# UVOD

Glavni namen in cilji diplomskega dela Kreiranje stisnjenih (zip) datotek je razumevanje delovanja t. i. stiskalnih algoritmov in kreiranje enostavnega programa za stiskanje datotek s pripadajočimi funkcionalnostmi, pregled zgodovine stiskalnih algoritmov in kako so ti nastali in možne dodelave oz. predelave stiskalnih algoritmov.

Metode izdelave naloge bodo najprej zgodovinski pregled in načini stiskanja datotek, programiranje enostavnega programa v programskem jeziku C# s pomočjo programa Visual Studio, z vsemi funkcionalnostmi ki so: stiskanje datotek, odpiranje stisnjene datoteke in ekstrahiranje (razširjanje) vsebine datoteke, enkripcija datotek s pomočjo gesla in kreiranje vizualnega uporabniškega vmesnika (GUI).

Izdelal bom tudi analizo in testiral hitrosti stiskanja datotek s pomočjo testnih orodij v okolju Visual Studio. Najbolj zanimive izseke iz kode bom tudi komentiral v nalogi, v priloge bom dodal celotno skripto programa s komentarji.

# TEORETIČNI DEL

V teoretičnem delu naloge, se osredotočam na t.i. Lossles stiskalne algoritme ali algoritme z najmanjšo možno izgubo podatkov. Najprej se posvetim zgodovini stiskalnih algoritmov. V drugem delu poglavja, se nahajajo opisi najbolj uporabljenih algoritmov, ki so prikazani na praktičnih primerih in kako se uporabljajo. Poglavje zaključim s primeri enkripcije in implementacije v sodobni programski opremi.

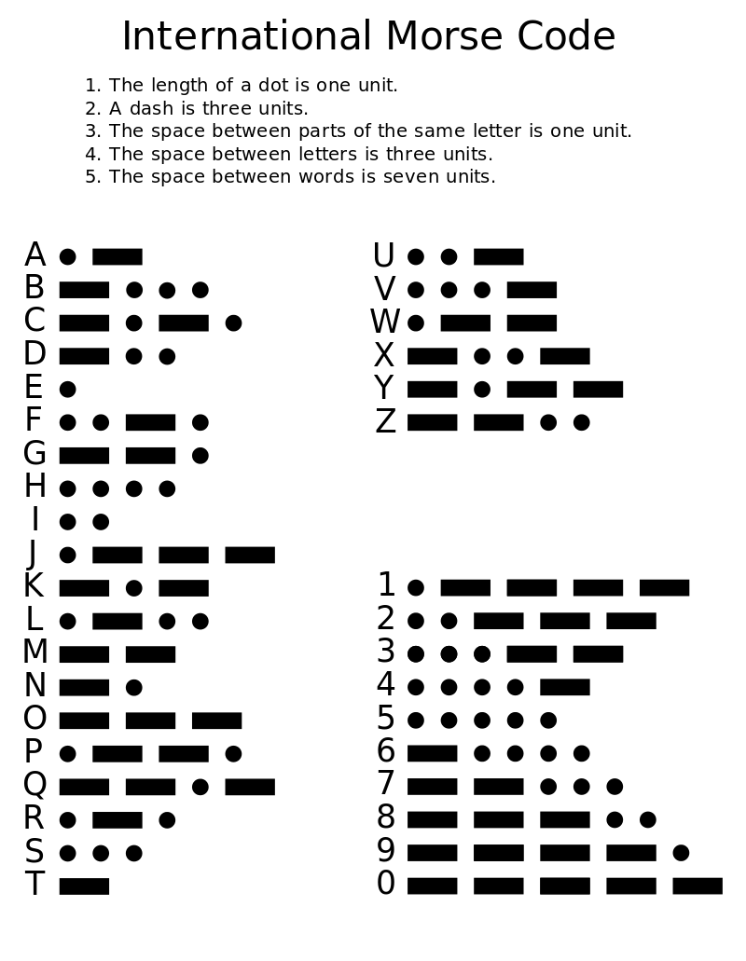
## ZGODOVINA

Kot prvi stiskalni algoritem se navaja že Morsejeva koda ali Morsejeva abeceda. Razvila sta jo izumitelj telegrafa Samuel Morse in Alfred Vail, leta 1835. Leta 1847 jo je Friedrich Clemens Gerke izpopolnil v obliko, kot jo poznamo danes.

Internacionalna Morsejeva abeceda je sestavljena iz šestindvajsetih angleških črk od A do Z, nekaj ne-angleških črk, postopkovnih znakov (ang. prosigns) in arabskih številk. Vsebuje samo velike črke in uporablja kombinacije črt in pik. Črte v znakih so trikrat daljše od pik, med vsako piko oz. črto je presledek v dolžini pike, med vsako črko v besedi je presledek v dolžini treh pik in razmak med besedami je v dolžini sedem pik.

Morsejeva abeceda je bila aktivno v uporabi v mornarici in oboroženih silah do leta 1999. Še vedno jo uporabljajo taborniki in skavti kot način enostavne komunikacije.

Slika 1: Internacionalna Morsejeva abeceda



Vir: <https://en.wikipedia.org/wiki/Morse_code#/media/File:International_Morse_Code.svg>

Velik preobrat se je začel dogajati med leti 1940 in 1948 ko je Claude E. Shannon s svojo disertacijo »A Mathematical Theory of Communication« objavljeno v Bell System Technical Journal julija in oktobra 1948, postavil danes znano Teorijo informacij. Iz teorije informacij, danes poznamo statistično sklepanje, kriptografijo, komunikacijske mreže, termodinamiko, kvantno računanje itd.

Pozneje leta 1949 sta Shannon v Bell Labs in R. M. Fano na MIT-ju skoraj istočasno razvila Shannon-Fano metodo za kodiranje simbolov, ki je bila odvisna od vedenja verjetnosti pojavitve posameznega znaka v nekem sporočilu. Po podani verjetnosti, je bila skonstruirana tabela znakov, ki je imela podane značilnosti:

* Različni znaki imajo različno število bitov.
* Znaki simbolov majhne verjetnosti prikaza imajo večje število bitov in znaki simbolov večje verjetnosti prikaza imajo manjše število bitov.
* Čeprav so simboli različnih dolžin bitov, so lahko edinstveno dekodirani.

Dve leti pozneje, leta 1951 je David Huffman obiskoval študij Informacijske teorije na MIT-ju in obiskoval predavanja Roberta Fana. Fano je za opravljanje predmeta, dal skupini študentov možnost pisanja izpita ali izdelave seminarja. Huffman se je odločil za izdelavo seminarja na temo iskanja najučinkovitejše metode binarnega kodiranja. Po nekajmesečnem raziskovanju in brez rešitev, je bil Huffman pripravljen zaključiti z delom in se pripraviti na končni izpit. Ravno takrat je doživel razodetje in ugotovil zelo podobno a učinkovitejšo tehniko Shannon-Fano metode. Glavna razlika je v branju drevesa. Pri Shannon-Fano se drevo bere od zgoraj dol, pri Huffmanovem kodiranju pa od spodaj gor. Huffmanov algoritem je bila prva implementacija algoritma, ki je vsebovala minimalno redundanco (izgubo) podatkov. Hitro je postalo jasno, da ima Huffmanov algoritem najboljše možne rezultate stiskanja.

Prve izvedbe Shannon-Fano in Huffman kodiranja so bile izvedene z uporabo strojne in »hardcoded«[[1]](#footnote-1) kode. Šele v 1970-ih letih, ko se je začel razvijati in uporabljati internet in spletna shramba programske opreme, so bile Huffmanove kode dinamično ustvarjene na podlagi vhodnih podatkov.

Kasneje leta 1977 sta Abraham Lempel in Jacob Ziv razvila prelomni algoritem LZ77 imenovan tudi LZ1. To je bil prvi algoritem, ki je za kodiranje uporabil slovar, namesto drevesa. Natančneje, LZ77 je uporabljal dinamični slovar, s pomočjo protokola »Sliding window protocol«[[2]](#footnote-2), ki se v nekaterih implementacijah uporablja še danes. Več o tem v naslednjem poglavju.

Naslednje leto, leta 1978 sta Lempel in Ziv izboljšala svoj algoritem in objavila algoritem LZ78, imenovan tudi LZ2, ki pa je za razliko od LZ77 najprej razčlenil podatke in generiral statični slovar, namesto dinamičnega.

Oba algoritma LZ77 in LZ78 sta hitro pridobila na popularnosti, kar je vodilo do mnogih variant (glej sliko). Večina teh algoritmov je zamrla, le peščica se jih je ohranila do danes (LZMA, DEFLATE in LZX).

Leta 1984 je Terry Welch nadgradil LZ78 algoritem in objavil algoritem LZW, ki je izboljšal izvajanje algoritma LZ78 s tem, da je pohitril procese izvajanja na strojnem nivoju. Ta algoritem se je uporabljal in se še uporablja pri stiskanju slik v format GIF.

Večina standardnih algoritmov uporabljenih danes, temelji na algoritmu LZ77, ne zaradi tehnične superiornosti ampak zaradi tega, ker so algoritmi postali patentno obremenjeni, ko je korporacija Sperry leta 1984 patentirala LZW algoritem in začela tožiti prodajalce programske opreme, skrbnike strežnikov in celo končne uporabnike za uporabo GIF formata slik brez licence. Takrat je program za stiskanje v UNIX sistemih, ki je uporabljal modifikacijo LZW algoritma imenovanega LZC, bil kmalu ukinjen zaradi kršenja pravic patenta. Tudi drugi razvijalci na sistemu UNIX so začeli odstopati od uporabe LZW algoritma zaradi pojavitve odprtokodnih algoritmov.

Korporacije in druge večje organizacije, so uporabljale stiskanje podatkov od objave Lempel-Ziv algoritmov naprej, saj so imele vedno večje potrebe po shranjevanju podatkov. Ampak stiskanje podatkov ni dobilo večje prepoznavnosti in možnosti uporabe do pojava interneta v poznih 1980-ih, ko se je potreba po stiskanju podatkov eksponentno povečala. Pasovna širina pri internetu je bila omejena in draga, zato je stiskanje podatkov pomagalo ublažiti ozka grla pri prenosu podatkov. Stiskanje podatkov je postalo še kako zaželjeno, ko je internet dosegel vsakdanje uporabnike, ki so pošiljali, delili datoteke, slike ipd. Za izpolnitev teh potreb je bilo razvitih več novih arhivskih formatov kot so ZIP, GIF in PNG.

Thom Henderson je prvi komercialno uspešen arhivski format imenovan ARC, izdal leta 1986 prek svojega podjetja System Enhancement Associates (SEA). ARC je bil eden prvih programov in formatov, ki je lahko združeval in stiskal več datotek hkrati in bil odprtokoden. ARC je bil hibrid Huffmanovega in LZW algoritma.

Pozneje je Phil Katz s podjetjem PKWARE Inc. opazil priljubljenost ARC-a in prišel na idejo, da bo algoritem izboljšal s pisanjem rutin v strojnem jeziku. Leta 1987 je izdal svoj program PKARC kot shareware[[3]](#footnote-3). SEA je PKWARE tožila zaradi kršenja blagovne znamke in avtorskih pravic. Sodišče je za primerjavo obeh programov imenovalo neodvisnega strokovnjaka za programsko opremo, Johna Navasa, ki je ugotovil, da je PKARC izpeljan del ARC-a, pri katerem so komentarji v obeh programih pogosto enaki, vključno s pravopisnimi napakami. Tožnik in tožene stranke so 2. avgusta 1988 napovedali poravnavo tožbe, ki je vključevala Zaupno pogodbo o navzkrižni licenci, po kateri je SEA licencirala PKWARE za vse programe, združljive z ARC, ki jih je PKWARE objavil v obdobju, ki se začne s prvo izdajo PKARC konec leta 1985 do 31. julija 1988 v zameno za nerazkrito plačilo. PKWARE je v sporazumu plačal SEA za pridobitev licence, ki je dovoljevala distribucijo programov, združljivih s PKWARE, do 31. januarja 1989, nato pa PKWARE ne bo licenciral, objavljal ali distribuiral programov ali pripomočkov, ki so združljivi z ARC. V zameno je PKWARE licenciral SEA za uporabo svoje izvorne kode za programe, združljive s PKWARE ARC. Podjetje PKWARE se je tudi strinjalo, da bo prenehalo uporabljati SEA zaščitni znak "ARC" in da bo spremenilo imena ali znamke, ki se uporabljajo v programih PKWARE, na nejasne oznake. Preostale podrobnosti sporazuma so bile zapečatene. Toženci pri doseganju poravnave niso priznali nobene krivde. S sklepom sodišča je bilo razvidno, da je bilo naloženo plačilo odškodnine tožniku zaradi dejanj toženca, ki kršijo tožnikove avtorske pravice, blagovno znamko in dejanja nepoštene trgovinske prakse in nepoštene konkurence.

Leta 1989 je Phil Katz zaradi tožbe predelal ARC algoritem in izdelal danes najbolj uporabljeni format ZIP. Format je na začetku uporabljal LZW algoritem ampak zaradi uporabe patenta, je Katz pozneje z izdajo programa PKZIP 2.0 LZW zamenjal za DEFLATE. Ta različica arhivske datoteke je v isti obliki prisotna še danes.

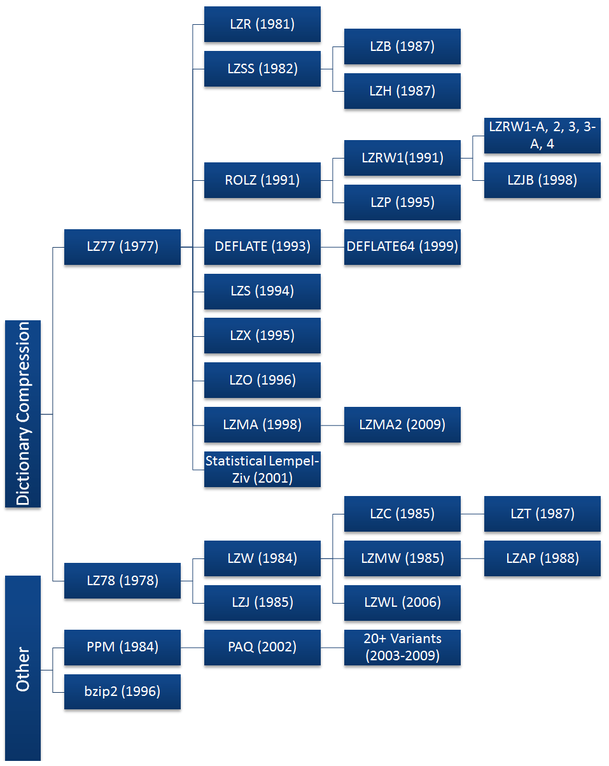
Format ZIP in drugi formati bazirani na DEFLATE algoritmu so bili najboljši do sredine 1990-ih, ko so se začeli pojavljati novi in izboljšani formati. Leta 1993 je Eugene Roschal izdal program WinRAR, ki uporablja format RAR. Zadnja verzija RAR formata uporablja kombinacijo LZSS in PPM algoritma, za katera ni znano, kako sta se razvila. RAR je postal standardni format za pošiljanje datotek prek spleta, predvsem za distribucijo piratske programske opreme.

Odprtokodna implementacija algoritma Burrows-Wheeler transformacije imenovana BZIP2 predstavljena leta 1996 je hitro dobila na popularnosti na UNIX platformi in izpodbila DEFLATE baziran GZIP format. Še en odprtokodni program za stiskanje podatkov, ki se je pojavil leta 1999 je bil 7-Zip ali format 7z. 7-Zip bi lahko bil prvi format, ki je izzval prevlado ZIP in RAR formatov, zaradi na splošno visokega kompresijskega razmerja ter modularnosti. Ta oblika ni omejena na uporabo enega algoritma ampak lahko izbira med BZIP2, LZMA in PPMd algoritmi.

Zadnja najnovejša in najbolj sposobna različica stiskalnega algoritma je leta 2002 izdan algoritem imenovan PAQ, ki za delovanje uporablja statistične modele kontekstnega mešanja. Izdal ga je Matt Mahoney.

Prihodnost stiskalnih algoritmov je mogoče dokaj zanesljivo napovedati na podlagi trenutnih trendov, z uporabo PAQ algoritma in njegovih prihodnjih različic, ki dosega najvišja razmerja stiskanja. Algoritem Prediction by Partial Matching PPM oz. PPMd iz katerega izhaja PAQ, tudi lahko v prihodnosti vidi veliko izboljšav. Trenutno se je algoritem verige Lempel-Ziv-Markov oz. LZMA izkazal kot odličen kompromis med hitrostjo in visokim razmerjem stiskanja in je implementiran v večino datotek.

Slika 2: Diagram razvoja stiskalnih algoritmov



Vir: <https://ethw.org/History_of_Lossless_Data_Compression_Algorithms>, dostopno 1. 8. 2019

### Dizajn algoritmov in primeri kodiranja

Kot sem razdelal v prejšnjem poglavju, so se stiskalni algoritmi spreminjali skozi čas in s tem tudi njihova zgradba in implementacija. V tem podpoglavju si bomo pogledali kako so zgrajeni najbolj uporabljeni algoritmi. Za ta namen sem izbral algoritme Shannon-Fano, Huffmanov algoritem, LZMA in DEFLATE, ki so še danes največkrat uporabljeni.

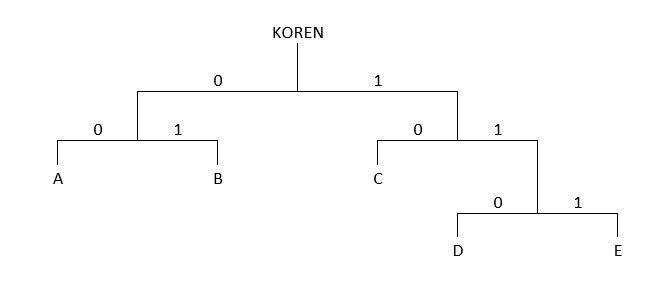
#### Shannon-Fano metoda

Shannon-Fano metoda za kodiranje simbolov, je odvisna od vedenja verjetnosti pojavitve posameznega znaka v nekem sporočilu. Po podani verjetnosti, je skonstruirana tabela znakov, ki ima podane značilnosti:

* Različni znaki imajo različno število bitov.
* Znaki simbolov majhne verjetnosti prikaza imajo večje število bitov in znaki simbolov večje verjetnosti prikaza imajo manjše število bitov.
* Čeprav so simboli različnih dolžin bitov, so lahko edinstveno dekodirani.

V našem primeru imamo podano enostavno abecedo sestavljeno iz petih črk. Iz teh definicij oblikujemo drevesno strukturo prikazano na spodnji sliki.

Slika 3: Enostavno Shannon-Fano drevo



Vir: Nelson, Mark, Gailly, Jean-loup (1996). The Data Compression Book, 2nd edition, M&T Books, str. 28. (Vir je preveden za potrebe naloge)

Iz drevesne strukture lahko razberemo kode posameznih črk, kot je prikazano v tabeli.

Tabela 1: Tabela kod abecede razbranih iz Shannon-Fano drevesa

|  |  |
| --- | --- |
| **Simbol** | **Koda** |
| A | 00 |
| B | 01 |
| C | 10 |
| D | 110 |
| E | 111 |

Vir: Nelson, Mark, Gailly, Jean-loup (1996). The Data Compression Book, 2nd edition, M&T Books, str. 28. (Vir je preveden za potrebe naloge)

Po tem, ko imamo določene kode posameznih znakov, moramo vedeti, kolikokrat se posamezen znak pojavi v nekem besedilu. Algoritem je potem preprost:

1. Za podan seznam simbolov razvij ustrezen seznam verjetnosti ali števila frekvenc (pojavitev) tako, da bo znana relativna frekvenca vsakega simbola.
2. Seznam sortiraj glede na pogostost pojavitev tako, da bodo simboli z največ pojavitvami na vrhu in simboli z najmanj pojavitvami na dnu.
3. Seznam razdeli na dva dela pri čemer je seštevek pojavitev zgornje polovice čim bližje seštevku pojavitev spodnje.
4. Zgornji polovici seznama je dodeljen binarni 0, spodnji pa 1. To pomeni, da se bodo kode za simbole v prvi polovici začele z 0, v drugi pa z 1.
5. Rekurzivno uporabi koraka 3 in 4 na vsaki od obeh polovic, pri čemer razdeli skupine in dodajaj bite, dokler vsak simbol ne postane ustrezen kodni list na drevesu.

Za primer vzemimo spodnjo tabelo frekvenc simbolov.

Tabela 2: Število frekvenc simbolov

|  |  |
| --- | --- |
| **Simbol** | **Frekvenca** |
| A | 15 |
| B | 7 |
| C | 6 |
| D | 6 |
| E | 5 |

Vir: Nelson, Mark, Gailly, Jean-loup (1996). The Data Compression Book, 2nd edition, M&T Books, str. 29. (Vir je preveden za potrebe naloge)

Če sledimo navodilom algoritma, vidimo, da lahko razdelimo simbole tako, da dobimo simbola A in B v eno skupino in C, D in E v drugo, kar pomeni, da imamo seštevek A in B, 22 in seštevek C, D, E, 17. To pomeni, da bo imela prva skupina začetno števko 0 in druga 1.

Ko ponovimo koraka 3 in 4, dodajamo števke simbolom. Tako dobimo naslednjo tabelo.

Tabela 3: Simboli in njihove pripadajoče binarne kode po stiskanju

|  |  |
| --- | --- |
| **Simbol** | **Koda** |
| A | 00 |
| B | 01 |
| C | 10 |
| D | 110 |
| E | 111 |

Kot vidimo iz tabele, smo pravilno sledili algoritmu in tako spoznali, da imajo simboli z večjo frekvenco, manjše število bitov in obratno. Od tu lahko izračunamo povprečno velikost simbola v bitih s formulo:

Simboli A, B in C imajo 2 bita v kodi, zato vse tri seštejemo in pomnožimo z 2 plus simbola D in E imata 3 bite v kodi, zato ju zmnožimo s 3. Celoten izračun, nato delimo s številom vseh znakov, ki jih je 39 in dobimo rezultat 2,28, kar je povprečje bita na simbol.

Tu lahko ugotovimo, da če zapišemo znake v ASCII kodi, ki ima 8 bitov na simbol, ugotovimo, da lahko uporabimo 8 krat 39, kar znaša 312 bitov. Po stiskanju podatkov ugotovimo, če pomnožimo vse simbole s povprečjem 2,28, dobimo 88,92 bitov za celotno sporočilo, kar je zelo dobra konverzija.

#### Huffmanov algoritem

Huffmanov algoritem ima zelo podoben princip Shannon-Fano metodi, saj podobno ustvarja variabilno strukturo bitov glede na znak. Simboli z večjo frekvenco dobijo manjše število bitov in obratno. Izboljšava je le v tem, da imajo Huffmanove kode dodan atribut edinstvene predpone za vsak znak, kar pomeni, da so vedno pravilno dekodirani čeprav imajo spremenljivo dolžino bitov.

Gradnja Huffmanovega drevesa se izvede s pomočjo drugačnega algoritma kot pri Shannon-Fano metodi. Shannon-Fano drevo je zgrajeno od zgoraj navzdol, medtem ko je Huffmanovo drevo zgrajeno od spodaj navzgor, začenši z listi drevesa in deluje proti korenu drevesa. Drevo je zgrajeno po naslednjih korakih:

1. Ustvari list za vsak simbol in ga dodaj v prednostno čakalno vrsto pri čemer uporabi njegovo frekvenco.
2. Medtem ko je v čakalni vrsti več kot eno vozlišče:

* Iz čakalne vrste odstrani obe vozlišči z najmanjšo verjetnostjo pojavitve ali frekvenco
* Dodaj 0 ali 1 vsaki kodi, ki je že dodeljena tem vozliščem
* Dodaj novo vozlišče v čakalno vrsto

1. Preostalo vozlišče je koren, kar pomeni, da je drevo končano.

Za primer vzemimo simbole in pripadajoče frekvence iz prejšnjega primera: A: 15, B: 7, C: 6, D: 6, E: 5. Pri Huffmanovem algoritmu poimenujemo simbole vozlišča.

Prvi prehod skozi drevo, identificira dve prosti vozlišči z najmanjšo frekvenco; D: 6 in E: 5. Ti dve vozlišči sta združeni v matično vozlišče, ki mu je dodeljena frekvenca 11. Vozlišči D in E sta nato odstranjeni iz seznama.

Po prvem prehodu vemo kakšni bodo najmanj pomembni biti v kodah za D in E. Pri naslednjem prehodu skozi seznam prostih vozlišč sta izbrani vozlišči B in C kot dve z najmanjšo težo. Nato so pritrjeni na novo nadrejeno vozlišče. Nadrejenemu vozlišču je dodeljena teža 13, B in C pa se odstranita s seznama prostih vozlišč.

Pri naslednjem prehodu sta dve vozlišči z najnižjo utežjo matični vozlišči za para B / C in D / E. Ti so povezani skupaj z novim matičnim vozliščem, ki mu je dodeljena teža 24, otroci pa se odstranijo s prostega seznama. Na tej točki smo Huffmanovim kodam za B, C, D in E dodelili po dva bita, za kodo A pa moramo še dodeliti en bit.

Končno sta na zadnjem prehodu le še dve prosti vozlišči. Starš s težo 24 je povezan z vozliščem A, da ustvarimo novega starša s težo 39. Po odstranitvi obeh otroških vozlišč s prostega seznama nam ostane samo en starš, kar pomeni, da je drevo popolno. Končni rezultat je prikazan na spodnji sliki.

Slika 4: Huffmanovo drevo

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | KOREN | |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  | 0 | 1 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  | 39 | |  | 0 | 1 |  |  |  |
|  |  |  |  | 24 | |  |  |  |
|  |  | 0 | 1 |  |  | 0 | 1 |  |
|  |  | 13 | |  |  | 11 | |  |
|  |  |  |  |  |  |  |  |  |
|  | 7 | | 6 | | 6 | | 5 | |
| 15 | | B | C | | D | | E | |
| A | |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

Vir: Nelson, Mark, Gailly, Jean-loup (1996). The Data Compression Book, 2nd edition, M&T Books, str. 33. (Vir je preveden za potrebe naloge)

Za določanje kod simbolom, se moramo sprehoditi od spodaj navzgor (od listov do korena). Biti se nam vrnejo v obratnem vrstnem redu, zato jih moramo obrniti (kar je 1, postane 0 in obratno). Spodnja tabela prikazuje kode, ki jih dobimo s tem postopkom.

Tabela 4: Kode pridobljene iz Huffmanovega drevesa

|  |  |
| --- | --- |
| **Simbol** | **Koda** |
| A | 0 |
| B | 100 |
| C | 101 |
| D | 110 |
| E | 111 |

Naredimo primerjavo z Shannon-Fano metodo kodiranja. V naslednji tabeli vidimo števila bitov, ki pripadajo posameznemu simbolu. Razvidno je, da imamo različna števila bitov. A ima pri Huffmanovem algoritmu samo en bit, napram dvema pri Shannon-Fano kodiranju, itd. V tabeli je razvidna tudi razlika v bitih po stiskanju/kodiranju.

Tabela 5: Primerjava med Shannon-Fano in Huffmanovim algoritmom

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Simbol** | **Frekvenca** | **Shannon-Fano dolžina (bit)** | **Shannon-Fano velikost (bit)** | **Huffman dolžina (bit)** | **Huffman velikost (bit)** |
| A | 15 | 2 | 30 | 1 | 15 |
| B | 7 | 2 | 14 | 3 | 21 |
| C | 6 | 2 | 12 | 3 | 18 |
| D | 6 | 3 | 18 | 3 | 18 |
| E | 5 | 3 | 15 | 3 | 15 |

Prilagoditev velikosti kode pri Huffanovem algoritmu doda 13 bitov za simbola B in C in prihrani 15 bitov pri simbolu A, kar pomeni, da generalno prihranimo 2 bita informacij napram Shannon-Fano metodi. Če izračunamo povprečno porabo bitov pri kodiranju dobimo naslednji rezultat:

Pri tem rezultatu vidimo, da s Huffmanovim algoritmom prihranimo 0,05 bita na znak.

Na splošno sta Shannon-Fano metoda in Huffmanov algoritem zelo učinkovita. Ker imata oba algoritma podobno količino procesne moči.

#### Algoritem LZMA

Algoritem LZMA (Lempel-Ziv-Markov Chain Algorithm) je mešanica algoritma LZ77 in kodiranja obsega (Range encoding). Deluje na podlagi slovarja. Prvič je bil uporabljen pri formatu .7z. LZMA algoritem odlikuje visoko kompresijsko razmerje in spremenljiva velikost slovarja (do 4GB), pri tem pa ohranja hitrost kompresije, podobno kot drugi pogosto uporabljeni algoritmi.

Pri modelu LZ77 si predstavljajte, da kopirate besedilo, znak po znak, vendar želite vzeti nekaj bližnjic. Pošljete bodisi posamezne znake, bodisi par številk (razdalja, dolžina), kar pomeni: kopiraj znak, začenši z besedo, v kopiranem besedilu, od točke, kjer je trenutno kazalec.

Kodiranje obsega je metoda kodiranja, ki deluje po principu entropije, ki jo je določil Nigel N. Martin v dokumentu iz leta 1979. Glede na tok simbolov in njihove verjetnosti, kodiranje obsega ustvari prostorno učinkovit tok bitov, ki predstavlja simbole.

Kodiranje obsega je zelo podobno aritmetičnemu kodiranju, le da se kodiranje izvaja s številkami, namesto z biti, saj je tako hitrejše pri uporabi večjih baz (npr. Bajtov) z manjšimi stroški pri učinkovitosti stiskanja.

Pri kompresiji LZMA je stisnjen tok (stream), tok bitov, kodiran s pomočjo binarnih kod za prilagodljiv binarni obseg. Tok je razdeljen na pakete, pri čemer vsak paket opisuje bodisi en bajt, bodisi zaporedje algoritma LZ77, katerega dolžina in razdalja sta implicitno ali eksplicitno kodirana. Vsak del paketa je modeliran neodvisno, zato so predvidevanja verjetnosti za vsak bit korelirana z vrednostmi tega bita (in z njim povezanih bitov iz istega polja) v prejšnjih paketih iste vrste.

#### Algoritem DEFLATE

DEFLATE je lossless algoritem, ki ga je leta 1993 razvil Phil Katz. Deluje na podlagi večih stisnjenih blokov. Vsak blok vsebuje glavo (header) ki je sestavljena iz treh bitov:

* Prvi bit: Zadnji blok v toku podatkov:
* 1 pomeni zadnji blok v toku podatkov
* 0 pomeni, po tem bloku imamo še več blokov za obdelavo
* Drugi in tretji bit: Metoda kodiranja, uporabljena za ta blok
  + 00: shranjen/surov/dobesedni odsek, dolg med 0 in 65535 bajti
  + 01: statični Huffmanov stisnjen blok z uporabo vnaprej dogovorjenega Huffmanovega drevesa
  + 10: stisnjen blok skupaj s priloženo Huffmanovo tabelo
  + 11: rezerviran bit

Metoda kodiranja (00 v drugem in tretjem bitu) se uporablja za nestisljive podatke.

Večina stisljivih podatkov bo na koncu kodirana z metodo 10, Huffmanovim kodiranjem, ki ustvari optimizirano Huffmanovo drevo, prilagojeno za vsak blok podatkov posebej. Navodila za ustvarjanje potrebnega Huffmanovega drevesa takoj sledijo glavi bloka. Statična možnost Huffman se uporablja za kratka sporočila, pri čemer fiksni prihranek, pridobljen z izpustitvijo drevesa, odtehta odstotek izgube stiskanja zaradi uporabe neoptimalne kode.

Stiskanje dosežemo z dvema korakoma:

* Ujemanje in zamenjava podvojenih nizov s kazalci.
* Zamenjava simbolov z novimi simboli glede na pogostost uporabe.

Če je znotraj stisnjenih blokov opažen podvojeni niz bajtov, se vstavi povratna referenca, ki namesto tega povezuje prejšnje mesto istega niza. Kodirano ujemanje s prejšnjim nizom je sestavljeno iz 8-bitne dolžine (3–258 bajtov) in 15-bitne razdalje (1–32,768 bajtov) do začetka dvojnika. Relativne reference se lahko vrnejo v poljubnem številu blokov, če se pojavi razdalja v zadnjih 32 KB dekodiranih nestisnjenih podatkov.

Če je razdalja manjša od dolžine, se dvojnik prekriva, kar kaže na ponovitev. Na primer, potek 10 enakih bajtov je lahko kodiran kot en bajt, ki mu sledi dvojnik dolžine 9, začenši s prejšnjim bajtom.

Iskanje po predhodnem besedilu za podvojene nize je najslabši del algoritma DEFLATE in delovanje, na katerega vplivajo nastavitve ravni stiskanja.

Druga stopnja stiskanja je nadomestitev pogosto uporabljenih simbolov s krajšimi predstavitvami in manj pogosto uporabljenih simbolov z daljšimi predstavitvami. Uporabljena metoda je Huffmanovo kodiranje, ki ustvarja nepregledno drevo z intervali, ki se ne prekrivajo, pri čemer je dolžina vsakega zaporedja obratno sorazmerna z verjetnostjo, da je treba ta simbol kodirati. Bolj verjetno je, da je treba kodirati simbol, krajša bo njegova sekvenca bitov.

## ENKRIPCIJA

Pri stiskanju datotek je možna tudi enkripcija datotek, da zagotovimo njihovo varnost. Poznamo več vrst enkripcije:

* AES 128, 192 in 256 bit
* Standard ZIP 2

AES enkripcija (Advanced Encryption Standard) znan tudi po imenu Rijndael je specifikacija za šifriranje elektronskih podatkov, ki jo je vzpostavil ameriški nacionalni inštitut za standarde in tehnologijo NIST leta 2001. AES je distribucija algoritma, ki sta ga razvila belgijska kriptografa Vincent Rijmen in Joan Daemen.

AES deluje po načelu matričnih permutacij in ima fiksno velikost bloka 128, 192 ali 256 bitov.

Velikost ključa, ki se uporablja za šifro AES, določa število krogov transformacije, ki pretvorijo vhod, imenovan navaden tekst (plaintext), v končni izhod, ki se imenuje šifriran tekst (cyphertext). Število krogov je naslednje:

* 10 krogov za 128-bitne kode,
* 12 krogov za 192-bitne kode
* 14 krogov za 256-bitne kode

Vsak krog je sestavljen iz več korakov obdelave, vključno s tistim, ki je odvisen od samega šifrirnega ključa. Za preoblikovanje šifriranega teksta nazaj v prvotni navaden tekst z uporabo istega šifrirnega ključa, se uporabi niz povratnih krogov.

ZIP 2 enkripcija je preprost simetrični sistem šifriranja na osnovi gesla ki je znan po zelo enostavnih vdorih in plaintext napadih.

## IMPLEMENTACIJA

Danes poznamo kar nekaj programske opreme, ki nam omogoča stiskanje datotek. Med najbolj popularnimi so WinZIP, WinRAR in 7-Zip.

### WinZIP

WinZip 1.0 je izšel aprila leta 1991 kot grafični uporabniški vmesnik (GUI) za PKZIP program Phil-a Katz-a. V začetku januarja 1991 je Nico Mak Computing izdal grafični uporabniški vmesnik za OS/2 Presentation Manager, imenovan PMZIP. Uporabil je različici OS/2 programov PKWARE, Inc. PKZIP in PKUNZIP. Leta 1993 je WinZip objavil uradno podporo strankam na forumu Windows Utility, v katerem je več kot 100.000 članov, ki zagotavljajo posodobitve. WinZip, ki ga je mogoče brezplačno naložiti, se je kmalu znašel med najbolj prodajanimi računalniškimi programi Windows kot del vključene programske opreme, vključno knjigo Windows 3.0, Windows Secrets, avtorja Briana Livingstona. Do leta 1994 je WinZip postal uradno orodje za stiskanje.

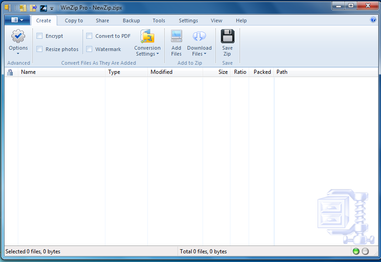
Ustvarjalci WinZip so od različice 5.0 leta 1993 vključili kompresijsko kodo iz projekta Info-ZIP, s čimer so odpravili potrebo po izvajanju PKZIP stiskanja.

WinZip 1.0 za Mac OS X je izšel novembra 2010. Ta različica je združljiva z Mac OS X 10.6 »Snow Leopard«. WinZip Mac Edition 2 vključuje podporo za OS X 10.8 »Mountain Lion«.

Podprte funkcije arhiva .ZIP:

* 128- in 256-bitno AES šifriranje poleg manj varne metode šifriranja PKZIP 2.0, uporabljene v prejšnjih različicah.
* Z WinZip 9.0, so podprti arhivi ZIP64, ki odstranijo največjo omejitev 65535 bitov za posamezen arhiv in omejitev velikosti 4 GB bodisi za arhiv kot za vsako datoteko.
* Podpora dodatnih metod stiskanja: bzip2 (9.0), PPMd (10.0), WavPack (11.0), LZMA (12.0), JPEG (12.0), Zipx (12.1), xz (18.0), MP3 (21.0).
* Podpora Unicode za zagotovitev prikaza mednarodnih črk za imena datotek. WinZip pred različico 11.2 ne podpira znakov Unicode v imenih datotek. Poskus dodajanja teh datotek v arhive vrne napako: Datoteke ni mogoče prebrati.

Slika 5: Program WinZIP



Vir: <https://en.wikipedia.org/wiki/WinZip#/media/File:Winzip-17-pro-screenshot.png> (dostopno 5. 8. 2019)

### WinRAR

Format datotek WinRAR in RAR se je začel razvijati leta 1993. Podpora za arhivsko obliko RAR5 z uporabo iste pripone datoteke RAR kot prejšnje različice je bila dodana v različici 5.0. Datoteke v starejši obliki datoteke RAR so bile od takrat imenovane RAR4. Različice WinRAR pred 5.0 ne podpirajo arhivov RAR5.

Oblika datoteke RAR5 je največjo velikost slovarja povečala na 1 GB. Na voljo je 11 različnih velikosti imenika stiskanja od 1 MB do 1 GB, privzeto v različici 5 pa se je povečalo s 4 MB na 32 MB, kar običajno izboljša razmerje stiskanja. Šifriranje AES se je pri uporabi povečalo s 128 na 256-bit. Največja dolžina imen za datoteke v arhivih RAR in ZIP se poveča na 2048 znakov.

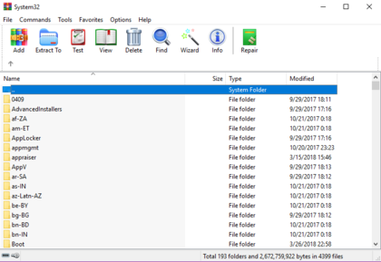
Možnosti, dodane v v5.0, vključujejo 256-bitni algoritem kriptiranja datotek BLAKE2 namesto privzetega 32-bitnega CRC32, odkrivanje podvojenih datotek, trde in simbolične povezave NTFS in zapis hitrega odpiranja, ki omogoča hitrejše odpiranje velikih arhivov.

Oblika datoteke RAR5 je odstranila komentarje za vsako datoteko (čeprav komentarji o arhivu še vedno ostajajo), preverjanje pristnosti in specializirane algoritme za stiskanje besedilnih in večpredstavnostnih datotek. RAR5 je tudi spremenil ime datoteke za razdeljene zvezke iz ime.rNN v ime.partNN.rar.

Podprte funkcije:

* Izdelava arhivov RAR ali ZIP,
* Dekompresiranje arhivov ARJ, BZIP2, CAB, GZ, ISO, JAR, LHA, RAR, TAR, UUE, XZ, Z, ZIP, ZIPX, 7z, 001 in datotek EXE, ki vsebujejo te arhivske formate,
* Preverjanje integritete za arhive ARJ, BZIP2, CAB, GZ, BZIP2, RAR, XZ, ZIP in 7z arhive,
* Stiskanje in dekompresija večpredstavnostnih datotek,
* Pri ustvarjanju arhivov RAR, podpora za največjo velikost datoteke 16 EB, približno 1,8 × 1019 bajtov ali 18 milijonov TB,
* Kompresijski slovar od 1 MB do 1 GB (omejen je na 256 MB v 32-bitnih izdajah sistema Windows, čeprav 32-bitni Windows še vedno lahko dekompresira arhive z 1 GB slovarjem; privzeta velikost je 32 MB),
* Možnosti, privzeto omogočene za optimizacijo stiskanja izvedljivih datotek za procesorje x86 in zastarelega 64-bitnega Itanium ter delta stiskanje,
* Izbirni 256-bitna enkripcija datoteke,
* Izbirni "hitri odprti zapis" za hitrejše odpiranje datotek RAR,
* Sposobnost ustvarjanja arhivov z več zvezki (split),
* Sposobnost ustvarjanja datotek, ki se avtomatsko dekompresirajo,
* Podpora za napredne možnosti datotečnega sistema NTFS, kot so trde in simbolične povezave NTFS
* Izbirno shranjevanje časovnih žigov datotek: ustvarjanje, zadnji dostop.

Slika 6: Program WinRAR



Vir: <https://en.wikipedia.org/wiki/WinRAR#/media/File:WinRAR_screenshot.png> (dostopno 5. 8. 2019)

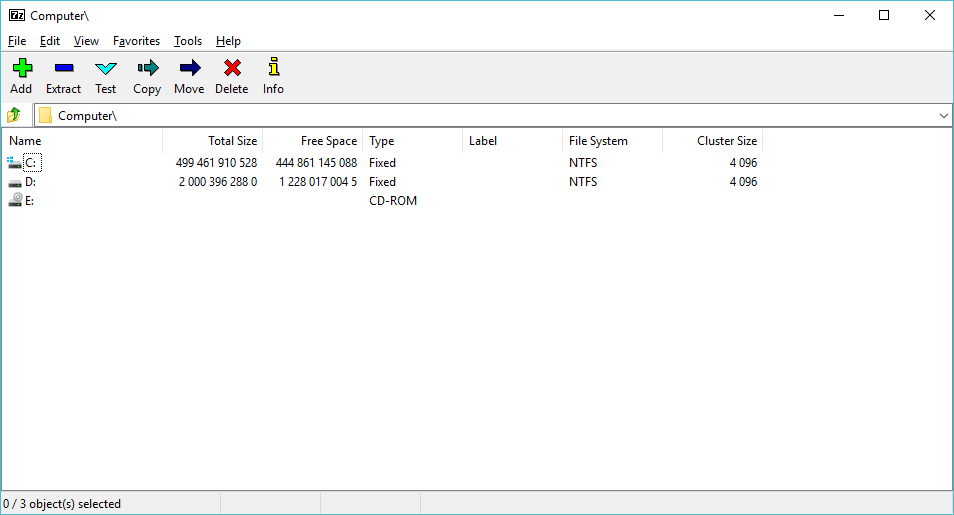
### 7-Zip

7-Zip je brezplačni in odprtokodni program za arhiviranje datotek. Razvil ga je Igor Pavlov leta 1999. 7-Zip uporablja lastno 7z arhivsko datoteko, lahko pa bere in piše več drugih arhivskih formatov. Program se lahko uporablja iz ukazne lupine kot ukaz p7zip, ali pa prek grafičnega uporabniškega vmesnika, ki vsebuje tudi integracijo lupine. Večina 7-Zip izvorne kode je pod licenco GNU/GPL, vendar je koda unRAR pod GNU/GPL z unRAR omejitvijo, ki pravi, da razvijalci ne smejo uporabljati kode za kompresijo RAR.

7-Zip podpira:

* 256-bitno AES šifriranje. Ko je datoteka šifrirana, morajo uporabniki predložiti geslo za ogled datotek v arhivu,
* WinZip datoteke AES šifrirni standard na voljo za enkripcijo ZIP arhivov z 256-bitnim AES geslom, vendar ne ponuja šifriranja imen datotek kot v arhivih 7z,
* Dinamično spremenljive velikosti, ki omogoča uporabo varnostnih kopij na izmenljivih nosilcih, kot so CD-ji in DVD-ji,
* upravitelj datotek,
* odpiranje datotek EXE kot arhiv, kar omogoča dekompresijo podatkov iz številnih programov tipa "Setup" ali "Installer" ali "Extract", ne da bi jih morali zagnati,
* dekompresija arhivov s poškodovanimi datotekami in preimenovanje datotek po potrebi,
* vmesnik ukazne lupine,
* grafični uporabniški vmesnik. Različica sistema Windows ima lasten GUI, Unix/Linux pa svojega.

Slika 7: Program 7-Zip



Vir: <https://en.wikipedia.org/wiki/7-Zip#/media/File:7-Zip.png> (dostopno 5. 8. 2019)

# PRAKTIČNI DEL

Naloga je sestaviti in implementirati enostavno aplikacijo za stiskanje datotek. V tem poglavju najprej opišem problem, kaj želim z aplikacijo doseči, katere so končne določbe in parametri, ki se jih je potrebno držati za uspešno programiranje aplikacije za stiskanje podatkov.

## OPIS PROBLEMA

Ustvariti želim enostavno aplikacijo z grafičnim uporabniškim vmesnikom za ustvarjanje stisnjenih datotek s pomočjo C# programskega jezika za Windows platformo.

Aplikacija mora imeti možnost odpiranja in ogleda datotek v nekem pogovornem oknu podobno kot program Raziskovalec na Windows platformi, v nadaljevanju raziskovalec. Funkcionalnosti Raziskovalca morajo biti odpiranje map, ki vsebujejo datoteke, ki jih želimo stisniti, drag/drop funkcionalnost, odpiranje že stisnjenih datotek in njihov pregled vsebine.

Ostale možnosti programa naj bi bile stiskanje/razširjanje datotek in intuitiven uporabniški vmesnik (GUI).

### Analiza trenutnih rešitev

Trenutne rešitve v svetu stiskanja datotek, so že implementirane v aplikacijah WinZIP, WinRAR in 7-Zip. V okolju Visual Studio in programskem jeziku C# imamo tudi že narejene implementacije algoritmov kot so DEFLATE in LZMA. Najdemo tudi zunanje knjižnice kot je SharpZipLib, ki imajo dodatne in bolj zmogljive funkcionalnosti, kot vgrajene skripte v Visual Studiu.

V nadaljevanju bom bolj podrobno analiziral hitrosti stiskanja v prej omenjenih implementacijah v aplikacijah in v moji aplikaciji.

### Končne določbe

Po analizah in pregledu možnih rešitev, sem prišel do zaključka, da programiranje svojega algoritma za stiskanje po načelu že nekega algoritma kot je npr. DEFLATE, je brezpredmetno, saj že obstajajo narejene implementacije teh algoritmov, ki nam omogočajo hitro in učinkovito implementacijo stiskanja v katerokoli aplikacijo, kjer to potrebujemo.

Odločil sem se za uporabo dveh knjižnic: DotNetZip za datoteke ZIP in SharpZipLib za ostale datoteke. DotNetZip uporablja DEFLATE algoritem in ima zelo enostavno implementacijo stiskanja z le nekaj vrsticami kode. SharpZipLib pa vsebuje implementacijo LZMA algoritma, ki lahko kompresira TAR, BZIP, BZIP2 in GZIP formate. Vsaki metodi v programu sem dodal tudi Stopwatch (štoparica) ki na koncu procesa stiskanja izpiše čas stiskanja datotek v milisekundah.

Funkcionalnosti aplikacije so odpiranje datotek in map s prikazom vsebine v raziskovalcu, ki je implementiran znotraj programa, stiskanje/razširjanje datotek in map in enkripcija ZIP datotek.

Za programiranje aplikacije sem uporabil Microsoft Visual Studio 2019 Community Edition in programiral Windows Forms aplikacijo v jeziku C#.

## REŠITEV PROBLEMA

Pri rešitvi problema se bom najprej posvetil vizualni in programski zasnovi programa, kako bo program deloval. Za vizualno zasnovo programa uporabljam aplikacijo Balsamiq Mockups 3.

### Programska zasnova

V C# Windows Forms delu z aplikacijami imamo znano začrtan postopek dela. Najprej sestavimo uporabniški vmesnik s pomočjo Form Designerja, kjer enostavno z drag/drop kreiramo izgled programa. Več o vizualni zasnovi v nadaljevanju.

Ko imamo želeno postavitev in izgled programa, lahko z dvoklikom miške na element npr. nek gumb programiramo funkcionalnosti.

Program je zasnovan tako, da imamo v sredini okno raziskovalca s katerim prikazujemo podatke iz datotek in map, ki so shranjene na našem računalniku. Imamo tri gumbe, ki imajo funkcionalnosti odpiranja, stiskanja in razširjanja datotek, kar upravlja glavni del aplikacije in kar nas tudi v tej nalogi najbolj zanima.

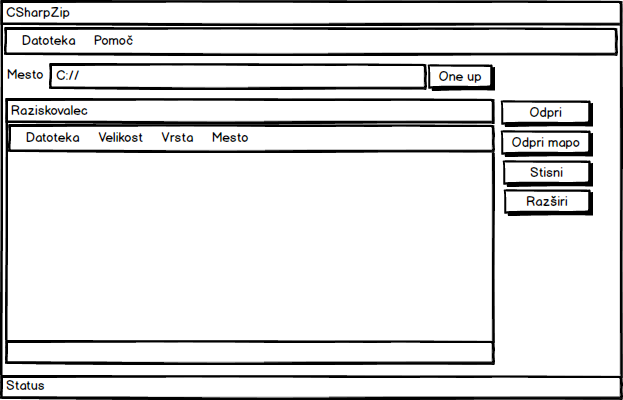
Program je zasnovan iz treh delov forme. Forma 1 je glavni del programa, s katerim upravljamo datoteke in mape. Druga forma je forma s funkcionalnostjo stiskanja datotek in map. Tu imamo na izbiro kateri način stiskanja bomo uporabili, ali bomo datoteko kriptirali z geslom in kam bomo to datoteko shranili. Tretje okno je okno shrani v, ki nam omogoča razširjanje datotek iz arhiva na poljubno mesto v našem računalniku.

### Vizualna zasnova

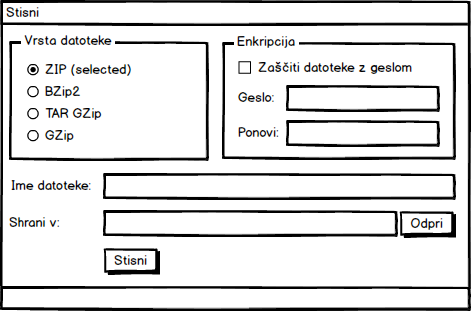
Z uporabo aplikacije Balsamiq Mockups 3 sem izdelal vizualno zasnovo aplikacije.

Prva slika prikazuje postavitev uporabniškega vmesnika za glavno formo aplikacije.

Slika 8: Vizualna zasnova osnovnega okna aplikacije



Slika 9: Vizualna zasnova okna s funkcionalnostmi stiskanja datotek



### Stiskanje datotek

Stiskanje datotek poteka s pomočjo stiskalnih algoritmov, ki so vključeni v izbrani knjižnici SharpZipLib in DotNetZip.

Stiskanje datotek poteka tako, da v glavnem oknu programa kliknemo na gumb Odpri mapo, ki sproži metodo BtnOpenFolder\_click(). Ta metoda najprej pobriše elemente v raziskovalcu, pokliče okno FolderBrowserDialog pri katerem izberemo direktorij. Ko izberemo direktorij, se FolderBrowserDialog zapre, v vnosno polje z imenom txtPath se izpiše pot mape in v raziskovalcu se izpišejo datoteke in mape, katere izpis ureja klic metode ListDirectory().

private void BtnOpenFolder\_Click(object sender, EventArgs e)

{

// Pobriši morebitne elemente v file explorerju

fileExplorer.Items.Clear();

// uporabi FolderBrowserDialog

using (FolderBrowserDialog fbd = new FolderBrowserDialog() { Description = "Izberi mapo" })

{

if (fbd.ShowDialog() == DialogResult.OK)

{

// Zapiši pot do direktorija v txtPath vnosno polje

txtPath.Text = fbd.SelectedPath;

// Klic zunanje metode za prikaz podatkov odprtega direktorija

ListDirectory(fbd.SelectedPath);

}

}

}

Ko izberemo datoteke, ki jih želimo stisniti, kliknemo na gumb stisni, ki pokliče metodo BtnCompress\_click(), ki najprej preveri, če so v raziskovalcu izbrane datoteke, in shrani poti do izbranih datotek v StringCollection z imenom filesList. Skripta nato odpre pogovorno okno Compress. Če ni bila izbrana nobena datoteka, se pojavi okno s sporočilom o napaki.

private void BtnCompress\_Click(object sender, EventArgs e)

{

// StringCollection izbranih datotek

int i = 0;

filesList = new StringCollection();

if (fileExplorer.SelectedItems.Count > 0)

{

// Preberi izbrane datoteke in jih shrani v StringCollection filesList

foreach (ListViewItem item in fileExplorer.SelectedItems)

{

string file = item.SubItems[0].Text;

filesList.Add(item.SubItems[3].Text + @"\" + file);

i++;

}

// Odpri dialog Compress

Compress compWin = new Compress();

compWin.ShowDialog();

}

else

{

MessageBox.Show("Nobena datoteka ni bila izbrana. Prosimo izberite datoteke.", "Napaka", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

}

V pogovornem oknu Compress imamo na voljo izbiro v katero vrsto datoteke želimo stisniti izbrane datoteke (ZIP, BZIP2, GZIP ali TAR), ali želimo datoteke zaščititi z geslom, ime in kam želimo datoteko shraniti.

S klikom na gumb »direktorij« se nam odpre pogovorno okno FolderBrowserDialog, kjer izberemo mesto kamor želimo datoteko shraniti. S klikom na Stisni, se sproži metoda BtnCompress\_Click(), ki izvede stiskanje datotek.

Metoda pridobi seznam datotek iz raziskovalca glavnega obrazca, iz FolderBrowserDialog-a pridobi mesto shranjevanja datoteke in iz vnosnega polja Ime datoteke pridobi željeno ime za shranjevanje. Metoda nato preveri vsa pravila, ki so bila določena iz strani uporabnika in izvede potrebno metodo stiskanja (vrsta datoteke in zaščita).

Za ZIP datoteke, je uporabljena knjižnica DotNetZip. DotNetZip za stiskanje uporablja dve metodi. Prva metoda AddFile, dodaja datoteke v arhiv, druga metoda Save pa shrani ZIP datoteko v želen direktorij na našem računalniku. Metoda nato še preveri, če je bila datoteka uspešno ustvarjena in pokaže pogovorno okno o uspehu izvajanja, če izvajanje ni bilo uspešno, opozori na napako.

//ustvari ZIP

using (ZipFile zip = new ZipFile())

{

Stopwatch stopwatch = Stopwatch.StartNew();

foreach (string file in filesList)

{

zip.AddFile(file, "");

}

zip.Save(fileName + ".zip");

stopwatch.Stop();

lblElapsed.Text += stopwatch.Elapsed.TotalMilliseconds.ToString();

}

if (File.Exists(fileName + ".zip"))

{

MessageBox.Show("Datoteka uspešno kreirana", "Uspeh!", MessageBoxButtons.OK, MessageBoxIcon.Information);

this.Hide();

}

else

{

MessageBox.Show("Prišlo je do napake, poskusite ponovno.", "Napaka", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

Za ostale vrste datotek, je uporabljena knjižnica SharpZipLib. SharpZipLib za delovanje uporablja tokove podatkov (Stream). Stream ali tok podatkov, je objekt, ki se ga uporablja za interakcijo s podatki ali datotekami (odpiranje, ustvarjanje datotek, pisanje v datoteke). Ostale vrste datotek (TAR, BZIP2 in GZIP) so ustvarjene precej podobno. Tokovi podatkov se konvertirajo v tok TAR datoteke s pomočjo metode CreateOutputTarArchive() in nato dodajajo v arhiv. Enako kot pri kreiranju ZIP datotek, metoda tudi tu na koncu izvajanja preveri, če je bila datoteka uspešno ustvarjena.

using (Stream tarFile = File.Create(fileName + ".tar"))

using (Stream tarStream = new TarOutputStream(tarFile))

using (TarArchive tar = TarArchive.CreateOutputTarArchive(tarStream))

{

Stopwatch stopwatch = Stopwatch.StartNew();

foreach (string file in filesList)

{

TarEntry tarEntry = TarEntry.CreateEntryFromFile(file);

tarEntry.Name = Path.GetFileName(file);

tar.WriteEntry(tarEntry, false);

}

stopwatch.Stop();

lblElapsed.Text += stopwatch.Elapsed.TotalMilliseconds.ToString();

}

if (File.Exists(fileName + ".tar"))

{

MessageBox.Show("Datoteka uspešno kreirana", "Uspeh!", MessageBoxButtons.OK, MessageBoxIcon.Information);

this.Hide();

}

else

{

MessageBox.Show("Prišlo je do napake, poskusite ponovno.", "Napaka", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

### Ekstrahiranje (razširjanje) datotek

Razširjanje datotek poteka podobno kot stiskanje. V glavnem pogovornem oknu programa, kliknemo na Odpri datoteko. Sproži se metoda BtnOpenFile\_Click(), ki najprej pobriše elemente v raziskovalcu, nato pokliče pogovorno okno OpenFileDialog, kjer izberemo želeno stisnjeno datoteko, in nato s pomočjo metode ListFileContents() izpiše vsebino datoteke v raziskovalec.

private void BtnOpenFile\_Click(object sender, EventArgs e)

{

// Pobriši morebitne elemente v file explorerju

fileExplorer.Items.Clear();

using (openFileDialog1)

{

if(openFileDialog1.ShowDialog() == DialogResult.OK)

{

// Zapiši pot do datoteke v txtPath vnosno polje

string filePath = openFileDialog1.InitialDirectory + openFileDialog1.FileName;

txtPath.Text = filePath;

// Klic zunanje metode za prikaz podatkov odprte datoteke

ListFileContents(filePath);

}

}

}

Razširjanje datotek, nato poteka tako, da moramo izbrati datoteke, ki jih želimo razširiti. Če tega ne storimo, aplikacija vrne pogovorno okno, ki javi napako, da ni bila izbrana nobena datoteka.

Ko izberemo datoteke, kliknemo na gumb Razširi in s pomočjo metode BtnExtract\_Click(), se odpre pogovorno okno FolderBrowserDialog, v katerem izberemo destinacijo razširjenih datotek. Pot destinacije, se shrani v spremenljivko savePath, nato metoda začne z branjem izbranih datotek v raziskovalcu in s preverjanjem vrste datotek (ZIP, TAR, BZIP2 ali GZIP).

Pri ZIP datotekah, metoda še dodatno preveri, če so datoteke kriptirane in odpre pogovorno okence za vpis gesla. Če je geslo pravilno, se datoteke razširijo na izbrano destinacijo, če ne se postopek ponovi.

Na koncu, metoda še preveri, če datoteke že obstajajo in ali jih želimo zamenjati.

if (Path.GetExtension(txtPath.Text) == ".zip")

{

ZipFile zip = Ionic.Zip.ZipFile.Read(txtPath.Text);

ZipEntry entry = zip[file];

if (zip[file].UsesEncryption == true)

{

PasswordPrompt passWin = new PasswordPrompt();

passWin.ShowDialog();

zip.Password = passWin.pass;

try

{

entry.ExtractWithPassword(savePath, passWin.pass);

}

catch (BadPasswordException)

{

if (MessageBox.Show("Napačno geslo", "Napaka", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error) == DialogResult.Retry)

passWin.ShowDialog();

}

}

string enExists = savePath + @"\" + entry.FileName;

if (File.Exists(enExists))

{

if (MessageBox.Show("Datoteka " + file + " že obstaja. Ali jo želite zamenjati?", "Datoteka obstaja", MessageBoxButtons.YesNo, MessageBoxIcon.Information) == DialogResult.Yes)

entry.Extract(savePath, ExtractExistingFileAction.OverwriteSilently);

else

break;

}

else

{

entry.Extract(savePath);

}

}

Razširjanje TAR, BZIP2 in GZIP datotek se enako kot stiskanje, izvaja s pomočjo podatkovnih tokov. Datoteka se prebere z metodo glede na vrsto datoteke, tok se spremeni v TAR in nato izpiše na želeno destinacijo. Dodatek k temu delu skripte sta še BinaryWriter (binarni zapisovalec) in Buffer ali nastavitev velikosti posameznega binarnega bloka toka podatkov, ki se dinamično nastavlja glede na velikost datotek.

else if (Path.GetExtension(txtPath.Text) == ".tgz")

{

Stream str = new FileStream(txtPath.Text, FileMode.Open, FileAccess.Read);

GZipInputStream gzStr = new GZipInputStream(str);

TarInputStream tar = new TarInputStream(gzStr);

TarEntry curEntry = tar.GetNextEntry();

while (curEntry != null)

{

if (curEntry.Name == file)

{

byte[] outBuffer = new byte[curEntry.Size];

FileStream fs = new FileStream(savePath + @"\" + curEntry.Name, FileMode.Create, FileAccess.Write);

BinaryWriter bw = new BinaryWriter(fs);

tar.Read(outBuffer, 0, (int)curEntry.Size);

bw.Write(outBuffer, 0, outBuffer.Length);

bw.Close();

}

curEntry = tar.GetNextEntry();

}

tar.Close();

}

### Odpiranje datotek in direktorijev za pregled vsebine

Za odpiranje datotek in direktorijev sta zadolženi dve metodi. Metoda ListDirectory() za prikaz vsebine direktorija in ListFileContents() za prikaz vsebine stisnjenih datotek.

Metoda ListDirectory() deluje tako, da pridobi podatke o direktoriju (DirectoryInfo) v katerih se nahajajo podatki o datotekah in pod direktorijih. Metoda nato preveri, če direktorij vsebuje pod direktorije in jih s pomočjo foreach zanke izpiše. Če direktorij nima pod direktorijev, se izpišejo samo datoteke in njihove informacije, ki so: ikona, ime datoteke, končnica ali vrste datoteke in mesto datoteke. Vsi podatki se izpišejo v raziskovalcu.

private void ListDirectory(string path)

{

DirectoryInfo directory = new DirectoryInfo(path);

DirectoryInfo[] subdirectories = directory.GetDirectories();

// Če ima direktorij poddirektorije prikaži direktorije in datoteke, če ne samo datoteke

if (subdirectories.Length != 0)

{

//Pridobi direktorije

foreach (DirectoryInfo subDirectory in subdirectories)

{

Icon iconForFile = DefaultIcons.FolderLarge;

ListViewItem row = new ListViewItem(subDirectory.Name);

imageList1.Images.Add(subDirectory.FullName, iconForFile);

row.ImageKey = subDirectory.FullName;

fileExplorer.Items.Add(row);

}

//Pridobi datoteke

foreach (string item in Directory.GetFiles(path))

{

FileInfo file = new FileInfo(item);

Icon iconForFile = SystemIcons.WinLogo;

ListViewItem row = new ListViewItem(file.Name);

iconForFile = Icon.ExtractAssociatedIcon(file.FullName);

imageList1.Images.Add(file.Extension, iconForFile);

row.ImageKey = file.Extension;

row.SubItems.Add(file.Length.ToString());

row.SubItems.Add(file.Extension);

row.SubItems.Add(file.Directory.ToString());

fileExplorer.Items.Add(row);

}

}

else

{

foreach (string item in Directory.GetFiles(path))

{

FileInfo file = new FileInfo(item);

Icon iconForFile = SystemIcons.WinLogo;

ListViewItem row = new ListViewItem(file.Name);

iconForFile = Icon.ExtractAssociatedIcon(file.FullName);

imageList1.Images.Add(file.Extension, iconForFile);

row.ImageKey = file.Extension;

row.SubItems.Add(file.Length.ToString());

row.SubItems.Add(file.Extension);

row.SubItems.Add(file.Directory.ToString());

fileExplorer.Items.Add(row);

}

}

}

Metoda ListFileContents() deluje podobno kot metoda ListDirectory(), s tem da najprej preveri vrsto datoteke, ki jo odpiramo (ZIP, TAR, BZIP2 ali GZIP), nato s pomočjo tokov podaatkov prebere datoteke in njihove informacije.

private void ListFileContents(string path)

{

fileExplorer.Items.Clear();

if (Path.GetExtension(path) == ".zip")

{

using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))

using (var zf = new ICSharpCode.SharpZipLib.Zip.ZipFile(fs))

{

foreach (ICSharpCode.SharpZipLib.Zip.ZipEntry item in zf)

{

if (item.IsDirectory)

continue;

Icon iconForFile = SystemIcons.WinLogo;

ListViewItem row = new ListViewItem(item.Name);

iconForFile = Icon.ExtractAssociatedIcon(path);

imageList1.Images.Add(item.GetType().ToString(), iconForFile);

row.ImageKey = Path.GetExtension(item.Name);

row.SubItems.Add(item.Size.ToString());

row.SubItems.Add(Path.GetExtension(item.Name));

fileExplorer.Items.Add(row);

}

}

}

else if (Path.GetExtension(path) == ".tar")

{

using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))

using (var tarFile = new TarInputStream(fs))

{

TarEntry entry;

while ((entry = tarFile.GetNextEntry()) != null)

{

Icon iconForFile = SystemIcons.WinLogo;

ListViewItem row = new ListViewItem(entry.Name);

iconForFile = Icon.ExtractAssociatedIcon(path);

imageList1.Images.Add(Path.GetExtension(entry.Name), iconForFile);

row.ImageKey = Path.GetExtension(entry.Name);

row.SubItems.Add(entry.Size.ToString());

row.SubItems.Add(Path.GetExtension(entry.Name));

fileExplorer.Items.Add(row);

}

}

}

else if (Path.GetExtension(path) == ".bz2")

{

using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))

using (BZip2InputStream bzFile = new BZip2InputStream(fs))

using (var tarFile = new TarInputStream(bzFile))

{

TarEntry entry;

while ((entry = tarFile.GetNextEntry()) != null)

{

Icon iconForFile = SystemIcons.WinLogo;

ListViewItem row = new ListViewItem(entry.Name);

iconForFile = Icon.ExtractAssociatedIcon(path);

imageList1.Images.Add(Path.GetExtension(entry.Name), iconForFile);

row.ImageKey = Path.GetExtension(entry.Name);

row.SubItems.Add(entry.Size.ToString());

row.SubItems.Add(Path.GetExtension(entry.Name));

fileExplorer.Items.Add(row);

}

}

}

else if (Path.GetExtension(path) == ".tgz")

{

using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))

using (GZipInputStream gzFile = new GZipInputStream(fs))

using (var tarFile = new TarInputStream(gzFile))

{

TarEntry entry;

while ((entry = tarFile.GetNextEntry()) != null)

{

Icon iconForFile = SystemIcons.WinLogo;

ListViewItem row = new ListViewItem(entry.Name);

iconForFile = Icon.ExtractAssociatedIcon(path);

imageList1.Images.Add(Path.GetExtension(entry.Name), iconForFile);

row.ImageKey = Path.GetExtension(entry.Name);

row.SubItems.Add(entry.Size.ToString());

row.SubItems.Add(Path.GetExtension(entry.Name));

fileExplorer.Items.Add(row);

}

}

}

}

### Enkripcija in dekripcija

Kriptiranje datotek je možno samo pri datotekah ZIP, zaradi enostavne implementacije. Knjižnica DotNetZip, za to uporablja metodo Password, ki prebere vnos gesla iz vnosnega polja in zapiše geslo v datoteko med stiskanjem. Metoda najprej preveri, če obe vnosni polji vsebujeta enaki gesli in nato nadaljuje z stiskanjem.

if (chkEncrypt.Checked == true)

{

if (txtPass.Text == txtPass2.Text)

{

//do encrypt

using (ZipFile zip = new ZipFile())

{

zip.Password = txtPass2.Text;

Stopwatch stopwatch = Stopwatch.StartNew();

foreach (string file in filesList)

{

zip.AddFile(file, "");

}

zip.Save(fileName + ".zip");

stopwatch.Stop();

lblElapsed.Text += stopwatch.Elapsed.TotalMilliseconds.ToString();

}

}

}

## UGOTOVITVE

Opis glavnih ugotovitev pri rešitvi problema in možne dodelave.

### Analiza in meritve

Analiza in meritve hitrosti izvajanja programa in primerjava z komercialnimi programi uporabljenimi vsakodnevno.

# SKLEPI

Ugotovitve, cilji in opredelitev razumevanja podanega problema.

# VIRI in LITERATURA

Podani viri.

1. “Hardcode” kodiranje programov je postopek programiranja, ki vključuje celotno kodo in potrebne podatke znotraj programa in ne iz zunanjih virov. (<https://en.wikipedia.org/wiki/Hard_coding>, dostopno 5. 8. 2019) [↑](#footnote-ref-1)
2. “Sliding window protocol” je značilnost paketnih protokolov za prenos podatkov, ki se uporabljajo kadar je potrebna zanesljiva oddaja paketov po vrstnem redu. Največ se uporablja pri internetnih povezavah v ISO/OSI podatkovnem modelu in pri TCP/IP povezavi. (<https://en.wikipedia.org/wiki/Sliding_window_protocol>, dostopno 5. 8. 2019) [↑](#footnote-ref-2)
3. Shareware programska oprema, je programska oprema, ki je uporabnikom na voljo brezplačno z neformalno možnostjo plačila licence. [↑](#footnote-ref-3)