Kompresia dát

Vo všeobecnosti poznáme dátovú kompresiu stratovú (po dekódovaní nie je možné získať všetky informácie) a bezstratovú. Po dekompresií dát zakódovaných pomocou bezstratovej kompresie dostaneme pôvodné dáta. Prezentované algoritmy boli v čase ich vzniku primárne určené na kódovanie reťazcov, ale je ich možné použiť (a aj sa používajú) na kódovanie akýchkoľvek dát. Znak je „iba“ zhluk bitov.
Pre názornosť sú algoritmy prezentované a vysvetľované na kompresií textu. Algoritmy a ich rôzne varianty sa používajú dodnes (kompresia gif obrázkov, deflate algoritmus využívaný .zip formátom...).

Reťazec

- **Ak S je ľubovoľná konečná množina, tak pole 1..n, ktorého prvky patria S, je reťazec w nad množinou S.**
- **|w| je označenie pre dĺžku reťazca w**
- **množina S je abeceda**
- **prvky patriace do S nazývame znaky (character).**
- **prázdny reťazec označme *e***
- **text určený na zakódovanie označme E**

Pamäťová reprezentácia reťazcov (textu)

Obvyklá je enormná dĺžka reťazcov => dôležitosť úspornej reprezentácie.
Typicky text obsahuje veľké množstvo znakov.

Pre úsporu pamäte je možné pre každý znak zadefinovať kód ako postupnosť bitov. Text bude reprezentovaný postupnosťou kódov. Veľmi často je frekvencia výskytu jednotlivých znakov v texte rôzna, čo nám umožňuje navrhnúť pamäťovo efektívnu reprezentáciu textu.

Našim cieľom je minimalizovať celkový počet bitov výslednej reprezentácie textu.

Proces prevodu textu na jeho bitovo úspornú reprezentáciu budeme nazývať kódovanie alebo kompresia textu.
Predpokladom tohto procesu je, že po kompresii sa nebude text modifikovať, ani sa nebudú znaky náhodne sprístupňovať.

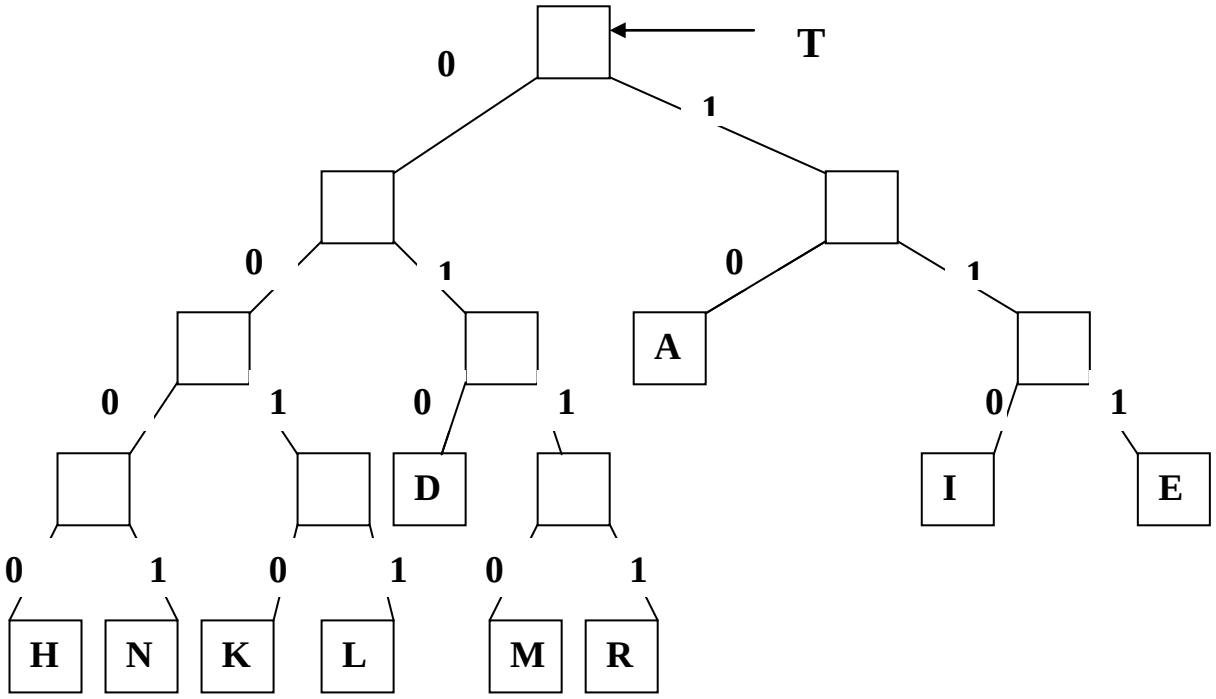
Huffmanovo kódovanie

Princíp: čím frekventovanejší znak, tým bude kratšia sekvencia bitov v jeho kóde. Prvé experimenty s kódovaním v roku 1952.
Riziko: dekódujeme správne originálny text?

Príklad: A:101 K:110 Q:101110
Kód 101110 môže ale reprezentovať reťazec AK aj Q. A je totiž prefixom kódu Q.

Možnosť: Ak zabezpečíme, že pre žiadnu dvojicu kódov
c_1, c_2 nie je c_1 prefixom c_2 =>
neexistujú reťazce w_1, w_2 také, že
kód (w_1) = kód (w_2).

Reprezentácia Huffmanovho kódu : digitálny znakový strom (trie)



Abeceda na obrázku obsahuje iba 10 písmen.

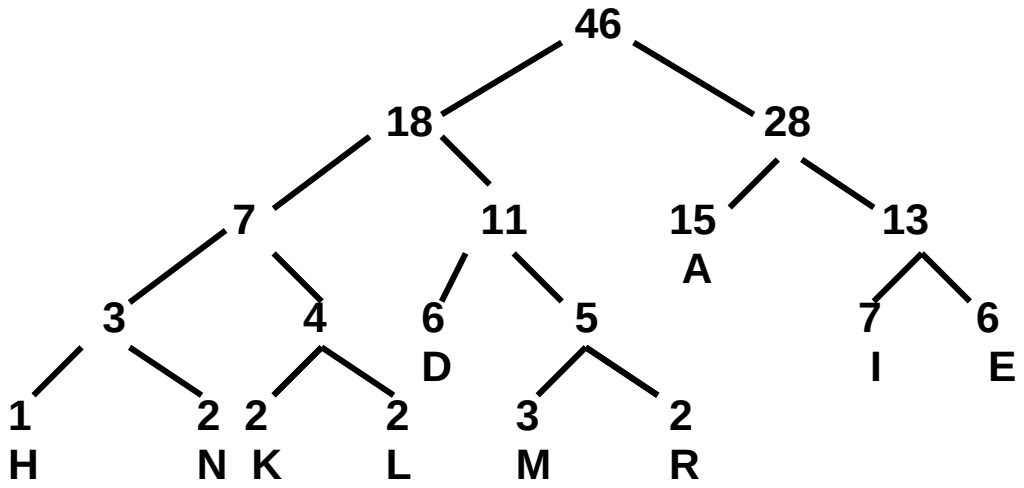
V každom liste je práve jeden znak Huffmanovho kódu. Každý znak z S sa práve raz v nachádza v niektorom z listov stromu. Postupnosť bitov kódujúcich znak je jednoznačne určená cestou z koreňa do listu so znakom.
Keďže znaky sa nachádzajú iba v liste znamená to, žiaden kód nie je prefixom iného a teda budeme vždy schopný jednoznačne dekódovať text.
Samotný kódovací strom je úplný, to znamená, že "nelist" má vždy 2 synov.

Napríklad kód textu MIRKA z obrázku je b = 011011001110010 10. Ak pri prechádzaní stromom narazíme na list zapíšeme znak, ktorý sa v tomto liste nachádza do dekódovaného textu. Ďalší bit patrí ďalšiemu znaku a tak začneme strom opäť prechádzať od koreňa až do niektorého z listov.

Problémom je ako skonštruovať optimálny strom.
Predpoklad pre jeho zostrojenie je, že pre každý znak c_i textu E poznáme frekvenciu jeho výskytu v texte f_i (koľkokrát sa vyskytne v texte E).

Vytvorenie <u>Huffmanovho kódovacieho stromu</u> T:

- **Vytvor vrchol pre každý znak z E.**
- **Každý tento vrchol bude listom stromu T.**
- **Každý vrchol obsahuje *váhu* vrcholu.**
- **Priraď každému listu so znakom c_i váhu f_i.**
- **Opakuj pokým nevznikne jediný strom:**
    - **vyber dva vrcholy n_1, n_2 s najmenšou váhou a vytvor im otca so súčtom váh, ďalšieho výberu sa zúčastní otec vrcholov n_1, n_2 (pri viacerých možnostiach vyber ľubovoľnú)**



Pri tvorbe stromu je výhodné použiť vhodnú implementáciu prioritného frontu, ktorá nám umožní efektívne vyhľadať dva vrcholy s najmenšou frekvenciou (napr. párovaciu haldu). Po vytvorení kódovacieho stromu je potrebné výsledné kódy uložiť vo vhodne zvolenej štruktúre, ktorá umožní k zadanému znaku priradiť bitový kód.
Frekventovanejšie znaky budú vo výslednom strome bližšie ku koreňu a majú kratší kód. Čím je znak frekventovanejší tým bude kódovaný menším počtom bitov.

Ak by A malo frekvenciu 20, kódovalo by sa iba jediným bitom! Žiaden iný kódovací strom nemá kratšiu reprezentáciu textu (Huffmanova optimalita).

Problém tohto kódovania je, že frekvencia znakov musí byť vopred známa.

Možnosti:

A.) <u>2 - fázové Huffmanovo kódovanie</u>

- **1 fáza - zistenie frekvencií, 2 fáza - kódovanie**
- **časovo náročnejšie hlavne pri externých médiách**
- **neaplikovateľné pre generované prúdy textov (program, komunikačný kanál, ...)**

B.) <u>Statické Huffmanovo kódovanie</u>

Jeden kódovací strom pre všetky príbuzné texty (napr. beletria v slovenskom jazyku). Toto riešenie môže byť blízke k optimálnemu riešeniu.

C.) <u>Adaptívne Huffmanovo kódovanie</u>

- **Začať s prázdnym stromom: každý znak z E má frekvenciu 0.**
- **Po prečítaní každého znaku z E aktualizovať strom T, aby bol optimálny pre doteraz kódovanú časť E.**
- **Existuje viacero algoritmov na budovanie stromu online - FGK (Faller-Gallager-Knuth) algoritmus, Vitterov algortimus.**

Kódovanie Shannon - Fano

- **Claude Elwood Shannon a Robert Mario Fano, vytvorili v roku 1949 kódovanie s variabilnou dĺžkou kódu**
- **Kompresný algoritmus prechádza vstupné dáta dva krát (dvojprechodové kódovanie). Pri prvom prechode sa zistí, ktoré jednotlivé symboly sa nachádzajú vo vstupných dátach a zároveň početnosť výskytu symbolov, rovnako ako pri dvojfázovom Huffmanovom kódovaní.**

<u>Postup pre vytvorenie jednotlivých kódov</u>:

I. **usporiadame symboly podľa početnosti (vzostupne, príp. zostupne)**
II. **rozdelíme symboly na dve časti tak, aby sa suma početnosti výskytu symbolov na ľavej strane rovnala alebo čo najviac rovnala sume početnosti výskytu symbolov na pravej strane, všetkým symbolom na ľavej strane pridáme kód 0 a všetkým na pravej strane kód 1**
III. **s každou časťou opakujeme krok II. - pokým nezískame časti s jediným symbolom.**

Po vytvorení tabuľky kódov, prechádzame vstupné dáta druhý krát a vytvárame výstupný bitový kód podľa získaných kódov.

Príklad pre vytvorenie tabuľky kódov:

Majme reťazec *"shannon-fano".* Prejdeme vstupné dáta prvý krát a zistíme početnosti výskytu jednotlivých symbolov.

Vytvoríme tabuľku kódov:

Symbol	n	a	o	s	h	-	f
Výskyt	4	2	2	1	1	1	1
Krok č. 1	0		1				
Krok č. 2	0	1	0		1		
Krok č. 3			0	1	0	1	
Krok č. 4						0	1
Bitový kód	00	01	100	101	110	1110	1111

Po prvom prechode vstupnými dátami sme symboly usporiadali zostupne podľa ich početnosti výskytu. V kroku číslo 1 sme symboly rozdelili na dve časti. Prvá časť (ľavá strana) pozostáva zo symbolov „n“ a „a“. Druhá časť (pravá strana) pozostáva zo symbolov „o“, „s“, „h“, „-“ a „f“. Symboly sme rozdelili na tieto dve časti, pretože suma početností výskytu symbolov na pravej strane sa rovná sume početností výskytu symbolov na ľavej strane. Keď máme symboly rozdelené, pridáme každému symbolu na ľavej strane bitový kód 0 a každému symbolu na pravej strane bitový kód 1.

V kroku číslo 2 opakujeme rovnaký postup s jednotlivými časťami ako v kroku číslo 1. V tomto kroku nám vznikli časti, ktoré

majú iba jeden symbol. Časti s jedným symbolom už ďalej nerozdeľujeme. Tento postup opakujeme až kým nemáme žiadnu časť, ktorá by mala dva symboly.

Po rozdelení všetkých častí, sme získali bitové kódy pre jednotlivé symboly a tabuľku kódov máme hotovú.

Symbol	Bitový kód
n	00
a	01
o	100
s	101
h	110
-	1110
f	1111

Pri druhom prechode už len aplikujeme vzniknutú tabuľku a máme výsledný bitový kód.

Na dekompresiu potrebujeme poznať tabuľku kódov. Následne prechádzame binárnymi dátami až pokým nevytvoríme postupnosť bitov, ktorá sa v kódovej tabuľke nachádza.

Kódovanie LZ78

- **publikované Abrahamom Lempelom a Jacobom Zivom v roku 1978**
- **existuje viac možností implementácie (s ohraničeným slovníkov, s neohraničených slovníkom, ...)**
- **ide o jednoprechodový systém (dáta prechádzame jediný raz) na rozdiel od 2-fázového Huffmanovho kódovania a kódovania Shannon - Fano**
- **metóda je pamäťovo náročná**
- **Pri kompresií sa vytvárajú dvojice <pozícia, rozširujúci bajt> . . .". Pozícia predstavuje pozíciu v slovníku na ktorom sa**

nachádza reťazec bajtov, ktorý je rozšírený o rozširujúci bajt (znak) z danej dvojice.

<u>Postup pre kompresiu:</u>

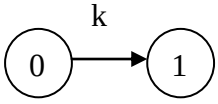
V každom kroku sa z nezakódovanej časti nájde v slovníku najdlhší reťazec. Na výstup je vložená dvojica zložená z indexu v slovníku a prvého znaku zatiaľ nezakódovanej časti. Nová fráza je v slovníku o tento znak rozšírená a označená najmenším možným nepoužitým číslom.

Pri implementácií existuje viacero spôsobov realizácie slovníka. My si pre názornosť ukážeme slovník znázornený pomocou stromu. Tento „slovník“ nie je potrebné si pamätať. Do výstupu sa zapisujú dvojice a „slovník“ sa v procese dekompresie bude budovať podľa prečítaných dvojíc.

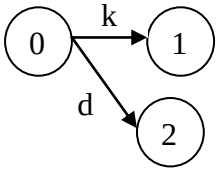
Majme reťazec „kdoononn“.

Vytvoríme koreňový vrchol 0.

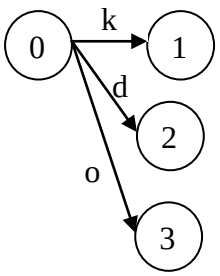
V slovníku sa nič nenachádza. Vytvoríme prvú dvojicu <0, k>.



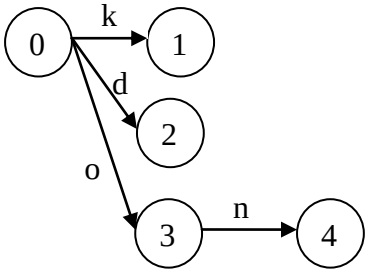
V slovníku sa “d” nenachádza a preto pridáme dvojicu <0, d>.



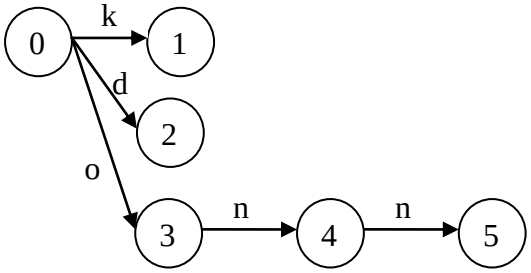
V slovníku sa “o” nenachádza a preto pridáme dvojicu <0, o>.



Zostáva zakódovať “ononn”. Najdlhší reťazec zo slovníka je “o”. Pridáme do slovníka preto dvojicu <3, n>.



Zostáva zakódovať “onn”. Najdlhší reťazec zo slovníka je “on”. Pridáme do slovníka preto dvojicu <4, n>.



Výstupný zakódovaný text sa bude skladať z dvojíc <0, k>, <0, d>, <0, o>, <3, n>, <4, n>.
Zámerne nie je uvedený binárny tvar dát. Ten by sa skladal samozrejme z dvojíc vhodne prevedených do binárneho tvaru.

Postup pre dekompresiu:

Pre dekompresiu nám postačia zakódované dáta. Pri čítaní sa postupne vytvára tabuľka, ktorá začína na indexe 1.

Index	Kódované slovo
1	k
2	d
3	o
4	on
5	onn

Napr. po prečítaní dvojice <3, n> sa do tabuľky doplnilo kódové slovo na indexe 3 + znak "n", teda "on". Po načítaní <4, n> sa do tabuľky doplní slovo z indexu 4 („on") + znak „n", teda „onn". Takto sme dekódovali celý reťazec.

Lempel – Ziv - Welch kódovanie (LZW – kódovanie)

- **základným princípom je kódovať celú sekvenciu znakov (nie iba znakov jednotlivých) jediným kódom**
- **často sa opakujúce sekvencie znakov (bez ohľadu na ich dĺžku) možno kódovať relatívne malým počtom bitov**
- **algoritmus LZW - kódovania využíva tzv. slovník (V), ktorý uchováva sekvencie znakov dynamicky vyberané z kódovaného textu**
- **každej sekvencii znakov q slovník priraďuje jednoznačný kód kód(q) ≡ #(q), všetky kódy majú rovnakú bitovou dĺžku (napr. 12 bitov – max. dĺžka slovníka je 2^{12} = 4096 sekvencií znakov)**
- **slovník je inicializovaný vložením všetkých možných jednoznakových sekvencií - teda každému znaku $c_i \in E$ je priradený kód #(c_i)**
- **jedno - priechodový typ kódovania**

Kódovací algoritmus – i - tý krok:

a) **je určený najdlhší aktuálny prefix P_i nachádzajúci sa na začiatku textu E, ktorý sa už nachádza v slovníku V**
b) **ak i > 1 tak polož q = P_{i-1} + E[1] (prvý nezakódovaný znak z E), inak (ide o inicializáciu) polož q = *e* (prázdny reťazec)**
c) **ak nie je q ešte v slovníku V tak je tam vložené a je mu priradený „najbližší“ voľný kód**
d) **do výstupu je zapísaný kód #(P_i)**
e) **P_i je odobraté z E**

Pre implementáciu (dynamického) slovníka je vhodné využiť štruktúru *znakového stromu*
S LZW - kódovaným textom je uložený iba inicializačný slovník (teda nie celý slovník, ktorý bol vybudovaný v priebehu kódovania daného textu).

Príklad:
LZW - kódovanie textu S = ’mame kokosy a ananasy’ po vykonanej inicializácii slovníku.

i-tý KROK	#(P_i) (výstup kódu)	P_{i-1} + E[1] (vlož do slovníka)	E (pred vykonaním i-tého kroku)
1	#(’m’)	-	’mame□kokosy□a□ananasy
2	#(’a’)	’ma’	’ame□kokosy□a□ananasy’
3	#(’m’)	’am’	’me□kokosy□a□ananasy’
4	#(’e’)	’me’	’e□kokosy□a□ananasy’
5	#(’□’)	’e□’	’□kokosy□a□ananasy’
6	#(’k’)	’□k’	’kokosy□a□ananasy’
7	#(’o’)	’ko’	’okosy□a□ananasy’
8	#(’ko’)	’ok’	’kosy□a□ananasy’
9	#(’s’)	’kos’	’sy□a□ananasy’
10	#(’y’)	’sy’	’y□a□ananasy’
11	#(’□’)	’y□’	’□a□ananasy’
12	#(’a’)	’□a’	’a□ananasy’
13	#(’□a’)	’a□’	’□ananasy’
14	#(’n’)	’□an’	’nanasy’
15	#(’a’)	’na’	’anasy’
16	#(’na’)	’an’	’nasy’
17	#(’s’)	’nas’	’sy’
18	#(’y’)	’sy’	’y’

Dekódovanie:

- **rovnaké ako kódovanie**
- **reprezentácia obsahuje iba postupnosť bitových kódov rovnakej dĺžky => ľahké dekódovať kód po kóde**
- **k dekódovaniu nepotrebujeme mať zapamätaný celý slovník, iba inicializačný, pretože ho priebežne pri dekódovaní budujeme rovnako ako pri kódovaní**

-

Algoritmus dekódovania:

- **prečítaj kód**
- **vyhľadaj ho v slovníku a daj na výstup odpovedajúcu sekvenciu znakov z V**
- **vlož do slovníka V predchádzajúcu sekvenciu rozšírenú o prvý znak aktuálnej sekvencie**

Iný príklad na zakódovanie a dekódovanie textu:

Pomocou 4 bitového LZW kódovania zakódujeme text: „azszsszsazszsb“

Najskôr vytvoríme kódy pre všetky znaky nachádzajúce sa v texte. Napríklad: #(’a’) = 0000, #(’b’) = 0001, #(’s’) = 0010, #(’z’) = 0011. Tieto kódy tvoria inicializačný slovník a iba tento malý slovník je potrebný na neskoršie dekódovanie celého textu.

Potom nájdeme v slovníku najdlhší možný kód pre znaky, ktoré sú na začiatku doteraz nezakódovanej časti. V našom prípade je to kód pre „a“.

Aktuálny výstup: 0000

Aktuálny stav kódovania: „azszsszsazszsb“ (červená časť už je zakódovaná vo výstupe).

Postup opakuje až pokým nie je zakódovaný celý text.

V slovníku nájdeme kód pre „z“.
Aktuálny výstup: 0000|0011
Do slovníka pridáme minule zapísaný reťazec „a“ spolu s ďalším nezakódovaným znakom. Vložíme tam teda #(’az’) = 0100.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „s“.
Aktuálny výstup: 0000|0011|0010
Do slovníka vložíme kód pre #(’zs’) = 0101.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „zs“.
Aktuálny výstup: 0000|0011|0010|0101
Do slovníka vložíme kód pre #(’sz’) = 0110.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „sz“.
Aktuálny výstup: 0000|0011|0010|0101|0110
Do slovníka vložíme kód pre #(’zss’) = 0111.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „s“.
Aktuálny výstup: 0000|0011|0010|0101|0110|0010
Do slovníka vložíme kód pre #(’szs’) = 1000.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „az“.
Aktuálny výstup: 0000|0011|0010|0101|0110|0010|0100
Do slovníka vložíme kód pre #(’sa’) = 1001.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „szs“.
Aktuálny výstup: 0000|0011|0010|0101|0110|0010|0100|1000
Do slovníka vložíme kód pre #(’azs’) = 1010.
Aktuálny stav kódovania: „azszsszsazszsb“

V slovníku nájdeme kód pre „b“.
Definitívny výstup:
0000|0011|0010|0101|0110|0010|0100|1000|0001
Do slovníka vložíme kód pre #(’szsb’) = 1011.

Ak predpokladáme 8 bitovú dĺžku znakov, tak pôvodná veľkosť textu bola 15 bajtov. Po zakódovaní má text veľkosť 5 bajtov (potrebujeme 36 bitov). To znamená, že zakódovaný text zaberá len 30% pôvodnej veľkosti. K zakódovanému textu, je ale potrebné pridať aj inicializačný slovník.

Pri dekódovaní postupujeme obdobne. Máme inicializačný slovník, ktorý je nutný pre správne dekódovanie.

#(’a’) = 0000, #(’b’) = 0001, #(’s’) = 0010, #(’z’) = 0011

Vezmeme inicializačný slovník a nájdeme v ňom kód pre prvý zakódovaný reťazec, teda „a“.

Aktuálny zakódovaný reťazec:
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a

Dekódujeme ďalší znak, teda „z“.
Do slovníka pridáme minule dekódovaný reťazec + prvý znak aktuálne dekódovaného reťazca, teda #(’az’) = 0100.
Aktuálny zakódovaný reťazec:
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z

V slovníku nájdeme kód pre 0010 - „s“.
Do slovníka vložíme kód pre #(’zs’) = 0101.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z|s

V slovníku nájdeme kód pre 0101 - „zs“.
Do slovníka vložíme kód pre #(’sz’) = 0110.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z|s|zs

V slovníku nájdeme kód pre 0110 - „sz“.
Do slovníka vložíme kód pre #(’zss’) = 0111.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z|s|zs|sz

V slovníku nájdeme kód pre 0010 - „s“.
Do slovníka vložíme kód pre #(’szs’) = 1000.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z|s|zs|sz|s

V slovníku nájdeme kód pre 0100 - „az“.
Do slovníka vložíme kód pre #(’sa’) = 1001.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z|s|zs|sz|s|az

V slovníku nájdeme kód pre 1000 - „szs“.
Do slovníka vložíme kód pre #(’azs’) = 1010.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Aktuálny výstup: a|z|s|zs|sz|s|az|szs

V slovníku nájdeme kód pre 0001 - „b“.
Do slovníka vložíme kód pre #(’szsb’) = 1011.
0000|0011|0010|0101|0110|0010|0100|1000|0001
Definitívny výstup: a|z|s|zs|sz|s|az|szs|b

Problémom LZW – kódovania je možnosť vyčerpania všetkých kódov.

Riešenie problému vyčerpania všetkých kódov – príklady:

- **ukončenie vkladania do slovníka (ďalšie kódovanie využije raz vybudovaný a celkom zaplnený slovník)**
- **odobratie najmenej frekventovaných sekvencií zo slovníka**
- **zväčšenie bitovej veľkosti kódov**

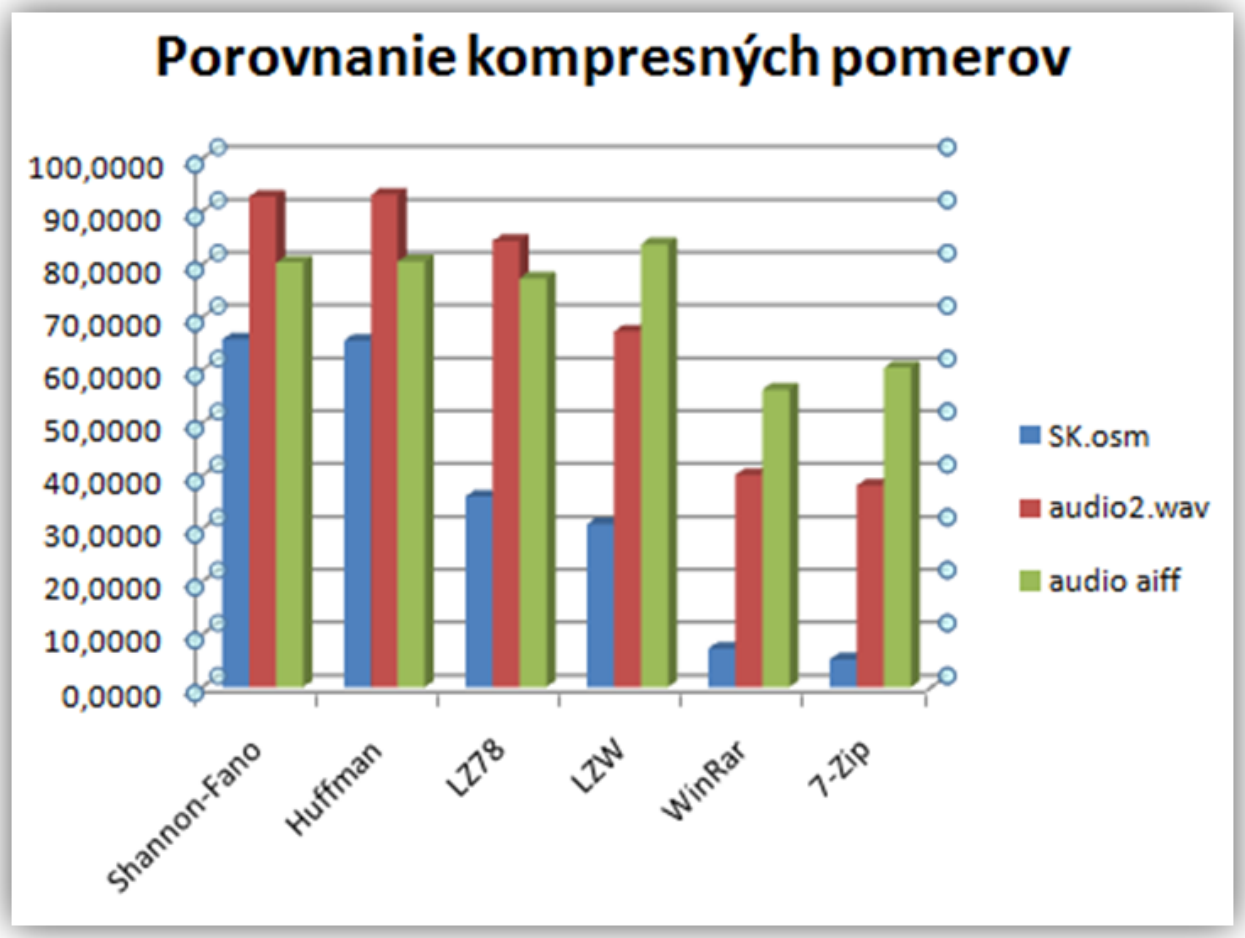
Vlastnosti:

- **LZW kód je typicky menší ako Huffmanov kód**
- **je vždy jednopriechodový**
- **súčasťou kódovanej reprezentácie nie je žiadna väčšia dodatočná štruktúra**

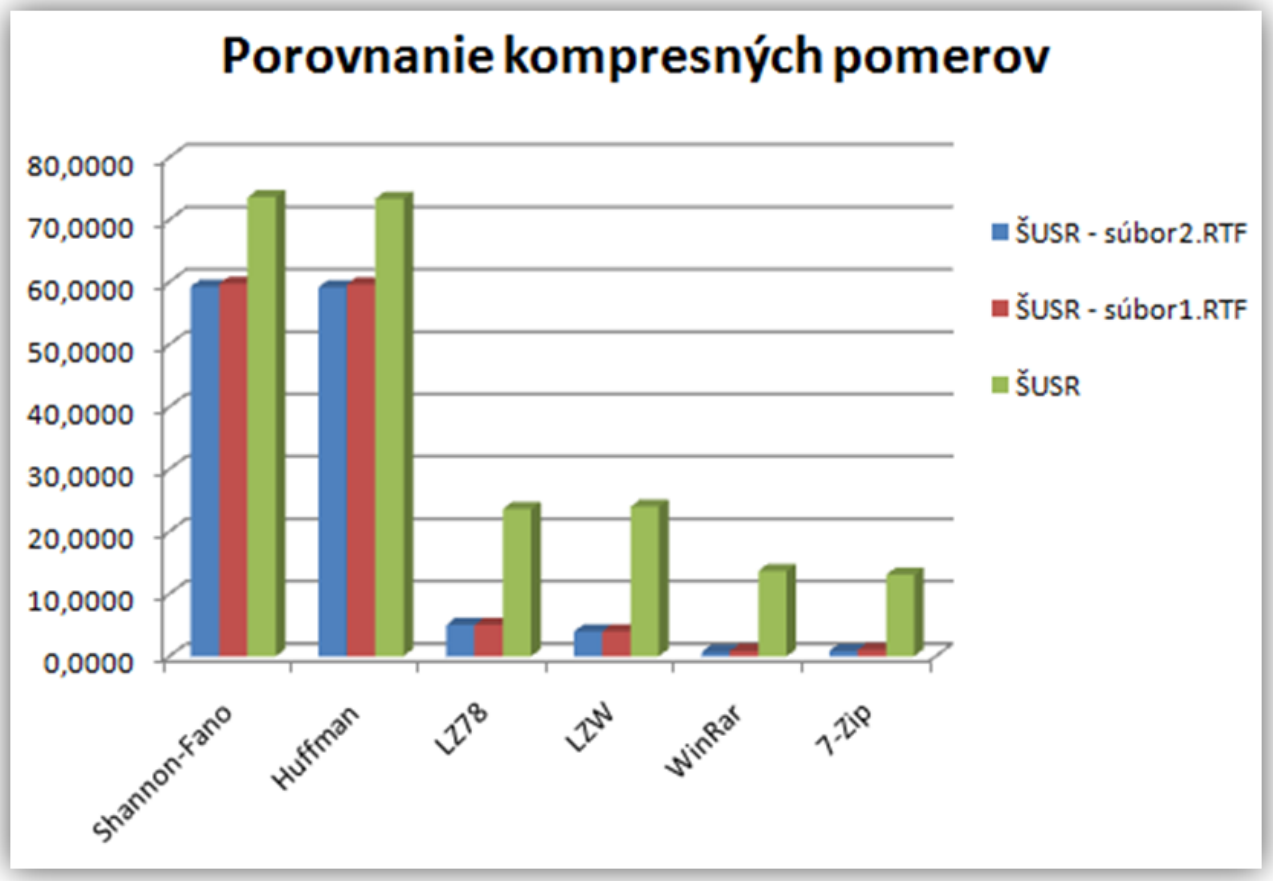
Lempel – Ziv – Welch kódovanie ako aj Huffmanovo kódovanie má mnoho variantov. Napriek jednoduchým algoritmom je implementácia pomerne náročná na dobre zvolené pomocné štruktúry. Napríklad rýchlosť LZW – kódovania závisí najmä od rýchlosti akou dokážeme zistiť aký najdlhší reťazec P_i ešte nezakódovaného textu E v slovníku.

Experimetnálne porovnanie kódovacích algoritmov

V bakalárskej práci boli implementované 4 kódovania. Vzniknutá aplikácia je schopná skomprimovať vybraný súbor, prípadne adresár rovnako ako iné komerčné aplikácie. Pri testovaní kompresie v aplikáciách WinRar a 7-Zip bola vždy nastavená najsilnejšia kompresia.



Ako sa ukázalo aj pri iných testoch, veľmi dobrý kompresný pomer na xml súboroch (SK.osm) dosahuje LZW kódovanie, naopak má slabšie výsledky pri kódovaní nekomprimovaného zvuku (.wav).



Pri kompresií textového súboru sa opäť ako dobré riešenie ukazujú kódovania LZW a LZ78. Môžeme si všimnúť, že v tomto prípade je dosiahnutý kompresný pomer aj v porovnaní s inými aplikáciami „primerane dobrý“.

Zdroj: Činčala P.: Kompresia dát (2017)