Algoritmy používané pro kompresi dat

Zdeněk Sawa

22. května 2000

1 Úvod

Často potřebujeme uchovávat značné objemy dat, avšak jsme limitováni kapacitou paměťových médií. Jednou z možností, jak tento problém řešit, je využít toho, že data většinou obsahují určitou redundantní informaci, a uložit tato data úspornějším způsobem. Tento proces se nazývá ***komprese dat*** (data compression).

O datech můžeme předpokládat, že jsou zapsána jako posloupnost symbolů z nějaké abecedy (***abeceda*** je konečná množina ***symbolů*** (***znaků***)). Taková posloupnost symbolů se nazývá ***slovo*** (používají se také termíny ***řetězec*** nebo ***zpráva***). Příkladem abecedy je třeba množina ASCII znaků (každý takový znak je možné uložit jako 8-bitové číslo, tady 1 byte). Slovy nad touto abecedou jsou pak například soubory uložené na disku, kde každý z těchto souborů je tvořen posloupností bytů.

Úkolem je najít algoritmus, který by libovolnému slovu $u = u_1u_2 \dots u_m$ přiřadil jiné slovo $v = v_1v_2 \dots v_n$ tak, aby bylo možné z výsledného slova v rekonstruovat původní slovo u a navíc, aby slovo v obsahovalo méně redundantní informace než slovo u. Proces, kterým k danému slovu u vytvoříme slovo v, označujeme jako ***kódování*** (používají se také pojmy ***komprese*** nebo ***komprimace***). Opačný proces, kdy ke komprimovanému slovu v sestrojíme původní slovo u, se nazývá ***dekódování*** (nebo též ***dekomprese*** či ***dekomprimace***).

Pokud dekompresí komprimovaného slova v dostaneme původní slovo u, hovoříme o ***bezztrátové*** (lossless) kompresi. Pokud po provedení komprese a následné dekompresi dojde k určité ztrátě informace, tj. výsledkem bude nějaké slovo u', které se bude od původního slova u lišit (i když by se mělo lišit jen velmi málo, protože jinak by takový systém neměl praktický význam), hovoříme o kompresi ***ztrátové***. Ta se využívá zejména pro kompresi multimediálních dat (obraz, zvuk, ...), kde určitá ztráta informace nevadí.

Poměr velikosti komprimovaných dat k velikosti původních dat se nazývá ***kompresní poměr***:

$$Kompresní\ poměr = \frac{Délka\ komprimovaných\ dat}{Délka\ původních\ dat}$$

Je samozřejmě žádoucí, aby kompresní poměr byl co nejmenší. Dá se ukázat, že nemůže existovat algoritmus pro bezztrátovou kompresi, pro který by platilo, že pro libovolné slovo je kompresní poměr ≤ 1 a pro některá slova je kompresní poměr < 1, tj. algoritmus by dokázal některá slova zkrátit, aniž by délka jiných slov vzrostla. V praxi je důležité, aby algoritmus dosahoval dobrého kompresního poměru pro typická data, pro která bude používán, a aby u dat, která obsahují jen málo redundantní informace (např. data, která již byla zkompresována), došlo v nejhorším případě jen k malému zvětšení. Toho je možno dosáhnout například tak, že pokud algoritmus zjistí, že velikost dat po kompresi vzrostla, data nebude kompresovat, ale pouze je zkopíruje, přičemž na začátek přidá informaci o tom, že to co následuje jsou nezkompresovaná data. Velikost dat se tak zvětší pouze o tuto informaci.

V následujícím textu je pojednáno pouze o metodách používaných pro bezztrátovou kompresi, konkrétně o metodách používaných pro kompresi souborů, u kterých nemáme žádné bližší informace o konkrétním charakteru dat (tj. může to být např. textový soubor, programový soubor, multimediální data nebo cokoliv jiného).

Existuje celá řada různých programů, které se používají pro kompresi souborů, např. `zip`, `gzip`, `arj`, `lha`, `zoo`, `compress`, `rar` a mnoho dalších. Některé z těchto programů slouží nejen ke kompresi, ale také k archivaci souborů, tj. umožňují uložit více různých komprimovaných souborů (včetně struktury adresářů) do souboru jediného.

Metody používané pro kompresi souborů v těchto programech je možné rozdělit do tří skupin:

1. Metody založené na myšlence, že v souborech se většinou některé znaky vyskytují častěji než jiné, a proto je výhodné zakódovat tyto znaky pomocí kratších posloupností bitů za cenu, že znaky, které se vyskytují méně často musí být zakódovány pomocí delších posloupností bitů. Tyto metody jsou označovány jako ***statistické metody*** komprese dat. Příklady těchto metod jsou Shannon-Fanovo kódování, Huffmanovo kódování (statické či adaptivní) nebo aritmetické kódování.

2. Metody založené na pozorování, že v souborech se některé posloupnosti znaků opakují (např. v textovém souboru se opakují slova). Taková opakující se posloupnost může být uložena v datech pouze jednou a na všech ostatních místech, kde se opakuje, je nahrazena odkazem. Tyto metody se označují jako ***substituční*** (substitutional) nebo také ***slovníkové metody***, protože v průběhu kódování je vytvářen „slovník" všech slov, která se zatím v datech vyskytla. Dosud nezakódovaná data jsou vyhledávána v tomto slovníku a pokud se v něm vyskytují tak jsou nahrazena odpovídajícím odkazem. Většina těchto metod je variací jedné ze dvou metod označovaných podle jmen svých autorů, jimiž jsou Lempel a Ziv, a podle roku vzniku jako LZ77 (metoda posuvného okna) a LZ78 (metoda s rostoucím slovníkem). Tyto dvě metody se od sebe liší zejména způsobem vytváření slovníku.

3. Metody založené na tzv. Burrows-Wheelerově transformaci (Burrows-Wheeler transformation), viz [7]. Při této transformaci nejsou data kompresována, ale jsou převedena do tvaru, který je mimořádně vhodný pro kompresi standardními metodami, např. použitím Huffmanova nebo aritmetického kódování.

V praxi je často použita kombinace více metod. Nejčastěji je nejprve použita některá slovníková metoda, pomocí které jsou z dat odstraněny opakující se sekvence symbolů, a výsledek je pak dále zkompresován pomocí některé statistické metody.

V následujícím textu jsou popsány některé konkrétní metody používané pro kompresi dat. Jsou popsány zejména algoritmy používané pro efektivní implementaci těchto metod. Jejich praktické využití je demonstrováno na formátu DEFLATE, který je požíván pro kompresovaná data např. populárními programy `zip`, `gzip` či knihovnou `zlib`.

Formát DEFLATE byl zvolen proto, že je volně k dispozici jak specifikace tohoto formátu (viz [2]), tak i zdrojové kódy programu `gzip` (viz [1]), který tento formát používá. V porovnání s jinými podobnými programy dosahuje `gzip` velmi dobrého kompresního poměru.

Navíc žádný z algoritmů používaných tímto programem není patentován, takže je možné tyto algoritmy volně používat.

Při kompresi dat do formátu DEFLATE je nejprve použit algoritmus LZ77 a data jsou pak dále zakódována pomocí statického Huffmanova kódování.

V kapitole 2 je popsán algoritmus LZ77 a způsob jeho implementace programem `gzip`. V kapitole 3 je popsáno Huffmanovo kódování, zejména pak algoritmy pro efektivní implementaci tohoto kódování. Nakonec je v kapitole 4 popsáno, jakým způsobem jsou obě tyto metody použity při kompresi dat do formátu DEFLATE.

2 Algoritmus LZ77

2.1 Princip algoritmu

Algoritmus LZ77 slouží k nalezení takových sekvencí symbolů, které se ve slově, které chceme zkompresovat, vyskytují opakovaně. Taková opakující se sekvence symbolů je pak v kompresovaných datech uložena pouze jednou a každý její další výskyt je nahrazen odkazem na tuto sekvenci. Kompresovaná data jsou tvořena posloupností, která obsahuje:

1. symboly z původního slova, nazývané ***literály***,
2. ***odkazy*** na sekvence symbolů, které se již ve slově vyskytly.

Každý odkaz je tvořen dvojicí čísel $(vzdálenost, délka)$. Hodnota *vzdálenost* udává, kde se nachází začátek opakující se sekvence, konkrétně o kolik symbolů před místem, kde se nachází odkaz. Hodnota *délka* udává počet znaků v této sekvenci.

Pro uložení odkazu tvořeného dvojicí čísel $(vzdálenost, délka)$ v kompresovaných datech potřebujeme určitý počet bitů, proto nemá smysl nahrazovat odkazem velmi krátké sekvence, které je možné úsporněji uložit přímo jako posloupnost symbolů. Je proto vhodné zvolit určitou ***minimální délku*** a nahrazovat odkazem pouze sekvence, které dosahují alespoň této minimální délky.

Postup kódování je možné popsat následovně:
Nechť vstupem je například slovo $u = u_1u_2 \dots u_n$. Aktuální pozici ve slově budeme označovat i. Na začátku výpočtu je i rovno jedné. V každém okamžiku rozděluje aktuální pozice i slovo u na dvě části — na již zakódovanou část $u_1u_2 \dots u_{i-1}$ a dosud nezakódovanou část $u_iu_{i+1} \dots u_n$. Na začátku výpočtu je zakódovaná část prázdná a nezakódovanou část tvoří celé slovo u.

V každém kroku je nejprve v zakódované části slova vyhledáno nejdelší podslovo, které se rovná prefixu nezakódované části. Pokud je toto slovo kratší než minimální délka, je ke kompresovaným datům přidán jako literál první symbol z dosud nezakódované části a tento symbol je přesunut z nezakódované části do zakódované (i se zvětší o jedničku). Pokud má nalezené slovo délku, která je větší nebo rovná minimální délce, je ke kompresovaným datům přidán odkaz na toto slovo.

Celý postup je popsán algoritmem 1.

Algoritmus pro dekódování je následující:
Čteme postupně kompresovaná data. Pokud narazíme na literál, prostě ho překopírujeme do dekódovaných dat. Pokud narazíme na odkaz přidáme k dekódovaným datům sekvenci symbolů určenou odkazem (tuto sekvenci najdeme v již dekódovaných datech).

Algoritmus 1 LZ77 — kódování

Vstup: slovo $u = u_1 u_2 \dots u_n$

```
i ← 1
while i ≤ n do
  najdi nejdelší podslovo u_k u_{k+1} ... u_{k+l-1} slova u takové, že k < i a
      u_k u_{k+1} ... u_{k+l-1} = u_i u_{i+1} ... u_{i+l-1}
  if l < MIN_LEN then      — MIN_LEN je min. délka slova nahrazovaného odkazem
    output literál u_i
    i ← i + 1
  else
    output odkaz (i − k, l)
    i ← i + l
  end if
end while
```

Algoritmus 2 LZ77 — dekódování

Vstup: kompresovaná data, tj. posloupnost literálů a odkazů

```
u ← ε
while není konec kompresovaných dat do
  read x
  if x = odkaz(d, l) then      — kopíruj sekvenci symbolů, na kterou ukazuje odkaz
    j ← |u| − d + 1
    for i = 1 to l do
      u ← u · u_j
      j ← j + 1
    end for
  else      — je to literál
    u ← u · x
  end if
end while
```

Výstup: slovo u

Tento postup je popsán algoritmem 2.

Pokud chceme zkompresovat například slovo `ABCBCDABCBA`, bude zakódování tohoto slova vypadat následovně (jako minimální délka nahrazovaného slova byla zvolena hodnota 2):

Zakódovaná část slova	Nezakódovaná část slova	Kompresovaná data
ε	ABCBCDABCBA	ε
A	BCBCDABCBA	A
AB	CBCDABCBA	AB
ABC	BCDABCBA	ABC
ABCBC	DABCBA	ABC(2, 2)
ABCBCD	ABCBA	ABC(2, 2)D
ABCBCDABCB	A	ABC(2, 2)D(6, 4)
ABCBCDABCBA	ε	ABC(2, 2)D(6, 4)A

Původní slovo `ABCBCDABCBA`, tvořené 11 symboly, bylo tedy zakódováno jako posloupnost `ABC`(2, 2)`D`(6, 4)`A`, tvořená 7 prvky.

Při dekódování bude postup opačný:

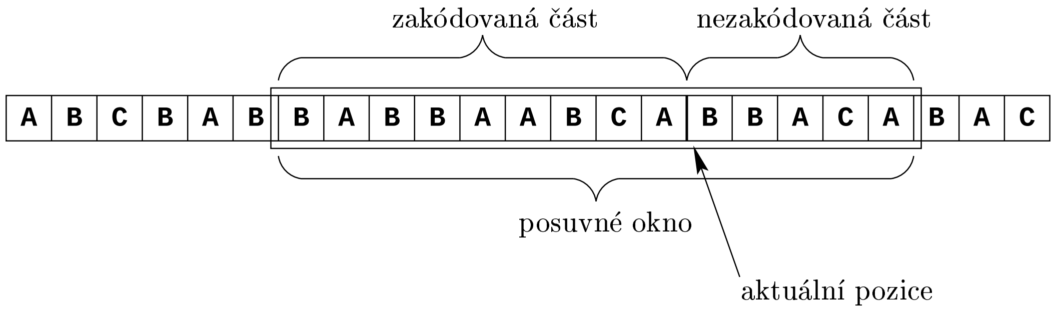
Kompresovaná data	Dekódovaná data
ABC(2, 2)D(6, 4)A	ε
BC(2, 2)D(6, 4)A	A
C(2, 2)D(6, 4)A	AB
(2, 2)D(6, 4)A	ABC
D(6, 4)A	ABCBC
(6, 4)A	ABCBCD
A	ABCBCDABCB
ε	ABCBCDABCBA

Je dobré si všimnout, že podslovo vyhledané v již zakódované části slova může zasahovat i do ještě nezakódované části slova. Například slovo `ABCABCABCABCABCABCA` bude zakódováno jako posloupnost `ABC`(3, 16). Při dekódování je pak možné narazit na odkaz, který ukazuje na podslovo, které v daném okamžiku ještě není celé známo. To ovšem nevadí, protože je znám jeho začátek. Tím, že přidáme k dekódovaným datům tento začátek, zjistíme další symboly tohoto podslova, jejich přidáním zase další symboly a tak dále až postupně dekódujeme celé podslovo.

2.2 Posuvné okno

Zřejmou nevýhodou předchozí metody je, že kódované slovo musí být celé přítomno v paměti, aby bylo možné prohledat celou jeho již zakódovanou část. S postupem kódování je také nutné prohledávat stále větší a větší část slova.

Těmto problémům je možné se vyhnout tím, že použijeme tzv. ***posuvné okno*** (sliding window) určité velikosti. Toto okno vymezuje v každém okamžiku určitou část slova. Poloha okna je vždy určená aktuální pozicí. Okno obsahuje určitý počet znaků před touto pozicí (zakódovaná část) a určitý počet znaků za touto pozicí (nezakódovaná část). Velikost zakódované



Obrázek 1: Posuvné okno

a nezakódované části slova je pevně určena a během výpočtu se nemění. Během výpočtu je nutné udržovat v paměti pouze obsah okna. Viz obrázek 1.

Při výpočtu je vždy vyhledáván co nejdelší prefix nezakódované části okna v zakódované části okna. Pokud je při kódování vytvořen odkaz (*vzdálenost*, *délka*), může *vzdálenost* nabývat maximálně takové hodnoty, jako je délka zakódované části okna, a *délka* může nabývat maximálně takové hodnoty, jako je délka nezakódované části okna. Díky tomu je velikostí okna omezen i počet bitů potřebných k zakódování odkazu.

Přirozený způsob, jak posuvné okno implementovat, je použít tzv. ***kruhový buffer*** (circular buffer), to znamená použít pole stejné velikosti jako je velikost okna, které budeme zaplňovat postupně od začátku tak dlouho, až dojdeme na konec, přičemž pak začneme ukládat data opět od začátku (čímž přepíšeme dříve uložená data). Veškeré výpočty s indexy v tomto poli je třeba provádět mod n, kde n je velikost bufferu.

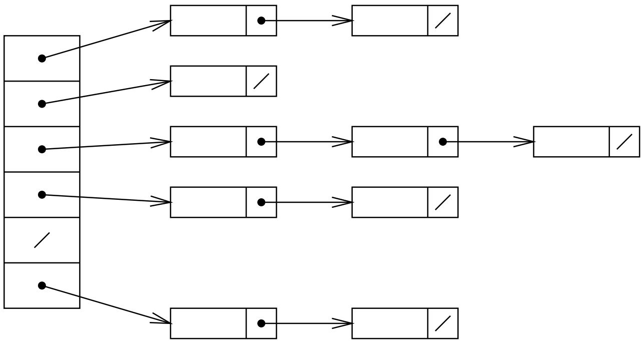
2.3 Vyhledávání slov pomocí hashovací tabulky

Časově nejnáročnější operací při kódování je vyhledávání nejdelšího podslova, které začíná v zakódované části okna a shoduje se s prefixem nezakódované části okna.

Nejjednodušší přístup je použít hrubou sílu a systematicky procházet celou zakódovanou část okna směrem od aktuální pozice k začátku okna. To znamená, že, pokud aktuální pozice je i a délka zakódované části okna je n, slovo začínající na pozici i je nejprve porovnáno se slovem začínajícím na pozici $i-1$, pak se slovem začínajícím na pozici $i-2$, a tak dále, až je nakonec porovnáno se slovem začínajícím na pozici $i-n$, a poté je vybráno z těchto slov takové, které má se slovem začínajícím na pozici i nejdelší společný prefix. Časová složitost jednoho vyhledání je pak $O(mn)$, kde n je délka zakódované části okna a m je maximální délka hledaného slova.

Hledané slovo se bude lišit od většiny slov, se kterými bude porovnáváno, často už v prvním znaku, resp. v několika málo prvních znacích. Přitom má smysl ho porovnávat pouze se slovy, u kterých máme jistotu (nebo alespoň velkou naději), že se s hledaným slovem shodují alespoň v několika prvních znacích.

Metoda při, které není hledané slovo porovnáváno se všemi slovy, ale pouze s těmi, které se s ním shodují alespoň v několika prvních znacích, se dá realizovat tak, že vytvoříme tabulku, jejíž řádky odpovídají všem možným k-ticím symbolů, kde k je nějaká vhodně zvolená nepříliš velká konstanta (typicky 2 nebo 3), k-tice symbolů tedy tvoří indexy řádků této tabulky. Každý řádek tabulky pak bude obsahovat odkazy na všechna slova v okně, která začínají danou k-ticí symbolů.



Obrázek 2: Hashovací tabulka

Při vyhledávání se pak hledané slovo porovnává se všemi slovy uvedenými na řádku, jehož index odpovídá prvním k symbolům hledaného slova.

Řádky tabulky je vhodné reprezentovat jako seznamy, kde každý prvek obsahuje ukazatel na následující prvek seznamu. Protože k-tic symbolů může být velmi mnoho a tabulka by pak byla obrovská, používá se často tzv. ***hashovací funkce***, která mapuje prvky s nějaké velké množiny (např. množiny všech k-tic symbolů) na prvky nějaké menší množiny (např. na celá čísla z nějakého určeného intervalu). Řádky tabulky (tj. seznamy prvků) jejichž indexy jsou namapovány na tutéž hodnotu jsou pak spojeny v jeden seznam. Takové datová struktura se nazývá ***hashovací tabulka*** (hash table). Je schématicky znázorněna na obrázku 2.

Jednotlivé prvky seznamu obsahují odkazy na slova v okně. Tyto odkazy jsou uloženy jako indexy prvního znaku slova, na které ukazují.

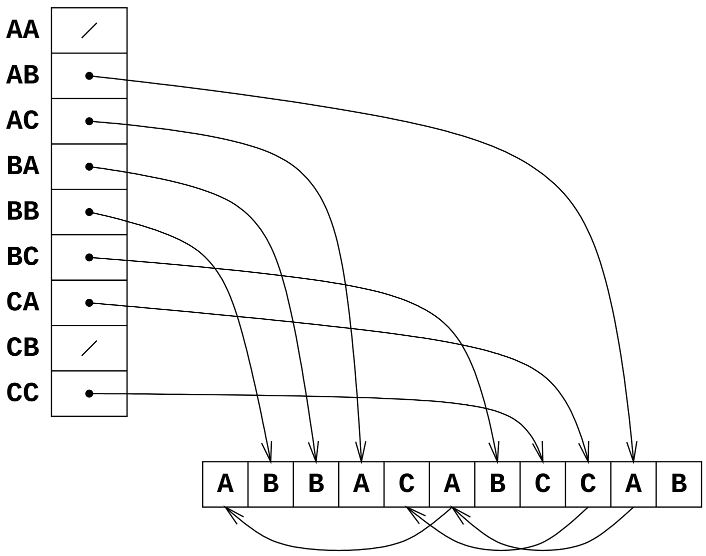
Slova jsou v seznamu uspořádána v opačném pořadí než v okně. Nové slovo je vždy přidáváno na začátek seznamu. (U nově přidaného prvku je odkaz na následující prvek nastaven na dosavadní začátek seznamu a začátek seznamu je nastaven na tento nový prvek). Při prohledávání je tedy hledané slovo porovnáno nejdříve se slovy, která leží nejblíž aktuální pozice.

Prvky všech seznamů v hashovací tabulce jsou všechna slova v okně, přičemž každé slovo leží v právě jednom seznamu. Pro uložení prvků všech seznamů je tedy možné použít jediné pole stejné velikosti jako je velikost bufferu. Prvek tohoto pole s indexem i bude odpovídat slovu v bufferu, které začíná v bufferu na pozici i. Prvek obsahuje vždy index následujícího prvku seznamu, to znamená index začátku předchozího slova v bufferu, které začíná stejnou k-ticí znaků. Takto realizovaná hashovací tabulka je znázorněna na obrázku 3.

Při každém posunu okna je třeba přidat do hashovací tabulky odkaz na všechna slova, která přibudou v zakódované části okna. (Bude jich přesně tolik, o kolik znaků se okno posunulo.) Odkazy není třeba z hashovací tabulky odstraňovat. Pokud algoritmus při procházení seznamem narazí na odkaz, který ukazuje před začátek okna, prostě ho ignoruje.

Při použití hashovací tabulky není sice časová složitost v nejhorším případě menší než při použití hrubé síly, pro běžná data však dochází k výraznému zrychlení vyhledávání.

Použití hashovací tabulky není jediný způsob jak vyhledávat slova v okně, existuje několik dalších metod, které jsou většinou založené na použití tzv. sufixového stromu (suffix tree), viz např. [6]. Tyto metody sice mají asymtoticky lepší časovou složitost v nejhorším případě než algoritmus využívající hashovací tabulku, pro typická data však vyžadují většinou více času a paměti než tato jednoduchá metoda.



Obrázek 3: Hashovací tabulka s odkazy na slova v okně

2.4 Několik poznámek k implementaci

U formátu DEFLATE je vstupem kompresního algoritmu posloupnost bytů. Symboly abecedy jsou jednotlivé byty (tedy 8-bitové hodnoty). Abeceda, se kterou se pracuje, tedy obsahuje $2^8 = 256$ znaků.

Data jsou nejprve kompresována algoritmem LZ77. Minimální délka slov, která jsou nahrazována odkazem, je 3 znaky, maximální délka je 258 znaků. Odkaz může odkazovat na slovo ve vzdálenosti nejvýše 32768 (32K) znaků od aktuální pozice. To znamená, že velikost okna je $32768 + 258 = 33026$ znaků. V programu pro kompresi může být skutečná velikost okna menší, například při nedostatku paměti, ale použití menšího okna vede k horší kompresi. V programu pro dekompresi nesmí být velikost okna menší, aby bylo možné dekompresovat data kompresovaná s maximální velikostí okna.

Kompresovaná data nejsou posílána na výstup, ale jsou ukládána do paměti, aby mohla být později zakódována s využitím Huffmanova kódování.

V programu `gzip`, který kompresuje data do formátu DEFLATE je při implementaci algoritmu LZ77 použita hashovací tabulka. Velikost této tabulky bývá 32768. Pro výpočet indexu do této tabulky se používají první tři znaky každého slova.

V programu `gzip` je používáno okno maximální velikosti, jakou formát DEFLATE připouští. Okno ovšem není implementováno jako kruhový buffer v pravém slova smyslu. Je použit buffer dvakrát větší než je velikost okna. Buffer je zaplňován od poloviny, data jsou přidávána pouze do jeho druhé poloviny, a když je dosaženo konce bufferu, je obsah druhé poloviny překopírován do poloviny první a data jsou znovu přidávána od poloviny bufferu.

V okamžiku, kdy je prováděn přesun druhé poloviny bufferu do první, je vždy nutné přepočítat hodnoty v hashovací tabulce i odkazy v seznamech. V tomto okamžiku jsou také z hashovací tabulky odstraněny všechny odkazy, které by po přesunu ukazovaly před začátek bufferu.

Výhodou tohoto přístupu je, že na rozdíl od kruhového bufferu nikdy nedojde k situaci, kdy by se začátek okna nacházel na konci bufferu a okno by pokračovalo na začátku bufferu, okno tedy vždy tvoří souvislou část bufferu. Nevýhodou je nutnost kopírovaní značného množství dat.

V programu `gzip` je použito několika různých mechanismů, které ovlivňují činnost algoritmu LZ77. Činnost těchto mechanismů závisí na hodnotách určitých konstant, které může uživatel nastavit. V praxi nenastavuje uživatel přímo tyto konstanty, ale zvolí číslo z intervalu 1 (nejrychlejší, ale nejhorší komprese) až 9 (nejlepší komprese, ale nejpomalejší), a podle této volby se nastaví hodnoty konstant.

Seznam v hashovací tabulce může být v nepříznivém případě velmi dlouhý. Proto je počet slov, se kterými je hledané slovo porovnáváno, omezen konstantou c_1, jež závisí na volbě uživatele. Čím je hodnota této konstanty větší, tím je dosažená komprese lepší, ale výpočet trvá déle.

Při hledání výskytu slova je vždy nejprve nastaven čítač na hodnotu c_1. Při každém porovnání hledaného slova se slovem v seznamu je hodnota čítače snížena o jedničku. Když čítač dosáhne hodnoty nula, prohledávání se ukončí. Pokud je při prohledávání nalezeno slovo, jehož délka je větší než c_2, což je další konstanta určená uživatelem, sníží se hodnota tohoto čítače na polovinu.

Prohledávání se ukončí také v případě, že je nalezeno slovo, jehož délka je větší než c_3. (Je zřejmé, že musí platit $c_2 < c_3$.) Pokud tedy algoritmus nalezne dostatečně dlouhé slovo, nevěnuje již příliš mnoho času hledání ještě delšího slova.

V případě, že uživatel zvolil, že chce dosáhnout lepší komprese, je používáno tzv. ***odložené vyhodnocování*** (lazy evaluation). V případě, že bylo nalezeno slovo a na výstup má být poslán odkaz na toto slovo, otestuje se nejdříve, co by se stalo, kdyby byl na výstup místo tohoto odkazu poslán literál. Pokud by následně bylo nalezeno slovo delší než dříve nalezené, je na výstup místo původně nalezeného odkazu skutečně poslán literál a celý proces se opakuje, to znamená, že se opět zjišťuje, co by se stalo, kdyby byl na výstup poslán místo nově nalezeného odkazu další literál. V opačném případě, to znamená, pokud by po poslání literálu na výstup nebylo nalezeno delší slovo než bylo nalezeno původně, je na výstup poslán původně nalezený odkaz.

Pokud je nalezeno slovo delší, než určitá konstanta c_4, v odloženém vyhodnocování se nepokračuje a na výstup je rovnou poslán nalezený odkaz.

V případě, že uživatel zvolil rychlejší kompresi, se výpočet zrychlí tím, že pokud je nalezeno slovo delší než určitá konstanta c_5 a na výstup je tedy poslán odkaz na toto slovo, přičemž okno je posunuto o délku tohoto slova, nepřidávají se při tomto posunu do hashovací tabulky nová slova. To vede ke značnému zhoršení komprese, ale ušetří se tím čas výpočtu.

Vzhledem k tomu, že hashovací funkce mapuje trojice znaků (tj. 24 bitů) na 15-bitové hodnoty, dochází samozřejmě k tomu, že různým trojicím znaků může odpovídat tentýž index v hashovací tabulce. Není tedy možné se spoléhat na to, že všechna slova v příslušném seznamu začínají stejnou trojicí znaků jako hledané slovo, a je tedy vždy nutné porovnat i první tři znaky.

Při hledání nejdelšího slova, si algoritmus udržuje informaci o délce dosud nejdelšího nalezeného slova. Označme tuto délku m. Dříve než začne algoritmus porovnávat hledané slovo $u_i u_{i+1} u_{i+2} \dots$ se slovem $u_j u_{j+1} u_{j+2} \dots$ znak po znaku, porovná spolu nejdříve znaky u_{i+m} a u_{j+m} a znaky u_{i+m-1} a u_{j+m-1}, protože pokud se slova neshodují v těchto znacích, tak nemá smysl je spolu porovnávat, protože toto porovnání určitě nepovede k nalezení delšího slova než je nejdelší dosud nalezené slovo.

3 Huffmanovo kódování

Při použití Huffmanova kódování je každý symbol nahrazován posloupností bitů. Délka této posloupnosti však může být pro různé symboly různá, pro symboly, které se vyskytují častěji, jsou použity kratší posloupnosti bitů, naopak pro symboly, které se vyskytují méně často jsou použity delší posloupnosti.

Posloupnost symbolů je tedy zakódována jako posloupnost bitů. Aby bylo možné tuto posloupnost bitů jednoznačně dekódovat, musí platit, že kód žádného znaku není prefixem žádného jiného znaku.

Huffmanův kód je vhodné si představit jako binární strom, kde každý vrchol kromě listů má právě dva potomky. Listy stromu jsou ohodnoceny symboly dané abecedy, kterou chceme kódovat. Hrany vedoucí z vrcholu do jeho levého potomka jsou ohodnoceny nulou, hrany vedoucí z vrcholu do jeho pravého potomka jsou ohodnoceny jedničkou. Tento strom se nazývá ***Huffmanův strom*** (Huffman tree).

V Huffmanově stromě můžeme cestu od kořene k listu popsat posloupností nul a jedniček, kterými jsou ohodnoceny hrany, kterými po této cestě procházíme. Kód symbolu je posloupnost nul a jedniček, která popisuje cestu od kořene k listu ohodnocenému tímto symbolem.

V Huffmanově stromu by mělo platit, že listy, které odpovídají znakům s velkou pravděpodobností leží blíže kořene než listy odpovídající znakům s malou pravděpodobností. Cílem je, aby kódy znaků s velkou pravděpodobností byly krátké.

Pokud se během procesu kódování Huffmanův strom nemění, hovoříme o ***statickém*** Huffmanově kódování, pokud se strom během kódování mění, mluvíme o ***adaptivním*** Huffmanově kódování. Nevýhodou statického Huffmanova kódování je, že je nutné spolu se zakódovanými daty uložit i popis Huffmanova stromu použitého při kódování nebo je nutné použít nějaký pevně zvolený strom, který pak ale nemusí odrážet skutečnou pravděpodobnost výskytu jednotlivých znaků. Proti adaptivnímu kódování je však statické kódování podstatně rychlejší.

V následujícím textu je popsáno pouze statické Huffmanovo kódování.

3.1 Konstrukce Huffmanova stromu

Při konstrukci Huffmanova stromu se předpokládá, že známe pravděpodobnosti výskytu jednotlivých znaků. V praxi se často místo pravděpodobnosti používá frekvence výskytu jednotlivých znaků, tj. kolikrát se daný znak v datech vyskytuje (v tom případě je však třeba, abychom měli data k dispozici již před začátkem kódování).

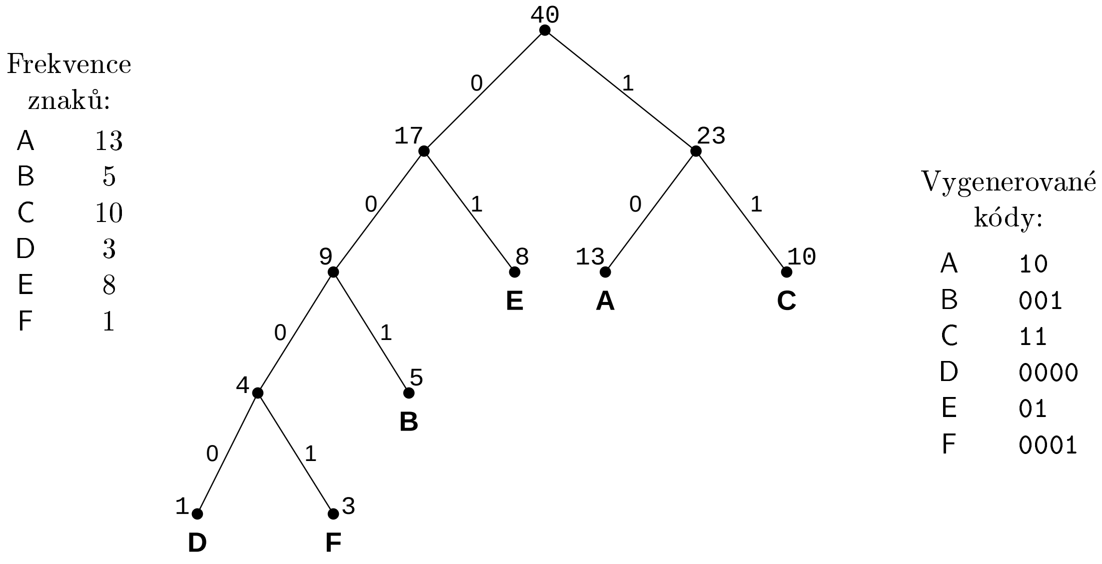
Huffmanův strom se konstruuje „zdola nahoru“, to znamená od listů ke kořeni.

Nejprve pro každý znak s nenulovou pravděpodobností (znaky s nulovou pravděpodobností se v datech nemohou vyskytovat, nemá tedy smysl pro ně vytvářet kódy) vytvoříme jeden list stromu ohodnocený tímto znakem. Tomuto listu přiřadíme pravděpodobnost tohoto znaku.

Ze všech vrcholů, které dosud nemají přiřazeného rodiče, vybereme dva vrcholy, které mají nejmenší pravděpodobnost. Vytvoříme nový vrchol, který bude společným rodičem obou těchto vrcholů. Jeho pravděpodobnost spočteme jako součet pravděpodobností jeho potomků. Tento krok opakujeme tak dlouho dokud nám nezůstane jediný vrchol bez rodiče — kořen stromu.

Příklad takové konstrukce je na obrázku 4 (místo pravděpodobností je v obrázku použit počet výskytů znaků).

Dá se snadno ukázat, že Huffmanův strom, který má n listů, má vždy celkem $2n-1$ vrcholů. Pro reprezentaci stromu v paměti můžeme tedy použít pole velikosti $2n-1$. Každý



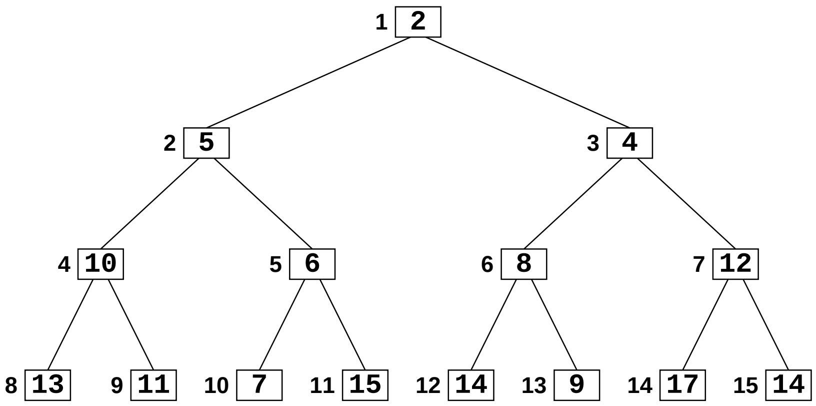
Obrázek 4: Příklad konstrukce Huffmanova stromu

prvek tohoto pole bude odpovídat jednomu vrcholu a bude obsahovat index jeho rodiče. Prvních n prvků pole budou listy a nové vrcholy budou přidávány jako prvky s indexy $n+1$ až $2n-1$. Rodič bude tedy vždy mít vyšší index než jeho potomci a prvek s indexem $2n-1$ bude kořenem stromu. Podobné pole se stejnými indexy je možné použít pro uložení pravděpodobností jednotlivých vrcholů.

Postup vytváření Huffmanova stromu je popsán algoritmem 3. Při vhodné implementaci je složitost tohoto algoritmu $O(n \log n)$, kde n je počet symbolů abecedy.

Algoritmus 3 Vytvoření Huffmanova stromu

Vstup: $\mathit{freq}[1], \ldots, \mathit{freq}[n]$ — pravděpodobnosti (resp. frekvence) jednotlivých znaků
  $Q \leftarrow \emptyset$
  for all $i \in \{1, 2, \ldots n\}$ taková, že $\mathit{freq}[i] > 0$ **do**
    $Q \leftarrow Q \cup \{i\}$
  end for
  $l \leftarrow n+1$
  while $|Q| > 1$ **do**
    najdi taková $u_1, u_2 \in Q$, pro která platí $\mathit{freq}[u_1], \mathit{freq}[u_2] \leq \mathit{freq}[u']$ pro $\forall u' \in Q \setminus \{u_1, u_2\}$
    $Q \leftarrow Q \setminus \{u_1, u_2\}$
    $par[u_1] \leftarrow l, \;\; par[u_2] \leftarrow l$
    $\mathit{freq}[l] \leftarrow \mathit{freq}[u_1] + \mathit{freq}[u_2]$
    $Q \leftarrow Q \cup \{l\}$
    $l \leftarrow l+1$
  end while
Výstup: pole par obsahující indexy rodičů jednotlivých vrcholů



Obrázek 5: Příklad haldy

3.2 Použití haldy při konstrukci Huffmanova stromu

Množina Q v algoritmu 3 je v podstatě tzv. ***prioritní fronta*** (priority queue), což je datová struktura, do které je možné přidávat a ze které je možné vybírat prvky. Každý prvek má přiřazenu určitou prioritu. Prvky je možné přidávat do fronty v libovolném pořadí, ale vybrat je možné pouze prvek s nejvyšší prioritou (nebo jeden z takových prvků pokud je jich víc).

V případě vytváření Huffmanova stromu je priorita dána pravděpodobností (resp. frekvencí) výskytu daného vrcholu — čím bude jeho pravděpodobnost menší, tím bude jeho priorita vyšší.

Pro implementaci prioritní fronty se používá datová struktura, která se nazývá ***halda*** (heap). Jedná se v podstatě o binární strom uložený v poli takovým způsobem, že pokud je nějaký vrchol uložen v prvku pole s indexem i, pak jeho potomci mají indexy $2i$ a $2i+1$ (index rodiče prvku s indexem i je tedy možné spočítat jako $\lfloor i/2 \rfloor$). Kořen stromu má index 1. Navíc musí platit, že hodnota každého prvku je menší nebo rovna hodnotám jeho potomků (pokud nějaké má). Nejmenší prvek je tedy uložen v kořeni stromu, to znamená v prvním prvku pole. Pokud halda obsahuje n prvků, je výška binárního stromu $O(\log n)$.

Příklad takové haldy je na obrázku 5. U jednotlivých vrcholů jsou uvedeny indexy a hodnoty prvků pole.

Označme vlastnost prvku haldy, že jeho hodnota je menší nebo rovna hodnotám jeho potomků, jako vlastnost A. Počet prvků haldy budeme označovat n.

Pokud o nějakém prvku víme, že pro všechny prvky, které leží ve stromě pod ním, platí, že mají vlastnost A, pak můžeme snadno dosáhnout toho, že i tento prvek bude mít vlastnost A: Porovnáme jeho hodnotu s hodnotami obou jeho potomků. Pokud je jeho hodnota menší nebo rovna těmto hodnotám, prvek už má vlastnost A. V opačném případě ho vyměníme s menším z jeho potomků a znovu ho porovnáváme s jeho potomky. Celý postup opakujeme tak dlouho dokud nebude mít vlastnost A. Je zřejmé, že tato úprava haldy vyžaduje v nejhorším případě čas $O(\log n)$.

Při vytváření haldy se postupuje zdola nahoru. Začínáme s haldou, ve které jsou prvky umístěny v libovolném pořadí, to znamená, že nemusí mít vlastnost A. Využijeme toho, že listy stromu, to znamená prvky s indexy $n/2+1$ až n již vlastnost A mají (nemají totiž žádné potomky). Pak provádíme výše uvedenou úpravu haldy postupně pro prvky s indexy $n/2$ až 1, přičemž využíváme toho, že v okamžiku, kdy provádíme tuto úpravu pro určitý prvek, máme zajištěno, že všechny prvky ležící ve stromě pod ním již vlastnost A mají. Časová složitost vytvoření haldy je $O(n)$. (Na první pohled se zdá, že tato složitost je $O(n \log n)$, podrobnější analýza ale ukáže, že pro většinu prvků vyžaduje jejich zařazení i v nejhorším případě méně než $O(\log n)$ operací.)

Při výběru prvku z haldy vybereme prvek s indexem 1, to znamená kořen stromu a tedy prvek s nejmenší hodnotou. Na jeho místo přesuneme prvek s indexem n, velikost haldy n zmenšíme o jedničku a postupem popsaným v předchozích odstavcích zajistíme, že kořen bude mít opět vlastnost A. Operace vybrání nejmenšího prvku z haldy vyžaduje čas $O(\log n)$.

Při přidávání nového prvku postupujeme tak, že ho přidáme na pozici $n+1$, velikost haldy zvětšíme o jedničku a nově přidaný prvek porovnáme s hodnotou jeho rodiče. Pokud je nově přidaný menší než jeho rodič, vyměníme ho s rodičem a opět ho porovnáme s prvkem nad ním. Tento postup opakujeme tak dlouho, dokud je prvek menší než jeho rodič. Celá operace opět vyžaduje čas $O(\log n)$.

Pokud používáme haldu k implementaci prioritní fronty, odpovídají hodnoty prvků prioritám těchto prvků, prvkům s vyšší prioritou budou přiřazeny menší hodnoty. Při vybírání prvku z haldy bude vždy vybrán prvek s nejvyšší prioritou. Jak přidání, tak vybrání jednoho prvku z nebo do prioritní fronty pak vyžaduje čas $O(\log n)$, kde n je počet prvků, které se právě ve frontě nacházejí.

Je dobré si všimnout, že při použití haldy při vytváření Huffmanova stromu není nutné používat operaci přidávání nového prvku. Na začátku totiž nejprve přidáme prvky pro všechny znaky, které chceme kódovat. Ty můžeme uložit neuspořádaně a pak z nich vytvořit haldu postupem, který byl popsán výše. Při další práci s haldou se pak střídá vybírání dvou prvků s přidáváním jednoho prvku. Při vybírání můžeme první prvek vybrat běžným způsobem (tj. nahradit ho prvkem z konce haldy), zatímco po výběru druhého prvku nahradíme tento prvek nově přidávaným prvkem.

3.3 Vygenerování kódů pro jednotlivé znaky

Konstrukce Huffmanova stromu popsaná v kapitole 3.1 nemusí vést vždy k témuž výsledku. Jak vypadá výsledný strom závisí na tom, v jakém pořadí vybereme stejně pravděpodobné vrcholy z prioritní fronty a také na tom, který ze dvou vybraných vrcholů přiřadíme nově vytvořenému vrcholu jako levého potomka a který jako pravého potomka.

Jaké jsou však konkrétní kódy přiřazené jednotlivým symbolům, není příliš důležité. Z hlediska komprese jsou podstatné pouze délky těchto kódů.

Jestliže používáme statické Huffmanovo kódování, je většinou potřeba nějakým způsobem spolu s kompresovanými daty uložit informaci o použitém Huffmanově stromu, aby bylo možné data dekódovat. Jako vhodný způsob uložení stromů se jeví právě uložení délek kódů jednotlivých symbolů. Problém je, že tyto délky neurčují kód jednoznačně, může existovat mnoho různých Huffmanových kódů s danými délkami. Pokud však určíme, že kód musí navíc splňovat následující dvě podmínky, bude určen jednoznačně:

1. Kratší kódy budou lexikograficky menší než delší kódy;
2. Kódy stejné délky budou lexikograficky uspořádány ve stejném pořadí jako symboly, které reprezentují.

Při generování kódů se strom vytvořený algoritmem 3 použije pouze k zjištění délek kódů jednotlivých symbolů. Podle těchto délek se pak vygenerují kódy pro jednotlivé znaky tak, aby výsledný kód splňoval výše uvedené dvě podmínky.

Nejprve je třeba zjistit délky kódů jednotlivých symbolů. Tuto informaci zjistíme snadno z pole *par*, které obsahuje indexy rodičů jednotlivých vrcholů a které bylo vypočítáno algoritmem 3.

Při výpočtu použijeme ještě jedno pole stejné velikosti, do kterého budeme ukládat informaci o vzdálenosti daného vrcholu od kořene. Začneme prvkem s nejvyšším indexem, tj. kořenem, kterému přiřadíme vzdálenost 0, a pokračujeme prvky se stále nižšími a nižšími indexy. U každého z nich určíme jeho vzdálenost ze vzdálenosti jeho rodiče.

Při tomto výpočtu využíváme toho, že rodič má vždy vyšší index než jeho potomci, takže v okamžiku, kdy zjišťujeme vzdálenost potomka, už vzdálenost rodiče známe. Časová složitost tohoto výpočtu je $O(n)$, kde n je počet symbolů abecedy.

Pokud je abeceda tvořena n symboly, může být délka nejdelšího kódu až $n-1$ bitů. S příliš dlouhými kódy se špatně pracuje, proto je vhodné maximální délku kódů omezit. Například ve formátu DEFLATE je maximální povolená délka kódu 15 bitů. Výhodou je, že kód každého znaku může být uložen jako celé číslo, se kterým se pak pracuje pomocí logických operací a bitových posunů. Další výhodou je, že na uložení informace o délce kódu pro jeden znak potřebujeme jen malý počet bitů.

Kódová slova, jejichž délka přesahuje maximální povolenou délku, se zkrátí na tuto délku, s tím, že se o odpovídající počet bitů prodlouží kódy znaků, jejichž délka je menší. Jako znaky, jejichž kódy se prodlouží, je vhodné zvolit znaky s co nejmenší pravděpodobností.

Tyto operace je jednodušší provádět na poli, které obsahuje celkové počty znaků s kódy dané délky, tedy kde i-tý prvek obsahuje celkový počet kódů délky i. Podle tohoto pole se pak přiřadí jednotlivým znakům délky kódů podle jejich pravděpodobnosti. K takové operaci je vhodné mít k dispozici znaky uspořádané podle pravděpodobnosti. Pole obsahující znaky uspořádané podle pravděpodobnosti můžeme vytvořit během vytváření Huffmanova stromu, neboť prvky vybíráme z prioritní fronty v pořadí od nejméně pravděpodobných k nejvíce pravděpodobným.

Toto vše můžeme opět provést v čase $O(n)$.

Nyní, když známe délky kódů všech znaků, můžeme přistoupit k vlastnímu vygenerování těchto kódů.

Nejdříve spočítáme celkové počty kódů pro jednotlivé délky. Tyto hodnoty uložíme do pole *bl_count*, kde $bl\_count[i]$ obsahuje celkový počet kódů délky i.

Z těchto hodnot vypočteme hodnoty lexikograficky nejmenších kódů pro každou délku, přičemž tyto hodnoty budou uloženy jako celá čísla — kód délky i bude uložen v dolních i bitech čísla a to směrem od významnějších bitů k méně významným. K těmto kódům pak snadno najdeme lexikograficky následující kódy, protože pokud kód reprezentujeme jako binární číslo, dostaneme lexikograficky následující kód zvětšením tohoto čísla o jedničku. Celý postup je popsán v algoritmu 4. Algoritmus má opět časovou složitost $O(n)$.

Mějme například abecedu $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}\}$ s délkami kódů pro jednotlivé symboly $(3, 3, 3, 3, 3, 2, 4, 4)$. Nejprve určíme hodnoty v poli *bl_count*:

$$bl\_count[1] = 0 \quad bl\_count[2] = 1 \quad bl\_count[3] = 5 \quad bl\_count[4] = 2$$

Algoritmus 4 Vygenerování kódů jednotlivých znaků při Huffmanově kódování

Vstup: $bl\_count[1], \ldots, bl\_count[MAX\_BITS]$ — celkové počty kódů jednotlivých délek
$len[1], \ldots, len[n]$ — délky kódů jednotlivých symbolů abecedy

$c \leftarrow 0$
$bl\_count[0] \leftarrow 0$
for $j = 1$ to MAX_BITS **do** — výpočet lex. nejmenšího kódu pro každou délku
  $c \leftarrow (c + bl\_count[j-1]) * 2$
  $next\_code[j] \leftarrow code$
end for
for $i = 1$ to n **do** — přiřazení kódů jednotlivým symbolům
  if $len[i] \neq 0$ **then**
    $code[i] \leftarrow next\_code[len[i]]$
    $next\_code[len[i]] \leftarrow next\_code[len[i]] + 1$
  end if
end for

Výstup: pole *code* obsahující kódy jednotlivých symbolů

Dále vypočteme lexikograficky nejmenší kód pro každou délku:

$$\begin{array}{rcrl} next\_code[1] & = & 0 & (\texttt{0}) \\ next\_code[2] & = & 0 & (\texttt{00}) \\ next\_code[3] & = & 2 & (\texttt{010}) \\ next\_code[4] & = & 14 & (\texttt{1110}) \end{array}$$

Nakonec vygenerujeme kódy pro všechny symboly:

Symbol	Délka	Kód
A	3	010
B	3	011
C	3	100
D	3	101
E	3	110
F	2	00
G	4	1110
H	4	1111

Z toho, co bylo dosud řečeno, vyplývá, že časová složitost vytvoření Huffmanova kódu při známých pravděpodobnostech výskytu symbolů je $O(n \log n)$, kde n je počet symbolů abecedy. Časově nejnáročnější je vytvoření Huffmanova stromu, ostatní operace je možné provést v čase $O(n)$.

3.4 Efektivní implementace Huffmanova stromu pro dekódování

Nejjednodušší způsob, jak dekódovat data zakódovaná Huffmanovým kódováním, je vytvořit Huffmanův strom podobně jako byl vytvořen při kódování (s využitím informace o délkách kódů jednotlivých symbolů).

Tento strom můžeme reprezentovat v paměti skutečně jako strom, takové řešení je však dosti pomalé, protože při dekódování musíme postupovat po jednotlivých bitech. Jestliže bude délka kódu nějakého symbolu d bitů, bude dekódování tohoto symbolu vyžadovat čas $O(d)$.

Jiné možné řešení spočívá v tom, že vytvoříme tabulku velikosti 2^m, kde m je délka nejdelšího kódu. Jednotlivé kódy pak budou představovat m-bitová čísla použitá jako indexy do této tabulky, přičemž kódy, jejichž délka je menší než m, budou doplněny na m bitů všemi možnými posloupnostmi nul a jedniček. Prvky tabulky budou obsahovat symboly odpovídající jednotlivým kódům a informaci o délce kódu daného symbolu. Prvky obsahující symboly s kratšími kódy než m budou v tabulce obsaženy vícekrát, konkrétně $2^{m-m'}$ krát, kde m' je délka kódu daného symbolu.

Při dekódování přečteme ze zakódovaných dat vždy m bitů, které použijeme jako index do tabulky, čímž zjistíme odpovídající symbol a délku jeho kódu. Podle této délky zjistíme, kolik z m přečtených bitů není součástí kódu právě dekódovaného znaku, a tyto bity použijeme při dalším čtení.

Tento přístup je velmi rychlý, dekódování jednoho znaku vyžaduje konstantní čas ($O(1)$). Velkou nevýhodou však je, že tabulka může být velmi velká, takže vytvoření takové tabulky vyžaduje mnoho času a paměti.

Řešení, které má výhody obou výše popsaných metod a které zároveň eliminuje jejich nevýhody, využívá opět tabulku, ale tabulku podstatně menší. Velikost této tabulky je 2^d, kde d je nějaká vhodně zvolená konstanta, vhodná je hodnota d stejná nebo o něco málo větší než je průměrný počet bitů na jeden symbol. Tabulka bude vypadat podobně jako výše popsaná tabulka s tím rozdílem, že prvky s indexy, které odpovídají prefixům kódů delších než d bitů, nebudou obsahovat dekódovaný symbol, ale ukazatel na další tabulku. Ta bude vytvořena podobně jako právě popsaná tabulka, ale bude obsahovat pouze informace o kódech začínajících daným prefixem. K indexování v této tabulce bude použito následujících d bitů (může to být případně i méně bitů). Tato tabulka opět může obsahovat ukazatele na další tabulky, takže celá hierarchie může mít i více úrovní.

Každá tabulka by měla obsahovat informaci o tom, kolik bitů se používá pro indexování v této tabulce. Kromě dekódovaných symbolů by měly prvky tabulky obsahovat také informaci o tom, kolik z bitů použitých pro indexování je skutečně součástí kódu daného znaku.

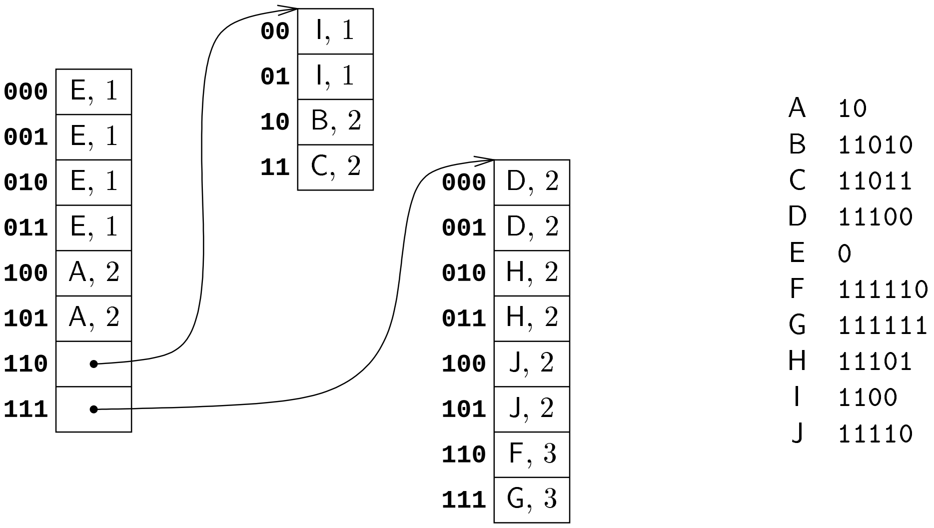
Symboly s krátkými kódy se v datech vyskytují nejčastěji. Takové symboly budou nalezeny již v první tabulce a tedy v konstantním čase. Procházet do dalších tabulek je nutné pouze pro symboly s delšími kódy, které se v datech vyskytují méně často. Tabulky navíc nejsou příliš velké, takže je možné je rychle vytvořit a nezabírají příliš mnoho paměti.

Příklad takové hierarchie tabulek použitých k dekódování Huffmanova kódu je na obrázku 6.

4 Kombinace obou metod ve formátu DEFLATE

Ve formátu DEFLATE tvoří kompresovaná data posloupnost bitů. Tato posloupnost je uložena jako posloupnost bytů, přičemž 8 bitů je v jednom bytu uloženo v pořadí od nejméně významného k nevýznamnějšímu bitu.

Posloupnost bitů obsahuje jednak Huffmanovy kódy, jednak binárně zakódovaná čísla. U čísel je jako první uložen nejméně významný bit.



Obrázek 6: Tabulky použité k dekódování Huffmanova kódu

4.1 Bloky

Kompresovaná data jsou tvořena posloupností bloků. Počet bloků i jejich délka mohou být libovolné.

Každý blok začíná trojicí bitů, kde první z těchto bitů určuje, zda se jedná o poslední blok, a zbývající dva bity určují, jaká metoda byla použita pro kompresi dat v daném bloku. Celkem jsou k dispozici tři různé metody:

1. Data nejsou kompresována. Na začátek dat je přidána informace o jejich délce, za kterou následují zkopírovaná nezkompresovaná data.
2. Data jsou kompresována použitím metody LZ77 a Huffmanova kódování. Pro Huffmanovo kódování jsou použity pevně definované Huffmanovy stromy.
3. Data jsou kompresována stejnou metodou jako v předchozím případě, s tím rozdílem, že pro Huffmanovo kódování jsou použity stromy odrážející skutečné rozložení pravděpodobností výskytu jednotlivých symbolů v datech. V tomto případě musí být součástí kompresovaných dat také popis použitých Huffmanových stromů.

Velikost bloku sice může být libovolná, ale v praxi je omezena množstvím paměti, které má aplikace k dispozici. Data zkompresovaná metodou LZ77 jsou totiž ukládána do paměti až do okamžiku, kdy aplikace s použitím určitých heuristik „usoudí", že by bylo vhodné blok ukončit. Poté se spočítá, jaká by byla velikost kompresovaných dat při použití každé z výše uvedených tří metod, a je zvolena ta z nich, která dává nejlepší výsledek.

Je vhodné, aby velikost bloku nebyla příliš velká, aby aplikace mohla rychleji reagovat na změny v charakteru kompresovaných dat (například kód programu, obsahující jen málo redundance, může být následován značně redundantními daty).

4.2 Uložení kompresovaných dat

Výsledkem komprese algoritmem LZ77 je posloupnost literálů (což jsou hodnoty z intervalu 0–255) a odkazů (což jsou dvojice ($vzdálenost, délka$), kde vzdálenost je z intervalu 1–32768 a délka je z intervalu 3–258).

Odkazy jsou ukládány v pořadí délka, vzdálenost.

Pro Huffmanovo kódování jsou literály a délky spojeny do jediné abecedy. Tato abeceda obsahuje symboly označené čísly 0–285, kde symboly 0–255 jsou literály, symbol 256 je speciální literál použitý k označení konce bloku a symboly 257–285 odpovídají délkám. Každý ze symbolů 257–285 reprezentuje určitý interval délek. Za tímto symbolem pak následuje binární číslo, které určuje hodnotu v rámci tohoto intervalu. Počet bitů tohoto čísla je dán specifikací formátu DEFLATE a je různý pro různé symboly, může to být 0–5 bitů.

Specifikací je například určeno, že symbol 269 reprezentuje délku z intervalu 19–22 a je následován dvěma bity. Mají-li například tyto bity hodnotu `10`, tedy decimálně 2, reprezentuje hodnota 269 následovaná těmito bity délku $19 + 2 = 21$.

Pro menší délky jsou voleny kratší intervaly, takže symboly reprezentující tyto délky jsou následovány menším počtem bitů.

Další abeceda tvořená symboly očíslovanými 0–29 slouží k reprezentaci vzdáleností. Podobně jako v případě délek odpovídá každému symbolu určitý interval vzdáleností a za symbolem následuje určitý počet bitů, které určují vzdálenost v rámci tohoto intervalu. Těchto bitů může být 0–13.

Pro každou z těchto dvou abeced je vytvořen samostatný Huffmanův strom, jeden pro literály a délky, druhý pro vzdálenosti. Při kódování jsou pak symboly z těchto abeced nahrazeny odpovídajícími Huffmanovými kódy, přičemž, jak bylo popsáno výše, za kódy, které označují délky a vzdálenosti v odkazech, mohou následovat další bity.

4.3 Uložení informací o Huffmanových stromech

V případě použití metody 2 jsou pevně stanoveny Huffmanovy stromy použité při kódování. Tyto stromy jsou stanoveny tak, že kódy pro literály mají délku 8–9 bitů, kódy pro délky 7–8 bitů a kódy pro vzdálenosti 5 bitů. V tomto případě není nutné ukládat žádné informace o těchto stromech.

V případě, že je použita metoda 3, musí být před vlastními kompresovanými daty uložena informace o Huffmanových stromech použitých při kódování, aby bylo možné později tato data dekódovat.

Z toho, co bylo řečeno v kapitole 3.3, vyplývá, že k zrekonstruování Huffmanova stromu použitého při kódování nám stačí znát délky kódů jednotlivých symbolů. Před vlastní kompresovaná data jsou tedy uloženy posloupnosti délek kódů symbolů pro oba stromy, přičemž tato informace je rovněž kompresována.

Posloupnosti délek kódů pro oba stromy jsou uloženy za sebou a jsou spojeny v jedinou posloupnost. Tato posloupnost obsahuje pouze čísla z intervalu 0–15 (0 je použita pro označení symbolů, kterým není přiřazen žádný kód, maximální povolená délka kódu je 15 bitů). Nemusí být nutně uvedeny délky kódů pro všechny symboly, nýbrž mohou být uvedeny pouze délky pro prvních k symbolů dané abecedy. Při dekódování se délky zbylých symbolů nastaví na 0. Toto k však musí být v datech také uloženo.

Při kompresi této posloupnosti je nejprve použita metoda RLE (run-length encoding) a poté Huffmanovo kódování.

Při použití metody RLE, jsou sekvence stejných hodnot nahrazeny informací o počtu opakování. Konkrétně je používána abeceda symbolů očíslovaných 0–18, kde symboly 0–15 reprezentují délky kódů 0–15, symbol 16 označuje, že předchozí hodnota má být zopakována 3 krát až 6 krát, přičemž konkrétní počet opakování je určen následujícími dvěma bity.

Symbol 17 označuje, že hodnota 0 má být zopakována 3–10 krát (počet opakování je určen následujícími třemi bity) a symbol 18, že má být zopakována 11–138 krát (počet opakování je určen následujícími sedmi bity).

Symboly této abecedy jsou zakódovány pomocí Huffmanova kódování, přičemž maximální povolená délka kódu je 7 bitů. Před informacemi o délkách kódů musí být tedy nejprve uloženy délky kódů použitých při jejich zakódování. Tyto délky jsou ovšem uloženy v tomto pořadí: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. Opět nemusí být uloženy všechny, ale může být uloženo jen několik prvních délek, přičemž číslo udávající počet skutečně uložených délek musí být uloženo před nimi. Ostatní neuvedené délky se doplní nulami. Toto pořadí bylo zvoleno, protože se předpokládá, že v tomto pořadí klesá pravděpodobnost, že se daná hodnota bude mezi délkami vyskytovat.

Reference

[1] Gailly, J. L., Adler, M.: *Zdrojové kódy a dokumentace programu* `gzip`. Jsou dostupné například na adrese ftp://prep.ai.mit.edu/pub/gnu/gzip/.

[2] Deutsch, P.: *RFC 1951: DEFLATE Compressed Data Format Specification version 1.3.* 1996. Text je dostupný například na adrese ftp://ftp.uu.net/pub/archiving/zip/doc/.

[3] Feldspar, A.: *An Explanation of the* DEFLATE *Algorithm.* 1997. Text je dostupný na adrese http://www.gzip.org/deflate.html.

[4] Melichar, B.: *Textové informační systémy.* Vydavatelství ČVUT, 1996.

[5] Gutmann, P.: *Introduction to data compression.* Text je dostupný na adrese http://www.faqs.org/faqs/compression-faq/part2/section-1.html.

[6] Fiala, E. R., Greene, D. H.: *Data Compression with Finite Windows.* Communications of ACM, 32, 4 (1989), pp. 490–505.

[7] Burrows, M., Wheeler, D. J.: *A Block-sorting Lossless Data Compression Algorithm.* SRC Research Report 124, 1994. Text je dostupný na adrese http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html.

Obsah