Katedra počítačovej grafiky
Fakulta matematiky, fyziky a informatiky
Univerzita Komenského

Diplomová práca

Peter Hudec

BRATISLAVA 2003

Katedra počítačovej grafiky
Fakulta matematiky, fyziky a informatiky
Univerzita Komenského

Interaktívna výuka kompresie dát

Autor: Peter Hudec
Vedúci diplomovej práce: doc. Ing Jaroslav Polec, PhD.

Bratislava, Apríl 2003

Čestne prehlasujem, že som diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry a iných zdrojov.

.....

Obzvlášť ďakujem svojmu diplomovému vedúcemu diplomovej práce doc. Ing. Jaroslavovi Polecovi, PhD za všestrannú pomoc, cenné rady a pripomienky pri vypracovaní diplomovej práce.

Obsah

Zoznam tabuliek

Zoznam obrázkov

Kapitola 1

Úvod

1.1 Význam kompresie dát

V dnešnej dobe hovoríme o digitálnych systémoch, sieťach, digitálnej podobe filmov, hudby, obrázkov alebo hlasu. Prečo digitálne? Digitálne signály sa dajú ľahko uložiť alebo prenášať na dlhé vzdialenosti bez kumulovania chýb.

Samozrejme, má to aj svoje nevýhody. Digitálna verzia signálov (hlas, hudba, film) potrebuje viac bitov na sekundu pri ukladaní alebo transmisii. V *tabuľke 1.1* sú zhrnuté potrebné dátové toky pre niektoré signály v digitálnej podobe. Napríklad na bežný CD-ROM nosič (700 MB) uložíme iba 23,5 sekúnd nekomprimovaného videa. Na DVD-5 je to už okolo troch minút. Metóda, ktorá rieši tento problém, sa volá **kompresia**.

Kompresia dát je vlastne efektívna reprezentácia zdroja. Presnejšie povedané, kompresia dát je reprezentácia zdroja v digitálnej podobe s čo najmenším možným počtom bitov. Toto sa dá dosiahnuť dvoma spôsobmi: stratová kompresia (lossy) znamená, že zdrojový signál nie je z dát 100% rekonštruovateľný a bezstratová kompresia (lossless) znamená, že sme schopní rekonštruovať zdrojový signál do jeho presnej podoby.

1.2 Použitie

S algoritmami tohto typu sa stretávame v bežnom živote denne bez toho, aby sme si to vôbec uvedomovali.

Kľúčovou aplikáciou, kde je nutná bezstratová kompresia, je kompresia dátových súborov na disk počítača. Dekompresiou komprimovaného súboru musí vzniknúť súbor, ktorý

typ	parametre	dátová náročnosť
telefón (200–3400 Hz)	8000 Hz x 12 bits/sec	96 kbps
hovorené slovo (50–7000Hz)	16,000 Hz x 14 bits/sec	224 kbps
audio (20–20.000 Hz)	44,100 Hz x 2 kanály x 16 bit/sec	1,412 Mbps
obrázok	512x512 bodov x 24 bit/bod	6.3 Mbit/obrázok
video	640x480 bodov x 24 bit/bod x 30 snímkov/sec	221 Mbps
HDTV	1280x720 bodov x 24 bit/bod x 60 snímkov/sec	1,3 Gbps

Tabuľka 1.1: Približné dátové toky pre nekomprimované dáta

je identický s pôvodným súborom pred kompresiou. Mnohé z nich majú implementované rôzne varianty Lempel-Zivových kódov; ako príklad sa dá uviesť Unixový Commpres, Stacker, gzip, ... Modemy, ktoré pracujú na prenosových rýchlostiach 14, 400 baudov a vyššie, používajú Lempel-Ziv kompresiu vo svojej štandardnej konfigurácii. Adaptívny Huffmanov kód je súčasťou Unix Compact. Modifikované Huffmanove kódy sú súčasťou medzinárodných štandardov pre Group 3 fax, Group 4 fax, JPEG a MPEG.

1.3 Cieľ práce

V súčasnosti je pre verejnosť dostupné veľké množstvo kníh alebo iných dokumentácií zaoberajúcich sa touto problematikou. Pri písaní tejto práce som bol nútený sa niektorými týmito publikáciami zaoberať. Spomenieme niektoré ich vlastnosti.

kniha - v niektorých knihách sú algoritmy vysvetlené podrobne do detailov, nie je tam len stručný popis algoritmu napísaný v pseudo kóde. Medzi tieto knihy môžme zaradiť [3] a [4]. V tomto prípade, bohužiaľ, nemáme možnosť interaktívne komunikovať s knihou a sme teda odkázaní iba na to, čo je v nej napísané. Pokiaľ nám nie je z výkladu jasné, ako by sa daný algoritmus správal na iných vstupoch okrem tých, ktoré sú v knihe uvedené ako príklady, nemáme žiadnu možnosť si to overiť. Toto je jedna z veľkých nevýhod kníh. Druhá je relatívne ťažká dostupnosť kvalitných kníh.

online dokumentácia - do veľkej miery ju môžme porovnať s knihou. Väčšina z nich sa venuje konkrétne jednému algoritmu, takže tam chýba akýsi úvod do problematiky, či

vysvetlené základné pojmy, ktoré sa tam používajú. V niektorých dokumentáciách sa nachádzali k algoritmu aj dynamické prvky, ktoré vysvetľovali fungovanie algoritmu, čos a dá pokladať za ich výhodu.

Cieľom tejto práce je vysvetliť princípy bezstratových kompresných algoritmov a zaviesť príslušnú terminológiu. Využíva pritom dobré vlastnosti kníh aj online dokumentácií. Ako výuková pomôcka nám bude pritom slúžiť *interaktívna učebnica*, kde je obsiahnutý úvod do problematiky, vysvetlené základné pojmy a podrobný popis algoritmov vysvetlený na príkladoch. Na interaktivitu s užívateľom slúžia aplety, kde máme možnosť zadať vlastný vstup a následne ho komprimovať, prípadne dekomprimovať. Ako dodatok sú popísané niektoré grafické formáty, ktoré dané algoritmy používajú.

Kapitola 2

Základné pojmy a vlastnosti

V tejto kapitole za oboznámime s vlastnosťami, ktoré musia bezstratové kompresné algoritmy spĺňať a vysvetlíme si pojmy, ktoré budeme v ďalšom texte používať.

2.1 Abeceda

Zdroj je reprezentovaný *zdrojovou abecedou* S.

$$S = \{s_1, s_2, s_3, \ldots, s_m\}, \tag{2.1}$$

kde s_i, $i = 1, 2, \ldots, m$, sú symboly zdrojovej abecedy. Ďalej existuje kódová abeceda A.

$$A = \{a_1, a_2, a_3, \ldots, a_n\}, \tag{2.2}$$

kde a_i, $i = 1, 2, \ldots, n$, sú symboly kódovej abecedy, teda symboly, ktoré sú výstupom kompresného algoritmu. V najjednoduchšom prípade sú tieto dve abecedy zhodné. Kódovanie je potom vlastne proces priradenia *kódového slova* zdrojovému *symbolu*.

$$s_i \rightarrow \{a_{i,1}, a_{i,2}, a_{i,3}, \ldots, a_{i,k}\} \tag{2.3}$$

Pri vytváraní tabuľky, ktorá určuje toto priradenie, aby bol proces čo najefektívnejší, nám pomáha aspoň približná znalosť frekvenčného výskytu jednotlivých znakov $p(x)$ (distribučná funkcia). Samozrejme musí platiť $\sum p(a_i) = 1$. V ďalšom texte budeme $p(a_i)$ značiť skráteným zápisom p_i.

V 19. storočí vznikol asi najznámejší takýto kód. Vytvoril ho F. B. Morse pre telegrafiu priradením kratších kódových reťazcov častejšie sa vyskytujúcim písmenám (znakom)

Zdrojový symbol	Početnosť	Kódové slovo
a_1	0.40	000
a_2	0.15	001
a_3	0.15	010
a_4	0.10	011
a_5	0.10	100
a_6	0.05	101
a_7	0.04	110
a_8	0.01	111

Tabuľka 2.1: **Kód A**: Štandardný binárny kód

Zdrojový symbol	Početnosť	Kódové slovo
a_1	0.40	0
a_2	0.15	1
a_3	0.15	00
a_4	0.10	01
a_5	0.10	10
a_6	0.05	11
a_7	0.04	000
a_8	0.01	001

Tabuľka 2.2: **Kód B**: Štandardný binárny kód

a dlhšie písmenám s menšou pravdepodobnosťou výskytu. Morseova abeceda mapuje 26 znakov anglickej abecedy do štvorprvkovej abecedy {*dot, dash, mark, space*}.

2.2 VLC kódy

Na ilustráciu si zoberme kód, ktorého zdrojová abeceda pozostáva z ôsmich znakov (a_1, a_2 , a_3, a_4, a_5, a_6, a_7, a_8). Priraďme jednotlivým znakom pravdepodobnosť ich výskytu: $p_1 = 0.4$, $p_2 = p_3 = 0.15$, $p_4 = p_5 = 0.1$, $p_6 = 0.05$, $p_7 = 0.04$, $p_8 = 0.01$. Samozrejme, je tu splnená podmienka $\sum_{i=1}^{8} p_i = 1$.

Tabuľky 2.1 až 2.4 znázorňujú jednotlivé kódy. V kóde, ktorý sa nachádza v tabuľke

Zdrojový symbol	Početnosť	Kódové slovo
a_1	0.40	010
a_2	0.15	011
a_3	0.15	00
a_4	0.10	100
a_5	0.10	101
a_6	0.05	110
a_7	0.04	1110
a_8	0.01	1111

Tabuľka 2.3: **Kód C**: Nedodržuje Morseov princíp

Zdrojový symbol	Početnosť	Kódové slovo
a_1	0.40	0
a_2	0.15	011
a_3	0.15	1010
a_4	0.10	1011
a_5	0.10	10000
a_6	0.05	10001
a_7	0.04	10010
a_8	0.01	10011

Tabuľka 2.4: **Kód D**: Jednoznačne dekódovateľný kód, ktorý nie je prefixový

2.1, je každému zdrojovému symbolu priradené kódové slovo s rovnakou dĺžkou. Takémuto kódu hovoríme *kód s pevnou dĺžkou slova* (fixed-lenght code), alebo tiež *blokový kód* (block code). Kód v tabuľke 2.2 sa nazýva *kód s premenlivou dĺžkou* (variable-length code, VL kód), lebo priradené kódové slová majú rôznu dĺžku. Kódy v tabuľkách 2.3 a 2.4 sú tiež VL-kódy (blokový kód je špeciálnym prípadom VL-kódu, nie naopak).

Označme l_j dĺžku binárneho kódového slova priradenému znaku a_j, potom očakávaný počet bitov na jeden zdrojový znak je

$$\bar{l} = \sum_{j=1}^{M} p_j l_j \tag{2.4}$$

Kód A (strana 5, tabuľka 2.1) je štandardná binárna reprezentácia ôsmich rôznych znakov. Tento kód, tzv. brute force, sa tiež nazýva *nekomprimovaná* binárna reprezentácia a používa sa ako kritérium (meradlo), podľa ktorého sa meria kompresný pomer. Ako sa dá očakávať, $\bar{l}_A = 3$.

Keď si zoberieme kód B (strana 5, tabuľka 2.2), ktorý spĺňa ideu Morseovho kódu, dostaneme $\bar{l}_B = (0.4 + 0.15) * 1 + (0.15 + .01 + .01 + 0.05) * 2 + (0.04 + 0.01) * 3 = 1.5$, čo znamená kompresný pomer 2:1 oproti brute-force kódu. Podobne vieme vypočítať aj $\bar{l}_C = 2.9$, $\bar{l}_D = 2.85$. **Kód B** má najlepší kompresný pomer, teda je aj najvhodnejším kandidátom, ale pre naše potreby je nepoužiteľný. Prečo, to sa dozvieme v ďalšej sekcii.

Ešte si zadefinujeme pojem *entropia*. Majme zdrojovú abecedu S, ktorá obsahuje m symbolov: $s_i;\ i = 1, 2, \ldots, m$. Pre každý symbol máme definovanú jeho početnosť výskytu: $p_i;\ i = 1, 2, \ldots, m$. Informačná hodnota symbolu je definovaná ako $I_i = -\log_2 p_i$ bitu. Entropia je potom priemerná informačná hodnota na jeden symbol a je definovaná nasledovne:

$$H = \sum_{i=1}^{m} p_i * \log_2 p_i \qquad \text{bitu.} \tag{2.5}$$

Táto hodnota, na rozdiel od hodnoty $\bar{l}$, je závislá iba od samotnej zdrojovej abecedy a informácii, ktorá sa má komprimovať. Entropia je akási spodná hranica pre VLC kódy, ktorú sa snažia dosiahnuť.

2.3 Jednoznačná dekódovateľnosť

Hovoríme, že kód je jednoznačne dekódovateľný, ak nie je viacznačne dekódovateľný. Inak povedané, existuje iba jedna možnosť, ako daný reťazec pozostávajúci z kódových symbolov

zdrojový symbol	kódové slovo
s_1	00
s_2	01
s_3	00
s_4	10

Tabuľka 2.5: Nejednoznačne dekódovateľný kód

dekódovať. Ak má byť kód použiteľný, tak by mal byť jednoznačne dekódovateľný.

Keď sa pozrieme na kód v tabuľke 2.5, zistíme, že nie je jednoznačne dekódovateľný. Pokiaľ máme dekódovať sekvenciu 00, tak nevieme presne určiť k nej prislúchajúci zdrojový znak, teda znak, z ktorého vznikla (s_1, s_3).

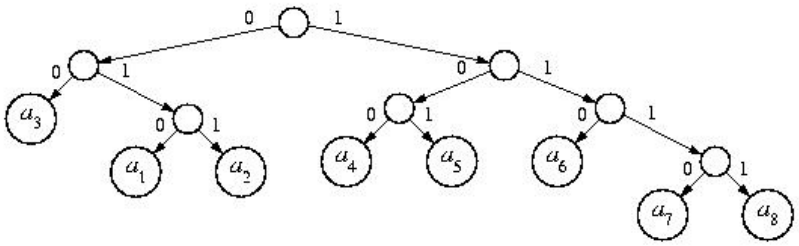
Vráťme sa ku kódu **B**, ktorý sme si zadefinovali na strane 5. Predpokladajme, že sme kódovali sekvenciu a_3a_7 (symbol a_3 nasledovaný symbolom a_7). Kód B, podľa tabuľky 2.2, zakóduje túto dvojicu ako 00 a 000, teda dostaneme sekvenciu 00000. Dekóder by v tomto momente nebol schopný jednoznačne určiť, akým spôsobom táto sekvencia vznikla (a_3a_7, a_7a_3, $a_3a_1a_3$, ...). Teda tento kód nespĺňa podmienku jednoznačnej dekódovateľnosti a preto je nepoužiteľný.

Kód A (strana 5, tabuľka 2.1) je jednoznačne dekódovateľný. Všetky kódové slová majú dĺžku tri, teda pri dekódovaní vieme, že nové kódové slovo začína každé tri symboly. Jeho hlavnou nevýhodou je veľké $\bar{l}$.

Kód C (strana 6, tabuľka 2.3) je tiež jednoznačne dekódovateľný, pretože spĺňa *podmienku prefixovosti*, tj. žiadne kratšie kódové slovo nie je prefixom dlhšieho. Kódy spĺňajúce túto podmienku nazývame *prefixové kódy*. Každý prefixový kód sa dá zobraziť ako binárny strom, ktorého listy predstavujú zdorjové symboly. Binárny strom, ktorý zobrazuje kód C, je na obrázku 2.1. Kódové slovo priradené danému zdrojovému symbolu je potom cesta po ohodnotených hranách v strome, ktorá vedie od koreňa stromu k danému zdrojovému symbolu. Pri dekódovaní kódového slova algoritmus sleduje cestu od koreňa stromu k jeho listom. Po dosiahnutí listu je k nemu prislúchajúci zdrojový symbol dekódovaný. Napríklad majme reťazec

$$000100110100100010101010001010100001110011011110101111.$$

Tento reťazec sa dá jednoducho rozdeliť na kódové slová



Obr. 2.1: Kód C v binárnom strome

$$00, 010, 011, 010, 010, 00, 101, 010, 100, 010, 101, 00, 00, 1110, 011, 011, 110, 101, 1111.$$

Tieto sa dajú preložiť do symbolov zdrojovej abecedy

$$a_3, a_1, a_2, a_1, a_1, a_3, a_5, a_1, a_4, a_1, a_5, a_3, a_3, a_7, a_2, a_2, a_6, a_5, a_8.$$

Platí tvrdenie, že každý prefixový kód je jednoznačne dekódovateľný, ale neplatí tvrdenie opačné. Existujú kódy, ktoré nie sú prefixové, ale sú jednoznačne dekódovateľné. Ako príklad si môžeme zobrať *kód D* (strana 6, tabuľka 2.4). Prefixovosť je porušená kódovým slovom 0 (prislúcha a_1), ktorý je prefixom 011 (prislúcha a_2). Všimnime si, že zvyšných šesť symbolov začína 1 a žiadny nie je prefixom iného. Naviac, žiadny z týchto šiestich kódových slov nezačína prefixom 11, takže sa nikdy nepomýlime na konci kódového slova pre a_2. Keď pri dekódovaní nového kódového slova je prvý sybmol 0, pozrieme sa ešte na ďalšie dva kódové symboly. Ak ich hodnota je 11, potom všetky tri symboly tvoria kódové slovo a_2, inak je prvý symbol kódové slovo a_1 a nasledujúci symbol je začiatkom ďalšieho kódového slova. Táto vlastnosť je aj rozhodujúca pre tento kód, aby bol jednoznačne dekódovateľný.

Keď si zoberieme reťazec z predchádzajúceho príkladu a pokúsime sa ho dekódovať naším kódom D, dostaneme nasledujúce sekvencie

$$0, 0, 0, 10011, 0, 10010, 0, 0, 1010, 1010, 0, 0, 1010, 10000, 11100 \ldots$$

Toto je príklad, keď pri dekódovaní narazíme na skupinu kódových znakov, ktoré nevieme dekódovať z dôvodu, že dané kódové slovo v tabuľke neexituje. V našom prípade je to kódové slovo začínajúce troma po sebe idúcimi 1. Tu môžeme predpokladať, že niekde pri kompresii, dekompresii alebo transmisii nastala chyba.

Kapitola 3

Bezstratová kompresia

V tejto kapitole sa budeme postupne venovať rôznym algoritmom bezstratovej kompresie dát.

Pri bezstratovej kompresii, ako už bolo povedané, sme schopní presne rekonštruovať pôvodné dáta na základe ich skomprimovanej verzie. Takéto algoritmy sa nachádzajú v rôznych softvérových alebo hardvérových zariadeniach. Obvyklé použitie je v archivátoroch súborov (zip, rar, gzip, bzip, commpress), vo väčšine modemov, ... Mnohé z nich sú používané v medzinárodných štandardoch, definujúcich protokoly alebo spôsob uloženia dát (Huffman - Group3 fax, Group4 fax, JPEG, MPEG).

3.1 Huffmanov kód

Kód D (tabuľka 2.4, strana 6)má najmenšie $\bar{l}_d$ spomedzi všetkých uvedených. Na druhej strane, malá zmena v kóde C (tabuľka 2.3, strana 6) spôsobí, že jeho entropia klesne ešte pod túto hranicu. Keď vymeníme kódové slová pre symboly a_1 a a_3, entropia pre kód C klesne o 0.25, teda $\bar{l}_c = 2.65$. Tento nový kód je tiež jednoznačne dekódovateľný.

Týmto postupom môžeme entropiu postupne znižovať a tým daný kód stále zlepšovať. *D.A. Huffman* zostrojil algoritmus (Huffman, 1952) na tvorbu optimálneho kódu s minimálnym $\bar{l}$. Algoritmus a kód, ktorý sa vytvára, boli nazvané podľa ich autora.

3.1.1 Vytváranie optimálneho kódu

Majme teda zdrojovú abecedu:

$$S = \{s_1, s_2, s_3, \ldots, s_m\}. \tag{3.1}$$

Ďalej, bez ujmy na všeobecnosti môžeme predpokladať, že početnosti výskytov jednotlivých zdrojových symbolov sú

$$p(s_1) \geq p(s_2) \geq \cdots \geq p(s_{m-1}) \geq p(s_m). \tag{3.2}$$

Keďže hľadáme optimálny kód pre S, musia dĺžky kódových slov pre jednotlivé znaky spĺňať podmienku

$$l_1 \leq l_2 \leq \cdots \leq l_{m-1} \leq l_m. \tag{3.3}$$

Na základe týchto požiadaviek na optimálny kód, Huffman odvodil nasledujúce pravidlá:

- $l_1 \leq l_2 \leq \cdots \leq l_{m-1} = l_m$. Táto rovnica a 3.2 hovoria, že keď sú zdrojové symboly usporiadané zostupne podľa ich početnosti, potom dĺžky ich kódových slov by mali byť v neklesajúcom poradí. Inak povedané, dĺžky kódových slov častejšie sa vyskytujúcich zdrojových symbolov nesmú byť dlhšie ako menej vyskytujúcich sa symbolov. Kódové slová priradené posledným dvom zdrojovým symbolom (s najmenšou početnosťou) majú rovnakú dĺžku.
- Zdrojové symboly s najmenšou početnosťou majú im prislúchajúce kódové slová líšiace sa len v poslednom bite.
- Každá možná sekvencia dĺžky $l_m - 1$ musí byť buď kódové slovo, alebo jeden z jej prefixov musí byť kódové slovo.

To, ako zabezpečiť tieto podmienky, je napísane v samotnej učebnici. Samotným algoritmom sa budeme zaoberať v nasledujúcej časti.

3.1.2 Algoritmus

Na základe týchto troch pravidiel vieme, že najmenej vyskytujúce sa zdrojové znaky majú rovnakú dĺžku kódových slov, ktoré im boli priradené. Tieto kódové slová sa líšia iba v poslednom bite (teda v 0 a 1). Preto tieto dva zdrojové symboly môžu byť spojené do nového

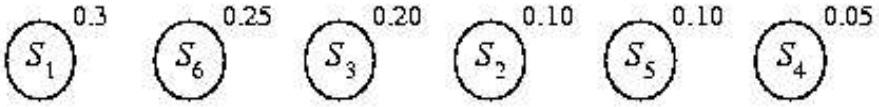
Zdroj. symbol	Početnosť	Kódové slovo	Dĺžka
S_1	0.3	00	2
S_2	0.1	101	3
S_3	0.2	11	2
S_4	0.05	1001	4
S_5	0.1	1000	4
S_6	0.25	01	2

Tabuľka 3.1: Príklad na Huffmanov kód

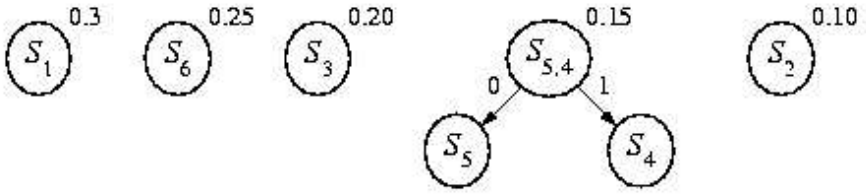
symbolu, ktorého početnosť bude $p(s_{m-1}) + p(s_m)$. Novovzniknutá abeceda má o jeden symbol menej ako pôvodná abeceda. Túto abecedu preusporiadame, aby sme znova splnili podmienku nerastúcich početností. Podobný postup môžeme aplikovať na túto abecedu a posledným dvom symbolom priradiť hodnoty 0 a 1 ktoré, spojíme do nového symbolu. Algoritmus sa zastaví v momente, keď spojíme posledné dva symboly a vznikne nám abeceda s jedným symbolom. Tomuto symbolu (pokiaľ sa nikde nevyskytla chyba) sme priradili početnosť 1.

Na ukážku si zoberme kód, ktorý je uvedený v tabuľke 3.1. Obrázky 3.1 až 3.6 ilustrujú postupné vytváranie Huffmanovho stromu. Sú na nich zobrazené len modifikované zdrojové abecedy, kde už sú jednotlivé symboly usporiadané podľa ich početnosti. Celý postup sa dá zhrnúť do nasledujúcich krokov.

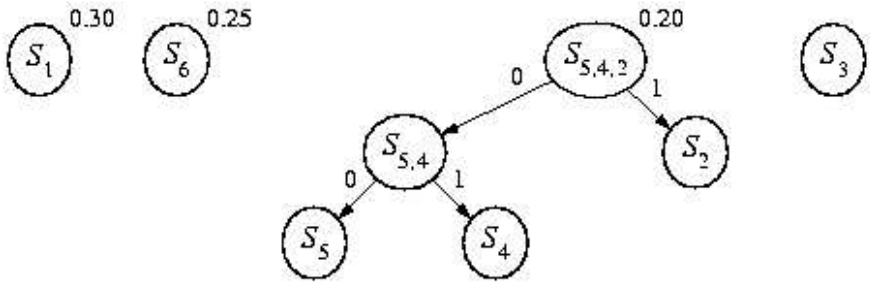
1. zostupne zoraď symboly zdrojovej abecedy podľa ich početnosti
2. spoj posledné dva symboly
   - vytvor z nich nový symbol, ktorého početnosť bude ich súčtom.
   - priraď im binárne 0 a 1.
3. tento postup opakuj, pokiaľ nevznikne abeceda s jedným symbolom
4. traverzovaním vzniknutého stromu priraď jednotlivým symbolom ich kódové slová.



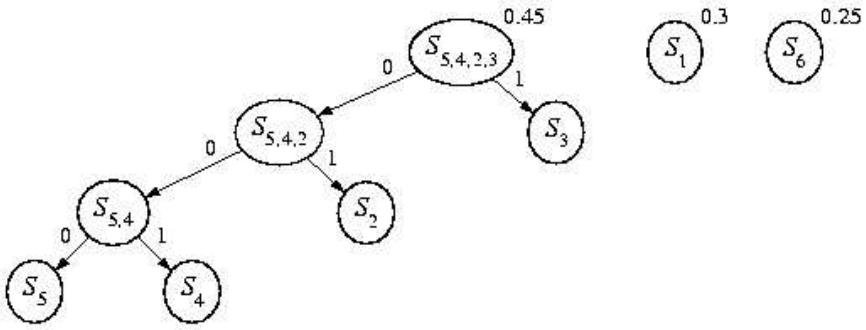
Obr. 3.1: Krok Huffmanovho algoritmu 1



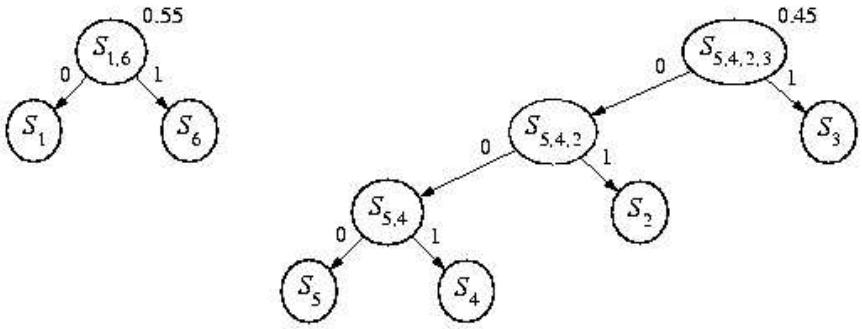
Obr. 3.2: Krok Huffmanovho algoritmu 2



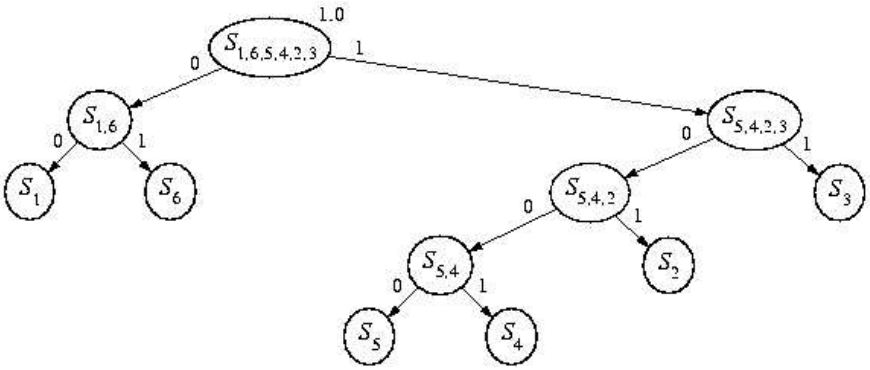
Obr. 3.3: Krok Huffmanovho algoritmu 3



Obr. 3.4: Krok Huffmanovho algoritmu 4



Obr. 3.5: Krok Huffmanovho algoritmu 5



Obr. 3.6: Krok Huffmanovho algoritmu 6

Zdroj. symbol	q_1	q_2	q_1q_1	q_1q_2	q_2q_1	q_2q_2
Početnosť	$\frac{1}{8}$	$\frac{7}{8}$	$\frac{1}{64}$	$\frac{7}{64}$	$\frac{7}{64}$	$\frac{49}{64}$
Huffman. kód	0	1	000	001	01	1

Tabuľka 3.2: Modifikácia Huffmanovho kódu

3.1.3 Modifikácie

Pokiaľ máme zdroj, ktorý má iba binárnu abecedu (prvá časť tabuľky 3.2), vytvorením Huffmanovho kódu nič neušetríme. Na druhej strane nám entropia hovorí o možnosti kompresie, pretože jej hodnota je $H = 0.554$. Tu sa ponúka možnosť kódovať *dibity*. Huffmanov kód pre dibity je v druhej časti tabuľky 3.2 a výsledná bitová náročnosť je potom 0.680 bitu na znak.

Ďalšími významnými modifikáciami sú Huffmanové kódy pre vopred neznáme postupnosti, kedy sa Huffmanova tabuľka mení adaptívne podľa kódovaných dát. Tieto kódy sú najdôležitejšie pri prenose dát v reálnom čase bez degradácie [3], [4].

3.1.4 Záverečný komentár

Vytváranie Huffmanovho kódu nie je jednoznačné. Nejednoznačnosť nastáva v prípadoch, keď dva symboly počas výpočtu majú rovnakú početnosť. Tu môže ľahko nastať zámena týchto dvoch symbolov pri vytváraní Huffmanovho stromu. Tento problém sa rieši vzájomnou dohodou (na triedenie sa použije stabilný algoritmus).

3.2 Shannonov - Fanov kód

Tento kód patrí tiež do rodiny VL kódov. Používa podobný princíp tvorby efektívneho kódu ako Huffmanov algoritmus, ale má o niečo menšiu účinnosť. Je ho výhodou je väčšia rýchlosť a ľajšia implementovateľnosť.

3.2.1 Algoritmus

Postupom kódovania pomocou kódu Shannon-Fanon sa nebudeme podrobne zaoberať, pretože je pomerne jednoduchý a dá sa pochopiť aj bez ukážky na príklade. Postup je nasledujúci:

- usporiadajme zdrojové znaky podľa ich početnosti výskytu
- rozdeľme ich na dve podskupiny s približne rovnakou relatívnou početnosťou. Hornej podskupine priradíme znak 0, dolnej 1.
- znaky z každej podskupiny rozdelíme analogicky ako v bode 2

- postup opakujeme, pokým v jednotlivých skupinách nezostane jeden symbol sám

3.3 Run-Length kódovanie (RLC)

Veľa dokumentov, listov, obrázkov sa dá prenášať použitím faxu. Pri tomto prenose sú dokumenty kvantizované do binárnej úrovne (biela a čierna farba). Rozlíšenie týchto dokumentov býva často pomerne veľké a na každom riadku dokumentu sa za sebou vyskytuje veľké množstvo bodov s rovnakou intenzitou (v tomto prípade buď biele alebo čierne). RLC kódy sa používajú práve na komprimovanie takýchto dokumentov.

3.3.1 1-D Run-Length kódovanie

V tejto verzii RLC kódu sa berie každý riadok ako nezávislý, teda v úvahu sa berie iba horizontálna koherencia. Ide teda o kódovanie postupnosti bielych a čiernych znakov (núl a jednotiek), kde sa kódujú vzdialenosti prechodov postupností. Predpokladá sa, že začiatok postupnosti sú biele znaky, ale táto dohoda nie je nutná (v najhoršom prípade bude dekódovaný dokument inverzný k pôvodnému). Ak je prvý bod čierny, dĺžka bielej sekvencie je 0. Proces komprimácie asi najlepšie vystihuje stavový diagram na obrázku 3.7. Na koniec každého riadku sa pridáva ešte jeden špeciálny symbol, EOL (end-of-line).

Na ukážku si zoberme postupnosť

$$0\,0\,0\,1\,1\,1\,1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,1\,1\,1\,1. \tag{3.4}$$

Výsledok kódovania je potom

$$3\,5\,1\,2\,6\,5\,1\,7. \tag{3.5}$$

Vo veľkej miere záleží na počte bitov, ktoré sme ochotní poskytnúť na kódovanie vzdialeností. Ak poskytneme 3 bity, potom výstupná sekvencia bude

$$011\ 101\ 001\ 010\ 110\ 101\ 001\ 111, \tag{3.6}$$

čo je 24 bitov oproti pôvodným 30 bitom. Ak ale budeme na vzdialenosti počítať 4 bity, vznikne nasledujúca postupnosť

$$0011\ 0101\ 0001\ 0010\ 0110\ 0101\ 0001\ 0111 \tag{3.7}$$

čo už je 32 bitov oproti tridsiatim z pôvodného zdroja.

Teda významne záleží na tom, akým spôsobom si zvolíme maximálnu dĺžku sekvencie znakov, aby kódovanie bolo optimálne. Existuje niekoľko spôsobov:

- štatisticky prehľadať kódované dáta a zistiť počty, ktoré sú najčastejšie
- metóda pokusov a omylov. Kódujeme pre rôzne počty.
- výsledok kódovania nerobíme binárne, ale využijeme niektorý z VLC (Huffman alebo jeho modifikácia).

Existujú rôzne varianty tohto kódu, ktoré už nepracujú na bitovej úrovni, ale na bajtovej, prípadne inej. Keď máme viac ako dve úrovne, už nám nestačí kódovať iba počet znakov, ktoré sa opakujú, ale aj samotné znaky. Napríklad to môže vyzerať následovne: $AAAAAAAAAAAAAA \rightarrow 15A$, $AAAAAAbbbXXXXXt \rightarrow 6A3b5X1t$. Náš stavový diagram na obrázku 3.7 sa potom mierne zmení. Budeme si pamätať aj kódovaný znak($RunValue$) a pred zapísaním hodnoty $RunCount$ zapíšeme hodnotu $RunValue$.

3.3.2 Rozšírenie 1-D RLC

Zaujímavým rozšírením je kódovanie nazývané 2-D RLC. Toto kódovanie berie v úvahu, že kódovaný dokument je dvojrozmerný objekt, takže okrem horizontálnej koherencie je braná v úvahu aj vertikálna.

Pri kódovaní vzdialenosti prechodov máme na výber

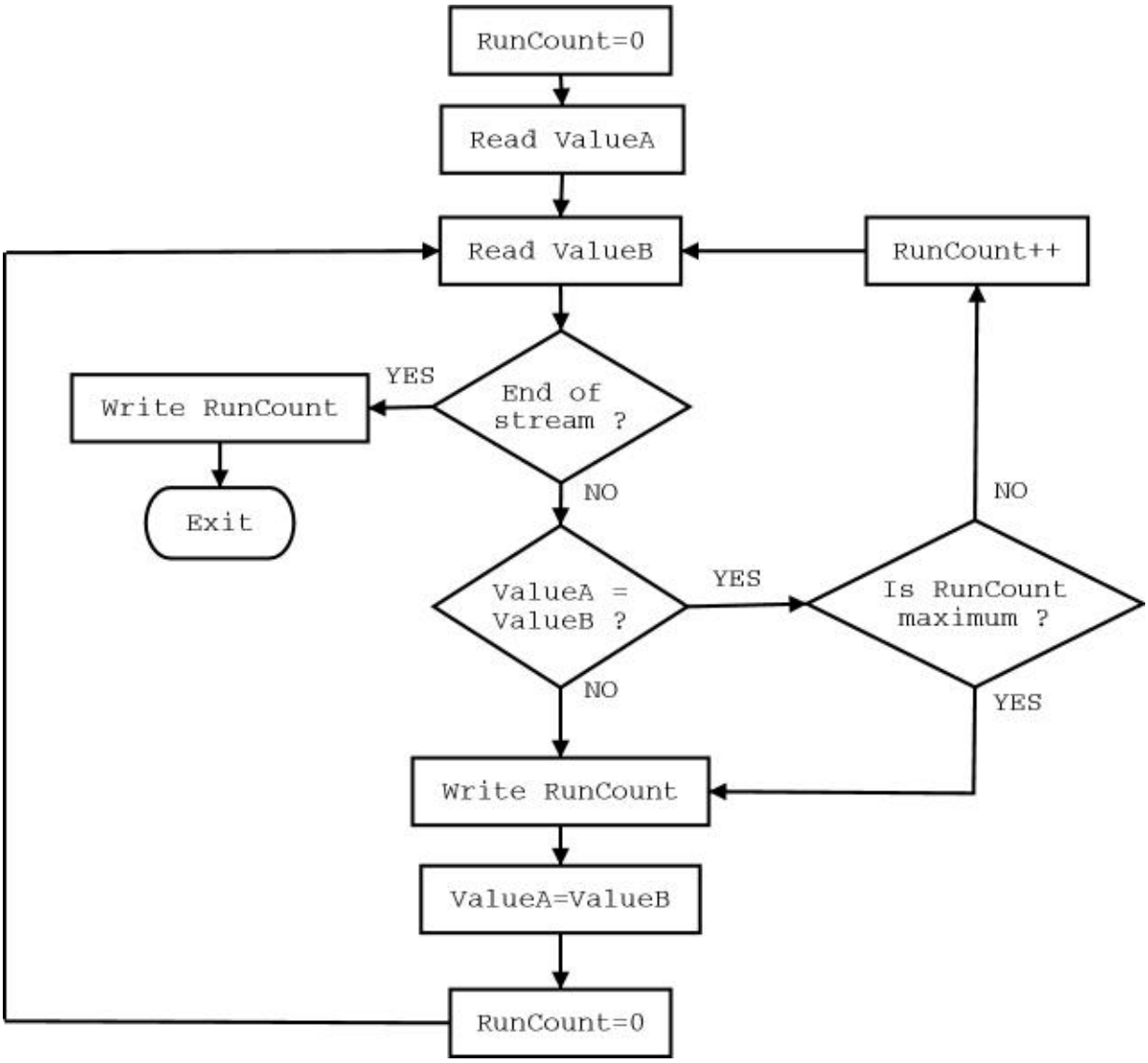
- vzdialenosť od posledného prechodu v tom istom riadku
- vzdialenosť od rovnakého prechodu v predchádzajúcom riadku smerom vľavo,
- vzdialenosť od rovnakého prechodu v predchádzajúcom riadku smerov vpravo.

Vyberá sa najkratšia z týchto vzdialeností. Pridáva sa ku nej ešte prefix, ktorý ju jednoznačne identifikuje. O tomto algoritme sa viac dočítate v [1], [3] a [4].

3.4 Slovníkové kódovanie

3.4.1 Úvod

Slovníkové kódovanie nie je založené, tak ako Huffman alebo aritmetické kódovanie, na štatistickom modeli. Tu je symbol alebo reťazec symbolov, ktoré vysiela zdroj, reprezen-



Obr. 3.7: Vývojový diagram pre RLE na bitovej úrovni

tovaný indexom v slovníku. Tento slovník bol vytvorený na základe zdrojovej abecedy. Existuje veľa príkladov slovníkov, ktoré používame v dennom živote bez toho, aby sme si to uvedomili. Napríklad slovo *September* je reprezentované v slovníku mesiacov číslom 9, a iné.

Slovník pozostáva z dvoch prvkov definovaných ako $D = (P, C)$, kde P je konečná množina fráz generovaná zo zdrojových symbolov, C je funkcia mapujúca P na množinu kódových slov. Hovoríme, že množina P je *kompletná*, ak akýkoľvek vstupný reťazec sa dá reprezentovať sériou fráz z množiny P. Funkcia C musí spĺňať podmienku prefixovosti. Pre dekompresiu ľubovoľného textu musí platiť, že množina fráz P musí byť kompletná a funkcia C spĺňať podmienku prefixovosti.

Najdôležitejšia pri tejto metóde je teda tvorba slovníka. Zle vytvorený slovník môže viesť k expanzii dát, čo je pri kompresii nevhodný jav.

Statický kódový slovník

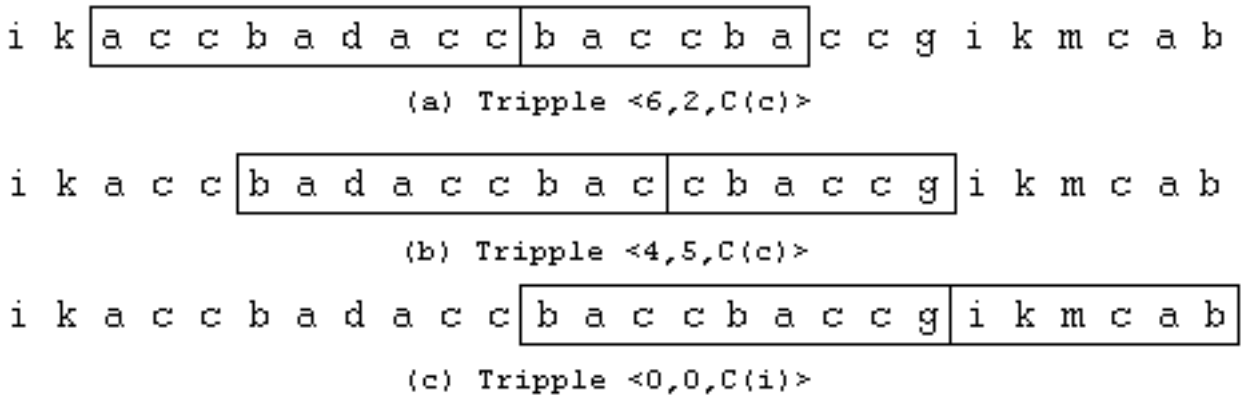
V niektorých aplikáciách, kde je známa zdrojová abeceda, početnosť výskytov znakov zdrojovej abecedy, prípadne iné vedomosti o zdroji, je vhodné vytvoriť slovník ešte pred samotným procesom kompresie alebo dekompresie. Tento slovník sa potom používa pri oboch týchto procesoch a je verejne známy. Nevýhody spočívajú v nízkej efektivite kódovania a menšej flexibilite oproti dynamickým metódam. Menšia flexibilita znamená, že slovník je vytvorený pre špeciálnu aplikáciu a nie je vhodný na všeobecné použitie.

Dynamický kódový slovník

Druhý spôsob vytvárania slovníka je dynamický. Vtedy neexistuje kompletný slovník pred procesom kompresie alebo dekompresie, ale vytvára sa počas jeho behu. Na začiatku existuje iba základný slovník, ktorý sa počas behu modifikuje. Všetky algoritmy, založené na dynamickej (adaptívnej) tvorbe slovníka, sa dajú rozdeliť do práce dvoch ľudí: Liv a Zempel (1977, 1978).

3.4.2 LZ77 - Sliding Window

V tomto algoritme sa ako slovník používa časť vstupného textu, ktorá bola nedávno zakódovaná. Text, ktorý sa kóduje sa porovnáva so symbolmi v slovníku. Najdlhšia nájdená zhoda v slovníku je charakterizovaná pointrom, ktorý pozostáva z trojice dát.



Obr. 3.8: LZ77 - kompresia

Algoritmus v LZ77 sa nazýva algoritmus posúvania okien. Okno pozostáva z dvoch častí: *search bufer* (SB) a *look-ahead bufer* (LAB). V SB je časť textu, ktorá už bola nedávno zakódovaná[1], LAB obsahuje text, ktorý ideme v nasledujúcom kroku komprimovať. Tieto dva bufre (okná) sa počas procesu komprimácie posúvajú.

Kompresia a dekompresia

Pozrime sa najprv na jeden príklad, ktorý je znázornený na obrázku 3.8. Vstupný text je *ikaccbadaccbaccbaccgikmcab*. Na obrázku má SB veľkosť 9 symbolov, LAB 6 symbolov. Pokúsime sa zakódovať symboly, ktoré sa nachádzajú v LAB. Proces začína hľadaním prvého znaku v LAB, teda b v SB odzadu. Prvú zhodu nájdeme na šiestom políčku odzadu. Ďalej zistíme, že najdlhšia zhoda, aká existuje, je dĺžky 2 (ba). Ukazovateľ je potom reprezentovaný trojicou $< i, j, k >$, kde i je vzdialenosť prvého symbolu od LAB (táto vzdialenosť sa tiež nazýva *offset*), čo je v našom prípade 6. Druhá položka, teda j, určuje dĺžku reťazca, ktorý sa zhoduje, teda 2. Tretí symbol k je znak, ktorý v LAB nasleduje hneď za zhodujúcim sa reťazcom, teda $C(c)$, kde C je funkcia mapujúca zdrojové symboly na kódové slová. Takže náš triplet po prvom kroku je $< 6, 2, C(c) >$ (obrázok 3.8a).

Dôvod, prečo sa pridáva k, je pomerne jednoduchý. V prípade, že sa v SB nenájde žiadna zhoda, potom obe i, j budú nulové. Tretia položka v tomto prípade ukazuje na prvý symbol v LAB. Keď dekóder dostane trojicu $< 0, 0, C(i) >$ vie, že nebola nájdená

[1] ako bolo povedané, to je náš slovník

žiadna zhoda a dekóduje iba symbol i. Tento prípad je znázornený na obrázku 3.8c.

Druhý krok začína tým, že sa obe okná posunú o tri symboly doprava, teda o jeden symbol viac, ako bola v predchádzajúcom kroku nájdená najdlhšia zhoda. Pri hľadaní zhodného podreťazca v SB, nájde prvú zhodu na offsete 1 s dĺžkou 1. Ďalšia zhoda je na offsete 4 s dĺžkou 5. Je zaujímavé, že maximálna zhoda môže presiahnuť hranice SB a tým zasiahnuť aj LAB. Prečo môže byť táto situácia akceptovateľná, si vysvetlíme pri dekompresii. Teda dĺžka tejto zhody je 5. Poslednú zhodu nájdeme na offsete 5 s dĺžkou 1. Keďže sa tu používa GREEDY algoritmus, je vybraná druhá zhoda s dĺžkou 5 a jej prislúchajúci triplet $< 4, 5, C(g) >$ (obrázok 3.8a).

V ďalšom kroku sa obe okná posunú o 6 miest doprava. Tento prípad je znázornený na obrázku 3.8c. Tu nie je nájdená žiadna zhoda pre symbol i, takže výsledný triplet je $< 0, 0, C(i) >$.

Komprimačný proces potom pokračuje podobne ďalej. Všetky možné situácie sú popísané v predchádzajúcich troch krokoch. Proces dekompresie je oveľa jednoduchší, pretože tu už netreba hľadať žiadne zhody ani porovnávať dĺžky. Je znázornený na obrázku 3.9.

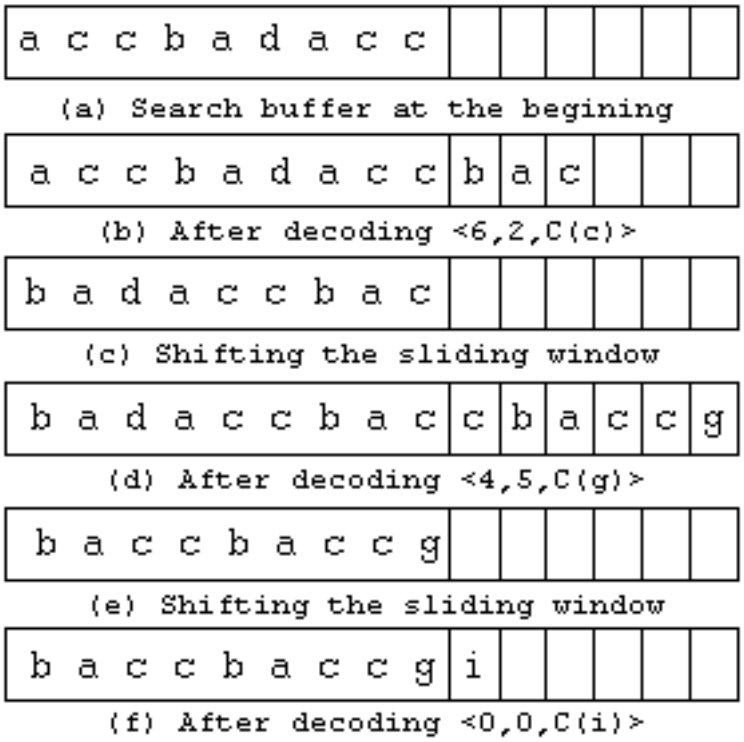
V časti (a) obrázka 3.9 je SB rovnaký ako v prvej časti dekódovania (3.8 (a)). Je v ňom uložený reťazec *accbadacc*, ktorý sme práve dekódovali.

Keď dostaneme na vstup triplet $< 6, 2, C(c) >$, dekóder sa posunie o 6 znakov doľava. Potom bude ukazovať na b, skopíruje dva znaky začínajúc znakom b, teda ba, do LAB. Symbol c skopíruje napravo od ba. Je to znázornené na obrázku 3.9b. Okná sa potom posunú o tri pozície doprava (3.9c).

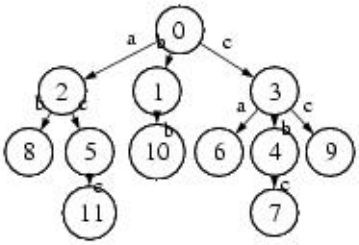
Pri $< 4, 5, C(g) >$ sa dekóder posunie o 4 pozície doľava (bude ukazovať na c) a skopíruje 5 znakov. Ako vidíme, na začiatku kopírovania máme k dispozícii iba 4 znaky. Piaty znak dostaneme až počas kopírovania. Všetky znaky, ktoré počas kopírovania vytvoríme, môžu ďalej slúžiť ako symboly na ďalšie kopírovanie. Nakoniec sa skopíruje ešte znak g (obrázok 3.9c), a posunie sa o 6 znakov doprava (obrázok 3.9e).

Triplet $< 0, 0, C(i) >$ hovorí, že nebola nájdená žiadna zhoda a iba samotné i je dekomprimované (obrázok 3.9f).

Dekompresiou týchto troch tripletov sme dostali *baccbaccgi*, čo je presne ten istý reťazec, ktorý bol pomocou nich v predchádzajúcej časti zakódovaný.



Obr. 3.9: LZ77 - dekompresia



Obr. 3.10: LZ78 graf

Záverečné poznámky

Označme si, tak ako už bolo predtým používané, search bufer ako SB, look-ahead bufer ako LAB, a veľkosť zdrojovej abecedy A. Tiež predpokladajme, že používame binárne kódovanie. Prístup LZ77 algoritmu je kódovať postupnosti s rôznou dĺžkou do kódových slov s fixnou dĺžkou. Na offset i použitých $\lceil log_2(SB) \rceil$ bitov, na LAB j $\lceil log_2(LAB + SB) \rceil$ a na zdrojový symbol k $\lceil log_2(A) \rceil$.

Pokiaľ sú rozostupy medzi opakujúcimi sa postupnosťami textu príliš veľké, väčšie ako veľkosť oboch bufrov, potom tento algoritmus nie je schopný komprimovať text.

Bolo vymyslených veľa modifikácií tohto algortimu, ktoré ho mali zefektívniť. Jednou z nich je LZSS [4].

3.4.3 LZ78

Algoritmus LZ78 eliminuje potrebu použitia druhej zložky pri metóde posúvania okien. Tiež používa zakódovaný text ako slovník, ale trocha inou metódou. Veľkosť slovníka pri tomto algoritme môže byť teoreticky neobmedzená, pretože každým krokom algoritmu sa jeho veľkosť zväčší. Maximálny výkon by dosiahol, keby kódovaný text bol nekonečne dlhý. V praxi však veľmi veľký slovník vedie k negatívnej efektívnosti tohto algoritmu. Preto po dosiahnutí určitej veľkosti slovníka sa prestane zväčšovať, prípadne sa môže vynulovať. Namiesto trojíc (tripletov), ktoré používa algoritmus LZ77, LZ78 používa iba dvojice, ktorých postupnosť je výsledkom komprimácie.

Kompresia a dekompresia

Tak, ako to bolo pri LZ77, tak aj tu si priebeh algoritmu vysvetlíme na príklade. Vytváranie

Index	Dvojica	Kód. symbol
1	$< 0, C(b) >$	b
2	$< 0, C(a) >$	a
3	$< 0, C(c) >$	c
4	$< 3, C(b) >$	cb
5	$< 2, C(c) >$	ac
6	$< 3, C(a) >$	ca
7	$< 4, C(c) >$	cbc
8	$< 2, C(b) >$	ab
9	$< 3, C(c) >$	cc
10	$< 1, C(b) >$	bb
11	$< 5, C(c) >$	acc

Tabuľka 3.3: Príklad kompresie LZ78 algoritmu

slovníka pri kompresii sa dá rozdeliť na niekoľko krokov. Na začiatku je slovník prázdny, prípadne, ak ho reprezentujeme pomocou n-árneho stromu, je tam iba koreň s hodnotou 0. Postup je nasledovn Postup je nasledovný:

- nájdi čo najdlhšiu zhodu v slovníku, ako sa dá. Nech jej dĺžka je l, pozícia na ktorej sme v zdrojom reťazci začínali b, dĺžka už existujúceho slovníka dl a pozíciu v slovníku, kde bola nájdená zhoda m.

- do slovníka pridaj (teda na pozíciu $dl+1$) položku $< m, S[b+l+1] >$, kde $S[b+l+1]$ je $b+l+1$-ty znak v komprimovanom reťazci. Pokiaľ máme slovník reprezentovaný tak ako v tabuľke 3.3, potom do slovníka môžeme pridať aj reťazec, ktorý sme zakódovali. Robí sa to pre lepšiu orientáciu. Pokiaľ je slovník reprezentovaný ako strom, potom hľadanie najdlhšej zhody je iba traverzovanie stromu od koreňa k listom po ohodnotených hranách a pridanie záznamu je iba pridanie jedného syna[2] pod nájdený uzol[3].

- ako dvojicu na výstup kódera uveď $< m, S[b+l+1] >$.

[2] prislúchjúca hrana bude mať hodnotu $S[b+l+1]$ s syn $dl+1$

[3] má hodnotu m

Majme vstupný text *baccbaccacbcabccbbacc*. Tabuľka 3.3 a obrázok 3.10 znázorňujú proces kompresie.

Vidíme, že pre prvé tri znaky sa v slovníku postupne nenájde žiadna zhoda, takže výstupom sú trojice $< 0, C(b) >$, $< 0, C(a) >$, $< 0, C(c) >$. Ďalším znakom je c, kde je nájdená zhoda v slovníku na treťom mieste, takže sa pridáva ďalší symbol b. Pre reťazec cb už zhoda v slovníku neexistuje a preto je tento reťazec zakódovaný ako dvojica $< 3, C(b) >$ a je pridaný do slovníka. Na vstupnom reťazci sa posunieme o dva symboly doprava. V tomto postupe by sme pokračovali až do konca kódovaného reťazca. V každom kroku veľkosť slovníka narastie o jednu položku a postupnosti znakov, ktoré jednotlivé položky v ňom kódujú, sa stávajú dlhšími. To znamená, že jeho efektivita postupne rastie.

Dekompresia je o niečo jednoduchšia. Tu sa tiež začína s prázdnym slovníkom. To znamená, že slovník pri prenose netreba posielať, pretože sa postupne buduje. Dvojica $< 0, C(b) >$ hovorí, že nebola nájdená žiadna zhoda, dekóduje sa znak b a táto dvojica sa pridá do slovníka. Podobný postup platí pre nasledujúce dve dvojice $< 0, C(a) >$, $< 0, C(c) >$. Pri dvojici $< 3, C(b) >$ vie, že na výstup má dať text, ktorý je zakódovaný v slovníku na pozícii 3[4] a na koniec pridať znak b a túto dvojicu znova pridať do slovníka.

Záverečné poznámky

Koľko bitov potrebujeme na zakódovanie každej dvojice? Je to $\lceil log_2(A) \rceil$ na druhý symbol. Počet bitov na prvý symbol závisí od veľkosti slovníka, ktorý, samozrejme, musí byť väčší ako počet rôznych zdrojových symbolov. Na druhej strane, keď je príliš vysoká horná hranica, pri krátkych textoch sa prejaví skôr negatívna kompresia. Ako je vidieť, v prvých krokoch prevláda negatívna kompresia a účinnosť tohto algoritmu sa prejavuje až po niekoľkých desiatkach krokov.

3.4.4 LZW

Algoritmus LZW patrí do skupiny LZ algoritmov. Je pomenovaný podľa autora jeho modifikácie, Welch (1984). Redukuje druhú zložku (symbol, ktorý nasledoval za nájdenou najdlhšou postupnosťou) algoritmu LZ78, čím ešte viac zvyšuje jeho efektívnosť. Inak povedané, dekóderu sa posielajú iba indexy do slovníka. Vďaka tomu sa zmenil aj počiatočný

[4] Tu je pri každej položke vhodné si zapamätať, aký reťazec kódujeme, aby sme nemuseli rekurzívne hľadať v slovníku, prípadne prechádzať strom. Urýchľuje to prácu algoritmu.

Index	Entry	Input Symbols	Encoded Index
1	*a*	initial dictionary	
2	*b*	initial dictionary	
3	*c*	initial dictionary	
4	*d*	initial dictionary	
5	*ac*	a	1
6	*cc*	c	3
7	*cb*	c	3
8	*ba*	b	2
9	*ad*	a	1
10	*da*	d	4
11	*acc*	a,c	5
12	*cba*	c,b	7
13	*accb*	a,c,c	11
14	*bac*	b,a	8
15	*cc*...	c,c,...	

Tabuľka 3.4: Príklad kompresie LZW algoritmu

slovník, ktorý na začiatku obsahuje všetky jednoznakové postupnosti zo zdrojovej postupnosti.

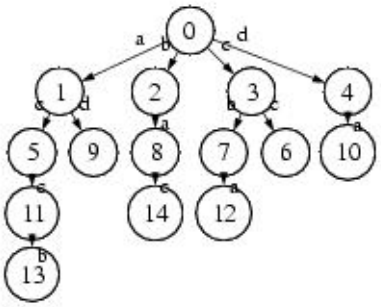
Kompresia a dekompresia

Pri kompresii, ako pri LZ78, sa hľadá najdlhšia zhoda v slovníku. Index, na ktorom v slovníku nastala zhoda je výstupom algoritmu a zväčšený reťazec[5] je pridaný ako nová položka do slovníka.

Ako vidíme, kompresia je pomerne jednoduchá, čo pri dekompresii neplatí. Tak, ako to bolo v predchádzajúcich prípadoch, aj tu použijeme na vysvetlenie príklad.

Majme na vstupe reťazec *accbadaccbaccbacc*. Vidíme, že zdrojová abeceda, $S = (a, b, c, d)$, pozostáva zo štyroch symbolov, ktoré sú na prvých miestach v našom slovníku (tabuľka 3.4, obrázok 3.11). Prvá nájdená najdlhšia postupnosť je a, preto je ďalší symbol c pri-

[5] nájdená zhoda nasledujúca ďalším znakom



Obr. 3.11: LZW graf

daný k tomuto reťazcu, do slovníka je pridaná položka ac na piatu pozíciu a na výstup poslaný znak 1[6]. Takto by dekóder pokračoval ďalej. Pozrime sa ešte na pozíciu 11 v našom slovníku. Vstupný symbol na tomto mieste je a. Keďže je nájdená zhoda, zoberieme ďalší znak c. Tu je nájdená zhoda s reťazcom ac na piatom mieste v slovníku, takže je pridaný nasledujúci znak c. Tu už nie je nájdená žiadna zhoda, tento „zväčšený“ reťazec je pridaný do slovníka a ako výstupný reťazec je znak 5. Konečná sekvencia zakódovaných indexov je $1, 3, 3, 2, 1, 4, 5, 7, 11, 8$. Tak, ako u LZ78, aj tu sa položky (indexy) stávajú čoraz väčšími a tým pádom potrebujeme na ich binárne zakódovanie viac a viac bitov. Tiež tu platí horné ohraničenie veľkosti slovníka, prípadne jeho reinicializácia počas behu.

Pri dekódovaní sa slovník vytvára, dá sa povedať, o jeden krok „dozadu“. Po dekódovaní posledného kódového slova bude, z tohto dôvodu, v slovníku o jeden záznam menej, čo vôbec nevadí, pretože tento záznam bol do slovníka pridaný ako posledný, a tým pádom nebol nikdy použitý. Na začiatku má dekóder v slovníku iba prvé 4 znaky. Keď dostane prvý symbol 1, dekóduje a[7]. Druhý symbol 3 hovorí, že bol zakódovaný znak c a do slovníka pridá reťazec ac. A ako vie, čo má pridať? Je to celý reťazec, ktorý bol výstupom v minulom kroku algoritmu[8] a prvý znak z reťazca dekódovaného v tomto kroku[9].

[6] pozícia v slovníku, kde bola nájdená najdlhšia zhoda
[7] prvý krok sa mierne líši od ostatných
[8] v našom prípade a
[9] v našom prípade c

Zdroj. symbol	Početnosť	Pridelený interval	CP
S_1	0.3	$< 0, 0.3)$	0
S_2	0.1	$< 0.3, 0.4)$	0.3
S_3	0.2	$< 0.4, 0.6)$	0.4
S_4	0.05	$< 0.6, 0.65)$	0.6
S_5	0.1	$< 0.65, 0.75)$	0.65
S_6	0.25	$< 0.75, 1)$	0.75

Tabuľka 3.5: Aritmetické kódovanie

Záverečné poznámky

Ako vidíme, efektivita sa zväčšila vďaka odstránenému druhému symbolu v kódovom slove, takže jeho veľkosť závisí naozaj už len od veľkosti samotného slovníka.

Pri kompresii obrázkov našiel algoritmus LZW svoje uplatnenie. Je používaný vo formáte GIF (graphic interchange format).

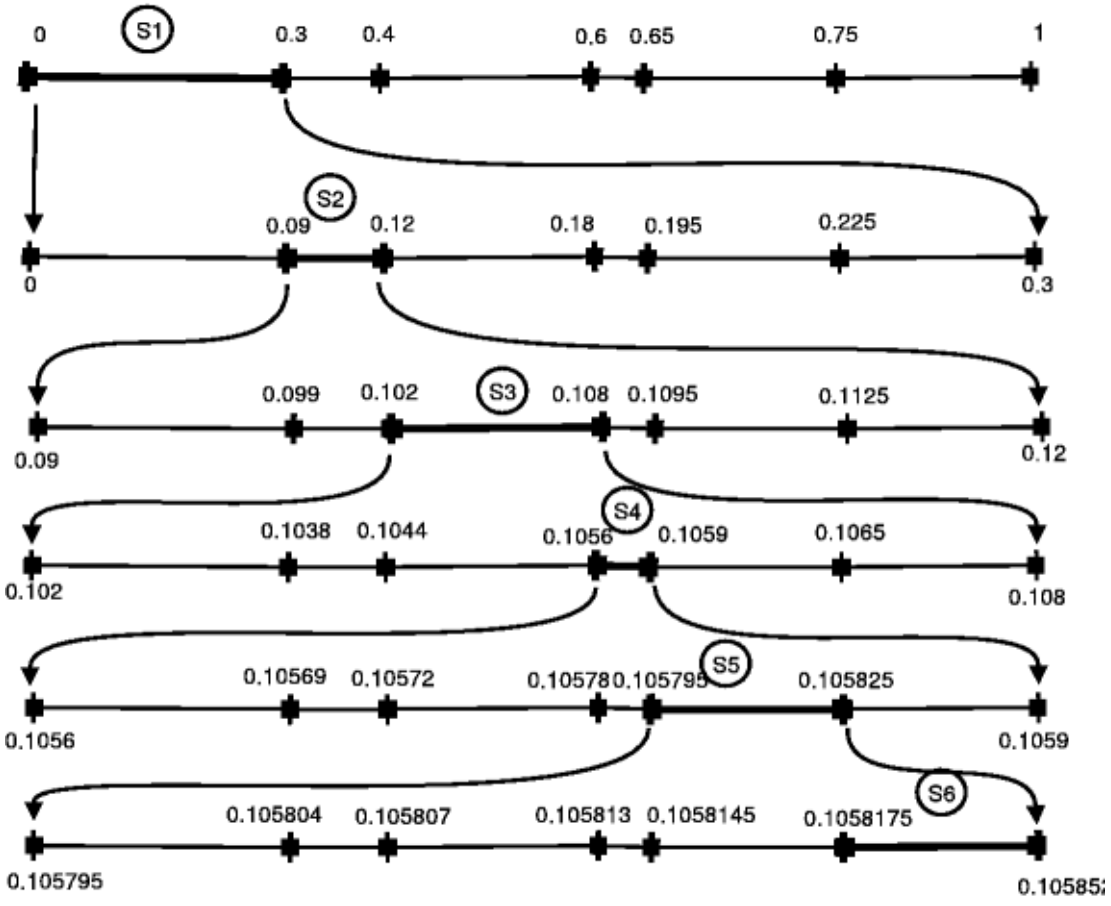
3.5 Aritmetické kódovanie

Aritmetické kódovanie sa dá zaradiť niekde medzi Huffmanov kód a LZ algoritmy. Potrebuje predspracovanie, alebo frekvenčnú tabuľku ako Huffmanov kód, ale na rozdiel od neho nie je to blokový kód.

3.5.1 Delenie intervalu $< 0, 1)$

V tomto algoritme nie je nutné usporiadať symboly vstupnej abecedy podľa ich početnosti a preto sú aj v tabuľke 3.5 usporiadané podľa abecedy. Interval medzi 0 a 1 je rozdelený na 6 podintervalov, každý veľkosti $p(s_i), i = 1, 2, \ldots, 6$. Pri každom symbole je uvedená jeho CP (*cumulative portability* - Langdon, 1984), kde $CP(s_1) = 0$. Teraz vidieť, že spodná hranica každého podintervalu je $CP(s_i)$ hodnota. Šírka podintervalu je rovná početnosti odpovedajúceho znaku zo zdrojovej abecedy.

$$CP(s_i) = \sum_{j=1}^{i-1} p(s_j)$$



Obr. 3.12: Delenie intervalu pri aritmetickom kódovaní

Tento algoritmus produkuje jedno kódové slovo, je to číslo z intervalu $< 0, 1)$, pre skupinu zdrojových znakov. Záleží iba na dohode, koľko znakov to je. To značí, že pri dekódovaní z jedného kódového slova vieme dekódovať hneď niekoľko zdrojových symbolov.

Na príklade si ukážeme kompresiu a dekompresiu reťazca $s_1 s_2 s_3 s_4 s_5 s_6$.

3.5.2 Kompresia

Na začiatku máme interval $< 0, 1)$ rozdelený na šesť častí (viď obrázok 3.12a). Keďže prvý symbol je s_1, zoberieme podinterval $< 0, 0.3)$. To znamená, že výsledné kódové slovo bude z tohto intervalu. Tento interval znova rozdelíme v takom istom pomere, teda podľa početnosti zdrojových symbolov, ako interval $< 0, 1)$ (viď obrázok 3.12b). Znak s_2 vyberie z tohto intervalu podinterval $< 0.09, 0.12)$, ktorý následne rozdelíme ako v predchádzajúcom kroku. Pomerne jednoducho sa dá napísať vzťah, podľa ktorého získame pozíciu nového podintervalu[10].

Nech j je index znaku, ktorý bol z daného intervalu vybratý. Potom pre všetky zdrojové symboly aplikujeme následovný výpočet:

$$L_{i,new} = L_{j,current} + W_{j,current} * CP_i,$$

$$W_{i,new} = W_{j,current} * p(s_i),$$

kde $L_{i,new}$ a $W_{i,new}$ sú nové hodnoty pre znak s indexom i, $L_{j,new}$ a $W_{j,current}$ sú hodnoty určujúce vybratý znak v pôvodnom intervale.

Tieto dve rekurzie, tiež nazývané dvojitá rekurzia (*double recursion*, Langdon, 1984), hrajú hlavnú úlohu v aritmetickom kódovaní. Podobným spôsobom by prebiehalo delenie intervalu pre s_3, s_4, s_5, s_6. Po poslednom delení nám ostane podinterval $< 0.1058175, 0.025250)$ (obrázok 3.12f). Tento interval reprezentuje dosiaľ zakódované dáta. Kódové slovo, ktoré bude reprezentovať tieto dáta, bude reálne číslo z tohto intervalu. Najvhodnejším kandidátom je číslo, ktorého zápis v binárnej podobe má najmenej znakov, aby úroveň kompresie bola čo najviac účinná.

3.5.3 Dekompresia

Dekódovanie je v tomto algoritme presne opačný proces. Na začiatku máme k dispozícii frekvenčnú tabuľku 3.5, interval $< 0, 1)$ a reálne číslo, ktoré nám reprezentuje zakódovanú

[10] na určenie nám stačí jeho spodná hranica a veľkosť

postupnosť. V našom prípade nech to reálne číslo je spodná hranica posledného intervalu, ktorý sme dostali pri kompresii, teda 0.1058175.

Na začiatku rozdelíme interval $< 0, 1)$ podľa tabuľky 3.5 rovnako ako v prvom kroku kompresie. Zistíme, v ktorom intervale z vytvorených sa nachádza naše kódové slovo. Zdrojový symbol prislúchajúci danému intervalu, teda s_1, je dekódovaný symbol. Hranice intervalu, ktorý reprezentoval tento znak budú slúžiť na ďalšie delenie, presne tak isto ako pri kompresii.

Preskočme niekoľko krokov a posuňme sa až k dekódovaniu symbola s_6. Po jeho dekódovaní nám ostane interval $< 0.1058175, 0.025250)$. Otázka znie: ako vieme, že máme v tomto bode prestať s ďalším delením intervalu? Tu záleží na dohode medzi kóderom a dekóderom. V princípe existuje niekoľko dohôd, spomeniem iba dva:

- kóduje sa pevný počet znakov[11]
- do zdrojovej abecedy sa pridá ďalší špeciálny znak (napr. \$), ktorý sa v nej predtým nevyskytoval s početnosťou 1. Keď sa kóder rozhodne ukončiť sekvenciu, tak zakóduje tento znak a na výstup dá reálne číslo z daného intervalu. Keď dekóder tento znak rozpozná, tak vie, že má skončiť, ale tento znak nepošle na výstup.

3.5.4 Záverečné poznámky

Oba procesy, kompresia aj dekompresia, používajú iba aritmetické operácie (sčítanie, násobenie, odčítanie, delenie). Preto sa tento algoritmus volá *aritmetické kódovanie*.

Kódovaním zdrojových znakov sa výsledný podinterval stáva menším a menším, teda spodná a horná hranica intervalu sa stále približujú. Tu sa stretávame s problémom presnosti reprezentácie reálnych čísel v počítačoch, kde ich presnosť je obmedzená a môže sa stať, že tieto hranice splynú do jedného čísla. Tento problém bol vyriešený v tzv. *inkrementálnej implementácii* aritmetického kódovania[12].

[11] v našom prípade by to bolo 6

[12] viac o tejto technike sa môžete dozvedieť v [3]

Kapitola 4

Implementácia projektu

Pre učebnicu, ktorá je vlastne hlavným produktom tejto práce, som sa rozhodol použiť kombináciu technológii HTML a Java. Obidve tieto technólogie sú nezávislé na platforme, čo pomerne ľahko zabezpečuje beh na ľubovoľnej platforme, ktorá ich podporuje.

4.1 HTML

Ako HTML štandard je použitý XHTML[1] a grafický design je dotváraný CSS2[2].

Pre korektné zobrazenie stránok sa odporúča použitie prehliadača, ktorý má tieto štandardy implementované. Čiastočný zoznam týchto prehliadačov je uverejnený na http://www.w3.org/Style/CSS/#browsers. Z tých známejších medzi ne patria `Opera 6` (Win, Lin), `Opera 7` (Win), `Mozilla 1.0` (Win, Lin), `Konqueror` (Lin) ale tiež mnohé ďalšie.

4.2 Java

Vďaka technológii Java, ktorá umožňuje na prezentácie umiestňovať `aplety`, je zabezpečená interaktivita tejto učebnice. Pre každý algoritmus existuje jeden java aplet (ďalej len aplet), ktorý ilustruje do istej miery činnosť algoritmu ako pri kompresii, tak pri dekompresii. Aby boli aplety na stránkach funkčné, je potrebné mať korektne nainštalovanú `Javu 1.4` alebo vyššie, ktorú si pre svoju platformu môžete stiahnuť na stránkach http://java.sun.com/.

[1] viac o tomto štandarde sa nachádza na http://www.w3.org/MarkUp/

[2] viac o tomto štandarde na http://www.w3.org/Style/CSS/

4.2.1 Implementácia

Bolo vytvorených niekoľko tried, ktoré zapuzdrujú spoločné vlastnosti komprimačných algoritmov, či už funkcionálne alebo vizuálne (pri vizualizácii).

Objekty a triedy

trieda `MyJComponent` je základný komponent pre vizuálne prvky apletov. Je odvodený od `JComponent`. Má rozšírené vlastnosti o akcie `drawBefore` a `drawAfter`, ktoré umožňujú kresliť na `Graphics` objektu pred, alebo po samotnom vykreslení objektu. Toto sa využíva hlavne pri zobrazovaní vstupu[3]. Ďalšou implementovanou vlastnosťou je `Drag2Move`, ktorá umožňuje pohodlne hýbať pomocou myšy obsahom okna.

trieda ShowString je základný komponent na vykresľovanie reťazcov. Je odvodený od `MyJComponent`. Má implementované metódy na pridávanie/mazanie znakov a zmenu ich atribútov. Keďže je to abstraktná trieda, samotné vykresľovanie znakov je ponechané na jej potomkov[4].

trieda Tracer je základom pre všetky kompresné algoritmy. Má implementované ich spoločné vlastnosti. Každý jej potomok musí mať implementované metódy `encStep`, `decStep` a `encode`.

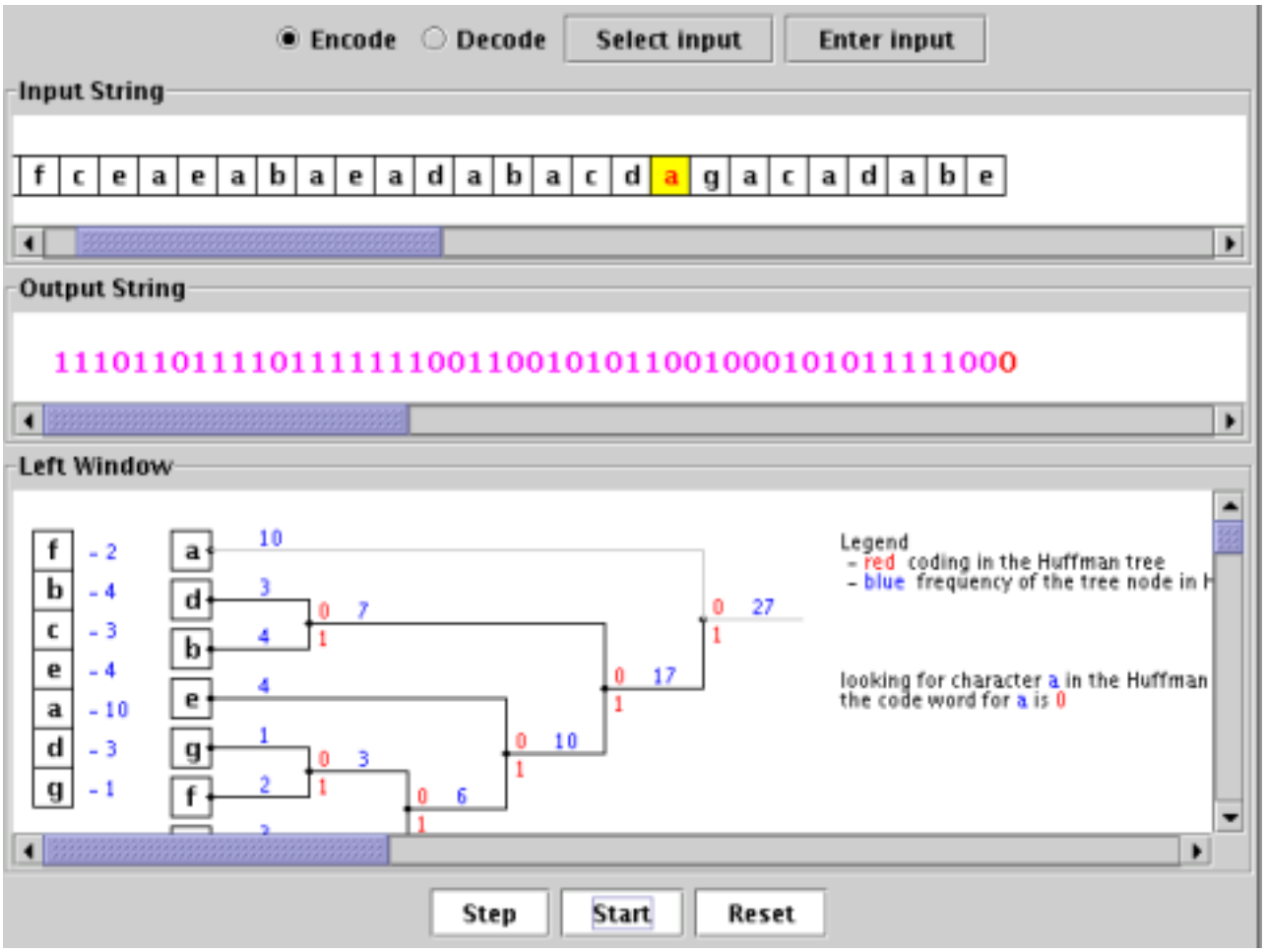
4.2.2 Popis apletu

Vzhľad apletu (obrázok 4.1) sa dá rozdeliť do štyroch základných častí.

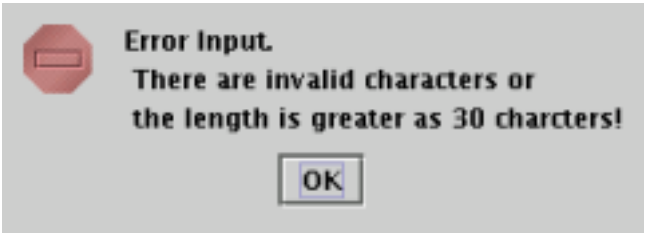
Parametre algoritmu zohrávajú asi najväčší význam pri jeho účinnosti. Pokiaľ pre algoritmus takéto parametre existujú, potom sa nachádzajú v tejto časti. Medzi štandardné nastavenia patrí zmena stavu algoritmu (kompresia/dekompresia), zadanie vopred zvoleného alebo vlastného vstupu. Vlastný vstup obmedzený počtom tridsiatich znakov a sú k dispozícii iba určité znaky (napr. pre RLE sú k dispozícii iba znaky 01). Po nedodržaní aspoň jedného pravidla sa zobrazí chybová hláška (obrázok 4.2).

Vstup a výstup algoritmu je možné vidieť v oknách pomenovaných ako `Input String` a `Output String`. Tu zobrazovanie informácii závisí od toho, v akom stave sa algorit-

[3] pozri časť *Ukážka apletu / Vstup a výstup*
[4] pozri časť *Ukážka apletu / Vstup a výstup*



Obr. 4.1: Ukážka apletu



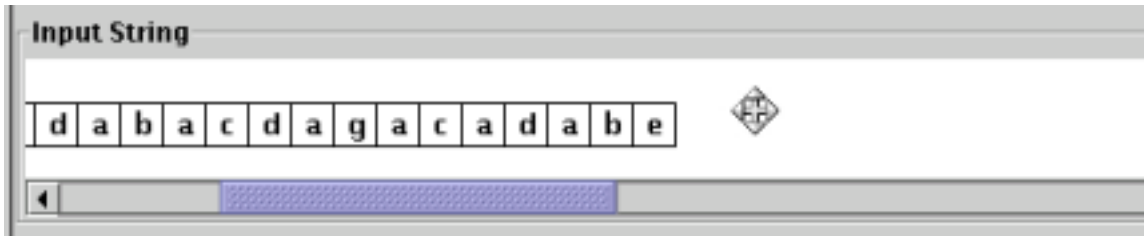
Obr. 4.2: Chybne zadaný vstup

mus práve nachádza (kompresia/dekompresia). Podľa toho je zobrazený celý vstupný reťazec (kompresia) a vo výstupnom reťazci postupne pribúdajú kódové slová alebo opačne, pri dekompresii je celý výstupný reťazec zobrazený (reťazec pozostávajúci z kódových slov) a vstupný reťazec sa zobrazuje postupne, ako je dekódovaný.

Vizualizácia algoritmu prebieha z väčšej časti v dolnom okne. V niektorých apletoch sú namiesto jedného okna dve vedľa seba. Tu bývajú znázornené hodnoty premenných, popísané udalosti, alebo stav istej štruktúry, ktorá sa používa v algoritme. V tejto časti užívateľ nemôže interaktívne zasahovať.

Ovládanie celého apletu je realizované troma tlačidlami (`Step`, `Start/Stop`, `Reset`) v dolnej časti apletu. Tlačidlo `Step` slúži na prevedenie jedného kroku algoritmu, `Start/Stop`, na automatický beh algoritmu po krokoch v určitých časových intervaloch, `Reset` na vrátenie apletu do stavu pred začatím samotného procesu kompresie alebo dekompresie. Toto tlačidlo je vhodné použiť pred zmenou parametrov algoritmu, alebo pred zmenou vstupného komprimovaného reťazca.

V niektorých prípadoch sa môže stať, že zobrazovaná informácia sa nezmestí do okna. Preto majú všetky zobrazovacie prvky implementovanú metódu `Drag2Move`, čo značí, že na dané okno stačí kliknúť myšou a posúvať kurzor. Po kliknutí sa zmení tvar kurzoru (obrázok 4.3).



Obr. 4.3: Zmena kurzora pri Drag2Move

Literatúra

[1] Jaroslav Polec a kolektív, *Vybrané metódy kompresie dát*, Univerzita Komenského 2000, ISBN 80-223-1392-0

[2] Murray J.D., VanRyper, W., *Encyklopedie grafických formátů*, Copmuter Press 1997, ISBN 80-7226-033-2

[3] Shi Y. Q., Sun H., *Image and video compresoin for mutlimedia engineering: fundamentals, algorithms and standarts*, CRC Press 2000, ISBN 0-8493-3491-8

[4] Gibson J. D., Berger T., Lookabaugh T, Lindberg D., Barker R. L., *Digital Compresion for Multimedia, Principles and Standarts*, Morgan Kaufman Publishers, Inc. San Francisco, California 1998

[5] Data Compresion Informations
http://datacompression.info/

[6] World Wide Web Consorcuim
http://www.w3.org

[7] Java 2 Platform, API Specification
http://java.sun.com/j2se/1.4/docs/api/

[8] Bruce Eckel, *Thinking in Java, 3rd Edition*, Prentice-Hall, December 2002, ISBN 0131002872
http://www.mindview.net/Books/TIJ/

[9] Matthew Robinson, Pavel Vorobiev, Pavel A. Vorobiev, David Karr, *Java Swing, Second Edition*, Manning Publications Company, ISBN 193011088X
http://manning.spindoczine.com/sbe/

[10] Juraj Štugel, *Výuka počítačovej grafiky na internete*
http://pg.netgraphics.sk/