KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

KOMPRESE DAT

ARNOŠT VEČERKA

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2008

1. Základní pojmy

V této části jsou uvedeny některé pojmy, které jsou používány v popisech teoretických základů komprese a popisech jednotlivých kompresních metod.

Kód, **kódování** - je způsob reprezentace dat, informací a hodnot při jejich uložení v paměti, v souboru, při přenosu po síti atd. Příkladem může být ASCII kód, který písmena a jiné znaky reprezentuje jako čísla uložená na jednom bytu.

Znak - je základní jednotka dat. V teorii komprese se pojmem znak označuje nejen znak textu, ale obecně libovolná hodnota uložená na jednom bytu.

Text - tento pojem se v teorii komprese používá pro označení jakýchkoliv výchozích dat, která jsou komprimována. Nejde v tomto případě výlučně o textovou informaci.

Kompresní poměr - je poměr délky zkomprimovaných dat vzhledem k délce původních dat. Kompresní poměr se často se vyjadřuje v procentech.

$$\text{Kompresní poměr} = \frac{\text{Délka po kompresi}}{\text{Délka před kompresí}}$$

Jiný způsob vyjádření kompresního poměru je v počtu bitů na byte (bpb - bits per byte), tj. kolik je při kompresi v průměru zapotřebí bitů pro uložení jednoho bytu výchozího souboru. Například kompresní poměr 0.4 odpovídá kompresní hodnotě 3.2 bpb, neboť jeden byte má 8 bitů a při kompresi na $\frac{4}{10}$ délky původního souboru dat je v průměru zapotřebí 3.2 bitů pro uložení hodnoty 1 bytu původního souboru.

Bezeztrátová komprese - je způsob komprese, při které nedochází ke ztrátě informace. Při dekompresi dostáváme stejná data, jaká jsme komprimovali. Všechny základní kompresní metody jsou bezeztrátové.

Ztrátová komprese - je způsob komprese, při které jsou výchozí hodnoty poněkud pozměněny nebo některé méně významné hodnoty jsou zanedbány, aby se dosáhlo vyššího kompresního poměru. Dekompresí dostáváme v tomto případě poněkud jiné hodnoty, než byly původně komprimovány. Ztrátové metody mají uplatnění v kompresi obrazu a zvuku.

Místo termínu bezeztrátová komprese se také používají označení přesná komprese nebo vratná komprese a naopak ztrátová komprese bývá označována jako nepřesná nebo nevratná komprese.

1.1 Entropie

Všechny kompresní metody vycházejí ze skutečnosti, že běžně používané způsoby reprezentace dat jsou navrženy tak, aby umožňovaly snadnou manipulaci s daty a v důsledku toho obsahují řadu redundancí. Metody kódování používané pro uložení dat neberou ohled na skutečnost, že v datech se mnohdy některé části opakují nebo některé hodnoty se vyskytují ve větší míře než jiné. Důsledkem toho je skutečnost, že pro uložení dat se používá větší prostor, než je nezbytně nutné.

Příklad. Snad nejjednodušším příkladem neúsporného kódování je uložení zdrojových textů programů, které vesměs používají jen znaky ze základní části tabulky ASCII. Znaky v základní části ASCII tabulky jsou vyjádřeny hodnotami v rozsahu 0-127 a pro jejich uložení vystačíme se 7 bity. Přesto jsou znaky zdrojového textu běžně ukládány na jednom bytu, ve kterém jeden bit zůstává takto nevyužit. Triviální kompresí, při které vynecháme nepoužívaný bit, dosáhneme kompresní poměr $\frac{7}{8}$.

Obecně je komprese takový způsob kódování, který odstraňuje opakující se výskyty a redundance v datech a tím snižuje nároky na prostor, který je potřebný pro jejich uložení. Z pohledu komprese nás tedy zajímá, zda délka dat je odpovídající vzhledem k množství informace obsažené v datech. Nemáme ale žádný algoritmus, který by dokázal přesně množství informace v datech zjistit. Nicméně určitý způsob, jak zjistit míru informace, zde existuje. Je to entropie používaná v teorii přenosu zpráv.

Její zápis uvedený v následujících odstavcích je přizpůsoben tak, aby entropie byla vyjádřena v bitech, neboť ty jsou základní jednotkou informace používanou při kódování dat.

Definice entropie: Nechť data se skládají z *n* různých prvků

$$a_1, a_2, a_3, ..., a_n$$

a tyto prvky se v datech vyskytují s pravděpodobnostmi

$$p_1, p_2, p_3, ..., p_n \quad .$$

Pak množství informace, která je reprezentována prvkem a_i, udává jeho entropie

$$E_i = -\log_2(p_i) \quad \text{bitů} \quad .$$

Entropie E_i vyjadřuje, jak velkou informaci nese výskyt prvku a_i. Je zřejmé, že čím je pravděpodobnost výskytu prvku a_i menší, tím je jeho entropie a tedy i míra informace větší.

Příklad. Koupíme si los, na který lze vyhrát televizor nebo video kazetu. Pravděpodobnosti jednotlivých výher jsou:

– pravděpodobnost výhry televizoru je 0.001

– pravděpodobnost výhry video kazety je 0.06

– zbývá případ, že nevyhrajeme nic, jeho pravděpodobnost je 0.939

Vypočítáme entropie uvedených jevů:

$$E_{televizor} = -\log_2 0.001 \cong 13.3$$
$$E_{kazeta} = -\log_2 0.06 \cong 3.6$$
$$E_{nic} = -\log_2 0.939 \cong 0.1$$

Jestliže po slosování zjistíme, že jsme nic nevyhráli, má to pro nás poměrně nízkou informační hodnotu (entropie této informace je jen 0.1), neboť bylo značně pravděpodobné, že to tak dopadne.

Naopak zpráva, že jsme vyhráli televizor, je značně překvapivá. Hodnota této informace je poměrně vysoká (její entropie je 13.3).

Entropie jevu je tím větší, čím je nižší pravděpodobnost, že tento jev nastane. Je to přirozené, neboť informace, že jev nastal, je tím významnější, čím je výskyt jevu řidší.

Vedle entropie jednotlivých prvků můžeme spočítat průměrnou, tj. střední entropii *E*. Tu dostaneme tak, že entropie jednotlivých prvků vynásobíme pravděpodobnostmi, s jakou se tyto prvky vyskytují, a tyto hodnoty sečteme:

$$E = p_1E_1 + p_2E_2 + ... + p_nE_n \quad \text{bitů} \quad .$$

Po dosazení za jednotlivé entropie E_i dostaneme pro střední entropii výraz:

$$E = -\sum_{i=1}^{n} p_i \log_2(p_i) \quad \text{bitů} \quad .$$

Zavedli jsem si entropii jako míru informace obsaženou v prvku dat. Doposud jsem se ale nezabývali otázkou, co považovat za prvek dat. Nejjednodušší je jako prvek dat zvolit znak, tj. hodnotu uloženou na jednom bytu. Mohli bychom ale například u textových souborů za prvky považovat slabiky nebo jiné části slov atd.

Entropie nám při kódování znaků umožňuje zjistit, do jaké míry je použitý kód optimální z pohledu dosažení co nejkratšího kódování. Uveďme jednoduchý příklad.

Příklad. Pro jednoduchost uvažujme, že data jsou složena jen ze čtyř písmen a,b,c,d. Pokud bychom chtěli taková data uložit, je nepřirozenější kódovat každý znak dvěma bity dle tabulky:

Znak:	a	b	c	d
Kód:	00	01	10	11

Předpokládejme, že všechny znaky se v textu vyskytují se stejnou pravděpodobností, tj. s pravděpodobností 0.25. To například splňuje text *abcddbac*. Tento text obsahuje 8 znaků a dle tabulky ho zakódujeme jako posloupnost 16 bitů 0001101111010010.

Vypočítáme nyní střední entropii:

$$E = -\sum_{i=1}^{4} 0.25 * \log_2 0.25 = -4 * 0.25 * \log_2 2^{-2} = 2 \text{ bity} \quad .$$

K zakódování jednoho znaku jsou zapotřebí průměrně 2 bity, což znamená, že kód použitý pro daný text *abcddbac* je optimální. Vezměme nyní případ, kdy znaky se v textu vyskytují s různou pravděpodobností:

Znak:	a	b	c	d
Pravděpodobnost výskytu:	0.5	0.25	0.125	0.125

Nyní výskyt písmena *a* je v textu 2× vyšší než písmena *b* a 4× vyšší než písmena *c* nebo *d*. Například tuto vlastnost má text *badacaab*. Střední entropie je v tomto případě:

$$E = -0.5 * \log_2 0.5 - 0.25 * \log_2 0.25 - 2 * 0.125 * \log_2 0.125 =$$

$$-\frac{1}{2}\log_2 2^{-1} - \frac{1}{4}\log_2 2^{-2} - \frac{2}{8}\log_2 2^{-3} = 1.75 \text{ bitů} \quad .$$

V tomto textu je již určitá pravidelnost - některé znaky se vyskytují častěji. Kód s 2 bity na znak již není optimální. Dle entropie by mělo stačit 1,75 bitu na znak. Takovým kódem je například:

Znak:	a	b	c	d
Kód:	0	10	110	111

Jestliže tento kód použijeme na text *badacaab*, dostaneme 10011101100010. Text v délce 8 znaků je nyní zakódován pomocí 14 bitů, což odpovídá průměrné hodnotě 1,75 bitu na znak.

Poznámka: Je přirozené, že kód musíme zvolit tak, abychom byli schopni text dekódovat. Kódy použité v předchozím příkladu tuto vlastnost splňují. Jsou to prefixové kódy, jejichž základní vlastností je, že žádný prefix kódu jednoho znaku není kódem jiného znaku. Prefixem přitom rozumíme jakoukoliv část kódu vzatou zleva. Následující obrázek ukazuje všechny možné prefixy kódu znaku *c*:

110 - kód znaku c
1 - prefix délky 1
11 - prefix délky 2
110 - prefix délky 3

Je zřejmé, že žádný z prefixů 1, 11 a 110 není kódem jiného znaku.

Prefixový kód má tu výhodu, že umožňuje text snadno dekódovat zleva doprava. Jednoduše se odebírají bity tak dlouho, dokud nedostaneme kód některého znaku, čímž je znak dekódován. Tímto způsobem dekódujeme jeden znak za druhým. Ukažme postup dekódování textu z předchozího příkladu.

Vstup	Bity odebírané ze vstupu	Dekódovaný znak
10011101100010	1	
0011101100010	10	b
011101100010	0	a
11101100010	1	
1101100010	11	
101100010	111	d
atd.		

1.2 Typy kompresních metod

Ve skriptu jsou popsány tyto základních kategorie kompresních metod:

1. **Statistické metody komprese** - jsou založeny na pravděpodobnosti, s jakou se vyskytují jednotlivé znaky v textu. Jednoduchý příklad statistické komprese byl uveden v předchozí části.
2. **Slovníkové metody komprese** - principem těchto metod je vyhledání opakujících se částí textu. Do zkomprimovaného textu se uloží jen první výskyt takové části. Všechny další výskyty jsou nahrazeny odkazem na předchozí výskyt.
3. **Komprese nepohyblivého obrazu** - tato část je věnována ztrátové kompresi rastrového obrazu, které poskytující vysoký kompresní poměr.
4. **Komprese pohyblivého obrazu** - zde jde o popis ztrátové metody komprese s vysokým kompresním poměrem.
5. **Komprese zvuku** – obsahuje popis principů používaných při ztrátové kompresi zvuku.

2. Statistické metody komprese

2.1 Huffmanovo kódování

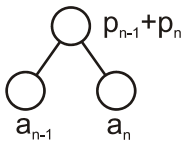
Huffmanovo kódování patří k nejstarším kompresním metodám (pochází z roku 1952). Jde o prefixový kód vytvořený podle jednoduché strategie. Znakům, které se v textu vyskytují často (s vysokou četností), se přiděluje kód s malým počtem bitů a naopak znaky, které se v textu vyskytují zřídka (s malou četností), se kódují delším kódem. Nejjednodušší postup sestavení Huffmanova kódu je pomocí binárního stromu.

Konstrukce Huffmanova kódu

Předpokládejme, že text se skládá z *n* znaků a_1, a_2, a_3, ..., a_n, a nechť tyto znaky jsou seřazeny tak, že pravděpodobnosti jejich výskytů p_1, p_2, p_3, ..., p_n tvoří nerostoucí posloupnost (tj. $p_i \geq p_{i+1}$ pro i=1, 2, ..., n-1).

Vytvoření binární stromu a sestavení Huffmanova kódu:

1. *První krok:* Z posledních dvou znaků posloupnosti a_{n-1} a a_n vytvoříme strom, jehož listy ohodnotíme těmito znaky a jehož kořen ohodnotíme součtem jejich pravděpodobností. Znaky a_{n-1} a a_n pak odstraníme z posloupnosti.

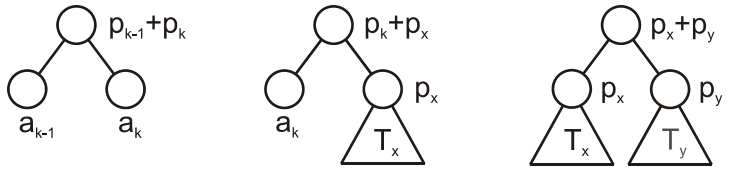


2. *Obecný krok:* Nechť ze znaků a_{k+1}, a_{k+2} ,..., a_n jsme již sestrojili jistý počet binárních stromů a v posloupnosti tedy zbývají ještě znaky a_1, a_2, ..., a_k.

   Z hodnot, které se nacházejí na konci posloupnosti pravděpodobností zbývajících znaků p_1, p_2, ..., p_k, a dále z hodnot v kořenech binárních stromů vybereme nejnižší dvě hodnoty. V případě, že existuje více dvojic splňující tuto podmínku, vybereme libovolnou z nich.

   Mohou nastat tři situace:

   a) Vybrali jsme hodnoty p_{k-1} a p_k z konce posloupnosti. Z nich sestrojíme strom dle obrázku a) a znaky a_{k-1} a a_k odstraníme z posloupnosti. Kořenu stromu přiřadíme hodnotu $p_{k-1}+p_k$.



a) b) c)

   b) Vybrali jsme hodnotu p_k z konce posloupnosti a hodnotu p_x v kořenu některého stromu T_x. Z nich sestrojíme nový strom dle obrázku b) a znak a_k odebereme z konce posloupnosti. Kořenu stromu přiřadíme hodnotu p_k+p_x, což je pravděpodobnost, s jakou se v textu vyskytuje dohromady znak a_k a znaky obsažené ve stromu T_x.

   c) Vybrali jsem dvě hodnoty p_x a p_y v kořenech stromů T_x a T_y. Z těchto dvou stromů sestrojíme jeden strom dle obrázku c). Jeho kořenu přiřadíme hodnotu p_x+p_y, což je pravděpodobnost, s jakou se v textu vyskytují dohromady znaky obsažené v obou stromech T_x a T_y.

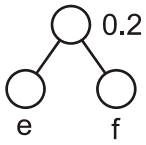
3. Krok 2 opakujeme tak dlouho, dokud není vyčerpána celá posloupnost znaků a všechny binární stromy nejsou spojeny v jeden strom.

4. *Sestavení kódu:* Nyní postupně projdeme všechny uzly stromu vyjma listových a v každém uzlu ohodnotíme jednu z hran vycházejících z uzlu číslicí 0 a druhou číslicí 1 (na pořadí nezáleží). Tím je konstrukce binárního stromu ukončena. Huffmanův kód každého znaku dostaneme jako posloupnost číslic 0 a 1 na cestě od kořene stromu k listovému uzlu s tímto znakem.

Příklad. Nechť text se skládá z 6 znaků *a*,*b*,*c*,*d*,*e*,*f*, které se v textu vyskytují s pravděpodobnostmi:

Znak:	a	b	c	d	e	f
Pravděpodobnost výskytu:	0.3	0.2	0.2	0.1	0.1	0.1

V prvním kroku sestrojíme strom ze znaků *e* a *f*.



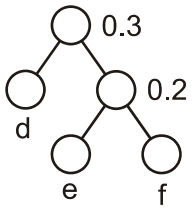
V posloupnosti zůstávají znaky *a*,*b*,*c*,*d*.

V následujícím kroku vybereme dvě nejnižší pravděpodobnosti z hodnot na konci zbývající části posloupnosti pravděpodobností 0.3, 0.2, 0.2, 0.1 a hodnoty 0.2 obsažené v kořenu binárního stromu sestaveného v prvním kroku. Zde máme dvě možnosti:

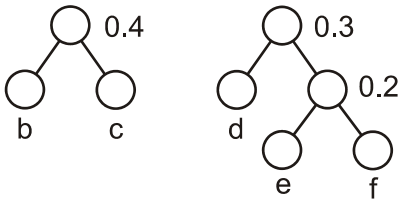
a) Můžeme vybrat pravděpodobnost 0.2 znaku *c* a pravděpodobnost 0.1 znaku *d*.

b) Můžeme vybrat pravděpodobnost 0.1 znaku *d* a pravděpodobnost 0.2 v kořenu stromu.

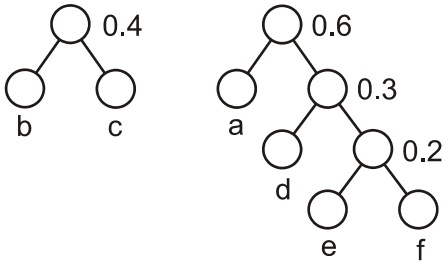
Zvolme třeba možnost b). Strom nyní vypadá:



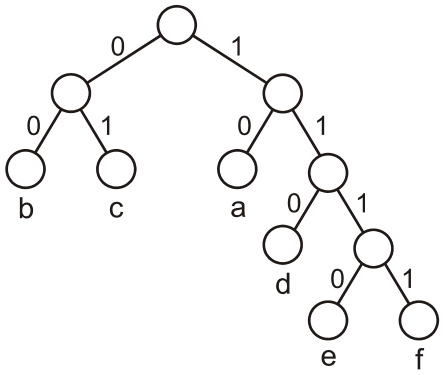
Nyní v posloupnost zbývají znaky *a*,*b*,*c*. Dvojice nejnižších pravděpodobností je v tomto kroku určena jednoznačně, jsou to pravděpodobnosti 0.2 a 0.2 znaků *b* a *c*. Z nich sestrojíme další strom:



V posloupnosti zbývá již jen znak *a*. Dvojici nejnižších pravděpodobností nyní tvoří pravděpodobnost 0.3 znaku *a* a pravděpodobnost 0.3 v kořenu pravého stromu.



Posloupnost je vyčerpána a máme dva stromy. Jejich spojením dokončíme konstrukci binárního stromu a nakonec hrany stromu ohodnotíme číslicemi 0 a 1 třeba tak, že levou hranu vycházející z uzlu k jeho následníku ohodnotíme číslicí 0 a pravou hranu číslicí 1.



Průchodem od kořene k jednotlivým listům stromu dostaneme Huffmanův kód pro jednotlivé znaky:

Znak:	a	b	c	d	e	f
Kód:	10	00	01	110	1110	1111

Spočítejme, kolik bitů bude obsahovat kódování textu délky 10 znaků *aaabbccdef*:

$$3*2 + 2*2 + 2*2 + 3 + 4 + 4 = 25 \text{ bitů} .$$

Odtud dostáváme střední délku kódování 2.5 bitů/znak. Pro srovnání spočítejme střední entropii:

$$E = -0.3*\log_2 0.3 - 2*0.2*\log_2 0.2 - 3*0.1*\log_2 0.1 = 2.446 .$$

Střední entropie v předchozím příkladu je menší než průměrná střední délka kódování jednoho znaku. Tedy skutečná účinnost sestaveného kódu je nižší, než předpovídá entropie. Snížení účinnosti je způsobeno rozdílem mezi entropií jednotlivých znaků a skutečnou délkou jejich kódování. Například entropie znaku *a* je

$$E_a = -\log_2 0.3 = 1.74 \text{ bitů} ,$$

ale tento znak je ve skutečnosti kódován 2 bity. Uvedený rozdíl je dán charakterem Huffmanova kódování. Každý znak je kódován odděleně a jeho kód musí mít celistvý počet bitů. Tento nedostatek nemá aritmetické kódování, které je popsáno v následující části.

Poznámka: Při praktickém použití Huffmanova kódování pravděpodobnosti znaků většinou počítáme pomocí zjištění počtu výskytů (frekvencí) znaků v textu. V tomto případě je mnohem výhodnější pracovat přímo s frekvencemi, které jsou vyjádřeny celými čísly, než používat pravděpodobnosti. Protože pravděpodobnosti jsou přímo úměrné frekvencím, lze snadno při konstrukci Huffmanova kódu pravděpodobnosti nahradit frekvencemi. Součtům pravděpodobností přitom odpovídají součty frekvencí a minimálním pravděpodobnostem odpovídají minimální frekvence.

2.2 Aritmetické kódování

Aritmetické kódování celý text kóduje jedním číslem z intervalu $\langle 0,1)$. Uveďme nejprve postup. Vycházíme z obdobných předpokladů jako u Huffmanova kódování: text je složen ze znaků $a_1, a_2, ..., a_n$, které se v něm vyskytují s pravděpodobnostmi $p_1, p_2, ..., p_n$. Uspořádání znaků může být libovolné.

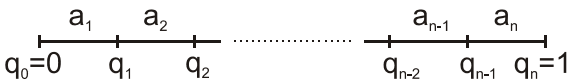
Principem aritmetického kódování je rozdělení intervalu $\langle 0,1)$ na *n* disjunktních intervalů. Velikosti jednotlivých intervalů jsou určené pravděpodobnostmi výskytů znaků a jsou dány předpisem:

$$\langle 0,p_1), \langle p_1,p_1+p_2), \langle p_1+p_2,p_1+p_2+p_3), ..., \langle p_1+p_2+ ... +p_{n-1},p_1+p_2+ ... +p_{n-1}+p_n)$$

Abychom zápis intervalů zjednodušili, zavedeme si kumulativní pravděpodobnosti:

$q_0=0,\ q_1=p_1,\ q_2=p_1+p_2,\ \ldots\ ,\ q_n=p_1+p_2+\ldots+p_n$

S jejich použitím interval, kterým je znak a_i v aritmetickém kódování reprezentován, zapíšeme jako $\langle q_{i-1},q_i)$. Vzájemné přiřazení intervalů a znaků ukazuje souhrnně následující obrázek.



Obrázek 1 Rozdělení intervalu při aritmetickém kódování

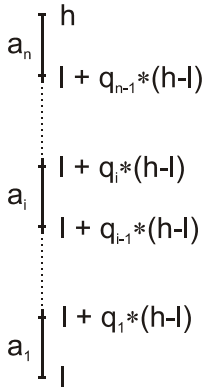
Už jsme uvedli, že výsledkem aritmetického kódování je číslo z intervalu $\langle 0,1)$. V průběhu aritmetického kódování se nepočítá přímo tato hodnota, ale postupně se vymezuje (zmenšuje) interval, ve kterém výsledná hodnota leží. V následujícím popisu postupu kódování označíme tento průběžný interval písmenem *I*.

Postup aritmetického kódování:

1. Na počátku hodnotu intervalu *I* položíme $I=\langle 0,1)$.
2. *Průběžný krok, kódování jednotlivých znaků textu:* Ze vstupu v každém kroku odebereme jeden znak. Vezmeme jemu odpovídající interval $\langle q_{i-1},q_i)$ a ze stávající hodnoty intervalu $I=\langle l,h)$ vypočítáme jeho novou hodnotu:

   $$I = \langle l+q_{i-1}*(h-l),\ l+q_i*(h-l))$$

   Tím se jako nový interval vybere z původního intervalu *I* ta jeho část, která odpovídá kódovanému znaku a_i , jak ukazuje následující obrázek



Obrázek 2 Výpočet nové hodnoty intervalu *I* při kódování znaku

3. Krok 2 opakujeme do vyčerpání vstupního textu.

Příklad. Mějme text *bdaccaccdc*. Tento text se skládá ze 4 znaků *a*,*b*,*c*,*d*, jejichž pravděpodobnosti výskytu jsou:

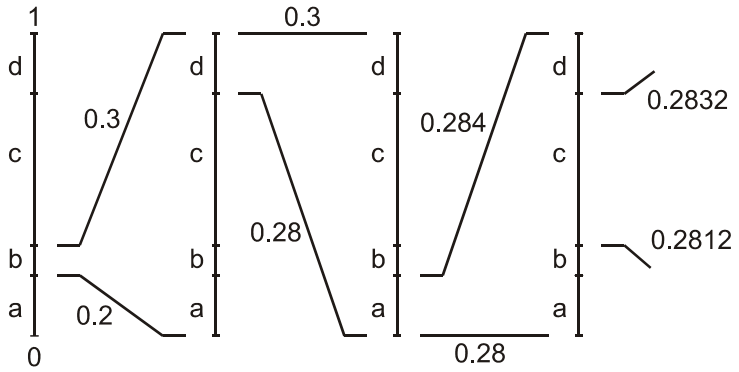
Znak:	a	b	c	d
Pravděpodobnost výskytu:	0.2	0.1	0.5	0.2
Odpovídající interval:	⟨0, 0.2)	⟨0.2, 0.3)	⟨0.3, 0.8)	⟨0.8, 1)

Ukažme si kódování prvních čtyř znaků *bdac*:

Vstupní znak	Vypočítaná hodnota intervalu *I*
b	$\langle 0+0.2*1,\ 0+0.3*1) = \langle 0.2,\ 0.3)$
d	$\langle 0.2+0.8*0.1,\ 0.2+1*0.1) = \langle 0.28,\ 0.3)$
a	$\langle 0.28+0*0.2,\ 0.28+0.2*0.2) = \langle 0.28,\ 0.284)$

c $\langle 0.28+0.3*0.004,\ 0.28+0.8*0.004) = \langle 0.2812,\ 0.2832)$

Grafické znázornění výpočtu intervalu *I* je na následujícím obrázku:



Výsledkem aritmetického kódování je libovolné číslo *c* obsažené ve vypočítaném intervalu *I*. Například jako výsledek kódování v předchozím příkladu, kde jsme dostali interval I=⟨0.2812,0.2832), můžeme zvolit číslo c = 0.282.

Dekódování probíhá obdobným způsobem jako kódování a znaky se dekódují ve stejném pořadí, jak byly kódovány. Vycházíme opět z rozdělení intervalu *I* na *n* částí podle pravděpodobností výskytu jednotlivých znaků. V každém kroku najdeme, v kterém z těchto *n* intervalů se nachází hodnota *c*. Tím je dekódován příslušný znak. Z něho určíme novou hodnotu intervalu *I* stejným způsobem jako při kódování a pokračujeme v dekódování dalšího znaku.

Postup dekódování.

1. Na počátku hodnotu intervalu *I* položíme I=⟨0,1).
2. *Dekódování znaku:* Vyjdeme ze současné hodnoty intervalu I=⟨l,h) a zakódované hodnoty *c*. Najdeme index *i*, pro který platí

   $l + q_{i-1}*(h-l) \le c < l + q_i*(h-l)$,

   nebo-li

   $$q_{i-1} \le \frac{c-l}{h-l} < q_i$$

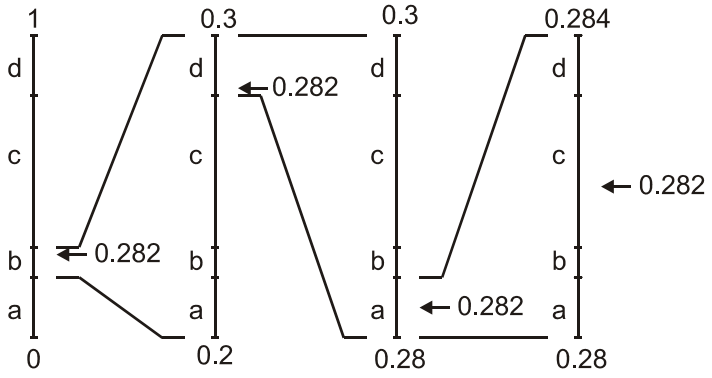
   Dle nalezeného indexu *i* dekódujeme jemu odpovídající znak a_i a vypočítáme novou hodnotu intervalu *I*:

   $l = \langle l+q_{i-1}*(h-l),\ l+q_i*(h-l))$
3. Krok 2 opakujeme, dokud není dekódován celý text.

Příklad. Ukažme si dekódování hodnoty aritmetického kódování c=0.282 vypočítané v předchozím příkladu:

Interval *I*	$\frac{c-l}{h-l}$	Nalezený interval $\langle q_{i-1}, q_i)$	Dekódovaný znak	Nová hodnota intervalu *I*
⟨0, 1)	0.282	⟨0.2, 0.3)	b	⟨0+0.2∗1, 0+0.3∗1) = ⟨0.2, 0.3)
⟨0.2, 0.3)	0.82	⟨0.8, 1)	d	⟨0.2+0.8∗0.1, 0.2+1∗0.1) = ⟨0.28, 0.3)
⟨0.28, 0.3)	0.1	⟨0, 0.2)	a	⟨0.28+0∗0.2, 0.28+0.2∗0.2) = ⟨0.28, 0.284)
⟨0.28, 0.284)	0.5	⟨0.3, 0.8)	c	

Graficky je postup dekódování znázorněn na dalším obrázku:



Z hodnoty aritmetického kódování *c* nelze určit, kdy máme dekódování ukončit, tj. kolik znaků bylo zakódováno. Buďto při kódování k zakódovanému textem přidáme údaj o délce textu nebo ke kódovaným znakům přidáme speciální znak, který bude označovat konec textu. Označme tento znak *KT* (Konec textu). Text nyní bude složen z *n+1* znaků

$a_1, a_2, ..., a_n, KT$.

Pravděpodobnosti použité při jejich kódování budou

$p_1', p_2', ..., p_n', p_{KT}$,

kde hodnotu pravděpodobnosti p_{KT} zvolíme velmi malou, neboť znak *KT* bude kódován jen jednou na konci textu. Hodnoty ostatních pravděpodobností p_1', p_2', ..., p_n' získáme snížením původních pravděpodobností $p_1, p_2,..., p_n$ tak, aby byla zachována podmínka

$p_1' + p_2' + ... + p_n' + p_{KT} = 1$.

Např. je můžeme vypočítat dle vztahu $p_i' = p_i * (1 - p_{KT})$.

Aritmetické kódování používá desetinná čísla, kterými jsou vyjádřeny meze intervalu. S každým zakódovaným znakem se velikost intervalu zmenší. Čím je interval menší, tím delší jsou čísla v jeho zápisu. Princip komprese pomocí aritmetického kódování spočívá ve zmenšování intervalu podle toho, jaké jsou pravděpodobnosti výskytů jednotlivých znaků. Při kódování znaků, které se v textu vyskytují často, se délka intervalu zmenšuje méně a tím i méně roste délka čísla, které určuje aritmetický kód. Naproti tomu kódování znaků s malou pravděpodobností výskytu naopak způsobí výraznější zmenšení délky intervalu a tím výraznější zvětšení délky čísla. Malá pravděpodobnost ale znamená, že tyto znak se v textu vyskytují zřídka a jejich celkový vliv na délku zakódovaného čísla je tím i malý. Nicméně v každé případě je zřejmé, že s rostoucí délkou kódovaného textu roste i délka čísel vyjadřující meze intervalu. Problémy spojené s uložením tak dlouhých čísel a obtížné provádění operací s desetinnými čísly s tak velkým počtem míst činí aritmetické kódování v této podobě pro praxi nepoužitelné. Proto byly vyvinuty metody aritmetického kódování s vlastnostmi:

- používají jen celá čísla, což je výhodnější, než použití desetinných čísel
- pracují s čísly s omezeným počtem míst (za cenu určitého zhoršení kompresního poměru, které je ale zanedbatelné).

Uveďme nyní popis praktické implementace aritmetického kódování.

Aritmetické kódování s použitím celých čísel

Místo intervalu I=⟨0,1) použijeme celočíselný interval I=⟨0,M⟩. Zde je účelné použít takový rozsah hodnot, jaký poskytují běžné datové typy programovacích jazyků. To jsou dnes 32-bitová čísla. Maximální hodnota 32-bitového čísla bez znaménka je 2^{32}-1. Při výpočtu ale může být k rozsahu intervalu M přičtena 1, což by u hodnoty 2^{32}-1 vedlo k přetečení, proto jako horní mez použitého intervalu zvolíme číslo o jeden bit menší, tedy hodnotu $M=2^{31}$-1.

Abychom se zcela vyhnuli výpočtům s desetinnými čísly, místo pravděpodobností použijeme četnosti výskytů znaků (frekvence). Pro ně zavedeme označení

$$f_1, f_2, ..., f_n \quad ,$$

kde běžně počet znaků je n=256, tj. všech 256 možných hodnot, které mohou být uloženy v jednom bytu.

Pro zjednodušení zápisu si zavedeme kumulativní četnosti:

$$g_0=0,\ g_1=f_1,\ g_2=f_1+f_2,\ ...\ ,\ g_n=f_1+f_2+...+f_n \quad .$$

Je zřejmé, že g_n je kumulativní četnosti všech znaků v textu a pravděpodobnosti výskytů znaků v textu lze napsat vztahy

$$p_1 = \frac{f_1}{g_n},\ \ldots.\ ,\ p_n = \frac{f_n}{g_n} \quad .$$

Nyní můžeme přejít k popisu principu celočíselného aritmetického kódování:

1. Na počátku hodnotu intervalu *I* položíme I=⟨0,M⟩.
2. Při kódování každého znaku vyjdeme ze stávající hodnoty intervalu I=⟨l,h⟩. Interval *I* rozložíme na disjunktní celočíselné intervaly podle frekvencí jednotlivý znaků:

Znak:	Jemu odpovídající interval:
a_1	$I_1 = \langle l+s*g_0,\ l+s*g_1-1\rangle = \langle l,\ l+s*g_1-1\rangle$
....	
a_i	$I_i = \langle l+s*g_{i-1},\ l+s*g_i-1\rangle$
....	
a_n	$I_n = \langle l+s*g_{n-1},\ l+s*g_n-1\rangle$

kde

$$s = \frac{(h-l+1)}{g_n} \quad ,$$

přičemž zápis zlomku reprezentuje celočíselné dělení.

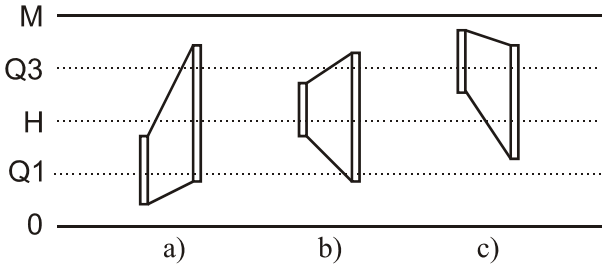
Velikost kumulativních frekvencí znaků je omezena nejen rozsahem zvoleného datového typu (32 bitů), ale i skutečností, že je zapotřebí s nimi provádět určité aritmetické operace. Proto hodnotu g_n omezíme na maximální rozsah 29 bitů, což znamená $g_n \le 2^{29}-1$. Pokud při kompresi nastane případ, že kumulativní četnost překročí tuto hodnotu, řeší se to snížením hodnot četností. Všechny četnosti, které jsou větší než 1, se dělí hodnotou 2. Po snížení hodnot jednotlivých četností se provede přepočet kumulativních četností.

Kódování znaku a_i znamená výpočet nové hodnoty intervalu *I* dle vzorců uvedených v předchozí tabulce. Vhledem k tomu, že délka intervalu se při kódování neustále zmenšuje, interval by se po určité době stal tak malým, že výpočet by přestal být korektní. Aby se tomu zabránilo, rozsah intervalu *I* je při kódování průběžně zvětšován a na výstup se o tom zasílá informace ve formě binární hodnoty (tj. 0 nebo 1). Při úpravě velikosti intervalu *I* se uvažuje stávající rozsah intervalu *I* vzhledem k hodnotám:

$Q1 = \frac{M+1}{4} = 2^{29}$	Čtvrtina maximálního rozsahu
$H = 2*Q1 = 2^{30}$	Polovina maximálního rozsahu
$Q3 = 3*Q1 = 3*2^{29}$	Tři čtvrtiny maximálního rozsahu

Zvětšení rozsahu intervalu *I* se provádí v následujících třech případech:

E1) Interval *I* je v dolní polovině zvoleného rozsahu čísla, tj. $I \subseteq \langle 0,H-1\rangle$, nebo-li $h < H$. Na výstup se zašle hodnota 0 a interval *I* se nahradí novým intervalem $I = \langle 2*l, 2*h+1\rangle$. Tento případ je znázorněn na následujícím obrázku jako a).



Obrázek 3 Jednotlivé případy zvětšení intervalu

E2) Interval *I* je v horní polovině zvoleného rozsahu čísla, tj. $I \subseteq \langle H,M\rangle$, nebo-li $l \geq H$. Na výstup se zašle hodnota 1 a interval *I* se nahradí novým intervalem $I = \langle 2*(l-H), 2*(h-H)+1\rangle$. Tento případ je na obrázku znázorněn jako c).

E3) Interval *I* je ve střední oblasti zvoleného rozsahu čísla, tj. $I \subseteq \langle Q1,Q3-1\rangle$, nebo-li $l \geq Q1$ a $h < Q3$. Interval *I* se nahradí novým intervalem $I = \langle 2*(l-Q1), 2*(h-Q1)+1\rangle$. Na obrázku je to případ b).

Bezprostřední kódování změny na výstup v případě E3 již nelze provést. Nicméně i v tomto případě existuje možnost, jak tuto změnu kódovat, její odvození je ale poněkud komplikovanější.

Nejprve uvažujme, jak se interval *I* změní, jestliže stejnou změnu provedeme k-krát za sebou. Hodnota intervalu *I* bude:

$$I_{k\times..} = \langle 2^k*l-2*(2^k-1)*X,\ 2^k*h-2*(2^k-1)*X+2^k-1\rangle \quad (x)$$

kde X=0 pro případ E1, X=H pro případ E2 a X=Q1 pro E3.

Dokažme indukcí platnost vztahu (x) třeba pro případ E2:

1. Pro k=1:

   $$I_{1\times E2} = \langle 2*l-2*H,\ 2*h-2*H+1\rangle \text{ ... platí}$$

2. Nechť (x) platí pro k-1:

   $$I_{(k-1)\times E2} = \langle 2^{k-1}*l-2*(2^{k-1}-1)*H,\ 2^{k-1}*h-2*(2^{k-1}-1)*H+2^{k-1}-1\rangle$$

   Dosazením do vztahu pro případ E2 dostaneme pro k:

   $$I_{k\times E2} = \langle 2*(2^{k-1}*l-2*(2^{k-1}-1)*H-H),\ 2*(2^{k-1}*h-2*(2^{k-1}-1)*H+2^{k-1}-1-H)+1\rangle$$

   Převedení tohoto vztahu na tvar (x) je už jen záležitostí kratší úpravy.

Naprosto stejným způsobem se dokáže platnost (x) pro zbývající případ E1 a E3.

Nyní předpokládejme, že k-krát nastal případ E3:

$$I_{k\times E3} = \langle 2^k*l-2*(2^k-1)*Q1,\ 2^k*h-2*(2^k-1)*Q1+2^k-1\rangle$$

a po něm následoval případ E2:

$$I_{k\times E3,1\times E2} = \langle 2*(2^k*l-2*(2^k-1)*Q1-H),\ 2*(2^k*h-2*(2^k-1)*Q1+2^k-1-H)+1\rangle$$

Dosazením H=2*Q1 a úpravou dostaneme

$$I_{k\times E3,1\times E2} = \langle 2^{k+1}*l-2^{k+1}*H,\ 2^{k+1}*h-2^{k+1}*H+2^{k+1}-1\rangle$$

Nyní vezmeme situaci, kdy nastane případ E2 a po něm k-krát případ E1

$$I_{1\times E2,k\times E1} = \langle 2^k*(2*l-2*H),\ 2^k*(2*h-2*H+1)+2^k-1\rangle$$

Je zřejmé, že výsledný interval je stejný.

Obdobně se dá ukázat, že sekvence k-krát případ E3 a pak případ E1 je shodná se sekvencí případ E1 a pak k-krát případ E2. Tedy pro kódování E3 a E1, E2 platí ekvivalence

$(E3)^k\,E1 = E1\,(E2)^k$ a $(E3)^k\,E2 = E2\,(E1)^k$.

Odtud už snadno odvodíme postup, jak kódovat případ E3. Zavedeme si čítač případů E3, který na počátku bude mít nulovou hodnotu. Čítač při každém výskytu případu E3 zvýšíme o hodnotu 1. Jakmile nastane případ E1 nebo E2 zakódujeme příslušnou hodnotu 0 nebo 1 a za ní tolik opačných hodnot (1 nebo 0), kolik je v čítači E3 evidováno výskytů případu E3. Čítač E3 pak vynulujeme.

Na závěr uveďme prakticky použitelné algoritmy pro aritmetické kódování a dekódování.

Algoritmus kódování:

1. Na začátku předpokládáme, že máme zadané četnosti výskytu jednotlivých znaků:

   $f_1, f_2, \ldots, f_n$.

   Z nich vypočítáme hodnoty kumulativních četností:

   $g_0 = 0,\ g_1 = f_1,\ \ldots\ ,\ g_n = f_1+\ldots+f_n$.

   Inicializujeme čítač E3=0 a nastavíme počáteční hodnotu intervalu $I=\langle 0,M\rangle$, kde $M=2^{31}-1$.

2. *Kódování znaku:* Přečteme ze vstupu znak a_i a vypočítáme novou hodnotu intervalu *I*:

   $$I = \langle l+s*g_{i-1},\ l+s*g_i-1\rangle,\ \text{kde}\ s = \frac{(h-l+1)}{g_n} \quad .$$

3. Jestliže se po zakódování znaku interval *I* zmenšil tak, že nastal některý z případů E1, E2 nebo E3 provedeme zvětšení intervalu *I* postupem:
   - V případě E1 zvětšíme již popsaným postupem hodnotu intervalu na $I=\langle 2*l,\ 2*h+1\rangle$ a na výstup zapíšeme hodnotu 0 a za ní tolik 1, kolik bylo evidováno výskytů případu E3 v čítači E3. Čítač E3 následně vynulujeme.
   - V případě E2 zvětšíme již popsaným postupem hodnotu intervalu *I* a na výstup kódujeme 1 a za ní tolik hodnot 0, jaká je hodnota čítače E3. Čítač E3 následně vynulujeme.
   - V případě E3 zvětšíme již popsaným postupem hodnotu intervalu *I* a hodnotu čítače E3 zvětšíme o 1.

   Při každém provedení kroku 3 se velikost intervalu *I* přibližně zdvojnásobí, nicméně může stále zůstávat malá. Proto krok 3 opakujeme tak dlouho, dokud pro interval *I* nastává některý z uvedených případů E1 až E3.

4. Kroky 2. – 3. opakujeme do zakódování celého textu.

Princip aritmetického kódování s použitím omezeného rozsahu čísel se výrazněji neliší od původního postupu kódování. Jestliže nastane případ, že horní i dolní mez intervalu má na nejvyšším místě (v binárním vyjádření) hodnotu 0, je zřejmé, že dalším zmenšováním intervalu při kódování dalších znaků se tato hodnota již nemůže změnit a je tedy poslána na výstup. Obdobně, je-li na nejvyšším místě dolní i horní meze intervalu hodnota 1, nemůže se již ani tato změnit a lze ji okamžitě poslat na výstup. Proto ani algoritmus dekódování se výrazněji neliší.

Algoritmus dekódování:

1. Počáteční hodnoty kumulativních četností $g_0, \ldots, g_n$ a interval *I* jsou stejné jako při kódování. Hodnota aritmetického kódu se v průběhu dekódování používá ve stejné délce, jakou používá interval *I*, tj. jako 31-bitové číslo bez znaménka. Označme proměnnou s hodnotou kódu písmenem *v*. Na začátku do proměnné *v* vložíme prvních 31 bitů aritmetického kódu:

   v ← prvních 31 bitů aritmetického kódu .

2. *Dekódování znaku:* Musíme najít interval, který obsahuje hodnotu aritmetického kódu *v*. Tedy musíme najít takovou hodnotu indexu *i*, pro kterou je splněno:

$$l+s*g_{i-1} \le v \le l+s*g_i-1, \text{ kde } s = \frac{(h-l+1)}{g_n}$$

Upravíme

$$l+s*g_{i-1} \le v < l+s*g_i$$

a dále

$$g_{i-1} \le \frac{v-l}{s} < g_i$$

Při jejím použití stačí vypočítat kumulativní četnost *c* odpovídající hodnotě kódu *v:*

$$c = \frac{v-l}{s}$$

a vyhledat, do kterého intervalu kumulativních četností $\langle g_{i-1},g_i)$ hodnota *c* patří. Tj. najít index *i*, pro který platí

$$g_{i-1} \le c < g_i$$

Tím je dekódován znak a_i původního textu. Následně zcela stejným způsobem jako při kódování vypočítáme novou hodnotu intervalu *I*:

$$I = \langle l+s*g_{i-1}, l+s*g_i-1\rangle, \text{ kde } s = \frac{(h-l+1)}{g_n}$$

3. Jestliže se interval *I* zmenšil tak, že nastal některý z případů E1, E2 nebo E3 provedeme zvětšení intervalu *I* postupem:

   V případě E1 zvětšíme již popsaným postupem hodnotu intervalu na $I=\langle 2*l, 2*h+1\rangle$ a dále vypočítáme novou hodnotu kódu *v* dle vztahu:

   $v = 2*v +$ další bit aritmetického kódu .

   což znamená, že hodnotu *v* posuneme o 1 bit doleva a na poslední uvolněný bit dáme další bit ze vstupní hodnoty.

   V případě E2 zvětšíme již popsaným postupem hodnotu intervalu na $I=\langle 2*(l-H), 2*(h-H)+1\rangle$ a vypočítáme novou hodnotu kódu *v*:

   $v = 2*(v-H) +$ další bit aritmetického kódu .

   V případě E3 zvětšíme již popsaným postupem hodnotu intervalu na $I=\langle 2*(l-Q1),2*(h-Q1)+1\rangle$ a vypočítáme novou hodnotu kódu *v*:

   $v = 2*(v-Q1) +$ další bit aritmetického kódu .

   Krok 3 vždy opakujeme tak dlouho, dokud pro interval *I* nastává některý z uvedených případů E1, E2 nebo E3.

Příklad. Pro jednoduchost zvolíme pro vyjádření intervalu *I* číslo v délce pouze 6 bitů. Maximální hodnota M a hodnoty Q1, H a Q3 pro 6-bitové číslo bez znaménka jsou:

M = 63	Maximální hodnota
Q1 = 16	čtvrtina rozsahu
H = 32	polovina rozsahu
Q3 = 48	tři čtvrtiny rozsahu

Kódovaný text bude *baacb* .Text je složen ze 3 znaků *a*,*b*,*c*. Četnosti jsou

$f_1 = 2, f_2 = 2, f_3 = 1$

a kumulativní četnosti jsou

$g_1 = 2, g_2 = 4, g_3 = 5$

Následující tabulka ukazuje jednotlivé kroky při kódování.

Činnost	Interval *I*	Hodnota *s*	Čítač E3	Výstup kódu
Inicializace hodnot	⟨0,63⟩		0	
Kódování prvního znaku – *b*	⟨24,47⟩	12		
Zvětšení intervalu - případ E3	⟨16,63⟩		1	
Kódování druhého znaku – *a*	⟨16,33⟩	9		
Zvětšení intervalu - případ E3	⟨0,35⟩		2	
Kódování třetího znaku – *a*	⟨0,13⟩	7		
Zvětšení intervalu - případ E1	⟨0,27⟩		0	011
Zvětšení intervalu - případ E1	⟨0,55⟩			
Kódování čtvrtého znaku – *c*	⟨44,54⟩	11		
Zvětšení intervalu - případ E2	⟨24,45⟩			1
Zvětšení intervalu - případ E3	⟨16,59⟩		1	
Kódování pátého znaku – b	⟨32,47⟩	8		
Zvětšení intervalu - případ E2	⟨0,31⟩		0	10
Zvětšení intervalu - případ E1	⟨0,63⟩			0

Výsledkem aritmetické kódování řetězce *baacb* je 01101100 .

Další část příkladu ukazuje zpětné dekódování:

Činnost	Interval *I*	Hodnota *v*	Zbývající bity kódu	Hodnota *c*	Dekódovaný znak
Inicializace hodnot	⟨0,63⟩	27	00		
Dekódování znaku	⟨24,47⟩			2	b
Zvětšení intervalu – případ E3	⟨16,63⟩	22	0		
Dekódování znaku	⟨16,33⟩			0	a
Zvětšení intervalu – případ E3	⟨0,35⟩	12			
Dekódování znaku	⟨0,13⟩			1	a
Zvětšení intervalu – případ E1	⟨0,27⟩	24			
Zvětšení intervalu – případ E1	⟨0,55⟩	48			
Dekódování znaku	⟨44,54⟩			4	c
Zvětšení intervalu - případ E2	⟨24,45⟩	32			
Zvětšení intervalu – případ E3	⟨16,59⟩	32			
Dekódování znaku	⟨32,47⟩			2	b

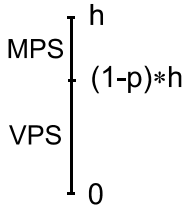
Z aritmetického kódu 01101100 je na začátku dekódování jeho prvních šest bitů 011011 vloženo do proměnné *v*. Zbývající dva bity 00 zůstávají na vstupu. Ty jsou při prvních dvou zvětšení intervalu *I* odebrány k výpočtu další hodnoty *v*. V posledních krocích při zvětšení intervalu *I* je vstup již prázdný. Další hodnoty *v* se v tomto případě počítají, jako kdyby na vstupu byla nula, tj. podle vztahu:

$$v = 2*v.$$

Q-kódování

Vedle popsaného postupu existuje ještě další způsob aritmetického kódování, který je velmi jednoduchý a rychlý. Místo po znacích znaků kóduje text po bitech, tedy jeho vstupem jsou číslice 0 a 1. Tyto dvě číslice rozděluje na méně pravděpodobnou a více pravděpodobnou. Méně pravděpodobná číslice (budeme ji označovat MPS – méně pravděpodobný symbol) má pravděpodobnost výskytu p, kde $p \in (0,0.5\rangle$, více pravděpodobná (tu budeme označovat VPS – více pravděpodobný symbol) má pak pravděpodobnost výskytu $1-p$.

Při kódování opět dochází ke zmenšováním intervalu I, který na začátku je opět $I=\langle 0,1)$. Při kódování je ale zmenšován tak, že dolní mez zůstává stále 0. Čímž průběžná hodnota intervalu je $I=\langle 0,h)$. Při kódování je interval rozdělen na dvě části. Spodní je pro VPS, horní je pro MPS.



Obrázek 4 Rozdělení intervalu při kódování binární hodnoty

Při kódování VPS je interval $\langle 0,h)$ nahrazen podintervalem $\langle 0,(1-p)*h)$. Při kódování MPS je odpovídající podinterval $\langle (1-p)*h,h)$ upraven tak, že od obou jeho mezí je odečtena hodnota $(1-p)*h$:

$$\langle (1-p)*h - (1-p)*h,\ h - (1-p)*h) = \langle 0,p*h) \quad .$$

Hodnota aritmetického kódu c se počítá průběžně během kódování. Na začátku je kód roven nule. Při kódování VPS se jeho hodnota nemění. Při kódování MPS se k ní přičítá hodnota $(1-p)*h$. Kódování tedy probíhá podle pravidel:

Kódování VPS: c – nezměněn $\quad \langle 0,h) \rightarrow \langle 0,(1-p)*h)$

Kódování MPS: $c \rightarrow c+(1-p)*h \quad \langle 0,h) \rightarrow \langle 0,p*h)$

Algoritmus kódování:

1. Na počátku hodnotu intervalu I položíme $I=\langle 0,1)$. Počáteční hodnotu kódu položíme $c=0$. Stanovíme, který symbol je méně pravděpodobný (MPS), a stanovíme jeho pravděpodobnost p.
2. *Průběžný krok, kódování jednotlivých binárních číslic:* Ze vstupu v každém kroku odebereme binární číslici.

   Je-li tato VPS, vypočítáme novou hodnotu horní meze intervalu

   $$h = (1-p)*h \quad .$$

   Je-li tato MPS vypočítáme novou hodnotu kódu a novou hodnotu horní meze intervalu

   $$c = c+(1-p)*h$$
   $$h = p*h \quad .$$

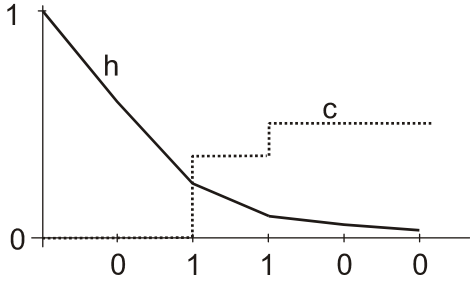
3. Krok 2 opakujeme do vyčerpání vstupního textu.

Příklad. Budeme kódovat binární číslo 01100. MPS je v tomto případě číslice 1, její pravděpodobnost výskytu v čísle je p=0.4.

Vstupní číslice	Hodnota kódu c	Horní mez intervalu h
0	0	$0.6*1 = 0.6$
1	$0+0.6*0.6 = 0.36$	$0.4*0.6 = 0.24$
1	$0.36+0.6*0.24 = 0.504$	$0.4*0.24 = 0.096$

0	0.504	0.6∗0.096 = 0.0576
0	0.504	0.6∗0.0576 = 0.03456

Graficky je postup kódování znázorněn na dalším obrázku:



Dekódování probíhá obráceně než kódování.

Algoritmus dekódování:

1. Na počátku hodnotu intervalu *I* položíme I=⟨0,1). Zadáme MPS a jeho pravděpodobnost *p*.
2. *Průběžný krok, dekódování jednotlivých binárních číslic:* Srovnáme hodnoty c a (1-p)∗h.

   Jestliže platí c<(1-p)∗h:

   dekódujeme VPS
   vypočítáme h = (1-p)∗h

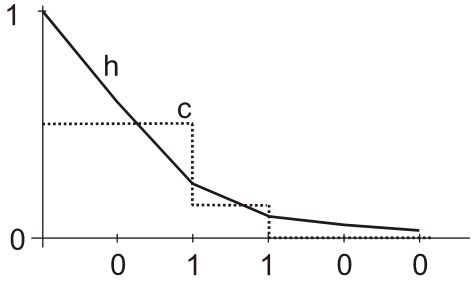
   V opačném případě (c>= (1-p)∗h):

   dekódujeme MPS
   vypočítáme c = c-(1-p)∗h, h = p∗h

Příklad. Budeme dekódovat kód c=0.65184 z minulého příkladu.

Horní mez intervalu *h*	Kód *c*	(1-p)∗h	Dekódovaná číslice	Nová hodnota kódu *c*	Nová hodnota meze intervalu *h*
1	0.504	0.6	0		0.6∗1 = 0.6
0.6	0.504	0.36	1	0.504-0.36 = 0.144	0.4∗0.6 = 0.24
0.24	0.144	0.144	1	0.144-0.144 = 0	0.4∗0.24 = 0.096
0.096	0	0.0576	0		0.6∗0.096 = 0.0576
0.0576	0	0.03456			0.6∗0.0576 = 0.03456

Graficky je postup dekódování znázorněn na dalším obrázku:



I v případě Q-kodéru se v praxi používá celočíselná varianta, která má označení QM-kodér a je popsána v následující části.

QM-kodér

QM-kodér používá běžně pro horní mez intervalu 16-bitové číslo. Jeho maximální hodnota je 2^{16}-1 = $FFFF_{16}$. Vzhledem k tomu, že horní mez je v určitých okamžicích kódování zdvojnásobována a tato hodnota by přitom překročena, je zvoleno, aby horní mez místo intervalu ⟨0,1) odpovídala přibližně intervalu ⟨0,1.5), tedy je zvolena korespondence mezi intervaly

$$\langle 0, 1.5) \approx \langle 0, 2^{16}\text{-}1),$$

nebo-li $1.5 \approx 2^{16}$-1. Odtud

$0.75 \approx (2^{16}\text{-}1)/2$. I když při celočíselném dělení je $(2^{16}\text{-}1)/2 = 2^{15}\text{-}1 = 777F_{16}$, zvolíme místo hodnoty $777F_{16}$ hodnotu o 1 vyšší, tj. 8000_{16}, která je výhodnější pro operace.

$1 = (0.75*4)/3 \approx (8000_{16}*4)/3 = AAAA_{16}$

$0.5 = 1/2 \approx AAAA_{16}/2 = 5555_{16}$.

Odtud dostáváme vzájemný vztah intervalů

$$\langle 0, 1) \approx \langle 0, AAAA_{16}) \ .$$

Pravděpodobnost MPS je rovněž vyjádřena celým číslem tak, že p=1 odpovídá opět hodnota $AAAA_{16}$.

Při popisu jednotlivých metod aritmetického kódování jsme doposud pro zjednodušení popisovaných algoritmů předpokládali, že pravděpodobnosti výskytu jednotlivých znaků nebo symbolů jsou stanoveny na začátku a v průběhu kódování zůstávají nezměněny. QM-kodér pravděpodobnost pro zakódování každého symbolu stanoví podle toho, jaké symboly byly obsaženy v určitém počtu naposledy kódovaných symbolů. Čímž během kódování dochází i k vzájemné výměně přiřazení MPS a VPS, jestliže v uvažované části předtím kódovaného textu začne symbol, který je určen jako MPS, převládat nad symbolem VPS, čímž pravděpodobnost MPS počítaná z daného počtu posledních symbolů překročí limitní hodnotu 0.5 .

QM-kodér používá pro kódování statisticky předem vypočtenou tabulku pravděpodobností, což velmi zrychluje kódování, neboť tím se v průběhu kódování výrazně zredukuje množství potřebných výpočtů. Jedna z takových tabulek je uvedena níže. Tabulky používané v praxi bývají ještě o něco podrobnější (mají o něco více řádku), což zvyšuje účinnost kódování

Na začátku je index aktuálního řádku nastaven na 0 a je zvolen MPS. Při kódování každého symbolu se vychází z aktuálního řádku. Na něm se najde index následujícího řádku ve sloupci pro další VP, je-li kódovaný symbol VPS, anebo ve sloupci pro další MP, je-li kódovaný symbol MPS. Navíc je-li aktuální řádkem řádek s indexem 0 a právě je kódován MPS, dojde k vzájemné výměně MPS a VPS, jak ukazuje poslední sloupec tabulky.

index	p - hexa	p - dekad	další VP	další MP	MP výměna
0	5608	0.5041	1	0	1
1	5408	0.4924	2	0	0
2	5008	0.4689	3	1	0
3	4808	0.4221	4	2	0
4	3808	0.3283	5	3	0
5	3408	0.3049	6	4	0
6	3008	0.2814	7	5	0
7	2808	0.2346	8	5	0
8	2408	0.2111	9	6	0
9	2208	0.1994	10	7	0
10	1C08	0.1643	11	8	0
11	1808	0.1408	12	9	0
12	1608	0.1291	13	10	0
13	1408	0.1174	14	11	0

14	1208	0.1057	15	12	0
15	0C08	0.0705	16	13	0
16	0908	0.0530	17	14	0
17	0708	0.0412	18	15	0
18	0508	0.0293	19	16	0
19	0388	0.0207	20	17	0
20	02C8	0.0163	21	18	0
21	0298	0.0152	22	19	0
22	0138	0.0071	23	20	0
23	00b8	0.0042	24	21	0
24	0098	0.0039	25	21	0
25	0058	0.0020	26	23	0
26	0038	0.0013	27	23	0
27	0028	0.0009	28	25	0
28	0018	0.0006	29	25	0
29	0008	0.0002	29	27	0

Ve druhém sloupci tabulky jsou uvedeny pravděpodobnosti MPS odpovídající jednotlivým řádkům v již uvedené celočíselné reprezentaci. Ve třetím sloupci jsou tyto pravděpodobnosti pro větší přehlednost rovněž vyjádřeny běžným způsobem jako desetinné číslo.

Je zřejmé, že i zde podobně jako u již uvedeného aritmetického kódování s celými čísly dochází k postupnému zmenšování intervalu. Řeší se to zde, pokud hodnota horní meze intervalu poklesne pod 8000_{16}, vynásobením horní meze a zároveň počítaného kódu dvěma.

Algoritmus kódování:

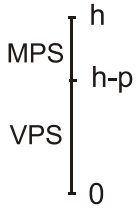
1. Na počátku hodnotu intervalu *I* položíme I=⟨0, $AAAA_{16}$).

   Počáteční hodnotu aritmetického kódu položíme c=0. Zavedeme čítač *b* počtu bitů kódu *c*. Na začátku ho nastavíme na b=15.

   Hodnotu indexu aktuálního řádku na začátku položíme r=0.

   Stanovíme, který symbol bude považován za méně pravděpodobný (MPS).

2. *Průběžný krok, kódování jednotlivých binárních číslic:* Ze vstupu odebereme binární číslici. Stávající interval I=<0,h> rozdělíme na dvě části. Dolní bude odpovídat kódování VPS a horní MPS.



   Je-li vstupní číslice VPS:

   - vypočítáme novou hodnotu horní meze intervalu – vezmeme dolní část rozděleného intervalu I

     $h = h-p$.

   - do indexu řádku *r* dáme novou hodnotu indexu uvedenou ve sloupci „další VP".
   - Je-li h<8000_{16}, pak zdvojnásobíme horní mez intervalu spolu s kódem a zvýšíme čítač počtu bitů kódu

     $h = 2*h$, $c = 2*c$, $b = b+1$.

   Je-li kódovaná číslice MPS

- vypočítáme novou hodnotu kódu a novou hodnotu horní meze intervalu – vezmeme horní část rozděleného intervalu

  $c = c+(h-p)$, $h = p$.

- Je-li v posledním sloupci „MP výměna“ hodnota 1, zaměníme MPS a VPS a do indexu řádku *r* dáme novou hodnotu indexu uvedenou ve sloupci „další VP”.

  Není-li v posledním sloupci „MP výměna“ hodnota 1, do indexu řádku *r* dáme novou hodnotu indexu uvedenou ve sloupci „další MP”.

- Následně zdvojnásobujeme horní mez intervalu spolu s kódem

  $h = 2*h$, $c = 2*c$, $b = b+1$,

  dokud nezačne platit $h \geq 8000_{16}$.

3. Krok 2 opakujeme do vyčerpání vstupního textu.

Dekódování je opět obráceným postupem ke kódování.

Algoritmus dekódování:

1. Na počátku hodnotu intervalu *I* položíme $I=\langle 0, AAAA_{16})$.

   Do proměnné *v* dáme prvních 15 bitů kódu *c*. Hodnotu *b* snížíme o počet bitů vložených do proměnné *v*, tj. b=b-15.

   Hodnotu indexu aktuálního řádku na začátku položíme r=0.

2. *Průběžný krok, dekódování jednotlivých binárních číslic:* Srovnáme hodnoty v a h-p.

   Jestliže je v<h-p:

   - dekódujeme VPS
   - vypočítáme novou hodnotu meze intervalu

     $h = h-p$.

   - do indexu řádku *r* dáme novou hodnotu indexu uvedenou ve sloupci „další VP”.
   - Je-li $h<8000_{16}$, pak zdvojnásobíme horní mez intervalu spolu s kódem, ke kterému přidáme další bit kódu *c*:

     $h = 2*h$, $v = 2*v$ + další bit kódu c .

   V opačném případě, tj. platí-li v>= h-p:

   - dekódujeme MPS
   - Vypočítáme novou hodnotu *v* a novou hodnotu horní meze intervalu

     $v = v-(h-p)$, $h = p$.

   - Je-li v posledním sloupci „MP výměna“ hodnota 1, zaměníme MPS a VPS a do indexu řádku *r* dáme novou hodnotu indexu uvedenou ve sloupci „další VP”.

     Není-li v posledním sloupci „MP výměna“ hodnota 1, do indexu řádku *r* dáme novou hodnotu indexu uvedenou ve sloupci „další MP”.

   - Následně zdvojnásobujeme horní mez intervalu spolu s kódem, ke kterému vždy přidáme další bit kódu *c*:

     $h = 2*h$, $v = 2*v$ + další bit kódu c .

     dokud nezačne platit $h \geq 8000_{16}$.

Příklad. Kódujeme binární číslo 0111. Následující tabulka ukazuje jednotlivé kroky při kódování.

Index	MPS	Horní mez	Kód	Čítač

Činnost	řádku *r*		intervalu *h*	*c*	*b*
Inicializace hodnot	0	1	AAAA	0	15
Kódování první číslice – 0	$0 \rightarrow 1$	1	AAAA-5608 = 54A2	0	
Zvětšení intervalu			A944	0	16
Kódování druhé číslice – 1	$1 \rightarrow 0$	1	5408	553C	
Zvětšení intervalu			A810	AA78	17
Kódování třetí číslice – 1	$0 \rightarrow 1$	$1 \rightarrow 0$	5608	FC80	
Zvětšení intervalu			AC10	1F900	18
Kódování čtvrté číslice – 0	$1 \rightarrow 2$	0	AC10-5408 = 5808	1F900	
Zvětšení intervalu			B010	3F200	19

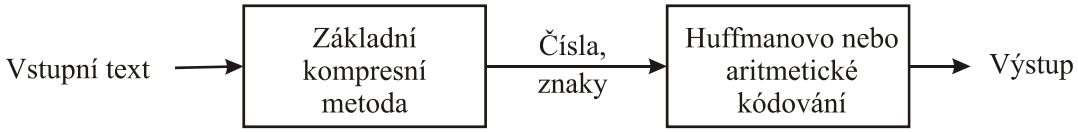
Další část příkladu ukazuje zpětné dekódování:

Činnost	Index řádku *r*	MPS	Horní mez intervalu *h*	Hodnota *v*	h-p	Dekód. číslice	Zbývající bity
Inicializace hodnot	0	1	AAAA	3F20			0000
Dekódování	$0 \rightarrow 1$	1	54A2	3F20	54A2	0	
Zvětšení intervalu			A944	7E40			000
Dekódování	$1 \rightarrow 0$	1	5408	2904	553C	1	
Zvětšení intervalu			A810	5208			00
Dekódování	$0 \rightarrow 1$	$1 \rightarrow 0$	5608	0	5208	1	
Zvětšení intervalu			AC10	0			0
Dekódování	$1 \rightarrow 2$	0	5808	0		1	
Zvětšení intervalu			B010	0			

Použití Huffmanova a aritmetického kódování

Kompresní poměr dosahovaný při použití Huffmanova i aritmetického kódování je nízký. V současné době máme k dispozici mnohem účinnější kompresní metody (jak statistické, tak slovníkové). Jejich popis bude uveden v dalších oddílech.

Huffmanovo ani aritmetické kódování se proto jako hlavní kompresní metoda nepoužívá. Nicméně tyto jednoduché metody statického kódování jsou velmi často využívány jako efektivní prostředek k uložení číselných nebo znakových hodnot, které jsou výstupem jiných, dokonalejších kompresních metod. Většina kompresních programů má základní uspořádání dle následujícího obrázku:



Obrázek 5 Typické uspořádání kompresních metod

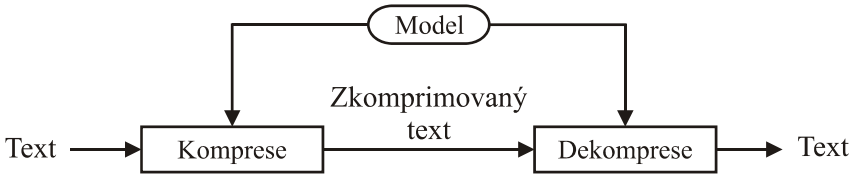
2.3 Model dat

Údaje o pravidelnostech a redundancích v komprimovaném textu nazýváme jeho modelem. Model je východiskem pro příslušnou kompresní metodu. Například u doposud popsaných metod statistického kódování jsou modelem pravděpodobnosti (nebo frekvence) výskytu jednotlivých znaků v textu. Huffmanův kód podle modelu stanoví délku kódování každého znaku, obdobně aritmetické kódování

pro jednotlivé znaky dle modelu stanoví velikost zmenšení intervalu hodnot. Modely dělíme na tři druhy:

- Statický
- Adaptivní
- Semiadaptivní

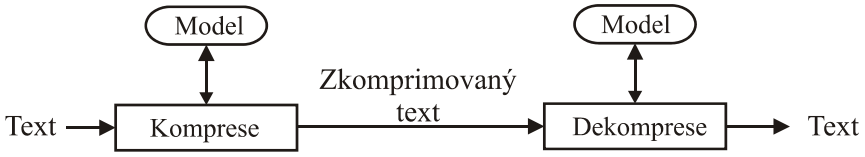
Statický model je pevně stanoven při implementaci kompresní metody. Je to jednotný model používaný pro kompresi všech vstupních textů. Účinnost komprese je ale závislá na tom, do jaké míry komprimovaný text odpovídá modelu. Čím více se text od modelu liší, tím více se snižuje účinnost komprese. Jestliže například model Huffmanova kódování předpokládá, že určitý znak se v textu bude vyskytovat s nízkou frekvencí, Huffmanův kód sestavený dle tohoto modelu zvolí pro uvedený znak kódování s větším počtem bitů. Pokud ale vstupní text tento předpoklad modelu nesplňuje a uvedený znak se v textu ve skutečnosti vyskytuje ve větší míře, než model předpokládá, projeví se to nižší účinností kódování, neboť se ve větší míře kóduje znak s delším kódem.



Obrázek 6 Statický model

Adaptivní model je způsob, kdy kompresní program si model vytváří až v průběhu komprese podle vstupního textu. Model ale je zapotřebí nejen pro kompresi, stejný model je rovněž potřebný při dekompresi. Což znamená, že kompresní program musí model vytvářet takovým způsobem, aby si stejný model mohl vytvořit i dekompresní program. Prakticky z toho vyplývá, že model lze vytvářet jen z té části textu, která je dostupná i pro dekompresní program, což je část textu, který dekompresní program již dekomprimoval.

Příkladem komprese s adaptivním modelem je aritmetické kódování popsané v minulé části. Model zde reprezentují kumulativní frekvence, které jsou průběžně načítány. Zde ale vzniká problém, že na počátku komprese žádný model zatím není. Ten je u aritmetického kódování řešen jednoduchým způsobem. Kompresní i dekompresní program na počátku volí frekvence všech znaků rovny 1. Po zakódování každého znaku kompresní program aktualizuje model, tj. zvýší hodnoty příslušných kumulativních frekvencí. Úplně stejným způsobem aktualizuje model i dekompresní program po dekódování každého znaku. Model je tedy v průběhu komprese (a dekomprese) neustále zpřesňován. Pokud by to dekompresní program dělal obráceně a nejprve dle znaku na vstupu aktualizoval model a pak teprve tento znak kódoval, nebyl by dekompresní program schopen znak dekódovat, protože znak ještě nezná a nedokázal by pro jeho dekódování příslušně aktualizovat model.



Obrázek 7 Adaptivní model

Semiadaptivního model je způsob, kdy kompresní program si nejprve před kompresí ze vstupního textu vytvoří model a pak s tímto modelem text zkomprimuje. Model musí přidat ke zkomprimovanému textu, neboť dekompresní program v tomto případě není schopen model sám vytvořit. Nutnost uložení modelu ve zkomprimovaném textu je základní nevýhoda semiadaptivního modelu, protože se tím zvětšuje celkový rozsah zkomprimovaných dat.

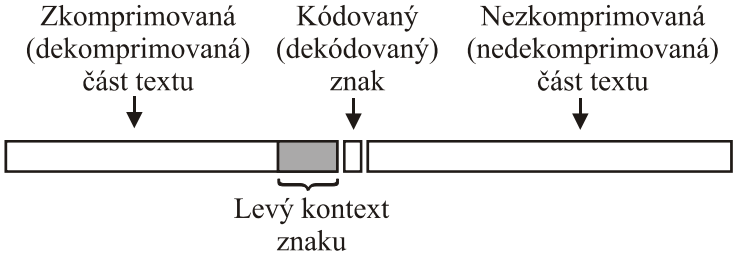
Semiadaptivní model se používá typicky u Huffmanova kódování. Kompresní program nejprve zjistí frekvence výskytu znaků v textu a z nich vytvoří strom s Huffmanovým kódem. Informace o tomto kódu v kompaktní formě nejprve uloží do výstupního souboru a pak provede kompresi textu. Dekompresní program načte informace o struktuře Huffmanova kódu, z kterých si kód rekonstruuje, a pak provede dekompresi textu. Bližší popis, jak úsporně uložit strukturu Huffmanova kódu, je v části *Metoda LZSS a její implementace* na straně 37.

Je zřejmé, že i Huffmanovo kódování by mohlo být implementováno jako adaptivní. Prakticky by ale kompresní program po zakódování každého znaku (a dekompresní program po dekódování každého znaku) musel znovu vytvářet strom s kódem, čímž by se komprese i dekomprese značně zpomalila.

Adaptivní model poskytuje největší účinnost komprese. Proto všechny dominantní kompresní metody jsou založeny na adaptivním modelu. U doplňujících způsobů kódování, které se používají pro uložení výstupů základních kompresních metod a u kterých volba modelu již nevlivní tak výrazně celkovou účinnost komprese, se poměrně často používá statický nebo semiadaptivní model, protože kódování s těmito modely je rychlejší. Příkladem metody, ve které se typicky používá statický nebo semiadaptivní model, je právě Huffmanovo kódování.

2.4 Metoda konečného kontextu (PPM)

V předchozí části jsme uvedli, že účinnost statistické komprese závisí na tom, jak přesně stanovíme frekvence výskytů znaků v textu (tj. model). Doposud popsané metody při kódování znaky uvažovaly samostatně bez ohledu na to, jaké znaky jsou v textu vedle nich. Cestou, jak při kódování přesněji určit frekvenci výskytu znaku, je zjištění kontextu, ve kterém se znak vyskytuje. Například při kompresi běžného textu v českém jazyce se písmeno *e* vyskytuje v mnohem větší míře po písmenu *s* než třeba po písmenu *w*. Kontextová metoda pro kódování výskytu *e* po *s* použije menší počet bitů než pro kódování výskytu *e* po *w*. Vzhledem k tomu, že tyto metody používají adaptivní model, kde kontext musí mít k dispozici i dekompresní program, připadá v úvahu jen levý kontext kódovaného znaku.



Obrázek 8 Levý kontext znaku

Problémem kontextového kódování je celkový počet frekvencí, který velmi narůstá s délkou kontextu. U bezkontextového kódování máme 256 různých znaků a tím i 256 frekvencí. Pokud bychom vzali jako kontext jeden znak, máme již 256^2 možných frekvencí, neboť každý z 256 znaků může mít v textu před sebou kterýkoliv z 256 znaků. U kontextu se 2 znaky už je to 256^3 možných frekvencí atd. Proto kontextové metody používají jen omezenou délku kontextu, přičemž typicky nebývá délka kontextu větší než 4. Odtud plyne i název *metoda konečného kontextu*.

Dalším problémem kódování v kontextu je situace, kdy znak se v daném kontextu vyskytuje poprvé. Frekvence jeho dosavadního výskytu a tedy i pravděpodobnost je v tomto případě nula. Nulovou pravděpodobnost ale statisticky kódovat nelze. Proto je nutné pro tyto případy vyčlenit nějakou nenulovou hodnotu pravděpodobnosti. Jak velkou tuto pravděpodobnost zvolit je určité dilema, neboť tato má nezanedbatelný vliv na účinnost komprese. Protože není k dispozici způsob, jak výpočet této pravděpodobnosti matematicky odvodit, bylo intuitivně navrženo několik možných přístupů řešení uvedeného problému. Jednotlivé navržené metody jsou pro odlišení označovány velkými písmeny. Experimentálně bylo ověřeno, že nejlepší výsledky dává metoda označená jako *C*. Než přistoupíme k jejímu popisu, zavedeme některé pojmy a označení:

Termín délka kontextu označuje počet znaků, který bereme v úvahu při kódovaní daného znaku.

z označuje kódovaný znak.

α označuje kontext.

$p_k(\alpha)$ je pravděpodobnost výskytu kontextu α s délkou *k*.

$f_k(\alpha)$ je počet výskytů (frekvence) kontextu α s délkou *k*.

$p_k(\alpha,z)$ je pravděpodobnost výskytu znaku *z* v kontextu α o délce *k*.

$f_k(\alpha,z)$ je počet výskytů (frekvence) znaku *z* v kontextu α o délce *k*.

$e_k(\alpha)$ je pravděpodobnost přidělená znaku, který se v kontextu α doposud nevyskytl.

Popis metody C

Pravděpodobnost nového znaku (znaku, který se v daném kontextu α doposud v textu nevyskytl) tato metoda stanoví dle vztahu

$$e_k(\alpha) = \frac{c_k(\alpha)}{f_k(\alpha) + c_k(\alpha)}$$

kde $c_k(\alpha)$ je počet všech různých znaků, které se doposud vyskytly v kontextu α. Frekvence $f_k(\alpha)$ je součet frekvencí $f_k(\alpha,z)$:

$$f_k(\alpha) = \sum_z f_k(\alpha,z)$$

přičemž *z* v sumě označuje všechny znaky, které se v kontextu α již vyskytly, tj. u kterých je $f_k(\alpha,z)>0$.

Při stanovení nenulové pravděpodobnost $e_k(\alpha)$ se sníží prostor pro pravděpodobnosti znaků, které se již v kontextu α vyskytly. Musíme výpočet pravděpodobností těchto znaků upravit tak, aby součet všech pravděpodobností v kontextu α zůstal roven 1. Pro znaky, které se již v kontextu vyskytly, metoda *C* stanoví výpočet pravděpodobností dle vztahu

$$p_k(\alpha,z) = \frac{f_k(\alpha,z)}{f_k(\alpha) + c_k(\alpha)}$$

Snadno se ověří, že je splněno $e_k(\alpha) + \sum_z p_k(\alpha,z) = 1$

Z uvedených vztahů plyne, že čím více se nějakých znaků před novým znakem v daném kontextu již vyskytlo (hodnota $f_k(\alpha)$), tím menší bude pravděpodobnost $e_k(\alpha)$. Intuitivně to vyjadřuje skutečnost, že pokud se znak doposud v nějaké delší části textu v daném kontextu zatím nevyskytl, bude jeho pravděpodobnost výskytu v tomto kontextu celkově zřejmě malá. Naopak ale čím více těchto znaků v tomto kontextu bylo různých (hodnota $c_k(\alpha)$), tím zřejmě hodnota pravděpodobnosti výskytu ještě dalšího nového znaku bude větší.

Příklad. Předpokládejme, že v textu mohou být jen znaky *a*,*b*,*c*,*d*. Dále předpokládejme, že doposud zkomprimovaná část textu je

bbabbcbbabb

Budeme uvažovat kontext délky 2, což v tomto případě je *bb*. V tomto kontextu se již vyskytly dva znaky *a* a *c*. Tedy platí

c_2("bb") = 2

Jejich frekvence výskytu jsou

f_2("bb",a) = 2

f_2("bb",c) = 1

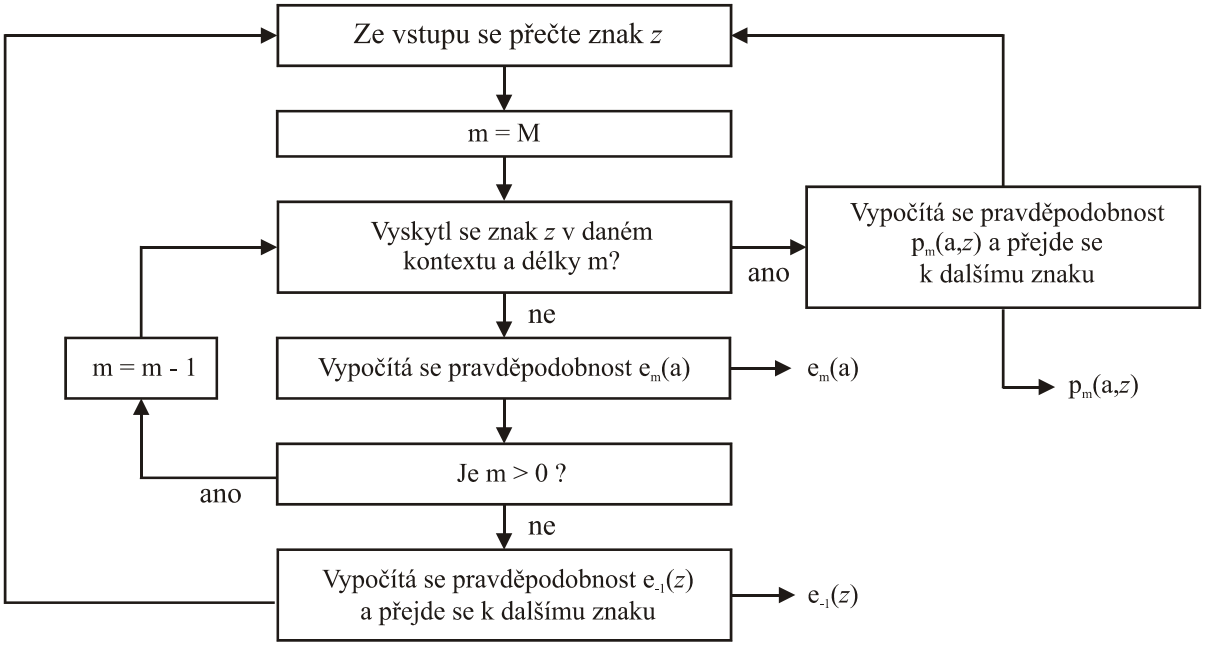
Odtud dostáváme celkový výskyt kontextu *bb*

$f_2("bb") = 3$

Pravděpodobnost, že se nyní v kontextu *bb* vyskytne některý další znak, který se v něm doposud nevyskytl (tedy znak *b* nebo *d*) je dle metody C:

$$e_2("bb") = \frac{2}{3+2} = \frac{2}{5}$$

Už jsme uvedli, že metoda konečného kontextu stanoví maximální délku sledovaného kontextu. Označme ji *M*. Kódování probíhá podle následujícího schématu. Písmeno *m* v něm označuje právě uvažovanou délku kontextu.



Obrázek 9 Postup kódování metodou pevného kontextu

Kódování každého znaku začíná vždy od nejdelšího kontextu m=M. Jestliže se znak v něm již vyskytl, vypočítá se pravděpodobnost $p_m(\alpha,z)$, zašle se na výstup a přechází se na další znak. Pokud ne, vypočítá se pravděpodobnost $e_m(\alpha)$, zašle se na výstup a přejde se na kontext o 1 nižší. Tento proces se cyklicky opakuje do té doby, než se najde taková délka daného kontextu, ve které se znak *z* již vyskytl, nebo se zjistí, že znak *z* se v textu ještě vůbec nevyskytl a pak se na závěr zakóduje hodnota $e_{-1}(\alpha)$.

Příklad. Uvažujme například text

aadbca d

kde prvních 6 znaků *aadbca* je již zakódováno a nyní kódujeme sedmý znak v pořadí *d*. Předpokládáme maximální délku kontextu M=3, což v tomto případě znamená kontext *bca*.

Na výstup se při kódování sedmého znaku zašlou tři pravděpodobnosti

$e_3("bca"), e_2("ca"), p_1("a",d)$

neboť znak *d* se v kontextu *bca* délky 3 předtím nevyskytl a ani se nevyskytl v kontextu *ca* délky 2. Vyskytl se ale v kontextu *a* délky 1.

Při dekompresi se podle hodnot $e_3("bca")$ a $e_2("ca")$ pozná, že právě dekódovaný znak se v uvažovaných kontextech délky 3 a 2 předtím nevyskytl a že se má přejít na kontext délky 1.

Příklad. Vezměme další příklad textu

abdbca d

Zde kódovaný znak *d* se v kontextu *bca* v žádné délce nevyskytl. Jeho kódování proto bude

e_3("bca"), e_2("ca"), e_1("a"), p_0(d)

Pravděpodobnost p_0(d) je bezkontextová pravděpodobnost, tj. je dána jen frekvencí předchozího výskytu znaku *d* bez vztahu ke kontextu.

Příklad. Poslední příklad textu

ababca d

Zde kódovaný znak *d* se předtím ještě vůbec nevyskytl. Jeho kódování bude

e_3("bca"), e_2("ca"), e_1("a"), e_0 ,e_{-1}(d)

kde e_{-1}(d) je kódování pravděpodobnosti znaku, který se doposud v textu vůbec nevyskytl.

Pravděpodobnost všech znaků, které se v textu doposud nevyskytly, se volí stejná a je tímto určena vztahem

$$e_{-1} = \frac{1}{n - c_0}$$

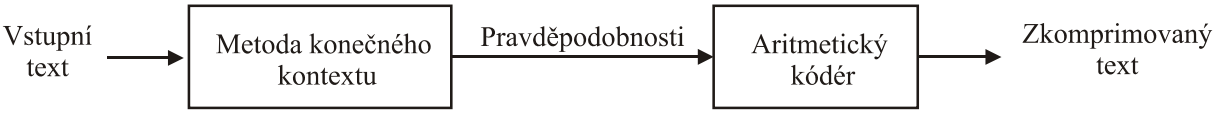
kde *n* je počet všech možných znaků v textu a c_0 je počet různých znaků, který se již v textu vyskytly (bez ohledu na kontext). Výraz $n-c_0$ ve jmenovateli zlomku tedy vyjadřuje počet všech znaků, které se v textu doposud nevyskytly.

Jestliže například se zatím v textu nevyskytlo 5 různých znaků z_1, z_2, z_3, z_4, z_5, bude každý z nich kódován se stejnou pravděpodobností

$$e_{-1}(z_1) = e_{-1}(z_2) = e_{-1}(z_3) = e_{-1}(z_4) = e_{-1}(z_5) = \frac{1}{5}$$

Poznámka: Pro pravděpodobnosti e_k se používá označení *escape* kódy.

Výstupem metody konečného kontextu jsou pravděpodobnosti výskytu jednotlivých znaků. Ty je zapotřebí vhodným způsobem zakódovat a uložit. K tomu se používá aritmetické kódování. Komprese metodou konečného kontextu má uspořádání:



Obrázek 10 Uspořádání komprese metodou konečného kontextu

Příklad. Pro jednoduchost budeme předpokládat, že v textu se může vyskytnout jen 6 znaků *a*,*b*,*c*,*d*,*e*,*f* a maximální délku kontextu zvolíme M=3.

Pro ukázku komprese zvolíme text

dbaccacbacacbdbacaacbac z

kde prvních 23 znaků je již zakódováno a *z* reprezentuje právě kódovaný znak. Ukážeme, jak by vypadalo kódování pro všech 6 možných znaků, které se na vstupu mohou vyskytnout, tj. postupně budeme uvažovat *z* = *a*,*b*,*c*,*d*,*e*,*f*.

Kontext kódovaného znaku je *bac*. Sestavíme pro tento kontext tabulku frekvencí a pravděpodobností:

Délka kontextu	Frekvence f_m a pravděpodobnosti p_m pro jednotlivé znaky. (Celá čísla označují frekvence, zlomky pravděpodobnosti.)						Escape kódy
m	a	b	c	d	e	f	e_m
3	$2\frac{2}{5}$	0	$1\frac{1}{5}$	0			$\frac{2}{5}$

2	–	3 $\frac{3}{4}$	–	0			$\frac{1}{4}$
1	–	–	–	0			1
0	–	–	–	2 $\frac{2}{3}$			$\frac{1}{3}$
-1					$\frac{1}{2}$	$\frac{1}{2}$	

V kontextu maximální délky *bac* se vyskytl znak *a* s frekvencí 2 a znak *c* s frekvencí 1. Odtud:

$$c_3(\text{"bac"}) = 2$$

$$p_3(\text{"bac"}, a) = \frac{2}{3+2} = \frac{2}{5}$$

$$p_3(\text{"bac"}, c) = \frac{1}{3+2} = \frac{1}{5}$$

$$e_3(\text{"bac"}) = \frac{2}{3+2} = \frac{2}{5}$$

V kontextu *ac* délky 2 se vyskytl znak *b* s frekvencí 3. Odtud:

$$c_2(\text{"ac"}) = 1$$

$$p_2(\text{"ac"}, b) = \frac{3}{3+1} = \frac{3}{4}$$

$$e_2(\text{"ac"}) = \frac{1}{3+1} = \frac{1}{4}$$

V kontextu *c* délky 1 se nevyskytl zatím žádný znak, který se zároveň nevyskytl již v kontextu delším. Což jinými slovy znamená, že se v této situaci je pro libovolný znak přechod ke kontextu nižšímu, tedy *escape* pravděpodobnost je rovna 1:

$$e_1(\text{"c"}) = 1$$

Bezkontextově (tj. v kontextu délky 0) se vyskytl znak *d* s frekvencí 2. Odtud:

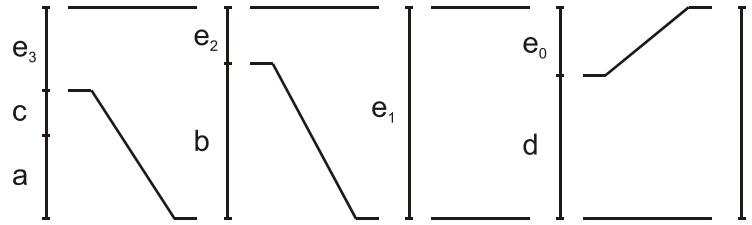
$$p_0(d) = \frac{2}{2+1} = \frac{2}{3}$$

$$e_0 = \frac{1}{2+1} = \frac{1}{3}$$

Zde se zastavme a ukažme kódování znaku *d*. Do aritmetického kodéru budou zaslány pravděpodobnosti:

$$e_3(\text{"bac"}),\ e_2(\text{"ac"}),\ e_1(\text{"c"}),\ p_0(d)$$

Následující obrázek znázorňuje průběh aritmetického kódování znaku *d*.



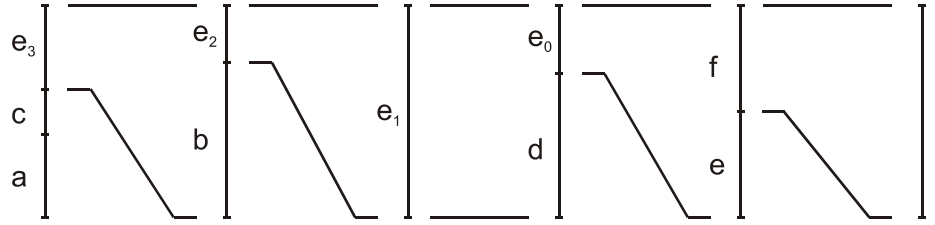
Spočítejme, kolik zabere kódování znaku *d* bitů:

$$-\log_2\left(\frac{2}{5}\right)-\log_2\left(\frac{1}{4}\right)-\log_2(1)-\log_2\left(\frac{2}{3}\right)=3.91 \text{ bitů}$$

Nakonec zbývají dva znaky *e* a *f*, které se doposud v textu nevyskytly. Oba budou kódovány se stejnou pravděpodobností

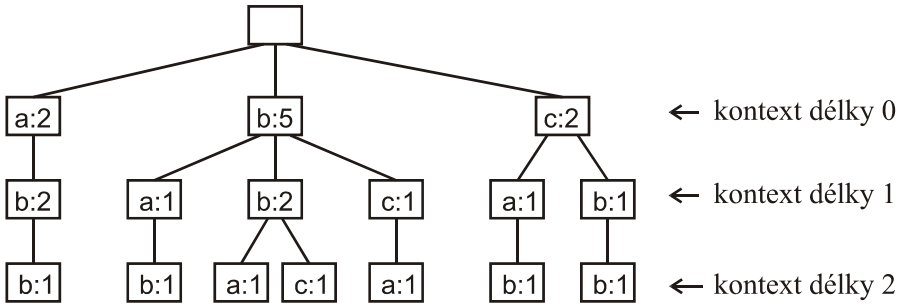
$$e_{-1}=\frac{1}{6-4}=\frac{1}{2}$$

Zobrazme průběh aritmetického kódování znaku *f*:



Z předchozího příkladu je zřejmé, že po každém přechodu k nižší délce kontextu se při výpočtu pravděpodobností uvažují jen ty znaky, které se předtím nevyskytly v kontextu větší délky. Například u kontextu délky 1 v předchozím příkladu už neuvažujeme znaky *a*, *b*, *c*, neboť ty se vyskytly v kontextu délky 3 a 2.

Na závěr několik poznámek k praktické implementaci metody konečného kontextu. Metoda pro svoji činnosti vyžaduje uložení kontextů, které se vyskytly v průběhu komprese. Jeden z možných způsobů, jak kontexty uložit, je stromové uspořádání. Na následujícím obrázku je příklad stromu pro text *cbbcabbab* a maximální délku kontextu M=2.



Obrázek 11 Reprezentace kontextu pomocí stromu

Výška stromu je M+1, tedy o 1 větší než je maximální kontext. Každý uzel reprezentuje určitý znak. Vedle znaku je frekvence výskytů v kontextu, jež je určen znaky obsaženými v uzlech na cestě od kořene k uzlu s daným znakem.

Uzly bezprostředně pod kořenem reprezentují bezkontextový výskyt. Například znak *b* se v textu vyskytl pětkrát. O jednu úroveň níž je kontext délky 1. Například v levé větvi je vidět, že znak *b* se v kontextu znaku *a* vyskytl dvakrát. Listy stromu reprezentují kontext maximální délky, tj. délky 2. Například v pravé větvi stromu pro znak *b* je vidět, že znak *b* se v kontextu *ca* vyskytl jedenkrát.

Strom reprezentuje model komprese a po zakódování každého znaku je aktualizován. Maximální hodnota frekvencí je omezena obdobně jako u aritmetického kódování. Jestliže dojde k jejímu překročení, jsou všechny frekvence větší než 1 děleny hodnotou 2.

Strom obsahuje všechny potřebné informace pro kódování. Nicméně vyhledání kontextu a výpočet příslušných pravděpodobností jsou operace značně časově náročné. Proto se do stromové struktury běžně přidávají další odkazy, které vedou ke zrychlení pohybu po stromu.

Dekompresní program si vytváří stejný model jako kompresní program. Po dekódování každého znaku dekompresní program model rovněž stejným způsobem aktualizuje. Proto čas dekomprese se na rozdíl od jiných metod příliš neliší od času potřebného pro kompresi.

Popsaná metoda končeného kontextu je známa pod označením PPMC. První tři písmena pochází z názvu metody *prediction by partial match*. Poslední písmeno označuje, že pro výpočet *escape* kódu se používá metoda *C*. Princip použití *escape* kódu k přechodu na nižší kontext je označován názvem *exkluze*.

Metoda PPMC patří mezi nejúčinnější kompresní techniky. Přesto se v praxi nepoužívá. Pro kompresi i dekompresi potřebuje značné množství paměti a je časově velmi náročná. Slovníkové techniky, které budou popsány v následující kapitole, poskytují o něco horší účinnost komprese, vyžadují ale podstatně méně paměti a zejména jsou výrazně rychlejší.

Vedle PPMC existuje ještě několik dalších metod statistické komprese. Žádná z nich ale není tak účinná jako PPMC a ani nemá praktický význam.

3. Slovníkové metody

Slovníkové metody jsou založeny na vyhledávání opakujících se částí textu. Při kompresi se do výstupního souboru uloží jen první výskyt části textu a další výskyty se nahradí odkazem. Podle toho, jako tento odkaz řeší, dělí se slovníkové metody na dvě základní třídy:

1) První třída metod nahrazuje výskyt opakující se části textu odkazem na místo, kde se v textu tato část předtím vyskytla.
2) Druhá třída si z částí textu, které se předtím již vyskytly, vytváří v paměti slovník a další výskyty nahrazuje odkazem na příslušné místo ve slovníku.

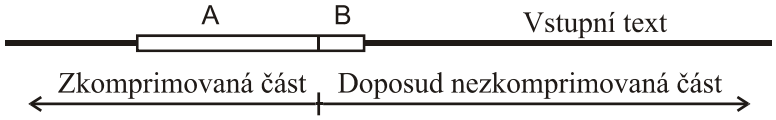
Slovníkovou kompresi založenou na prvním uvedeném principu publikovali pánové Ziv a Lempel v roce 1977. Jejich původní návrh byl postupem času různými způsoby modifikován a vznikla tak celá třída kompresních metod, pro které se používá označení LZ77. Stejní autoři o rok později publikovali další kompresní metodu, tentokrát založenou na druhém uvedeném principu. Opět z jejich původního návrhu postupem času vznikla celá třída kompresní metod označovaná LZ78.

Metody obou dvou tříd LZ77 a LZ78 se řadí do kategorie metod s adaptivním modelem.

Než přistoupíme k popisu jednotlivých metod, uvedeme ještě pojem fráze. Je to označení používané v teorii slovníkových metod pro opakující se části textu.

3.1 Metody LZ77

Většina metod třídy LZ77 vyhledává opakující se části textu jen v pevně stanové, omezeném délce. Základní postup ukazuje následující obrázek:



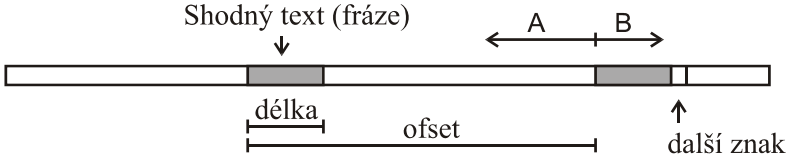
Obrázek 12 Posuvné okno v kompresi metodou LZ77

Po textu se pohybuje posuvné okno rozdělené na dvě části, které jsou na obrázku označené jako *A* a *B*. Část *A* okna obsahuje úsek na konci již zkomprimované části textu. Velikost této části okna bývá v rozmezí 8 - 64 KB. Část *B* okna obsahuje začátek doposud nezkomprimované části textu a její velikost bývá do 256 bytů. Při kompresi se v každém kroku v části *A* hledá text, který se v nějaké délce shoduje s textem v části *B*, tj. hledá se fráze. Pokud je v *A* nalezeno více frází, bere se nejdelší z nich. Pokud je v *A* více výskytů nejdelší fráze, bere se vždy ta, která je nejblíže k *B*.

Další pokračování kompresního kroku je kódování nalezené shody. Původní návrh metody používal ke kódování trojici

(ofset, délka, další_znak),

kde *ofset* je vzdálenost fráze od části *B*, za ní následuje délka fráze a poslední v trojici je znak následující v okně *B* za frází.



Obrázek 13 Fráze a další znak při kompresi LZ77

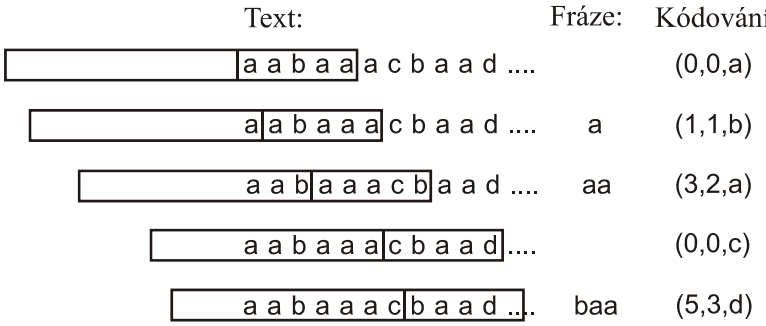
Kompresní krok se dokončí posunutím okna doprava o *délka_fráze+1* znaků.

V případě, že v *A* není nalezena žádná fráze shodná s textem v *B*, kóduje se trojice

(0,0,znak_na_ začátku_B)

a okno se posune o 1 znak doprava. Tím je zaručeno, že v každém kompresní kroku se zakóduje nejméně jeden znak.

Příklad. Na následujícím obrázku je komprese textu začínajícího znaky *aabaaacbaad*.



Na počátku komprese okno *A* je prázdné, proto v prvním kroku se vždy kóduje trojice (0,0,znak). Okno *A* se postupně zaplňuje a pravděpodobnost nalezení fráze se zvyšuje.

Dekomprese je velmi jednoduchá. Dekompresní program si v paměti udržuje dekomprimovaný text v délce okna *A*. Čte trojice a z nich dekóduje opakující se části textu a znaky za nimi. Na dekompresi je podstatné, že je velmi rychlá. Zatímco komprese pro nalezení fráze vyžaduje prohledávání okna *A*, dekomprese frázi pomocí ofsetu zjistí okamžitě.

Volba délky okna *A* je výsledkem kompromisu protichůdných aspektů. Je zřejmé, že čím *A* je větší, tím je větší pravděpodobnost, že se nalezne fráze. Naproti tomu prohledávání delšího okna vyžaduje delší čas a navíc velké hodnoty ofsetu jsou pro kompresi nepříznivé, protože ukládání velkých čísel na výstup vyžaduje více místa. Dnes se pro okno *A* typicky používá velikost 32 KB.

3.1.1 Metoda LZSS a její implementace

Ze všech metod třídy LZ77, které vznikly modifikováním původního přístupu popsaného v předchozí části, má největší význam metoda LZSS. Vychází z myšlenky, že není efektivní vždy kódovat znak za frází, neboť za touto frází může být další frází, která začíná tímto znakem. Znaky by se měly kódovat jen v případech, kdy fráze neexistuje. Proto tato metoda kóduje samostatně fráze ve tvaru

(příznak, ofset, délka)

a kóduje samostatně znaky ve tvaru

(příznak, znak)

kde *příznak* zabírá jeden bit a je v něm uložena hodnota 0 nebo 1 udávající, zda se kóduje fráze nebo znak.

Pokud se v kompresním kroku najde fráze, tato se zakóduje a přejde se k dalšímu kroku. Znak se v kompresním kroku kóduje jen v případě, kdy žádná fráze nebyla nalezena.

Rychlost komprese je závislá na rychlosti vyhledávání frází. Proto kompresní programy mají tuto část často psanou v asembleru, který z hlediska rychlosti umožňuje napsat efektivnější vyhledávání než programovací jazyky vyšší úrovně. Jiný způsob, jak zrychlit vyhledávání, je použití tabulky existujících frází, jejíž konstrukce je popsána v následujících odstavcích.

Pro vytvoření tabulky frází a rychlé vyhledávání v ní se používá hashovací funkce. Hashovací funkce na základě textu fráze určí místo v tabulce, na kterém bude uložena informace udávající pozici této fráze v okně *A*. Protože fráze jsou obecně různě dlouhé, hodnota hashovací funkce se počítá jen z počáteční části fráze, jejíž délka je rovna nejmenší používané délce fráze. Nejmenší délku fráze je účelné zvolit 3 znaky, neboť kódovat fráze délky 2 znaky není efektivní. Kódování ofsetu a délky fráze by na výstupu zabralo místo srovnatelné s kódováním dvou samostatných znaků. Další

záležitostí je volba velikosti hashovací tabulky. Obecně počet prvků tabulky volíme buďto prvočíslo anebo mocninu čísla 2. Prvočíslo přispívá k rovnoměrnému rozložení hodnot v hashovací tabulce, naproti tomu mocnina 2 je výhodná pro výpočet operace modulo, která se používá v hashovací funkci, neboť tu lze v tomto případě realizovat bitovou operací logického součinu (and). Použití této druhé alternativy bývá velmi časté.

Jestliže zvolíme počet prvků tabulky 2^{15} (32768) a minimální délku fráze 3, bude hashovací funkce zobrazením prvních tří znaků fráze $a_1a_2a_3$ do intervalu $<0,2^{15}-1>$:

$$hash(a_1a_2a_3) \rightarrow <0,2^{15}-1>$$

Je řada možností, jak takovou funkci zvolit. Vhodná je například funkce

$$hash(a_1a_2a_3) = ((((a_1 << 5) \wedge a_2) << 5) \wedge a_3) \,\&\, (2^{15}-1)$$

kde ^ označuje bitovou operaci nonekvivalence (xor) znaky << označují bitový posuv vlevo (počet bitů posuvu udává pravý operand) a znak & označuje bitovou operaci logického součinu (and), která se zde používá místo operace výpočtu zbytku dělení (operace modulo), neboť platí, že operace

$$výraz \,\&\, (2^{15}-1)$$

je ekvivalentní s výpočtem zbytku při dělení výrazu hodnotou 2^{15}, tj.

$$výraz \bmod 2^{15}$$

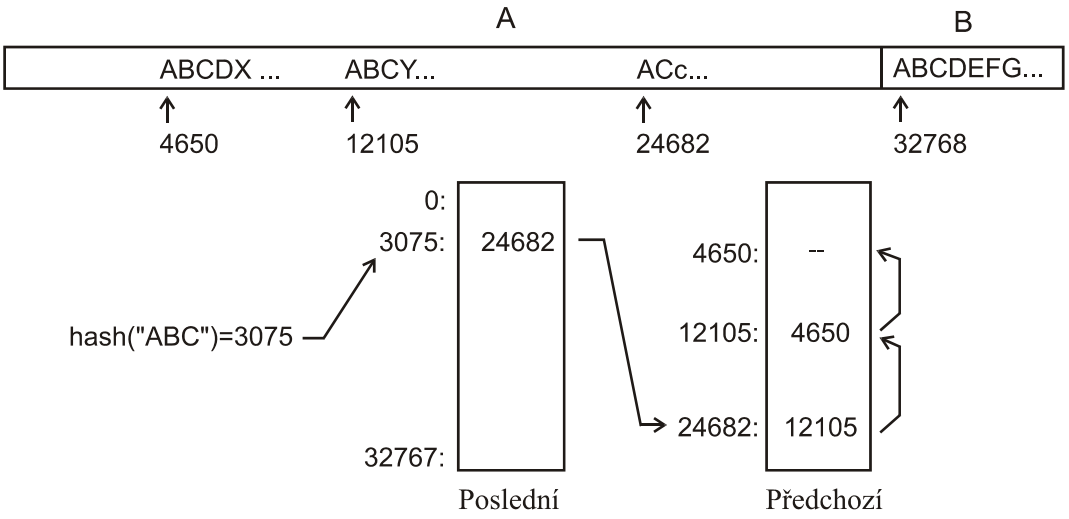
Výhoda uvedené volby hashovací funkce *hash* spočívá v možnosti rekurzivním výpočtu hodnoty funkce při přechodu o 1 znak doprava k další frázi. Uvažujme 4 znaky za sebou $a_1a_2a_3a_4$. Hodnotu funkce pro další frázi začínající znaky $a_2a_3a_4$ můžeme snadněji spočítat z hodnoty funkce pro předchozí frázi začínající znaky $a_1a_2a_3$, neboť z definice funkce plyne platnost vztahu:

$$hash(a_2a_3a_4) = (hash(a_1a_2a_3) << 5) \wedge a_4) \,\&\, (2^{15}-1)$$

Vlastní tabulka je tvořena dvěma poli. První pole, na obrázku označené jako *Poslední*, obsahuje v místě určeném hodnotou hashovací funkce pozici příslušné fráze v okně *A*. Je ovšem více frází, které mohou mít stejnou hodnotu hashovací funkce. V poli *Poslední* je ta fráze z nich, která byla nalezena jako poslední. Předchozí jsou v poli *Předchozí*. Délka pole *Předchozí* se standardně volí tak, aby měla dvojnásobný počet prvků jako je délka okna *A*. A jestliže okno *A* má délku 32 KB, bude pole *Předchozí* mít 65536 prvků.

Pozici poslední fráze s danou hodnotou hashovací funkce nalezneme v poli *Poslední*. Zároveň na této pozici najdeme v poli *Předchozí* pozici další fráze (tj. předposlední) se stejnou hodnotu hashovací funkce. Každá pozice v poli *Předchozí* ukazuje nejen na místo, kde se v okně *A* nachází příslušná fráze, ale i na místo v poli *Předchozí*, kde se nachází pozice další fráze se stejnou hodnotou hashovací funkce.

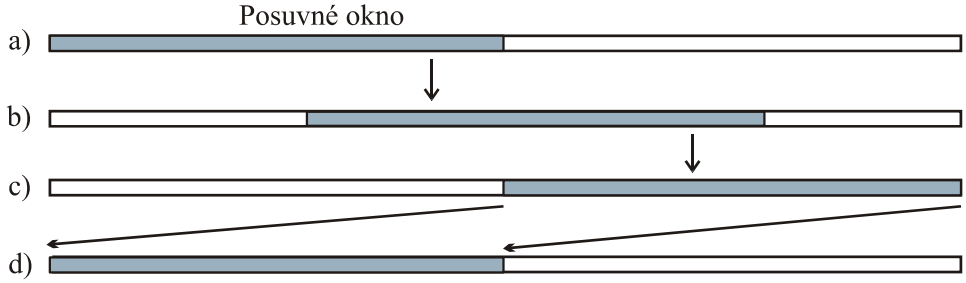
Na příkladu uvedeném v následujícím obrázku se hledá fráze textu *ABCDE...*, který je obsažen v okně *B*. Hodnota hashovací funkce pro první tři znaky *ABC* je 3075. Na tomto místě je v poli *Poslední* pozice 24682 fráze s touto hodnotou hashovací funkce. Nyní je nutné srovnat text v okně *B* s textem na pozici 24682 v okně *A*. V tomto případě zjistíme, že jde o jinou frázi, která má stejnou hodnotu hashovací funkce. V poli *Předchozí* na pozici 24682 najdeme pozici 12105 další fráze s hodnotou hashovací funkce 3075. Srovnáním zjistíme, že na této pozici je v okně *A* fráze délky 3 vzhledem k textu v okně *B*. Fráze je sice nalezena, ale prohledávání řetězce frází pokračuje, neboť může existovat ještě delší fráze. Na pozici 12105 najdeme v poli *Předchozí* pozici 4605 dalšího kandidáta na frázi. Srovnání textu na pozici 4605 v okně *A* s textem v okně *B* ukazuje, že jsme nalezli frázi délky 4. Tato fráze je v poli *Předchozí* poslední frází s hodnotou 3075 hashovací funkce, čímž hledání fráze v tomto kompresním kroku končí.



Obrázek 14 Způsob uložení frází v hashovací tabulce

Nyní proběhne zakódování nalezené fráze. Ofset se vypočítá jako rozdíl pozice okna *B* a pozice nalezené fráze, tj. 32768-4650, délka fráze je 4. Okno *B* se posune o 4 znaky doprava za nalezenou frázi, tj. na text *EG...* Zároveň s tím do tabulek *Poslední* a *Předchozí* zařadíme všechny fráze délky 3, o které bylo posunuto okno *B*. V našem příkladu to budou 4 fráze *ABC*, *BCD*, *CDE* a *DEG*.

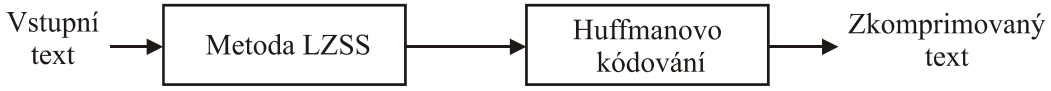
Jen krátce si povšimněme, jak je realizován princip posuvného okna. Program Zip to například řeší vyrovnávací pamětí o dvojnásobné délce velikosti okna. Okno se postupně posunuje od začátku vyrovnávací paměti (případ *a)* na následujícím obrázku) až postupně dosáhne její konec (případ *c)* na obrázku):



Obrázek 15 Posouvání okna ve vyrovnávací paměti během komprese

V tomto okamžiku je obsah okna překopírován na začátek vyrovnávací paměti (případ *d)* na obrázku) a proběhne aktualizace tabulky frází, při které jsou přepočítány pozice frází obsažených v přesunutém oknu a vyloučeny fráze, které jsou již mimo okno.

Výstupem metody LZSS jsou buďto hodnoty ofsetu a délky fráze nebo hodnota znaku. Pro efektivní uložení těchto hodnot se běžně používá Huffmanovo kódování. Typické uspořádání kompresního programu je:



Obrázek 16 Základní uspořádání kompresního programu

Podrobnější popis, jak se Huffmanovo kódování pro tento účel implementuje, je poněkud obšírnější. V zásadě kompresní program podle délky textu volí buďto semiadaptivní nebo statické kódování.

1) U semiadaptivního kódování program z frekvencí kódovaných hodnot sestaví příslušný Huffmanův kód. Tento v kompaktní formě uloží na výstup a s jeho použitím zakóduje výstupní hodnoty.

Uložení Huffmanova kódu do výstupního souboru při semiadaptivním kódování zabírá určité místo. Proto kompresní program volí tento způsob, jestliže je výhodnější než statické kódování, tj. kdy

délka uložení Huffmanova kódu + délka kódování s tímto kódem < délka statického kódování.

2) U statického kódování kompresní i dekompresní program používá pevně stanovený Huffmanův kód, tj. používá statický model, s kterým kóduje výstupní hodnoty.

Jako příklad statického modelu kódování uvedeme způsob, který používá program Zip. Ten kódování znaku a fráze nerozlišuje pomocí příznaku, ale obojí kóduje pomocí hodnoty z intervalu <0,285>.

Hodnoty 0 až 255 z tohoto intervalu se používají pro kódování znaků. Jestliže je na výstup zapsána hodnota 257 až 285 značí to kódování fráze. Zbývá ještě hodnota 256, která je vyhrazena pro označení konce bloku dat.

Hodnoty 257 až 285 nejen označují kódování fráze, ale zároveň určují délku fráze. Fráze může být dlouhá 3 až 258 znaků. U krátkých frází (3 až 10 znaků) je délka kódována přímo hodnotami 257 až 264, u delších frází (11 až 258 znaků) hodnoty 265 až 285 určují jen určitý interval délek a za nimi musí následovat další bity, kterými je vyjádřená příslušná délka z tohoto intervalu. Kódy, jim odpovídající délky fráze a počty dalších bitů potřebných pro jejich vyjádření jsou uvedeny v následující tabulce:

Kód	Délka	Kód+bitů	Délky	Kód+bitů	Délky	Kód+bitů	Délky	Kód+bitů	Délky
257	3	263	9	269 + 2bity	19-22	275 + 3bity	51-58	281 + 5bitů	131-162
258	4	264	10	270 + 2bity	23-26	276 + 3bity	59-66	282 + 5bitů	163-194
259	5	265 + 1bit	11-12	271 + 2bity	27-30	277 + 4bity	67-82	283 + 5bitů	195-226
260	6	266 + 1bit	13-14	272 + 2bity	31-34	278 + 4bity	83-98	284 + 5bitů	227-257
261	7	267 + 1bit	15-16	273 + 3bity	35-42	279 + 4bity	99-114	285	258
262	8	268 + 1bity	17-18	274 + 3bity	43-50	280 + 4bitů	115-130		

Samotné základní hodnoty 0 až 285 jsou kódovány prefixovým kódem v délce 7, 8 a 9 bitů

Hodnoty	Kódování	Hodnoty	Kódování
0 - 143	00110000 - 10111111	256 - 279	0000000 - 0010111
144 - 255	110010000 - 111111111	280 - 287	11000000 - 11000111

Ofset fráze se kóduje rovněž dvěma hodnotami. První je 5-bitová hodnota v intervalu <0,29>. První čtyři hodnoty intervalu 0 až 3 udávají přímo ofset 1 až 4. Další hodnoty 4 až 31 určují jen určitý interval hodnot ofsetů a za nimi musí následovat další bity určují příslušnou hodnotu ofsetu z tohoto intervalu.

Kód+bitů	Ofset	Kód+bitů	Ofset	Kód+bitů	Ofset	Kód+bitů	Ofset
0	1	8 + 3bity	17-24	16 + 7bitů	257-384	24 + 11bitů	4097-6144
1	2	9 + 3bity	25-32	17 + 7bitů	385-512	25 + 11bitů	6145-8192
2	3	10 + 4bity	33-48	18 + 8bitů	513-768	26 + 12bitů	8193-12288
3	4	11 + 4bity	49-64	19 + 8bitů	769-1024	27 + 12bitů	12289-16384
4 + 1bit	5-6	12 + 5bitů	65-96	20 + 9bitů	1025-1536	28 + 13bitů	16385-24576
5 + 1bit	7-8	13 + 5bitů	97-128	21 + 9bitů	1537-2048	29 + 13bitů	24577-32768
6 + 2bity	9-12	14 + 6bitů	129-192	22 + 10bitů	2049-3072		
7 + 2bity	13-16	15 + 6bitů	193-256	23 + 10bitů	3073-4096		

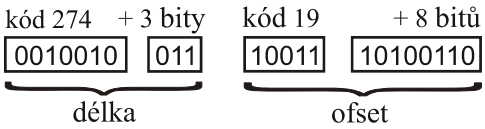
Příklad. Zakódujeme frázi s délkou 46 znaků a ofsetem 935.

V první tabulce zjistíme, že délce 46 odpovídá kód 274. Jeho binární hodnotu stanovíme pomocí druhé tabulky. Je to 0010010 (274-256 je 18, což binárně je 10010). Kód 274 má 3

další bity, jejichž hodnota vyjadřuje rozdíl mezi kódovanou délkou 46 a počátkem intervalu 43, který tento kód označuje. Tento rozdíl je 3, což binárně v délce 3 bitů je 011.

Jako další je kódování ofsetu. Hodnotě 935 odpovídá kód 19, což binárně je 10011. Tento kód má dalších 8 bitů, kterým je kódován rozdíl mezi ofsetem 935 a počátkem intervalu 769 kódu 19. Tento rozdíl je 166, což binárně v délce 8 bitů je 10100110.

Při kódování fráze kodér uloží na výstup 7+3+5+8, tedy celkem 23 bitů:



Příklad. Zakódujeme mezeru.

Hodnota mezery v ASCII tabulce je 32, což binárně je 00010000. V druhé tabulce zjistíme, že hodnotě 0 odpovídá binární kód 00110000. Kód mezery bude jejich součtem 00110000+00010000=01000000.

U semiadaptivního kódování je základní rozdělení kódu stejné jako u statického. Opět znaky a délky jsou kódovány hodnotami z intervalu 0-285 s použitím dalších bitů pro délky. Obdobně ofset je kódován hodnotami 0-29 s použitím dalších bitů. Rozdíl je v tom, že pro kódování jednotlivých hodnot z intervalu 0-285 udávající znaky a délky je nyní použit Huffmanův kód vypočtený z frekvencí jejich výskytů v kódovaném bloku textu. Obdobně je sestaven samostatný Huffmanův kód i pro kódy ofsetů 0-29.

Jak by se frekvence zjistily a jak se z nich sestaví příslušné Huffmanův kódy, si již umíme představit. Podstatnější je, jak tyto kódy úsporně uložit do výstupního souboru. Program Zip místo hodnot Huffmanova kódu ukládá délky jednotlivých hodnot kódu. Následující příklad ukazuje, jak ze znalosti délek hodnot Huffmanova kódu lze tyto hodnoty zjistit.

Příklad. Mějme 8 znaků *a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*, které jsou kódovány Huffmanovým kódem s délkami kódu (v bitech):

Znak	a	b	c	d	e	f	g	h
Délka kódu	3	3	3	4	3	2	3	4

Máme určit kódy jednotlivých znaků.

1. Nejprve zjistíme počty kódů pro jednotlivé délky:

Délka kódu v bitech	Počet kódů
1	0
2	1
3	5
4	2

2. V dalším kroku spočítáme počáteční hodnoty pro jednotlivé délky kódů dle algoritmu:

```
Kód=0;
Cyklus od Délka = 1 do Maximální_délka_kódu
  { Počáteční_hodnota_kódu[Délka] = Kód;
    Kód = 2 * ( Kód + Počet_kódů[Délka] );  }
```

V našem příkladu je Maximální_délka_kódu = 4. Vypočítané počáteční hodnoty pro jednotlivé délky kódu jsou v následující tabulce

Délka kódu v bitech	Počáteční hodnota kódu
1	0
2	0

3	2
4	14

3. Nyní stačí jednotlivým znakům o stejné délce Huffmanova kódu postupně přiřadit hodnoty od stanovené počáteční hodnoty postupně zvyšované o 1. Sestavení Huffmanova kódu pro náš příklad ukazuje následující tabulka:

Znak	Přiřazená hodnota	Huffmanův kód
a	2	010
b	3	011
c	4	100
d	14	1110
e	5	101
f	0	00
g	6	110
h	15	1111

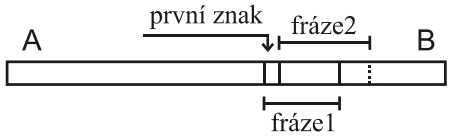
Vzhledem k tomu, že konstrukce Huffmanova kódu není obecně jednoznačná, je kód možné stanovit z délek jen za předpokladu, že při jeho vytváření byl dodržen určitý postup. Konkrétně se zde předpokládá, že kód byl konstruován takovým způsobem, že:

- Kratší kódy mají vždy menší hodnoty než delší kódy (v předchozím příkladu hodnota kódu znaku *f* je menší než hodnota kódu znaku *a*).
- Hodnoty kódů stejné délky jsou přiřazeny tak, že odpovídají uspořádání symbolů, které reprezentují (například znak *c* z předchozího příkladu je v abecedním uspořádání před znakem *e* a proto i hodnota jeho kódu 4 je menší než hodnota kódu 5 znaku *e*).

Při kompresi dále může nastat případ, že vstupní text nelze efektivně zkomprimovat. Například vstupním textem je již zkomprimovaný soubor. Kompresní programy proto běžně text ukládají do výstupního souboru bez komprese, jestliže by se jeho délka kompresí zvětšila místo toho, aby se zmenšila.

Způsob komprese založený na metodě LZSS a Huffmanově kódování používá řada kompresních programů (Zip, GZip, ARJ, RAR), neboť je rychlý a poskytuje dobré výsledky. Tyto programy běžně uživateli poskytují možnost vybrat si, zda preferuje účinnější a zároveň pomalejší kompresi nebo naopak rychlou, i když méně účinnou kompresi.

Jedna z možností, jak při kompresi dosáhnout vyšší účinnost, je postup označovaný jako "lazy matching". Při něm fráze nalezená v daném kompresním kroku (na obrázku *fráze1*) se nekóduje, ale pokračuje se v hledání, zda existuje fráze, která začíná v okně *B* o jeden znak dále (na obrázku *fráze2*). Jestliže ano, srovnají se jejich délky. Je-li *fráze2* delší než *fráze1*, kóduje se samostatně první znak okna *B* a za ním *fráze2*. Není-li delší, kóduje se *fráze1*. Vyhledání druhé fráze zvyšuje celkovou účinnost komprese, ale za cenu jejího zpomalení.



Obrázek 17 Vyhledání další fráze pro zvýšení účinnosti komprese

Zvýšení rychlosti komprese se dosahuje nastavením prahové hodnoty délky fráze (například 8 znaků). Jestliže program v průběhu vyhledávání nalezl frázi, která má aspoň prahovou délku, s nalezenou frází se spokojí a prohledávání okna v daném kompresním kroku ukončí.

Povšimněme si dále, že s rostoucím ofsetem se zvyšuje počet bitů potřebný pro jeho zakódování. Hodnota ofsetu 20 je kódována 8 bity, zatímco ofset 20000 je kódován již 18 bity. Kdybychom měli

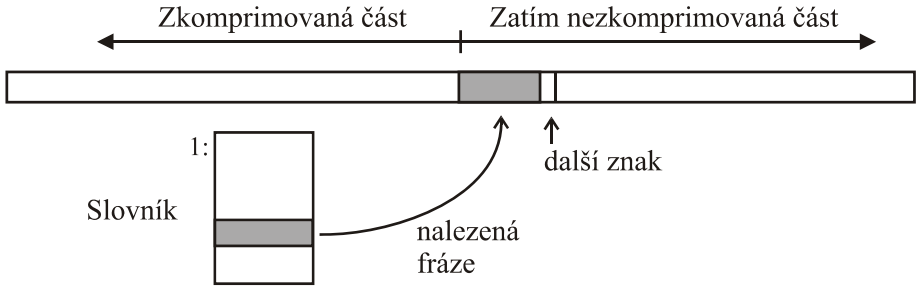
frázi délky 3 s ofsetem 20000, potřebujeme k jejímu zakódování ještě dalších 7 bitů na kód délky, tedy celkem 25 bitů. Kódovat takovou frázi se již nevyplatí. Krátkou frázi s velkým ofsetem proto kompresní programy ignorují a v daném kompresním kroku kódují jen znak.

3.2 Metody LZ78

Metody třídy LZ78 si z frází, které se vyskytly v doposud zkomprimované části textu, vytváří v paměti slovník. Fráze jsou ve slovníku označeny pořadovými čísly, první fráze má číslo 1. V každém kompresním kroku se ve slovníku hledá nejdelší fráze shodná s textem na vstupu. Po nalezení fráze se kóduje dvojice

(číslo_fráze, další_znak)

kde *číslo_fráze* označuje pořadové číslo nalezené fráze ve slovníku a *další_znak* označuje znak na vstupu, který následuje za frází.



Obrázek 18 Princip komprese LZ78

Po zakódování dvojice je zároveň do slovníku zařazena nová fráze, která je složena z právě použité fráze a znaku za ní. Má tedy tvar

nalezená_fráze + další_znak

a je o 1 znak delší než je délka kódované fráze.

Jestliže v daném kompresním kroku nebyla ve slovníku nalezena žádná fráze shodující se s textem na vstupu, kóduje se dvojice v níž místo čísla fráze je nula:

(0, znak_na_vstupu)

a do slovníku se zařadí fráze délky 1 obsahující právě zakódovaný znak.

Je zřejmé, že v každém kompresním kroku se zakóduje text o 1 znak delší než je délka nalezené fráze nebo přinejmenším 1 znak, jestliže ve slovníku nebyla nalezena žádná fráze shodná se vstupním textem.

Na počátku komprese je slovník prázdný. V každém kompresním kroku je do něho zařazena nová fráze. Velikost slovníku je omezená. Jakmile dojde k jeho zaplnění, slovník se jednoduše zruší (vymaže) a komprese dalšího textu začíná opět s prázdným slovníkem. Tento proces se cyklicky opakuje až do zakódování celého textu.

Příklad. Budeme komprimovat text *abacadacbacadba*.

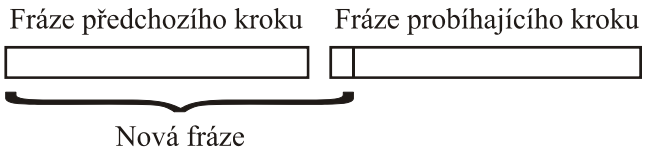
		Nová fráze	
Vstupní text	Výstup	Číslo	Fráze
abacadacbacadba	(0,a)	1	a
bacadacbacadba	(0,b)	2	b
acadacbacadba	(1,c)	3	ac
adacbacadba	(1,d)	4	ad
acbacadba	(3,b)	5	acb

acadba	(3,a)	6	aca
dba	(0,d)	7	d
ba	(2,a)	8	ba

3.2.1 Metoda LZW a její implementace

Ze všech metod třídy LZ78, které vznikly modifikováním původního přístupu popsaného v předchozí části, má největší význam metoda LZW. Tato vychází z myšlenky, že není efektivní kódování znaku za frází, a kóduje jen fráze. Protože metoda LZW nemůže kódovat samostatné znaky ale jen nalezené fráze, nemůže být slovník na začátku komprese prázdný. Je nutné ho inicializovat všemi frázemi délky 1, tj. je nutné do něho zařadit všechny znaky, které se mohou vyskytnout v textu.

V každém kompresním kroku, vyjma prvního kroku, se do slovníku zařadí nová fráze, která je složena z fráze použité v předchozím kompresním kroku a prvního znaku fráze použité v probíhajícím kompresním kroku.

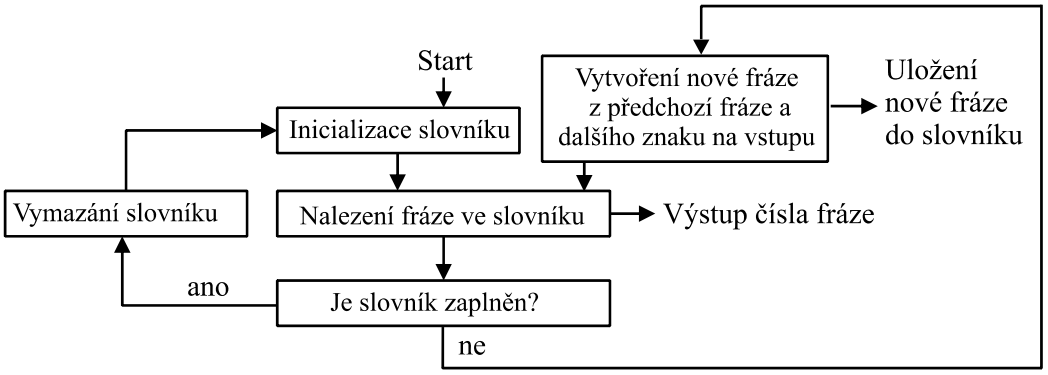


Obrázek 19 Vytvoření nové fráze

Na vytvoření fráze se při kompresi můžeme dívat i tak, že je vytvořena z fráze použité v předchozím kompresním kroku a znaku, který je právě na vstupu. Při dekompresi ale můžeme frázi vytvořit jen ze dvou naposledy použitých frází, neboť dekompresní program disponuje jen textem, který byl již dekomprimován.

Zatímco u původního návrhu komprese LZ78 byly fráze číslovány od hodnoty 1, neboť 0 byla příznakem nenalezené fráze, metoda LZW již tento příznak nepotřebuje a proto fráze čísluje od nuly.

Postup komprese LZW a vytváření slovníku ukazuje následující obrázek:



Obrázek 20 Schéma komprese LZW

Příklad. Provedeme kompresi textu *abacdacacadaad*, který je složen jen ze čtyř znaků *a*,*b*,*c*,*d*.

Na začátku slovník inicializujeme všemi znaky textu:

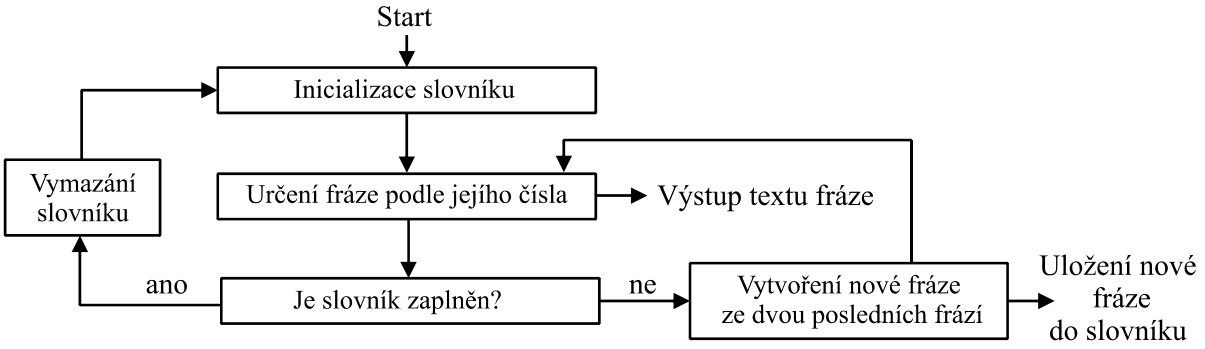
Číslo fráze	Fráze
0	a
1	b
2	c
3	d

Jednotlivé kroky komprese:

		Nová fráze		Kódovaná fráze	
Krok	Vstupní text	Číslo	Fráze	Fráze	Výstup
1	abacdacacadaad	-	-	a	0
2	bacdacacadaad	4	ab	b	1
3	acdacacadaad	5	ba	a	0
4	cdacacadaad	6	ac	c	2
5	dacacadaad	7	cd	d	3
6	acacadaad	8	da	ac	6
7	acadaad	9	aca	aca	9
8	daad	10	acad	da	8
9	ad	11	daa	a	0
10	d	12	ad	d	3

Výstupem komprese jsou čísla frází 0,1,0,2,3,6,9,8,0,3.

Dekomprese probíhá obdobným způsobem jako komprese a dekompresní program si vytváří zcela identický slovník jako kompresní program. Z čísel frází uložených kompresním programem se dekóduje původní text. Postup dekomprese ukazuje následující obrázek:



Obrázek 21 Schéma dekomprese LZW

Příklad. Provedeme dekompresi textu uvedeného v předchozím příkladu, tedy na vstupu máme posloupnost čísel 0,1,0,2,3,6,9,8,0,3. Inicializace slovníku je stejná jako při kompresi a proto tuto část přeskočíme a přejdeme přímo k dekompresi:

			Nová fráze	
Krok	Vstup	Výstup	Číslo	Fráze
1	0	a	-	-
2	1	b	4	ab
3	0	a	5	ba
4	2	c	6	ac
5	3	d	7	cd
6	6	ac	8	da
7	9	aca	9	aca
8	8	da	10	acad
9	0	a	11	daa
10	3	d	12	ad

Zastavme se v uvedeném příkladu u kroku 7, ve kterém je dekódována fráze s číslem 9, ačkoliv je v tomto okamžiku ve slovníku fráze s nejvyšším číslem 8. Tento specifický případ je způsoben rozdílem v postupu vytváření frází mezi kompresním a dekompresním algoritmem. Zatímco při

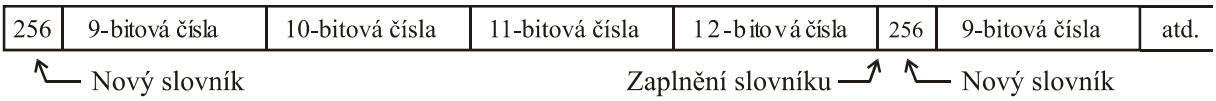
kompresi je v každém kompresním kroku nejprve vytvořena nová fráze a teprve pak se provede vlastní kódování, při dekompresi je v kompresním kroku nejdříve provedeno dekódování a teprve pak je vytvořena nová fráze. Jestliže kompresní program právě vytvořenou frázi ihned použil k zakódování (krok 7 v příkladu komprese), nemá tuto frázi dekompresní program ještě ve slovníku. To může nastat jen v případě, kdy nově vytvořená fráze (fráze 9) začíná stejným znakem jako předchozí fráze (fráze 8). Proto dekompresní program si může chybějící frázi (fráze 9) doplnit tak, že vezme naposledy vytvořenou frázi (fráze 8) a na její konec dá počáteční znak této fráze.

Na závěr se budeme věnovat praktické implementaci metody LZW. Slovník se používá v rozsahu 4096 položek, tedy pořadová čísla frází jsou v intervalu ⟨0,4095⟩. Na počátku je slovník inicializován všemi 256 možnými hodnotami, které lze uložit do jednoho bytu, tj. inicializací jsou ve slovníku vytvořeny fráze 0 až 255. Další dvě pořadová čísla 256 a 257 neoznačují fráze, ale slouží k řízení dekompresního programu. Pořadové číslo 256 označuje nový slovník. Kompresní program tuto hodnotu zapisuje na výstup na začátku komprese a pak pokaždé, když maže stávající slovník a vytváří nový. Pro dekompresní program je to signál, že má rovněž vytvořit nový slovník. Hodnota 257 označuje konec textu a pro dekompresní program je povelem k ukončení dekomprese. Zbývající hodnoty 258 až 4095 jsou pro nové fráze.

Metoda LZW nepoužívá na rozdíl od metod LZ77 k zakódování výstupu žádné statistické kódování. Čísla frází na výstup zapisuje v binární podobě s pevně stanoveným počtem bitů. Aby se zvýšila efektivita kódování, mění se tento počet od 9 do 12 bitů v závislosti na momentálním zaplnění slovníku, tj. dle nejvyššího čísla fráze ve slovníku.

Zaplnění slovníku - nejvyšší číslo fráze	Délka výstupní hodnoty v bitech
do 511	9
512 až 1023	10
1024 až 2047	11
2048 až 4095	12

Strukturu výstupního souboru ukazuje následující obrázek:

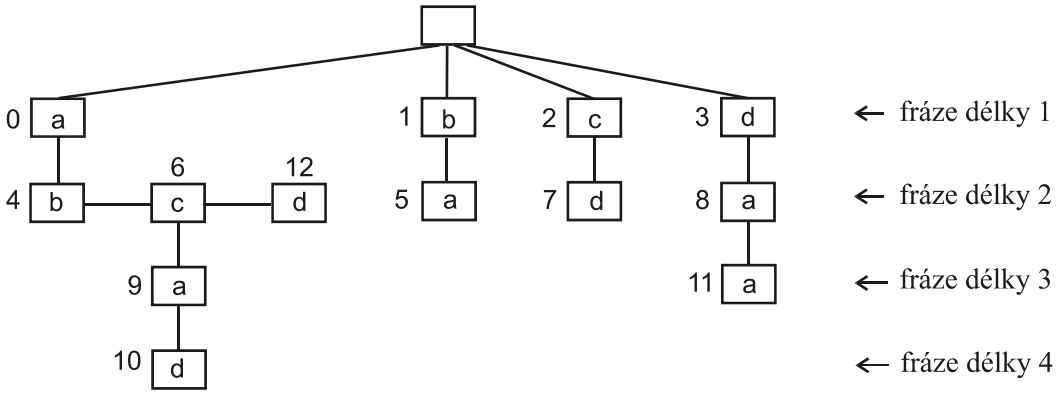


Obrázek 22 Struktura výstupního souboru při kompresi LZW

Vlastní slovník se při kompresi realizuje tabulkou se třemi sloupci a 4096 řádky. Řádky jsou číslovány od 0 do 4095 a tato čísla jsou zároveň čísly frází. V prvním sloupci je vždy znak fráze, ve druhém sloupci je číslo s řádkem obsahující následující znak ve frázi, ve třetím sloupci je číslo řádku obsahující následující znak v jiné takové frázi.

Příklad. Popišme slovník vytvořený v průběhu komprese textu *abacdacacadaad* z příkladu na začátku této části. Graficky se dá vyjádřit stromovou strukturou na následujícím obrázku. Uzly vyjma horního kořenového uzlu reprezentují jednotlivé fráze. Číslo fráze je vždy u příslušného uzlu. Hledání nejdelší fráze odpovídající textu na vstupu začíná kořenem. Nechť je například na vstupu text *acab*. Najde se následovník kořene odpovídající prvnímu znaku na vstupu, v našem případě je to uzel s číslem 0 obsahující znak *a*. Hledání přejde to tohoto uzlu a vezme se druhý znak na vstupu, tím je *c*. Zkusí se, zda se tento znak shoduje se znakem v následníku uzlu 0. Tím je uzel 4 se znakem *b*. Shoda tu není a hledání pokračuje dalšími uzly na horizontální úrovni (jsou-li nějaké). V našem případě se najde uzel 6 se znakem *c*. Hledání přejde do tohoto uzlu a vezme se další znak na vstupu, což je *a*. Zkusí se, zda se tento znak shoduje se znakem v následovníku současného uzlu, kterým je uzel 9. Zde je shoda splněna a hledání přejde do uzlu 9. Vezme se další znak na vstupu, což je *b*. Tento znak se neshoduje se

znakem v dalším uzlu 10. Protože tento uzel nemá žádného souseda na horizontální úrovni, je hledání ukončeno. Poslední dosažený uzel je 9, což označuje číslo nalezené fráze.



Popsaná stromová struktura se snadno realizuje tabulkou se třemi sloupci. Řádky tabulky jsou očíslovány počínaje číslem 0 a odpovídají příslušným uzlům stromu. Na každém řádku je v prvním sloupci znak v uzlu stromové struktury, v druhém sloupci je číslo následovníka a ve třetím sloupci je číslo souseda na horizontální úrovni, tedy jiného následovníka. Začátek tabulky (řádky 0 až 3) tvoří inicializační část obsahující všechny možné znaky.

	Znak	Následovník	Jiní následovníci
0:	a	4	
1:	b	5	
2:	c	7	
3:	d	8	
4:	b		6
5:	a		
6:	c	9	12
7:	d		
8:	a	11	
9:	a	10	
10:	d		
11:	a		
12:	d		

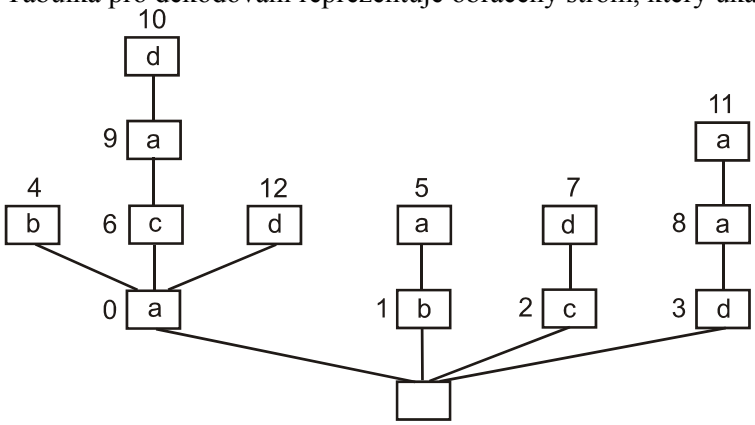
Popišme postup vyhledání nejdelší fráze pro text *acab*. Začínáme na řádku 0, kde je počáteční znak *a* vstupního textu. Zde s pomocí druhého sloupce najdeme následovníka - řádek 4. Na něm je znak *b*, který ale není druhým znakem fráze. Podíváme se tedy na tomto řádku do třetího sloupce po jiném následovníku. Zde je číslo řádku 6, ve kterém je již požadovaný druhý znak *c*. Máme nyní frázi *ac* délky 2 a hledáme, zda existuje ještě delší fráze. Na řádku 6 ve druhém sloupci najdeme číslo řádku 9, na kterém je znak *a*, který v tomto případě odpovídá třetímu znaku vstupního textu. Máme již frázi *aca*. Hledáme, zda existuje ještě delší fráze. Na řádku 9 ve druhém sloupci najdeme číslo řádku 10. Na něm je ale znak *d*, tedy nikoliv *b*,a navíc ve třetím sloupci zjistíme, že jiný následovník už není. Nejdelší fráze textu *acab* obsažená ve slovníku je tedy fráze *aca* s číslem 9.

Dekompresní program si vytváří ještě jednodušší reprezentaci slovníku. Je to tabulka se dvěma sloupci a 4096 řádky. Řádky jsou opět číslovány od 0 do 4095 a opět tato čísla označují čísla frází. Na každém řádku je v prvním sloupci znak fráze a v druhém sloupci vždy číslo řádku, kde se nachází předchozí znak ve frázi.

Příklad. Vezměme tabulku vytvořenou při dekompresi textu z předchozího příkladu. Předpokládejme, že máme dekódovat frázi s číslem 10. Na tomto řádku najdeme poslední znak fráze *d*. Ve druhém sloupci najdeme číslo řádku 9, kde je předchozí znak. Je to znak *a*. Máme již dekódován text *da*. Pokračujeme dalším předchůdcem uvedeným ve druhém sloupci na řádku 9. Je to řádek 6, na kterém najdeme další znak *c*. V této fázi je dekódován text *dac*. Na tomto řádku najdeme číslo řádku 0 dalšího předchůdce. Na řádku 0 je znak *a*. Dostali jsem text *daca* a tím dekódování končí, neboť na řádku 0 už žádný další předchůdce není uveden.

	Znak	Předchůdce
0:	a	
1:	b	
2:	c	
3:	d	
4:	b	0
5:	a	1
6:	c	0
7:	d	2
8:	a	3
9:	a	6
10:	d	9
11:	a	8
12:	d	0

Je zřejmé, že tímto postupem se dekódují znaky fráze v opačném pořadí, od posledního znaku fráze k prvnímu. Proto při dekódování fráze se znaky dočasně ukládají na zásobník a po zkompletování fráze se v opačném pořadí odeberou ze zásobníku a zašlou na výstup.
Tabulka pro dekódování reprezentuje obrácený strom, který ukazuje následující obrázek.



Z popisu používané implementace slovníků je zřejmé, že metoda LZW má malé nároky na paměť. Pokud budeme vycházet z toho, že pro uložení znaku potřebujeme jeden byte a pro uložení čísla fráze nám stačí 2 byty, zabírá slovník při kompresi 20 KB a při dekompresi 12 KB.

Sestavení programu pro kompresi metodou LZW je výrazně snadnější než u ostatních metod. Program se dá v poměrně krátkém čase realizovat z uvedeného popisu. Navíc komprese i dekomprese metodou LZW je rychlá. Její účinnost je ale menší než u jiných metod popsaných v tomto skriptu. Srovnávací tabulka účinnosti komprese různými metodami je uvedena na konci následující kapitoly.

V minulosti byla LZW komprese pro svoji rychlost a nenáročnost na implementaci i na paměť velmi oblíbená. Hojně se používala při kompresi dat ukládaných na disk. V současnosti je ale

nahrazována účinnějšími metodami. Mezi nejznámější programy používající LZW metodu patří kompresní program *compress* operačního systému UNIX, který je ale postupně nahrazován účinnějším programem *gzip* pracujícím na principu LZ77. Dále je tato metoda použita ke kompresi obrazu ve formátu GIF. I zde její význam rychle zaniká, neboť formát GIF je dnes nahrazován formátem JPEG, který používá zcela odlišný způsob komprese. Určitou překážkou v použití LZW metody je také okolnost, že je patentována, naštěstí ale jen v USA.

3.3 Komprese blokovým tříděním

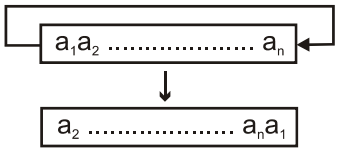
Doposud popsané kompresní metody lze dnes již považovat za klasické. Komprese je ale stále aktuální záležitostí a tvůrci kompresních programů se snaží o aplikaci nových přístupů, které by vedly k účinnější a rychlejší kompresi. Na závěr celé části věnované bezeztrátovým metodám popíšeme jeden novější způsob komprese s vysokou účinností.

Metoda komprese blokovým tříděním je založena na transformaci bloku textu o n znacích $a_1a_2....a_n$ na blok stejné délky vytvořený záměnou pořadí (permutací) znaků původního bloku $a_1a_2....a_n$. Transformace je volena tak, že splňuje vlastnosti:

- je reverzibilní, tj. text se dá zpětně převést do původní podoby
- po transformaci se stejné znaky velmi často vyskytují za sebou, což umožňuje jejich efektivní kódování.

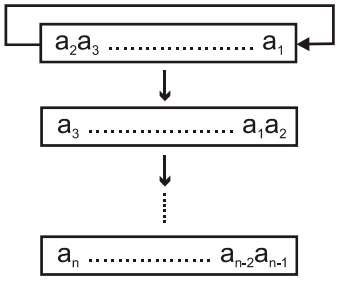
Popis algoritmu komprese:

1. Text v komprimovaném bloku $a_1a_2....a_n$ cyklicky posuneme doleva, čímž vytvoříme cyklickou rotaci bloku textu.



Obrázek 23 Vytvoření rotace bloku textu

Budeme-li pokračovat v cyklických posunutích, dostaneme dohromady n-1 různých rotací bloku textu.



Obrázek 24 Vytvoření zbývajících rotací bloku textu

Spolu s výchozím blokem textu máme celkem n bloků textu.

Jako příklad vezměme text o 7 znacích *barbara*. Jeho cyklické rotace jsou *arbarab*, *rbaraba*, *barabar*, *arabarb*, *rabarba*, *abarbar*.

Tyto rotace abecedně setřídíme, zapíšeme do řádků a řádky očíslujeme čísly od 0 do n-1.

Číslo řádku	Setříděné cyklické rotace textu
0	abarbar
1	arabarb
2	arbarab

3	barabar
4	barbara
5	rabarba
6	rbaraba

2. Setříděné rotace bloku o *n* znacích tvoří čtvercovou matici znaků *M* řádu *n*. Každý sloupec matice obsahuje právě všechny znaky komprimovaného bloku textu, ale v jiném pořadí.. Jako požadovanou transformaci vezmeme poslední sloupec této matice, který označíme *L*. V našem příkladu je poslední sloupec L=rbbraaa. Dále najdeme číslo řádku, na kterém je původní text, označme ho *I*. V našem příkladu je text *barbara* na řádku s číslem I=4.
3. V další fázi vezmeme všechny znaky, které se mohou vyskytnout v textu, a uspořádáme je abecedně do posloupnosti, kterou si označíme *Z*. V našem příkladu použijeme jen znaky, které se skutečně vyskytují v textu, tedy vezmeme posloupnost tří znaků Z={a,b,r}.
4. Nyní vezmeme transformovaný blok znaků a postupně z něho zleva odebíráme jednotlivé znaky. U každého znaku zjistíme jeho pozici v posloupnosti *Z* a číslo této pozice zakódujeme na výstup. Pozice jsou číslované tak, že první znak v posloupnosti je 0, druhého znaku je 1 atd.
   Po zakódování každého znaku provedeme cyklickou rotaci posloupnosti *Z* tak, aby právě zakódovaný znak se dostal na počáteční pozici. Kódování v našem příkladu bude:

Posloupnost Z	Transformovaný blok	Výstupní kód K
a,b,r	rbbraaa	2
r,a,b	bbraaa	2
b,r,a	braaa	0
b,r,a	raaa	1
r,a,b	aaa	1
a,b,r	aa	0
a,b,r	a	0

Výstupem komprese je:

- Číslo řádku *I* s původním textem. V našem příkladu je to číslo 4.
- Hodnoty výstupního kódu *K*. V našem příkladu jsou to hodnoty K={2,2,0,1,1,0,0}.
  Pro jejich uložení se použije aritmetické nebo Huffmanovo kódování.

Popis algoritmu dekomprese:

1. Ze zkomprimovaného souboru přečteme číslo řádku *I*, na kterém je v matici *M* původní text, a pomocí aritmetického dekodéru dekódujeme hodnoty uloženého kódu *K*. V našem příkladu to bude I=4 a K={2,2,0,1,1,0,0}.
2. Vezmeme posloupnost znaků abecedy *Z* a z kódu *K* získáme výchozí transformovaný blok textu, což je poslední sloupec *L* matice znaků *M*. V našem příkladu bude dekódování:

Posloupnost Z	Kód K	Poslední sloupec matice L
a,b,r	2	r
r,a,b	2	rb
b,r,a	0	rbb
b,r,a	1	rbbr
r,a,b	1	rbbra
a,b,r	0	rbbraa
a,b,r	0	rbbraaa

Postup dekódování kódu *K* na výchozí transformovaný text je zřejmý z uvedeného příkladu. Pomocí hodnoty kódu se najde příslušný znak v posloupnosti *Z* a cyklickou rotací se tento znak v posloupnosti *Z* přesune na její začátek stejným způsobem, jak se to dělalo při kódování.

3. Poslední a nejpodstatnější krok je zpětná transformace bloku textu, jejímž výsledkem bude původní komprimovaný text.

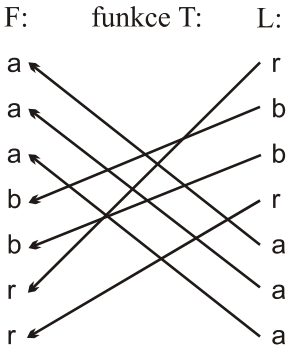
   Známe poslední sloupec *L* matice znaků *M*. O prvním sloupci matice *M* víme, že obsahuje stejné znaky jako poslední sloupec, a rovněž o něm víme, že znaky jsou v něm abecedně setříděné. Tudíž setříděním posledního sloupce *L* podle abecedy získáme první sloupec matice *M*, označme ho *F*. V našem příkladu setříděním L=rbbraaa dostaneme F=aaabbrr. Nyní sestavíme funkci *T*, která mapuje řádky posledního sloupce *L* na řádky prvního sloupce *F* s vlastnostmi:

   - Řádek sloupce *L* se vždy zobrazí na řádek *F* se stejným znakem
   - Na řádcích *L*, na kterých je stejný znak, je funkce *T* rostoucí. Tj. pro indexy řádků *i*,*j*, pro než je

     L[i]=L[j] (znaky na řádcích *i* a *j* jsou stejné),

     pro funkční hodnoty (čísla řádků) platí T(i)<T(j) právě když je i<j.
   - Funkce je prostá (bijektivní)

   V našem příkladu ukazuje konstrukci funkce T následující obrázek:



Obrázek 25 Zobrazení stejných znaků

Tedy T(0)=5, T(1)=3, T(2)=4, T(3)=6, T(4)=0, T(5)=1, T(6)=2.

Rekonstrukce původního textu $a_1a_2a_3...a_n$ se provádí odzadu. Poslední znak a_n zjistíme snadno. Známe index *I* řádku s původním textem a známe poslední sloupec *L*, odtud

$a_n=L[I]$

Nyní vezmeme řádek

$i_1=T[I]$

Na tomto řádku je ve sloupci *F* stejný znak. Tento řádek je cyklickou rotací textu a předchozí znak najdeme v jeho posledním sloupci, tedy

$a_{n-1}=L[i_1]$

Tento postup se cyklicky opakuje

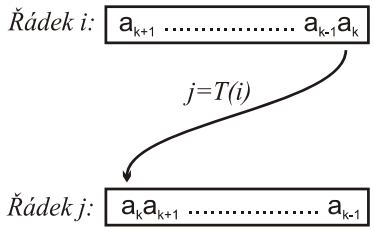
$i_2=T[i_1]$, $a_{n-2}=L[i_2]$

$i_3=T[i_2]$, $a_{n-3}=L[i_3]$

$i_4=T[i_3]$, $a_{n-4}=L[i_4]$ ... atd.

Uvedený princip je znázorněn na následujícím obrázku. Ukazuje případ, kdy máme již dekódován znak a_k a nyní dekódujeme předchozí znak a_{k-1}. Ten je na stávajícím řádku v předposledním

sloupci, který ale neznáme. Pomocí zobrazení *T* najdeme cyklickou tohoto řádku, v němž hledaný znak a_{k-1} je v posledním sloupci, čímž ho můžeme dekódovat.



Obrázek 26 Nalezení předchozího znaku pomocí funkce T

Vezměme náš příklad, kde I=4.

a = L[4]

0 = T(4), r = L[0]

5 = T(0), a = L[5]

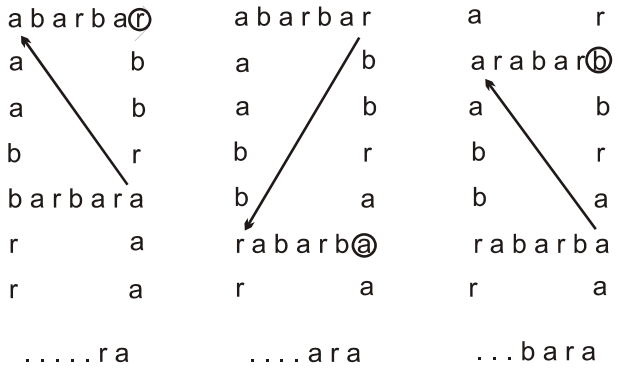
1 = T(5), b = L[1]

3 = T(1), r = L[3]

6 = T(3), a = L[6]

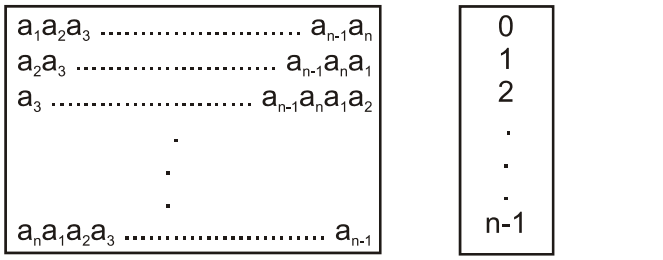
2 = T(6), b = L[2]

Postup dekódování prvních čtyř znaků ukazuje následující obrázek:



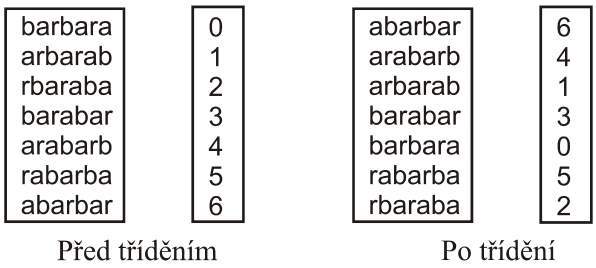
Na závěr několik poznámek k implementaci metody. Metoda blokového třídění je založena na transformaci bloku textu do podoby, ve které se často objevují sekvence stejných znaků za sebou. Proto účinnost metody roste s délkou textu. Ideální je provést transformaci a kódování celého textu najednou. Značně dlouhé texty ale bude nutné rozdělit do bloků a jednotlivé bloky zpracovat samostatně. Délku bloku volíme dostatečně velkou, řádově statisíce znaků.

Jádrem metody je setřídění cyklických rotací textu v bloku. Vzhledem k tomu, že blok textu bývá značně dlouhý, byla by vlastní matice rotací bloku neúnosně rozsáhlá. Proto místo ní se používá sloupcový vektor s čísly, jež vyjadřují počet, kolik znaků rotace v každém řádku původní matice obsahuje. Řádek s nerotovaným textem je reprezentován číslem 0, řádek s jedním rotovaným znakem číslem 1 až poslední řádek bude reprezentován číslem n-1.



Obrázek 27 Vektor čísel rotací

Třídění se provádí na sloupcovém vektoru výměnami. Vezmou čísla na dvou řádcích, z nich se dovodí, které rotace reprezentují, tyto se srovnají a na základě toho provede případná výměna čísel ve sloupcovém vektoru.
Matice a sloupcový vektor před tříděním a po třídění v našem příkladu ukazuje následující obrázek:

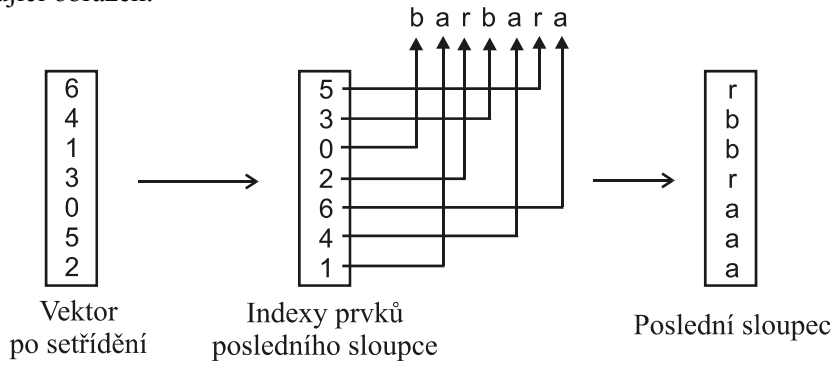


Zbývá určit, jak ze setříděného sloupcového vektoru dostaneme požadovaný poslední sloupec setříděné matice, tj. sloupec L. Ten zjistíme vypočítáním indexů prvků posledního sloupce vztahem

index_prvku_posledního_sloupce = (číslo_rotace + n-1) mod n

Tyto udávají indexy prvků posledního sloupec, které mají v bloku textu.

V našem příkladu indexy budou 5,3,0,2,6,4,1 a odtud poslední sloupec je *rbbraaa*, jak ukazuje následující obrázek:



Srovnání kompresních metod

Některé kompresní metody o něco lépe komprimují binární data, jiné naopak textové soubory apod. Aby bylo možné srovnat vzájemnou účinnost různých kompresních metod a různých programů, byl sestaven referenční vzorek obsahující 14 souborů s různými typy dat (anglické texty, obrazová data, zdrojové texty programů, přeložené programy). Tento souhrn se nazývá *Calgary corpus* a účinnost komprese se počítá jako průměr z komprese všech 14 souborů.

Následující tabulka ukazuje výsledky dosažené kompresními programy, které reprezentují jednotlivé kompresní metody popsané v předchozích částech. V tabulce jsou uvedeny metody:

- Statistická komprese metodou PPMC (maximální délka kontextu 4, program *Model*-2).

- Slovníková komprese LZ77 (metoda LZSS, program *gzip*).
- Slovníková komprese LZ78 (metoda LZW, program *compress*).
- Komprese blokovým tříděním (testovací program realizující blokové třídění, přičemž pro uložení je použito Huffmanovo kódování).

Doby komprese a dekomprese jsou v tabulce uvedeny ve vteřinách.

Metoda	Doba komprese	Doba dekomprese	Kompresní poměr v %
PPMC	603,2	614,1	30,4
LZ77 - LZSS	42,6	4,9	33,9
LZ78 - LZW	9,6	5,2	45,4
Blokové třídění	51,1	9,4	30,9

4. Komprese obrazu a zvuku

Další velmi výraznou oblastí aplikace kompresních algoritmů je komprese rastrových obrazů (nazývaných rovněž bitové mapy), neboť tyto při větším rozlišení a počtu barevných odstínů zabírají při uložení na vnější paměti mnoho místa. Než přejdeme k popisu způsobů komprese obrazů, uveďme několik základních informací o reprezentaci barev.

Modely barev

Nejběžnějším modelem barev je aditivní model RGB. Je založen na skutečnosti, že většinu barevných odstínů (nikoliv ale všechny) lze vytvořit smísením červené (Red), zelené (Green) a modré (Blue) barvy. Množství jednotlivých tří barev při smísení určuje jak barevný odstín, tak jas. I když lidské oko má různou citlivost pro jednotlivé barvy, zobrazovací zařízení jsou tomu přizpůsobena, aby při smísení stejného podílu všech tří barev vzniklo monochromatické světlo, tj. určitá úroveň šedého odstínu.

Základní reprezentace modelu barev RGB používá pro vyjádření podílu každé ze tří základních barev hodnoty 0 až 255. Jsou to hodnoty, které lze právě uložit na jednom bytu. Celkem pro vyjádření každého bodu rastrového obrazu jsou zapotřebí 3 byty. Uvedená reprezentace barev se nazývá *True Color* a umožňuje zobrazit 256^3 barevných odstínů, což je přes 16 miliónů odstínů.

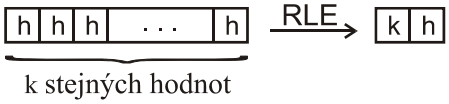
Další reprezentace RGB modelu je *High Color* (také označována jako *Hicolor*). Každý bod je reprezentován dvěma byty, tedy 16 bity. Každá barva je vyjádřena hodnotou uloženou na 5 bitech, což celkem dává 2^{15}, tj. 32768 barevných odstínů, nebo je těchto 16 bitů rozděleno na 5+6+5 bitů a v tomto případě vzniká 65536 možných barevných odstínů.

Při použití menšího počtu barevných odstínů, tj. 256 nebo 16 nebývá barva každého bodu vyjádřena přímo barevnými složkami, ale prostřednictvím palety. Paleta je tabulka obsahující vybraných 256 nebo 16 barevných odstínů v RGB modelu. Barva každého bodu obrazu je stanovena pomocí čísla příslušného barevného odstínu v paletě. U 256 barev je toto číslo v rozmezí 0 až 255 a u 16 barev je v rozmezí 0 až 15. To znamená, že u 256 barev je pro každý bod rastrového obrazu zapotřebí jeden byte a u 16 barev jsou pro každý bod zapotřebí 4 bity.

Existují ještě jiné modely barev. Pro kompresi obrazu jsou zejména významné modely YUV, ve kterých je barevný odstín reprezentován pomocí jasové složky (Y) a dvou barevných složek (U a V). Převody mezi RGB modelem a YUV modely jsou poměrně jednoduché. Jsou vyjádřeny pomocí transformační matice.

4.2 Bezeztrátová komprese obrazu

Úplně nejjednodušší kompresní technikou je RLE (Run Length Encoding). Tato technika za sebou více následujících stejných hodnot kóduje do dvojice počet a hodnota.



Obrázek 28 Princip metody RLE

Komprese RLE je použitá v obrazovém formátu PCX. Vzhledem k tomu, že tento formát byl původně navržen pro 16 barev, byla to v tomto případě poměrně účinná komprese, neboť u obrazů s malým počtem barevných odstínů často následuje více bodů se stejnou barvou za sebou.

Poměrně často používanou kompresní technikou v kompresi obrazu je LZW. Je obsažena ve formátech GIF a TIFF. Kdysi velmi používaný formát GIF je ale založen na paletě s 256 barevnými odstíny a jeho význam proto dnes už značně klesá. TIFF je velmi obecný a také značně komplikovaný

formát. Je v něm implementováno více kompresních technik, z bezeztrátových je to vedle LZW rovněž modifikace Huffmanova kódování a JBIG.

JBIG je na rozdíl od ostatních zmíněných technik RLE a LZW, které jsou obecnými kompresními metodami, kompresní technika speciálně vyvinutá pro bezeztrátovou kompresi obrazu. Je určena jen pro monochromatické obrazy, jako jsou faxové dokumenty, rentgenové snímky apod. Je zejména velmi účinná při kompresi obrazů se dvěma barvami (bílou a černou) nebo pro obrazy jen s malým počtem šedých odstínů (do 256). Tento fakt je vyjádřen i písmeny BI v názvu metody, které jsou odvozeny od označení Bi-level Image.

4.3 Ztrátová komprese obrazu - JPEG

Při kompresi obrazu pomocí metody LZW obsažené ve formátu GIF se dosahuje typický kompresní poměr 1:2. Existují sice účinnější bezeztrátové metody než je LZW, ale jejich použití by velikost uloženého obrazu již výrazněji nezmenšilo. Jediná cesta k dosažení podstatně vyššího kompresního poměru je ztrátová komprese.

Rastrový obraz je sám o sobě nepřesný. Původní spojitý obraz je rozdělen do diskrétních bodů, spojité barevné spektrum je nahrazeno konečným počtem barevných odstínů. Lidské oko ale řadu odlišností v obrazu nepostřehne. Na tomto faktu je založena ztrátová komprese. Při ztrátové kompresi dochází k zanedbání nebo změnění některých údajů v obrazu takovým způsobem, aby to mělo co nejmenší vliv na jeho kvalitu, tj. aby při pozorování lidským okem nebyly změny v obrazu patrné.

Pro ztrátovou kompresi obrazu byl vyvinut standard JPEG. Jeho název je odvozen od označení pracovní skupiny Joint Photographics Experts Group, která se řadu let zabývala přípravou tohoto standardu.

JPEG komprimuje obrazy s 24 bitovou reprezentací barev (True Color). Nepoužívá obvyklý model barev RGB, ale model YCbCr obsahující jasovou složku *Y* a dvě barevné složky *Cb* a *Cr*. Oba modely barev lze vzájemně převádět pomocí transformační matice:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} * \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Model barev YCbCr, který má jasovou složku oddělenou od barevných, je pro ztrátovou kompresi podstatně výhodnější. Lidské oko je podstatně méně citlivější na změny barev než na změny jasu. Proto u barev lze při kompresi připustit větší ztrátu informace než u jasu a dosáhnout tím celkově vyššího kompresního poměru.

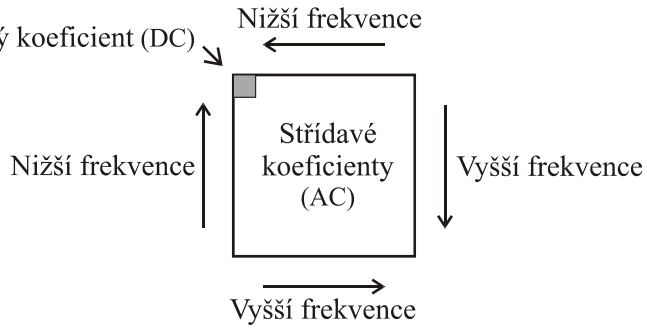
Při kompresi JPEG je rastrový obraz rozdělen na čtvercové bloky po 8×8 bodech. V každém bloku je nejdříve od hodnot v něm odečtena hodnota 128, čímž se hodnoty z intervalu 0..255 převedou do intervalu -128..127, čímž se sníží jejich absolutní velikost, a následně je na každý blok samostatně provedena diskrétní kosinová transformace (DCT - Discrete Cosine Transform) dle vztahu

$$F(u,v) = \frac{1}{4} C(u) C(v) \sum_{i=0}^{7} \sum_{j=0}^{7} f(i,j) \cos\left(\frac{(2i+1)u\pi}{16}\right) \cos\left(\frac{(2j+1)v\pi}{16}\right),$$

kde $C(0) = \frac{1}{\sqrt{2}}$ a $C(u) = C(v) = 1$ pro $u, v \neq 0$.

f(i,j) v uvedeném vztahu označují hodnoty jasové nebo barevných složek jednotlivých bodů bloku 8×8 a *F(u,v)* jsou vypočítané frekvenční koeficienty. Cílem DCT je oddělení informací s nízkými frekvencemi (pomalých změn jasu nebo barev) od vysokých frekvencí (náhlých změn jasu a barev). Koeficient *F(0,0)* reprezentuje stejnosměrnou složku (hodnoty kosinů jsou konstantní), ostatní koeficienty vyjadřují střídavé složky. Směrem k pravému dolnímu rohu (tj. směrem ke koeficientu

F(7,7)) se frekvence koeficientů zvyšují. V každém bloku se DCT provádí třikrát, jednou pro jasovou složku a dvakrát pro jednotlivé barevné složky.



Obrázek 29 Poloha stejnosměrného a střídavých koeficientů v bloku 8×8

Uveďme ukázkový blok 8×8 jasových hodnot *Y*, dále tento blok po odečtení hodnot 128 (*Y'*) a jemu odpovídající hodnoty koeficientů *F* (zaokrouhlené na nejbližší celá čísla):

$$Y = \begin{matrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{matrix} \qquad Y' = \begin{matrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{matrix}$$

$$F = \begin{matrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{matrix}$$

Z příkladu je zřejmé, že největší podíl informace nese stejnosměrný koeficient a střídavé koeficienty s nízkými frekvencemi.

Až doposud byl výpočet bezeztrátový. Nyní následuje kvantizace, při které se z DCT-koeficientů *F(u,v)* počítají koeficienty *C(u,v)*. Kvantizací, jež probíhá dělením koeficientů *F(u,v)* hodnotami kvantizační matice *Q* dochází ke zmenšení hodnot koeficientů *F(u,v)* a zejména malé hodnoty jsou kvantizací zanedbány, tj. vycházejí nulové. Výpočet koeficientů *C(u,v)* je dle vztahu

$$C(u,v) = zaokrouhleno\_na\_celá\_čísla\left(\frac{F(u,v)}{Q(u,v,)}\right),$$

kde *Q(u,v)* jsou hodnoty kvantizační matice 8×8. Uveďme nyní běžně používanou matici *Q* pro jasovou složku a koeficienty *C* spočítané z koeficientů *F* ukázkového bloku uvedeného v předchozím odstavci:

$$Q = \begin{matrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{matrix} \qquad C = \begin{matrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

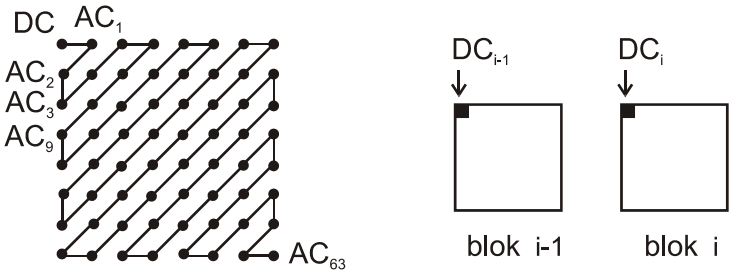
Ukázkový blok neobsahuje výraznější změny jasu. Proto vyšší střídavé složky jsou malé a po kvantizaci dostaneme vesměs nulové hodnoty. V uvedeném příkladu vypočítaná matice *C* obsahuje jen 20 nenulových hodnot. Největší velikost -26 má stejnosměrný koeficient *C(0,0)*. Hodnoty nenulových střídavých koeficientů jsou malé.

Jak už bylo poznamenáno, uvedená matice *Q* se používá pro jasovou složku *Y*. Pro zbývající dvě barevné složky *Cb* a *Cr* se používají matice poskytující ještě hrubější kvantizaci.

Povšimněme si, že pro nižší frekvence jsou hodnoty v matici *Q* menší (hodnoty v levém horním rohu matice) a ztráty při dělení jasové a barevných složek těmito hodnotami (při kvantizaci) jsou menší. Naopak pro vyšší frekvence jsou hodnoty větší a kvantizace probíhá s většími ztrátami. To je cílem celého výpočtu. Aby ztráty byly především v oblasti vysokých frekvencí, neboť ty jsou na obrazu méně patrné.

Poslední fází je statistické kódování hodnot *C* a jejich uložení na výstup. Používá se k tomu QM-coder, což je již uvedený adaptivní způsob aritmetického kódování, nebo Huffmanovo kódování. Zatímco ostatní části standardu JPEG jsou volně dostupné, QM-coder je patentovaný, což je značná překážka při jeho použití v JPEG. Proto se v praxi převážně používá Huffmanovo kódování, i když s ním je účinnost komprese o 10 až 15 procent nižší.

Koeficienty každého bloku 8×8 jsou kódovány v "cik-cak" pořadí směrem od stejnosměrného koeficienty v levém horním rohu ke koeficientu v pravém dolním rohu podle následujícího obrázku.



Obrázek 30 Pořadí kódování koeficientů

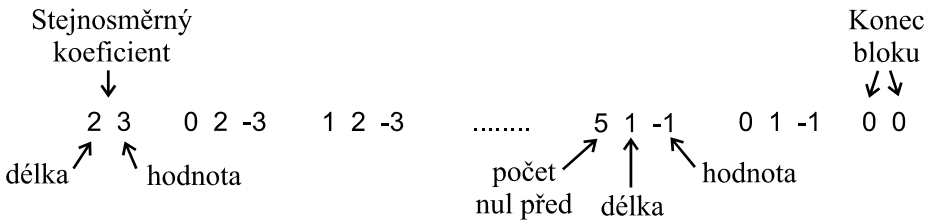
Toto uspořádání umožňuje účinnější kódování, neboť mezi koeficienty *C* se běžně vyskytuje značný počet nul. První koeficient, na obrázku označený DC (tj. stejnosměrný koeficient *C(0,0)*), je v bloku dominantní a jeho hodnota bývá poměrně vysoká. Z hlediska účinnosti komprese je vždy výhodnější, jestliže se kódují menší hodnoty. Proto se nekóduje přímo tato hodnota, ale kóduje se rozdíl mezí ní a hodnotou stejnosměrného koeficientu předchozího bloku (přirozeně vyjma prvního bloku v obraze). Tj. kóduje se rozdíl $DC_i – DC_{i-1}$, kde index *i* označuje právě kódovaný blok a *i-1* předchozí blok. Za koeficientem DC se kódují nenulové střídavé koeficienty. Kódování je ve tvaru dvou hodnot. První hodnota uvádí počet nul od předchozího nenulového koeficientu (metoda RLE) a druhá hodnota reprezentuje nenulový koeficient. Vlastní hodnota nenulového střídavého koeficientu je dále kódována do tvaru dvojice čísel. První číslo udává délku čísla a druhé je vlastní hodnota koeficientu.

Kódování každého nenulového koeficientu je takto vlastně trojice čísel - počet nul, kód délky hodnoty koeficientu a samotná hodnota koeficientu. Pro zvýšení účinnosti se tato trojice neukládá na výstup přímo, ale používá se při tom statické Hufmanovo kódování.

Na následujícím obrázku je kódování našeho ukázkového bloku, který obsahuje celkem 19 nenulových střídavých koeficientů. Přičemž kódované hodnoty až po poslední nenulovou hodnotu jsou:

-26, -3, 0, -3, -2, -6, 2, -4, 1, -4, 1, 1, 5, 1, 2, -1, 1, -1, 2, 0, 0, 0, 0, 0, -1, -1, EOB ,

kde na konci je příznak konce bloku *EOB*, který se kóduje dvěma nulami za sebou.



První je v bloku kódován stejnosměrný koeficient. Předpokládejme, že stejnosměrný koeficient předchozího bloku byl -23. Bude se tedy kódovat hodnota rozdílu -23-(-26). Hodnota stejnosměrného koeficientu se kóduje úplně stejně jako hodnoty střídavých koeficientů dvojicí čísel *délka+hodnota*. Délka je kódována pomocí následující tabulky a pro hodnotu 3 je délka 2. Za stejnosměrným koeficientem následuje kódování 19 nenulových střídavých koeficientů. Například předposlední nenulový střídavý koeficient je kódován trojicí 5,1,-1, kde první číslice 5 označuje, že před ním je 5 nulových střídavých koeficientů, kód délky hodnoty je dle tabulky 1 a poslední v trojici je samotná hodnota -1. Na konci je celého kódování je dvojice nul vyznačující konec bloku.

Vyjádření celočíselné hodnoty pomocí dvojice *délka* + *hodnota* se říká kódování čísla s proměnnou délkou. Provádí se dle tabulky, z níž uveďme aspoň počáteční část:

Délka	Rozsahy hodnot
1	-1 1
2	-3..-2 2..3
3	-7..-4 4..7
4	-15..-8 8..15
atd.	

První část *délka* udává, na kolika bitech je zakódována vlastní hodnota čísla. Například hodnoty v rozsahu

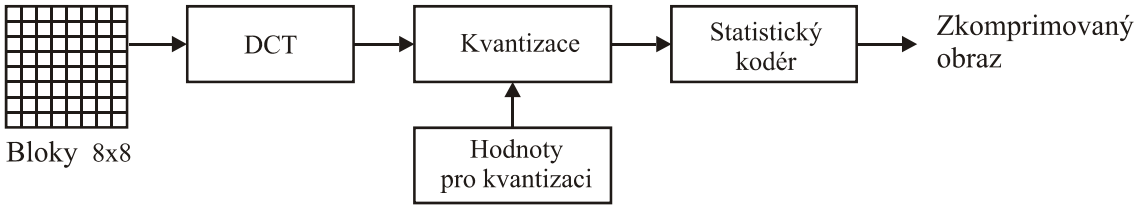
-7..-4 4..7

budou zakódovány na 3 bitech.

Kódování je přitom velmi jednoduché. Záporné hodnoty se přičtením vhodného čísla převedou na nezáporné. Například hodnoty -7..-4 v uvedeném příkladu se přičtením čísla 7 převedou na hodnoty 0..3. Tím se celkový rozsah hodnot -7..-4,4..7 převede na interval 0..7. Jeho hodnoty jsou již přímo tříbitová čísla, jak ukazuje následující tabulka:

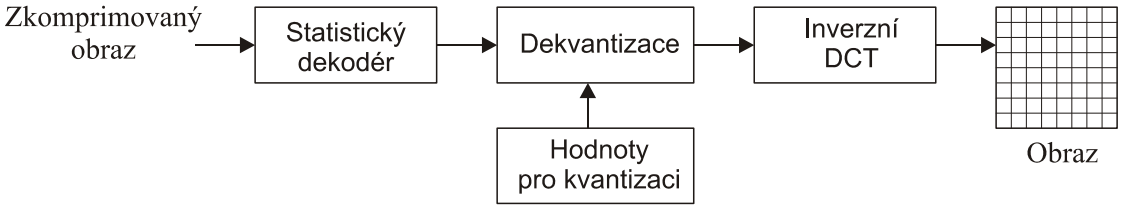
Hodnota	-7	-6	-5	-4	4	5	6	7
Kódování	0	1	2	3	4	5	6	7
Binární zápis	000	001	010	011	100	101	110	111

Celkově způsob komprese JPEG v základních rysech zobrazuje následující schéma:



Obrázek 31 Schéma komprese JPEG

Dekomprese je opačný proces:



Obrázek 32 Schéma dekomprese JPEG

Statistický dekodér (aritmetický nebo Huffmanův) dekóduje koeficienty *C*. Z nich se pomocí kvantizační matice *Q* a vztahu

$$F(u,v) = Q(u,v) * C(u,v)$$

vypočítají koeficienty *F*.

Z koeficientů *F* se inverzní kosinovou transformací vypočítají hodnoty *f* jasové a barevných složek. Vzorce pro inverzní DCT jsou prakticky identické s již uvedeným vzorci pro přímou DCT. Stačí v nich vzájemně zaměnit *F(u,v)* s *f(i,j)* a sumy sčítat přes *u* a *v* místo přes *i* a *j*:

$$f(i,j) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right),$$

Pro ilustraci uveďme, jak bude po dekvantizaci vypadat ukázkový blok, který jsme používali v předchozích odstavcích.

$$F_{\text{po dekvantizaci}} = \begin{matrix} -416 & -33 & -60 & 32 & 48 & -40 & 0 & 0 \\ 0 & -24 & -56 & 19 & 26 & 0 & 0 & 0 \\ -42 & 13 & 80 & -24 & -40 & 0 & 0 & 0 \\ -56 & 17 & 44 & -29 & 0 & 0 & 0 & 0 \\ 18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

A po následné inverzní kosinové transformaci a zpětnému přičtení hodnoty 128 dostaneme matici hodnot jasu:

$$Y_{\text{po dekompresi}} = \begin{matrix} 60 & 63 & 55 & 58 & 70 & 61 & 58 & 80 \\ 58 & 56 & 56 & 83 & 108 & 88 & 63 & 71 \\ 60 & 52 & 62 & 113 & 150 & 116 & 70 & 67 \\ 66 & 56 & 68 & 122 & 156 & 116 & 69 & 72 \\ 69 & 62 & 65 & 100 & 120 & 86 & 59 & 76 \\ 68 & 68 & 61 & 68 & 78 & 60 & 53 & 78 \\ 74 & 82 & 67 & 54 & 63 & 64 & 65 & 83 \\ 83 & 96 & 77 & 56 & 70 & 83 & 83 & 89 \end{matrix}$$

Srovnáním s původními hodnotami bloku zjistíme, že jednotlivé hodnoty po dekompresi zůstaly stejné nebo se změnily poměrně málo, jak ukazuje následující matice rozdílů hodnot.

$$Y - Y_{\text{po dekompresi}} = \begin{matrix} -8 & -8 & 6 & 8 & 0 & 0 & 6 & -7 \\ 5 & 3 & -1 & 7 & 1 & -3 & 6 & 1 \\ 2 & 7 & 6 & 0 & -6 & -12 & -4 & 6 \\ -3 & 2 & 3 & 0 & -2 & -10 & 1 & -3 \\ -2 & -1 & 3 & 4 & 6 & 2 & 9 & -6 \\ 11 & -3 & -1 & 2 & -1 & 8 & 5 & -3 \\ 11 & -11 & -3 & 5 & -8 & -3 & 0 & 0 \\ 4 & -17 & -8 & 12 & -5 & -7 & -5 & 5 \end{matrix}$$

Úroveň kvantizace a tím i kvality obrazu a velikost kompresního poměru může uživatel měnit pomocí Q-faktoru, který lze volit v rozmezí 1−100. Při kvantizaci jsou koeficienty matice Q děleny hodnotou $\frac{\text{Q–faktor}}{50}$. Odtud plyne, že čím bude Q-faktor menší, tím budou kvantizační koeficienty větší. Tím zároveň budou menší vypočítané koeficienty C, což v důsledku znamená větší kompresní poměr, ale i vyšší ztráty a tím i zhoršení kvality obrazu.

Implicitně bývá nastavena hodnota Q-faktoru = 75. Při ní se dosahuje typický kompresní poměru 1:10 a zároveň zůstává zachována kvalita obrazu tak, že na něm nejsou patrné žádné změny.

V době, kdy byl standard JPEG zaveden do používání (okolo roku 1990), byla to nejlepší známá ztrátová metoda komprese obrazu. Mezitím došlo k rozvoji další dvou metod ztrátové komprese:

- komprese založená na vlnkách (wavelets)
- fraktálová komprese

Zejména metoda založená na vlnkách (wavelets) v současné poskytuje již dobré výsledky. Při vyšších poměrech komprese je zhoršení kvality obrazu u této metody méně výrazné než u metody JPEG.

Pro úplnost poznamenejme, že JPEG není označení jen pro standard ztrátové komprese. Součástí standardu je i návrh bezeztrátové komprese obrazu. Její význam je ale podružný, neboť se v praxi příliš nepoužívá.

Na závěr malá ukázka komprese JPEG. Původní obrázek 256×256 ve formátu GIF (komprese LZW) byl ztrátovou kompresí převeden do formátu JPEG s hodnotami Q-faktoru 75 a 20.

	Formát, komprese	Velikost v KB
Původní obrázek	GIF, LZW	76,5
Levý obrázek	JPEG, Q-faktor = 75	12,6
Pravý obrázek	JPEG, Q-faktor = 20	4,6

Obrázek 33 Komprese s Q-faktorem = 75

Obrázek 34 Komprese s Q-faktorem = 20

Původní obrázek ve formátu GIF není uveden, protože nejsou patrné rozdíly mezi ním a obrázkem, který je zkomprimovaný JPEG metodou s Q-faktorem = 75.

4.4 Komprese pohyblivého obrazu - MPEG

Výhody použití digitálního obrazu ve srovnání s jeho analogovou formou jsou značné. Obraz není zkreslen při přenosu, jeho kvalita se nezhoršuje při běžném opotřebení nosiče, na kterém je uložen. Hlavní překážkou v tomto ohledu je skutečnost, že digitální obraz reprezentuje velký objem dat. Vezměme typický televizní signál. Běžná evropská norma pracuje s rozlišením 625 řádků na snímek. Kdybychom jako ekvivalent pro digitální přenos zvolili rastrový obraz s 600 řádky (tj. formát 800×600) v True Color, dostaneme 800×600×3 bytů na jeden snímek. Televizní obraz používá prokládané řádkování, při němž se v každém snímku střídavě přenáší jen liché nebo sudé řádky. Při běžně používané frekvenci 50 půlsnímků za vteřinu by typický film o délce 100 minut vyžadoval přenos nebo uložení objemu dat:

$$800 \times 600 \times 3 \times 25 \times 60 \times 100 \approx 200 \text{ GB}$$

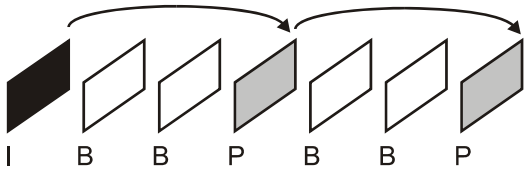
Dnes není k dispozici běžný nosič (CD-ROM), na který by bylo možné uložit 200 gigabajtů, a ani nelze takové množství dat přenášet televizním kanálem. Při použití JPEG komprese se sice objem údajů sníží řádově na desetinu, ale i to je stále velmi mnoho pro uložení i pro přenos.

Proto vznikly specifické metody komprese pohyblivého obrazu, které využívají skutečnost, že snímky jdou v poměrně rychlém sledu za sebou a mezi dvěma následujícími snímky se scéna většinou příliš nezmění. Při kompresi pohyblivého obrazu se samostatně kódují jen určité snímky a ostatní snímky se kódují pomocí jiného snímku nebo snímků tak, že se kódují rozdíly mezi snímky.

Zkomprimovaný pohyblivý obraz je složen ze tří typů snímků, které jsou označíme písmeny *I*, *P* a *B*. Snímky *B* jsou v obrazu vždy po dvou za sebou a mezi nimi je vždy jeden snímek *I* nebo *P*.

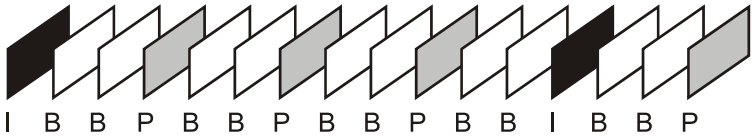
Popišme jednotlivé typy snímků:

- I (Intra-frame) – je samostatný snímek zkomprimovaný metodou JPEG.
- P (Forward-predicted frame) – je snímek kódovaný pomocí předchozího snímku typu *P* nebo *I*.



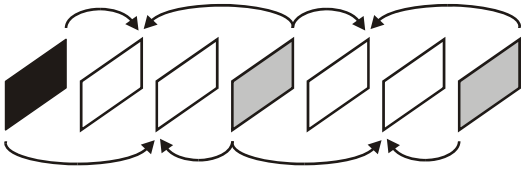
Obrázek 35 Vytvoření snímku typu P

Při kódování snímků *P* dochází ke ztrátě informace, čímž vznikají chyby, které se postupně kumulují. Proto vždy po třech snímcích *P* následuje znovu snímek typu *I*:



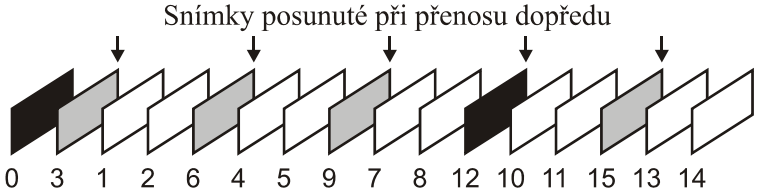
Obrázek 36 Každý 12. snímek je typu I

- B (Bidirectional-predicted frame) – je snímek kódovaný pomocí předchozího a následujícího snímku typu *P* nebo *I*.



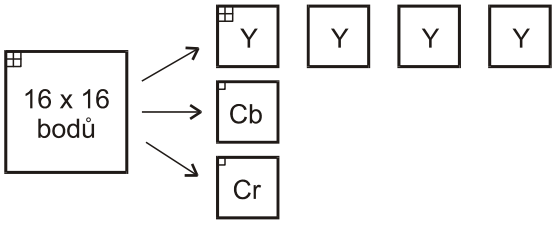
Obrázek 37 Vytvoření snímku B

Při přenosu obrazu se oba snímky, ze kterých vzniká snímek *B*, z technických důvodů přenáší před ním. Tímto se snímky ve skutečnosti přenáší v poněkud jiném pořadí, než v obraze následují za sebou:



Obrázek 38 Pořadí snímků při přenosu

Při kompresi snímků typu *I* metoda JPEG využívá skutečnosti, že oko je méně citlivé na změny barev. Barevné složky *Cb* a *Cr* vždy čtyř sousedních bodů v obrazu jsou vzorkovány a je z nich vytvořena jedna hodnota *Cb* a jedna hodnota *Cr*. Při kompresi tak na 4 bloky 8×8 jasové složky *Y* připadá po jednom bloku 8×8 barevných složek *Cb* a *Cr*. Tím se redukuje počet komprimovaných hodnot a zvyšuje kompresní poměr.

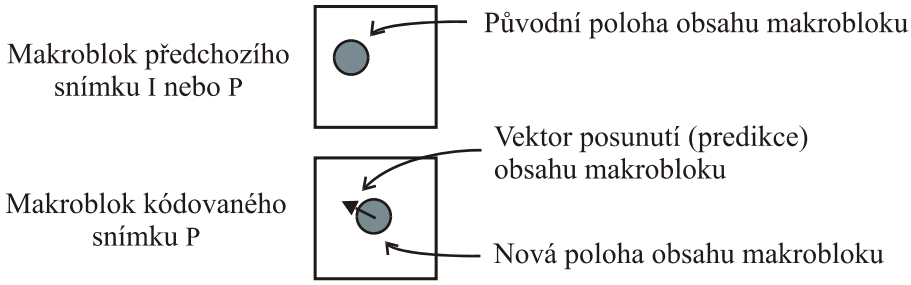


Obrázek 39 Vzorkování jasové a barevných složek

Snímky *P* jsou kódovány dopřednou predikcí. Tato predikce se postupně provádí pro jednotlivé makrobloky obrazu. Makrobloky mají velikost 16×16 bodů a reprezentují 16×16 hodnot jasové složky

a po 8×8 hodnotách barevných složek. Nalezení predikce se provádí společně pro jasovou i barevné složky a má jednu až tři fáze:

1. Nejprve se hledá posunutí makrobloku kódovaného snímku vzhledem ke stejnému makrobloku předchozího (referenčního) snímku *I* nebo *P*, při kterém se obsah obou makrobloků buďto ztotožní nebo je mezi makrobloky nejmenší rozdíl. Jestliže se najde posunutí, při kterém se obsah makrobloků ztotožní nebo rozdíly mezi nimi nepřesahují určitou velmi malou prahovou hodnotu, kóduje se vektor nalezeného posunutí.



Obrázek 40 Vektor predikce makrobloku

2. Jestliže v předchozím kroku nedošlo k zakódování predikce, postup v dalším kroku navazuje na jeho výsledky. Vychází z nalezeného vektoru posunutí, při kterém je rozdíl mezi makrobloky nejmenší. Jestliže tento rozdíl nepřesahuje určitou velikost, kóduje se rozdílový makroblok, jehož hodnoty jsou tvořeny rozdílem hodnot obou makrobloků, a příslušný vektor posunutí. Rozdílový makroblok se přitom kóduje stejným způsobem jako bloky u snímku *I*, tj. používá DCT, kvantizaci a na závěr statistické kódování.
3. Pokud rozdíly mezi makrobloky jsou velké a v druhém kroku proto nedošlo k zakódování, od predikce se upouští a obsah makrobloku se kóduje stejným způsobem jako u snímku typu *I*.

Kódování snímku *B* probíhá prakticky stejným způsobem jako kódování snímku *P* s tím rozdílem, že predikce je v tomto případě obousměrná, tedy jde o interpolaci. Makrobloky kódovaného snímku se srovnávají s hodnotami, které jsou aritmetickým průměrem hodnot stejných makrobloků obou referenčních snímků *P* a *I*, ze kterých je snímek *B* odvozen.

Zavedení snímků *P* a *B* značně zvýšilo účinnost komprese pohyblivého obrazu, neboť kompresní poměr je u snímku *P* přibližně 2,5× vyšší než u snímku *I* a u snímku *B* je tento poměr dokonce přibližně 14× vyšší než u *I*.

Popsaný způsob komprese je obsažen ve standardech MPEG. Tento název je odvozen od označení pracovní skupiny Moving Pictures Experts Group, která se přípravou tohoto standardu zabývá.

V současné době existují dva standardy:

- Standard MPEG-1 používá nižší rozlišení obrazu 352×288, 30 snímků/sec. Je určen především pro uložení obrazu na CD-ROM.
- Standard MPEG-2 má již rozlišení obrazu 720×576, 30 snímků/sec, což odpovídá úrovni standardní televize. Tento standard počítá i s přenosem televizního signálu s vysokým rozlišením 1440×1152, 60 snímků/sec pro současný televizní formát 4:3 a 1920×1152, 60 snímků/sec pro široký formát 16:9.

Komprese MPEG je výpočetně značně náročná záležitost a řeší se technickými prostředky. Dekomprese je sice již o něco jednodušší, přesto i tu na běžných počítačích lze programově realizovat jen pro obrazy se značně nízkým rozlišením, má-li probíhat v reálném čase. Přehrávání obrazu s většími rozlišeními, jako jsou standardy MPEG-1 a MPEG-2, je nutné zajistit technickými prostředky. V současnosti jsou již běžně dostupné integrované obvody pro kompresi a dekompresi MPEG.

4.5 Ztrátová komprese zvuku

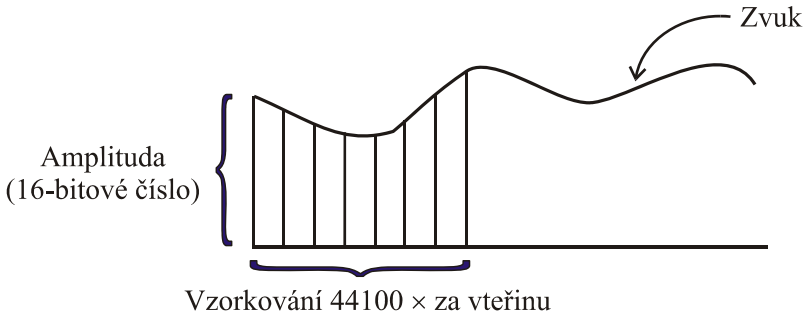
I když kapacita současných kompaktních disků je dostatečně velká pro uložení zvuku v digitalizované podobě, začíná se v poslední době i v této oblasti stále více prosazovat používání ztrátové komprese, neboť její použití poskytuje možnost na disk uložit několikanásobně více hudby, než tomu bylo doposud.

Ztrátová komprese zvuku je vedle komprese obrazu součástí standardů MPEG-1 a MPEG-2, které byly krátce popsány na konci minulé části. V nich jsou specifikovány tři postupy kódování zvuku, které se vzájemně liší složitostí a dosahovaným kompresním poměrem. Jsou označovány jako úroveň 1 až 3, přičemž úroveň 1 je nejjednodušší a má nejnižší kompresní poměr, zatímco úroveň 3 je nejsložitější a dosahuje nejlepší kompresní poměr, jak ukazuje následující tabulka:

Úroveň	Kompresní poměr
1	1:4
2	1:6 až 1:8
3	1:10 až 1:12

Nejrozšířenější je úroveň 3, která je známá pod názvem MP3.

Uvažujme, jaký objem dat představuje digitalizovaný zvuk v nekomprimované podobě. Zvuk získáváme z běžných zdrojů v analogové podobě. Nejprve je nutné ho převést do digitální podoby. To se provádí vzorkováním. Při vzorkování se v pravidelných intervalech měří amplituda (velikost) zvuku. Aby digitalizací nedošlo ke zkreslení zvuku, musí měření amplitudy probíhat s frekvencí aspoň dvakrát vyšší, než je frekvenční rozsah měřeného signálu (Nyquistovo kritérium). Jestliže uvažujeme slyšitelnou oblast zvuku jako 20 Hz až 20 kHz, musí vzorkovací kmitočet být nejméně 40 kHz. U standardních zvukových záznamů na kompaktních discích se používá frekvence vzorkování 44,1 kHz. Dále se musí zvolit, jak přesně je při vzorkování měřena amplituda zvuku. Zatímco při digitalizaci obrazu stačilo 256 možných hodnot (8-bitové číslo), zvuk je nutné měřit přesněji. Měří se s přesností 16-bitového čísla, tj. 65536 možných velikostí amplitudy.



Obrázek 41 Digitalizace zvuku

Odtud můžeme vypočítat, kolik bytů má 1 vteřina digitalizovaného zvuku:

$$44100 \times 2 = 88200 \text{ bytů}$$

Hodinový stereofonní záznam (2 kanály) bude mít velikost:

$$88200 \times 3600 \times 2 \approx 635 \text{ MB}$$

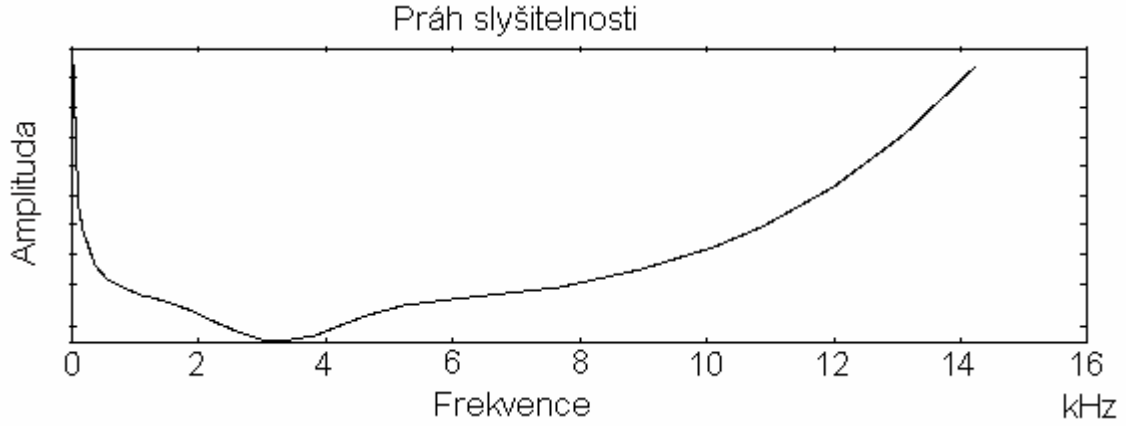
To je přibližně kapacita běžného kompaktního disku. Při kompresi 1:10 bychom tak na disku mohli mít místo jedné hodiny 10 hodin hudby.

Ztrátová komprese zvuku vychází z řady psychoakustických vlastností lidského ucha. Ty využívá k zanedbání těch informací ve zvuku, na jejichž změnu ucho není citlivé nebo které vůbec nepostřehne. Komprese se tímto snaží výrazně snížit objem dat bez znatelného vlivu na kvalitu zvuku.

V následujících odstavcích jsou základní principy ztrátové komprese:

Využití prahové úrovně slyšitelnosti

Lidské ucho zvuk slyší až od určité úrovně (hlasitosti), přičemž tato úroveň je pro různé tóny různá. Jak ukazuje následující graf, je ucho nejcitlivější na tóny v oblasti 2-4 kHz.



Obrázek 42 Závislost prahu slyšitelnosti na frekvenci tónu

Uplatnit individuálně práh slyšitelnost a další psychoakustické vlastnosti ucha pro každý tón je technicky neschůdné. Proto se to řeší rozdělením kmitočtového spektra na dostatečný počet dílčích pásem. Standard MPEG používá dělení na 32 pásem. Úrovně 1 a 2 mají všechna dílčí pásma stejně široká (750 Hz), což je technicky jednodušší, ale méně účinné. Těchto 32 pásem pokrývá rozsah

$$750 \times 32 = 24\,000 \text{ Hz}$$

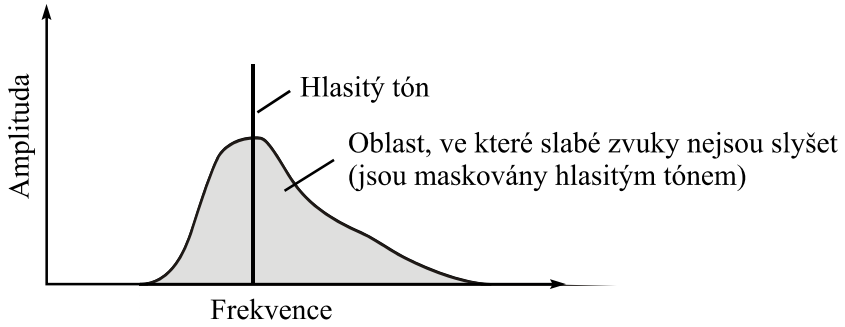
Tomu odpovídá vzorkovací kmitočet 48 kHz používaný standardem MPEG. To je více, než vyžaduje oblast slyšitelnosti, jejíž hranice je u člověka typicky 20 kHz. Rezerva ve vzorkovacím kmitočtu je zvolena s ohledem na určitá technická omezení systémů, které způsobují, že ve skutečnosti kmitočtové spektrum je poněkud menší než polovina vzorkovací frekvence. Poznamenejme, že standard MPEG vedle 48 kHz má i nižší vzorkovací frekvence 44,1 kHz a 32 kHz.

Úroveň 3 používá již dělení podle tzv. kritických šířek pásem, které umožňují účinněji uplatnit psychoakustické vlastnosti ucha. Kritické šířky pásem jsou pro nízké kmitočty malé (méně než 100 Hz) a směrem k vysokým kmitočtům se postupně zvětšují (až nad 4 kHz). Šířky pásem jsou voleny tak, aby psychoakustické vlastnosti všech tónů v každém z pásem byly přibližně stejné. Počet pásem použitých ve standardu MPEG byl zvolen tak, aby byl optimální jak z hlediska psychoakustických vlastností ucha tak z hlediska komprese.

Obrázek 43 Rozdělení akustického rozsahu na 32 pásem pro jednotlivé filtry

Maskování slabých zvuků silným zvukem (frekvenční maskování)

Je obecně známo, že slabé zvuky vnímáme jen, je-li poměrně ticho. Jakmile se objeví silnější zvuk, slabší zvuky přestávají být slyšet. Říkáme, že silný zvuk maskuje slabší zvuky. Silný zvuk maskuje zejména ty zvuky, které jsou mu kmitočtově nejblíže. Křivka závislosti maskování zvuku na frekvenci v okolí hlasitého tónuje na následujícím obrázku.

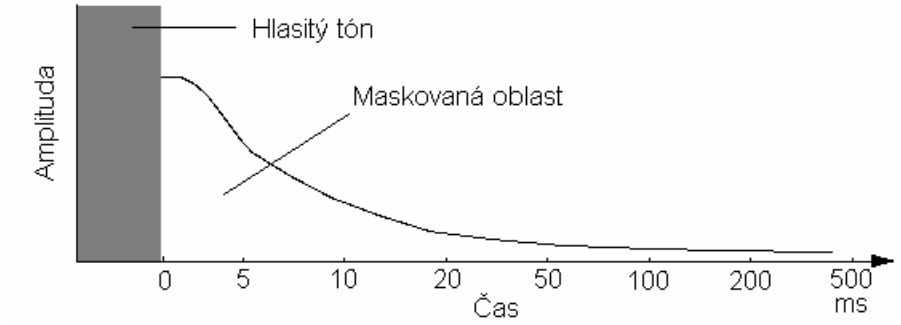


Obrázek 44 Maskování slabých zvuků v okolí hlasitého tónu

Je zřejmé, že křivka je strmější směrem k nižším kmitočtům. Proto také kritické šířky pásem při dělení popsaném v předchozím odstavci jsou voleny tak, že u nižších kmitočtů jsou pásma užší a u vyšších kmitočtů je naopak šířka pásem větší.
Standard MPEG velmi výrazně využívá maskovacího efektu a kóduje jen ty zvuky, které jsou slyšet.

Maskování po silném zvuku (časové maskování)

Když silný zvuk pomine, chvíli trvá, než lidské ucho začne vnímat slabší zvuky. Tuto setrvačnost ukazuje následující obrázek.



Obrázek 45 Časová závislost maskování slabých zvuků po ukončení hlasitého tónu

Efekt maskování se vyskytuje nejen po silném zvuku ale i před ním. Ukazuje se, že mozek potřebuje na zpracováním sluchového vjemu určitý čas. Maskovací efekt před silným zvukem je 2 až 5 ms, po silném zvuku až 100 ms.

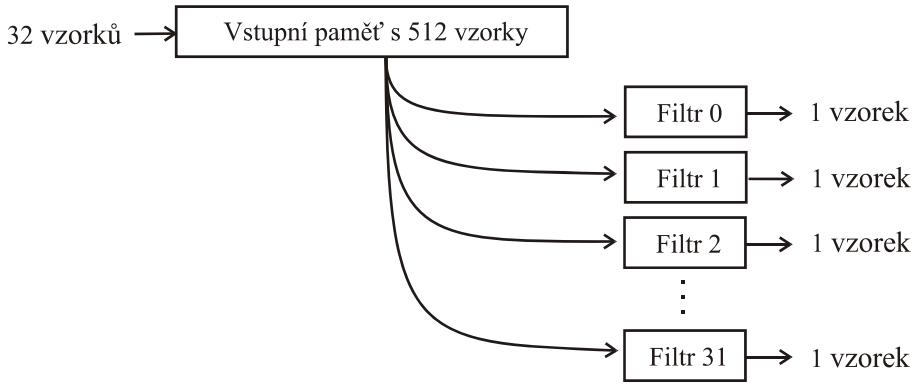
Využití vlastností stereofonního signálu

Psychoakustických vlastností vjemu stereofonního signálu a korelací mezi oběma kanály se využívá dvěma způsoby:
Některé veličiny se pro oba kanály kódují společně, což je úspornější, než kdyby se oba kanály kódovaly odděleně. To je využito ve všech třech úrovních standardu MPEG.
využívá se určitých podobností mezi zvuky v obou kanálech nebo naopak skutečnosti, že některé rozdíly mezi nimi nejsou pro stereofonní vjem podstatné. Tyto skutečnosti využívá až úroveň 3 kódování MPEG.

Kódování a dekódování

Kódování má tři fáze:

Nejprve se provede tzv. časově-frekvenční převod. Při něm se pro každých 32 vstupních vzorků vypočítá po jednom vzorku pro každé z 32 dílčích frekvenčních pásem. Vzorky se ve filtrech počítají z posledních 512 vstupních vzorků, jež jsou pro tento účel uchovávány ve vstupní paměti. To znamená, že před každým výpočtem se do vstupní paměti načte nových 32 vstupních vzorků, které v ní nahradí 32 nejstarších vzorků. Pro výpočet vzorků pro jednotlivá frekvenční pásma se ve filtrech používá diskrétní kosinová transformace nebo Fourierova transformace.



Obrázek 46 Při každých 32 vstupních vzorcích každý filtr vypočítá jeden vzorek

Následuje kvantizace. Je to ztrátová fáze komprese. Používá se při ní řada tabulek, které obsahují psychoakustický model. Kvantizace provádí maskování informací, které ucho nevnímá.
Poslední fází je Huffmanovo kódování.

Dekódování má dvě fáze:

Dekódování Huffmanova kódu.

Frekvenčně-časový převod. Při něm se ze všech 32 dílčích frekvenčních pásem zpětně počítají zvukové vzorky.

Dekódování lze v reálném čase provádět programy na běžných počítačích. Výkon dnešních procesorů je dostatečný pro přehrávání zvuku komprimovaného úrovní 3. Dnes již jsou rovněž k dispozici integrované obvody pro dekompresi úrovně 3.

5. Rejstřík