FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY UNIVERZITY KOMENSKÉHO V BRATISLAVE

Katedra informatiky

PODPORA VÝUČBY TEÓRIE KÓDOVANIA

Diplomová práca

Konzultant: Doc. RNDr. Daniel Olejár, PhD.

Bratislava 2006

Matúš Sekera

POĎAKOVANIE

Na tomto mieste vyslovujem úprimnú vďaku môjmu konzultantovi Doc. RNDr. Danielovi Olejárovi, PhD. za cenné rady a pripomienky.

Úprimne ďakujem i mojej priateľke Zuzane za morálnu podporu pri písaní tejto práce.

ČESTNÉ PREHLÁSENIE

Prehlasujem, že som diplomovú prácu vypracoval samostatne, pod odborným vedením Doc. RNDr. Daniela Olejára, PhD. a uviedol som v nej všetky použité literárne pramene.

Obsah

1. Úvod

V dnešnej dobe modernej výpočtovej techniky a internetu sa používanie počítača na komunikáciu stalo súčasťou nášho každodenného života. A to, že sa nielen číselná, ale aj textová, zvuková alebo obrazová informácia zapisuje (kóduje) digitálne už v súčasnosti berieme ako samozrejmosť.

Málo ľudí však vie, ako zapisovať údaje, aby sa dali spoľahlivo prenášať cez nespoľahlivý komunikačný kanál, a sotva pozná metódy zápisu údajov, umožňujúce efektívnu komunikáciu pri ktorej je potrebné prenášať veľké objemy dát. Našťastie to na úspešné používanie počítačov ani nepotrebujú. Iné to všetko je u informatikov-profesionálov, ktorých úlohou je návrh a správa informačných a komunikačných systémov.

Kým bežní používatelia vystačia s poznaním, že tieto problémy existujú a rieši ich za nich systém a nanajvýš vedia používať programy na kompresiu údajov, informatici musia poznať riešenia problémov spojených s efektívnym kódovaním (či už z hľadiska odolnosti voči chybám, alebo kompresie) údajov. Preto patrí teória kódovania do základov informatického štúdia.

V odbore Informatika na FMFI UK sa „Základy teórie kódovania“ prednášajú už skoro 20 rokov. K tejto prednáške bola v priebehu niekoľkých rokov vytvorená elektronická učebnica, v ktorej sú popísané algoritmy kódovania a dekódovania, princípy tvorby kódovacích tabuliek pre rôzne typy kódov a princípy samoopravných kódov. Túto prednášku som, spolu s viacerými svojimi kolegami absolvoval aj ja. Pri štúdiu teórie kódovania sme narazili na niekoľko zaujímavých problémov.

Samotná teória (tak ako je prezentovaná v uvedenej knihe) nie je, najmä keď sa na to pozeráme s odstupom času, až taká komplikovaná. Napriek tomu sme pri príprave na skúšku mali problémy pochopiť tak podstatu jednotlivých algoritmov, ako aj ich použitie. Ukázalo sa, že je časovo náročné odprednášať myšlienky algoritmov kódovania a dekódovania, pretože tieto algoritmy pracujú s neštandardnými typmi údajov (prvkami konečných polí, vektormi, polynómami, okruhmi a i.), ktoré sa počas práce algoritmu častokrát modifikujú. Problém robí práve zobrazenie modifikujúcich sa údajov, ktoré nie je vždy priamočiare. Aj

keď na tabuli sa dá aplikovať „zotieracia metóda“ (alebo použiť animovaná prezentácia), študent má problémy zachytiť dynamiku zmien; t.j. ako si čo najpresnejšie zachytiť myšlienku algoritmu do svojich poznámok a „nestratiť sa“ v technických detailoch, ale neodísť s hmlistou predstavou „takto nejako by to mohlo fungovať“, na ktorej sa sotva dá stavať.

Druhý problém je s prácou s rôznymi kódami a možnosťou ich praktického použitia. Prednášaná teória sa dá ilustrovať nanajvýš na malých, značne zjednodušených príkladoch. Prezentovať „reálne riešenia“, ktoré by mali absolventi informatiky byť schopní vytvoriť, naráža na príliš veľkú pracnosť principiálne jednoduchých riešení. Napríklad, keď vygenerujeme kódovaciu tabuľku pre Huffmanov kód na základe frekvenčnej analýzy slovenského textu, je skoro nemožné manuálne aplikovať kódovú transformáciu na reálny slovenský text a pozorovať výsledok.

Preto sa zrodila myšlienka vytvoriť programový systém, na podporu vyučovania teórie kódovania, ktorý by

1. uľahčil prezentáciu myšlienok teórie kódovania vizualizáciou algoritmov kódovania a dekódovania,
2. uľahčil samotné štúdium teórie kódovania tým, že by používateľovi poskytol účinné nástroje na realizáciu technicky náročných operácií, ktoré sa pri kódovaní/dekódovaní používajú.

Vytvorenie a prezentácia tohto programu sú náplňou predloženej diplomovej práce. Samotná práca je organizovaná nasledovne:

- Najdôležitejšou časťou je samotný programový systém. Tento umožňuje vizualizáciu vybraných algoritmov popísaných v učebnici. Obsahuje implementáciu algoritmov kódovania a dekódovania, generovania kódovacích tabuliek pre Huffmanov kód a pre Fanov kód. Obsahuje implementáciu samoopravných kódov ako sú BCH kódy, obdĺžnikové kódy a Hammingove kódy. Používateľ môže skúmať reálne výsledky po aplikácii jednotlivých algoritmov.
- Pri vytváraní programového systému sme museli riešiť problémy algoritmického aj implementačného charakteru. Aby sme mohli vysvetliť podstatu problémov a zdôvodniť zvolené riešenia a nemuseli sa zakažným

odvolávať na učebnicu, v kapitolách 1.3, 1.4 a 1.5 sme stručne popísali základné problému, ktoré sa v teórii kódovania riešia.

- Kapitoly 2.1, 2.2, 2.3 a 2.4 sú venované dokumentácii vytvoreného systému. Obsahujú
  - Požiadavky na aplikáciu
  - Popis dizajnu aplikácie
  - Popis implementácie
  - Popis metodiky ďalšieho rozširovania aplikácie

Napriek pomerne jednoduchej formulácii problému sa ukázalo, že nájsť uspokojivé riešenie nie je jednoduché. Pracovnú verziu pripomienkovali študenti informatiky, ktorí v školskom roku 2004/5 absolvovali uvedenú prednášku. Časti ich pripomienok sa podarilo zohľadniť , ale základným poznatkom z pokusnej prevádzky systému je, že systém musí byť koncipovaný ako otvorený, aby bolo možné ho rozširovať o ďalšie moduly, resp. aby na jeho základe bolo možné budovať ďalšie aplikácie.

Aby nedošlo k nedorozumeniam pri čítaní tejto práce, v ďalšom texte si najprv zavedieme základné pojmy, s ktorými budeme pracovať a popíšeme si implementované algoritmy kódovania a dekódovania.

1.1 Základné pojmy

Zavedieme najdôležitejšie pojmy. Pri kódovaní budeme predpokladať, že správy, ktoré kódujeme, majú podobu textov nad nejakou abecedou. Formalizujeme tieto intuitívne zrejmé pojmy.

Pod *abecedou* budeme rozumieť ľubovoľnú neprázdnu konečnú množinu symbolov. *Prvky abecedy* budeme nazývať *znakmi*, alebo *písmenami*. Abecedu budeme označovať symbolom $\sum$ a v prípade potreby indexovať. Tam, kde bude z kontextu jasné, o akú abecedu sa jedná, budeme kvôli jednoduchosti slová „nad abecedou $\sum$ " vynechávať.

Postupnosť znakov (písmen) budeme nazývať *slovo*. Slová budeme označovať znakmi z konca latinskej abecedy. Počet znakov slova *v* budeme označovať symbolom *l(v)* a nazývať *dĺžkou slova v*. Pripúšťame aj možnosť, že slovo tvorí postupnosť znakov nulovej dĺžky. Takéto slovo budeme nazývať prázdnym slovom a označovať ho symbolom ε. Súvislá postupnosť *u* slova *v* sa nazýva podslovom slova *v*. Ak je *u* počiatočnou (koncovou) podpostupnosťou slova *v*, tak sa nazýva *prefixom* (*sufixom*) slova *v*. Ako uvidíme neskôr, prefixové kódy zohrávajú v teórii kódovania mimoriadne dôležitú úlohu.

Množinu slov budeme nazývať *jazykom*. Označovať ho budeme symbolom *L*, v prípade potreby indexovať.

Kódom budeme nazývať množinu navzájom rôznych slov nad spoločnou abecedou. Prvky kódu budeme nazývať *kódovými slovami*.

Nech $\Sigma_S = \{s_1, ...s_{m-1}\}$ je zdrojová abeceda a nech sú $v_0, ..., v_{m-1}$ navzájom rôzne slová nad kódovou abecedou Σ_C. Potom zobrazenie $s_i \to v_i, 0 \le i < m$ budeme nazývať kódovaním symbolov zdrojovej abecedy slovami nad kódovou abecedou.

Kódovaná správa má podobu postupnosti symbolov kódovej abecedy. Ak sú všetky kódové slová rovnakej dĺžky, dekódovanie je principiálne jednoduché, stačí rozdeliť prijatú postupnosť kódových symbolov na bloky rovnakej dĺžky a podľa tabuľky dekódovania im prideliť (napr.) symboly zdrojovej abecedy. Ak však bol na kódovanie použitý

nerovnomerný kód, určiť, kde začína kódové slovo môže byť problém. Jedným z riešení je vyhradenie symbolu kódovej abecedy na označenie konca slova (kódy s čiarkou), ale existujú aj efektívnejšie riešenia. Kód sa nazýva rozdeliteľný, ak sa každá postupnosť kódových symbolov dá rozdeliť na kódové slová jednoznačne, alebo sa vôbec nedá rozdeliť na kódové slová. Formálne zapísané:

Kód $V = \{v_0, ...v_{m-1}\}$ nad abecedou Σ sa nazýva rozdeliteľný, ak pre ľubovolnú rovnosť postupností kódových slov $v_{i_1}...v_{i_k} = v_{j_1}...v_{j_l}$ platí $l = k$, $i_1 = j_1, ...i_k = j_k$.

Jedným s rozdeliteľných kódov je aj prefixový kód. Jeho myšlienka spočíva v tom, že žiadne slovo nie je prefixom iného slova. Formálne zapísané:

Kód $V = v_0, ..., v_{m-1}$ sa nazýva prefixový kód, ak pre ľubovolné $v_i, v_j \in V, i \neq j$ v_i nie je prefixom v_j.

Ak do prefixového kódu nevieme pridať také slovo, aby kód ostal prefixový (lebo pridané slovo je prefixom iného kódového slova, alebo kódové slovo je prefixom pridaného slova). Tento kód nazveme úplným prefixovým kódom. Formálne zapísané:

Binárny rozdeliteľný kód V sa nazýva úplný práve vtedy, ak pre ľubovolnú binárnu postupnosť $\beta \in B^*$ existuje také kódové slovo $v_i \in V$, že buď postupnosť β je prefixom slova v_i, alebo slovo v_i je prefixom postupnosti β.

Prefixové kódy sa nazývajú aj automatovými kódmi, lebo na ich dekódovanie môžeme použiť konečný automat. *Konečný automat* je zariadenie, ktoré má vstupnú pásku, výstupnú pásku, riadiacu jednotku, čítaciu a zapisovaciu hlavu. Čítacia hlava sa pohybuje po vstupnej páske zľava doprava. Na vstupnej páske je zapísaný vstup. Zapisovacia hlava sa pohybuje po výstupnej páske a zapisuje na ňu výstup. Jej pohyb je riadený riadiacou jednotkou a môže sa pohybovať zľava doprava, sprava doľava alebo ostať stáť (v tomto prípade nezapíše na pásku žiaden výstup). Automat začína pracovať v počiatočnom stave a svoju prácu končí po dočítaní vstupu. Formálne:

Konečný automat je usporiadaná šestica $A = (\Sigma_i, \Sigma_o, Q, \Phi, \Psi, q)$, kde Σ_i je vstupná, Σ_o je výstupná abeceda, Q je konečná množina stavov, $\Phi : \Sigma_i \times Q \to Q$ je prechodová funkcia, $\Psi : \Sigma_i \times Q \to \Sigma_o$ je výstupná funkcia a q je počiatočný stav konečného automatu A.

V teórii kódovania existuje niekoľko spôsobov kódovania znakov zdrojovej abecedy. Aby sme vedeli porovnávať efektivitu jednotlivých kódov, zavádzame pojem ceny kódu. *Cena kódu* je počet symbolov kódovej abecedy pripadajúcich na zakódovanie jedného znaku zdrojovej abecedy. Formálne:

Nech je $P=\{p_0,...,p_{m-1}\}$ rozdelenie pravdepodobností znakov zdrojovej abecedy $\Sigma_S=\{a_0,...,a_{m-1}\}$, nech $V=\{v_0,...,v_{m-1}\}$ je kód kódujúci znaky kódovej abecedy, $a_i \to v_i, i=0,...,m-1$ a nech $l_i=l(v_i)$ sú dĺžky kódových slov kódu *V*. Potom cenou kódu *V* pri rozdelení pravdepodobností *P* nazveme $L(P,V)=\sum_{i=0}^{m-1} l_i p_i$.

Už vieme porovnávať cenu kódu. To znamená, že niektoré kódy sú lepšie ako iné. Majme kód, ktorý je pre danú zdrojovú abecedu najlepší. To znamená, že pre danú zdrojovú abecedu neexistuje kód s lepšou cenou. Takýto kód nazývame *optimálny kód*. Formálne zapísané:

Kód $V=\{v_0,...,v_{m-1}\}$ s dĺžkami kódových slov $l(v_i)=l_i, i=0,...m-1$ nazveme optimálnym kódom pre rozdelenie pravdepodobností $P=\{p_0,...,p_{m-1}\}$, ak pre ľubovolný kód $W=\{w_0,...,w_{m-1}\}$ platí $L(P,V)\le L(P,W)$.

Formalizovali sme niektoré základné pojmi, ktoré budeme v práci používať. Môžeme si teraz špecifikovať, čo je cieľom tejto práce.

1.2 Ciele práce

Cieľom tejto práce bolo vytvoriť aplikáciu, ktorá by umožnila študentom s minimálnymi znalosťami základov teórie kódovania:

- experimentovať s rôznymi typmi kódov
- odbúrať pracné výpočty pri skúmaní kódov a práci s nimi
- vizualizovať myšlienky algoritmov používaných na kódovanie a dekódovanie, na generovanie kódov

Vytvorená aplikácia by mala byť integrovaná s elektronickou učebnicou teórie kódovania. Súčasťou našej práce by malo byť vytvorenie skúšobného modulu, programu, ktorý umožní generovanie príkladov na skúšku.

Aby čitateľ nemusel narábať pri čítaní tejto práce s mnohými učebnicami a článkami, stručne popíšeme implementované algoritmy kódovania a dekódovania.

1.3 Popis kompresných algoritmov

V tejto kapitole sa budeme venovať popisu kompresných algoritmov kódovania a dekódovania. Jej cieľom je oboznámiť čitateľa s metódami kódovania a priblížiť ich využitie a dôležitosť.

Budeme sa venovať dvom skupinám algoritmov používaných pri kódovaní. Prvou skupinou sú kompresné algoritmy. Tieto algoritmy sa v praxi využívajú na efektívne uchovávanie a efektívny prenos objemnejších dát. Ich cieľom je zapísať rovnakú informáciu s použitím kratšieho reťazca symbolov. V nasledujúcich kapitolách popíšeme Huffmanov kód, Shannonov kód a Fanov kód, ktoré využívajú štatistický výskyt jednotlivých symbolov v texte a algoritmy, ktoré využívajú štatistický výskyt jednotlivých slov v texte, čo sú algoritmus LZ77 a algoritmus LZW.

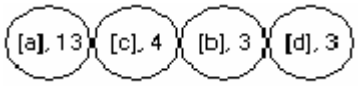
Druhú skupinu tvoria algoritmy, ktoré vstupný text nekomprimujú, ale transformujú ho na text, ktorý je vhodnejší na kompresiu použitím kompresného algoritmu využívajúceho štatistiké závislosti medzi symbolmi textu. Popíšeme transformáciu BWT a MTF.

1.3.1 Huffmanov kód

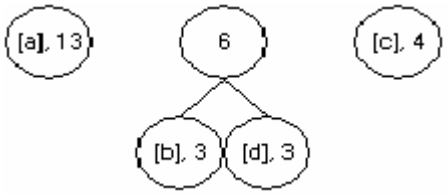
Huffmanov algoritmus, pôvodne vytvorený na efektívny prenos a ukladanie dát má dnes nezastupiteľnú úlohu aj mimo oblasti kompresie a kódovania. Napríklad pri konštrukcii optimálnych vyhľadávacích stromoch (Zimmerman 1959; Hu and Tucker 1971; Itai 1976), alebo pri spájaní zoznamov (Brent a Kung 1978), pri generovaní optimálnych vyhodnocovacích stromov pri preklade výrazov v kompilátoroch. (Lelewer, Hirschberg; 2005)

Huffmanov kód patrí medzi nerovnomerné kódy. Základná myšlienka Huffmanovho kódu je nasledovná: kódovať znaky s vysokou pravdepodbnosťou výskytu pomocou kratších reťazcov a znaky s nízkou pravdepodobnosťou výskytu pomocou dlhších reťazcov znakov kódovej abecedy (kódových slov).

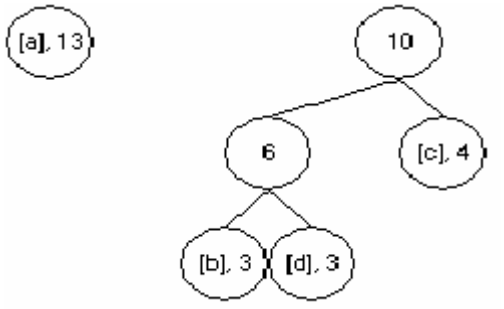
Konštrukcia optimálneho kódu sa dá jednoducho popísať pomocou stromov (acyklický graf). Napríklad pre kódovanie do binárnej abecedy použijeme binárny strom. Ilustrujme si vytváranie kódu na príklade. Majme vstupný reťazec v=acabacadaaabaddcabcaaaa. Počet výskytov symbolu „a“ v reťazci v je $\#_a(v)=13$, počet výskytov symbolu „b“ v reťazci v je $\#_b(v)=3$, $\#_c(v)=4$ a $\#_d(v)=3$. Pre každý symbol si zostrojíme strom. Strom bude obsahovať iba jeden vrchol, v ktorom si budeme uchovávať informáciu o tom, aký symbol reprezentuje a počet jeho výskytov v texte. Usporiadajme stromy podľa početnosti výskytu v koreni (v tomto prípade v jedinom vrchole stromu).

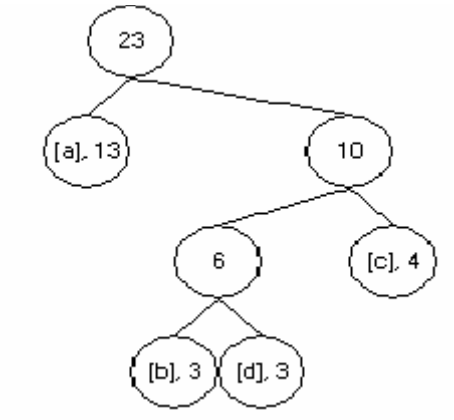


Zoberme posledné dva stromy a vytvorme z nich jeden a to nasledovne. Vytvoríme nový vrchol (koreň), ku ktorému pripojíme posledné dva stromy ako jeho synov a odstránime ich, pričom početnosť výskytu v tomto vrchole bude súčet výskytov v pripojených koreňoch. Utriedime stromy a dostaneme



Tento postup opakujeme, kým nám nevznikne jediný strom.





Ohodnotíme hrany vzniknutého stromu napríklad pravidlom: hrane vedúcej do ľavého vrcholu priradíme hodnotu „0“ a hrane vedúcej do pravého vrcholu priradíme hodnotu „1“. Sledovaním cesty z koreňa k listu dostaneme kód symbolu, ktorý je v liste uložený. V našom prípade symbol *a* budeme kódovať ako „0“, symbol *b* budeme kódovať ako „100“, symbol *c* ako „101“ a symbol *d* ako „11“.

Huffmanov algoritmus má dve časti a to určovanie dĺžok kódových slov a generovanie kódových slov. Je mnoho spôsobov ako vhodne modifikovať generovanie kódového slova. Jediná podmienka je, aby bol výsledný kód prefixový. Častý spôsob generovania kódových slov je taký, že sa hranám stromu smerujúcim do ľavých synov priradí hodnotu 0 a hranám smerujúcim do pravých synov priradí hodnotu 1. Kódové slovo v liste je tvorené postupnosťou 0 a 1 od koreňa k listu. Huffmanov algoritmus nie vždy nájde dĺžky kódových slov s minimálnou celkovou dĺžkou a minimálnym najdlhším kódovým slovom. Napríklad majme text s pravdepodobnosťami výskytu symbolov {0.4, 0.2, 0.2, 0.1, 0.1}. Huffmanov kód bude bude mať dĺžky kódových slov {1, 2, 3, 4, 4}. Schwartz prišiel s obmenou Huffmanovho algoritmu a to nasledovne. Pri vkladaní nového stromu do poľa sa triediaci algoritmus nepozerá len na hodnotu v koreni, ale aj na hĺbku stromu. Takáto modifikácia algoritmu nám generuje dĺžky kódov s minimálnou dĺžkou maximálneho kódového slova a s minimálnou celkovou dĺžkou kódových slov. Ak použijeme pravdepodobnosti symbolov {0.4, 0.2, 0.2, 0.1, 0.1}, potom dĺžky kódových slov budú {2, 2, 2, 3, 3}. Schwartz-Kallickov algoritmus a neskôr algoritmus od Conella používajú Huffmanov algoritmus na určenie dĺžok kódových slov a následne určia hodnoty kódovým slovám tak, aby mali vlastnosť numerickej postupnosti (kódové slová rovnakej dĺžky tvoria numerickú postupnosť). Táto vlastnosť sa dá využiť na dosiahnutie ucelenejšej reprezentácie kódu a k výraznému urýchleniu kódovania a dekódovania. (Lelewer, Hirschberg; 2005)

Kódovaciu tabuľku dokážeme generovať v čase *O(n)* za predpokladu, že máme utriedené pole pravdepodobností výskytov znakov v texte. Čas potrebný na kódovanie a dekódovanie je závislý od reprezentácie kódovacej tabuľky. Ak je kódovacia tabuľka reprezentovaná binárnym stromov, dekódovať znamená sledovať cestu z koreňa podľa vstupu a pri príchode do listu dáme na výstup dekódovaný symbol. Časová zložitosť je *O(l)*, kde *l* je dĺžka kódového slova. Tabuľka indexovaná podľa symbolov je užitočná pri kódovaní. Čas na zakódovanie je *O(l)*. Connellov algoritmus využíva indexovanú tabuľku a distribúciu dĺžok kódových slov na kódovanie a dekódovanie v čase *O(c)*, kde *c* je rozdiel dĺžok kódových slov. Tanaka prezentoval implementáciu Huffmanovho algoritmu na kódovanie založenú na konečných automatoch, ktorá sa dá jednoducho softvérovo aj hardvérovo implementovať. (Lelewer, Hirschberg; 2005)

Ďalšou obmenou Huffmanovho kódu je vytvorenie adaptívneho Huffmanovho kódu. Na adaptívne kódovanie pomocou Huffmanoývh kódov prišli nezávisle Faller a Gallager. (Lelewer, Hirschberg; 2005)

Myšlienka adaptívneho Huffamnovho kóu je nasledovná. Na základe odhadu pravdepodobností výskytu symbolov v ostávajúcom texte generujú kódovaciu tabuľku symbolu zo vstupu na kódové slovo. Kód je adaptívny, mení sa tak, aby bol optimálny vždy pre ostávajúci vstup. Adaptívne kódy reagujú na pozíciu vo vstupnom texte. Kóder reaguje na charakteristiky vstupného textu. Dekóder musí ostať synchrónny a tiež si musí priebežne aktualizovať kódovací strom. Ďalšou výhodou je jednoprechodovosť algoritmu, aj keď v porovnaní na počet prenesených bitov voči klasickému Huffmanovmu algoritmu nemožno hovoriť o veľkom zlepšení. Zaujímavé môže byť porovnanie výkonu s klasickým statickým Huffmanovým algoritmom. Vo všeobecnosti neplatí, že adaptívny Huffmanov algoritmus je výkonnejší a efektívnejší ako statický Huffmanov algoritmus. (Lelewer, Hirschberg; 2005)

1.3.2 Shannonov kód

Doteraz sme sa zamýšľali nad tým, ako vytvoriť (kvázi) optimálny kód. Nekládli sme žiadne požiadavky na dĺžky kódových slov. Tu si položíme dve otázky. Ako zistíme, či existuje pre dané rozdelenie dĺžok kódových slov rozdeliteľný kód. Ak tento kód existuje,

ako ho zostrojíme. Na prvú otázku nám dáva odpoveď *Kraft-Mc Millanova nerovnosť*, ktorá hovorí: Nech sú $l_0,...,l_{m-1}$ ľubovoľné nenulové prirodzené čísla. Potom rozdeliteľný kód $V=\{v_0,...,v_{m-1}\}$ s dĺžkami kódových slov $l_i=l(v_i);i=0,...,m-1$ existuje práve vtedy, ak platí nerovnosť $\sum_{i=0}^{m-1}2^{-l_i}\le 1$. Dôkaz tohto tvrdenia možno nájsť napríklad v knižke *Úvod do teórie kódovania*.

Teraz vieme povedať, či existuje rozdeliteľný kód s požadovanými dĺžkami kódových slov. Jedna z možností, ako nájsť takýto kód je použiť *Shannonov kód*, ktorý je nielen rozdeliteľný, ale aj prefixový. To znamená, že ak máme rozdeliteľný kód, nie je problém pomocou Shannonovho kódu k nemu nájsť prefixový kód.

Ukážeme si, ako zostrojíme Shannonov kód s požadovanými dĺžkami kódových slov. Nech požadujeme nasledovné dĺžky kódových slov: 2, 3, 3, 4. Keďže $2^{-2}+2^{-3}+2^{-3}+2^{-4}<1$ vypočítame kódové slová. Zavedieme čísla q_i nasledovne: $q_0=0$, $q_k=\sum_{i=0}^{k-1}2^{-l_i}$, $k=1,...,m-1$. Kódové slovo v_i pozostáva z prvých l_i číslic nasledujúcich po rádovej čiarke v binárnom rozvoji čísla q_i. V našom prípade $v_0=00$, $v_1=010$, $v_2=011$, $v_3=1000$, čo je prefixový kód so slovami dĺžok 2, 3, 3, 4.

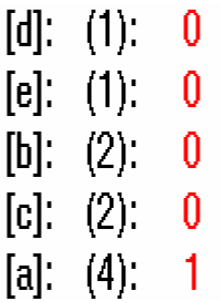
Shannonov kód nie je optimálny. Jeho konštrukcia je pomerne jednoduchá a rýchla v porovnaní s Huffmanovým kódom (nie je potrebné v pamäti uchovávať a preusporiadavať pole stromov). Niekedy je v praxi výhodnejšie použiť jednoduchší kód s horšími charakteristikami.

1.3.3 Fanov kód

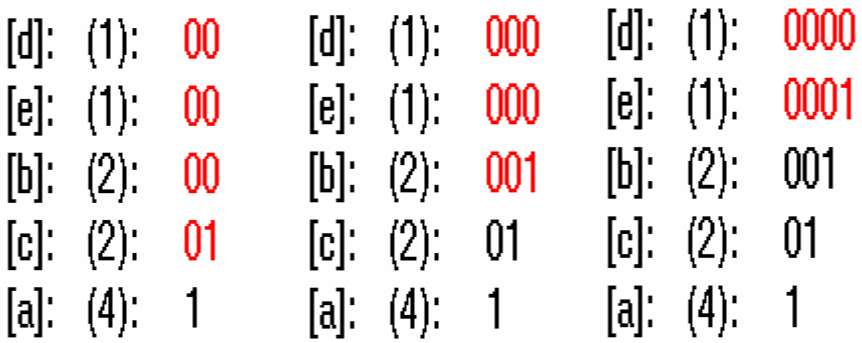
Fanov kód je ďalší kvázi optimálny kód s jednoduchou konštrukciou a dobrou cenou kódu.

Majme rozdelenie pravdepodobností *P*. Fanov kód pre dané rozdelenie pravdepodobnosti vstupných symbolov vygenerujeme nasledovne: utriedime pravdepodobnosti zostupne. Jednotlivým pravdepodobnostiam pridelíme prázdne slová. Ak tabuľka obsahuje aspoň dva riadky, rozdelíme ju na dve časti a to tak, aby sa súčet pravdepodobností v hornej časti líšil čo najmenej od súčtu pravdepodobností v dolnej časti tabuľky. Slová v hornej časti tabuľky zreťazíme sprava so znakom 0, slová z dolnej časti tabuľky zreťazíme sprava so znakom 1. Rekurzívne pokračujeme v spracovávaní hornej a dolnej časti tabuľky. (Olejár, Stanek; 2002)

Demonštrujem si generovanie Fanovho kódu na príklade. Majme vstupný reťazec *v*, pre ktorý platí $\#_a(v)=4$, $\#_b(v)=2$, $\#_c(v)=2$ a $\#_d(v)=1$, $\#_e(v)=1$. V prvom kroku rozdelíme tabuľku na dve časti tak, aby bol rozdiel pravdepodobností medzi hornou a spodnou časťou tabuľky minimálny. Hornej časti priradíme 0 a spodnej 1



Pokračujeme rekurzívne ďalej. Rozdelíme hornú časť tabuľky na dve časti, tak aby bol rozdiel pravdepodobností minimálny.



Rekurzívne delenie končí, keď časť tabuľky, ktorú máme deliť, obsahuje len jeden znak.

1.3.4 Algoritmus LZ77

Algoritmus kompresie LZ77 patrí medzi slovníkové metódy kompresie. Boli ním inšpirované aj ďalšie slovníkové algoritmy. Myšlienka algoritmu je v hľadaní podreťazcov v posledných *w* prečítaných znakoch, ktoré sú zároveň prefixom nasledujúcich *p* znakoch.

Označme vstupný text T a nech jeho dĺžka je n. Nech $p \in \{0,...,n-1\}$ označuje pozíciu. Okno je podreťazec $T[p-w,...,p-1]$ a buffer je postupnosť $T[p,...,n-1]$. Hľadáme maximálne $k \leq n-p$ také, že existuje $i \in \{p-w,...,p-k\}$: $T[i,...,i+k-1] = T[p,...,p+k-1]$. (Olejár, Stanek; 2002)

Algoritmus kompresie:

$p \leftarrow 1$
pokiaľ $p<n$ **opakuj {**
    nájdi *i, k* **take, že** $k \leq n-p \quad i \in \{p-w,...,p-k\} \quad T[i,...,i+k-1] = T[p,...,p+k-1]$
    výstup $\langle i-(p-w)+1, k, T[p+k] \rangle$
    $p \leftarrow p+k+1$
}

Dekompresia je jednoduchá. Budujeme si a udržiavame okno, v tomto prípade posledných *w* znakov daných na výstup. Na začiatku je okno prázdne. Postupne čítame trojice zo vstupu a na výstup dáme príslušný reťazec z okna určený indexom a dĺžkou doplnený o znak z trojice. Ak má vstupná trojica tvar $\langle 0,0,x \rangle$, na výstup dáme *x*. (Olejár, Stanek; 2002)

Algoritmus dekompresie:

$T \leftarrow []$
$p \leftarrow 1$
pokiaľ $p<n$ **opakuj {**
    nech $\langle a,b,c \rangle$ **potom** $j \leftarrow a; k \leftarrow b; x \leftarrow c$
    ak $j=0$ **and** $k=0$ **tak výstup** x
    inak {
        výstup $T[j,..,k].x$
    }
    $p \leftarrow p+1$
    $T \leftarrow T.x$
}

LZ77 je relatívne rýchly algoritmus. Dekódovanie je oveľa rýchlejšie ako kódovanie. Pri kódovaní vyhľadávame najdlhšie zhodné podreťazce. Pri dekódovaní len vypisujeme reťazec na výstup. Na kompresný pomer vplýva voľba veľkosti okna. Čím máme väčšie okno, tým dlhšiu zhodu môžeme nájsť, no zvyšuje sa časová náročnosť algoritmu. Dlhšie okno zvyšuje počet bitov potrebných na zápis indexu a dĺžky na výstupe a predlžujú výstup. Pri krátkom okne môžeme prísť o cenné informácie o predchádzajúcej podobe textu. (Olejár, Stanek; 2002)

1.3.5 Algoritmus LZW

Ďalším algoritmom patriacim do skupiny slovníkových metód kompresie je LZW. LZW je vylepšením algoritmu LZ78. Špeciálna implementácia tohto algoritmu sa využíva na kompresiu v grafickom formáte GIF. Myšlienka algoritmu je založená na budovaní slovníka, ktorý sa využíva na kódovanie. (Olejár, Stanek; 2002)

Algoritmus funguje nasledovne. V pomocnej premennej si pamätáme na začiatku prázdny reťazec. Do slovníka zaradíme všetky znaky abecedy. Program prečíta jeden znak. Ak sa tento znak zreťazený s reťazcom v pomocnej premennej nachádza v slovníku, zreťazíme ho s reťazcom v pomocnej premennej a načítame ďalší znak vstupu. Ak sa zreťazenie načítaného znaku a reťazca v pomocnej premennej v slovníku nenachádza, vložíme zreťazený reťazec do slovníka a na výstup pošleme pozíciu reťazca v pomocnej

premennej v slovníku. Do pomocnej premennej vložíme znak zo vstupu. (Olejár, Stanek; 2002)

Algoritmus kompresie:

```
zaradíme všetky znaky abecedy do slovníka
w ← "
načítaj znak zo vstupu k
pokiaľ (k je rôzne od "") opakuj {
        ak w.k je v slovníku {
                w ← w.k
        } inak {
                výstup: kód w v slovníku
                pridaj w.k do slovníka
                w ← k
        }
        načítaj znak zo vstupu k
}
```

Dekompresia prebieha analogicky ako kompresia. Postupne konštruujeme slovník, pomocou ktorého dekódujeme. Na začiatku slovník naplníme vstupnou abecedou. (Olejár, Stanek; 2002)

Algoritmus dekompresie:

```
zaradíme všetky znaky abecedy do slovníka
načítaj znak zo vstupu k
l ← −1
pokiaľ (k je rôzne od ``) opakuj {
        ak k je v slovníku {
                výstup: slovo v slovníku na pozícii k
                ak l>−1 tak {
                        pridaj do slovníka (slovo na pozícii l zreťazené s prvým
                        znakom slova na pozícii k)
                }
        } inak {
                ak l>−1 tak {
                        pridaj do slovníka( slovo na pozícii l zreťazené s prvým
                        znakom slova na pozícii k)
                        výstup: slovo na pozícii l zreťazené s prvým znakom slova na
                        pozícii k
                }

        }
        načítaj znak zo vstupu k
        l ← k
}
```

Existujú rôzne modifikácie algoritmu LZW. Môžeme pracovať s premenlivou dĺžkou zápisu kódových slov v závislosti od aktuálnej veľkosti slovníka. Môžeme tiež odstraňovať staré reťazce zo slovníka. (Olejár, Stanek; 2002)

1.3.6 Algoritmus BWT

BWT nie je algoritmus na kompresiu dát. Je to invertovateľná transformácia, ktorá transformuje reťazec znakov. Transformácia pracuje nad blokom textu, ktorý preusporiada pomocou triediaceho algoritmu. Výstupný, preusporiadaný blok je vhodnejší na kompresiu.

Kódovacia transformácia pracuje nad blokom dát dĺžky n. Z reťazca vytvoríme ďalších $n-1$ reťazcov tak, že pôvodný reťazec rotujeme. Získané pole reťazcov utriedime.

Na výstup dáme posledné znaky utriedených reťazcov a pozíciu pôvodného reťazca v poli reťazcov. (Olejár, Stanek; 2002)

Algoritmus transformácie:

načítaj reťazec dĺžky n zo vstupu do premennej k
$p \leftarrow 1$
$P = [k]$
pokiaľ $(k < n)$ opakuj {
    $P \leftarrow P.CSHIFT(P[p])$
    $p \leftarrow p + 1$
}
utrieď P
výstup: posledné znaky prvkov poľa P

Dekódovanie je o niečo zložitejšie ako kódovanie. Na vstupe máme reťazec, ktorý pozostáva z posledných znakov utriedených reťazcov. Utriedením znakov tohto reťazca dostaneme prvé znaky pôvodného poľa reťazcov. Teraz ku každému znaku v reťazci priradíme znak z utriedeného vstupného reťazca (prvého stĺpca v poli reťazcov). Začneme prvým znakom. Priradíme mu prvý neobsadený výskyt svojho znaku v utriedenom reťazci. Tento znak v utriedenom reťazci označíme ako obsadený. Takto priradíme všetkým znakom zo vstupného reťazca znak v utriedenom reťazci. Teraz môžeme rekonštruovať pôvodný reťazec. Vieme, že posledný znak v riadku je v reťazci pred prvým znakom v riadku. Dokážeme spätne prejsť a odzadu rekonštruovať požadovaný reťazec. Potom si vzniknutý reťazec zapíšeme zrotujeme cyklickým posunom o jednu pozíciu a opäť uložíme. Nech je dĺžka reťazca *n*. Reťazec rotujeme a ukladáme *n*-krát. Množinu uložených reťazcov utriedime. Výsledný reťazec vyberieme z pozície ktorú sme dostali na vstupe. Problém v rekonštrukcii nastane, keď sa prvý a posledný znak v prvom riadku zhodujú. V tomto prípade reťazec pozostáva len z tohto jedného znaku. (Olejár, Stanek; 2002)

Algoritmus spätnej transformácie:

načítaj reťazec dĺžky n zo vstupu do premennej k
utrieď znaky reťazca k a ulož do p
$i \leftarrow 1$
pokiaľ $(i \leq n)$ opakuj {
    $V[k[i]] =$ prvý neobsadený výskyt $k[i]$ v p
    Obsaď prvý neobsadený výskyt $k[n-i]$ v p
    $i \leftarrow i+1$
}
$i \leftarrow 1$
$r \leftarrow V[n]$
pokiaľ $(i < n)$ opakuj {
    $r \leftarrow V[n-i].r$
    $i \leftarrow i+1$
}
rotuj r n-krát a ukladaj do T
daj na výstup reťazec $T[r]$

1.3.7 Algoritmus MTF

Move to fromt (MTF) je transformácia, ktorá sa snaží pretransformovať pozičnú blízkosť rovnakých znakov do štatistickej významnosti znakov. MTF nie je kompresný algoritmus, ale algoritmus na transformáciu vstupného textu pred kompresiou. Jeho snahou je zníženie entropie vstupného textu. (Olejár, Stanek; 2002)

MTF transformácia pracuje s poľom, ktorom má usporiadané znaky. Po prečítaní znaku zo vstupu dá na výstup index pozície na ktorej sa znak v poli nachádza. Znak v poli zároveň presunie v poli na začiatok. Takto pokračuje, kým nevyčerpá celý vstup. (Olejár, Stanek; 2002)

Algoritmus transformácie:

```
T je uriedené pole znakov
c ← znak zo vstupu
kým c ≠ koniec vstupu {
        i ← INDEX c v T
        v T presuň c na začiatok poľa
        výstup i
        c ← znak zo vstupu
}
```

Dekódovanie prebieha podobne ako kódovanie. Začíname s tým istým poľom utriedených znakov. Prečítame index zo vstupu, príslušný znak z poľa dáme na výstup, presunieme znak v poli na začiatok a načítame ďalší index zo vstupu. Toto opakujeme kým neprečítame celý vstup. (Olejár, Stanek; 2002)

Algoritmus spätnej transformácie:

```
T je uriedené pole znakov
i ← znak zo vstupu
kým c ≠ koniec vstupu {
        c ← T[i]
        v T presuň c na začiatok poľa
        výstup c
        i ← znak zo vstupu
}
```

1.4 Popis algoritmov odhaľujúcich chyby

Praktické uchovávanie a prenášanie informácii naráža na rôzne problémy spôsobené napríklad vplyvmi prostredia. Je preto potrebné zapisovať informácie tak, aby sa chyby, ktoré následne v kódovom texte vzniknú, dali odhaliť, alebo dokonca opraviť. Ďalšiu skupinu algortimov kódovanian tvoria algoritmy, ktoré transformujú vstupný text tak, aby sa stal odolnejší voči chybám. Kódy, ktoré sa na takýto účel používajú, nazývame samoopravnými kódmi. Budeme popisovať kódy s testovaním parity, obdĺžnikové kódy, Hammingov kód a BCH kód.

Aby sme mohli vytvoriť odolnejší kód voči chybám, pozrieme sa, čo sa môže s kódovanou správou stať. Predpokladáme, že môže nastať zámena jedného symbolu kódovej abecedy za iný symbol kódovej abecedy, predpokladáme, že každý symbol má rovnakú pravdepodobnosť zmeny na iný symbol, to, že sa jeden symbol zmení alebo nezmení neovplyvňuje zmenu ostatných symbolov.

V praxi, ak pri používaní úplných kódov nastane chyba v odvysielanom slove, prijímateľ príjme opäť kódové slovo, pričom nevie, že toto slovo nebolo odvysielané. Pri samoopravných kódoch, ak pri prenose nastala chyba, prijaté slovo s veľkou pravdepodobnosťou nie je kódové slovo, lebo podstata samoopravných kódov je v tom, že kódové slová tvoria podmnožinu všetkých možných slov.

Aby sme sa mohli bližšie venovať samoopravným kódom, zavedieme niekoľko dôležitých pojmov. *Hamingova váha* vektora je počej jeho nenulových zložiek. *Hamingova vzdialenosť* dvoch vektorov udáva, v koľkých zložkách sa tieto dva vektory odlišujú (označme ju $d(u, v)$). Minimálnou vzdialenosťou kódu C, budeme nazývať prirodzené číslo $d^* = \min_{u,v \in C} d(u,v)$. Vo všeobecnosti platí, ak by bola minimálna vzdialenosť kódu C menšia alebo rovná ako t, potom chybou váhy menšej alebo rovnej t by sa mohlo jedno kódové slovo transformovať na iné kódové slovo. Ak by bola minimálna vzdialenosť kódu C rovná $t+1$, kód C dokáže odhaľovať chyby váhy t. Aby kód C opravoval chyby váhy t, musí byť jeho minimálna vzdialenosť väčšia alebo rovná $2t+1$ (hovoríme, že kód C má opravnú schopnosť).

Zaviedli sme si pojmový aparát a môžeme si popísať samoopravné kódy bližšie. Začneme s jednoduchšími kódmi.

1.4.1 Testovanie parity

Testovanie parity podpostupnosti znakov kódového slova (výpočet kontrolných súm) je základná operácia, na ktorej sú postavené mnohé zložitejšie algoritmy kódovania a dekódovania. Existuje aj jednoduchý kód s malou opravnou schopnosťou (odhaľovanie chýb nepárnej váhy), ktorý je postavený na testovaní parity všetkých znakov kódového slova.

Kódovanie reťazca je jednoduché. Algoritmus pridá na koniec vstupnému binárnemu reťazcu jeden bit tak, aby bol počet jednotkových bitov párny. Doplnený bit sa nazýva paritným bitom.

Dekódovanie reťazca je jednoduché. Skontrolujeme počet jednotkových bitov. Ak je počet jednotkových bitov v prijatom slove párny, odoberieme z neho posledný (paritný) bit a pošleme slovo na výstup. Ak je v prijatom slove počet jednotkových bitov nepárny, signalizujeme chybu pri prenose.

Minimálna vzdialenosť pre testovanie parity je 2, čo znamená, že vie odhaľovať chyby váhy 1. Ak sa pozrieme na kód lepšie, zistíme nasledovné. Ak pri prenose nastane chyba nepárnej váhy, v prijatom slove sa táto chyba prejaví nepárnym počtom jednotkových bitov, algoritmus bude signalizovať chybu pri prenose. Ak by pri prenose nastala chyba párnej váhy, v prijatom slove sa parita jednotkových bitov nezmení a algoritmus nebude signalizovať chybu pri prenose.

1.4.2 Obdĺžnikové kódy

Obdĺžnikové kódy sú kódy patriace ku kódom odhaľujúcim/opravujúcim chyby. Myšlienka kódu je jednoduchá. Reťazec sa uloží do matice, ktorej sa pridá jeden riadok (ktorého bity sú paritnými bitmi príslušných stĺpcov) a jeden stĺpec (ktorého bity sú paritnými bitmi príslušných riadkov).

Kódovanie prebieha nasledovne. Zapíšeme blok binárneho vstupu do matice $m \times n$. K matici pridáme jeden kontrolný riadok a jeden kontrolný stĺpec. Na *i*-tom riadku kontrolného stĺpca sa bude nachádzať paritný bit *i*-teho riadku. V *j*-tom stĺpci kontrolného stĺpca sa bude nachádzať paritný bit pre *j*-ty stĺpec v matici. Takto upravenú maticu odošleme ako reťazec na výstup.

Dekódovanie prebieha nasledovne. Vytvoríme maticu $(m+1) \times (n+1)$. Vložíme do nej vstupný reťazec. K matici opäť (ako pri kódovaní) pridáme kontrolný riadok a kontrolný stĺpec, kde na *i*-tom riadku kontrolného stĺpca sa bude nachádzať paritný bit *i*-teho riadku. V *j*-tom stĺpci kontrolného stĺpca sa bude nachádzať paritný bit pre *j*-ty stĺpec v matici. Keďže pôvodná matica bola vhodne doplnená paritnými bitmi, v prípade bezchybného prenosu budú v novom pridanom kontrolnom stĺpci len hodnoty 0 a v novom pridanom kontrolnom riadku tiež hodnoty 0. V prípade, že pri prenose nastala chyba váhy 1, v novom pridanom riadku matice sa prejaví táto chyba hodnotou 1 na pozícii stĺpca v ktorom chyba nastala. Chyba váhy 1 sa tiež rovnako prejaví aj na novom pridanom riadku, kde na pozícii riadku v ktorom chyba nastala bude hodnota 1. Takto dostaneme presnú „polohu“ chyby váhy 1 ktorá nastala. Chybu opravíme jednoduchým prevrátim bitu. Ak by pri prenose nastalo viac chýb, prejaví sa to vo výsledných pridaných riadkoch a stĺpcoch nastavením príslušných hodnôt na 1. No kontrolný vektor riadku/stĺpca môže obsahovať buď samé 0 alebo viac ako 1 jednotku a preto nevieme určiť pozíciu chyby v matici. Vieme len, že chyba nastala no nevieme ju opraviť.

Minimálna vzdialenosť pre obdĺžnikové kódy je 3, čo znamená, že tieto kódy odhaľujú chyby váhy 2 a opravujú chyby váhy 1.

1.4.3 Hammingov kód

Hammingove kódy patria medzi najdôležitejšie zo samoopravných kódov. Hoci ich samoopravná schopnosť nie je veľká (opravujú chyby váhy 1), sú v istom zmysle optimálne (dokonalé kódy), ľahko sa konštruujú a majú jednoduché algoritmy kódovania a dekódovania. Naviac, patria do viacerých význačných tried samoopravných kódov

(lineárne kódy, cyklické kódy, BCH kódy) a vďaka svojej jednoduchosti sa dajú výhodne využiť aj pri výklade zložitejších tried samoopravných kódov.

Kódového slovo Hammingovho (vo všeobecnosti samoopravného) kódu obsahuje dva druhy/typy symbolov – informačné a kontrolné. Hammingov kód určený dvojicou parametrov (n,k), kde $n=2^m-1$, $m\geq 3$, $m\in N$, $k=2^m-1-m$, n je dĺžka kódového slova, k je dĺžka informácie a $n-k$ je počet kontrolných bitov. Informačné bity sú nositeľom informácie, kontrolné bity tvoria redundantnú informáciu, pridanú do prenášaného slova.

Ilustrujeme si princíp vytvárania Hammingových kódov, kódovanie a dekódovanie na Hammingovom $(7,4)$-kóde. Predpokladajme, že sme už vytvorili kódové slovo $v=(v_1,...,v_7)$. Z jednotlivých komponentov kódového slova vytvoríme tri kontrolné sumy s_0,s_1,s_2, pomocou ktorých budeme schopní rozšifrovať osem rozličných udalostí a to: pri prenose nenastala žiadna chyba, pri prenose nastala chyba váhy jedna v prvom, druhom, ..., v ôsmom komponente kódového slova. Nech $\sigma(i,n)$ označuje n-bitový vektor reprezentujúci číslo i. Nech $u=(u_1,...,u_n)$ a $v=(v_1,...,v_n)$ sú binárne vektory, symbolom $u\,\&\,v$ budeme označovať vektor $u\,\&\,v=(u_1v_1,...,u_nv_n)$. Potom pre kontrolné sumy platí: $s_j=\bigoplus_{\sigma(i,3)\&\sigma(2^j,3)=\sigma(2^j,3)} v_i$, čo v našom prípade je: $s_0=v_1\oplus v_3\oplus v_5\oplus v_7$, $s_1=v_2\oplus v_3\oplus v_6\oplus v_7$, $s_2=v_4\oplus v_6\oplus v_6\oplus v_7$. Komponent v_i sa vyskytuje v toľkých kontrolných sumách, ako je Hammingova váha binárneho zápisu $\sigma(i,3)$. Keďže existujú práve 3 binárne vektory dĺžky tri s Hammingovou váhou 1, reprezentujúce čísla 1, 2, 4, každý s komponentov v_1,v_2,v_3 vystupuje v jednej kontrolnej sume. To znamená, že pri ľubovoľnej voľbe komponentov v_3,v_5,v_6,v_7 kódového slova a vhodnou voľbou komponentov v_1,v_2,v_4 dosiahneme, že kontrolné sumy budú pre kódové slovo nulové. Stačí položiť $v_1=v_3\oplus v_5\oplus v_7$, $v_2=v_3\oplus v_6\oplus v_7$, $v_4=v_5\oplus v_6\oplus v_7$.

Kódovanie správ pomocou Hammingovho $(7,4)$-kódu prebieha tak, že sa správa najprv rozdelí na bloky dĺžky 4 a tie sa doplnia troma kontrolnými symbolmi na kódové slovo.

Dekódovanie Hammingovho $(7,4)$-kódu je nasledovné. Predpokladajme, že pri prenose nastala chyba v *i*-tom komponente kódového slova. Chyba spôsobí, že všetky kontrolné sumy, ktoré obsahujú komponent v_i nadobudnú hodnotu jedna. To sú práve tie sumy s_j, pre ktoré platí $\sigma(i,4) \,\&\, \sigma(2^j,4) = \sigma(2^j,4)$, teda binárny vektor $s = (s_2, s_1, s_0)$ predstavuje číslo $\sigma(i,4)$. Vektor hodnôt jednotlivých kontrolných súm sa nazýva *syndróm chyby*. V tomto prípade syndróm chyby predstavuje pozíciu, na ktorej chyba váhy jedna v kódovom slove vznikla. Ak chyba pri prenose nevznikla, hodnota syndrómu chyby je rovná nule.

Teraz porovnáme redundanciu (nadbytočnosť) Hammingovho kódu s obdĺžnikovými kódmi. Počet kontrolných symbolov *nazývame absolútna* redundancia kódu. Keďže kódy s rozličnými dĺžkami môžu mať rozličné počty kontrolných symbolov k celkovej dĺžke kódového slova, zavedieme pojem *relatívnej redundancie* kódu. Je definovaná ako podiel počtu kontrolných symbolov k celkovej dĺžke kódového slova. Pre jednoduchosť budeme uvažovať, že kódové slovo obdĺžnikového kódu bude mať tvar štvorca zo stranou *m*. Relatívna redundancia takéhoto kódu je $\frac{2m-1}{m^2}$. Pri Hammingovom (n,k)-kóde, kde $n = 2^m - 1$ a m je počet kontrolných symbolov, je relatívna redundancia $\frac{m}{2^m - 1}$. Oba tieto kódy opravujú chyby váhy jedna, no z hľadiska relatívnej redundancie vychádzajú lepšie Hammingove kódy.

Kódy sme popísali jednoduchým spôsobom. Ďalej budeme popisovať kódy abstraktnejšie. Popíšeme si aparát na vytváranie zložitejších kódov s väčšími opravnými schopnosťami.

1.5 BCH kód

BCH kódy patria medzi cyklické kódy. Aby sme ich mohli popísať, zavedieme postupne potrebný pojmový aparát a priblížime si abstraktnejší pohľad na samoopravné kódy.

1.5.1 Lineárne kódy

Jednou s možností, ako konštruovať samoopravné kódy, je nájť vhodnú algebraickú štruktúru, ktorej prvky by reprezentovali kódové slová. Napríklad môžeme použiť konečnú grupu a kódové slová môžeme reprezentovať jej podgrupou. Na zostrojenie samoopravného kódu použijeme zložitejšie štruktútry (aby sme dosiahli efektívnejšie kódovanie a dekódovanie) ako sú grupy. Majme dané konečné pole *GF(q)*, kde *q* je mocnina nejakého prvočísla *p*. Množina $GF(q)^n$ *n*-tíc (vektorov) nad poľom *GF(q)* s aditívnou operáciou „+“ a s multiplikatívnou operáciou „.“ tvorí vektorový priestor. Lineárnym kódom nad abecedou *GF(q)* je ľubovoľný vektorový podpriestor vektorového priestoru $GF(q)^n$. Ak je dimenzia vektorového priestoru *C* rovná *k*, lineárny kód *C* sa nazýva lineárnym *(n, k)*-kódom.

Vlastnosti lineárnych samoopravných kódov sú zrejmé z vlastností vektorových priestorov. Pripomenieme, že lineárny kód je ľubovoľná neprázdna množina vektorov *C* taká, že pre ľubovoľné $v_1, v_2, ..., v_m \in C$, $a_1, a_2, ..., a_m \in GF(q)$ platí $a_1 v_1 + a_2 v_2 + ... + a_m v_m \in C$. To znamená, že pre kódové slovo *u* a prvok $a \in GF(q)$ patrí do kódu *C* aj *au*. Pre ľubovoľné dve kódové slová *u* a *v* kódu *C* platí, že *u+v* a *u-v* je z *C*. Keďže *u-u=0*, nulové slovo vždy patrí do kódu *C*.

Nech *C* je lineárny podpriestor vektorového priestoru $GF(q)^n$. Množina $C^{\perp}$ definovaná $C^{\perp} = \{u \in GF(q)^n; \forall v \in C; \langle u, v \rangle = 0\}$ tvorí ortogonálny doplnok lineárneho podpriestoru *C* a tvorí lineárny podpriestor vektorového priestoru $GF(q)^n$.

Vďaka tomu, že lineárny kód *C* predstavuje lineárny podpriestor vektorového priestoru $GF(q)^n$, možno ho popísať efektívnejšie, ako tie blokové kódy, ktoré nemali žiadnu rozumnú štruktúru a bolo ich potrebné popísať vymenovaním všetkých kódových slov. Lineárny podpriestor je jednoznačne zadaný pomocou množiny vektorov, ktorá ho generuje. Spomedzi všetkých množín vektorov, generujúcich daný lineárny podpriestor (lineárny kód) *C* vyberieme množinu s minimálnym počtom prvkov, bázu a vektory prvky bázy zapíšeme ako riadky matice *G*. Matica *G* sa nazýva *generujúcou maticou lineárneho kódu C*, pretože ľubovoľný vektor – kódové slovo kódu *C* možno zapísať pomocou lineárnej kombinácie riadkov – vektorov matice *G*. Ak je *C* lineárnym podpriestorom dimenzie k vektorového priestoru dimenzie *n*, tak jeho generujúca matica *G* má *k* (lineárne nezávislých) riadkov a *n* stĺpcov. Pripomíname, že samotný kód *C* má potom q^k kódových slov. (Olejár, Stanek; 2002)

Generujúca matica umožňuje efektívne vytváranie kódových slov. Ľubovoľný vektor *i* z $GF(q)^n$; $i = (i_1,...,i_n)$ môžeme chápať ako *k*-ticu informačných symbolov (informačný vektor) a transformovať ho na kódové slovo nasledujúcim spôsobom $u = iG$; kde *G* je generujúca matica lineárneho *(n, k)* kódu. Pripomíname, že existuje viacero spôsobov výberu generujúcej matice *G* lineárneho kódu a tak sa informačnému vektoru *i* v závislosti od výberu *G* vo všeobecnosti priradia rozličné slová. (Olejár, Stanek; 2002)

Pri dekódovaní správ zakódovaných pomocou lineárneho kódu *C* možno výhodne použiť generujúcu maticu duálneho kódu $C^\perp$, ktorú označíme symbolom *H*. Ak je C lineárny *(n,k)*-kód, $C^\perp$ je lineárny *(n,n-k)*-kód a generujúca matica kódu $C^\perp$ má *n-k* riadkov a *n* stĺpcov, pričom riadky matice *H* tvoria vektory bázy lineárneho podpriestoru $C^\perp$. Keďže *C* je ortogonálny doplnok lineárneho podpriestoru $C^\perp$, každý vektor (kódové slovo) *u* z *C* je ortogonálny na ľubovoľný vektor *v z* $C^\perp$ a špeciálne, na ľubovoľný vektor – riadok matice *H*. To znamená, že *u* je kódové slovo práve vtedy, ak $uH^T = 0$ kde 0 je v tomto prípade nulový vektor dĺžky *n-k*. Keďže matica *H* umožňuje overiť, či je nejaké slovo kódovým slovom kódu *C*, nazýva sa *kontrolnou maticou kódu C*. Pripomíname ešte, že generujúca matica *G* kódu *C* je kontrolnou maticou jeho duálneho kódu $C^\perp$. (Olejár, Stanek; 2002)

Pomocou lineárnych kódov dokážeme efektívne konštruovať samoopravné kódy. Práca s dlhšími lineárnymi kódmi je výpočtovo náročná. V ďalšom texte budeme skúmať triedu cyklických kódov.

1.5.2 Cyklické kódy

Cyklické kódy tvoria podtriedu lineárnych kódov, majú silnejšiu algebraickú štruktúru, čo môže prispieť k efektívnejším metódam kódovania a dekódovania pri zachovaní dobrých vlastností lineárnych kódov.

Lineárny kód je cyklickým, ak pre každé jeho kódové slovo platí, že jeho cyklický posun je tiež kódovým slovom. Formálne zapísané, lineárny kód C nazveme cyklickým kódom, ak pre ľubovoľné kódové slovo $u = (u_0, u_1 ... u_{n-1}) \in C$ platí $u' = (u_{n-1}, u_0, ..., u_{n-2}) \in C$.

Z hľadiska efektivity konštrukcie cyklických kódov, ich kódovania a dekódovania sa javí hvodná polynomická reprezentácia cyklických kódov, teda reprezentácia kódových slov z $C \subset GF(q)^n$ pomocou polynómov z faktorového okruhu $GF(q)[x] \mid x^n - 1$. Vektorový priestor $GF(q)^n$ môžeme prirodzeným spôsobom zobratiť na faktorový okruh polynómov $GF(q)[x] \mid x^n - 1$ nasledovne: $\forall u \in GF(q)^n; u = (u_0, u_1, ..., u_{n-1}) \leftrightarrow u_0 + u_1 x + ... + u_{n-1} x^{n-1}$. Vo faktorovom okruhu je definované modulárne násobenie polynómov a cyklický posun zodpovedá súčinu polynómov $x.u(x)$. Z jeho vlastností vieme povedať, že v ňom existuje polynóm *g(x)*, ktorý je nenulový, normovaný a má spomedzi všetkých (nenulových) prvkov množiny C minimálny stupeň a je daný jednoznačne.[1] Polynóm *g(x)* budeme nazývať generujúci polynóm kódu *C*. Vieme dokázať, že *generujúci polynóm* delí $x^n - 1$, teda $x^n - 1 = g(x)h(x)$. Polynóm *h(x)* budeme nazývať *kontrolný polynóm* cyklického kódu *C*.

Polynóm $x^n - 1$ vyjadríme ako súčin ireducibilných polynómov nad poľom *GF(q)*: $x^n - 1 = f_1(x) f_2(x) ... f_l(x)$. Generujúci polynóm *g(x)* cyklického kódu dĺžky *n* nad poľom *GF(q)* sa dá potom vyjadriť ako súčin vybraných ireducibilných polynómov z rozkladu

[1] Čitateľ môže nájsť presné zdôvodnenie existencie takéhoto polynómu napríklad v OLEJÁR, D. STANEK, M. 2002: *Úvod do teórie kódovania*

$x^n - 1$: $g(x) = f_{i_1}(x) f_{i_2}(x) ... f_{i_j}(x)$. Keďže polynóm $x^n - 1$ má l ireducibilných faktorov nad poľom *GF(q)*, existuje 2^l rozličných generujúcich polynómov a práve toľko cyklických kódov dĺžky *n* nad poľom *GF(q)*. Cyklický kód však tvoria kódové polynómy, ktoré sú násobkami generujúceho polynómu; *u(x) = a(x)g(x)*. Kódový polynóm má stupeň nanajvýš $n-1$. Ak bude mať generujúci polynóm stupeň *n - k*, tak potom kód bude obsahovať 2^k kódových slov. Z toho vyplýva, že niektoré polynómy nebudú generovať použiteľné kódy. (Olejár, Stanek; 2002).

1.5.3 Kódovanie pomocou cyklických kódov

Správu, ktorú potrebujeme zakódovať pomocou cyklického *(n,k)*-kódu *C* (s generujúcim polynómom *g(x)* stupňa $n-k$ rozdelíme na postupnosť disjunktných blokov dĺžky *k* (informačné vektory). Každému informačnému vektoru $i = (i_0, i_1, ..., i_k)$ priradíme informačný polynóm $i(x) = i_0 + i_1 x + ... + i_{k-1} x^{k-1}$ stupňa nanajvýš $k-1$. Keďže kódové slová (polynómy) cyklického kódu *C* sú násobkami generujúceho polynómu *g(x)*, stačí informačný polynóm vynásobiť generujúcim polynómom a dostaneme kódový polynóm kódu *C* $u(x) = i(x) g(x)$. Takýto spôsob kódovania je korektný, ale je nesystematický, pretože z kódového polynómu *u(x)* sa nedá bezprostredne určiť informačný polynóm *i(x)*. Existuje aj systematický spôsob kódovania, ktorého podstata je nasledovná:

- Informačný polynóm *i(x)* vynásobíme polynómom x^{n-k}.
- Vypočítame $x^{n-k} \cdot i(x) \bmod g(x)$.
- Od polynómu $x^{n-k} \cdot i(x)$ odčítame $x^{n-k} \cdot i(x) \bmod g(x)$ a dostávame hľadané kódové slovo $x^{n-k} \cdot i(x) - x^{n-k} \cdot i(x) \bmod g(x)$.

Keďže stupeň polynómu $x^{n-k} \cdot i(x) \bmod g(x)$ je nanajvýš $n-k-1$ a prvky informačného vektora tvoria v kódovom slove koeficienty pri mocninách $x^{n-k}, ..., x^{n-1}$, v kódovom slove sú jednoznačne oddelené „informačné“ a „kontrolné“ symboly tak, ako sme požadovali. (Olejár, Stanek; 2002).

1.5.4 Dekódovanie pomocou cyklických kódov

Keďže cyklické kódy sú podmnožinou lineárnych kódov, možno na ich dekódovanie použiť tie isté metódy ako na dekódovanie lineárnych kódov. Pri dekódovaní lineárnych kódov s väčšou opravnou schopnosťou sme narážali na to, že si bolo potrebné pamätať rozsiahlu dekódovaciu tabuľku (obsahujúcu zoznam syndrómov chýb a im prislúchajúcich chybových vektorov). Využijeme algebraickú štruktúru cyklických kódov na zostrojenie efektívnejšieho algoritmu dekódovania. Predpokladajme, že informácia, ktorú spracovávame, je zapísaná vo forme polynómov. Do popisu spracovania zahrnieme aj kódovanie správ:

1. informačný vektor *i* transformujeme na informačný polynóm *i(x)*,
2. informačný polynóm (napríklad nesystematicky) transformujeme na kódový polynóm $u(x) = i(x) \cdot g(x)$ s vektorom koeficientov *u*,
3. koeficienty kódového polynómu sa prenášajú prenosovým kanálom. Počas prenosu vznikne chyba *e*, ktorá transformuje prenášané kódové slovo na slovo $v = u + e$. V polynomickom vyjadrení $v(x) = u(x) + e(x)$.
4. Príjemca interpretuje prijaté slovo *v* ako polynóm *v(x)*, vydelí prijatý polynóm generujúcim polynómom a vypočíta zvyšok po delení: $v(x) \bmod g(x) = \big(u(x) + e(x)\big) \bmod g(x) = u(x) \bmod g(x) + e(x) \bmod g(x) = e(x) \bmod g(x) = s(x)$

Výsledkom delenia je polynóm *s(x)*, ktorý sa nazýva syndrómový polynóm. Je zrejmé, že $\deg\big(s(x)\big) < \deg\big(g(x)\big)$. Keďže ľubovoľné kódové slovo *u(x)* je násobkom generujúceho polynómu *g(x)*, $u(x) \bmod g(x) = 0$ a teda syndrómový polynóm nezávisí od odvysielaného kódového polynómu, ale len od polynómu chýb. (Olejár, Stanek; 2002).

Pre syndrómový polynóm *s(x)* platí, že neexistujú dve rôzne chyby váhy menšej alebo rovnej opravnej schopnosti kódu s tým istým syndrómom.[2] Preto pre dekódovanie potrebujeme udržiavať v pamäti množiny dvojíc $\big(s(x), e(x)\big)$. Existujú efektívnejšie algoritmy dekódovania cyklických kódov, ktoré využívajú vlastnosti silnejšej algebraickej štruktúry cyklických kódov (napríklad Meggitov algoritmus alebo Error-trapping dekódovanie).

[2] Dôkaz tohto tvrdenia môže čitateľ nájsť napríklad v OLEJÁR, D. STANEK, M. 2002: *Úvod do teórie kódovania*

1.5.5 BCH kódy

V predchádzajúcich kapitolách sme si ozrejmili lineárne kódy, zamerali sme sa na ich podtriedu cyklických kódov. Ďalej sa budeme venovať BCH kódom, ktoré tvoria významnú podtriedu cyklických kódov.

BCH kódy sú definované nad poľom *GF(q)*, ktoré sú definované pomocou prvkov z rozšírenia pôvodného poľa $GF(q^m)$. Formálne zapísané, cyklický kód *C* sa nazýva BCH kódom, ak $\beta \in GF(q^m)$ je prvok rádu *n*, *t* je ľubovoľné prirodzené číslo a *l* je ľubovoľné celé číslo, $g(x) = lcm\left\{m_{\beta^{l+1}}(x), m_{\beta^{l+2}}(x), ..., m_{\beta^{l+2t}}(x)\right\}$, kde $m_{\beta^j}(x)$ je minimálny polynóm prvku β^j. BCH kód má svoje parametre. Podobne ako Hammingov kód, aj BCH kód označujeme ako BCH (n,k) kód, kde *n* je dĺžka kódového slova, *k* je počet informačných symbolov. Ďalším parametrom BCH kódov sú prvočíslo *q*, číslo *m* pričom platí $n = q^m - 1$, ďalej definujeme opravnú schopnosť kódu. Označujeme ju symbolom *t*.

Na základe opravnej schopnosti kódu určujeme *konštrukčnú vzdialenosť BCH kódu*. Jej hodnota je $2t-1$ [3] a je daná dĺžkou súvislej postupnosti mocnín prvku β, ktoré sú koreňmi generujúceho polynómu. Generujúci polynóm je definovaný ako najmenší spoločný násobok minimálnych polynómov svojich koreňov a a nie vždy musí postupnosť pozostávať len týchto polynómov, ktoré sú koreňom generujúceho polynómu. Potom je skutočná vzdialenosť kódu väčšia, ako je jeho konštrukčná vzdialenosť. Existujú odhady minimálnej vzdialenosti BCH kódov.

Kódovanie BCH kódov nie je nijak výnimočné, ide o systematický spôsob kódovania cyklických kódov, preto sa mu nebudeme špeciálne venovať. Pre dekódovanie BCH kódov existujú efektívnejšie metódy dekódovania ako pre cyklické kódy. Popíšeme si fungovanie Peterson-Gorenstein-Zierlerovho algoritmu (dekódera):[4]

1. Na základe prijatého slova $v(x)$ vypočítaj syndróm $S_1, S_2, ..., S_{2t}; S_j = v\left(\beta^j\right)$.

[3] Dôkaz tohto tvrdenia môže čitateľ nájsť napríklad v OLEJÁR, D. STANEK, M. 2002: *Úvod do teórie kódovania*
[4] Čitateľ môže nájsť presné zdôvodnenie korektnosti a úplnosti tohto algoritmu napríklad v OLEJÁR, D. STANEK, M. 2002: *Úvod do teórie kódovania*

2. Nájdi najväčšie prirodzené v také, že matica $M_v = \begin{bmatrix} S_1 & \dots & S_{v-1} & S_v \\ S_2 & \dots & S_v & S_{v+1} \\ \vdots & & & \vdots \\ S_v & \dots & S_{2v-2} & S_{2v-1} \end{bmatrix}$ je regulárna.
3. Vyrieš systém lineárnych rovníc $\begin{bmatrix} S_1 & \dots & S_{v-1} & S_v \\ S_2 & \dots & S_v & S_{v+1} \\ \vdots & & & \vdots \\ S_v & \dots & S_{2v-2} & S_{2v-1} \end{bmatrix} \cdot \begin{bmatrix} \lambda_v \\ \lambda_{v+1} \\ \vdots \\ \lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{v+1} \\ -S_{v+1} \\ \vdots \\ -S_{2v} \end{bmatrix}$ a urči koeficienty $\lambda_1, ..., \lambda_v$. Zostroj polynóm lokátorov chýb $\Lambda(x) = 1 + \lambda_1 x + \lambda_2 x^2 + ... + \lambda_v x^v$.
4. Nájdi korene polynómu lokátorov chýb $\Lambda(x)$ a urči pozície chýb. Ak je daný BCH kód binárny, tak invertuj bity na pozíciách určených lokátormi chýb a skonči, ináč pokračuj krokom 5.
5. Vyrieš systém lineárnych rovníc pre hodnoty chýb

$$\begin{aligned} Y_1 X_1 + Y_2 X_2 + ... + Y_v X_v &= S_1 \\ Y_1 X_1^2 + Y_2 X_2^2 + ... + Y_v X_v^2 &= S_2 \\ \vdots \qquad\qquad\qquad & \vdots \\ Y_1 X_1^v + Y_2 X_2^v + ... + Y_v X_v^v &= S_v \end{aligned}$$

Priblížili sme si teoretické základy, na ktorých je budovaná teória kódovania. Jej náročnosť rastie s „rafinovanosťou“ kódov. Operácie nad algebraickými štruktúrami sú často pracné, no dajú sa automatizovať. V ďalšom texte budeme popisovať softvérový program, ktorý má slúžiť ako pomôcka pri narábaní s kódmi.

2. Aplikácia

Súčasťou našej práce je program, ktorý má slúžiť hlavne ako pomôcka pri výučbe teórie kódovania. V nasledujúcich kapitolách si zhrnieme požiadavky na aplikáciu, popíšeme si jej dizajn, implementáciu a pravidlá pre rozširovanie jej funkčnosti. Návod na používanie aplikácie sme umiestnili do príloh tejto práce.

2.1 Požiadavky na aplikáciu

Aplikácia je určená študentom (prioritne študnetom informatiky), ktorí sa zaujímajú o teóriu kódovania a chcú pochopiť podstatu rozličných kódov, naučiť sa ich vytvárať a pracovať s nimi. Preto sa od aplikácie očakáva:

- možnosť experimentovania s kódmi
- odbúranie pracných výpočtov pri práci s kódmi
- vizualizácia algoritmov kódovania a dekódovania

Aplikácia by mala byť schopná kódovať a dekódovať pomocou Huffmanovho kódu text načítaný zo vstupného súboru alebo z klávesnice, generovať Huffmanove kódy na základe frekvenčnej analýzy, kódovať a dekódovať pomocou Fanovho kódu text načítaný zo vstupného súboru alebo z klávesnice, generovať Fanov kód na základe frekvenčnej analýzy textu, kódovať a dekódovať text algoritmom LZ77, kódovať a dekódovať text algoritmom LZW, transformovať text transformáciami BWT a MTF, kódovať text pomocou samoopravných obdĺžnikových kódov, Hammingovho $(15,11)$ kódu a BCH kódov. Aby sme mohli pozorovať opravnú schopnosť jednotlivých samoopravných kódov, aplikácia by mala vedieť simulovať prenosový kanál s nastaviteľnou spoľahlivosťou.

Keďže práca algoritmov sa ťažko staticky prezentuje (pri výklade učiva pri tabuli), aplikácia by mala umožňovať vizualizáciu generovania Huffmanovho kódu, Fanovho kódu, samotného procesu kódovania a dekódovania, vizualizáciu kompresie algoritmom LZ77 a algoritmom LZW, vizualizáciu kódovania pomocou obdĺžnikových kódov a Hammingovho $(15,11)$ kódu.

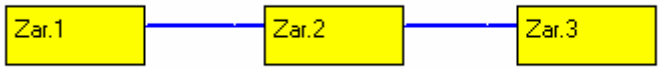
Aby bolo možné experimentovať s kódmi, aplikácia by mala umožniť použiť pri kódovaní a dekódovaní viaceré algoritmy, aby mohol užívateľ pozorovať ako sa jednotlivé algoritmy ovplyvňujú (napríklad aplikácia by mala umožniť najprv komprimovať text kompresným algoritmom, potom použiť samoopravný kód, alebo pred použitím kompresného algoritmu aplikovať na text transformáciu MTF alebo BWT).

Aj napriek tomu, že je aplikácia určená hlavne študentom informatiky, mala by byť spustiteľná bez aplikácii tretích strán. Jej používanie očakávame hlavne na pôde našej fakulty, preto by aplikácia mala byť bez problémov spustiteľná na väčšine počítačov prístupných na našej fakulte študentom.

Na základe týchto požiadaviek sme vytvorili dizajn aplikácie, ktorý si popíšeme v nasledujúcej kapitole.

2.2 Dizajn aplikácie

Aplikácia bude vo všeobecnosti transformovať/generovať text. Proces kódovania textu možno rozdeliť na nasledovné kroky: načítanie textu zo vstupu, transformácia textu, uloženie spracovaného textu na výstup. Napríklad kódovanie Fanovým kódom vieme rozdeliť nasledovne: načítanie vstupného textu, vygenerovanie kódovacej tabuľky pre Fanov kód, zakódovanie pomocou kódovacej tabuľky Fanovho kódu, uloženie na výstup. V každom kroku je potrebné vykonať operácie. Každý krok musí byť implementovaný. Nech jedno zariadenie implementuje jeden krok. Vo všeobecnosti zariadenie môže očakávať na vstupe vstupný text, kódovaciu tabuľku alebo aj vstupný text aj kódovaciu tabuľku. Na výstupe každého zariadenia môže byť výstupný text, alebo kódovacia tabuľka. Zariadenie je všeobecnosti ľubovoľná implementácia. Zariadenie samo o sebe nevie fungovať, preto je potrebné zariadenia navzájom „poprepájať“. Poprepájané zariadenia budeme nazývať systémom.



Kroky potrebné implementovať zobrazuje nasledovná tabuľka:

Krok	Vstup	Výstup	Popis
Načítaj dáta zo súboru	-	D[5]	Načíta dáta zo súboru na pevnom disku
Načítaj vstup z klávesnice	-	D	Načíta dáta zadané z klávesnice
Ulož dáta do súboru	D	-	Uloží dáta do súboru na pevnom disku
Zobraz dáta na obrazovku	D	-	Zobrazí dáta na obrazovku
Zakóduj dáta	D, KT[6]	D	Je kóder, ktorý na základe kódovacej tabuľky transformuje vstupné dáta a dá ich na výstup
Dekóduj dáta	D, KT	D	Je dekóder, ktorý vstupné dáta dekóduje podľa kódovacej tabuľky
Kódovacia tabuľka pre anglický text a Huffmanov kód	-	KT	Výstup je kódovacia tabuľka Huffmanov-ho kódu vygenerovaná na základe frekvenčnej analýzy anglického textu

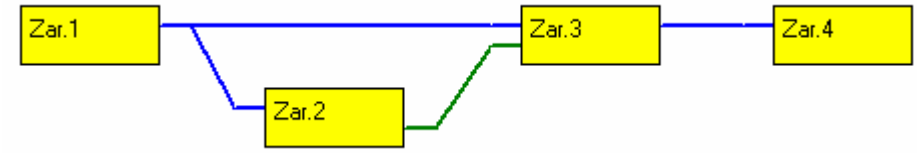
[5] Dáta
[6] Kódovacia tabuľka

Krok	Vstup	Výstup	Popis
Kódovacia tabuľka pre slovenský text a Huffmanov kód	-	KT	Dáva na výstup kódovaciu tabuľku Huffmanovho kódu vygenerovanú na základe frekvenčnej analýzy slovenského text
Kódovacia tabuľka pre anglický text a Fanov kód	-	KT	Dáva na výstup kódovaciu tabuľku Fanovho kódu vygenerovanú na základe frekvenčnej analýzy anglického textu
Kódovacia tabuľka pre slovenský text a Fanov kód	-	KT	Dáva na výstup kódovaciu tabuľku Fanovho kódu vygenerovanú na základe frekvenčnej analýzy slovenského textu
Generátor Huffmanovho kódu	D	KT	Vygeneruje kódovaciu tabuľku Huffmanovho kódu na základe frekvenčnej analýzy vstupných dát
Generátor Fanovho kódu	D	KT	Vygeneruje kódovaciu tabuľku Fanovho kódu na základe frekvenčnej analýzy vstupných dát
Kompresia LZ77	D	D	Vstupné dáta komprimuje algoritmom LZ77
Dekompresia LZ77	D	D	Vstupné dáta dekomprimuje algoritmom LZ77
Kompresia LZW	D	D	Vstupné dáta komprimuje algoritmom LZW
Dekompresia LZW	D	D	Vstupné dáta dekomprimuje algoritmom LZW
BWT transformácia	D	D	Vstupné dáta transformuje BWT transformáciou
BWT spätná transformácia	D	D	Na vstuných dátach vykoná spätnú BWT transformáciu
MTF transformácia	D	D	Vstupné dáta transformuje MTF transformáciou
MTF spätná transformácia	D	D	Na vstuných dátach vykoná spätnú MTF transformáciu

Krok	Vstup	Výstup	Popis
Kóder obdĺžnikového kódu	D	D	Vstupné dáta zakóduje obdĺžnikovým kódom
Dekóder obdĺžnikového kódu	D	D	Vstupné dáta dekóduje obdĺžnikovým kódom
Kóder Hammingovho kódu	D	D	Vstupné dáta zakóduje Hammingovým kódom
Dekóder Hammingovho kódu	D	D	Vstupné dáta dekóduje Hammingovým kódom
BCH kóder	D	D	Vstupné dáta zakóduje BCH kódom
BCH dekóder	D	D	Vstupné dáta dekóduje BCH kódom
Prenosový kanál	D	D	Náhodne modifikuje vstupné dáta

Celý systém sa bude aktivovať od konca, keďže na konci systému musí byť zariadenie, ktoré nemá výstup. Demonštrujme si myšlienku na jednoduchom príklade. Najprv si zoberme príklad, ktorý len načíta vstup zo súboru a uloží ho do výstupného súboru. Zariadenie ukladajúce dáta do súboru nemá žiaden výstup, teda systém začne pracovať od tohto zariadenia. Toto zariadenie si vypýta svoj vstup od zariadenia, ktoré načítava dáta zo súboru na dáva ich na výstup, teda v tomto prípade nášmu koncovému zariadeniu, ktoré ich uloží do súboru.

Zoberme si zložitejší systém. Chceme zakódovať text Huffmanovým kódom. Potrebujeme na to nasledovný systém



kde Zar.1 reprezentuje zariadenie, ktoré načíta vstupné dáta, Zar.2 reprezentuje zariadenie generátora Huffmanovho kódu, Zar.3 reprezentuje zariadenie kódera a Zar.4 reprezentuje zariadenie na ukladanie do súboru. Systém začne pracovať od konca, teda Zar.4 si vypýta svoj vstup od Zar.3, Zar.3 potrebuje na svoju prácu vstupné dáta, ktoré si vypýta od Zar.1 (to ich načíta zo súboru a pošle Zar.3), ďalej potrebuje kódovaciu tabuľku, ktorú si vypýta od Zar.2, Zar.2 potrebuje dáta na vygenerovanie kódovacej tabuľky (tie dostane od Zar.1)

a vygenerovanú kódovaciu tabuľku pošle Zar.3, Zar.3 zakóduje dáta od Zar.1 podľa tabuľky od Zar.2 a pošle ich Zar.4, Zar.4 ich uloží do súboru.

V ďalšom texte si popíšeme implementáciu aplikácie.

2.3 Implementácia aplikácia

Stáli sme pred úlohou implementácie aplikácie. Boli nám známe požiadavky a máme popísaný dizajn. Na implementáciu sme si zvolili Delphi 6.

2.3.1 Popis triedy TZariadenie

Na základe dizajnu aplikácie sme sa rozhodli vytvoriť framework, ktorý obsahuje triedu TZariadenie (zariadenie v dizajne). Jej potomkovia budú implementovať jednotlivé kroky procesu kódovania/dekódovania. Inštancie jej potomkov sa musia vedieť zapájať do systému a navzájom spolu komunikovať. Jednotlivé inštancie jej potomkov majú svoje atribúty, ktorých nastavovanie a udržiavanie je cez všeobecné rozhranie a je zastrešené touto triedou. Súčasťou jej verejného rozhrania sú atribúty *PGetSource*, *PGetCode*, *Poradie*, *Name*. V atribúte *PGetSource* si inštancia udržiava referenciu na inštanciu, ktorá jej poskytne na vstupe dáta. V atribúte *PGetCode* si inštancia pamätá referenciu na inštanciu, ktorá jej poskytne na vstupe kódovaciu tabuľku. Atribút *Poradie* je index v poli inštancií. V atribúte *Name* je uložené meno inštancie triedy. Verejné rozhranie obsahuje metódu *Run*, ktorá vykoná potrebné kontroly a zavolá metódu *Body* (abstraktná metóda implementovaná v potomkoch a implementuje konkrétny krok v procese kódovania/dekódovania). Metóda *SetName* nastaví inštancii meno. Metóda *GetSource* vráti dáta, ktoré inštancia dáva na výstup. Metóda *GetCode* vráti kódovaciu tabuľku vygenerovanú inštanciou. Na zistenie, či inštancia vracia kódovaciu tabuľku slúži metóda *ReturnCode*. Keď potrebujeme zistiť, či inštancia vracia dáta zavoláme metódu *ReturnSource*. Informáciu o tom, či daná inštancia potrebuje pre svoj beh na vstupe kódovaciu tabuľku, nám poskytne metóda *NeedCode*. Na zistenie potreby dát na vstupe slúži metóda *NeedSource*. Každá inštancia potomka triedy TZariadenie v konštruktore definuje svoje vlastnosti (nejde o vlastnosti v pravom slova zmysle, sú to parametre daného kódu, ktorý trieda implementuje). Zoznam vlastností vracia metóda *GetProperties*. Na prácu s týmto typom vlastností slúžia metódy *SetProperty* a *GetProperty*. Metóda *SetProperty* nastaví danú vlastnosť na požadovanú hodnotu, metódou *GetProperty* získame hodnotu danej vlastnosti. K vlastnostiam sme zaviedli typ. Zadefinovali sme 5 druhov typov vlastností a to vlastnosť jednoduchá (táto vlastnosť môže nadobúdať ľubovolné hodnoty), vlastnosť zoznamovú (vlastnosť môže nadobúdať len hodnoty

definované v jej zozname – napríklad hodnoty áno/nie), vlastnosť číslo (hodnota vlastnosti je číselná hodnota), vlastnosť s vybranými znakmi v hodnote (hodnota vlastnosti môže obsahovať len vybrané znaky – napríklad pri vlastnostiach, kde požadujeme hodnotu v inej ako desiatkovej sústave), vlastnosť s programovanou kontrolou (pri zadaní vlastnosti je možné použiť špeciálne dialógové okno – napríklad zadávanie ireducibilného polynómu). A zistenie typu danej vlastnosti slúži metóda *GetPropertyType*. Na zistenie povolenej abecedy reťazca v prípade vlastnosti s vybranými znakmi v hodnote slúži metóda *GetPropertyAllowedVal*. Keďže je možné systémy zariadení ukladať do súboru a neskôr ich z nich načítať, metóda *SaveDevice* uloží vlastnosti inštancie do súboru a metóda *LoadDevice* ich zo súboru obnoví. Tá istá inštancia/zariadenie v systéme môže poskytovať svoj výstup pre viaceré ďalšie zariadenia. Napríklad, ak daná inštancia generuje kódovaciu tabuľku a poskytuje ju viacerým inštanciám, aby táto tabuľka nemusela byť generovaná vždy keď sa o ňu požiada, vygeneruje sa len pri prvom požiadaní a potom ostane zapamätaná. Keď potrebujeme túto tabuľku zmazať z pamäte, zavoláme nad danou inštanciou metódu *ClearDevice*. Trieda TZariadenie zastrešuje aj funkčnosť vizualizácie algoritmov, no túto funkcionalitu si popíšeme neskôr v samostatnej kapitole. Definíciu triedy TZariadenie pripájame v prílohe tejto práce.

Ďalšou triedou vo frameworku je trieda TPloha. Inštancia tejto triedy tvorí „živnú pôdu“ pre inštancie potomkov triedy TZariadenie. Táto trieda vytvára inštancie potomkov tiredy TZariadenie, spravuje ich a nastavuje im vlastnosti. Stará sa tiež o ich vizualizáciu na obrazovke a spúšťa celý systém do chodu. Podrobnejšie si túto triedu popíšeme neskôr. Najprv si popíšeme potomkov triedy TZariadenie.

2.3.2 Potomkovia triedy TZariadenie

V tejto kapitole si popíšeme potomkov triedy TZariadenie. Pokiaľ nebude povedané ináč, každá trieda má implementované metódy *Body* a *Inicializácia*. V metóde body je implementácia daného kroku procesu a v metóde *Inicializácia* sú definované vlastnosti/parametre implementovaného kroku a ich prednastavená hodnota.

TFileInput.
Vlastnosti: Súbor – názov súboru, ktorý sa má načítať

Funkcia: načíta daný súbor

V/V: výstup je načítaný súbor

TKeyboardInput

Vlastnosti: -

Funkcia: načíta vstupný reťazec z klávesnice

V/V: výstup je zadaný text z klávesnice

TFileOutput

Vlastnosti: Súbor – názov súboru pre uloženie

Funkcia: uloží dáta zo vstupu do súboru

V/V: vstupom sú dáta, ktoré treba uložiť do súboru

TMonitorOutput

Vlastnosti: -

Funkcia: zobrazí dáta zo vstupu na obrazovku

V/V: vstupom sú dáta, ktoré treba zobraziť

TSKGenHuff

Vlastnosti: -

Funkcia: generuje kódovaciu tabuľku pre Huffmanov kód na základe frekvenčnej analýzy slovenského jazyka

V/V: výstup je vygenerovaná kódovacia tabuľka

TENGenHuff

Vlastnosti: -

Funkcia: generuje kódovaciu tabuľku pre Huffmanov kód na základe frekvenčnej analýzy anglického jazyka

V/V: výstup je vygenerovaná kódovacia tabuľka

TSKGenFano

Vlastnosti: -

Funkcia: generuje kódovaciu tabuľku pre Fanov kód na základe frekvenčnej analýzy slovenského jazyka

V/V: výstup je vygenerovaná kódovacia tabuľka

TENGenFano

Vlastnosti: -

Funkcia: generuje kódovaciu tabuľku pre Fanov kód na základe frekvenčnej analýzy slovenského jazyka

V/V: výstup je vygenerovaná kódovacia tabuľka

TTextGenHuff

Vlastnosti: -

Funkcia: generuje kódovaciu tabuľku pre Huffmanov kód na základe frekvenčnej analýzy vstupných dát

V/V: vstupom sú dáta na frekvenčnú analýzu, výstup je vygenerovaná kódovacia tabuľka

TTextGenFano

Vlastnosti: -

Funkcia: generuje kódovaciu tabuľku pre Fanov kód na základe frekvenčnej analýzy vstupných dát

V/V: vstupom sú dáta na frekvenčnú analýzu, výstup je vygenerovaná kódovacia tabuľka

TKoder

Vlastnosti: -

Funkcia: transformuje/kóduje vstupné dáta podľa vstupnej kódovacej tabuľky

V/V: vstupom sú dáta na kódovanie a kódovacia tabuľka, výstupom sú zakódované dáta

TDekoder

Vlastnosti: -

Funkcia: transformuje/dekóduje vstupné dáta podľa vstupnej kódovacej tabuľky

V/V: vstupom sú dáta na dekódovanie a kódovacia tabuľka, výstupom sú zakódované dáta

TLZ77Koder

Vlastnosti: Buffer – veľkosť buffera, Okno – veľkosť okna

Funkcia: kompresia algoritmom LZ77

V/V: vstupom sú dáta na kompresiu, výstupom sú komprimované dáta

TLZ77Dekoder

Vlastnosti: Buffer – veľkosť buffera, Okno – veľkosť okna

Funkcia: dekompresia algoritmom LZ77

V/V: vstupom sú dáta na dekompresiu, výstupom sú dekomprimované dáta

TLZWKoder

Vlastnosti: -

Funkcia: kompresia algoritmom LZW

V/V: vstupom sú dáta na kompresiu, výstupom sú komprimované dáta

TLZWDekoder

Vlastnosti: -

Funkcia: dekompresia algoritmom LZW

V/V: vstupom sú dáta na dekompresiu, výstupom sú dekomprimované dáta

TBWTKoder

Vlastnosti: Blok – veľkosť bloku

Funkcia: vykonanie BWT transformácie

V/V: vstupom sú dáta na transformáciu, výstupom sú transformované dáta

TBWTDekoder

Vlastnosti: Blok – veľkosť bloku

Funkcia: vykonanie spätnej BWT transformácie

V/V: vstupom sú dáta na spätnú transformáciu, výstupom sú spätne transformované dáta

TMTFKoder

Vlastnosti: -

Funkcia: vykonanie MTF transformácie

V/V: vstupom sú dáta na transformáciu, výstupom sú transformované dáta

TMTFDekoder

Vlastnosti: -

Funkcia: vykonanie spätnej MTF transformácie

V/V: vstupom sú dáta na spätnú transformáciu, výstupom sú spätne transformované dáta

TOrakulum

Vlastnosti: Predpoved – reťazec predpovedajúci vstup

Funkcia: vykonanie operácie $\oplus$ nad vstupným reťazcom a reťazcom predpovede.

V/V: vstupom sú dáta pre operáciu $\oplus$, výstupom sú dáta po operácii $\oplus$

TPrenosKanal

Vlastnosti: Spolahlivost – miera spoľahlivosti prenosového kanála

Funkcia: náhodná modifikácia vstupných dát podľa miery spoľahlivosti

V/V: vstupom sú dáta na modifikáciu, výstupom sú dáta, ktoré sú možno modifikované

TObdlznikKoder

Vlastnosti: Strana A, Strana B – veľkosť strán obdĺžnika

Funkcia: aplikácia obdĺžnikového kódu

V/V: vstupom sú dáta, výstup sú dáta v obdĺžnikovom kóde

TObdlznikDekoder

Vlastnosti: Strana A, Strana B – veľkosť strán obdĺžnika

Funkcia: dekódovanie obdĺžnikového kódu

V/V: vstupom sú dáta v obdĺžnikovom kóde, výstup sú dáta

THammingKoder

Vlastnosti: -

Funkcia: aplikácia Hammingovho kódu

V/V: vstupom sú dáta, výstup sú dáta v Hammingovom kóde

THammingDekoder

Vlastnosti: -

Funkcia: dekódovanie Hammingovho kódu

V/V: vstupom sú dáta v Hammingovom kóde, výstup sú dáta

TBCHKoder

Vlastnosti: q, m, polynom – Parametre pre BCH kód

Funkcia: aplikácia BCH kódu

V/V: vstupom sú dáta, výstup sú dáta v BCH kóde

Metódy navyše: SetProperty – implementuje kvôli zadávaniu a kontrole na prvočísla a zadávanie ireducibilného polynómu

TBCHDekoder

Vlastnosti: q, m, polynom – Parametre pre BCH kód

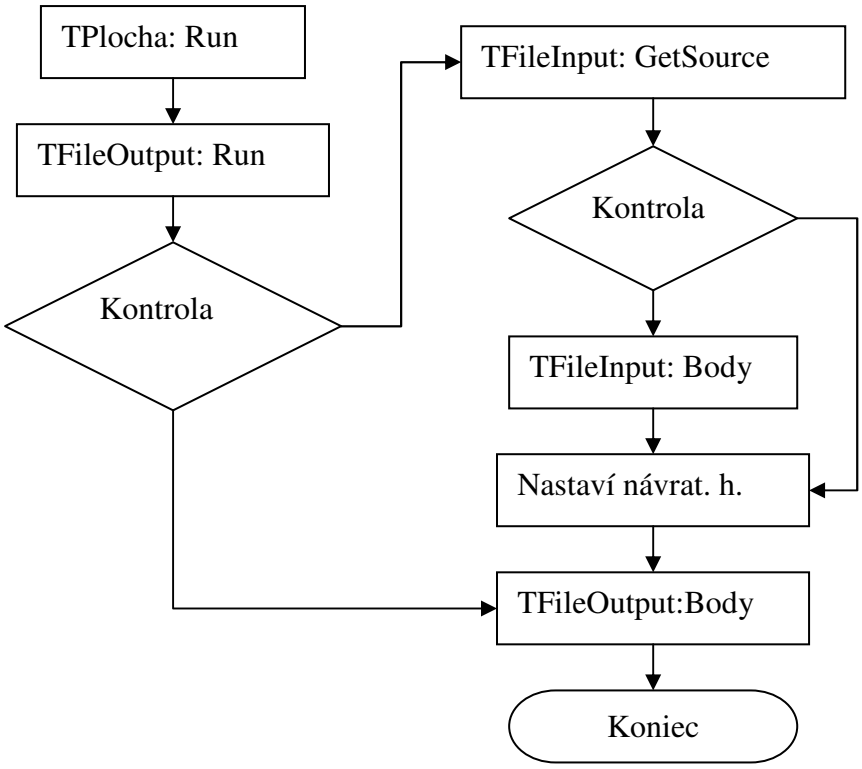
Funkcia: dekódovanie BCH kódu

V/V: vstupom sú dáta v BCH kóde, výstup sú dáta

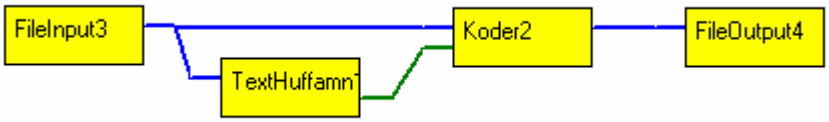
Metódy navyše: SetProperty – implementuje kvôli zadávaniu a kontrole na prvočísla a zadávanie ireducibilného polynómu

2.3.3 Príklad práce systému

Funkčnosť celého systému si môžeme demonštrovať na jednoduchom príklade. Najprv si zoberme príklad, ktorý len načíta vstup zo súboru (*TFileInput*) a uloží ho do výstupného súboru (*TFileOutput*). Zavolá sa metóda *Run* inštancovanej triedy *TPlocha*. Tá uvedie systém do prevádzky nasledovne. V poli zariadení na ploche vyhľadá objekt, ktorý nevracia ani text ani kód. V našom prípade systém nájde inštanciu triedy *TFileOutput* a zavolá jej metódu *Run*. V tejto medóde prebehne kontrola, ktorá preverí či je nastavený na vstupe zariadenia objekt vracajúci text. Ak áno, tak zavolá metódu *GetSource* objektu vracajúceho text nastaveného na vstupe. V metóde *GetSource* sa inštancia pozrie, či text nie je vygenerovaný. Ak áno, tak objekt vráti text vygenerovaný v predchádzajúcom volaní. Ak nie, tak objekt zavolá metódu *Run*, ktorá v tomto prípade vie, že neočakáva žiaden vstup a preto prejde rovno k načítaniu vstupného súboru, ktorý sa realizuje v metóde *Body* ktorá sa následne zavolá. Po načítaní vstupného súboru sa riadenie vráti z inštancie triety *TFileInput* opäť na inštanciu triedy *TFileOutput*. Tá prevezme aj vstupné dáta. Keďže inštancia triedy *TFileOutput* na svoju prácu nepotrebuje kód, môže sa začať vykonávať metóda *Body*. V inštancii tirety *TFileOutput* metóda *Body* vykoná zápis do súboru. Po dokončení tohto úkonu sa riadenie dodvzdá inštancii triedy *TPlocha*, ktorá dokončí prehľadávanie poľa inštancií.

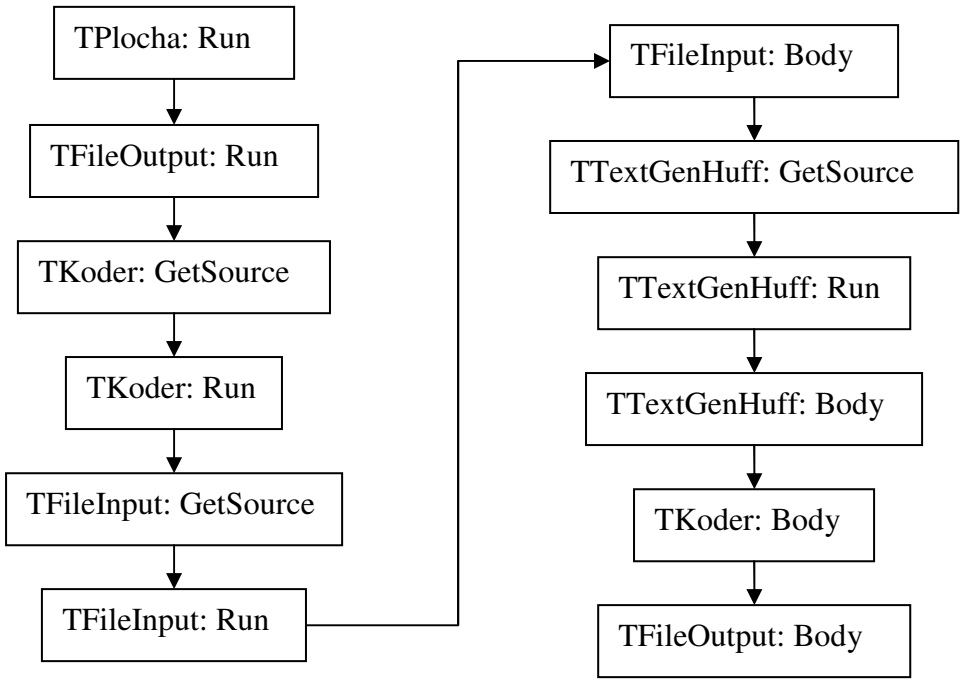


Zoberme si zložitejší systém. Chceme zakódovať text Huffmanovým kódom. Potrebujeme na to inštanciu triedy *TFileInput*, *TFileOutput*, *TKoder* a *TTextGenHuff*. Vstupný text pre inštanciu triedy *TKoder* nastavíme inštanciu triedy *TFileInput*. Ďalej pre inštanciu triedy *TKoder* musíme nastaviť aj vstupný kód a ten nám poskytne inštancia triedy *TTextGenHuff*. Inštancii triedy *TTextGenHuff* nastavíme ako vstup opäť inštanciu triedy *TFileInput*. Výsledok inštancie *Tkoder* uložíme do súboru pomocou inštancie triedy *TFileOutput*.



Po spustení sa bude systém správať nasledovne. Zavolá sa metóda *Run* inštancie triedy *TPlocha.* Táto metóda prehľadá pole inštancií triedy *TZariadenie*. Pri prehľadávaní tohto poľa zavolá metódu *Run* tej inštancie triedy, ktorá nevracia žiaden výstupný text ani kód. V našom prípade je to inštancia triedy *TFileOutput*. Skontroluje sa, či inštancia triedy nastavená ako zdroj textu na uloženie naozaj vracia text a zavolá sa metóda *GetSource*

inštancie triedy *TKoder*. Ak nebol generovaný výstup z inštancie triedy *TKoder* (v tomto prípade nebol), tak sa zavolá jej metóda *Run*, ktorá overí, či inštancia triedy *TFileInput* vracia text. Zavolá sa metóda *GetSource* triedy TFileInput. Tá zavolá svoju metódu *Run* kde sa overí, či bol už súbor načítaný. V tomto prípade súbor ešte nebol načítavaný a preto sa zavolá metóda *Body* ktorá ho načíta a pošle na výstup. Riadenie prevezme inštancie triedy *TKoder*, ktorá skontroluje, či inštancia triedy *TTextGenHuff* generuje kód. Zavolá metódu *GetCode* inštancie triedy *TTextGenHuff*. Metóda skontroluje, či už kód nebol vygenerovaný, keďže nebol, zavolá svoju metódu *Run*. Metóda *Run* zistí, či trieda *TFileInput* vracia text a zavolá metódu *GetSource* inštancie triedy *TFileInput*. Metóda *GetSource* skontroluje, či už bol súbor načítaný. V tomto prípade už súbor načítaný bol, keď si ho vyžiadala inštancia triedy *TKoder*, preto sa nezavolá metóda *Run* inštancie triedy *TFileInput*, ale sa na výstup odovzdá výstup z predchádzajúceho volania *TFileInput* a riadenie sa vráti inštancii triedy *TTextGenHuff*. Inštancia triedy zavolá metódu *Body* a vygeneruje optimálny Huffmanov kód pre text vo vstupnom súbore. Tento kód pošle na výstup a riadenie sa vráti inštancii triedy *TKoder*. Inštancia triedy *TKoder* zavolá svoju metódu *Body*, ktorá prekóduje text na vstupe podľa kódovacej tabuľky, ktorú dostane od inštancie triedy *TTextGenHuff*. Inštancia triedy *TKoder* vygeneruje text v Huffmanovom optimálnom kóde pre vstupný text, pošle ho na výstup a vráti riadenie inštancii triedy *TFileOutput*. Inštancia triedy *TFileOutput* zavolá svoju metódy *Body*. Tá vykoná zápis do výstupného súboru. Riadenie sa odovzdá inštancii triedy *TPlocha* a tá ho vráti aplikácii.



2.3.4 Popis triedy TPlocha

V aplikácii potrebujeme objekt, ktorý spravuje inštancie potomkov triedy *TZariadenie*. Na tento účel sme zaviedli novú triedu *TPlocha*. Úlohou jej inštancie je spravovať inštancie triedy *TZariadenie*. Pod správou inštancií triedy *TZariadenie* rozumieme nasledovné činnosti: vytváranie inštancií, udržiavanie poľa inštancií, kontrola nastavení (test na cyklus), vizualizáciu inštancií na obrazovke, rušenie inštancií, nastavovanie vlastností inštancií, spúšťanie systému.

Popíšeme si verejné rozhranie tejto triedy spolu s jej funkčnosťou.Verejný atribút *OnZmenaZariadenia* slúži na uloženie mena metódy, ktorá sa má vykonať v prípade zmeny označeného zariadenia (ak užívateľ klikne na grafickú reprezentáciu zariadenia). Očakávaná funkčnosť v metóde je reakcia aplikácie na zmenu aktuálneho zariadenia (zobrazenie vlastností aktuálneho zariadenia v paneli nastavení vlastností). Metóda *PridajZariadenie* vytvára inštanciu zariadenia a pridáva ju do zoznamu aktuálnych zariadení. Metóda *OdoberZariadenie* odoberá zariadenie z pracovnej plochy. Metóda *NastavZariadenie* slúži ako rozhranie pre nastavenie vlastností zariadenia. Metódou *NastavVstupKod* nastavíme zariadeniu referenciu na inštanciu zariadenia od ktorého očakáva kódovaciu tabuľku. Metódou *NastavVstupText* nastavíme zariadeniu referenciu na inštanciu zariadenia od ktorého očakáva vstupný text. Metódou *Run* uvedieme celý systém do prevádzky. Metóda

Reset spustí metódu *ClearDevice* v inštanciách potomkov triedy *TZariadenie*. Metóda *AkceptujeVstupKod* s parametrom mena inštancie zistí, či daná inštancia čakáva na vstupe kódovaciu tabuľku. Metóda *AkceptujeVstupText* s parametrom mena inštancie zistí, či daná inštancia očakáva na vstupe dáta. Metóda *ZoznamDavajucichKod* s parametrom zariadenia vráti mená inštancií, ktoré očakávajú na vstupe kódovaciu tabuľku, okrem mena inštancie poslaného ako parameter. Metóda *ZoznamDavajucichText* s parametrom zariadenia vráti mená inštancií, ktoré očakávajú na vstupe dáta, okremmena inštancie poslaného ako parameter. Metóda *ZoznamZariadeni* vráti zoznam aktuálnych inštancií potomkov triedy *TZariadenie*. Metóda *ZoznamVlastnostiSA* vracia inštancií potomkov triedy *TZariadenie* v inom formáte ako metóda *ZoznamZariadeni*. Metóda *ZoznamTypovVl* s parametrami mena inštancie a menom vlastnosti vracia typ danej vlastnosti inštancie. Metóda *ZoznamAllowedVals* s parametrami mena zariadenia a menom vlastnosti vráti zoznam povolených hodnôt vlastnosti ak je vlastnosť typu zoznam povolených vlastností. Metóda *ZobrazPText* s parametrom mena inštancie vracia meno inštancie, poskytujúcej vstup pre danú inštanciu, ktorej meno je parametrom metódy. Metóda *ZobrazPKod* s parametrom mena inštancie vracia meno inštancie, poskytujúcej kódovaciu tabuľku pre danú inštanciu. Metóda *ZmenMeno* slúži na zmenu mena inštancie. Metóda *SaveToFile* uloží celú plochu s inštanciami potomkov triedy *TZariadenie* do súboru. Metóda *LoadFromFile* načíta plochu z uloženého súboru. Metódou *ZmazPlochu* vyčistíme plochu, uvoľní všetky inštancie triedy *TZariadenie*.

2.3.5 Krokovanie algoritmov

Pre jednoduchšie pochopenie teórie kódovanie sme sa rozhodli umožniť užívateľom aj vizualizovať niektoré algoritmy/kroky kódovacie (dekódovacieho) procesu. Ide hlavne o algoritmy na vytváranie kódov a samotný princíp kódovania. Naša univerzálna objektová štruktúra nám to veľmi jednoduchým spôsobom umožnila.

Na vizualizáciu algoritmu potrebujeme sledovať hodnoty premenných v čase. Vytvorili sme mechanizmus na vizualizáciu algoritmu, ktorý by nebol reprezentovaný samotným „zdrojovým kódom“, ale jeho grafickou predstavou. Tu sa nám úloha trocha skomplikovala o generovanie grafickej predstavy každého kroku algoritmu. Každý algoritmus má svoju individuálnu grafickú interpretáciu. Spoločným menovateľom je, že ide

o grafickú reprezentáciu. Z toho nám vyplynuli nasledovné požiadavky. Každá trieda musí mať implementovaný vlastný vizualizačný algoritmus. Každá trieda musí mať vlastné deliace body jednotlivých krokov (jeden deliaci bod je bod, ktorý reprezentuje stav možný vizualizácie). V praxi to znamená, že konkrétna implenemtácia daného algoritmu je v teoretickej rovine popísaná niekoľkými krokmi, z ktorých má každý krok svoju grafickú interpretáciu. V programovacom jazyku sa každý krok skladá z niekoľkých príkazov. Preto sme museli zaviesť identifikátor, ktorý nám hovorí, kedy zobraziť grafickú interpretáciu daného stavu algoritmu (v ktorom kroku v programovacom jazyku je možná vizualizávia).

Pre ľahšie pochopenie problematiky je vhodné, keď je možné algoritmus posúvať oboma smermi (dopredu aj dozadu). Ak by sme chceli vyriešiť tento problém, potrebovali by sme mať možnosť pustiť algoritmus v opačnom smere. To je v praxi pomerne komplikované, preto sme sa rozhodli o metódu odpamätávania stavov výpočtu. V každom deliacom bode si zapamätáme vizualizáciu stavu.

Samotné krokovanie prebieha priamočiaro. Spustí sa systém. Pri bode, ktorý je označený ako deliaci sa vygeneruje grafická reprezentácia stavu algoritmu, ktorá sa zobrazí na obrazovku. Užívateľ má možnosť ísť o krok dopredu (na ďalší vizualizačný/deliaci bod), vrátiť sa o krok spať, alebo pustiť beh algoritmu do konca. Keď sa užívateľ posunie o krok dopredu, zatvorí sa okno s vizualizáciou stavu a algoritmus pokračuje k ďalšiemu deliacemu bodu. Tu sa opäť vygeneruje grafická reprezentácia stavu. Ak užívateľ klikne na posun o krok spať, zobrazí sa reprezentácia, ktorá sa odpamätala v predchádzajúcom deliacom bode. Keď sa v tomto okamihu užívateľ rozhodne posunúť o krok dopredu, algoritmus nebude pokračovať, len sa zobrazí odpamätaná vizualizácia aktuálneho kroku. Algoritmus bude pokračovať pri posune dopredu až vtedy, keď sa dostaneme do aktuálneho kroku behu algoritmu (nemáme odpamätaný nasledujúcu viualizáciu).

V praxi implementácia nášho „debugera“ spočívala v zadefinovaní metódy *Step* s parametrom bitmapa. Po zavolaní tejto metódy sa bitmapa z parametra zapamätá a zobrazí sa krokovací dialóg. Krokovací dialóg sa ďalej stará o zobrazovanie predchádzajúcich stavov a stavov, ktoré už boli zobrazené a púšťanie výpočtu do ďalšieho deliaceho bodu.

Ďalej si popíšeme, ako rozšíriť aplikáciu o nové zariadenie (nového potomka triedy *TZariadenie*).

2.4 Metodika rozširovania funkčnosti aplikácie

V tejto kapitole si popíšeme pravidlá, ako rozširovať aplikáciu o ďalšie zariadenia. Nevyhnutným predpokladom je kompilátor pre Delphi 6 a zdrojový kód aplikácie.

Aby sme jednoducho vedeli pridávať nové zariadenia od rôznych zdrojov (spolužiakov – programátorov), je potrebné vytvoriť novú unit-u. Telo unit-y doplníme na nasledovný tvar

```
unit Rozsirenie;

interface

uses Zariadenie;

type
  TMojeZariadenie = class (TZariadenie)
  end;

implementation

end.
```

Aby sme zariadenie (novú triedu) mohli zapájať do systému, potrebujeme upraviť ešte unit-u *masinka.pas* nasledovne. Za klauzulu *uses* na začiatku súboru pridáme názov našej novej unit-y

```
unit Masinka;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Plocha, Zariadenie, Triedy, StdCtrls, ExtCtrls, Buttons,
  ComCtrls, Menus, Grids, ValEdit, Rozsirenie;
```

a do procedúry *NoveZariadenia* pridáme nasledovné volanie metódy *NoveZariadenie*

```
procedure NoveZariadenia;
begin
  NoveZariadenie(TMojeZariadenie, 'Moje');
end;
```

kde prvý parameter je nami vytvorená trieda a druhý je meno, ktoré sa zobrazí v zozname zariadení po spustení aplikácie.

Vytvorili sme nové zariadenie, ktoré nevieme použiť, nedá sa zapojiť do systému. Potrebujeme mu vytvoriť metódy *Inicializacia* a *Body*. Deklarácia triedy bude nasledovná

```
TMojeZariadenie = class (TZariadenie)
  procedure Body(mode: integer); override;
  procedure Inicializacia; override;
end;
```

Zostáva nám implementovať nové metódy. Metóda *Inicializacia* slúži na zadefinovanie vlastností/nastavení zariadenia a nastavenie mena. Vlastnosti pridávame volaním metódy *AddProperty*, meno nastavíme priradením reťazca premennej *Name*. Metódami *SetReturnCode, SetReturnSource, SetNeedCode* a *SetNeedSource* systému povieme, či zariadenie akceptuje na vstupe dáta či kódovaciu tabuľka a na výstup dáva dáta alebo kódovaciu tabuľku. V metóde *Body* je implementovaná funkčnosť/algoritmus. Vstupné dáta v metóde *Body* získame zavolaním metódy *GetPSource*, kódovaciu tabuľku získame metódou *GetPCode*. Dáta na výstup pošleme metódou *SetPSource* a kódovaciu tabuľku pošleme na výstup metódou *SetPCode*.

Ako príklad si popíšme rozšírenie o zariadenie implementujúce Cézarovej šifry (nebudeme sa obmedzovať len na bežnú abecedu, ale budeme šifrovať celú tabuľku znakov). Zariadenie bude mať vlastnosť „heslo“ reprezentované celým číslom, bude na vstupe očakávať dáta a na výstup bude posielať dáta. Deklarácia triedy bude nasledovná

```
TCaesarCipher = class (TZariadenie)
  procedure Body(mode: integer); override;
  procedure Inicializacia; override;
end;
```

Metóda *Inicializacia* bude nasledovná

```
procedure TCaesarCipher.Inicializacia;
begin
  AddProperty('Heslo', '3', ptNumber);
  SetReturnSource;
  SetNeedSource;
  Name := 'CaesarCipher';
end;
```

Takto implementujeme metódu *Body*

```
procedure TCaesarCipher.Body(mode: integer);
var
  heslo: integer;
  vstup: TText;
  i: integer;
begin
  heslo := StrToInt(GetProperty('Heslo'));
  vstup := GetPSource;
  for i := Low(vstup) to High(vstup) do
    vstup[i] := Char((Ord(vstup[i]) + heslo) mod 256);
  SetPSource(vstup);
end;
```

Najprv zistíme hodnotu parametra „heslo“, potom načítame vstup do premennej *vstup*. Zašifrujeme vstupný reťazec a pošleme ho na výstup.

Modifikujeme procedúru *NoveZariadenia*, skompilujeme aplikáciu a máme hotové rozšírenie o Cézarovu šifru. Implementovali sme len kóder pre Cézarovu šifru. Aby bola aplikácia úplná, je potrebné podobným spôsobom vytvoriť zariadenie na dekódovanie Cézarovej šifry. Skúsenejší programátor môže modifikovať kóder tak, aby fungoval korektne aj ako dekóder. Implementáciu dekódera ponechávame na čitateľa ako malé cvičenie.

Popísali sme si rozširovanie aplikácie. Teraz má čitateľ v rukách dobrý nástroj, pomocou ktorého vie jednoducho experimentovať s rôznymi druhmi transformácii (nemusí ísť nutne o kódovacie transformácie).

3. Záver

Našim cieľom bolo vytvoriť aplikáciu na podporu výučby teórie kódovania. Aplikácia mala byť ľahko rozšíriteľná o rôzne typy kódov a mala umožňovať vizualizáciu vybraných algoritmov kódovania a dekódovania. Aplikácia mala umožňovať experimentovanie s kódmi.

Tieto ciele sa nám podarilo naplniť, vytvorili sme aplikáciu založenú na elektornickej skladačke, ktorá pozostáva s viacerých komponentov a umožňuje skúmať praktické vplyvy použitia kódovacích algoritmov v rôznych podmienkach a s rôznymi vstupmi. Teda používateľ (študent) môže vyskladať kódovací systém pozostávajúci s jednotlivých algoritmov kódovania, transformácii, algoritmov samoopravných kódov, štatistických generátorov kódu a kóderov. Stavebnicový systém umožňuje sekvenčnú aplikáciu jednotlivých algoritmov na vstupný text. Aplikácia vizualizuje vybrané algoritmy, ktoré je možné krokovať a sledovať ich myšlienku.

Modul na generovanie príkladov na skúšku sa nám z časových dôvodov už nepodarilo implementovať. Naša aplikácia však môže slúžiť ako pomôcka pri generovaní príkladov na skúšku, no nie je v nej možné generovať výstupy použiteľné ako zadania na skúške.

V aplikácii sú implementované nasledovné algoritmy, ktoré sa nachádzajú aj v elektronickej učebnici:

- Generovanie tabuľky Huffmanovho kódu podľa frekvenčnej analýzy
- Generovanie tabuľky Fanovho kódu podľa frekvenčnej analýzy
- Kódovanie podľa kódovacej tabuľky
- Dekódovanie podľa kódovacej tabuľky
- Kódovanie pomocou algoritmu LZ77
- Dekódovanie pomocou algoritmu LZ77
- Kódovanie pomocou algoritmu LZW
- Dekódovanie pomocou algoritmu LZW
- Vykonávanie transformácie BWT
- Vykonávanie transformácie MTF
- Kódovanie samoopravných obdĺžnikových kódov

- Dekódovanie samoopravných obdĺžnikových kódov
- Kódovanie samoopravného Hammingovho $(15,11)$ kódu
- Dekódovanie samoopravného Hammingovho $(15,11)$ kódu
- Kódovanie pomocou BCH kódov
- Dekódovanie pomocou BCH kódov

V aplikácii sme neimplementovali algoritmy na kódovanie Markovského zdroja a algoritmy na rozšírenie kódu.

Aplikácia umožňuje vizualizáciu nasledovných algoritmov:

- Generovanie tabuľky Huffmanovho kódu podľa frekvenčnej analýzy
- Generovanie tabuľky Fanovho kódu podľa frekvenčnej analýzy
- Kódovanie podľa kódovacej tabuľky
- Dekódovanie podľa kódovacej tabuľky
- Kódovanie pomocou algoritmu LZ77
- Kódovanie pomocou algoritmu LZW
- Kódovanie samoopravných obdĺžnikových kódov
- Dekódovanie samoopravných obdĺžnikových kódov
- Kódovanie samoopravného Hammingovho $(15,11)$ kódu
- Dekódovanie samoopravného Hammingovho $(15,11)$ kódu

Súčasťou našej aplikácie je framework, pomocou ktorého je možné jednoducho rozšíriť funkčnosť aplikácie. Zabezpečuje vytváranie a prepájanie (komunikáciu) počítajúcich komponentov a tiež vizualizáciu implementovaných algoritmov alebo procesov. Pomocou tohto frameworku je možné aplikáciu rozšíriť o matematický modul, ktorý umožní vizualizáciu matematických funkcií. Pomocou tohto modulu by bolo možné napríklad vizualizovať derivovanie funkcie, alebo rôzne fyzikálne transformácie. Aplikácia by mohla byť tiež rozšírená o kryptografické funkcie, keďže kryptológia je ďalšou z prednášok na našej fakulte.

4. Summary

The aim of this work was to create an application to support teaching of coding theory. The application was expected to be easily expandable on different types of codes and to enable visualization of chosen algorithms of coding and decoding as well.

The targets were met; we created the application based on an electronic puzzle, a multicomponent one, which enables us to inspect a real consequences of the use of coding algorithms in various conditions and on different levels.

The following algorithms are implemented:

- Formation of the Huffman Code Table by frequency analysis
- Formation of the Fan Code Table by frequency analysis
- Coding by the Coding Table
- Decoding by the Coding Table
- Coding by LZ77 algorithm
- Decoding by LZ77 algorithm
- Coding by LZW algorithm
- Decoding by LZW algorithm
- Running BWT transformation
- Running MTF transformation
- Coding of error-correcting rectangular codes
- Decoding of error-correcting rectangular codes
- Coding of error-correcting Hamming (15,11) code
- Decoding of error-correcting Hamming (15,11) code
- Coding by BCH codes
- Decoding by BCH codes

Visualization of following algorithms is granted by the application:

- Formation of the Huffman Code Table by frequency analysis
- Formation Fan Code Table by frequency analysis
- Coding by Coding Table

- Decoding by Coding Table
- Coding by LZW algorithm
- Coding by LZW algorithm
- Coding of error-correcting rectangular codes
- De-Coding of error-correcting rectangular codes
- Coding of error-correcting Hamming (15,11) code
- De-Coding of error-correcting Hamming (15,11) code

It is possible to use the framework of the application for formation and connection of any components and visualization of any algorithms. Another extension of the application might be a mathematical module which will provide visualization of mathematical functions. Using this module it might be possible to visualize the function derivation, or various physical transformations. The application could be enlarged on cryptographic functions since cryptology is one of further lectures at the Faculty of Mathematics, Physics and Informatics.

5. Zoznam použitej literatúry

HPOCROFT, J. E. ULLMAN, J. D 1978: *Formálne jazyky a automaty*. Alfa SNTL, Bratislava, Praha

JOYNER, W. D. 2001: Basic procedures for Cyclic, Hamming, Binary Reed-Muller, BCH, and Golay codes. United States Naval Academy, Division of Engineering and Weapons, Annapolis

LELEWER, D. A. HIRSCHBERG, D. S. 2005: Data Compression. University of California, Irvine

LIDL, R. 1994: *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge

OLEJÁR, D. STANEK, M. 2002: *Úvod do teórie kódovania*. Fakulta Matemitiky, Fyziky a Informatiky Univerzity Komenského, Bratislava

RAFFO, D. 2001: Codage et Cryptographie: Constructeur de Codes BCH. Maîtrise Informatique, Université de Marne la Vallée

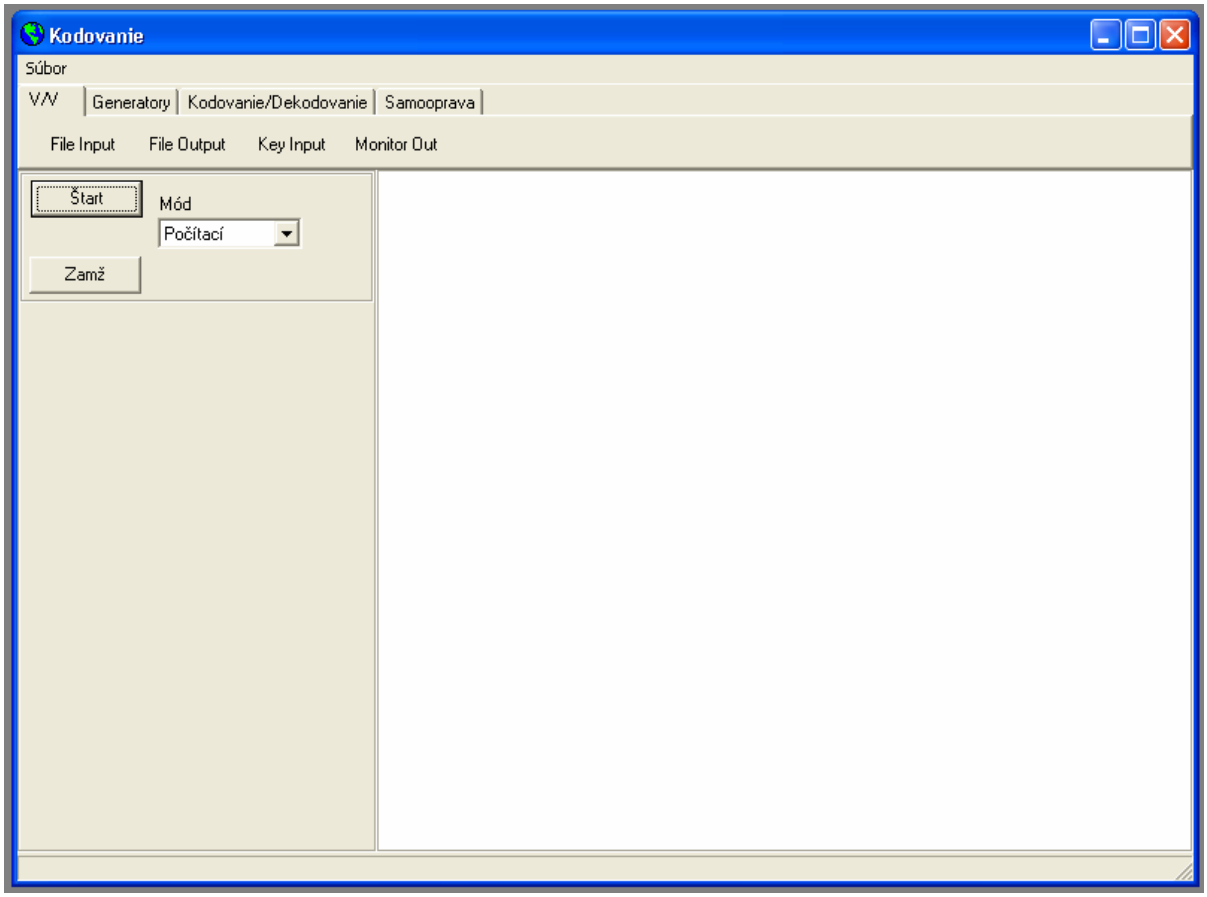
WALLIS, J. L. HOUGHTEN, S. K. 2002: A Comparative Study of Search Techniques Applied to the Minimum Distance Problem of BCH Codes. Brock University, Department of Computer Science St. Catharines, Ontario

6. Prílohy

6.1 Návod na použitie aplikácie

Nutným predpokladom na spustenie aplikácie je počítač s operačným systémom Windows 98 a novším. Aplikáciu nie je potrebné inštalovať, pozostáva z jedného exe súboru, ktorý jednoducho spustíme.

Po spustení aplikácie sa nám zobrazí nasledovné

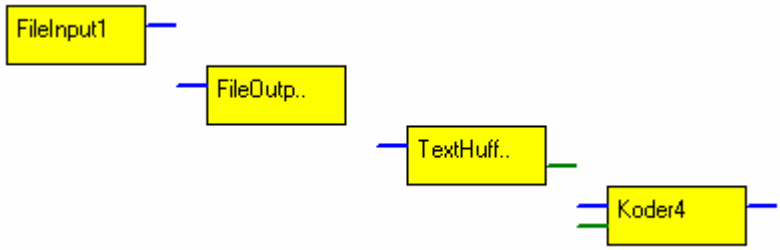


Obrazovka aplikácie má v hornej menu. Menu obsahuje položky s povelmi na prácu z aplikáciou a to povel na ukončenie aplikácie, povel na uloženie práce, povel na načítanie práce a povel na začatie novej práce (zmazanie pracovnej plochy).

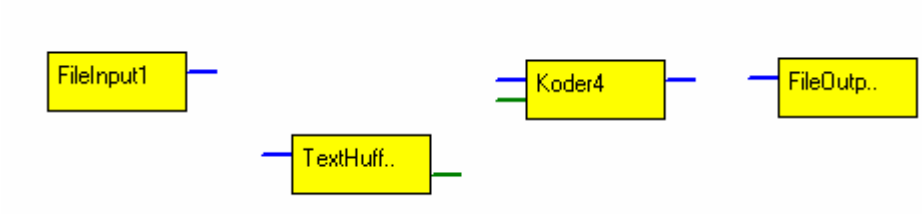
Nižšie sa nachádza panel so záložkami. V každej záložke sa nachádzajú tlačidlá na pridávanie zariadení do pracovnej plochy. Pod panelom so záložkami je aplikácia rozdelená na dve časti. V pravej časti sa nachádza pracovná plocha a v ľavej časti je panel na

nastavovanie vlastností zariadení. V hornej časti panelu vlastností sa nachádza štartovacie tlačidlo – „Štart“ (uvádza systém do chodu) a tlačidlo na odobratie zariadenia z pracovnej plochy – „Zmaž“. Tiež sa tu nachádza prepínač režimu behu systému. Systém je spustiteľný v dvoch režimoch a to v počítacom a vizualizačnom.

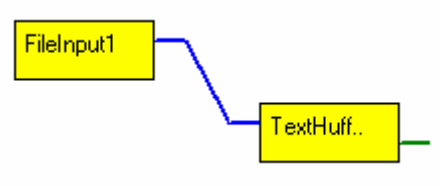
Ďalej si popíšeme postup vytvárania systému zariadení. Potrebujeme vytvoriť systém, ktorý načíta vstup zo súboru, na základe tohto vstupu (jeho frekvenčnej analýzy) vytvorí kódovaciu tabuľku pre Huffmanov kód, zakóduje vstupný text a uloží do súboru. Na plochu potrebujeme vložiť zariadenie na čítanie súboru, zapisovanie do súboru, generátor Huffmanovho kódu a kóder. Najprv na plochu vložíme zariadenie *File Input* zo záložky V/V (zariadenie vložíme kliknutím na jeho názov v záložke), potom pridáme zariadenie *File Output*. Zo záložky Generátory vložíme na plochu zariadenie *Huff txt*, ktoré generuje kódovaciu tabuľku pre Huffmanov kód a nakniec pridáme do plochy *Koder* z tretej záložky. Na ploche máme



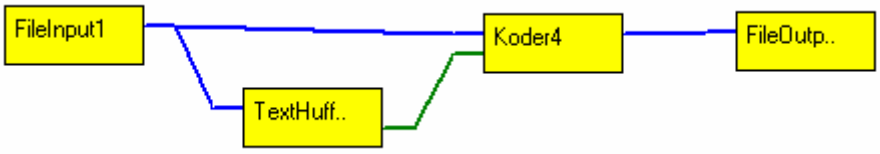
Zariadenia preusporiadame tak, že na jednotlivé zariadenie klikneme myšou ľavým tlačidlom myši a potiahneme (tlačidlo držíme stlačené), presunieme zariadenie na požadované miesto a tlačidlo myši uvoľníme. Usporiadajme si plochu nasledovne



Po kliknutí na zariadenie sa v paneli vlastností zobrazia jeho vlastnosti. Zariadenia môžeme nastavovať v ľubovoľnom poradí, poďme zľava. Klikneme na prvé zariadenie a nastavme meno vstupného súboru. Klikneme na ďalšie zariadenie a nastavme mu vlastnosť *text* na *FileInput1*. Týmto zariadeniu povieme, že vstupné dáta má čerpať zo zariadenia pre načítanie zo vstupu. Toto nastavenie sa na ploche prejaví spojením týchto dvoch zariadení modrým pruhom.

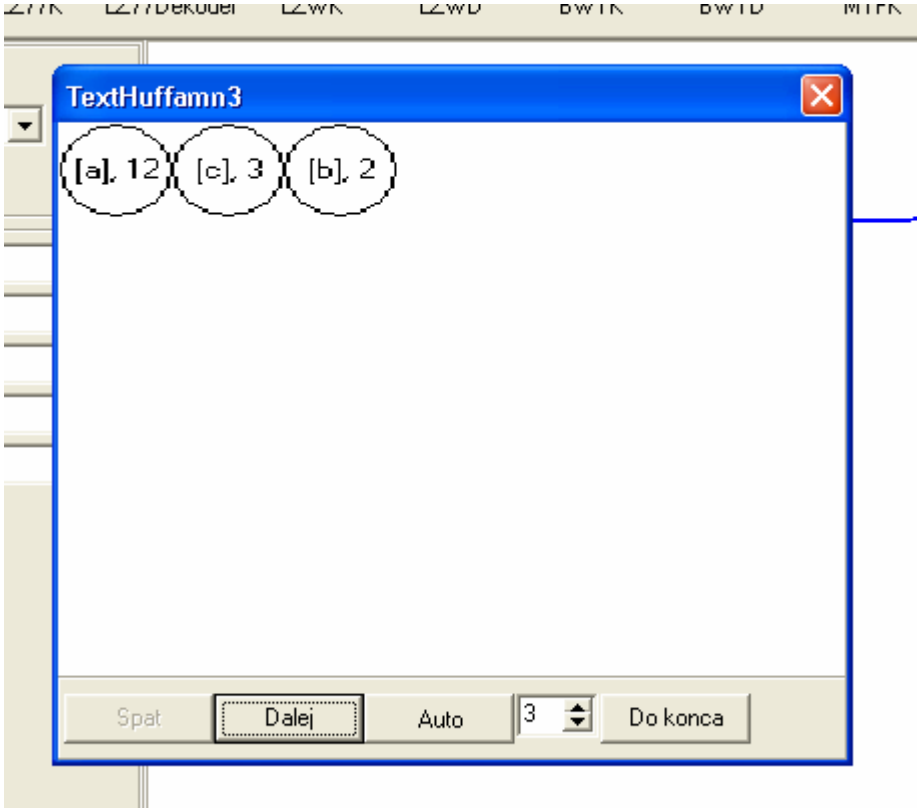


Pokračujme ďalej a nastavme kóderu vlastnosť *text* na *FileInput1* a vlastnosť kód na *TextHuffman3*. Poslednému zariadeniu nastavme vlastnosť *text* na *Koder4*. Vizuálne sa nastavenia prejavia nasledovne



Po kliknutí na tlačidlo štart uvedieme celý systém do chodu. Po skončení behu aplikácia vypíše čas behu systému. Vo výstupnom súbore môžeme pozorovať výsledok.

Ak prepneme mód na vizualizačný (v hornej časti panelu vlastností), aplikácia pri generovaní kódovacej tabuľky pre Huffmanov kód zobrazí krokovací dialóg.

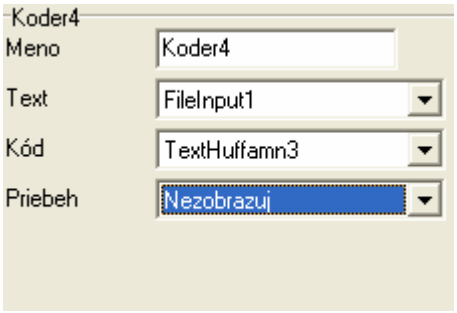


V ňom je možné pozorovať a krokovať vytváranie kódovacej tabuľky pre Huffmanov kód. Kliknutím na tlačidlo „Dalej“ sa zobrazí ďalší krok/stav algoritmu. K predchádzajúcemu kroku sa môžeme vrátiť kliknutím na tlačidlo „Spat“. Tlačidlo „Auto“ slúži na automatické krokovanie (algoritmus sa po čase sám posunie o krok dopredu – čas je definovaný

v sekundách hneď za tlačidlom „Auto“). Keď klikneme na tlačidlo „Do konca“, algoritmus pre dané zariadenie dobehne do konca bez ďalšej vizualizácie. Pozor, krokovanie algoritmu je pomerne náročné na pamäť. Pri dlhších a zložitejších algoritmoch môže mať aplikácia problém s nedostatkom pamäte.

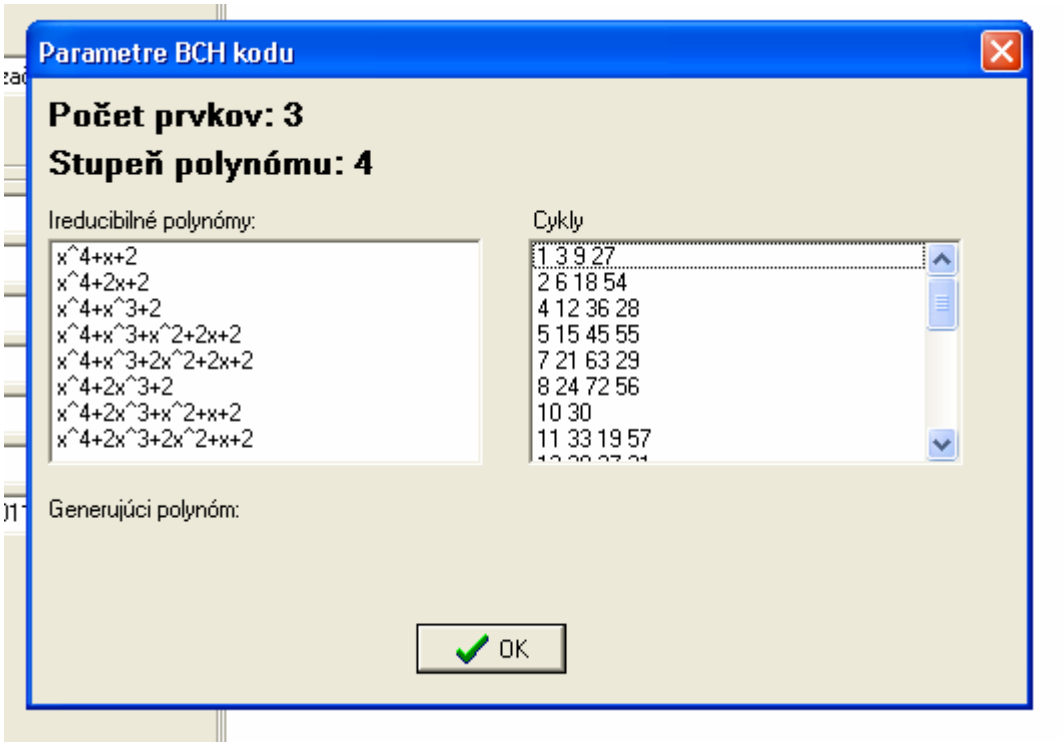
Keď dobehne generovanie kódovacej tabuľky, aplikácia začne zobrazovať kódovanie (zobrazenie prepisu vstupu do binárneho zápisu). Po dokončení tohto zariadenia systém skončí a vo výstupnom súbore môžeme pozorovať výsledok kódovania.

Ak považujeme krokovanie/vizualizáciu niektorých zariadení za nepotrebnú v danej situácii, môžeme ju vypnúť a to prepnutím vlastnosti zariadenia *Priebeh* na hodnotu *Nezobrazuj*. Ak túto vlastnosť nastavíme na kódery, tak aplikácia bude krokovať/vizualizovať len generovanie kódovacej tabuľky.

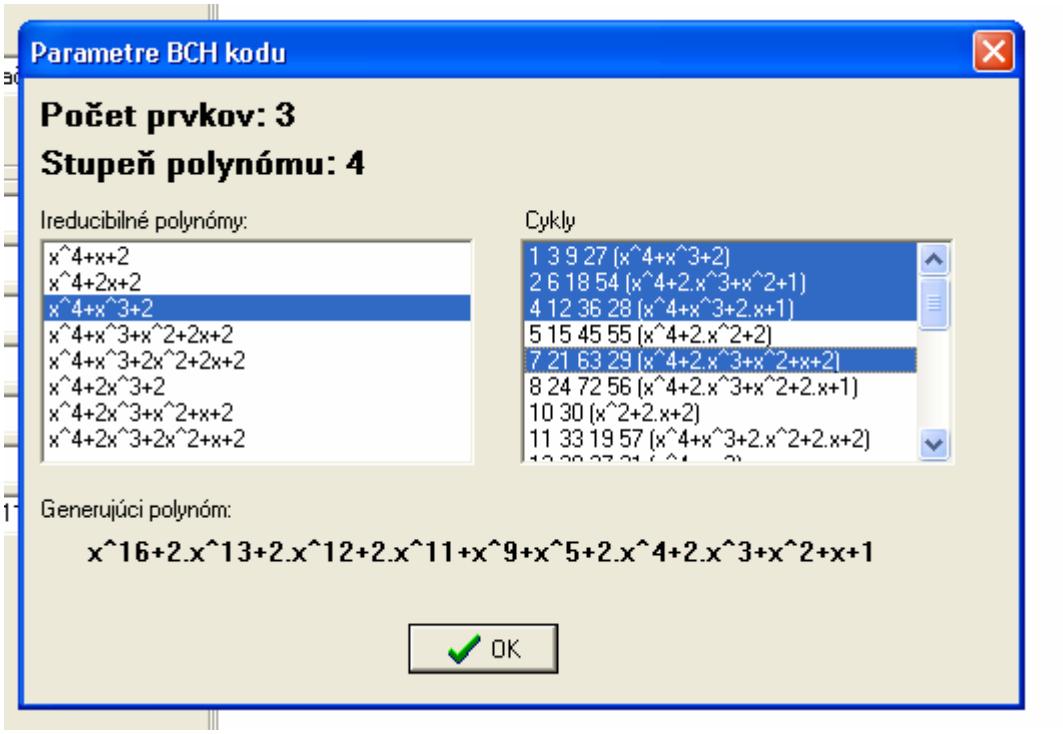


Jednoduchým spôsobom si vieme poskladať aj komplikovanejší systém. Zostavenie systému na dekódovanie Huffmanovho kódu prenechávame na čitateľa. Ďalej si popíšeme nastavovanie vlastností pre BCH kódy, nakoľko je o niečo komplikovanejšie.

Zariadenie pre kódovanie do BCH kódov má nastavenia *q*, *m*, *Polynom*. Parametre *q* a *m* a ich prípustné hodnoty sú popísané v teoretickej časti práce. Pri kliknutí na vlastnosť *Polynom* sa zobrazí nasledovný dialóg



V ľavej časti je zoznam ireducibilných polynómov. Myšou označíme potrebný polynóm. Po dvojkliku na polynóm aplikácia doplní polynómy zodpovedajúce jednotlivým cyklom zobrazeným v pravej časti. Kliknutím na cyklus ho označíme a aplikácia vygeneruje generujúci polynóm a zobrazí ho v dolnej časti. Podľa potreby môžeme označiť viac cyklov (podľa požiadavky na opravnú schopnosť kódu). Aplikácia pregeneruje generujúci polynóm a zobrazí ho v dolnej časti.



Po kliknutí na tlačidlo „Ok“ sa polynóm nastaví do vlastnosti *Polynom*.

Poskladané systémy zariadení aj s ich aktuálnym nastavením vlastností je možné uložiť do súboru a neskôr znovu načítať a pracovať ďalej. Povely na tieto kroky sa nachádzajú v menu aplikácie. Aplikáciu ukončíme kliknutím na krížik v pravom hornom rohu, alebo povelom „Koniec“ z menu aplikácie.

6.2 Definícia triedy TZariadenie

```
TZariadenie = class
private
   kod: TKod;
   text: TText;
   RetCode: boolean;
   RetSource: boolean;
   NeeCode: boolean;
   NeeSource: boolean;
   ZarProperty: TProperties;
   generated: boolean
   allowch: boolean;
   steps: array of TBitmap;
   dstav: TStav;
   function PropertyIndex(PropName: string): integer;
protected
   procedure Inicializacia; virtual; abstract;
   procedure Body(mode: integer); virtual; abstract;
   procedure AddProperty(PropName: string; PropValue: string; PropType: TPropertyType =
ptSingle); overload;
   procedure AddProperty(PropName: string; PropValue: string; AllowedVal: TStringArray);
overload;
   procedure AddProperty(PropName: string; PropValue: string; Abeceda: string); overload;
   procedure SetReturnCode;
   procedure SetReturnSource;
   procedure SetNeedCode;
   procedure SetNeedSource;
   procedure SetPCode(code: TKod);
   procedure SetPSource(source: TText);
   procedure Step(mode: integer; bmp:TBitmap);
   function GetPCode: TKod;
   function GetPSource: TText;
   function SetPropertyAllowedVal(Propname: string; AllowedVal: TStringArray): boolean;
   function isCorrectVal(Propname: string; Propval: string): boolean;
   function Bin2Char(txt: TText): TText;
   function Char2Bin(txt: TText): TText;
public
   PGetSource: TZariadenie;
   PGetCode: TZariadenie;
   Poradie: integer;
   Name: string;
   constructor Create; virtual;
   procedure Run(mode: integer);
   procedure SetName(Meno: string);
   function GetSource(mode: integer): TText;
   function GetCode(mode: integer): TKod;
   function ReturnCode: boolean;
   function ReturnSource: boolean;
   function NeedCode: boolean;
   function NeedSource: boolean;
   function GetProperties: TProperties;
   function SetProperty(Propname: string; PropValue: string): boolean;
   function GetProperty(PropName: string): string;
   function GetProptyType(PropName: string): TPropertyType;
   function GetProptyAllowedVal(PropName: string): TStringArray;
   function SaveDevice: TDevice; virtual;
   function LoadDevice(device: TDevice): boolean; virtual;
   procedure ClearDevice;
end;
```

6.3 Definícia triedy TPlocha

```
TPlocha = class
private
   Zariadenia: array of TZariadenie;
   Visual: array of TDesign;
   _Owner: TWinControl;
   _Canvas: TCanvas;
   OldX, OldY: integer;
   Active: integer;
   function jeVolneMeno;
   procedure ZmazCiary;
   procedure NakresliCiary;
   procedure KresliSource(farba: TColor);
   procedure KresliCode(farba: TColor);
   procedure OnImgMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
   procedure OnImgMouseUp(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
   procedure OnImgMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
protected
   function testCyklu(i: integer): boolean;
public
   OnZmenaZariadenia: TZmenaZariadenia;
   constructor Create(Owner: TWinControl; Canvas: TCanvas);
   procedure PridajZariadenie(trieda: CZariadenie);
   procedure OdoberZariadenie(meno: string);
   procedure NastavZariadenie(meno: string; vlastnost: string; hodnota: string);
   procedure NastavVstupKod(meno: string; vstup: string);
   procedure NastavVstupText(meno: string; vstup: string);
   procedure Run(mode: integer);
   procedure Reset;
   function AkceptujeVstupKod(meno: string): boolean;
   function AkceptujeVstupText(meno: string): boolean;
   function ZoznamDavajucichKod(meno: string): TStrings;
   function ZoznamDavajucichText(meno: string): TStrings;
   function ZoznamZariadeni: TStrings;
   function ZoznamZariadeniSA: TStringArray;
   function ZoznamVlastnosti(meno: string): TStrings;
   function ZoznamVlastnostiSA(meno: string): TStringArray;
   function ZoznamTypovVl(meno: string; zarproperty: string): TPropertyType;
   function ZoznamAllowedVals(meno: string; zarproperty: string): TStringArray;
   function ZobrazNastavenie(meno: string; vlastnost: string): string;
   function ZobrazPText(meno: string): string;
   function ZobrazPKod(meno: string): string;
   procedure ZmenMeno(Stare: string; Nove: string);
   procedure SaveToFile(meno: string);
   procedure LoadFromFile(meno: string);
   procedure ZmazPlochu;
end;
```