Kompresia dát a jej použitie alebo

Veľa muziky na malom diskovom priestore

Záverečná práca

PETER VOOK

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta
2016

1 Reálna situácia alebo Zo života

Anička a Zuzka si často posielajú správy. Majú ale problém. Dáta poskytnuté ich operátorom sú obmedzené a rodičia im odmietajú neustále obnovovať kredit. Chcú od nás teda, aby sme im poradili, ako majú svoje správy posielať tak, aby minuli čo najmenej dát a aby obsah prijatej správy bol rovnako zrozumiteľný ako pôvodná správa.

Obsahom správ sú najčastejšie texty a obrázky. Niekedy ide o zmysluplné vety, niekedy sú to číslice napísané za sebou a niekedy dokonca len postupnosti bielych a čiernych znakov reprezentovaných nulami a jednotkami. Nebudeme preto riešiť, čo si dievčatá posielajú, ale budeme sa im snažiť poradiť pre rôzne prípady.

V tejto práci sa teda budeme zaoberať metódami spracovania dát s cieľom zmenšiť ich objem tak, aby sa **zachovala informácia** uložená v nich. Samozrejme potrebujeme metódy, pri ktorých vie druhá strana dekódovať informáciu, ktorú zakódoval odosielateľ. Teda ak nebude triviálne, pozrieme sa aj na dekódovanie. Najprv si však musíme zadefinovať nejaké pojmy.

2 Definícia pojmov alebo Čo treba vedieť

Proces, ktorým sa budeme zaoberať, sa nazýva **kompresia**. Tento sa definuje takto:

Definícia 1

Kompresia dát je proces, pri ktorom sa spracúvajú dáta, pričom cieľom spracovania je, aby bolo množstvo výstupných dát čo najmenšie. V praxi sa snažíme o čo najmenšiu časovú náročnosť. Obyčajne sa aplikuje s účelom zmenšenia dátového toku pri prenose alebo pamäťového priestoru pri ukladaní informácie.

Podľa výslednej informácie sa kompresia delí na 2 základné druhy:

Definícia 2

Stratová kompresia (tzv. lossy) je kompresia, pri ktorej sú niektoré informácie nenávratne stratené (nedajú sa zrekonštruovať). Výsledný signál po dekompresii nie je totožný s pôvodným signálom, ale je mu do istej miery podobný. Veľkou výhodou je významné zmenšenie súboru. Stratová kompresia sa využíva hlavne pri spracovaní obrazu, videa a zvuku (pri ich vnímaní si chýbajúce údaje človek nevšimne, resp. si ich vie domyslieť).

Definícia 3

Bezstratová kompresia (tzv. lossless) je kompresia, po použití ktorej je možné pôvodný signál obnoviť bez akejkoľvek straty informácie. Používa sa v prípadoch, keď je zachovanie pôvodnej informácie nutnou podmienkou. V praxi sa aplikuje hlavne pri prenose textu alebo počítačových dát.

Základy kompresie dát formuloval v roku 1948 Claude Elwood Shannon (nazývaný tiež otec teórie informácie) vo svojej práci 'A Mathematical Theory of Communication'. V nej zaviedol pojem entropia ako základný limit pre bezstratovú kompresiu dát. Inak povedané, pri bezstratovej kompresii nemôžeme dáta "zhustiť" viac, než dovoľuje ich entropia. Shannon definoval entropiu takto:

> "Entropia je miera množstva neurčitosti o nejakom náhodnom deji odstránená realizáciou tohto deja."

Entropia znamená mieru neurčitosti v správe. Naopak, informáciu Shannon chápal ako odstránenie tejto neurčitosti.

Definícia 4

Majme zdrojovú abecedu A, ktorá má m symbolov: $s_1, s_2, \dots, s_m$. Pre každý symbol s_i máme definovanú početnosť jeho výskytu p_i.

Informačná hodnota symbolu je definovaná ako $I_i = -\log_2 p_i$ bitov.

Definícia 5

Entropia H je potom priemerná informačná hodnota na jeden symbol a je definovaná takto:

$$H = -\sum_{i=1}^{m} p_i \cdot \log_2 p_i$$

Formálne dodefinujeme $0 \cdot \log 0 = 0$.

Po (bezstratovej) komprimácii teda prenesieme rovnaké množstvo dát, ale použijeme na to menšie množstvo symbolov. Informačná hodnota symbolu je tým pádom väčšia; jedným symbolom prenesieme väčšie množstvo informácie. Komprimovaním sa teda zväčšuje entropia dát. Jednotkou entropie je *bit/symbol.*

Na vyjadrenie efektívnosti kompresie použijeme tieto ukazovatele:

Definícia 6

Pomer kompresie P je definovaný takto: $P = \frac{\text{in}}{\text{out}}$, kde in je veľkosť vstupného (nekomprimovaného) súboru dát a out je veľkosť výstupného súboru (po komprimácii).
My chceme postupy, pri ktorých je pomer kompresie väčší, než 1. Ak $P < 1$, tak hovoríme o expanzii (väčšinou nežiadúca).

Faktor kompresie F definujeme takto: $F = \frac{1}{P} = \frac{\text{out}}{\text{in}}$, kde out a in sú ako v predchádzajúcej časti.
$F < 1$ znamená kompresiu, $F > 1$ expanziu.

Kompresiu delíme aj podľa výpočtovej zložitosti na 2 typy:

Definícia 7

Symetrická kompresia je taká pri ktorej proces kompresie a proces dekompresie trvajú zhruba rovnako dlho (v praxi napr. JPEG).
Nesymetrická je taká, kde jedna z operácií (kompresia alebo dekompresia) vyžaduje výrazne viac času. V praxi je to obyčajne operácia kompresie (napr. MPEG).

3 Algoritmy kompresie alebo Ideme pomáhať

Najprv sa pozrieme na metódy používané pri bezstratovej kompresii.

RLC – Run-Length kódovanie

V praxi sa množstvo dokumentov dá prenášať pomocou faxu. Pri tomto prenose sú dokumenty kvantizované do binárnej úrovne. Na každom riadku dokumentu sa za sebou vyskytuje veľké množstvo bodov s rovnakou intenzitou (v tomto prípade čierne alebo biele). RLC kódy sa používajú práve pri komprimovaní takýchto dokumentov. Základným typom RLC je jedno-dimenzionálne (tzv. 1-D) kódovanie.

1-D Run-Length kódovanie

V tejto verzii RLC kódu sa každý riadok považuje za nezávislý, teda komprimujeme iba horizontálne. Kódujeme teda postupnosti bielych a čiernych znakov (núl a jednotiek) a kódujú sa vzdialeností prechodov postupností (teda dĺžky "jednofarebných" podpostupností). Predpokladá sa, že začiatok postupností sú biele znaky (ak by sme túto konvenciu porušili, tak vytvoríme dokument inverzný k pôvodnému, čo však nie je problém) – ak je prvý znak čierny, tak dĺžka bielej sekvencie je 0. Na konci riadku sa ešte pridá špeciálny znak `EOL` (End-of-line).

Ukážka:
Na ukážku si vezmeme postupnosť: **0 0 0 1 1 1 1 1 0 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1**
Výsledkom kódovania je postupnosť: **3 5 1 2 6 5 1 7**

Veľmi však závisí na tom, koľko bitov sme ochotní poskytnúť na kódovanie vzdialeností.
Ak napríklad poskytneme 3 bity, tak po zakódovaní postupnosti 3 5 1 2 6 5 1 7 máme výstupnú sekvenciu **011 101 001 010 110 101 001 111**, ktorá zaberá 24 bitov oproti pôvodným 30 bitom. Faktor kompresie je teda 0,8. Avšak ak kódujeme vzdialenosti 4 bitmi, tak máme výstupnú postupnosť **0011 0101 0001 0010 0110 0101 0001 0111**, ktorá zaberá 32 bitov a pri porovnaní s pôvodnými 30 bitmi vidíme, že ide o expanziu.

Významne teda záleží na tom, ako si zvolíme maximálnu dĺžku sekvencie znakov. Práve tu nastupuje práca matematikov, ktorí musia nájsť správny počet. Existuje niekoľko spôsobov, aby sa podarilo nájsť optimálne kódovanie:

- štatisticky prehľadať kódované dáta a zistiť, ktoré sú najpočetnejšie
- metóda pokus-omyl – skúšame rôzne počty a hľadáme najoptimálnejšie
- výsledok kódovania nerobíme binárne, ale využijeme niektoré z kódovaní s premenlivou dĺžkou výstupného kódu (napr. Huffmanovo kódovanie)

Viacrozmerné verzie

Aj keď sa RLC najčastejšie používa na komprimáciu bitového prúdu, existujú rôzne varianty, ktoré nepracujú na bitovej úrovni, ale na inej, napr. bajtovej. Keď máme viac, ako dve úrovne, tak nám nestačí kódovať len počet znakov, ale aj znaky samotné. Kompresia teda môže vyzerať takto:
AAAAAAAABAAAAACCCCCAAABBBAAAAAACCC → **8A1B5C5B3A3B6A3C**

Nevýhodou je, že jedna samostatná hodnotu v prúde bytov sa kóduje dvojicou bytov (počet-prvok), a preto pre nevhodné vstupné dáta môže byť kompresia neúčinná a môže dôjsť až k expanzii s kompresným faktorom 2. Toto sa dá odstrániť tak, že sa pomocou RLC budú kódovať až postupnosti dlhšie než 2 (vhodné sú preto vstupné súbory, ktoré obsahujú dlhé sekvencie rovnakých znakov) Na úpravu vstupu tak, aby obsahoval viac sekvencií rovnakých znakov takisto existujú postupy, napríklad Burrows-Wheelerova transformácia, kde sa zo vstupu spravia všetky možné permutácie, tieto sa zoradia a ako výstup sa zoberú posledné znaky permutácií. Toto sa potom dešifruje pomocou sufixového stromu.

V praxi sa toto kódovanie využívalo pri prenose televízneho signálu pred rokom 1967. V počítačovej grafike sa používa pri kompresii obrazu, ktorý má veľké plochy rovnakej farby. Používal sa pri komprimácii počítačových ikon a aj pri uvítacej obrazovke v operačných systémoch Windows 3.x. Spolu v kombinácii s inými kódovaniami sa využíva aj dnes pri faxovaní. Dodnes ho ako pomocný algoritmus používa aj kompresia JPEG, konkrétne pri kvantizácii farebných blokov.

Huffmanovo kódovanie

Huffmanov algoritmus generuje binárne stromy (strom, v ktorom každý vrchol má najviac dvoch synov), kde cesty z koreňa do listu umožňujú vytvoriť kódové slová, pričom najčastejšie vstupné symboly majú najkratší výstupný kód.

Definícia 8

Prefixové kódy sú také, ktoré spĺňajú podmienku prefixu. Podmienka prefixu je splnená vtedy, keď žiadne kódové slovo nie je prefixom iného kódového slova[5].

Prefixový kód je jednoznačne dekódovateľný. Každý prefixový strom sa dá zobraziť pomocou binárneho stromu.

Kód s kódovou abecedou {1, 21, 22, 221, 222, 24, 35, 355, 7} nie je prefixový, lebo symbol 22 je prefixom iných kódových symbolov, konkrétne 221 a 222, takisto 35 je prefixom 355. Ak by sme chceli dekódovať postupnosť 2472217, tak to nemôžeme jednoznačne určiť, lebo pôvodná postupnosť by mohla byť 24, 7, 22, 1, 7 ale aj 24, 7, 221, 7.

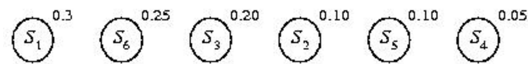
Tento algoritmus vynašiel David A. Huffman v roku 1951, počas štúdia na MIT, kde mu bola pridelená semestrálna práca na tému Najefektívnejší binárny kód. Huffman prišiel s nápadom frekvenčného binárneho stromu a dokázal, že jeho metóda je najefektívnejšia. Myšlienkou stavať strom odhora nadol, nie naopak, odstránil najväčšiu trhlinu kódovania, ktoré vymyslel jeho profesor, Robert Fano.

Ukážka:
Na konkrétnom príklade[2] si ukážeme konštrukcia Huffmanovho kódu.

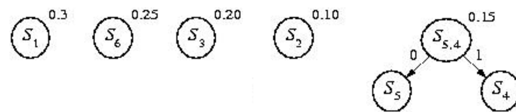
1. V prvom kroku si prejdeme súbor a vytvoríme štatistiku početností výskytov znakov.

Zdrojový symbol	S_1	S_2	S_3	S_4	S_5	S_6
Početnosť	0,3	0,1	0,2	0,05	0,1	0,25

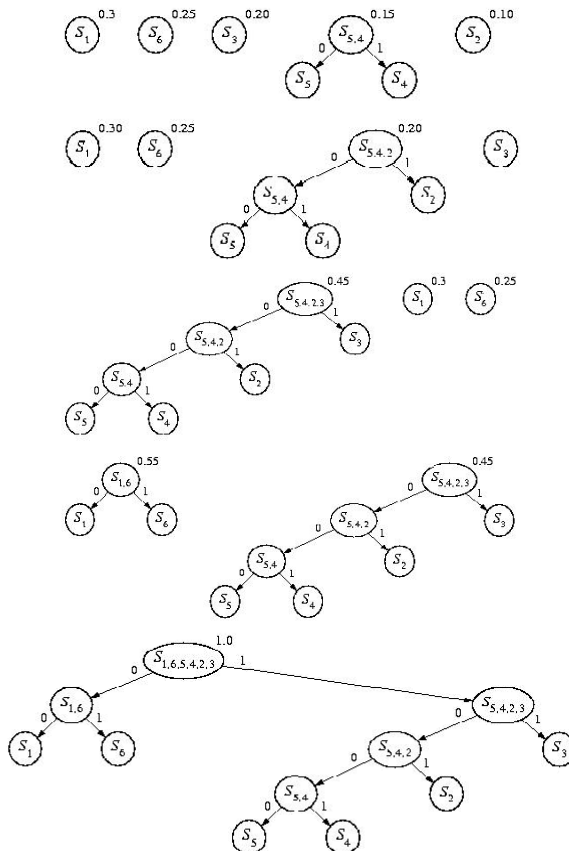
2. Zostupne zoradíme symboly zdrojovej abecedy podľa ich početnosti.



3. Spojíme posledné dva symboly:
   - Vytvoríme nový symbol, ktorého početnosť bude súčtom početností dvoch posledných symbolov.
   - Binárne im priradíme 0 a 1.



4. Kroky 2 a 3 opakujeme, kým sa nedopracujeme k jednému znaku, ktorý má početnosť 1 (koreň vzniknutého binárneho stromu).



5. Traverzovaním vzniknutého stromu priradíme jednotlivým symbolom kódové slová (napr. S_4 je 1001).

Konštrukcia stromu je nejednoznačná, teda výstupom môže byť viac kódov. Toto nastáva vtedy, keď pri spájaní znakov s najmenšími pravdepodobnosťami máme na výber z viacerých možností. Aj napriek tomuto sú Huffmanove kódy najkratšie a majú rovnakú priemernú dĺžku[3]. Kódy sú najkratšie v tom zmysle, že majú minimálnu strednú dĺžku kódového slova. Z teoretického hľadiska je týmto Huffmanovo kódovanie optimálne. Dekompresia dekóduje reťazce premenlivej dĺžky čím sa radí medzi tzv. VLC (variable-length code) kódovania.

Aj keď ide o algoritmus bezstratovej kompresie, paradoxne sa využíva aj pri stratovej kompresii, konkrétne v poslednej fáze kompresie JPEG a takisto pri kompresii videa (MPEG) a zvuku (MP3, Ogg/Vorbis, WMA, ACC).

Shannon-Fanové kódovanie

Je to štatistická metóda bezstratovej kompresie pomenované po Claudovi Shannonovi a Robertovi Fanovi navrhnutá v roku 1949. Od Huffmanovho kódovania sa líši len konštrukciou binárneho stromu:

Množina znakov sa vždy rekurzívne delí na polovice tak, aby bol súčet početností jednotlivých znakov v obidvoch podmnožinách približne rovnaký. Binárne sa potom jednej podmnožine priradí 1, druhej 0.

Tento strom sa tvorí od koreňa smerom k listom a na rozdiel od Huffmanovho kódovania nedosahuje najmenšiu možnú dĺžku kódov. Na druhú stranu garantuje, že dĺžku kódu bude vrámci jedného bitu od teoreticky ideálnej dĺžky $-\log P(x)$.

Kvôli menšej optimálnosti sa používa menej, než Huffmanovo kódovanie, no i napriek tomu sa využíva pri kompresii dát vo formáte ZIP.

Aritmetické kódovanie

Kódovanie nepracuje na princípe nahradzovania vstupného znaku špecifickým kódom, ale kódovaný vstupný tok znakov nahradí jedným reálnym číslom z intervalu $\langle 0; 1)$. Čím je správa dlhšia, tým viac sa zmenšuje interval, z ktorého sa toto číslo vyberá. S tým narastá počet bitov potrebných na presné určenie intervalu.

Symboly správy redukujú veľkosť intervalu podľa ich pravdepodobnostných výskytov vygenerovaných modelom (ten môže byť ľubovoľný; jednoduchý statický, ale aj zložitejší adaptívny). Častejšie symboly redukujú rozsah viac, ako menej časté symboly a preto pridávajú menej bitov do správy.[6]

Pri tomto algoritme nie je potrebné mať symboly vstupnej abecedy zoradené podľa početností. Na začiatku je interval $\langle 0; 1)$ rozdelený na n podintervalov podľa početnosti (n je počet znakov vstupnej abecedy). Vstupom do každého kroku programu je teda interval, ktorý delíme (na začiatku $\langle 0; 1)$), znak, ktorý chceme zakódovať a pravdepodobnostný model pre jednotlivé znaky (pri adaptívnych modeloch sa môže zmeniť počas priebehu algoritmu, v statickom používame početnosti).

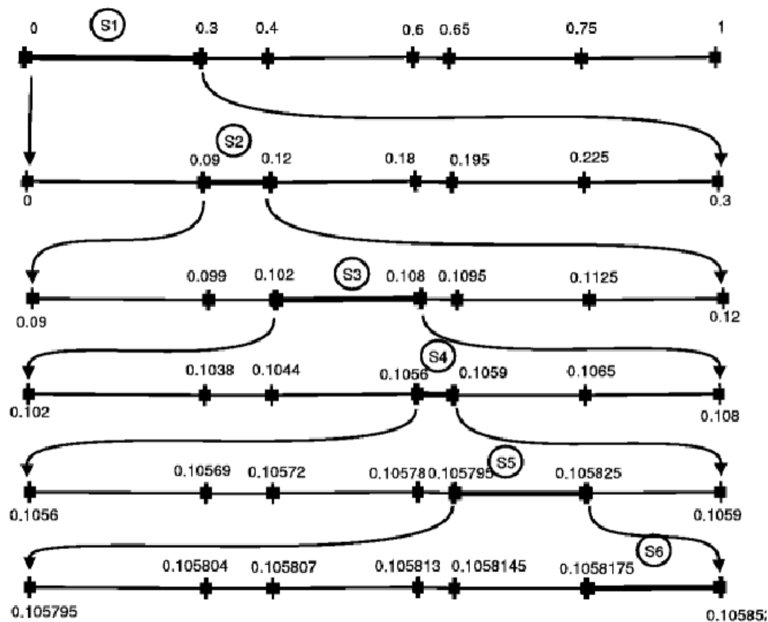
Kódovanie vždy rozdelí aktuálny interval na podintervaly, z nich každý je zlomkom aktuálneho intervalu s veľkosťou úmernou pravdepodobnosti kódovaného znaku. Vzniknutý interval sa stáva intervalom použitým v ďalšom kroku.

Ukážka:
Algoritmus si ukážeme na príklade vstupnej abecedy so šiestimi znakmi $\{S_1, S_2, S_3, S_4, S_5, S_6\}$ a reťazci $S_1S_2S_3S_4S_5S_6$. Početnosti a príslušné intervaly vidíte v tabuľke:

Zdrojový symbol	S_1	S_2	S_3	S_4	S_5	S_6
Početnosť	0.3	0.1	0.2	0.05	0.1	0.25
Pridelený interval	$\langle 0; 0.3)$	$\langle 0.3; 0.4)$	$\langle 0.4; 0.6)$	$\langle 0.6; 0.65)$	$\langle 0.65; 0.75)$	$\langle 0.75; 1)$

Na nasledujúcom obrázku[2] (na ďalšej strane) vidíte, ako sa vybraný interval postupne delí a zmenšuje, až nám vznikne interval pre reťazec $S_1S_2S_3S_4S_5S_6$.

Po poslednom delení nám ostane podinterval $\langle 0.1058175; 0.105825)$. Tento interval reprezentuje doposiaľ zakódované dáta. Kódové slovo, ktoré bude reprezentovať tieto dáta, bude reálne číslo z toho intervalu. Najvhodnejším kandidátom je číslo, ktorého zápis v binárnej podobe má najmenej znakov.



Proces dekódovania prebieha takto:

Vezmime si ako reprezentáciu intervalu spodnú hodnotu 0,1058175. Táto patrí do intervalu $\langle 0; 0.3)$, teda prvý dekódovaný znak je S_1. Teraz potrebujeme znak S_1 odstrániť zo zakódovanej hodnoty. Toto robíme tak, že od zakódovanej hodnoty odčítame krajnú hodnotu intervalu pre tento znak. (Máme $0.1058175 - 0 = 0.1058175$.) Takto získanú hodnotu predelíme veľkosťou intervalu pre tento znak ($0.1058175/0.3{=}0.352725$). Toto zodpovedá znaku S_2 $\left(0.352725 \in \langle 0.3; 0.4)\right)$ atď.

Tento postup opakujeme pre všetky znaky; na konci máme postupnosť $S_1S_2S_3S_4S_5S_6$. Postupné rátanie hodnôt je ľahkým matematickým cvičením pre čitateľa.

Ak má teda prijímateľ zakódovanú hodnotu a pravdepodobnostný model, vie správu dekódovať.

Aritmetické kódovanie je veľmi podobné Huffmanovmu, ale keďže namiesto kódovania symbolov znak po znaku uloží celú správu ako číslo, tak sa niekedy priblíži k optimálnemu (entropickému) kódovaniu viac, než dokáže Huffman. Oproti Huffmanovmu kódovaniu potom dáva aritmetické kódovanie priemerne o 5-10 % lepší kompresný pomer, avšak za cenu náročných aritmetických operácií s reálnymi číslami. Kódovanie je náročné na pamäť aj výkon procesora. Nielen preto, ale takisto aj pre problémy s patentami, sa v praxi oveľa častejšie využíva práve Huffmanovo kódovanie.

Slovníkové algoritmy

Na rozdiel od predchádzajúcich algoritmov tieto algoritmy nevyužívajú štatistický model početností, ale prijatý symbol alebo reťazec symbolov je reprezentovaný indexom v slovníku. Slovníkom nazveme nejakú indexovanú množinu objektov. Množstvo slovníkov používame v bežnom živote bez toho, aby sme si to uvedomovali. Napríklad slovo október je reprezentované v slovníku mesiacov číslom 10.

Pri týchto algoritmoch je nesmierne dôležitá tvorba slovníka. Slovníky rozdeľujeme do dvoch základných druhov.

Statický kódový slovník je slovník, ktorý sa vytvorí ešte pred procesom kopmresie (dekompresie). Tento slovník je verejne známy a používa sa pri kompresii aj dekompresii. Tieto slovníky su však málo flexibilné, tým pádom menej efektívne, a preto sa používa len pri špeciálnych aplikáciách a nie sú vhodné pre všeobecné použitie.

Druhý typ slovníka je **dynamický kódový slovník**. Pri ňom sa slovník vytvára počas priebehu kompresie a dekompresie. Všetky práce založené na dynamickej (adaptívnej) tvorbe slovníka sa dajú rozdeliť do práce dvoch ľudí: Lempl a Ziv (1977, 1978).

LZ77

Prvým zo slovníkových algoritmov je LZ77. Pri tomto algoritme sa ako slovník používa časť vstupného textu, ktorá bola nedávno zakódovaná. Kódovaný text sa porovnáva so symbolmi v slovníku. Najdlhšia nájdená zhoda sa uloží ako pointer, ktorý obsahuje trojicu dát (takzvaný triplet).

Algoritmus v LZ77 sa nazýva aj algoritmus posúvania okien. Okno pozostáva z dvoch častí - prvou je `search buffer` (*SB*) a druhou `look-ahead buffer` (*LAB*). V *SB* je časť textu, ktorá bola už nedávno zakódovaná (to, čo sme nedávno prečítali; pre nás je to slovník), *LAB* obsahuje text, ktorý ideme v nasledujúcom kroku algoritmu komprimovať. Počas priebeh algoritmu sa tieto dve okná posúvajú.

Kompresiu si ukážeme na texte "accbadaccbaccbaccgikmcab". *SB* má veľkosť 9, *LAB* 6 symbolov. Prvým krokom je nájdenie prvého znaku z *LAB* v *SB* – *b*. V *SB* hľadáme smerom odzadu. Prvú zhodu nájdeme na šiestom políčku (odzadu). Ďalej zistíme, že najdlhšia zhoda v reťazcoch má dĺžku 2 ("*ba*").

Triplet potom obsahuje tri hodnoty $[i, j, k]$:

- i je vzdialenosť prvého symbolu z *LAB* od konca *SB*, v našom prípade teda 6. Táto vzdialenosť sa nazýva ofset. Ak by sa takýto symbol v *LAB* nenachádzal, tak je táto vzdialenosť 0.
- j je dĺžka najdlhšieho zhodného podreťazca. Program prehľadáva reťazce pre rôzne ofsety (začiatky) a vyberie najdlhšiu. Zaujímavé je, že zhoda môže presiahnuť hranice *SB* a tým presiahnuť aj do *LAB*. Toto uvidíme v druhom kroku.
- k je prvý znak za najdlhším spoločným podreťazcom **v *LAB***. Pre nás je to najprv *c*, lebo v *LAB* máme *baccba* a najdlhší podreťazec je *ba*. Za prvým výskytom reťazca *ba* v *LAB* sa nachádza práve *c*.

V každom cykle sa algoritmus posunie o $j + 1$ znakov, až kým nie je zakódovaný celý text a algoritmus končí. Priebeh pre náš príklad vidíte na ďalšom obrázku.

```
|a c c b a d a c c|b a c c b a|c c g i k m c a b
 a c c|b a d a c c b a c|c b a c c g|i k m c a b
 a c c b a d a c c|b a c c b a c c g|i k m c a b|
```

Z prvého stavu máme triplet $[6, 2, c]$ Tak, ako sme popísali, sa okná po prvom kroku posunú o 3 (2+1) znaky doprava. Tam sa prvá zhoda nájde na ofsete 1 s dĺžkou 1 (c). Ďalšia zhoda je na ofsete 4 s dĺžkou 5 (*cbacc*). Práve tu vidíme, že nám zhoda presahuje mimo *SB* do *LAB*. Po reťazci *cbacc* máme v *LAB* znak *g*. Výsledný triplet je teda $[4, 5, g]$. Ďalej sa okná posunú o 6 znakov. Tu je nájdená nulová zhoda, preto je triplet $[0, 0, i]$.

Dekompresia je oveľa jednoduchšia, pretože pri nej nie je potrebné porovnávať dĺžky ani hľadať zhody. Čitateľ si môže dekompresiu vyskúšať na tom istom príklade; na vstupe potrebuje počiatočný slovník (teda obsah *SB* na začiatku) a triplety z priebehu algoritmu. Počas dekompresie sa naspäť vylúšti celý vstupný text.

Tentokrát vidíme, že pomocou deviatich znakov (ako počiatočný slovník) a niekoľkých tripletov (alebo, ak chcete, trojrozmerných vektorov) dokážeme dekódovať pomerne dlhý text.

V praxi sa najviac algoritmus využíva spolu s Huffmanovým kódovaním v algoritme nazvanom `DEFLATE`.

LZ78

Tento algoritmus takisto používa zakódovaný text ako slovník, ale trocha iným spôsobom. Veľkosť slovníka pri tomto algoritme môže byť teoreticky neobmedzená, pretože v každom kroku sa jeho veľkosť zväčší. Maximálny výkon by teda teoreticky dosiahol vtedy, kedy by bol kódovaný text nekonečný. V praxi však veľmi veľký slovník vedie k horšej efektívnosti algoritmu. Toto sa rieši tak, že sa po dosiahnutí istej veľkosti slovník prestane zväčšovať, prípadne sa úplne anuluje.

Namiesto trojíc (tripletov), používa algoritmus iba dvojice [index, znak].

Ukážka:

Ukážeme si kódovanie na vstupe *baccbaccacbc*. Pri úplne novom znaku x je výsledná dvojica $[0, x]$ a tento znak sa uloží na nové miesto v slovníku. Inak sa kódovaný znak porovnáva s obsahom slovníka a uloží sa ako $[\text{index}, x]$, kde index je poradie prvku slovníka, ku ktorému prilepujeme nový znak x.

Pre náš príklad sa teda najprv uloží b s dvojicou $[0, b]$, potom a s dvojicou $[0, a]$ a c s dvojicou $[0, c]$. Potom nasleduje znak c, ktorý už poznáme, tak prečítame ďalší znak a vzniknutý reťazec cb uložíme s dvojicou $[3, b]$ (3 je pozícia znaku c). To isté spravíme aj s reťazcami ac ($[2, c]$) a ca ($[3, a]$). Potom prečítame znak c, ktorý poznáme, prečítame ďalší znak b, reťazec cb avšak už tiež poznáme (index 4), tak prečítame ďalší znak c a tento nový reťazec cbc uložíme na nové miesto v slovníku s dvojicou $[4, c]$. Týmto sme uložili celý náš reťazec, ale už poznáme spôsob, ktorým by sme dokázali zakódovať aj oveľa dlhší reťazec.

LZW

Algoritmus LZW patrí do skupiny LZ algoritmov, bol vymyslený v roku 1984 ako zlepšenie algoritmu LZ78 a je pomenovaný podľa tvorcov Lempela, Ziva a Welcha. V tomto algoritme sa oproti LZ78 redukuje druhá zložka dvojice (nasledujúci prvok), a tým sa algoritmus stáva ešte efektívnejší. Proces je rýchly a nenáročný na pamäť.

Výstupom (alebo vstupom pre dekompresiu) je iba postupnosť indexov, ktoré hovoria, na ktorom mieste sa nachádza šifrovaný reťazec. Vďaka tomu sa zmenil aj počiatočný slovník, ktorý obsahuje všetky jednoznakové postupnosti - štandardne 256 znakov ASCII.

Ukážka:

Kódovanie si ukážeme na reťazci *accbadaccbaccba* so zdrojovou abecedou $S = \{a, b, c, d\}$. Tieto symboly sa nachádzajú na prvých 4 miestach v slovníku. Prvá nájdená najdlhšia postupnosť, ktorá sa už nachádza v slovníku, je *a*. Ďalší symbol c je k nemu prilepený a do slovníka sa pridá reťazec *ac* na piatu pozíciu. Takto algoritmus pokračuje ďalej a do slovníka sa postupne pridajú reťazce *cc, cb, ba, ad, da* na pozície 6 až 10. Ďalej algoritmus prečíta znak a, ktorý pozná, ďalej znak c, avšak reťazec *ac* už tiež pozná, a preto prečíta ďalší znak (c) a na pozíciu 11 sa uloží reťazec *acc*. To isté pokračuje s reťazcom *cba* (tentokrát sme sa posunuli o 2 znaky), potom nasleduje reťazec *acc*, ktorý už poznáme a na pozíciu 13 sa uloží štvorznakový reťazec *accb*. Na konci ostal ešte reťazec *ba*, ktorý poznáme, normálne by sme čítali ďalej, tu ale končíme.

Na nasledujúcej tabuľke vidíte, ako sa postupne zaplnil celý slovník. Pri každom uloženom znaku sa ukladá aj index "otca", t.j. prvku slovníka, ku ktorému sa pridal nový znak. Výstupom programu potom bude sekvencia týchto indexov. V tabuľke sa tieto indexy nachádzajú v treťom stĺpci. Znak $ zodpovedá tomu, že znak je v počiatočnom slovníku (init_dict).

Index	**Reťazec**	**Index "otca"**
1	*a*	$ (init_dict)
2	*b*	$ (init_dict)
3	*c*	$ (init_dict)
4	*d*	$ (init_dict)
5	*ac*	1
6	*cc*	3
7	*cb*	3
8	*ba*	2
9	*ad*	1
10	*da*	4
11	*acc*	5
12	*cba*	7
13	*accb*	11
14	*ba...*	(8)

Konečná výstupná sekvencia je 1 3 3 2 1 4 5 7 11.

Pri dekompresii dostaneme takúto sekvenciu a zdrojovú abecedu (verejne známa). Napríklad pri našej postupnosti by sme najprv číslom dekódovali a, číslom 3 c a teraz už vieme, že na ďalšiu pozíciu (5) musíme pridať reťazec *ac*. Proces dekompresie trvá dovtedy, kým nedekódujeme celú sekvenciu čísel.

V praxi sa používa pri kompresii GIF (grafika) a v rôznych variantoch je algoritmus patentovaný rôznymi firmami.

Aničku a Zuzku sme už naučili množstvo algoritmov, ktorými vedia svoje dáta komprimovať. Vedia kódovať binárnu postupnosť, ale aj slovné reťazce. Avšak všetky algoritmy, ktoré sme ich zatiaľ naučili, komprimujú bezstratovo a v praxi sa nepoužívajú (napríklad) pri komprimácii fotiek. Ostáva nám teda ešte jeden kompresný štandard – JPEG.

4 Stratová kompresia – JPEG alebo Chceme posielať aj fotky

JPEG je štandardná metóda stratovej kompresie, ktorá sa používa pre ukladanie počítačových obrázkov vo fotorealistickej kvalite. Medzinárodným štandardom sa stal v roku 1992. V skutočnosti je formát komprimovaných súborov JFIF, čo znamená *JPEG File Interchange Format.* Akronym JPEG znamená *Joint Photographic Experts Group*, čo je konzorcium, ktoré túto kompresiu navrhlo. JPEG nie je jeden algoritmus, ale súbor komprimačných techník.

JFIF je najčastejší formát používaný na prenášanie a ukladanie fotografií na webe. Podľa článku z roku 2000 bolo viac než 80% všetkých obrázkov prenášaných cez internet sú uložené používajúc štandard JPEG.

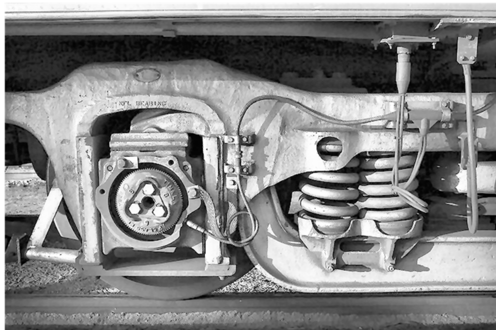
Aj napriek populárnosti tohto formátu členovia konzorcia JPEG po zverejnení odhalili niekoľko problémov s formátom a takisto vytvorili a zverejnili zoznam zlepšení, ktoré by mali byť zahrnuté v ďalšej generácii formátu.

V tomto texte sa pozrieme na základný popis algoritmu, ukážeme si jeho použitie na príklade a takisto preberieme problémy s formátom.

Základný algoritmus

Algoritmus JPEG má množstvo pomocných častí, ktorým sa tu nebudeme venovať, ale bude popísaná hlavná myšlienka algoritmu. Tento rozdelíme na 4 hlavné časti: predpríprava, transformácia, kvantizácia, zakódovanie.

Na vysvetlenie algoritmu si ho budeme ukazovať na obrázku pod týmto textom; konkrétne v rozlíšení 240x160 pixelov (príklad uvedený v [1]).



Predpríprava – Preprocessing

Ak chceme komprimovať farebný obrázok do formátu JPG, tak najprv musíme červené, zelené a modré (RGB) kanály previesť do priestoru YCbCr. YCbCr je chromatické farebné prostredie. Y zodpovedá jasovej zložke; Cb a Cr sú zložky nesúce informáciu o farbe. Väčšina vizuálnych informácií, na ktoré je ľudské oko najviac citlivé, sa vyskytuje v jasových komponentoch (Y); komponenty Cb a Cr obsahujú farebnú informáciu, na ktorú je ľudské oko menej citlivé a väčšina týchto informácií môže byť odstránená ich podvzorkovaním.

To je proces, pri ktorom sa jasovému kanálu (Y) ponecháva plné rozlíšenie no farebné kanály Cb a Cr sa takzvane podvzorkujú, čím sa zmenší ich rozlíšenie na polovicu (alebo štvrtinu). Takéto ušetrenie nie je možné priamo pri farebnom modeli RGB, lebo každá zo zložiek (R, G, B) obsahuje nejakú jasovú informáciu a zásah do nich je hneď viditeľný.

RGB sa do YCbCr prevádza priamo, pre 8-bitové RGB sa to robí pomocou týchto vzorcov. Výsledkom sú hodnoty v rozmedzí 0-255.

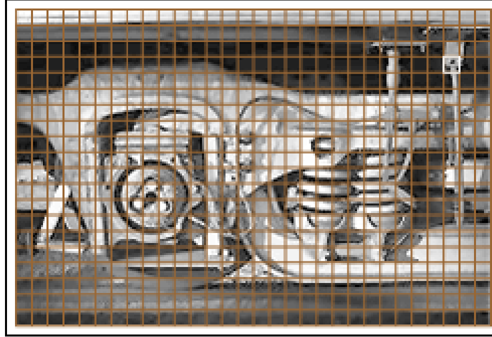
$$\mathrm{Y} = 0,299\ \mathrm{R} + 0,587\ \mathrm{G} + 0,114\ \mathrm{B}$$

$$\mathrm{Cb} = -0,1687\ \mathrm{R} - 0,3319\ \mathrm{G} + 0,5\ \mathrm{B} + 128$$

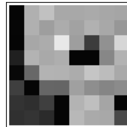
$$\mathrm{Cr} = 0,5\ \mathrm{R} - 0,4187\ \mathrm{G} - 0,0813\ \mathrm{B} + 128$$

V ďalšom kroku sa komprimovaný obrázok rozdelí na bloky s veľkosťou 8x8 pixelov. Ďalšie opatrenia sú potrebné, ak rozmery obrázku nie sú deliteľné číslom 8, ale pre ukážku algoritmu sme si vzali obrázok s

rozmermi 240x160 pixelov, takže nemáme problém s delením na bloky. Týmto krokom tak rozdelíme obrázok na $\frac{240}{8} \cdot \frac{160}{8} = 600$ blokov. Na nasledujúcom obrázku vidíte pôvodný obrázok rozdelený na bloky.



Všimnite si, že na obrázku je vyznačený blok v 4. riadku a v 28. stĺpci. Tento použijeme na znázornenie ďalších krokov algoritmu. Na ďalšom obrázku vidíte tento blok zväčšený. Pod ním sa nachádza matica 8x8, v ktorej sú zapísané intenzity jednotlivých pixelov tohto bloku.
(Vždy, keď budeme potrebovať pri popise algoritmov zapísať pole s intenzitami, tak použijeme maticu.)



$$\begin{pmatrix} 5 & 176 & 193 & 168 & 168 & 170 & 167 & 165 \\ 6 & 176 & 158 & 172 & 162 & 177 & 168 & 151 \\ 5 & 167 & 172 & 232 & 158 & 61 & 145 & 214 \\ 33 & 179 & 169 & 174 & 5 & 5 & 135 & 178 \\ 8 & 104 & 180 & 178 & 172 & 197 & 188 & 169 \\ 63 & 5 & 102 & 101 & 160 & 142 & 133 & 139 \\ 51 & 47 & 63 & 5 & 180 & 191 & 165 & 5 \\ 49 & 53 & 43 & 5 & 184 & 170 & 168 & 74 \end{pmatrix}$$

Posledným krokom procesu predprípravy obrázku je, že sa od každej hodnoty v matici odčíta 127. Tento krok vycentruje intenzity okolo nuly (predtým boli hodnoty v rozmedzí 0-255), čo zjednodušuje proces transformácie a kvantizácie. Pre náš ukážkový blok nám tak vznikne takáto matica:

$$A = \begin{pmatrix} -122 & 49 & 66 & 41 & 41 & 43 & 40 & 38 \\ -121 & 49 & 31 & 45 & 35 & 50 & 41 & 24 \\ -122 & 40 & 45 & 105 & 31 & -66 & 18 & 87 \\ -94 & 52 & 42 & 47 & -122 & -122 & 8 & 51 \\ -119 & -23 & 53 & 51 & 45 & 70 & 61 & 42 \\ -64 & -122 & -25 & -26 & 33 & 15 & 6 & 12 \\ -76 & -80 & -64 & -122 & 53 & 64 & 38 & -122 \\ -78 & -74 & -84 & -122 & 57 & 43 & 41 & -53 \end{pmatrix}$$

Transformácia

Kľúčovou zložkou kompresného štandardu JPEG je krok transformácie. Cieľom je transformovať predpripravený obrázok do nastavenia, kde kódujúca časť kompresného algoritmu dokáže byť efektívnejšia. Metóda kódovania funguje najlepšie, ak je tam relatívne málo rozdielnych hodnôt.

Ako to dosiahneme pomocou transformácie? Pozrime sa na to takto – digitálne obrázky väčšinou pozostávajú z oblastí, v ktorých je zmena v intenzite čiernobielej škály malá. Potrebujeme transformáciu, ktorá tento fakt využije – hľadáme transformáciu, ktorá vezme blok $m \times n$ pozostávajúci z podobných hodnôt a premení ho na maticu rovnakej veľkosti, kde väčšina informácií o pôvodnom bloku je uschovaných v relatívne málo prvkoch a ostatné prvky sú nulové alebo veľmi blízke nule. Ak každá oblasť podobných hodnôt je zmapovaná do hodnôt blízkych nule, tak výstup transformácie bude pozostávať z veľkého počtu takmer nulových hodnôt.

Ako transformácia pri štandarde JPEG bola vybraná diskrétna kosínusová transformácia (DCT).

Diskrétna kosínusová transformácia (DCT) je typ diskrétnej Fourierovej transformácie (DFT), čo je transformácia, ktorá vyjadrí diskrétny signál (teda napríklad postupnosť čísel) na zloženie sínusoíd (funkcie sínus a kosínus) s rôznymi frekvenciami a amplitúdami. Na rozdiel od DFT používa DCT iba funkciu kosínus. Existuje 8 typov DCT (označené I-VIII), pričom najviac sa používajú prvé štyri. Vôbec najpoužívanejšou je transformácia DCT-II, ktorá sa práve využíva aj pri kompresii obrázkov. (Vďaka svojej používanosti sa jednoducho v anglických textoch označuje aj ako *'the DCT'*).

Ak teda uvažujeme bloky dát určitej veľkosti, z praktických dôvodov ich môžeme považovať za vektory. V DCT potom vektory vyjadrené v "kanonickej báze" vyjadrujeme v báze tvorenej sínusoidami rozličných frekvencií.

V algoritme JPEG teda prevedieme každý blok 8×8 na maticu frekvencií. To spravíme tak, že na blok 8×8 aplikujeme dvojdimenzionálnu (2D) DCT:

$$D_{i,j} = \frac{1}{4} C(i)C(j) \sum_{x=0}^{7} \sum_{y=0}^{7} p_{(x,y)} \cos\left[\frac{(2x+1)i\pi}{16}\right] \cos\left[\frac{(2y+1)j\pi}{16}\right],$$

kde $D_{i,j}$ je prvok na mieste (i,j) v transformovanej matici a $p_{(x,y)}$ je prvok na mieste x, y v bloku reprezentujúcom obrázok. [7]

2D-DCT môžeme reprezentovať aj pomocou matice, táto vyzerá takto:

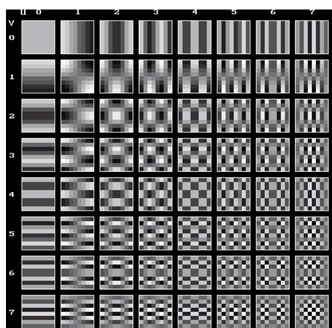
$$U = \frac{1}{2}\begin{pmatrix}
\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\
\cos\frac{\pi}{16} & \cos\frac{3\pi}{16} & \cos\frac{5\pi}{16} & \cos\frac{7\pi}{16} & \cos\frac{9\pi}{16} & \cos\frac{11\pi}{16} & \cos\frac{13\pi}{16} & \cos\frac{15\pi}{16} \\
\cos\frac{2\pi}{16} & \cos\frac{6\pi}{16} & \cos\frac{10\pi}{16} & \cos\frac{14\pi}{16} & \cos\frac{18\pi}{16} & \cos\frac{22\pi}{16} & \cos\frac{26\pi}{16} & \cos\frac{30\pi}{16} \\
\cos\frac{3\pi}{16} & \cos\frac{9\pi}{16} & \cos\frac{15\pi}{16} & \cos\frac{21\pi}{16} & \cos\frac{27\pi}{16} & \cos\frac{33\pi}{16} & \cos\frac{39\pi}{16} & \cos\frac{45\pi}{16} \\
\cos\frac{4\pi}{16} & \cos\frac{12\pi}{16} & \cos\frac{20\pi}{16} & \cos\frac{28\pi}{16} & \cos\frac{36\pi}{16} & \cos\frac{44\pi}{16} & \cos\frac{52\pi}{16} & \cos\frac{60\pi}{16} \\
\cos\frac{5\pi}{16} & \cos\frac{15\pi}{16} & \cos\frac{25\pi}{16} & \cos\frac{35\pi}{16} & \cos\frac{45\pi}{16} & \cos\frac{55\pi}{16} & \cos\frac{65\pi}{16} & \cos\frac{75\pi}{16} \\
\cos\frac{6\pi}{16} & \cos\frac{18\pi}{16} & \cos\frac{30\pi}{16} & \cos\frac{42\pi}{16} & \cos\frac{54\pi}{16} & \cos\frac{66\pi}{16} & \cos\frac{78\pi}{16} & \cos\frac{90\pi}{16} \\
\cos\frac{7\pi}{16} & \cos\frac{21\pi}{16} & \cos\frac{35\pi}{16} & \cos\frac{49\pi}{16} & \cos\frac{63\pi}{16} & \cos\frac{77\pi}{16} & \cos\frac{91\pi}{16} & \cos\frac{105\pi}{16}
\end{pmatrix}$$

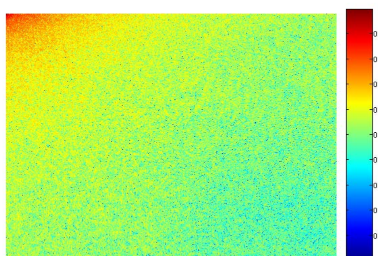
Proces 2D-DCT na blok A rozmeru 8×8 (z fotky) je potom $B = UAU^T$. Stĺpce matice U tvoria ortonormálnu postupnosť, matica U je teda ortogonálna. To je super, pretože vieme ľahko určiť súradnice vektoru v novej báze (a takisto vieme ľahko vytvoriť inverz matice U).

Z hľadiska lineárnej algebry je teda U **matica prechodu** z pôvodnej bázy do novej bázy.

DCT transformuje blok 8×8 vstupných hodnôt (intenzít pixelov) na **lineárnu kombináciu** 64 vzorov, ktoré sa nachádzajú na obrázku vľavo dole[8].

Tieto vzory označujeme ako *prvky báze dvojdimenzionálnej DCT*. Výstupné hodnoty transformácie (teda prvky matice B, ktorá vznikla transformovaním matice A) nazývame *koeficienty lineárnej kombinácie* (alebo *transformácie*). Pôvodný blok 8×8 by sme teda dostali zložením jednotlivých vzorov, pričom vzor na mieste (u, v) by sme vynásobili koeficientom na mieste (u, v) v matici B ($\forall u, v \in \{0, \dots, 7\}$).





Pre väčšinu obrázkov platí, že väčšina informácie je umiestnená na nižších frekvenciách, čomu zodpovedá ľavý horný roh – pozícia $(0, 0)$. Čím viac smerom doprava a nadol sa posúvame, tým tieto pozície prislúchajú väčším frekvenciám. Inak povedané, keďže obrázky väčšinou pozostávajú z rovnako-farebných plôch, tak najväčší koeficient lineárnej kombinácie budeme mať pri vzore na mieste $(0, 0)$ z obrázku vľavo. DCT má teda tendenciu sústrediť všetku informáciu v ľavom hornom rohu. (Teraz nám je asi jasné, prečo nie je JPEG vhodný pre komprimáciu obrázkov obsahujúcich ostré hrany). Na obrázku vpravo[4] vidíte, ako DCT sústreďuje informáciu (červená farba – veľa uloženej informácie). Je vidieť, že najviac informácie sa nachádza v ľavom hornom rohu.

Operácia DCT je invertovateľná. Dá sa dokázať, že matica U je ortogonálna (teda $U^T = U^{-1}$). Ortogonalita matice U je dôležitá pri dekódovaní (inverznej operácii), pretože $U^{-1} = U^T$, nemáme preto problém určiť inverz.

Ak $\mathbf{a}$ nazveme jeden riadok matice A, potom na výpočet $U\mathbf{a}$ potrebujeme 8 násobení a 7 sčítaní; pre všetky riadky spolu 120 operácií. Existujú avšak aj rýchle algoritmy, založené na rýchlej Fourierovej transformácii, ktoré dokážu signifikantne znížiť čas potrebný na výpočet $U\mathbf{a}$.

Teda máme algoritmus, ktorý je invertovateľný, na počítanie vrámci neho máme rýchle spôsoby a výstup pre takmer konštantné matice je matica s veľkým množstvom takmer nulových hodnôt. Pre štandard JPEG je to teda takmer dokonalý algoritmus.

Algoritmus DCT má však aj jednu výraznu nevýhodu. Zatiaľ čo vstupom v bloku 8×8 sú celé čísla, výstupom sú typicky čísla reálne. Preto ešte potrebujeme krok kvantizácie, aby sme vytvorili výstup, ktorý obsahuje len celé čísla.

Na nasledujúcich dvoch obrázkoch vidíte použitie DCT na náš obrázok; najprv na vybraný pixel a potom na celý obrázok. Na ňom vidieť jasovú informáciu sústredenú v istých bodoch - tieto zodpovedajú ľavým horným rohom mriežok 8×8.

Kvantizácia

Keďže prvky matice U z DCT sú iracionálne čísla a vstupom z obrázku sú celé čísla, tak nám proces DCT nám vytvára reálne čísla. Keďže kódovanie dát funguje najlepšie, keď pracuje s celými číslami, tak musíme do algoritmu JPEG pridať krok kvantizácie.

Budeme pracovať s maticou z obrázka, ktorý sme používali. Túto upravenú maticu sme nazvali A a v texte sa nachádza na strane 10. Po použití DCT z maticou U (strana 9) dostaneme takúto maticu $B = UAU^T$ (čísla zaokrúhlené na 3 desatinné miesta):

$$B = \begin{pmatrix} -27,500 & -213,468 & -149,608 & -95,281 & -103,750 & -46,946 & -58,717 & 27,226 \\ 168,229 & 51,611 & -21,544 & -239,520 & -8,238 & -24,495 & -52,657 & -96,621 \\ -27,198 & -31,236 & -32,278 & 173,389 & -51,141 & -56,942 & 4,002 & 49,143 \\ 30,184 & -43,070 & -50,473 & 67,134 & -14,115 & 11,139 & 71,010 & 18,039 \\ 19,500 & 8,460 & 33,589 & -53,113 & -36,750 & 2,918 & -5,795 & -18,387 \\ -70,593 & 66,878 & 47,441 & -32,614 & -8,195 & 18,132 & -22,994 & 6,631 \\ 12,078 & -19,127 & 6,252 & -55,157 & 85,586 & -0,603 & 8,028 & 11,212 \\ 71,152 & -38,373 & -75,924 & 29,294 & -16,451 & -23,436 & -4,213 & 15,624 \end{pmatrix}$$

Na kvantizovanie takého bloku algoritmus JPEG najprv predelí každý bod predpísanou hodnotou a potom výsledky zaokrúhli, čiže vytvoríme celé čísla. Čísla, ktorými delíme jednotlivé prvky matice B sú zapísané v tejto matici (vždy delíme prvkom na tej istej pozícii v matici Z). Tejto matici sa hovorí aj kvantizačná tabuľka.

$$Z = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

Vytvoríme novú maticu Q, ktorej prvky sú $q_{jk} = \mathrm{Round}(b_{jk}/z_{jk})_{\ j,k=1,\dots,8}$, kde Round je zaokrúhľujúca funkcia, teda vráti zaokrúhlené číslo, ktoré dostalo na vstupe. Môžeme si všimnúť, že najväčšie hodnoty sú v tabuľke Z vpravo dole, teda keď nimi delíme čísla z pravého dolného rohu matice B, tak sa stanú malými a proces zaokrúhľovania z nich spraví 0.

Pre náš príklad vznikne takáto matica Q:

$$Q = \begin{pmatrix} -2 & -19 & -15 & -6 & -4 & -1 & -1 & 0 \\ 14 & 4 & -2 & -13 & 0 & 0 & -1 & -2 \\ -2 & -2 & -2 & 7 & -1 & -1 & 0 & 1 \\ 2 & -3 & -2 & 2 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & -1 & -1 & 0 & 0 & 0 \\ -3 & 2 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Všimnite si, že DCT aj delenie koeficientami sú návratné operácie, no krok zaokrúhľovania sa nedá vrátiť späť. Preto je JPEG stratová kompresia. Teraz sme už nenávratne stratili možnosť získať pôvodný obrázok. Každý blok obrázku sa takto kvantizuje a vzniknuté celé čísla sa pošlú do kódovacej časti algoritmu. Vo všeobecnosti sme však z veľkého množstva čísel spravili 0, čiže kódovanie bude efektívnejšie.

Na ďalšom obrázku vidíte náš blok aj celý obrázok po použití DCT aj kvantizácie:

Pozrieme sa ešte na proces dekódovania takto zakódovaných údajov; najprv vytvoríme maticu B', kde prvky našej matice Q vynásobíme prvkami z kvantizačnej tabuľky. Na takto upravenú maticu použijeme inverzný proces z DCT: vypočítame $A' = U^T B' U$. Maticu A' uvádzam nižšie. Môžete si všimnúť vzniknutú stratu oproti pôvodnej matici A.

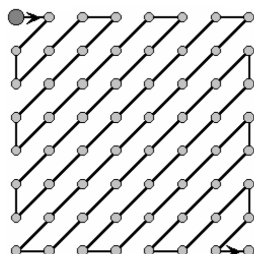
$$A' = \begin{pmatrix} -128 & 45 & 71 & 63 & 22 & 32 & 22 & 52 \\ -110 & 39 & 4 & 48 & 41 & 68 & 65 & 13 \\ -115 & 40 & 50 & 152 & 13 & -88 & -4 & 76 \\ -105 & 51 & 43 & 18 & -126 & -99 & 19 & 60 \\ -116 & -24 & 56 & 63 & 33 & 80 & 62 & 28 \\ -67 & -118 & -47 & -24 & 30 & 17 & -12 & 29 \\ -67 & -79 & -60 & -116 & 49 & 69 & 12 & -108 \\ -78 & -69 & -80 & -138 & 63 & 41 & 49 & -67 \end{pmatrix}$$

Poznámka:
Uvedená kódovacia tabuľka Z sa používa pre kódovanie (jasového) kanálu Y. Na kódovanie signálov Cb a Cr sa zvykne používať takáto kódovacia tabuľka (farebné dáta sa kvantujú "viac" než jasové):

$$Z' = \begin{pmatrix} 17 & 18 & 24 & 47 & 66 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 69 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix}$$

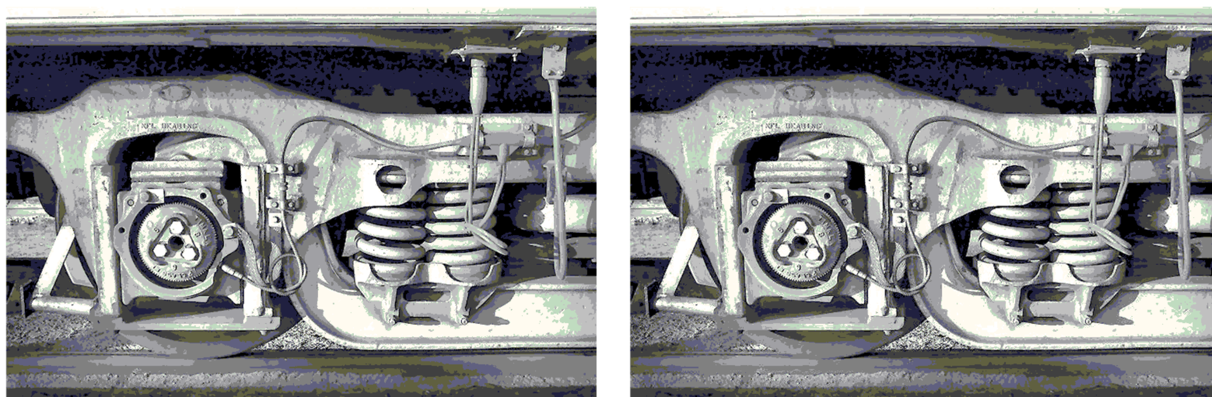
Kódovanie

Posledným krokom v procese JPEG je zakódovať transformovaný a kvantizovaný obrázok. Štandard JPEG používa rozšírené Huffmanovo kódovanie, ktoré už poznáme (ale existuje aj modifikácia, pri ktorej sa používa aritmetické kódovanie). Pred samotným kódovaním je pole 8×8 koeficientov po DCT a kvantizácii preorganizované do postupnosti "cik-cak" (vidíte ho na obrázku pod týmto odsekom). Toto preorganizovanie spôsobí, že prvky matice obsahujúce najviac informácií sa dostanú na začiatok cik-cakovej postupnosti. Ostatné (väčšinou) nulové koeficienty vytvárajú postupnosť za sebou idúcich núl. Použité (entropické) kódovanie je bezstratové a slúži len na na zníženie redundancie vstupných dát po kvantizácii a uľahčí proces kódovania kódom s premenlivou dĺžkou.



Invertovanie procesu je viacmenej priamočiare. Najprv sa dekóduje postupnosť zakódovaná Huffmanovým kódovaním aby sa získal obrázok po kvantizácii a použití DCT. Ďalej sa použije postup popísaný v časti o kvantizácii. Nakoniec sa ešte vráti naspäť proces predpríprav tak, že sa ku každej hodnote v bloku pričíta 127. Vzniknutý obrázok je aproximácia pôvodného obrázku.

Pôvodný obrázok mal rozmery 160×240, takže na uloženie na disku bolo potrebných 160*240*8=307 200 bitov. Po aplikovaní Huffmanovho kódovania na transformovaný a kvantizovaný obrázok potrebujeme iba 85 143 bitov. To predstavuje ušetrenie viac než 70% pôvodných údajov! (Faktor kompresie je približne 0,277.) Na obrázku vidíte porovnanie pôvodného a nového obrázku; napriek veľkému množstvu ušetrených údajov nevidíme takmer žiaden rozdiel.



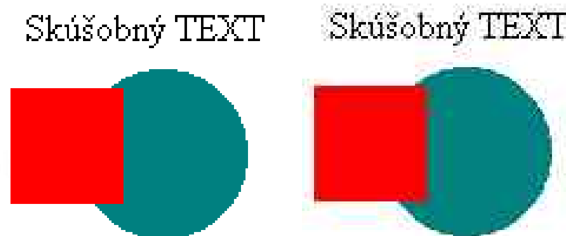
vľavo pôvodný, vpravo komprimovaný obrázok

Problémy s formátom

Štandard JPEG je veľmi efektívna metóda na komprimáciu digitálnych obrázkov, no aj napriek tomu spôsobuje niekoľko problémov. Jeden problém spôsobuje delenie obrázku na bloky 8×8 – niekedy sa výsledný obrázok zdá rozštvorčekovaný – vidieť pozostatky po delení (procesom DCT sa z pixelov vedľa seba stanú pixely s rozdielnymi farbami).

Proces kvantizácie robí z algoritmov štandardu JPEG stratovú kompresiu. Pri niektorých použitiach, ako prehliadanie webu, je prijateľné. Pri aplikáciách ako fotenie vo vysokom rozlíšení pre časopisy je strata rozlíšenia nežiadúca.

Algoritmus JPEG nie je vhodný pre kompresiu grafiky obsahujúcej ostré hrany, pretože ich "rozmaže". Na ďalšom obrázku vidíme obrázok po nesprávnom použití kompresie; okolo textu a geometrického tvaru je vidieť šum.



Vlnenie vychádzajúce z miest pôvodných hrán, ktoré je na obrázku vidieť, sa nazýva Gibbsov jav.

Na odstránení problémov sa pracovalo a bol vymyslený štandard JPEG2000, ktorý celý proces vylepšil. Tento dokáže komprimovať väčšie obrázky, viac kanálov naraz (až do 256, potrebné napríklad pri satelitných snímkoch), dokáže rozpoznať oblasti záujmu (najdôležitejšie prvky obrázku) a komprimuje ich tak, aby mali vo výslednom obrázku väčšie rozlíšenie. Takisto nevytvára nutne bloky 8×8, ale logickejšie rozdlaždičkuje obrázok na neprelínajúce sa časti, ktoré sa spracujú podobne ako pri štandarde JPEG. Celkovo sa mu darí vytvárať kvalitnejšie obrázky s menším počtom bitov na pixel, kompresia je teda účinnejšia.

5 Zhrnutie alebo Máme hotovo

Aničke a Zuzke sme ponúkli veľké množstvo postupov, ako môžu posielané veci skomprimovať, a tým šetriť mobilné dáta. Vedia postupy aplikovať na rôzne vstupy a takisto aj podľa typu vstupu vybrať vhodný algoritmus. Vedia zašifrovať text, binárnu postupnosť a nakoniec sme im ukázali ako komprimovať obrázok, aj keď s nejakými stratami. Oboznámili sa aj s rôznymi typmi algoritmov a teda sa aj niečo naučili.

Dúfam, že sme sa niečo naučili aj my. Článok poskytol hlavnú myšlienok a popis základných algoritmov na komprimáciu dát. Pre názornosť sa takisto pri väčšine algoritmov nachádzala aj ukážka ich použitia na príklade.

6 Zoznam použitých zdrojov

[1] *Why Do Math?* – http://www.whydomath.org/node/wavlets/basicjpg.html

[2] HUDEC, Peter: Interaktívna výuka kompresie dát [diplomová práca] Univerzita Komenského, Fakulta matematiky, fyziky a informatiky. Rok obhajoby: 2003. http://projects.hudecof.net/diplomovka/online/ucebnica/html/uvod.html

[3] KRAVECOVÁ, Daniela. *Základy kódovania.* Košice: Technická univerzita v Košiciach, 2012.

[4] Diskrétní kosinová transformace – Wikipédia; https://cs.wikipedia.org/wiki/Diskr%C3%A9tn%C3%AD_kosinov%C3%A1_transformace

[5] IVÁNEK, Jiří. *Vybrané kapitoly z kódování informací.* Praha, 2007.

[6] WITTEN Ian, Neal Radford. *Arithmetic coding for data compression.* 1987.

[7] CABEEN Ken, GENT Peter. *Image Compression and the Discrete Cosine Transform.* https://www.math.cuhk.edu.hk/~lmlui/dct.pdf

[8] SOVIČ Dušan. *Dušanove stránky o kompresii [webová stránka].* http://pakuj.brek.sk/jpeg/jpeg.html

Typeset: LaTeX. Práca bola napísaná v editore TeXstudio (http://www.texstudio.org) v auguste 2016.