uporabe, tako da sem jih lahko uporabil tudi v preostalih metodah.

Moj glavni cilj je bil, da dosežem čimboljše rezultate na stiskalnem učinku, kar bi lahko preprosto povedali tako, da želim predstaviti določeno količino vhodnih podatkov s čim manj podatki (čim manj prostora), vendar tako, da jih kasneje lahko ponovno razširim. To je na implementacijo vplivalo tako, da nisem upošteval optimalnih načinov, ki bi pripomogli k hitrosti, sem pa želel uporabiti čimveč pristopov, ki bi mi pripomogli k končnemu razmerju. Zato sem pri algoritmih Re-Pair in metodi s pomočjo bisekcije pomagal tudi z metodo Huffman (poglavje 2.2.7), ki strmi k krajšim zapisom podatkov na podlagi simbolov in njihove frekvence pojavitev.

3.5.1. LZW

Algoritem LZW kar se tiče implementacije ni med najbolj zapletenimi, saj potrebuje le vhodne podatke, slovar, kamor shranjuje novo nastale nize ter izhod za sprotno zapisovanje novih vrednosti zaporednih števil (kazalci), ki predstavljajo stisnjene podatke.

Moja implementacija algoritma LZW uporablja zapisovanje simbolov s spremenljivimi dolžinami in omogoča določitev maksimalnega števila zapisov v slovarju. To pomeni, da se slovar ob maksimalnem številu nizov izprazni, kar posledično povzroči stiskanje podatkov po blokih. Na mestu izpraznjenja slovarja je le-to potrebno zabeležiti tudi v izhodnih podatkih, kar storim na način posebej rezerviranega simbola, ki pri razširjanju povzroči praznjenje slovarja. Ta način ima tudi prednosti, kot so manjša poraba spomina, hitrejše iskanje nizov, itd.

Za branje vhodnih podatkov potrebujemo vhodni tok (stream), ki zna brati podatke po byte-ih, saj algoritem LZW deluje na nivoju znakov (byte). V okolju Delphi že obstaja razred TFileStream, ki podpira komunikacijo z datotekami na nivoju byte-ov. Nadgradil sem ga v razred TLZWStreamReader in vanj vključil medpomnilnik, ki ga potrebujem pri preostalih dveh algoritmih.

Sam proces stiskanja in razširjanja poteka v razredu TLZWCompressor, kjer se izvajajo posamezne iteracije glavne zanke (slika 14), ki stiska vhodne podatke, išče nize v slovarju, jih vanj dodaja in na izhod izpisuje stisnjene podatke s pomočjo izhodnega toka.

```
Result := False;
FEncodeString := '';
if (FFileReader.Size > 0) then begin
  FFileReader.ReadBuffer(Character, 1);
  SetLength(FEncodeString, 1);
  FEncodeString[1] := AnsiChar(Character);
  while (FFileReader.Position < FFileReader.Size) do begin
    FFileReader.ReadBuffer(Character, 1);
    if (FEncodeDictionary.FindString(FEncodeString + AnsiChar(Character)) <> -1) then begin
      FEncodeString := FEncodeString + AnsiChar(Character);
    end
    else begin
      WriteBytesToFile(FEncodeString);
      AddStringToDictionary(FEncodeString + AnsiChar(Character));
      SetLength(FEncodeString, 1);
      FEncodeString[1] := AnsiChar(Character);
    end;

    if not FExecuteTiming and Assigned(FOnByteChange) then
      FOnByteChange(FFileReader.Position, FFileReader.Size);
  end;
  WriteBytesToFile(FEncodeString);
  Result := True;
end;
```

Slika 14: Izsek kode, glavna zanka stiskanja LZW

Slovar pri algoritmu LZW je preprost, saj zahteva le shranjevanje nizov, ki so predstavljeni kot zaporedje byte-ov. Načeloma bi to lastnost lahko izkoristil tako, da bi za slovar uporabil razred TStringList, ki omogoča shranjevanje znakovnih nizov in ne prav hitro iskanje le-teh, zato sem raje uporabil že obstoječo knjižnjico (TAVLTree), in ustvaril razred TLZWAVLDictionary, ki ima za osnovo AVL drevo, ta pa omogoča hitrejše iskanje nizov. Na začetku slovar napolnim z naborom eno-byte-nih nizov (vrednosti od 0 do 255) in rezerviranim simbolom (slika 15), ki pomeni praznjenje slovarja.

```
for i := 0 to 255 do
  AddString(AnsiChar(i));

//reserve clear table code
AddString(''); //index: 256
```

Slika 15: Inicializacija slovarja pri LZW

Za zapisovanje izhodnih podatkov sem uporabil ponovno razred TFileStream, katerega sem nadgradil v razred TLZWStreamWriter, ki ima vgrajen medpomnilnik, s čimer lahko zapisujem decimalne vrednosti s spremenljivimi bitnimi dolžinami (variable codes).

Bitne dolžine se določujejo glede na vrednost simbola (slika 16), saj določena dolžina omejuje nabor vrednosti, tako npr. z dolžino 9 bitov lahko zapisujemo vrednosti, ki so manjše od 512. Ko nastopi vrednost simbola, ki je zadnja v možnem naboru z že obstoječo bitno dolžino, potem se dolžina zapisa poveča za en bit. Pri razširjanju je postopek enak, le da bitno dolžino modificiramo korak prej.

```
function GetMinNumberOfBitsForDecimal(ADecimalNumber: Longint): Integer;
var
  LogValue: Extended;
begin
  LogValue := Log2(ADecimalNumber + 1);
  if (LogValue - Trunc(LogValue) > 0) then
    LogValue := LogValue + 1;
  Result := Trunc(LogValue);
end;
```

Slika 16: Izračun minimalne dolžine zapisa za določeno vrednost

Posebnost pri metodi LZW je ta, da lahko pri razširjanju podatkov pride do situacije, ko prebrane vrednosti (zaporednega števila) še ni v slovarju. Rešitev tega problema je opisana v poglavju 2.3.2 in sem jo tako tudi implementiral.

3.5.2. Re-Pair

Algoritem Re-Pair je glede implementacije bolj zahteven, saj je potrebno implementirati kar dva dvojna seznama, ki služita za hitro pomikanje po simbolih in parih. Poleg tega potrebuje tudi slovar parov, ki je tokrat implementiran v obliki »hash« tabele, saj tako lahko zagotovim hiter dostop do obstoječih parov.

Branje vhodnih podatkov je zagotovljeno s pomočjo razreda TLZWStreamReader, ki omogoča branje podatkov s podano bitno dolžino, v našem primeru 8, in je bistvu enaka kot, če bi brali byte. Vsak na novo prebran simbol dodam v dvojni seznam in ugotovim, kateri pari so se na novo pojavili, ter povečam njihove števce.

```
procedure CompressValue(var AProgram: TProgramSettings; AValue: Integer);
var
  Sequence: pSequenceNode;
  HashNode: pHashNode;
begin
  Sequence := InitSequenceNode(AValue);

  AddSequence(AProgram, Sequence);

  //if more then one sequence in list
  if (AProgram.StartSequence <> AProgram.EndSequence) then begin
    HashNode := AddPair(AProgram, AProgram.EndSequence.PreviousNode, AProgram.EndSequence);

    if (HashNode.Count >= 2) then begin
      //execute replacing (pair is occurs twice or more)
      ReplacePairs(AProgram, HashNode);
    end;
  end;
end;
```

Slika 17: Stiskanje prebrane vrednost in po potrebi izvajanje zamenjave parov

V primeru, ko je števec para večji od 1, kar pomeni, da se par pojavi več kot enkrat, potem nastopi proces zamenjave izbranega para v celotnem seznamu (slika 17), kjer je potrebno poiskati prvi par, ga zamenjati z novim simbolom in enako storiti še pri preostalih. Za lažji dostop do prvega para sem uporabil kazalce, in sicer sem v strukturo, ki opisuje par, dodal tudi kazalec na prvi obstoječi par. Za nadaljne pomike pa skrbi dvojni seznam parov. Ko v samem seznamu simbolov ne obstaja več para, ki bi se pojavil več kot enkrat, potem se postopek zamenjave zaključi.

Pri dodajanju in zamenjavi simbola pa je potrebno paziti še na situacijo, ko se v zaporedju simbolov pojavijo najmanj trije enaki, saj v takšnem primeru lahko nastopi prekrivanje (overlap) (slika 8). To je v sami definiciji metode prepovedano, zato je potrebno poskrbeti, da do tega ne pride.

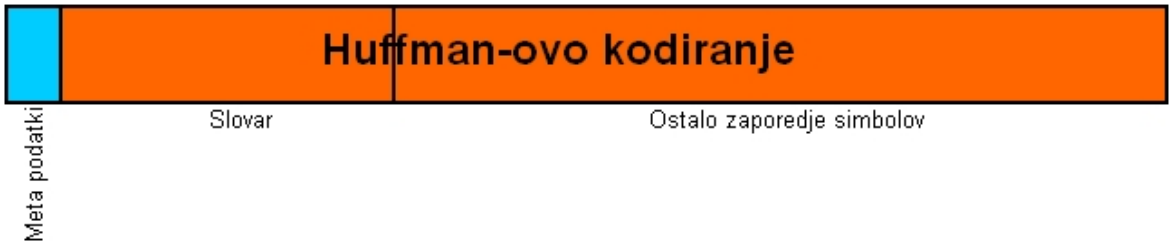
Shranjevanje izhodnih podatkov sem implementiral tako, da sem jih razdelil v tri kategorije (slika 18):

1) meta podatki, kjer so zapisani podatki o samih bitnih dolžinah, številu končnih simbolov, številu parov v slovarju;

2) preostali simboli v dvojnem seznamu, kjer so zapisani simboli, ki so ostali v seznamu po samem procesu zamenjave;

3) slovar, kjer so zapisani vsi zamenjani pari (uporabljeni).

Kategorijo meta podatkov sem zapisal izključno samo z byte-i, da jih pri razširjanju lahko brez težav preberem, kategorijo preostalih simbolov in slovarja pa sem zapisal v začasen pomnilnik (THuffStreamWriter), upoštevajoč bitne dolžine, ter jih na koncu zakodiral s pomočjo metode Huffman, rezultat pa uporabil kot izhodni podatek.



Slika 18: Prikaz deleža izhodnih podatkov po kategorijah

Razširjanje podatkov je pri algoritmu Re-Pair zelo preprosto, zato je tudi hitro. Podatke najprej s pomočjo TLZWStreamReader-ja preberem iz datoteke, jih glede na zgoraj opisano stiskanje najprej odkodiram z metodo Huffman, iz rezultata naložim preostale simbole v dvojni seznam in slovar v tabelo, kjer je zaporedna številka para, njegova koda. Pri tem upoštevam vse bitne dolžine, ki jih najprej preberem, ter se lotim razširjanja podatkov, na način, da se »sprehajam« po seznamu in nadomeščam kode s pari simbolov, ki jih imam v slovarju. To počnem toliko časa, dokler v seznamu ni več nobene kode, ki bi jo lahko nadomestil s parom simbolov. Med samo operacijo zamenjav sproti preverjam začetni del seznama in zaporedja simbolov, ki ne predstavljajo kod, s pomočjo TLZWStreamWriter-ja zapišem v datoteko. S tem zmanjšam uporabo samega pomnilnika.

3.5.3. Bisekcija

Metoda stiskanja s pomočjo bisekcije je s strani implementacije ena manj zahtevnih, če izvzamemo uporabo rekurzije, ki je najlažja rešitev za implementacijo bisekcije. Ker čas ni bila moja prioriteta, tega nisem storil. Algoritem s pomočjo bisekcije temelji na pozicijah simbolov, zato sem moral definirati posebno strukturo, ki je shranjevala skrajno levo in skrajno desno pozicijo.

Branje vhodnih podatkov pri tej metodi potrebujemo samo na način, da pridobimo vrednost simbola na točno določeni poziciji v datoteki, za kar sem ponovno uporabil TLZWStreamReader.

Sam proces bisekcije po celotni dolžini podatkov sem opravil z rekurzivnim klicem (slika 19), ki obdela točno določeno območje podatkov. Ta proces najprej razdeli območje na dva dela, in sicer tako, da izračuna največjo možno potenco x števila 2, ki je manjša od same dolžine območja podatkov, ta vrednost 2^x pa je ločnica za delitev. Območje delim toliko časa, dokler je daljše od dveh simbolov, ko to dosežem, si v slovar zapišem prvi par s preslikavo (novim simbolom). Ker je moja rešitev rekurzivna, na koncu zapišem par, ki je korenski člen v strukturi drevesa, ima pa najmanjšo vrednost novonastalega simbola, kar omogoča lažjo definicijo začetnega para pri razširjevanju.

```
if (Range > 2) then begin
  MaxPower := CalcHighLowerLog2OfRange(Range);
  LeftRange := Trunc(Power(2, MaxPower));
  RightRange := Range - LeftRange;

  LeftPair := CompressPart(AProgram, ALeftIndex, ALeftIndex + LeftRange - 1, AGeneration);
  RightPair := CompressPart(AProgram, ARightIndex - RightRange + 1, ARightIndex, AGeneration);

  Result := GetPairFromHashArray(AProgram, LeftPair.Generation, RightPair.Generation);
  if not Assigned(Result) then begin
    Result := InitHashNode(LeftPair.Generation, RightPair.Generation, nil);
    Result.Generation := CurrentGeneration;

    //add to hash array
    AddPairToHashArray(AProgram, Result);
  end
  else
    Dec(AGeneration);

end
```

Slika 19: Rekurzivni klici procedure "CompressPart"

Na koncu je potrebno podatke zapisati na izhod, kar sem ponovno storil z delitvijo podatkov na kategorije:

1) meta podatki, ki imajo informacije o samih bitnih dolžinah in o količini parov,

2) slovar, ki vsebuje vse uporabljane preslikave parov.

Zaradi zmanjševanja porabe prostora sem za slovar, ki si ga lahko predstavljamo kot drevo, uporabil premi vrstni red zapisa (oče, levi sin, desni sin). Pri obhodu drevesa sem upošteval že obiskane člene, kar je vplivalo na končno dolžino zapisa. Za člen, ki ga še nisem obiskal, sem zapisal stanje 0, za že obiskan člen pa stanje 1 in takoj za njim zaporedno številko člena. Za vsak list v drevesu sem zapisal stanje 2, za njim pa kodo simbola. Pri tem sem stanja vedno zapisal z dolžino dveh bitov, zaporedno število člena pa z izračunano dolžino, glede na največje obstoječe število. Tako sem pri ponavljanju členov lahko privarčeval pri dolžini zapisa, še vedno pa je iz njega moč prebrati vhodno drevo. Nato sem zapis s Huffman-ovo metodo še dodatno skrajšal.

Razširajanje takih podatkov ne predstavlja posebnih težav. Potrebno je poznati, katera preslikava para je korenska (začetna), in slovar, katerega sem implementiral v obliki tabele, saj mi ta z zaporednimi števili omogoča najhitrejši dostop do preslikav. Začetni simbol ima v moji razvrstitvi najmanjšo zaporedno število. Tako kot je stiskanje potekalo z rekurzijo (slika 20), je tudi razširjanje, saj je potrebno nadomestiti vse simbole, ki so novonastali in so del svojih korenskih preslikav (tabela 4). V datoteko tako zapisujem vse simbole, ki so končni (ne obstaja preslikava), rekurzivni klici pa zagotavljajo pravilen vrstni red zapisovanja.

```
procedure UnpackPair(var AProgram: TProgramSettings; APairNode: pHashNode);
begin
  if Assigned(APairNode) then begin

    if (APairNode.Left < MaxInt) then begin
      if (APairNode.Left > 255) then begin
        UnpackPair(APRogram, APRogram.DictionaryArray[APairNode.Left - 256]);
      end
      else begin
        AProgram.StreamWriter.WriteDecimal(APairNode.Left, 8);
      end;
    end;

    if (APairNode.Right < MaxInt) then begin
      if (APairNode.Right > 255) then begin
        UnpackPair(APRogram, APRogram.DictionaryArray[APairNode.Right - 256]);
      end
      else begin
        AProgram.StreamWriter.WriteDecimal(APairNode.Right, 8);
      end;
    end;

  end;
end;
```

Slika 20: Rekurzivno razširjanje pri metodi s pomočjo bisekcije

4. Primerjava algoritmov

Kot je znano, je moj cilj primerjava posameznih metod na izbranih formatih datotek. Izbral sem naslednje datoteke, nad katerim sem opravil meritve:

1) grafična datoteka (BMP),
2) tekstovna datoteka (TXT),
3) datoteka z oblikovanim besedilom (PostScript),
4) datoteka z Javino programsko kodo.

Nad vsemi datotekami sem izvedel različne metode in si statistične rezultate zabeležil, ter kasneje iz njih razbral ugotovitve.

V spodnjih tabelah se v stolpcih z imenom »Stiskalno razmerje« nahajajo odstotki, ki ponazarjajo, kolikšen prostor zaseda novonastala (stisnjena) datoteka. Pri metodi LZW obstaja več rezultatov, saj upoštevam dodatno možnost omejevanja slovarja, in sicer z možnostmi brez omejitve, in količinsko od $2^9 = 512$ do $2^{20} = 1048576$ s koraki povečevanja potence za 1.

Kar se tiče opazovanja časov, so v moji nalogi zelo odvisni od same implementacije algoritmov in modulov, saj ne vsebujejo vseh možnih optimizacij.

4.1. Grafična datoteka BMP

Za grafično datoteko sem vzel preprosto sliko formata BMP, ki prikazuje par vzporednih pasov, različnih barv, kjer so zagotovljeni ponavljajoči se vzorci. Grafični format BMP sem vzel zato, ker je eden redkih, ki ni stisnjen, kot je to npr. JPG. Omenil sem že, da je stisnjene podatke nesmiselno ponovno stiskati, saj je malo verjetno, da še vedno vsebujejo relevantno količino redundance, ki je potrebna za stiskanje.

Grafična datoteka BMP (183KB)				
Metoda		Stiskanje		Razširjanje
	Maks. velikost slovarja	Stiskalno razmerje	Čas (s)	Čas (s)
LZW	Neomejen	2,1%	3,63	2,87
	512	7,2%	3,33	3,06
	1024	3,4%	3,41	2,90
	2048	2,5%	3,68	2,86
	4096	2,1%	3,53	2,94
	8192	2,1%	3,66	2,91
Re-Pair		0,9%	4,39	3,45
Bisekcija		0,5%	2,63	3,28

Tabela 5: Rezultati meritev na grafični datoteki BMP

Kot je mogoče opaziti v tabeli, so vse motode opravile stiskanje, ki zaseda manj kot 10% prvotnega prostora. Metodi Re-Pair in s pomočjo bisekcije sta dosegli odlične rezultate, saj je njuna novonastala datoteka velika manj kot 1% začetnega prostora.

Metoda s pomočjo bisekcije je vodilna tudi glede hitrosti, čeprav ima najboljše stiskalno razmerje. Pri primerjanju časov in stiskalnega razmerja bi lahko pričakovali premo soodvisnost, pa temu ni tako.

Mislim, da ima stiskanje s pomočjo bisekcije prednosti pri samem zapisu v datoteko, kjer lahko privarčuje na prostoru. To dokazuje, da sama definicija metode ne zadostuje za uspeh, ampak potrebuje tudi učinkovit zapis rezultatov (slovar).

4.2. Tekstovna datoteka TXT

Za vsebino tekstovne datoteke sem vzel uprizoritev tragedije Hamlet, avtorja Williama Shakespeara. Sicer vsebuje nekaj vzorcev nizov, ki se ponavljajo, to so predvsem imena vlog, drugače pa je besedilo v osnovi zelo raznoliko. Raznolikost besedila naj bi še posebej vplivala na rezultate.

Tekstovna datoteka TXT (198KB)				
Metoda		Stiskanje		Razširjanje
	Maks. velikost slovarja	Stiskalno razmerje	Čas (s)	Čas (s)
LZW	Neomejen	43,1%	4,18	2,93
	512	77,0%	4,98	4,85
	1024	64,8%	4,39	4,06
	2048	58,1%	4,20	3,71
	4096	53,5%	4,02	3,47
	8192	49,9%	4,07	3,23
	16384	47,0%	4,07	3,10
	32768	44,8%	4,10	3,01
	65536	43,1%	4,01	2,91
Re-Pair		79,7%	5,14	4,27
Bisekcija		70,2%	3,05	3,90

Tabela 6: Rezultati meritev na tekstovni datoteki TXT

V zgornji tabeli lahko opazimo, da sem pri tej vrsti datoteke dosegel več kot polovično stiskanje. To omogoča metoda LZW, ki se najbolje obnaša brez omejevanja slovarja, saj pri tej opciji zasede le malo več kot 43% izvornega prostora.

Ostali dve metodi se po razmerju ne moreta kosati z LZW, saj metoda s pomočjo bisekcije dosega le 30% stiskanje, medtem ko metoda Re-Pair doseže le 20% stiskanje.

Časovno so si metode zelo blizu, lahko pa vidimo, da je razširjanje tako v teoriji, kot tudi v praksi skoraj vedno hitrejše od stiskanja, le metoda z bisekcijo pri tem izstopa. Pri tej metodi še obstajajo možnosti optimizacije, ki bi to dosegle.

4.3. Datoteka z oblikovanim besedilom (Post Script)

Besedilna datoteka s formatom Postscript vsebuje besedilo, ki je oblikovano. Oblika je opisana z meta podatki. V takšni datoteki se običajno nahajajo zelo raznoliki vzorci podatkov, saj je besedilo razdeljeno na več delov, pri čemer ima vsak del predpisano obliko.

Datoteka z oblikovanih besedilom PS (185KB)				
Metoda		Stiskanje		Razširjanje
	Maks. velikost slovarja	Stiskalno razmerje	Čas (s)	Čas (s)
LZW	Neomejen	79,7%	4,91	3,98
	512	94,3%	5,09	5,04
	1024	89,7%	4,82	4,66
	2048	88,5%	4,71	4,43
	4096	87,6%	4,68	4,27
	8192	85,5%	4,69	4,18
	16384	82,7%	4,65	4,08
	32768	80,9%	4,64	3,94
	65536	79,8%	4,81	3,90
	131072	79,7%	4,90	3,92
Re-Pair		122,7%	6,31	5,65
Bisekcija		113,4%	4,23	5,28

Tabela 7: Rezultati meritev na datoteki z oblikovanim besedilom PS

V razpredelnici je videti stiskanje samo v vrsticah z metodo LZW, kar pomeni, da se le-ta obnaša najbolje. Ponovno se vede najbolje, ko slovar ni omejen, saj takrat zaseda okoli 80% prostora.

Preseneča me to, da ima metoda Re-Pair v tem primeru slabše rezultate kot metoda s pomočjo bisekcije, obe pa ne dosegata stiskanja, saj sta njuni končni datoteki večji kot izvorna.

Čas, ki ga potrebuje metoda LZW za svoj rezultat je okoli 5 sekund. To sicer za takšno količino podatkov (185KB) ni malo, vendar obstaja še mnogo optimizacij algoritma, ki bi metodo definitivno pospešile.

4.4. Datoteka z Javino programsko kodo

Datoteka z Javino programsko kodo, ki je del odprto kodne aplikacije, predstavlja vzorec podatkov, kjer je verjetnost pogostejših nizov večja. Vsak programski jezik ima točno določena pravila, ki predpisujejo način uporabe programskih stavkov. Tem pravilom pravimo gramatike, s katerimi se preverja pravilnost kode. Zato je ta vzorec podatkov za nas še posebej zanimiv.

Datoteka z Javino programsko kodo (107KB)				
Metoda		Stiskanje		Razširjanje
	Maks. velikost slovarja	Stiskalno razmerje	Čas (s)	Čas (s)
LZW	Neomejen	33,4%	1,99	1,44
	512	64,4%	2,39	2,39
	1024	51,1%	2,11	1,97
	2048	44,3%	2,07	1,69
	4096	40,3%	2,05	1,63
	8192	37,2%	2,05	1,51
	16384	35,1%	2,03	1,34
	32768	33,4%	2,01	1,32
Re-Pair		74,7%	2,88	2,43
Bisekcija		56,6%	1,67	2,13

Tabela 8: Rezultati merive datoteke z Javino programsko kodo

Ponovno se je za najboljšo metodo izkazala metoda LZW, ki je s 33% stiskalnim razmerjem dosegla maksimum, sledi ji metoda s pomočjo bisekcije in nato še metoda Re-Pair, ki je prihranila le malo več kot 25% prostora.

Tudi pri hitrosti so vse metode izredno hitre, kar poudarja pomen vzorca podatkov v izvorni datoteki.

Kot zanimivost lahko opazimo, kako z omejevanjem slovarja pri metodi LZW stiskalno razmerje pada. To je odvisno tudi od najvišjega razmerja in je teoretično utemeljeno, saj omejevanje slovarja pri metodi LZW pomeni v bistvu stiskanje podatkov po blokih, kar pa bo uspešno le v redkih primerih. Pri dovolj veliki omejitvi slovarja ponovno doseže rezultat, dosežen brez omejitve, to pomeni, da v tem primeru omejitve ne preseže in tako nikoli ne izprazni slovarja. S tem lahko približno ocenimo maksimalno število nizov v slovarju.

5. Zaključek

Problem neomejenega shranjevanja podatkov je že od nekdaj ena temeljnih težav računalništva, tudi sodobnega. Obstaja ogromno definiranih metod za stiskanje podatkov, njihovih različic in verzij. V današnjem svetu računalništva pa lahko rečem, da je še vedno najbolj uveljavljena prav metoda LZW, kot osnova. V nalogi smo sicer obravnavali metode s pomočjo kontekstno neodvisnih gramatik, ki so se izkazale za uporabne pri posebnih vzorcih podatkov.

Realizacija izbranih metod večinoma ni bila zahtevna, sem pa ugotovil zanimivost, da je metoda LZW, ki je najbolj učinkovita, tudi najpreprostejša. To potrjuje dejstvo, da kompleksnost algoritma ni vedno ključ do uspeha.

Po svetu nastajajo nove metode, ki so bodisi samosvoje ali pa kombinacija že obstoječih. Nekatere se ne uspejo niti uveljaviti, ko so na obzorju že nove, zato menim, da bo nastalo še kar nekaj takšnih ali drugačnih metod za stiskanje, preden bodo izumili takšno, ki bo zadovoljila potrebe po stiskanju množičnih podatkov v računalništvu.

Celotna izvorna koda programa, ki je bila razvita z namenom testiranja algoritmov, se nahaja na priloženem CD-ju.

Kazalo slik

Kazalo tabel

Viri in literatura

[1] Mengyi Pu Ida, »Fundamental Data Compression«, 2006

[2] Salomon David, »Data Compression, The Complete Reference«, »Third Edition«, 2004

[3] (2011) Larsson N. Jesper, Moffat Alistair, »Offline Dictionary-Based Compression«. Dostopno na:
http://www.larsson.dogma.net/dcc99.pdf

[4] (2011) Lehman Eric, »Approximation Algorithms for Grammer-Based Data Compression«. Dostopno na:
http://compression.graphicon.ru/download/articles/grammar/lehman_phd_2002_approximation_algorithms.pdf

[5] (2011) Obstojče metode za stiskanje podatkov. Dostopno na:
http://en.wikipedia.org/wiki/Data_compression

[6] (2011) Metoda LZW. Dostopno na:
http://marknelson.us/1989/10/01/lzw-data-compression/

[7] (2011) Kontekstno neodvisne gramatike. Dostopno na:
http://en.wikipedia.org/wiki/Context-free_grammar

[8] (2011) Huffman-ova metoda. Dostopno na:
http://www.explainth.at/en/delphi/huff.shtml

[9] (2011) Začetki stiskanja. Dostopno na:
http://www.wolframscience.com/reference/notes/1069b