Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Implementace metody Tournament kódování pro kompresi celočíselných posloupností

Plzeň 2013 Václav Štengl

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 27. června 2013

Václav Štengl

Abstract

English

The main goal of this bachelor thesis is the implementation of the non-statical coding technique for compression of integer sequences. In the theoretical part of this thesis Tournament coding is thoroughly described and investigated. The practical part is focused on efficient implementation of this method. One of requirements for the method is the ability of sequence coding up to the 10^8 of elements. The implementation is proposed regard to minimal memory demand.
The method has been compared to commonly used compression methods. Huffman, arithmetic and Fibonacci coding have been chosen for the comparison. Sequences with uniform, exponential, Laplace and normal distribution have been used for testing.

Česky

Hlavním cílem této bakalářské práce je implementace nestatistické kompresní metody pro kódování číselných posloupností. V teoretické části je Tournament kódování důkladně popsáno a analyzováno.
Praktická část je zaměřena na efektivní implementaci metody. Jedním z požadavků na metodu je schopnost zakódovat posloupnosti řádu 10^8 prvků. Implementace je navržena s ohledem na minimální pamět'ovou náročnost.
Metoda byla porovnána s běžně používanými kompresními metodami. Pro srovnání bylo zvoleno Huffmanovo, aritmetické a Fibonacciho kódování. Metody byly testovány na posloupnostech s normálním, exponenciálním, rovnoměrným a Laplaceovým rozdělením.

Obsah

1 Úvod **1**

2 Teoretická část **2**

2.1 Invertované soubory . . . 3
2.2 Entropie . . . 4
2.3 Kódování čísla z omezeného intervalu . . . 6
2.3.1 Kód s pevnou délkou . . . 6
2.3.2 Kód s proměnlivou délkou . . . 7
2.4 Fibonacciho kódování . . . 9
2.5 Huffmanovo kódování . . . 11
2.6 Aritmetické kódování . . . 14
2.7 Tournament kódování . . . 17
2.7.1 Komprese . . . 18
2.7.2 Dekomprese . . . 20
2.7.3 Vylepšení . . . 20

3 Praktická část **21**

3.1 Implementace Tournament kódování . . . 21
3.1.1 Uložení stromu . . . 21
3.1.2 Komprese a dekomprese . . . 24
3.1.3 Ukládání pozice prvku . . . 25
3.1.4 Rozsah kódovaných čísel . . . 26
3.1.5 Rozložení kódovací tabulky . . . 27
3.2 Srovnání metod . . . 28
3.2.1 Rovnoměrné rozdělení . . . 28
3.2.2 Normální rozdělení . . . 29
3.2.3 Exponenciální rozdělení . . . 30
3.2.4 Laplaceovo rozdělení . . . 31
3.2.5 Zhodnocení výsledků . . . 32

4 Závěr **35**

Seznam obrázků **37**

Seznam tabulek **38**

1 Úvod

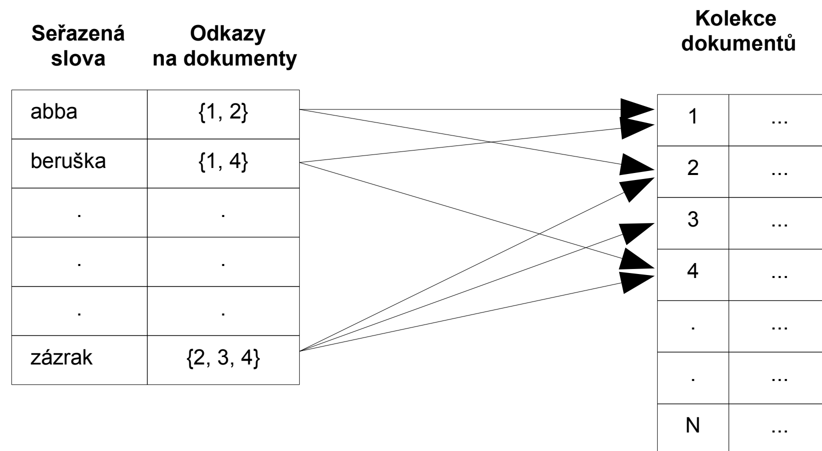
V dnešní době existuje mnoho kompresních metod, jejichž společným cílem je maximální komprese vstupních dat. Každá z metod pracuje odlišně a podle nároků, které jsou na metodu kladeny, je vybrána ta s požadovanými vlastnostmi. Pokud je například kladen důraz na výsledný kompresní poměr, je doporučena některá ze statistických metod. Hlavními kandidáty jsou Huffmanovo kódování [9] a aritmetické kódování [3]. Tyto dvě metody ovšem mají nevýhodu, protože k samotnému kódování potřebují pravděpodobnostní model. Ten musí být spočítán před samotným kódováním a pro větší množství prvků může nabýt značné velikosti - například extrémní případ, kdy je každý prvek obsažen ve vstupních datech pouze jednou. Existují také adaptivní verze Huffmanova a aritmetického kódování, které si pravděpodobností model vytváří během samotného kódování, ale tyto metody jsou relativně pomalé.
Právě tady byl spatřen prostor pro zlepšení, a proto byla představena nová kompresní metoda pro kódování kladných celočíselných posloupností nazvaná *Tournament kódování*. Tato metoda se řadí mezi nestatické metody - u nestatických metod není prvek kódován samostatně, ale je rozhodující jeho pozice vůči ostatním prvkům. Nelze tedy spoléhat na to, že každý prvek je zakódován jedním kódem jako u statických metod, ale jeden kód může označovat v různých fázích více prvků. Tournament kódování je navrženo s důrazem na schopnost efektivně zakódovat velké posloupnosti s velkým rozsahem hodnot. Dále je kladen důraz nejen na rychlost kódování, ale také na dekódování.
Tato bakalářská práce si klade za cíl popsat Tournament kódování, jeho implementaci a provést srovnávací testy s běžně užívanými kódovacími metodami. Pro srovnání je použito Fibonacciho kódování a výše zmiňované Huffmanovo a aritmetické kódování.

2 Teoretická část

Komprese je v dnešní době velmi rozšířena a široce využívána při ukládání, archivaci nebo přenosu libovolného typu dat. Potřebujeme-li efektivně uložit nebo co nejrychleji přenést po síti větší objem dat, využití komprese je takřka neodmyslitelná záležitost. V závislosti na typu dat volíme mezi ztrátovou a bezeztrátovou kompresí.
Metody ztrátové komprese běžně dosahují většího kompresního poměru než metody bezeztrátové, nevýhodou ovšem může být ztráta některých dat při následné dekompresi. Ztrátové metody totiž nezaručují, že dekomprimovaná data budou totožná s původními daty, ale výsledkem jsou data, která jsou původním datům velmi podobná. Informace, které algoritmus vyhodnotí jako nedůležité, jsou nenávratně ztraceny. Při ztrátové kompresi jsou tedy data rozdělena na důležitá a nedůležitá, aby nedošlo ke ztrátě významných dat. Nedůležitá data jsou zahozena a na důležitá data se může aplikovat např. libovolná bezeztrátová metoda. Tento způsob komprese se nejčastěji využívá při ukládání zvukových nebo obrazových záznamů, kde díky nedokonalosti lidských smyslů vznikne stále dostatečně kvalitní rekonstrukce původních dat. Ztrátová komprese je využívána například u formátů JPEG, MPEG, H.264, MP3 a v mnoha jiných.
Druhým typem komprese, do kterého spadají všechny kompresní metody použité v této práci, je tzv. bezeztrátová komprese. Metody bezeztrátové komprese nedosahují tak dobrých kompresních poměrů, výsledná data po dekompresi jsou ovšem shodná s původními daty. Žádná data tedy nejsou ztracena, což je žádoucí např. u textových dat nebo u dat, u kterých si nemůžeme dovolit byt' minimální změnu.
Komprese je zmenšení velikosti vstupních dat pomocí kompresní metody. Vstupem mohou být libovolná (číselná, textová, binární, ...) data a na výstupu jsou obdržena data binární, jejichž celková velikost by měla být menší než velikost původních dat. Pomocí dekomprese lze poté z výsledných binárních dat zrekonstruovat data původní.
Tournament kódování se řadí k bezeztrátovým kompresním metodám a je určeno ke kódování číselných posloupností. To ale není omezující pro data, která nejsou celočíselná, protože číselnými posloupnostmi lze reprezentovat různé typy dat (text reprezentovaný číselnou posloupností jednotlivých ASCII hodnot znaků, binární data seskupením jednotlivých bytů nebo skupiny bytů, ...). Efektivní komprese číselných posloupností je klíčová například u *invertovaných souborů*, u kterých je kladen důraz nejen na efektivitu komprese, ale především na rychlost dekomprese.

2.1 Invertované soubory

Jak je uvedeno v [2], invertovaný soubor je datová struktura používaná pro vyhledávání dokumentů splňujících zadanou podmínku týkající se termínů charakterizujících dokument. Jednoduchá podmínka na dokument může být výskyt klíčových slov, složitější podmínka může zahrnovat vzájemnou pozici slov, počet výskytů nebo výskyt slova v určité části dokumentu. Rozsah prohledávaných dokumentů je často velmi značný. Může se jednat o knihovní fond reprezentovaný bibliografickými záznamy uchovávanými v počítači, v případě webového vyhledávače je to ohromné množství textových dat. Invertovaný soubor je nejčastěji reprezentován seřazeným seznamem slov (termů) a každému slovu přísluší seznam dokumentů, ve kterých se vyskytuje. Jednoduchá reprezentace invertovaného souboru se může skládat z množiny dvojic (slovo, {seznam souřadnic}), viz obrázek 2.1.



Obrázek 2.1: Základní struktura invertovaného souboru

Pro velmi rychlé vyhledání konkrétního slova v záznamech ukládá invertovaný soubor pro každé slovo nejen seznam čísel dokumentů, ale navíc pozice výskytů slova v dokumentu. Tím se velikost invertovaného souboru zvětšuje a při velkém množství dokumentů vznikají rozsáhlé číselné posloupnosti, které je třeba co nejúsporněji uchovat v paměti. Při kódování invertovaných souborů je kladen důraz nejen na výsledný kompresní poměr, ale také na čas

potřebný ke kompresi a hlavně k dekompresi. Tato datová struktura je určena pro rychlé vyhledávání v dokumentu, a proto musí být čas dekódování co nejmenší. Kódováním indexů se zabývá [6], která je velmi blízká Tournament kódování.

2.2 Entropie

Běžně používané způsoby pro reprezentaci dat jsou navrženy tak, aby umožňovaly snadnou manipulaci s daty a v důsledku toho obsahují řadu redundancí. Toho se snaží využít všechny kompresní metody, které se tyto redundantní informace snaží najít a odstranit [3]. Příkladem takovéto reprezentace je obyčejné ukládání ASCII znaků. Znaky v základní části ASCII tabulky jsou vyjádřeny rozsahem 0-127, pro uložení jednoho znaku tedy postačí 7 bitů. V praxi jsou ale tyto znaky pro jednodušší manipulaci ukládány na celý byte. Tímto způsobem vznikne pro každý znak jeden nevyužitý bit. Kompresní metody se obecně snaží odstranit redundanci a opakující se výskyty v datech a tím snižují nároky na paměť, která je potřebná pro uložení dat. Obecně lze tedy říci, že cílem komprese je omezení velikosti dat do takové míry, aby odpovídala množsví informace v nich obsažené. Způsob, jak zjistit míru informace v datech, se nazývá entropie, někdy také nazývaná Shannonova entropie po Claude E. Shannonovi, který zformuloval mnoho klíčových poznatků teoretické informatiky.

Definice entropie [3]: Nechť se data skládají z n různých prvků

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

a tyto prvky se v datech vyskytují s pravděpodobnostmi

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

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

$$E_i = -log_2(p_i). \tag{2.1}$$

Entropie i-tého prvku E_i vyjadřuje, jak velkou hodnotu informace nese jeho výskyt. Platí tedy, že čím je pravděpodobnost výskytu prvku a_i větší, tím je jeho míra informace menší a tedy entropie také menší.

Příklad: Ve fotbalovém zápase se mají střetnout dva týmy. Domácí tým je favoritem zápasu, proto sázkové kanceláře vypsaly kurz 1.25 na jeho vý-

hru. Hostující tým je v roli outsidera s kurzem 33.3 na výhru. Na remízu je vypsán kurz 5.88. Zjistěme, jak budou vypadat entropie jednotlivých jevů.

Podle definice (2.2) lze spočítat ze známých kurzů pravděpodobnosti jednotlivých jevů a podle vzorce (2.1) se vypočítají entropie uvedených jevů:

Definice: Nechť je dán jev j_i, na který sázková kancelář vypíše kurz k_i, pak pravděpodobnost jevu j_i vypočteme podle vztahu:

$$p_i = \frac{1}{k_i}. \tag{2.2}$$

Tabulka 2.1: Entropie jevů

jev	**kurz**	**ppst jevu**	**entropie**
výhra domácích	1.25	0.8	0.32
remíza	5.88	0.17	2.56
výhra hostů	33.3	0.03	5.06

Z hodnot uvedených v tabulce 2.1 je patrné, že entropie je tím větší, čím je menší pravděpodobnost jevu. Výskyt prvku s vysokou entropií má tedy vyšší informační hodnotu než výskyt prvku s menší entropií.
Průměrné množství informace v každém symbolu je rovno tzv. střední entropii. Ta se spočte tak, že se entropie jednotlivých prvků vynásobí s jejich pravděpodobností výskytu a tyto hodnoty se poté sečtou:

$$E = \sum_{i=1}^{n} p_i \cdot E_i = -\sum_{i=1}^{n} p_i \cdot log_2(p_i) \tag{2.3}$$

Příklad: Potřebujeme uložit text, který obsahuje pouze znaky *a,b,c,d,e*. Ukládaný text může vypadat například takto: *abcdeabc*. Relativní četnosti jednotlivých znaků, ze kterých je odhadnuta pravděpodobnost a entropie, jsou uvedeny v tabulce 2.2.

Tabulka 2.2: Entropie jevů

znak	**četnost**	**ppst výskytu**	**entropie**
a	2	0.25	2
b	2	0.25	2
c	2	0.25	2
d	1	0.125	3
e	1	0.125	3

Celková entropie vypočtená podle vzorce (2.3)

$$E = 0.25 \cdot 2 + 0.25 \cdot 2 + 0.25 \cdot 2 + 0.125 \cdot 3 + 0.125 \cdot 3 = \\ = 0.5 + 0.5 + 0.5 + 0.375 + 0.375 = 2.25 \text{ bitů na symbol}$$

Výsledná celková entropie odráží celkovou míru informace v datech. K zakódování jednoho znaku by podle entropie mělo stačit 2.25 bitů. Pro srovnání je následně uvedeno, jak moc se k této hranici přiblíží dvě jednoduché kódovací metody, které jsou využívány pro kódování čísel z předem známého intervalu.

2.3 Kódování čísla z omezeného intervalu

Během komprese pomocí Tournament kódování často nastane situace, ve které je potřeba zakódovat číslo c_i, o kterém se ví pouze to, že leží v intervalu $[a_i, b_i]$. Je potřeba rychlá a zároveň efektivní kompresní metoda, která jednoznačně zakóduje toto číslo bez ohledu na interval, který je proměnlivý. Jediná dodatečná informace, která může být známa, je pravděpodobnost výskytu jednotlivých čísel, která může vylepšit kompresní poměr.

2.3.1 Kód s pevnou délkou

Jak už je patrné z názvu metody, kódy všech čísel spadajících do omezujícího intervalu disponují stejnou délkou kódu. Nebere se tedy v úvahu entropie čísla, ale ke všem číslům je přistupováno stejně. Každému číslu z intervalu $<a, b>$ je přiřazen kód o délce $\lceil log_2(m) \rceil$, kde m označuje celkový počet čísel

v intervalu, tedy $m = a - b + 1$. Toto kódování je vhodné, pokud neznáme entropie jednotlivých čísel a pokud je m rovno mocnině čísla 2. Pro ostatní hodnoty m existuje vždy několik kódů stejné délky, které zůstanou nepřiřazeny.
Použití kódu s pevnou délkou je ukázáno na posledním příkladu s entropií.

Tabulka 2.3: Kód s pevnou délkou

znak	**kód**
a	000
b	001
c	010
d	011
e	100

Pokud je text *abcdeabc* z posledního příkladu zakódován podle tabulky 2.3, výsledný kód je 000001010011100000001010. Délka kódu je 24 bitů a k zakódování jednoho znaku jsou tedy potřeba 3 bity. Podle celkové entropie je ale hodnota informace 2.25 bitů na znak. Tato kompresní metoda tedy není pro uvedený příklad optimální, protože znaky, které se vyskytují častěji, jsou kódovány stejně jako znaky, které se vyskytují jen zřídka. Tento nedostatek se snaží odstranit kódování s proměnlivou [1] délkou kódu.

2.3.2 Kód s proměnlivou délkou

Narozdíl od předchozích kódů, které jsou identifikovány podle délky, u kódů s proměnlivou délkou na to nelze spoléhat. Každý kód musí být jednoznačně definovaný svojí hodnotou - žádný z kódů nesmí být prefixem jiného. Obrovským přínosem této metody je možnost přiřazení kratších kódů častěji se vyskytujícím číslům v případě znalosti pravděpodobnostního modelu. Tím je dosaženo nejen kratšího celkového kódu, ale také lepšího kompresního poměru.
Kód s proměnlivou délkou přiřazuje m_1 číslům kódy délky $\lfloor log_2(m) \rfloor$, kde $m_1 = 2^{\lceil log_2(m) \rceil}$, a kódy o délce $\lceil log_2(m) \rceil$ zbytku čísel $m_2 = m - m_1$. V tabulce 2.4 jsou vygenerovány kódy pro vybrané intervaly. Z tabulky lze vypo-

[1] Existuje více kódů s proměnlivou délkou a popisovaný kód je jedním z nich. V anglických pracích je nazýván semi-fixed-length coding. V českém jazyce zatím není definovaný překlad, a proto je v celé práci použit název kód s proměnlivou délkou.

zorovat, že pro délku intervalu rovnou mocnině 2 se proměnlivé kódy mění na kódy s pevnou délkou. Pro ostatní délky intervalu se vždy vezme první nejkratší kód z předchozího intervalu a rozšíří se na dva nové kódy přidáním nuly a jedničky na konec kódu. Díky tomuto systému generování kódu nemusí být předávána tabulku kódů, ale podle známé délky intervalu je možno tuto tabulku vygenerovat.

Tabulka 2.4: Kódy s proměnlivou délkou.

	délka intervalu									
číslo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	00	00	000	000	000	000	0000	0000
1		1	01	01	001	001	010	001	0001	0001
2			1	10	01	010	010	010	001	0010
3				11	10	011	011	011	010	0011
4					11	10	100	100	011	010
5						11	101	101	100	011
6							11	110	101	100
7								111	110	101
8									111	110
9										111

U posledního příkladu je třeba zakódovat 5 znaků, pro zakódování tedy bude použita tabulka kódů pro délku intervalu 5.
Jsou-li přiřazeny kódy s proměnlivou délkou k jednotlivým znakům, viz tabulka 2.5, zakódovaný text bude vypadat následovně: 011011000001011011. Tento kód obsahuje 18 bitů, což je o šest méně než kód s pevnou délkou. Průměrný počet bitů na znak je roven $18/8 = 2.25$ bitů a to je i hodnota vstupní entropie. Tato metoda je tedy pro uvedený příklad optimální.

Tabulka 2.5: Kód s proměnlivou délkou

znak	**entropie**	**kód**
a	2	01
b	2	10
c	2	11
d	3	000
e	3	001

Kódy s proměnlivou délkou jsou použity např. v [6] a také v Tournament kódování. Podle [1] existuje exponenciální počet možných přiřazení kódů jednotlivým číslům, ale čtyři z nich jsou nejvíce používané a také snadno spočítatelné. Tyto čtyři způsoby demonstruje tabulka 2.6.

Tabulka 2.6: Přiřazení kódů s proměnlivou délkou číslům {0,...,5}

Číslo	**Low-short**	**High-short**	**Mid-short**	**Mid-long**
0	10	000	000	10
1	11	001	001	000
2	000	010	10	001
3	001	011	11	010
4	010	10	010	011
5	011	11	011	11

Pro jednotlivá přiřazení musí platit, že žádný z kódů není prefixem jiného. Není známo, jak dlouhý daný kód může být, a proto musí být jednoznačně identifikován svojí hodnotou. Pro dekódování originálního čísla je třeba znát interval, pro který byl daný kód vytvořen. Neplatí totiž, že číslo je kódováno vždy stejně pro různé intervaly. Díky tomu, že číslo nabývá pro různé intervaly různých hodnot, nemá smysl tabulku kódů dopředu počítat, protože se často mění. Naopak výpočet musí být rychlý, protože je používán velmi často.
Tournament kódování používá přiřazení *High-short* a *Low-short*. Jak je vysvětleno v [6], efektivita kompresní metody může být vylepšena volbou přiřazení, které přiřadí kratší hodnoty frekventovanějším prvkům. Také Tournament kódování uvažuje nad volbou správného přiřazení a různá přiřazení jsou aplikována v různých krocích metody.

2.4 Fibonacciho kódování

V této sekci bude vysvětlen základní princip Fibonacciho kódování druhého řádu. Tato metoda je založena na Fibonacciho číslech [7] a jejich specifických vlastnostech. I-té číslo Fibonacciho posloupnosti je definováno následovně:

$$\begin{aligned} F_i = F_{i-1} + F_{i-2}, pro\ i \geq 1, \\ kde\ F_{-1} = F_0 = 1. \end{aligned} \tag{2.4}$$

Jak je uvedeno v [8], Fibonacciho kód je počítán podle definice 1:

Definice 1. *Nechť $F(n) = a_0a_1a_2...a_p$ jsou Fibonacciho čísla pro kladné n, pak hodnota Fibonacciho kódu, označená $V(F(n))$, je definována následovně:*

$$V(F(n)) = n = \sum_{i=0}^{p} a_i \cdot F_i \quad (a_i \in \{0,1\},\ 0 \leq i \leq p) \tag{2.5}$$

Každý bit výsledného kódu reprezentuje jedno Fibonacciho číslo F_i a tento kód má specifickou vlastnost, že se vedle sebe nikdy neobjeví dva bity s kladnou hodnotou. Zvolením vhodné orientace, ve které poslední bit Fibonacciho kódu reprezentuje nejvyšší Fibonacciho číslo, je získán kód s kladným posledním bitem. Připojením kladného bitu za výsledný Fibonacciho kód se na konci objeví dva kladné bity, které jednoznačně určují konec kódu. Princip a ukázka Fibonacciho kódování pro vybraná čísla jsou uvedeny v tabulce 2.7.

Tabulka 2.7: Fibonacciho kódy řádu 2 pro čísla $\{1, \ldots, 10\}$

n	**F(n)**					**kód**
1	1	0	0	0	0	11
2	0	1	0	0	0	011
3	0	0	1	0	0	0011
4	1	0	1	0	0	1011
5	0	0	0	1	0	00011
6	1	0	0	1	0	10011
7	0	1	0	1	0	01011
8	0	0	0	0	1	000011
9	1	0	0	0	1	100011
10	0	1	0	0	1	010011
F_i	1	2	3	5	8	

Dekódování probíhá analogicky ke kódování: zleva se načítají jednotlivé bity, které zastupují Fibonacciho čísla (1. bit = F_1, 2. bit = F_2, ...). Čtení probíhá do té doby, než se narazí na dva sousední kladné bity (poslední kladný bit se nebere v potaz). Celkový součet Fibonacciho čísel patřících ke všem kladným bitům vrátí původní kódované číslo.

Obrovskou výhodou Fibonacciho kódování je nezávislost jednotlivých kódů a jejich snadná identifikace, díky které je možno číst data i z poškozeného

streamu. Pokud například přijde stream s jedním změněným bitem, tato chyba může ovlivnit načtení dvou čísel, kdy se u prvního čísla správně nerozezná konec a bude načteno společně s dalším číslem. V druhém případě bude mít jeden prvek nesprávnou hodnotu, ale zbytek posloupnosti už bude v pořádku. U většiny ostatních kompresních metod znamená byt' minimální změna ve streamu ztrátu veškerých dat.
Fibonacciho čísla lze generovat pro libovolně velké kladné číslo, a proto není toto kódování nijak omezeno rozsahem hodnot. Pro velká čísla se ale moc nehodí, protože například poslední Fibonacciho číslo, které je menší než maximální hodnota neznaménkového integeru ($2^{32} - 1$), je v pořadí 46-té. Z toho vyplývá, že pro zakódování velkých čísel pohybujících se právě u této maximální hranice je potřeba $46 + 1 = 47$ bitů namísto 32, které jsou použity u počítačové reprezentace. Naopak malým číslům jsou přiřazeny kratší kódy, a proto se při volbě Fibonacciho kódování předpokládá větší výskyt malých hodnot.

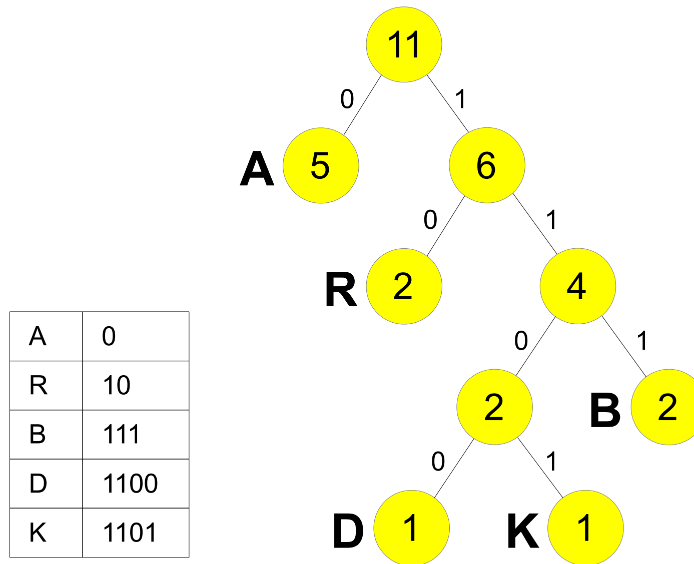
2.5 Huffmanovo kódování

Huffmanovo kódování [9] pochází z roku 1952 a jeho algoritmus byl navržen Davidem Huffmanem. Je založeno na kódování symbolů pomocí prefixových kódů, kde jsou prvkům s vysokou četností přiděleny kódy s menším počtem bitů a naopak prvky, které se vyskytují zřídka (s malou četností), se kódují delším kódem. K získání jednotlivých kódů je využit binární strom, který je vytvořen podle algoritmu popsaného v následujícím odstavci.
Algoritmus nejprve spočte četnosti (pravděpodobnosti) jednotlivých znaků, podle kterých sestaví binární strom. Listy binárního stromu obsahují kódované znaky, zatímco vnitřní uzly obsahují součet četností svých potomků. Strom je vytvořen tak, aby vzdálenost od kořene k listu byla nepřímo úměrná s entropií znaku. Platí tedy, že čím vyšší je četnost a tedy i pravděpodobnost výskytu, tím menší je vzdálenost daného znaku od kořene. Když je binární strom sestaven, jsou všechny levé hrany ohodnoceny hodnotou 0 a všechny pravé hodnotou 1. Cesta od kořene k listu tak vytváří binární kód pro každý znak. Jelikož jsou všechny znaky uloženy v listech, nemůže nastat situace, že jeden kód je prefixem druhého. Tyto kódy jsou tedy prefixové. Přesný postup algoritmu je uveden níže a je převzatý z [9].

Algorithm 1 Algoritmus pro vytvoření Huffmanova binárního stromu

```
begin
spočíst četnosti jednotlivých znaků (zdrojových jednotek)
output (zakódované četnosti ve Fibonacciho kódu 2. řádu)
vytvořit list (znak, četnost c, levý syn = NULL, pravý syn = NULL)
    stromu pro každý znak a vložit listy do fronty F
seřadit je do neklesající posloupnosti
while (|F| ≥ 2) do
   z fronty vybrat dva první uzly (u1, u2) s nejnižšími četnostmi
   vytvořit uzel ohodnocený součtem vybraných jednotek,
      následníci jsou vybrané jednotky (eps, c(u1) + c(u2), u1, u2)
   zařadit nový uzel do fronty
end while
list ohodnotit cestou z kořene do listu (levý potomek 1, pravý potomek 0)
vytvořit výstup z takto zakódovaných vstupních písmen
end
```

Pro představu je uvedeno Huffmanovo kódování na konkrétním příkladě. Binární strom vytvořený podle Huffmanova algoritmu pro řetězec ABRAKADABRA je zobrazen na obr. 2.2. Písmeno A má v tabulce kódů přiřazený nejkratší možný kód o délce 1. Jelikož se toto písmeno nachází v kódovaném textu celkem pětkrát, je tento způsob komprese velmi efektivní. Na druhou stranu znaky D a K, které se v textu vyskytují pouze jednou, mají přiřazeny kódy o délce 4 bity.



A	0
R	10
B	111
D	1100
K	1101

Obrázek 2.2: Huffmanovo strom pro zakódování textu ABRAKADABRA

Huffmanovo kódování je tedy vhodné pro posloupnosti s často se opakujícími znaky. Pro takové posloupnosti se naplno projeví potenciál prefixových kódů a jejich přiřazení na základě pravděpodobnosti. Naopak pro posloupnosti s velkým rozsahem hodnot není výsledný kompresní poměr tak dobrý jako pro posloupnosti s opakujícími se znaky, protože výsledné kódy se mohou začít podobat kódům s pevnou délkou kódu - všem znakům je přiřazen kód se stejnou délkou. Navíc je potřeba předem spočítat a zakódovat pravděpodobnostní model a ten pro velké množství hodnot může nabýt ohromné velikosti.
Zaměníme-li kódy získané Huffmanovo kódováním za znaky původních vstupních dat, výsledný text se komprimuje na následující posloupnost bitů: 01111001101011000111100. K výslednému kódu musí být také přiložen binární strom, pokud není známý předem, a tak se efektivita Huffmanovo kódování projeví až pro zprávy větší délky.
Huffmanovo kódování podává nejlepší výsledky, pokud je rozdíl mezi entropií

znaku a skutečnou délkou přiřazeného kódu minimální. Jelikož je každý znak kódován odděleně, jeho kód musí mít celočíselný počet bitů. Pokud tedy entropie znaku bude například 1.44 bitů a tento znak bude kódován 2 bity, nebude výsledek moc dobrý. Pro Huffmanovo kódování je tedy nejlepší, když jsou jednotlivé entropie celočíselné. Tento nedostatek nemá aritmetické kódování, které je popsané v následující části.

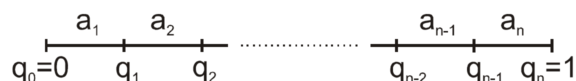
2.6 Aritmetické kódování

Aritmetické kódování je specifické tím, že výsledkem komprese je jediné číslo číslo z intervalu $\langle 0,1)$. Počáteční interval $\langle 0,1)$ je rozdělen na n disjunktních intervalů, kde n značí počet kódovaných symbolů. Velikosti jednotlivých intervalů jsou určeny pravděpodobnostmi výskytů znaků v kódované sadě. Rozdělení intervalů je dáno následujícím předpisem, kde p_i označuje pravděpodobnost výskytu znaku a_i :

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

Pro zjednodušení zápisu je vhodné zavést kumulativní pravděpodobnosti podle předpisu:

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



Obrázek 2.3: Rozdělení intervalu pro aritmetické kódování

Pokud jsou vypočítané jednotlivé intervaly, viz obr. 2.3, může být zahájena komprese. Ta probíhá tím způsobem, že při každém načtení znaku se zpřesňuje výsledný interval a jeho horní a dolní mez se k sobě přibližují. Postup algoritmu je uveden na následujícím příkladu, kde je použito aritmetické kódování ke kompresi textu *abac*.

Tabulka 2.8: Rozdělení intervalů

znak	**četnost**	**interval**
a	2	$\langle 0, 0.5)$
b	1	$\langle 0.5, 0.75)$
c	1	$\langle 0.75, 1)$

Kódování začíná s celým intervalem $I = \langle 0, 1)$. Pro každý znak z kódovaného textu se vezme jeho odpovídající interval $\langle q_{i-1}, q_i)$ a vypočte se nová hodnota intervalu I=$\langle l, r)$, ze kterého zůstane pouze ta část odpovídající kódovanému znaku a_i. Výpočet nové hodnoty intervalu I je dán následujícím vzorcem:

$$I =< l + q_{i-1} \cdot (r - l);\ l + q_i \cdot (r - l)) \qquad (2.6)$$

Má-li být zakódován text *abac*, bude interval I omezen následovně:
První znak v textu je *a*, který je reprezentován intervalem $\langle 0, 0.5)$. Interval I bude omezen podle vzorce 2.6:

$$I =< l + q_{i-1} \cdot (r - l);\ l + q_i \cdot (r - l)) =$$
$$< 0 + 0 \cdot (1 - 0);\ 0 + 0.5 \cdot (1 - 0)) = < 0; 0.5)$$

Tento výpočet je aplikován analogicky na všechny znaky kódovaného textu. Výpočet je zanesen do tabulky 2.9.

Tabulka 2.9: Komprese aritmetického kódování pro řetězec *abac*.

znak	**část intervalu**	**původní** I	**nové** I
a	$\langle 0; 0.5)$	$\langle 0; 1)$	$\langle 0; 0.5)$
b	$\langle 0.5; 0.75)$	$\langle 0; 0.5)$	$\langle 0.25; 0.375)$
a	$\langle 0; 0.5)$	$\langle 0.25; 0.375)$	$\langle 0.25; 0.3125)$
c	$\langle 0.75; 1)$	$\langle 0.25; 0.3125)$	$\langle 0.296875; 0.3125)$

Jak už bylo uvedeno dříve, výsledkem aritmetického kódování je jediné číslo z intervalu $\langle 0; 1)$. Podle tabulky 2.9 je ale v posledním kroku získán interval. To ovšem není překážka, protože za výsledek je možné považovat libovolné číslo nacházející se ve výsledném intervalu. V tomto případě to může být např. číslo 0.3. Obecně je výhodné použít z výsledného intervalu takové číslo, jehož binární reprezentace zabere minimální počet bitů.

Dekomprese probíhá obdobným způsobem a znaky jsou získávány ve stejném pořadí, v jakém byly kódovány. Začíná se opět s intervalem $I = \langle 0, 1)$. Pro dekódování znaku je vycházeno z výsledné hodnoty komprese c (v tomto případě $c = 0.3$) a z aktuálního intervalu I. Nejprve je nalezen index i, pro který platí následující nerovnice:

$$l + q_{i-1} \cdot (r - l) \ \leq \ c \ < \ l + q_i \cdot (r - l) \tag{2.7}$$

nebo analogicky:

$$q_{i-1} \ \leq \ \frac{c - l}{r - l} \ < \ q_i \tag{2.8}$$

Podle nalezeného indexu je dekódován jeden znak a vypočtena nová hodnota intervalu I podle vzorce 2.6. Tento způsob je analogicky opakován, dokud není dekódován určitý počet znaků. Na začátek zprávy musí být připojen počet zakódovaných znaků, aby dekódovací algoritmus včas skončil s dělením intervalu. Následující tabulka ukazuje postup dekódování:

Tabulka 2.10: Dekomprese aritmetického kódování pro řetězec *abac*.

aktuální I	$\frac{c-l}{r-l}$	**část intervalu**	**znak**	**vypočtené** I
$\langle 0; 1)$	0.3	$\langle 0; 0.5)$	a	$\langle 0; 0.5)$
$\langle 0; 0.5)$	0.6	$\langle 0.5; 0.75)$	b	$\langle 0.25; 0.375)$
$\langle 0.25; 0.375)$	0.4	$\langle 0; 0.5)$	a	$\langle 0.25; 0.3125)$
$\langle 0.25; 0.3125)$	0.8	$\langle 0.75; 1)$	c	

Pro velký počet znaků se interval I nebezpečně zmenšuje, až jsou čísla tak malá, že je nelze reprezentovat datovými typy jako je float nebo double. Tyto typy totiž mají danou strojovou přesnost, která není nekonečná. Je zde potřeba ukládat opravdu malá čísla, která mohou být daleko za touto hranicí. Proto je třeba použít takovou techniku, která je schopna pracovat s „nekonečným” počtem desetinných míst.
Jedno z možných řešení je ukládat samostatně čísla na řádech, ve kterých se obě meze intervalu rovnají. Je-li například uprostřed výpočtu interval I = <0.63; 0.67>, výsledné číslo bude vždy obsahovat hodnotu 6 na řádu desetin, a proto může být rovnou uložena. Po uložení hodnoty se řád obou čísel omezujících interval posune směrem doleva. Jedná se o tzv. vysunutí.

Algoritmus pokračuje dále s intervalem $I = <0.3; 0.7>$. Jednotlivá čísla jsou kódována pro každý řád zvlášť a pro uložení lze použít např. tabulku s proměnlivou délkou kódů vygenerovanou pro délku intervalu 10. Při této volbě ukládání desetinných čísel není třeba volit číslo z výsledného intervalu, které je prohlášeno za výsledek komprese, ale výsledné číslo je ukládáno průběžně, a po skončení komprese obsahuje minimální počet desetinných míst a tím zároveň zaručuje nejmenší možné množství ukládaných bitů.
Další z možných řešení je celočíselná reprezentace obou čísel určujících hranice intervalu I. Tyto hranice jsou počítány binárně a pokud mají obě čísla společný nejvyšší bit, tak je tento bit zapsán na výstup a interval I je normalizován.

Stejně jako u Huffmanova kódování, kde musel být k výsledné zprávě přiložen binární strom (pokud nebyl dohodnut předem), tak i u aritmetického kódování musí být ke zprávě přiloženo rozložení jednotlivých znaků na intervalu I. Aritmetické kódování je efektivní až pro větší množství vstupních dat.

2.7 Tournament kódování

Tournament kódování je kompresní metoda, která slouží pro kódování celočíselných posloupností. Jediné omezení na posloupnost, které metoda vyžaduje, je nezáporná hodnota prvků. Pro všechna p_i z posloupnosti $P = \{p_1, p_2, p_3, ..., p_n\}$ tedy musí platit nerovnost $p_i \geq 0$. Tato podmínka je pro zakódování celočíselných posloupností velmi limitující, proto je třeba prvky posloupnosti převést na nezáporné hodnoty. Velmi jednoduchý způsob je následující transformace, díky které je možno libovolnou posloupnost převést na posloupnost kladných čísel tak, že je vypočtena nová hodnota podle vztahů:

$$p'_i = 2 \cdot p_i \quad \text{pro nezáporné hodnoty } p_i,$$
$$p'_i = -2 \cdot p_i \text{ - } 1 \quad \text{pro záporné hodnoty } p_i.$$

Pro zpětnou rekonstrukci stačí sudé prvky vydělit dvěma, u lichých je třeba dělit minus dvěma a vybrat celou část výsledného podílu.

Další možností, jak převést posloupnost na nezáporné hodnoty, je nalezení nejmenšího prvku posloupnosti a přičtení jeho absolutní hodnoty ke všem prvkům. Toto řešení zachová původní rozdělení prvků, pouze posune hodnoty o konstantní číslo. Výhodou je skutečnost, že pro nezáporné posloupnosti

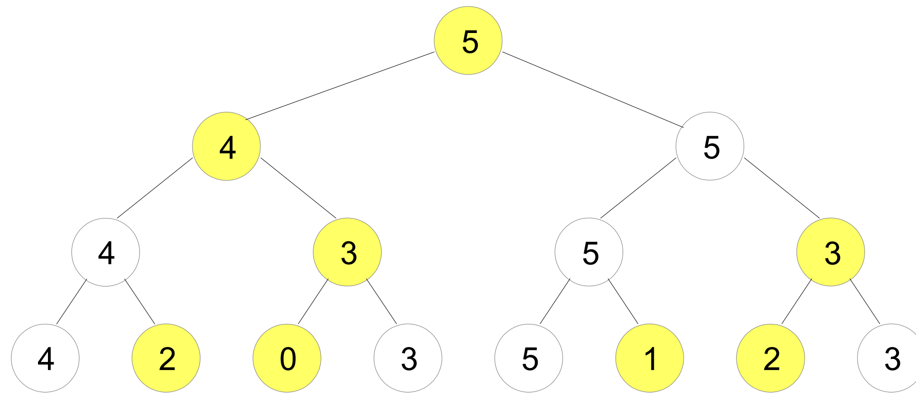
s nulovým minimálním prvkem zůstane posloupnost nezměněna. Nevýhodou tohoto řešení je znalost minimálního prvku, bez kterého není možné provést zpětný přepočet. Toto minimum tedy musí být zakódováno společně s daty.

2.7.1 Komprese

Podle názvu metody lze usoudit, že algoritmus bude nějakým způsoben souviset s turnajem (soutěží). V případě Tournament kódování se jedná o soutěžení číselných dvojic, ze kterého vyjde vítězně číslo s větší hodnotou. To je poté použito k uložení menšího čísla pomocí proměnlivé délky kódu, viz kapitola 2.3.1. Větší číslo z dvojice slouží k vymezení intervalu, pro který se generuje tabulka kódů, a tím je minimalizována délka jednotlivých kódů. Menší číslo je pak zakódováno kódem z této tabulky.
V prvním kole spolu soutěží sousední prvky a vítězná čísla z daných dvojic postupují do druhého kola. Pro lepší představu tvoří posloupnost kódovaných čísel listy binárního vyhledávacího stromu a vítězná čísla z každé dvojice se stanou rodiči obou soutěžících čísel. Nad původní posloupností tedy vzniká další posloupnost s polovičním počtem prvků (v případě lichého počtu prvků je na konec přidána nula), která obsahuje pouze vítězné prvky z prvního kola. V druhém kole se spárují vítězové z prvního kola a vítězná čísla opět postupují do dalšího kola. Tento způsob je aplikován rekurzivně a ve výsledku se vytvoří binární vyhledávací strom, jehož kořenem se stane maximální prvek posloupnosti. Vytvořený strom poté stačí projít pomocí BFS (prohledávání do šířky) a menšího z potomků vždy zakódovat pomocí svého rodiče. Pro jednoduchost budou kódy ukládané pomocí rozložení *High-short*. Je totiž pochopitelné (alespoň pro rovnoměrné rozložení), že ve vyšších patrech stromu se budou objevovat lokální maxima posloupnosti, která by měla mít podobnou velikost. Pro představu je výše uvedený postup uveden na jednoduchém příkladu:

$$P = \{4,\ 2,\ 0,\ 3,\ 5,\ 1,\ 2,\ 3\}$$

Rekurzivní aplikací výše uvedeného postupu na posloupnost P je vytvořen binární strom, který je zobrazen na obrázku 2.4. Větší číslo z každé dvojice se vždy stane rodičem a absolutní vítěz se stane kořenem celého stromu. Žlutě vyznačené uzly jsou menší z dvojice v rámci stejného patra a právě tyto uzly je třeba zakódovat, aby bylo možno při dekompresi znovu sestrojit tento strom a následně zrekonstruovat původní posloupnost. Výjimku tvoří samotný kořen, který musí být zakódován samostatně libovolnou kódovací metodou.



Obrázek 2.4: Strom reprezentující turnaj číselné posloupnosti.

Z dosud uvedených informací nelze zatím jednoznačně sestavit binární strom. Není možné rozhodnout, zda dekódované číslo zařadit jako levého či pravého potomka. To je velmi důležité, protože jediná změna ve stromu může vést ke ztrátě všech dat. Nejsnáze se to může vyřešit tak, že za kódované číslo je připojen jeden bit navíc, který bude signalizovat pozici vůči svému rodiči. Kódování posloupnosti P, kde číslo v kulaté závorce ohraničuje délku intervalu prefixových kódů a číslo v hranaté závorce určuje pozici vůči rodiči (0 - levá, 1 - pravá), bude vypadat následovně:

<5, 4(5)[0], 3(4)[1], 3(5)[1], 2(4)[1], 0(3)[0], 1(5)[1], 2(3)[0]>

Pokud pro uložení kořene použijeme Fibonacciho kód druhého řádu, kód bude poté vypadat následovně:

00011 10 0 10 1 011 1 01 1 00 0 001 1 10 0

Za povšimnutí stojí jeden rozdíl oproti statickému kódování - číslo není kódováno vždy stejně, ale může být reprezentováno více kódy. V tomto případě je to dobře vidět na čísle 3, které je nejprve zakódováno jako kód 10 a hned poté jako 011. Záleží totiž na ohraničujícím čísle, pro které byla vygenerovaná tabulka kódů.

2.7.2 Dekomprese

Dekomprese probíhá analogicky jako kódování: nejprve je dekódován kořen pomocí předem známého algoritmu. Ten je použit k omezení intervalu pro generování prefixových kódů, ze kterých je dekódován jeden z potomků. Dále musí být načten indikační bit, podle kterého je potomek připojen na správnou stranu. Druhým potomkem se stává rodič (v tomto případě kořen) a oba prvky jsou přidány do fronty. V dalších krocích je vždy vybráno číslo z fronty, které je následně použito k vygenerování tabulky kódů a podle ní je dekódováno další číslo. Podle načteného orientačního bitu je spolu s rodičem přidáno do fronty ve správném pořadí. Pokud je při dekódování nalezen nulový uzel, není potřeba generovat tabulku kódů, ale do fronty jsou přidáni dva nuloví potomci.

Stejně jako u aritmetického kódování, tak i u Tournament kódování je potřeba dopředu znát počet kódovaných znaků, aby algoritmus poznal, kdy se při dekódování zastavit. Proto je na začátek zprávy připojen počet kódovaných znaků zakódovaný univerzální kódovací metodou.

2.7.3 Vylepšení

Praktická část se zabývá problémy Tournament kódování a vhodným rozložením kódů (High-short, Low-short, Mid-short, ...), viz tabulka 2.5. Je navržen efektivnější způsob uložení pozice prvku vůči rodiči a jsou uvážena různá rozložení kódů pro listy a vnitřní uzly.

3 Praktická část

3.1 Implementace Tournament kódování

Dle zadání je cílem implementace kompresní metody, která je schopna zakódovat až 100 000 000 prvků. Je-li spočítána velikost paměti, kterou zabírá posloupnost právě s tímto počtem prvků, je získáno nemalé číslo M:

$$M = \frac{100\ 000\ 000 \cdot 4B}{1024 \cdot 1024} = \text{ 381.5 MB}$$

Tuto paměť tedy mohou zabrat samotná vstupní data. Zakódovaná data jsou ukládána do bytového streamu. Maximální velikost tohoto streamu je při nepravděpodobném, ale teoreticky možném, nulovém kompresním poměru také M. Jak bylo zmíněno v teoretické sekci, pro zakódování je použit binární strom. Ten je sestavován odspodu, ale dekódován je odshora. Pro kódování je tedy třeba mít sestavený strom a to znamená další přídavnou paměť pro jeho uložení. Listy stromu jsou rovny vstupním datům, která jsou již uložena. Pro uložení každého vyššího patra je potřeba polovina uzlů předchozího patra, a to v součtu zabere dalších M MB fyzické paměti.
Uložení posloupnosti, vytvoření stromu a uložení výsledných dat spotřebuje celkem $3M$ fyzické paměti, což je více než 1.1 GB. Pokud se k tomuto číslu připočte režie programu (zásobník, lokální proměnné, ...), celkové paměťové nároky se ještě zvětší. Pokud navíc bude program spuštěn na víceúlohovém (preemptivním) operačním systému, kde mohou běžet i jiné programy, je zde velká pravděpodobnost odkládání fyzické paměti na disk a to znamená velké zpomalení programu.
Kvůli velké paměťové náročnosti popsané v předchozím odstavci je program naimplementován tak, že si vystačí pouze se vstupním polem prvků a výstupním streamem, do kterého ukládá výsledný kód. Zvolené řešení nevyužívá žádnou přídavnou paměť pro uložení stromu.

3.1.1 Uložení stromu

Jelikož není možné uchovat celý strom v paměti, musí si algoritmus vystačit pouze s jedním patrem stromu. Patro stromu reprezentuje číselnou posloupnost, na kterou může být aplikován turnaj. Prohrávající čísla se ovšem musí

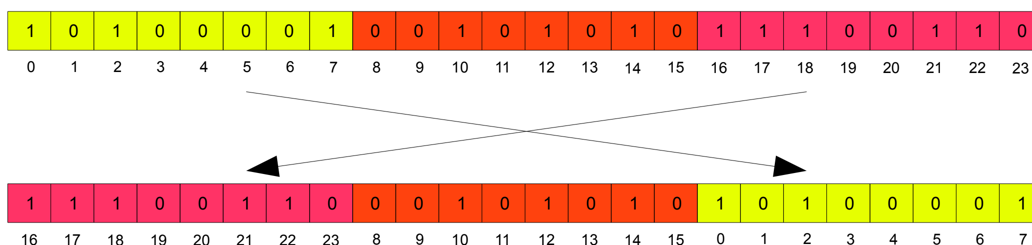
ihned zapsat do výstupního streamu, jinak dojde k jejich ztrátě. Pokud se dvojice v rámci patra začnou kódovat zleva a kódy prohrávajících čísel jsou ukládány přímo do bitového streamu, po skončení algoritmu vznikne ve streamu zvláštní reprezentace stromu. Pro jeho zpětnou rekonstrukci je totiž třeba číst data ze streamu odzadu a navíc jsou prvky v rámci patra načítány zprava. Proto je výhodnější kódovat dvojice při turnaji zprava a kódy prohrávajících čísel rovnou ukládat do bitového streamu. Po skončení algoritmu vznikne výhodnější reprezentace stromu, u které je ale pořád nezbytné číst ze streamu odzadu. To není optimální řešení a navíc znamená uložení dalších informací do streamu - na začátku streamu by muselo být uloženo další číslo udávající pozici, od které začít číst. Pro malé posloupnosti by to znamenalo zhoršení kompresního poměru.
Řešením je přeuspořádání dat tak, aby se informace potřebné k sestavení stromu daly číst od začátku streamu. Změna pořadí bytů ve streamu, po které je možno číst data v obráceném pořadí, než byla vložena, se mění podle předpisu:

$$p'_i = p_1 + b - p_i - 1, \text{ kde}$$

b	počet bytů ve streamu,
p_i	původní pozice i-tého prvku,
p'_i	nová pozice i-tého prvku.

Výsledkem uvedené operace je pouze prohození pořadí všech vložených bytů. Ve vzorci se objevuje pozice p_1, díky které se pracuje pouze s byty uloženými při kompresi - stream totiž nemusí být nutně prázdný, ale mohou se v něm nacházet ještě jiná data. Díky úpravám pozic jednotlivých bytů je program nyní schopen načítat kódy ze streamu v opačném pořadí, než byly vloženy, a z načtených hodnot stavět strom. Čtení dat ovšem musí být přizpůsobeno použitému uložení. Situace je objasněna na příkladu, pro který je popsán princip načítání dat ze streamu.

Příklad: Do streamu jsou uložena 3 binární čísla 1010000100, 1010101110 a 0110. Pořadí jednotlivých bytů je poté zaměněno podle výše zmíněného přepočtu. Nastíněnou situaci zobrazuje obrázek 3.1.



Obrázek 3.1: Přeskupení bytů ve streamu

Předpokládá se, že před každým čtením ze streamu je znám počet bitů, které je potřeba načíst - pro předchozí příklad tedy 4, 10 a 10.
Pro první načítané číslo, které je uloženo na 4 bitech, je situace jednoduchá. Celý kód je obsažen v jednom bytu a je roven posledním čtyřem bitům prvního bytu. Jelikož se do streamu zapisovalo zleva, po přeskupení je nutné načítat jednotlivé kódy zprava. První číslo je tedy:

$$c_1 = \{20, 21, 22, 23\} = 0110$$

U druhého čísla je situace o trochu složitější. Číselný kód, který bude načten, totiž zasahuje do více bytů ve streamu. V prvním bytu streamu ještě zbývají 4 nepřečtené bity, z druhého bytu je tedy třeba načíst $10 - 4 = 6$ bitů. Nejprve je třeba přečíst 6 bitů zprava z druhého bytu a k nim přidat 4 zbývající bity z prvního bytu. Tím je získáno druhé číslo:

$$c_2 = \{10, 11, 12, 13, 14, 15\} + \{16, 17, 18, 19\} = 1010101110$$

Třetí číslo je načteno analogicky jako číslo druhé. V druhém bytu zbývají 2 bity, z třetího se tedy načte $10 - 2 = 8$ bitů. Vždy se začíná od nejvyššího bytu, tedy třetí číslo je:

$$c_3 = \{0, 1, 2, 3, 4, 5, 6, 7\} + \{8, 9\} = 1010000100$$

Pro čtení ze streamu platí, že se vždy zpracovávají jednotlivé byty od vyšší pozice a jednotlivé bity z nich se vybírají zprava. Načtené skupiny bitů se skládájí za sebe a po načtení a poskládání všech skupin bitů je získán výsledný kód, který je poté reprezentován jako číslo. Po načtení všech čísel ze streamu je získána originální posloupnost pouze s opačným pořadím, než ve kterém byla jednotlivá čísla původně zakódována.
U předchozího příkladu bylo předpokládáno, že před každým čtením ze streamu je znám počet načítaných bitů. To ale není úplně pravda. Jediné, co program zná, je větší číslo z kódované dvojice, ze kterého byla vygenerována tabulka kódů, a právě jeden z kódů patřící do této tabulky je při následujícím čtení dekódován. Tabulka obsahuje vždy maximálně dvě různé délky kódů

lišící se o jeden bit. Nejprve je ze streamu načten počet bitů rovný délce kratšího kódu a je zjištěno, zda je načtený kód obsažen v tabulce kódů. Pokud není, jedná se o delší kód a je potřeba načíst ještě zbývající jeden bit.

3.1.2 Komprese a dekomprese

Před samotnou kompresí je posloupnost celých čísel převedena na posloupnost přirozených čísel (je dovolena i nula). Pro tento postup je použita transformace na sudá a lichá nezáporná čísla, viz teoretická sekce. Toto řešení bylo zvoleno, protože nevyžaduje uložení nadbytečných dat do výstupního streamu. Upravenou posloupnost již lze zakódovat pomocí Tournament kódování.
Jak již bylo zmíněno výše, algoritmus si udržuje v paměti pouze jedno patro stromu, na které je aplikován algoritmus turnaje. Ten rozdělí prvky do dvojic a prohrávající prvky zakóduje do streamu za pomoci vítězných prvků. Dvojice jsou zpracovávány zprava (odzadu) kvůli zpětné rekonstrukci stromu. Pokud patro obsahuje lichý počet prvků, poslední číslo je spárováno s nulovým prvkem. Toto ulehčení má za následek přidání několika nul na konec posloupnosti, původní data ale zůstanou nezměněna. Před samotným dekódováním je kromě největšího prvku znám ještě celkový počet kódovaných čísel, a proto přidané nuly do výsledku nezasáhnou. Po zakódování poslední dvojice je nezbytné uložit maximum posloupnosti a počet zakódovaných čísel. Obě tato čísla jsou uložená pomocí předem zvolené univerzální metody.

Před samotným dekódováním stromu jsou ze streamu načtena dvě čísla, která určují počet čísel posloupnosti a maximální prvek. Tato čísla jsou uložena pomocí známého algoritmu.
Strom je stavěn s použitím datové struktury *fronta*, do které je na začátku algoritmu přidán kořen stromu - maximální prvek. Algoritmus poté vždy vybere první číslo z fronty, zjistí možné délky kódu s proměnlivou délkou a vrátí dekódované číslo, které spolu s rodičem přidá na konec fronty. Po dekódování prvku je třeba určit, zda prvek zařadit jako levého či pravého potomka. V teoretické sekci se za každé číslo přidal indikační bit, který nesl informaci o pozici vůči rodiči. Tento způsob indikace ovšem není optimální, podrobněji se tomuto problému věnuje následující kapitola. Druhým potomkem se vždy stává rodič a správně seřazení potomci jsou přidáni do fronty. Algoritmus končí, obsahuje-li fronta takový počet prvků, který byl na začátku načten jako délka posloupnosti. Celá posloupnost poté musí být převedena zpět na posloupnost celých čísel, která byla předána metodě k zakódování. Liché prvky posloup-

nosti jsou proto vyděleny číslem -2, sudé číslem 2. Z výsledných čísel se vezme celá část a tímto způsobem je získána původní kódovaná posloupnost.

3.1.3 Ukládání pozice prvku

V teoretické části byl za prefixový kód přidáván indikační bit, který označoval pozici ve stromu vůči svému rodiči. Každému menšímu číslu z intervalu vymezeném větším čísel byl tedy přidán jeden indikační bit bez ohledu na hodnotu čísla. Tady ovšem vznikla malá redundance, která nastane při rovnosti obou čísel. V takovém případě totiž existují dva kódy pro jeden případ. Pro lepší pochopitelnost je problém uveden na příkladu.
Mějme zakódovat číslo 3 a číslo x, kde $x \in \{0, 1, 2, 3\}$. Číslo 3 bude vždy vyhrávající, a proto bude použito k vygenerování prefixových kódů, viz tabulka 3.1.

Tabulka 3.1: Prefixové kódy pro číslo 3 - indikace bitem

číslo	**prefixový kód**	**kód s pozicí**	**význam**
0	00	000	0 3
		001	3 0
1	01	010	1 3
		011	3 1
2	10	100	2 3
		101	3 2
3	11	110	3 3
		111	3 3

Z tabulky 3.1 je zřejmé, že dvojici čísel (3, 3) jsou přiřazeny dva kódy, které označují stejnou situaci. Tento speciální případ stačí kódovat jedním kódem, v tomto případě originálním kódem 11.
Implementované řešení, které je uvedeno také v [1], je použito právě pro odstranění tohoto speciálního případu. V předchozím příkladu je třeba umět prefixově zakódovat 7 čísel pro jednoznačnou identifikaci pozice ve stromu. Lze tedy vygenerovat tabulku prefixů pro délku intervalu 7, čímž je získáno 7 prefixových kódů minimální možné délky.
Levé číslo z kódované dvojice je označeno L a číslo napravo R. Menší číslo se zakóduje podle následujících pravidel:

$$\begin{array}{ll} 2 \cdot R & \text{pokud } L \geq R, \\ 2 \cdot L + 1 & \text{pokud } L < R. \end{array}$$

Se znalostí těchto pravidel je nyní možné z jednoho kódu jednoznačně identifikovat jak hodnotu, tak pozici čísla vůči svému rodiči.

Tabulka 3.2: Úspornější uložení pozice

číslo	**prefixový kód**	**význam**
0	000	3 0
1	001	0 3
2	010	3 1
3	011	1 3
4	100	3 2
5	101	2 3
6	11	3 3

Jak je vidět z tabulky 3.2, v případě rovnosti dvou čísel je ušetřen jeden bit pro identifikaci pozice. Pro dekompresi čísla stačí zjistit paritu kódu, která skrývá pozici (lichá parita = číslo zařadit doleva, sudá parita = číslo zařadit doprava). Pro zjištění hodnoty je třeba zakódované číslo vydělit dvěma a celá část výsledného podílu je rovna původnímu číslu.
V případě rovnosti dvou čísel je tedy ušetřen jeden bit. Následkem je vliv na rozsah kódovaných čísel.

3.1.4 Rozsah kódovaných čísel

Dvojnásobným zvětšením kódovaného čísla a možným zvětšením o jedna je totiž omezen rozsah kódovaných čísel. Základní verze Tournament kódování (verze s indikačním bitem) je schopna zakódovat čísla typu integer z intervalu
$<0;\ 2^{31}-1>$. Kvůli indikaci pozice vůči rodiči se kódované číslo dvakrát zvětší, tzn. že interval se o polovinu zmenší: $<0;\ 2^{30}-1>$.
Jelikož metoda dokáže zakódovat pouze kladná čísla, celá posloupnost se musí přepočítat na čísla z tohoto intervalu. Postup pro převod na nezápornou posloupnost je popsán v teoretické části a kvůli dvojnásobnému zvětšení hodnot bude algoritmus fungovat pouze pro čísla z intervalu $<-2^{29};\ 2^{29}-1>$.

Tímto intervalem jsou tedy omezeny všechny prvky posloupnosti, které mají být zakódovány pomocí Tournament kódování.

3.1.5 Rozložení kódovací tabulky

Stejně jako Tournament kódování, tak i interpolativní kódování [6] využívá pro zakódování čísel proměnlivou délku kódů. V různých fázích kódování je u interpolativního kódování použito jiné rozložení těchto kódů k získání co nejlepšího kompresního poměru. Proto se také u Tournament kódování testovalo použití různých rozložení pro různé fáze.
Až do teď byla všechna čísla kódována pomocí High-short rozložení. To znamená, že větší čísla byla kódována kratším kódem. To je pochopitelné alespoň u rovnoměrného rozložení čísel, kde proti sobě soupeří lokální maxima posloupnosti, a lze tedy předpokládát, že se bude jednat o větší hodnoty. V [1] je uvedeno, že experimenty tuto domněnku potvrdily, a proto je High-short rozložení použito v Tournament kódování.
Dále je v [1] uvedeno, že pro zakódování listů stromu libovolného rozložení je výhodnější použít Low-short rozložení. Kódování listů Low-shortem přebírá Tournament kódování od [6], u kterého byly prováděny experimenty s různými rozloženími, a právě tato kombinace měla nejlepší úspěšnost.

Listy jsou tedy kódovány pomocí Low-short rozdělení, ostatní uzly jsou kódovány pomocí High-short rozdělení. Je tedy třeba přizpůsobit implementaci této myšlence.
Kódování je jednoduché - jako první jsou zpracovávány listy stromu (originální posloupnost), které se zakódují s použitím Low-short rozdělení. Dále jsou všechny ostatní uzly zakódovány s použitím High-short rozdělení. Zásah do algoritmu je téměř nulový, pouze se ukládájí jiné kódy do výstupního streamu.
Dekódování čísel s různými rozloženími už je o něco obtížnější. Je totiž třeba správně určit, kdy už se nenačítají vnitřní uzly, ale listy stromu. Pokud by se algoritmus spletl i jen o jedno číslo, výsledkem dekomprese by nebyla původní posloupnost, ale znehodnocená data.
Jak tedy v programu poznat, že se jedná o listy stromu? Řešením je využití jednoduchosti binárního stromu, který má následující vlastnosti. Je-li binární strom úplný, tzn. že z každého vnitřního uzlu vycházejí dva potomci, pak počet prvků patra $n - 1$ je o polovinu menší než počet prvků patra n. Dále je pro úplný strom počet prvků libovolného patra roven mocnině čísla 2.
Pro ukládání prvků stromu je použita fronta, která v průběhu dekódování

obsahuje vždy jen podmnožinu vnitřních uzlů a na konci dekódování pouze množinu listů. V průběhu dekódování je znám celkový počet kódovaných čísel a s jeho znalostí lze určit, kolik prvků obsahuje patro nad ním. Stačí tedy kontrolovat počet prvků ve frontě a při správném počtu prvků, ze kterého je identifikováno uložení celého předposledního patra, je změněno rozložení kódovací tabulky na Low-short.
Při lichém počtu prvků je v průběhu kódování k poslednímu číslu přiřazena nula zprava a tato dvojice je poté zakódována. Tato aplikace u všech pater má za následek, že při dekódování je vždy vytvořen kompletní strom, který na chybějících pozicích obsahuje vložené nuly. Tím je počet listů stromu roven již zmíněné mocnině 2 a tedy stačí z původního počtu zjistit nejbližší nižší mocninu dvou. Její hodnota je rovna počtu prvků v předposledním patře.

3.2 Srovnání metod

V této kapitole je porovnáno Tournament kódování se všemi výše popsanými kompresními metodami. Pro měření jsou použita data s rovnoměrným, normálním, exponenciálním a Laplaceovým rozložením. Testy jsou prováděny na datech s různým rozsahem hodnot. Pro vybrané délky posloupností je měněn rozptyl pro získání vhodných entropií. Měření je cíleno na výslednou kompresi dat, která je porovnána vůči vstupní entropii, a také na časy komprese a dekomprese.

3.2.1 Rovnoměrné rozdělení

První měření je provedeno pro posloupnosti s rovnoměrným rozdělením prvků. U tohoto rozdělení je pravděpodobnost výskytu všech hodnot z daného intervalu stejná. Měření je realizováno pro tři různé délky. Výsledek měření je zaznamenán v tabulce 3.3, kde hodnoty ve sloupcích s názvy metod značí celkový počet bitů potřebný pro zakódování jednoho znaku. Každé měření bylo opakováno desetkrát a průměrná hodnota z těchto opakování byla poté prohlášena za výsledek.

Tabulka 3.3: Výsledky: rovnoměrné rozdělení

počet hodnot	**rozsah**	**entropie**	**Tournament**	**Huffman**	**aritmetické**	**Fibonacci**
1 000	10	3,453	3,987	3,656	3,889	4,320
	100	6,587	7,133	7,606	7,946	8,694
100 000	100	6,657	7,062	6,732	7,585	8,711
	1000	9,959	10,403	10,092	12,161	13,416
	10 000	13,215	13,759	14,843	17,701	18,231
10 000 000	1 000	9,967	10,403	9,978	12,157	13,421
	100 000	16,602	17,018	16,844	22,777	23,036
	1 000 000	19,858	20,351	22,088	27,668	27,821

Z naměřených výsledků je na první pohled zřejmé, že Huffmanovo kódování podává nejlepší výsledky komprese pro posloupnosti s malým rozsahem hodnot. Počet bitů potřebných k zakódování jednoho znaku je v tomto případě velmi blízko vstupní entropii. Naopak pro větší rozsah hodnot už nejsou výsledky tak dobré a v tomto případě dominuje Tournament kódování. Ostatní kompresní metody podávají dobré výsledky pro data s malou entropií, pro vyšší entropii už ale kompresní poměr není dobrý. Celkově lze tedy říci, že pro data s rovnoměrným rozdělením hodnot je vhodné použít buď Tournament kódování nebo Huffmanovo kódování. Tournament kódování podává stabilní výsledky pro všechny měřené intervaly a rozsahy, kompresní poměr je ale pro malé rozsahy hodnot horší než u Huffmana.

3.2.2 Normální rozdělení

Druhé měření je provedeno na datech s normálním rozdělením. Toto rozdělení je spojité a symetrické kolem střední hodnoty, která se značí μ. Dalším parametrem je rozptyl hodnot, který se značí σ.
Hustota pravděpodobnosti normálního rozdělení má zvonovitý tvar a v nejvyšším bodu tohoto zvonu se nachází střední hodnota. Jelikož náhodná veličina řídící se normálním rozdělením může nabýt téměř libovolných hodnot, je třeba dát si pozor na záporná čísla, která Tournament a Fibonacciho kódování neumí zpracovat. Posloupnost tedy musí být před samotným kódováním převedena na nezápornou posloupnost. Pro převod je zvoleno posunutí všech čísel o minimální hodnotu, viz teoretická sekce. Toto jednoduché posunutí hodnot zachová původní rozdělení prvků a výsledky měření odpovídají původním datům.

Pro měření byla použita střední hodnota $\mu = 0$ a rozptyl σ byl obměňován k získání různých entropií pro požadovaný interval.

Tabulka 3.4: Výsledky: normální rozdělení

počet hodnot	**entropie**	**Tournament**	**Huffman**	**aritmetické**	**Fibonacci**
1 000	2,127	3,71	2,307	2,32	4,499
	5,617	6,848	6,357	6,32	8,868
	8,519	10,082	13,878	10,15	13,518
100 000	5,685	7,082	5,721	6,039	9,547
	9,056	10,422	9,216	9,801	14,288
	12,326	13,789	13,521	14,806	19,132
	15,17	17,038	23,583	20,101	23,83
10 000 000	9,062	10,568	9,118	9,794	14,554
	12,383	13,822	12,559	14,8	19,205
	15,534	16,926	16,847	19,914	23,852
	18,327	20,309	27,486	24,778	28,582
	19,688	23,679	42,656	29,66	33,476

Huffmanovo a aritmetické kódování jsou vhodná pro kompresi posloupností s menší a střední entropií pro všechny měřené rozsahy hodnot. Se zvětšující se entropií zároveň stoupá počet různých hodnot a výsledky se postupně zhoršují. S velkým rozsahem hodnot naopak nemá problémy Tournament kódování, které právě pro posloupnosti s velkým rozsahem hodnot vrací nejlepší výsledky. To je nejvíce znatelné na posloupnostech s největším měřeným počtem hodnot. Pro posloupnosti s menší a střední entropií jsou naměřené výsledky Tournament kódování o něco horší než výsledky statistických metod, jedná se ale o malý rozdíl.

3.2.3 Exponenciální rozdělení

Třetí měření je provedeno na datech s exponenciálním rozdělením. Toto rozdělení je spojité a je definované pouze pro nezáporné hodnoty. U tohoto rozdělení je největší pravděpodobnost výskytu nejmenších hodnot, naopak čím větší hodnota, tím menší pravděpodobnost jejího výskytu. Jediným vstupním parametrem rozdělení je λ, která určuje rychlost konvergence exponenciály a samozřejmě také ovlivňuje pravděpodobnosti výskytů jednotlivých hodnot.

Pro měřené délky posloupností je hodnota tohoto parametru měněna k získání různých entropií.

Tabulka 3.5: Výsledky: exponenciální rozdělení

počet hodnot	**entropie**	**Tournament**	**Huffman**	**aritmetické**	**Fibonacci**
1 000	2,762	3,039	2,996	3,002	2,941
	5,27	5,656	6,045	5,986	5,801
	7,759	8,467	11,486	9,118	9,644
100 000	5,572	5,795	5,621	5,903	6,09
	8,147	8,377	8,274	8,752	9,667
	11,084	11,355	11,822	12,959	13,93
	13,509	13,94	16,739	17,041	17,652
10 000 000	7,934	8,156	7,965	8,480	9,356
	10,515	10,741	10,611	12,028	13,055
	14,158	14,385	14,284	17,739	18,296
	16,573	16,972	19,969	21,727	22,024
	20,284	20,734	25,095	27,309	27,438

Podle naměřených výsledků si s měřenými posloupnostmi nejlépe poradilo Tournament kódování, a to dokonce i pro malé entropie, u kterých je rozdíl oproti Huffmanovo kódování téměř minimální. Pro malé a střední entropie, pro které počet různých hodnot v posloupnostech ještě není značný, jsou výsledky Tournament, Huffmanova a aritmetického kódování téměř shodné. Výjimkou jsou posloupnosti s vysokou entropií, u kterých se výsledky jednoznačně přiklánějí k Tournament kódování.

3.2.4 Laplaceovo rozdělení

Čtvrté měření je provedeno na datech s Laplaceovo rozdělením. Toto rozdělení je spojité a symetrické kolem střední hodnoty, která se značí μ. Dalším parametrem je rozptyl, který se značí b. Pravděpodobnostní funkce je vlastně zrcadlené exponenciální rozdělení. Pravděpodobnost výskytu hodnot vzdalujících se od střední hodnoty, tedy stejně jako u exponenciálního rozdělení, klesá exponenciálně. U exponenciálního rozdělení se to týkalo jen kladných hodnot, u Laplaceova rozdělení jsou tyto hodnoty i záporné. Čím je vzdálenost od střední hodnoty větší, tím je pravděpodobnost výskytu prvku menší. Jelikož hodnoty mohou nabývat i záporných čísel, měřená data jsou pro Tour-

nament a Fibonacciho kódování posunuta stejným způsobem jako u normálního rozdělení. Pro měření byla použita střední hodnota $\mu = 0$ a rozptyl b byl obměňován k získání různých entropií pro požadovaný interval.

Tabulka 3.6: Výsledky: Laplaceovo rozdělení

počet hodnot	**entropie**	**Tournament**	**Huffman**	**aritmetické**	**Fibonacci**
	2,91	5,072	3,19	3,134	6,654
1 000	4,685	6,416	5,286	5,128	8,408
	6,131	7,845	7,434	6,882	10,495
	4,587	6,941	4,638	4,76	9,494
	6,282	8,564	6,338	6,618	11,881
100 000	8,059	10,404	8,182	8,519	14,354
	11,353	13,555	12,21	12,883	18,946
	12,854	15,309	15,065	15,32	21,477
	8,066	10,552	8,105	8,517	14,924
	9,66	12,33	9,729	10,411	17,001
10 000 000	14,677	17,354	15,694	18,075	24,11
	16,178	18,676	18,776	20,554	26,743
	18,776	22,054	32,215	25,571	31,13

Z naměřených hodnot pro nevelké entropie lze vypozorovat větší rozdíl mezi výsledky Tournament kódování a statistickými metodami. Statistické metody si počínají velmi dobře pro většinu měřených posloupností, výjimku tvoří posloupnosti s největší naměřenou entropií, kterým opět vévodí Tournament kódování.

3.2.5 Zhodnocení výsledků

Testy byly provedeny na posloupnostech různé délky. Toto rozdělení se během měření ukázalo jako rozumné, protože měřené metody podávaly pro různé délky odlišné výsledky. Příkladem je Huffmanovo kódování, které pro malé rozsahy hodnot podávalo velmi dobré výsledky. U posloupností s větším rozsahem hodnot se ale výsledky vzdalují od entropie.
Velkou výhodou Tournament kódování je jednoduchost algoritmu a schopnost rychle a efektivně zakódovat obrovské posloupnosti. To je vidět z tabulky 3.10, která zachycuje časovou náročnost komprese jednotlivých metod. Časy komprese jsou pro měřená rozdělení velmi podobné, na délku běhu tedy

nemá rozdělení prvků vliv. Pro zobrazení časové náročnosti byla zvolena data s rovnoměrným rozdělením.

Tabulka 3.7: Výsledky: naměřené časy komprese

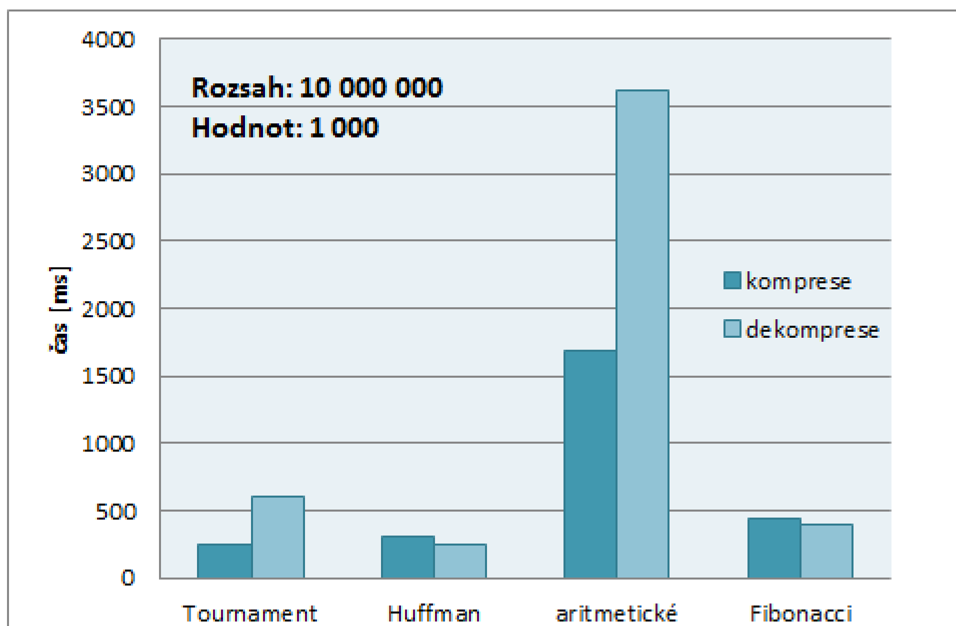
počet hodnot	**rozsah**	**Tournament** [ms]	**Huffman** [ms]	**aritmetické** [ms]	**Fibonacci** [ms]
	100	19,4	22,8	121,8	42,7
100 000	1 000	23,6	33	250,8	48,5
	10 000	27,4	49,2	347,8	56,4
	1 000	242	303	1690	442
10 000 000	100 000	332	687	2403	558
	1 000 000	373	2272	4530	604

Nejlepších časů komprese dosahuje Tournament kódování a to pro všechny měřené posloupnosti. Naopak nejvíce časově náročné je aritmetické kódování.

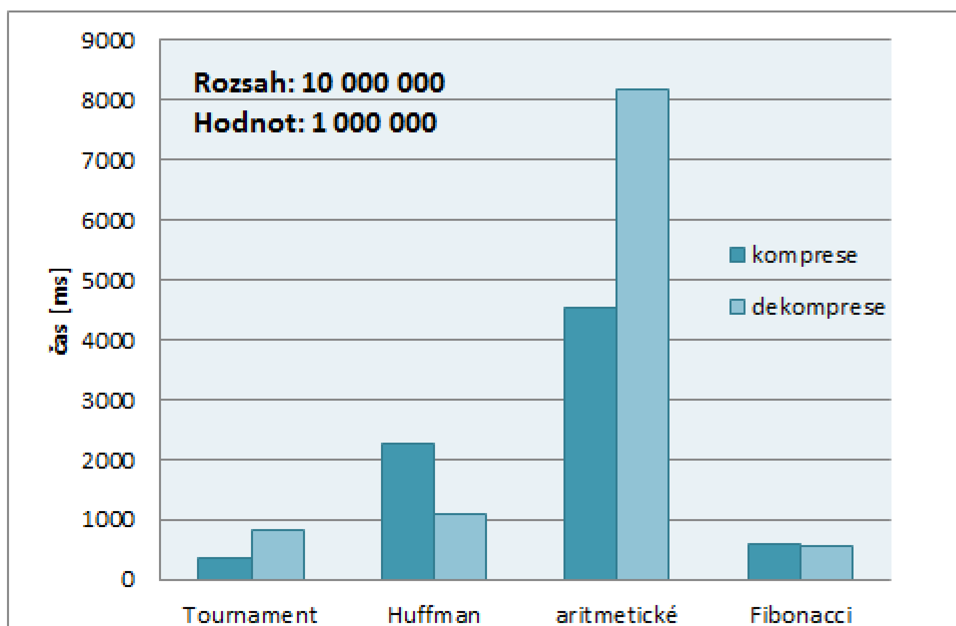
Tabulka 3.8: Výsledky: naměřené časy dekomprese

počet hodnot	**rozsah**	**Tournament** [ms]	**Huffman** [ms]	**aritmetické** [ms]	**Fibonacci** [ms]
	100	19,4	22,8	121,8	38,6
100 000	1 000	60	26,4	670,6	42
	10 000	67	38,6	1112,2	48,4
	1 000	598	256	3623	392
10 000 000	100 000	703	580	5297	484
	1 000 000	820	1090	8191	557

Tabulka 3.11 zachycuje časy dekomprese u vybraných posloupností s rovnoměnrným rozdělením. Nejlepších časů dosahuje Huffmanovo kódování, ovšem pro největší měřený rozsah hodnot je nejrychlejší Tournament kódování. Časy aritmetického kódování jsou opět nejvyšší.



Obrázek 3.2: Časy běhů (1 000 hodnot)



Obrázek 3.3: Časy běhů (1 000 000 hodnot)

4 Závěr

Cílem této práce byla implementace Tournament kódování a porovnání jeho výsledků s vybranými kompresními metodami.
V porovnání s ostatními kompresními metodami dosahuje Tournament kódování nejzajímavějších výsledků pro posloupnosti s velkým rozsahem a počtem prvků. Na těchto datech je algoritmus turnaje velmi stabilní a výstupní entropie je ze všech metod nejlepší pro všechna měřená rozdělení. Naopak u posloupností s malým počtem hodnot jsou výsledky o něco horší a zde dominují statistické metody. Výjimkou je exponenciální rozdělení, u kterého jsou výsledky statistických metod a Tournament kódování velmi blízké i pro malé entropie. U posloupností s normálním a rovnoměrným rozdělením jsou výsledky pro střední rozsah hodnot téměř totožné s výsledky statistických metod.
Tournament metoda je srovnatelná se statistickými metodami a podává stabilní výsledky pro všechny měřené posloupnosti čísel. Časy komprese jsou nejrychlejší ze všech porovnávaných metod. U dekomprese jsou o něco horší než u Huffmanova kódování, jedná se ale o nepatrné rozdíly. Naimplementovaná metoda by mohla být použita například pro kódování invertovaných souborů, u kterých je právě rozsah prvků a hodnot značný.
Z mého pohledu byla práce velice poučná a přínosná, a to z hlediska získání teoretických a praktických poznatků týkajících se komprese. Naimplementovaná metoda je určena pouze pro testovací účely a jako autor se zříkám veškeré zodpovědnosti při použití k jiným účelům.

Literatura

[1] Teuhola, Jukka *Tournament coding of Integer sequences*,
The Computer Jurnal, Vol. 52 No. 3 (2009)

[2] Sklenák, V. a kol. *Data, informace, znalosti a Internet*
1. vydání, Praha : C. H. Beck 2001.

[3] Večerka A. *Komprese dat*
Olomouc, 2008

[4] T. Hrabálek, *Podpora vytváření virtuálních topologií ve virtuální laboratoři počítačových sítí s využitím technologie tunelování*, diplomová práce, VŠB-TU FEI, Ostrava, 2007

[5] P. Němec, *Virtuální sít'ová laboratoř*, diplomová práce, VŠB-TU FEI, Ostrava, 2005

[6] Moffat, A. and Stuiver, L. *Binary interpolative coding for effective index compression*,
Inf. Retr., 3, 25-47 (2005)

[7] Leonardo of Pisa (Fibonacci) *Liber Abaci*,
(1202)

[8] Walder, J. and Krátký, M. and Platoš J. *Fast Fibonacci Encoding Algorithm*,
http://ceur-ws.org/Vol-567/paper14.pdf

[9] D. A. Huffman, *A method for the construction of minimum redundancy codes*, Proc. IRE, vol. 40, pp.1098 -1101 1952

Seznam obrázků

2.1 Základní struktura invertovaného souboru 3

2.2 Huffmanovo strom pro zakódování textu ABRAKADABRA . 13

2.3 Rozdělení intervalu pro aritmetické kódování 14

2.4 Strom reprezentující turnaj číselné posloupnosti. 19

3.1 Přeskupení bytů ve streamu 23

3.2 Časy běhů (1 000 hodnot) 34

3.3 Časy běhů (1 000 000 hodnot) 34

Seznam tabulek

2.1 Entropie jevů 5

2.2 Entropie jevů 6

2.3 Kód s pevnou délkou 7

2.4 Kódy s proměnlivou délkou. 8

2.5 Kód s proměnlivou délkou 8

2.6 Přiřazení kódů s proměnlivou délkou číslům {0,. . . ,5} 9

2.7 Fibonacciho kódy řádu 2 pro čísla {1, . . . , 10} 10

2.8 Rozdělení intervalů 15

2.9 Komprese aritmetického kódování pro řetězec *abac*. 15

2.10 Dekomprese aritmetického kódování pro řetězec *abac*. 16

3.1 Prefixové kódy pro číslo 3 - indikace bitem 25

3.2 Úspornější uložení pozice 26

3.3 Výsledky: rovnoměrné rozdělení 29

3.4 Výsledky: normální rozdělení 30

3.5 Výsledky: exponenciální rozdělení 31

3.6 Výsledky: Laplaceovo rozdělení 32

3.7 Výsledky: naměřené časy komprese 33

3.8 Výsledky: naměřené časy dekomprese 33