

## Komprese obrazových dat s využitím Huffmanova kódování

Tato dokumentace popisuje kompresní program `huff_codec` implementovaný v jazyce C++ do předmětu kódování a komprese dat. Program slouží pro kódování a dekódování obrázků v odstínech šedi s využitím Huffmanova kanonického kódu. Pro zvýšení kompresního výkonu je uplatněno adaptivní skenování po blocích o velikosti 64x64 bitů a dále jsou využity čtyři modely pracující s rozdílem sousedících pixelů. Tento projekt byl vypracován výhradně na základě informací uvedených na přednáškách.

### 1 Model

V programu jsou využity čtyři modely pracující s rozdílem hodnot sousedních pixelů. Byly implementovány modely procházející obraz po řádcích zleva doprava, po sloupcích shora dolů, paralelně s hlavní diagonálou shora dolů a paralelně s vedlejší diagonálou shora dolů. Kompresní výkon modelu je aproximován velikostí abecedy modelu (počtem rozdílných hodnot). Pro kódování je využit pouze model s nejmenší abecedou. Pro použití modelu během kódování programem je nutné zadat přepínač `-m`.

#### 1.1 Po řádcích

Při průchodu matice bodů  $X$  po řádcích zleva doprava je za vztažný bod zvolen levý horní bod  $x_{0,0}$ . Rozdíly sousedících bodů jsou vypočítány pomocí následujícího předpisu.

$$y_{i,j} = \begin{cases} x_{i,j} - x_{i,j-1} & j > 0 \\ x_{i,j} - x_{i-1,j} & j = 0 \wedge i > 0 \\ 0 & \text{jinak} \end{cases}$$

#### 1.2 Po sloupcích

Při průchodu matice bodů  $X$  po sloupcích shora dolů je za vztažný bod zvolen levý horní bod  $x_{0,0}$ . Rozdíly sousedících bodů jsou vypočítány pomocí následujícího předpisu.

$$y_{i,j} = \begin{cases} x_{i,j} - x_{i-1,j} & i > 0 \\ x_{i,j} - x_{i,j-1} & i = 0 \wedge j > 0 \\ 0 & \text{jinak} \end{cases}$$

#### 1.3 Paralelně s hlavní diagonálou

Při průchodu matice bodů  $X$  paralelně s hlavní diagonálou z levého horního bodu ke spodnímu pravému je za vztažný bod zvolen levý horní bod  $x_{0,0}$ . Rozdíly sousedících bodů jsou vypočítány pomocí následujícího předpisu.

$$y_{i,j} = \begin{cases} x_{i,j} - x_{i-1,j-1} & i > 0 \wedge j > 0 \\ x_{i,j} - x_{i-1,j} & i > 0 \wedge j = 0 \\ x_{i,j} - x_{i,j-1} & i = 0 \wedge j > 0 \\ 0 & \text{jinak} \end{cases}$$

## 1.4 Paralelně s vedlejší diagonálou

Při průchodu matice bodů  $X$  paralelně s vedlejší diagonálou z pravého horního bodu ke spodnímu levému je za vztažný bod zvolen pravý horní bod  $x_{n,n}$ . Rozdíly sousedících bodů jsou vypočítány pomocí následujícího předpisu.

$$y_{i,j} = \begin{cases} x_{i,j} - x_{i-1,j+1} & i > 0 \wedge j < n \\ x_{i,j} - x_{i-1,j} & i > 0 \wedge j = n \\ x_{i,j} - x_{i,j+1} & i = 0 \wedge j < n \\ 0 & jinak \end{cases}$$

## 2 Adaptivní skenování

Velikost vstupní abecedy lze snížit s využitím adaptivního skenování, které rozdělí matici dat na menší bloky o velikosti 64x64 bitů. Během testů se ukázalo, že menší velikost bloku než 64x64 bitů nezvyšuje kompresní výkon, naopak zvyšuje velikost hlaviček v komprimovaných datech. Pro využití adaptivního skenování v programu je nutné zadat přepínač `-a`.

## 3 Kanonický Huffmanův kód

Táto sekce popisuje využití kanonického Huffmanova kódu pro kódování a dekódování obrazových dat.

### 3.1 Kódování

Pro kódování dat v matici  $X$  s abecedou  $\Sigma_{EOF} = \Sigma \cup \{EOF\}$ , kde  $\Sigma$  je abeceda matice  $X$  a  $EOF$  je speciální znak, který se v  $\Sigma$  nevyskytuje, byl využit kanonický Huffmanův kód, který byl získán transformací stromu  $T$  reprezentujícího Huffmanův kód pro  $\Sigma_{EOF}$ . Listový uzel  $n \in leaves(T)$  stromu  $T$  obsahuje znak  $a \in \Sigma_{EOF}$ , informaci o jeho četnosti výskytu v matici  $X$  ( $cnt(a) \in \mathbb{N}$ ), kde  $cnt(EOF) = 1$ , a o hloubce uzlu ve stromu ( $depth(n) \in \mathbb{N}$ ).

---

#### Algorithm 1: Konstrukce stromu Huffmanova kódu

---

**Input:** matice dat  $X$ , abeceda matice  $\Sigma_{EOF}$

**Output:** strom  $T$  Huffmanova kódu

---

```

1  $pQueue \leftarrow reversePriorityQueue()$  // uzly s nejmenším  $cnt$  jsou na čele
2 foreach  $a \in \Sigma_{EOF}$  do
3    $node \leftarrow Node()$  // vytvoř nový uzel
4    $node.symbol \leftarrow a$ 
5    $node.cnt \leftarrow cnt(a)$ 
6    $pQueue.put(node)$ 
7 end
8 while  $|pQueue| > 1$  do
9    $n \leftarrow pQueue.pop()$ 
10   $m \leftarrow pQueue.pop()$ 
11   $node \leftarrow Node()$  // vytvoř nový uzel
12   $node.cnt \leftarrow n.cnt + m.cnt$ 
13   $pQueue.put(node)$ 
14 end
15 return  $pQueue.pop()$ 
```

---

**Algorithm 2:** Konstrukce kanonického Huffmanova kódu

---

**Input:** matice dat  $X$ , abeceda matice  $\Sigma$   
**Output:** kódovací funkce  $\text{huff} : \Sigma_{EOF} \rightarrow \mathbb{N}_0$

```

1  $\Sigma_{EOF} \leftarrow \Sigma \cup \{EOF\}$ 
2  $tree \leftarrow \text{getHuffmanTree}(X, \Sigma_{EOF})$ 
3  $sortedLeaves \leftarrow \text{ascendSort}(\text{leaves}(tree))$  // podle  $\text{depth}(node)$  a  $(node.symbol)$ 
4  $n_0 \leftarrow sortedLeaves[0]$ 
5  $\text{huff}(n_0.symbol) \leftarrow 0$ 
6  $i \leftarrow 1$ 
7 while  $i < |sortedLeaves|$  do
8    $n_{i-1} \leftarrow sortedLeaves[i-1]$ 
9    $n_i \leftarrow sortedLeaves[i]$ 
10   $\text{huff}(n_i.symbol) \leftarrow \text{huff}(n_{i-1}.symbol + 1) << \text{depth}(n_i) - \text{depth}(n_{i-1})$ 
11   $i \leftarrow i + 1$ 
12 end
13 return  $\text{huff}$ 

```

---

**Dekódování**

Pro dekodování jsou z hlavičky dat kanonického Huffmanova kódu získány: maximální délka kódového slova  $N$ , funkce  $L : \mathbb{N} \rightarrow \mathbb{N}_0$ , kde  $L(i)$  udává počet kódových slov délky  $i$ , a kódová abeceda  $\Sigma$  s uspořádáním odpovídajícím  $sortedLeaves$  z předchozího algoritmu. Dekódování dat je prováděno přímo bez konstrukce binárního stromu.

**Algorithm 3:** Dekódování kanonického Huffmanova kódu

---

**Input:** vstupní proud dat  $\text{inStream}$ , maximální délka kódového slova  $N$ ,  $L : \mathbb{N} \rightarrow \mathbb{N}_0$ ,  $\Sigma$   
**Output:** dekodovaná data  $\text{out}$

```

1  $c \leftarrow 0$ 
2  $s \leftarrow 1$ 
3 for  $i = 0 \dots N$  do
4    $\text{firstCode}(i) \leftarrow c$ 
5    $\text{firstSymbol}(i) \leftarrow s$ 
6    $s \leftarrow s + L(i)$ 
7    $c \leftarrow (c + L(i)) << 1$ 
8 end
9  $l \leftarrow c \leftarrow 0$ 
10  $symbol \leftarrow NaN$ 
11 while  $symbol \neq EOF$  do
12    $c \leftarrow (c << 1) + \text{getBit}(\text{inStream})$ 
13   if  $(c << 1) < \text{firstCode}(l + 1)$  then
14      $symbol \leftarrow \Sigma[\text{firstSymbol}(l) + c + \text{firstCode}(l) - 1]$ 
15     if  $symbol \neq EOF$  then
16        $\text{out.append}(symbol)$ 
17     end
18      $l \leftarrow c \leftarrow 0$ 
19   else
20      $l \leftarrow l + 1$ 
21   end
22 end
23 return  $\text{out}$ 

```

---

## 4 Hlavičky

Zakódovaná data obsahují hlavičky dvou úrovní. Nejvyšší úroveň informuje o šířce dekódovaného obrazu (**Image Width**), zda bylo použito adaptivní skenování (**Adaptive** = 1) a zda byl použit model (**Model** = 1). Za hlavičkou následují data jednotlivých zakódovaných bloků obrazu. Na konci dat se nachází zarovnání.

Image Width	16b
Adaptive	1b
Model	1b
Data Blocks	
Padding	0 ... 7b

Obrázek 1: Význam jednotlivých bitů v zakódovaném obraze.

Hlavička jednotlivých bloků obsahuje informaci o směru průchodu datovou maticí v případě využití modelu (**Pass**, kde **00** = průchod zleva doprava, **01** = průchod shora dolů, **10** = průchod paralelně s hlavní diagonálou a **11** = průchod paralelně s vedlejší diagonálou), referenční bod modelu (**Ref Point**), maximální délku slova v kanonickém Huffmanově kódu (**N**), počet kódových slov délky  $i$  ( $L_i$  pro  $0 < i \leq N$ ) a symboly abecedy ( $\Sigma_0$  až  $\Sigma_K$ ). V případě, že při kódování nebyl využit model, nemají pole **Pass** a **Ref Point** význam. Za hlavičkou se nachází surová data určena pro dekódování. Data jsou zakončena zakódovaným znakem EOF.

Pass	2b
Ref Point	8b
N	16b
$L_1$	9b
$\vdots$	
$L_N$	9b
$\Sigma_0$	9b
$\vdots$	
$\Sigma_K$	9b
Huffman Data	

Obrázek 2: Význam jednotlivých bitů v zakódovaném datovém bloku.

## 5 Experimentální výsledky

Testování bylo prováděno na 8 jádrovém procesoru AMD Ryzen 7 3800XT s 32 GB RAM. V následujících tabulkách jsou uvedeny efektivita jednotlivých kompresí: bez využití modelu a bez adaptivního skenování, s využitím adaptivního skenování (-a), s využitím modelu (-m) a s využitím adaptivního skenování a modelu (-a -m). Tabulka 1 udává výsledky pro jednotlivé soubory ze zadání. Tabulka 2 uvádí průměrnou efektivitu kompresí v rámci všech souborů.

		df1h	df1hvx	df1v	hd01	hd02	hd07	hd08	hd09	hd12	nk01
	Entropie	8.00	4.51	8.00	3.83	3.64	5.58	4.21	6.62	6.17	6.47
Efektivita	NaN	8.01	4.58	8.01	3.88	3.70	5.61	4.23	6.66	6.20	6.50
	-a	6.18	3.91	6.18	3.87	3.77	4.64	3.84	5.59	5.41	6.28
	-m	1.00	1.96	1.00	3.41	3.34	3.75	3.29	4.50	4.19	5.48
	-a -m	1.04	1.54	1.03	3.40	3.35	3.38	3.20	4.30	3.85	5.62
Čas [s]	NaN	.026	.016	.022	.020	.020	.024	.020	.027	.025	.029
	-a	.021	.016	.021	.024	.024	.026	.021	.031	.029	.033
	-m	.014	.021	.014	.036	.037	.036	.034	.042	.037	.051
	-a -m	.017	.020	.017	.041	.043	.038	.040	.048	.044	.059

Tabulka 1: Tabulka uvádí výsledky komprese s využitím kombinací adaptivního skenování (-a) a modelu (-m). Efektivita komprese udává průměrný počet bitů potřebných k zakódování jednoho pixelu.

Přepínače	Efektivita
NaN	4.78
-a	4.14
-m	2.66
-a -m	2.39

Tabulka 2: Tabulka uvádí průměrnou efektivitu pro různé kombinace adaptivního skenování (-a) a modelu (-m).

## 6 Překlad a spuštění

Tato sekce poskytuje základní informace o sestavení projektu, jeho spuštění a automatickém testování.

### 6.1 Překlad

Po zadání příkazu `make` v kořenovém adresáři projektu dojde k přeložení a vytvoření spustitelného souboru `huff_codec`. Příkazem `make clean` pak jsou odstraněny všechny soubory vytvořené během překladu, a také soubor `huff_codec`.

### 6.2 Spuštění

Program lze spustit v režimu kódování s přepínačem `-c`, nebo dekódování `-d`. Při kódování je nutné zadat šířku obrazu (`-w`). Šířka musí být s intervalu  $\langle 1; 65535 \rangle$ . Pro kódování lze použít model (`-m`) a adaptivní skenování (`-a`). Programu je nutné vždy specifikovat vstupní (`-i`) a výstupní (`-o`) soubory.

### 6.3 Automatické testy

Za účelem testování kompresního programu byly vytvořeny automatické testy, které se nacházejí ve složce `test`. Automatické testy lze spustit z kořenové složky projektu zadáním příkazu `make test`.