Fulltextové systémy - rešerše

Komprese dat

Lukáš Čejka

4. května 2021

Obsah

0 Úvod

V této rešerši je uvedena problematika komprese dat, konkrétně textu. V první části budou popsány základní pojmy z oblasti komprese textu a v druhé části budou popsány vybrané základní algoritmy pro kompresi dat.

Počátkem 21. století se postupně začalo svět digitalizovat, například Lopez a Hilbert ve své publikaci *The World's Technological Capacity to Store, Communicate, and Compute Information*[1] uvádějí jako počátek digitální éry rok 2002. S počátkem této digitální éry se začalo zaznamenávat mnohem více informací, např. v mobilních zařízení, notebooků, softwarových logů, cookies a informacích o uživatelích, apod.
S nárůstem zaznamenaných dat se zvětšili i úložné možnosti, ale ne postačujícím způsobem. Dále se objevil problém

s přesunem dat - pokud zůstanou v původní nezpracované podobě (raw data), tak by se přes síť pohybovali pomaleji a celkově práce s nimi by byla náročnější. Těmto problémům se dá předejít s pomocí komprese dat.

Počátky komprese dat jsou datovány již v polovině 20. století, kde různé techniky se už používali, ale nebyli zdaleka tak rozšířené. Například dnes se v praxi při přenosu větších souborů přes internet často využívají archivní soubory formátu `.zip`, `.rar`, apod.

Komprese dat je specifický způsob kódování, který má za cíl vzít původní data a zmenšit jejich velikost. Tohoto se obecně snaží docílit pomocí odstraňování nadbytečných informací, či ušetření místa tím, že opakující informace se uloží pouze jednou a pak na pozicích kde se ty data vyskytují vloží ukazatel na hodnotu oněch dat. Konkrétní postup komprimování dat se nazývá kompresní algoritmus, který musí obsahovat jak postup pro kompresi dat, tak postup pro dekompresi dat - získání původních dat z komprimovaných.

1 Základní pojmy v oblast komprese dat

Pro ulehčení práce v oblasti komprese dat je vhodné definovat výběr základních pojmů [2].

Kódování Způsob, jakým jsou data reprezentována v souboru, ať už v paměti, při přenosu po síti apod.

Znak Základní jednotka dat, která může mít podobu znaku v textu nebo hodnoty, která je v paměti uložená právě na jednom bytu.

Text V rámci komprese dat pod pojmem text je myšlen jak lidmi srozumitelný text, tak i libovolná komprimovaná výchozí data.

Kompresní poměr Nechť je vstupní soubor do procedury komprese dat o velikosti 20 MB. Dále nechť kompresní algoritmus sníží jeho velikost na 50% původní velikost, tj. na 10 MB. Formálně je kompresní poměr zadefinován následně [2]:

$$\text{Kompresní poměr} = \frac{\text{Délka po kompresi}}{\text{Délka před kompresí}} \tag{1}$$

Velice podobný způsob označování kompresního poměru výše zmíněného příkladu je kompresní poměr 2:1 [3].
Mezi další možné způsoby značení patří **bpb** ("*bits per byte*", neboli průměrný počet bitů na byte) - jinými slovy, při kompresi se vyjadřuje průměr bitů potřebných k uložení jednoho bytu výchozího souboru. Například, mějme kompresní poměr 0.4, který podle vzorce 1 bude vyjádřen jako $\frac{4}{10}$. Jelikož 1 byte má 8 bitů, tak bude **bpb** $= 0.4 \cdot 8 = 3.2$, což znamená, že pro každý 1 byte (8 bitů) vstupního souboru, je v komprimovaném souboru potřeba v průměru pouze 3.2 bitů.

1.1 Bezeztrátová a ztrátová komprese

V rámci komprese dat se rozlišují algoritmy podle svého postupu na bezeztrátové a ztrátové (tj. bezeztrátová a ztrátová komprese).
Algoritmy zprostředkovávající **bezeztrátovou kompresi** komprimují data takovým způsobem, že nedojde k žádné ztrátě informací. Jinými slovy, umožňují přesně zpětně zrekonstruovat komprimovaná data. Na druhé straně, **ztrátové algoritmy** při kompresi trvale eliminují méně důležitá data, čímž ale dochází ke ztrátě informací, které se z komprimovaného souboru již nedají zrekonstruovat.
Ztrátová komprese je užitečná u grafiky, zvuku a videa, kde odstranění některých datových bitů má malý nebo žádný zřetelný účinek na reprezentaci obsahu [3]. Například formáty pro ukládání grafických informací jsou obvykle navrženy na základě jakési kompresi informace, jelikož čistá data, neboli raw data (fotky, videa, apod.) např. z digitální kamery zabírají velké množství úložného místa a lidské oko/ucho všechny tyto informace nepotřebuje - velmi zřídka pozná rozdíl mezi původním a komprimovaným souborem. Mezi nejznámější formáty pro ukládání obrázků patří např. JPEG, který používá ztrátovou kompresi, GIF a PNG, které používají bezeztrátovou kompresi [4][5].

1.2 Typy kompresních metod

V *Komprese dat* od Arnošta Večerky [2] je popsáno 5 druhů kompresních metod:

1. **Statické metody komprese** - metody zakládající na pravděpodobnosti výskytu jednotlivých znaků v textu.
2. **Slovníkové metody komprese** - metody vyhledávající části v textu, které se opakují. První výskyt každé části bude obsahovat stejná data jako ve vstupním souboru, ale každý další výskyt stejné části už bude nahrazen odkazem na tu první část.

3. **Komprese nepohyblivého obrazu** - metody používající ztrátovou kompresi ke komprimování bitmapové grafiky (tj. obrázky uložené pomocí sousedních bodů s barevnými hodnotami - pixely)

4. **Komprese pohyblivého obrazu** - metody využívající ztrátovou kompresi, ale na rozdíl od *komprese nepohyblivého obraz* mají větší poměr komprese.

5. **Komprese zvuku** - metody používající ztrátovou kompresi pro kompresi audio souborů.

V této rešerši nebudou popsány komprese obrazu a zvuku.

1.3 Frekvenční analýza

Při ukládání dat běžným způsobem neprobíhá jakási analýza obsahu dat. Jinými slovy se nebere v potaz, jestli náhodou daný soubor neobsahuje např. pouze identické znaky, tj. jestli náhodou poskytované informace nejsou duplicitní a jestli je nejde uložit efektivněji, aby nezabírali zbytečně mnoho místa. Exemplární příklad neúsporného kódování bude převzat z *Komprese dat* od Arnošta Večerky [2]:
Nechť je cílem uložit zdrojové texty programů, které vesměs používají jen znaky ze základní části tabulky ASCII. Znaky v základní části ASCII tabulky jsou vyjádřeny hodnotami v rozsahu 0-127 a pro jejich uložení tedy stačí 7 bitů. Přesto jsou znaky zdrojového textu běžně ukládány na jednom bytu, ve kterém jeden z 8 bitů není používán. Pokud by se použila komprese, při které by došlo k vynechání nepoužívaného bitu, tak by kompresní poměr byl $\frac{7}{8}$. Z toho příkladu je patrné, že při kompresi dat jde o uložení dat potřebných k předání původní informace, jež vstupní soubor obsahoval. O potřebnosti určitých dat se dá rozhodnout pomocí entropie z teorie přenosu zpráv [2].

Entropie V rámci definování entropie bude platit předpoklad, že je vyjádřena v bitech. Tento předpoklad tkví z toho, že při kódování dat se jako základní jednotka informace používají právě bity.
Definice [2]: Mějme nějaká data, která jsou tvořena posloupností n různých prvků a_i $(i \in \hat{n})$ - jako prvek se uvažuje znak, tj. hodntu uloženou na jednom bytu - s pravděpodobnostmi výskytu v datech p_i:

$$a_1, a_2, a_3, ..., a_n \text{ s pravděpodobnostmi } p_1, p_2, p_3, ..., p_n \tag{2}$$

Tedy prvek a_i reprezentuje informaci a entropie pro tento i-tý prvek (E_i) znázorňuje množství (míru) informace:

$$E_i = -\log_2(p_i) \quad \text{bitů} \tag{3}$$

Jinými slovy, entropie E_i říká, jak velkou informaci nese výskyt prvku a_i [2]. Z tohoto lze odvodit, že čím bude pravděpodobnost výskytu prvku nižší, tak tím větší bude jeho entropie (míra informace).
Z entropie jednotlivých prvků lze vyjádřit průměrnou entropii pro celý soubor. Průměrná entropie pro celý soubor (E) je vyjádřena střední hodnotou ze součtu entropií všech prvků vynásobené patřičnými pravděpodobnostmi výskytu:

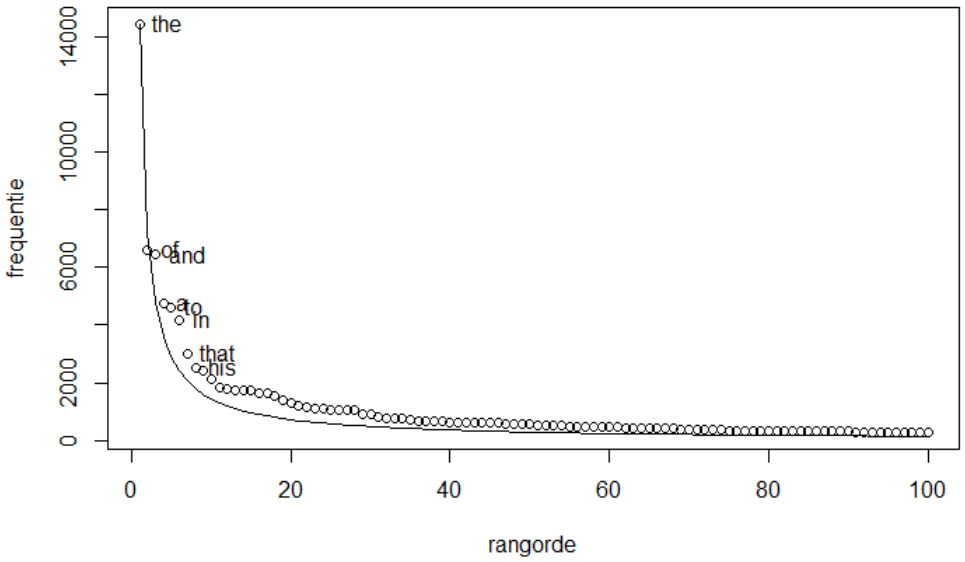
$$E = p_1E_1 + p_2E_2 + p_3E_3 + ... + p_nE_n = -\sum_{i=1}^{n} p_i \log_2(p_i) \quad \text{bitů} \tag{4}$$

Entropie nám při kódování znaků umožňuje zjistit, do jaké míry je použitý kód optimální z pohledu dosažení co nejkratšího kódování [2].

1.4 Zipfův zákon

Zipfovy zákony jsou vztahy mezi frekvencí výskytu jednotky a její distribucí v jazyce. Objevují se nejčastěji při analýze struktury jazyka v textu [6]. Tyto 3 zákony zformuloval americký lingvista George Kingsley Zipf:

1. Zipfův zákon Nechť existuje text, který je napsán v přirozeném jazyce (jazyk, který vznikl přirozeným vývojem v čase - protipříkladem je Esperanto) - tato vlastnost je ale stále předmětem diskuzí Lingvistů [7].
Pak *1. Zipfův zákon* říká, že v tomto textu četnosti výskytů určitých slov mají určité rozdělení - od slova s nejvyšším výskytem až po slovo s nejmenším výskytem, tj. od slova s rankem `1` až po slovo s rankem `n`. Formálněji řečeno, součin pořadí a četnosti každého slova bude přibližně konstanta k, tj. $f \cdot r = k$ (kde f je frekvence slova a r je rank onoho slova). Jako důsledek lze usoudit, že základem každého jazyka je určitá menší kolekce slov se silnou frekvencí výskytu [7]. Příklad frekvenční analýzy distribuce slov v knize Moby Dick od Herman Melville je zobrazen v obrázku 1. Další příklad textu na kterém byl Zipfův zákon ověřen je Odysseus od Jamese Joyce, nebo telefonní seznam.



Obrázek 1: Graf distribuce frekvence prvních 100 nejvíce vyskytujících se slov v knize Moby Dick od Herman Melville. Na svislé ose se nachází počet výskytů slov a horizontální osa znázorňuje rank daného slova. Křivka je výsledkem předpovědi Zipfova zákonu a kolečka reprezentují skutečné četnosti frekvencí výskytů slov v textu. Příklad byl převzat z článku *Unzipping Zipf's Law: Solution to a century-old linguistic problem* od Radboudské univerzity [8].

2. Zipfův zákon Mějme stejný text jako v definici *1. Zipfova zákonu.*
Pak *2. Zipfův zákon* říká, že existuje vztah mezi počtem slov se stejnou frekvencí výskytu a touto frekvencí [7]. Zipf tento vztah vyjádřil následně: $a \cdot f^2 = k$ (kde a je počet slov, které mají frekvenci výskytu f a k je opět konstanta). Tento zákon je v podstatě rozšíření důsledku *1. Zipfova zákonu* (základem každého jazyka je malý počet silně frekventovaných slov). Jinými slovy, *2. Zipfův zákon* říká, že čím vyšší je frekvence výskytu slov, tak tím méně slov se na této frekvenční úrovni nachází [9].

3. Zipfův zákon Mějme opět stejný předpoklad pro text, jako v předchozích Zipfových zákonech.
Pak *3. Zipfův zákon* vyjadřuje vztah mezi frekvencí výskytů určitého slova a počtem významů onoho slova. Tento vztah vyjádřil Zipf následně: $\frac{m}{\sqrt{f}} = k$ (kde m je počet významů daného slova, f je frekvence slova a k je opět konstanta). Jinými slovy, *3. Zipfův zákon* říká, že slova s vyšší frekvencí výskytu v daném textu mají ve většině případů více významů než slova, které mají menší frekvenci výskytu - některé zdroje dokonce uvádějí, že slova s velice nízkou frekvencí výskytu mají většinou pouze jeden význam [9]. Podle článku *George Kingsley Zipf* z webové stránky *WikiKnihovna* [9] tento zákon je obtížné dokázat, jelikož některé významy slov mohou být subjektivní.

Kvůli nejasnostem, které jsou spojené s důkazem *3. Zipfova zákona* se v praxi spíše používá pouze *1. a 2. Zipfův zákon* [6].

2 Algoritmy pro kompresi dat

Komprese je prováděna programem, který používá vzorec nebo algoritmus k určení, jak zmenšit velikost dat. Například, algoritmus může převést řetězec bitů - nuly nebo jedničky - na menší řetězec nul a jedniček pomocí slovníku pro převod mezi nimi. Druhý možný postup (pomocí vzorce) spočívá ve vložení odkazu nebo ukazatele do řetězce na místo nul a jedniček, které program již viděl.
Komprimovat lze prakticky jakýkoli typ souboru, ale při výběru, které z nich komprimovat, je důležité dodržovat osvědčené postupy. Například některé soubory již mohou být komprimovány, takže komprimace těchto souborů by neměla významný dopad.
V této části rešerše budou předvedeny následující bezeztrátové algoritmy pro kompresi dat:

- Run-length encoding (RLE) (zdroje: [10][11][12][13])
- Shannon-Fano kódování (zdroje: [14][15][16])
- Huffmanovo kódování (Statické, Adaptivní - FGK, Vitter) (zdroje: [17][18][19][20])
- LZ77 (zdroje: [21][22])
- LZ78 (zdroje: [23][24])
- LZ7W (zdroje: [25][26])

2.1 Run-Length Encoding (RLE)

Run-Length Encoding (RLE), neboli tzv. *Kódování délkou běhu* je primitivní algoritmus, který je založen na principu komprimování opakujících se znaků v datech.

RLE kódování Postup algoritmu je následující:

1. Načíst ze vstupních dat první znak.
2. Napočítat kolik těchto znaků se vyskytuje přímo za sebou a zaznamenat tyto informace na konec výstupního souboru v následném pořadí: <počet_opakování><znak> (bez symbolů "<"a ">").
3. Pokud není konec vstupu, tak se posunout na další znak a přejít na krok 2. Jinak přejít na krok 4.
4. Pokud na vstupu už není další znak, tak výsledný výstupní soubor je původní komprimovaný pomocí RLE.

Jinými slovy, RLE algoritmus zakóduje znaky ze vstupních dat do dvojic: (počet opakování znaku, znak).

RLE dekódování Postup algoritmu dekódování dat komprimovaných pomocí RLE je následující:

1. Načíst ze vstupních komprimovaných dat první znak (podle formátování kódování pomocí RLE bude první znak číslice)
2. Načítat znaky, dokud se nenarazí na nečíselný znak. Dosavadně načtené číslice dávají dohromady číslo, ozn. jako `n`.
3. Načíst onen nečíselný znak a do výstupního souboru napsat `n` těchto znaků za sebou.
4. Pokud není konec vstupu, tak se posunout na další znak (číslice) a přejít na krok 2.
5. Pokud na vstupu není další znak, tak výsledný výstupní soubor je dekomprimovaný vstupní komprimovaný soubor.

Příklad Jako názorná ukázka běhu algoritmu dat bude vstupní soubor obsahovat následující data:

```
AAAAAAbbbCCCCCCCCd
```

Tento soubor zakódovaný pomocí RLE bude vypadat následně:

```
6A3b8C1d
```

Je vidět, že pokud by vstupní soubor (který má být komprimován) neobsahoval žádné posloupnosti po sobě jdoucích znaků, tak by algoritmus kódování RLE zdvojnásobil velikost původního souboru, jelikož by pro každý znak ve vstupním souboru do výstupního dal dva znaky (číslici 1 a daný znak).
RLE algoritmus se používá občas v počítačové grafice při kompresi obrazů, které se skládají z větších ploch stejné barvy. Mezi nejznámější formáty pro ukládání grafických souborů, které využívají RLE algoritmus pro kompresi je např. formát PCX [13].

2.2 Shannon-Fano kódování (SHF)

Shannon-Fano (SHF) je způsob kódování, se kterým nezávisle na sobě v roce 1949 přišli pánové Claude Elwood Shannon (s kolegou Warrenem Weaverem) a Robert Mario Fano [16]. SHF se velice podobá Huffmanovu kódování v tom, že má za cíl pro každý unikátní znak z nějaké vstupní zprávy vytvořit odpovídající kód. Podoba kódu pro každý znak záleží na počtu výskytů daného znaku, případně na jeho pravděpodobnosti výskytu ve vstupní zprávě. Jinými slovy, pomocí stromové struktury se SHF snaží redukovat počet bitů co zaberou nejčastěji vyskytující se znaky ve zprávě.

Algoritmus

1. Ze vstupního souboru načíst všechny unikátní znaky a jejich počet výskytů, nebo jejich pravděpodobnost.
2. Tento seznam znaků seřadit podle hodnoty (počet výskytů, nebo pravděpodobnosti) od největší do nejmenší.
3. Inicializovat hodnotu kódů pro všechny znaky jako prázdný řetězec.
4. Pokud je v seznamu více než jeden prvek, tak seznam rozdělit na dva seznamy takovým způsobem, aby součet hodnot v každém z těch nových seznamů byl přibližně stejný. Pokud je v seznamu už jen jeden znak, tak běh algoritmu pro tento seznam ukončit.
5. Na konec kódů pro znaky v 1. seznam připsat `0`, na konec kódů pro znaky v 2. seznam připsat `1`.
6. Pro každý nově vytvořený seznam se vrátit na krok 4 (tj. rekurzivně pro každou půlku přejít na krok 4).

Výsledné kódy pro znaky se nazývají prefixové, tzn., že kód žádného ze znaků není prefix jiného, což zaručuje jednoznačnost kódování a dekódování. Také je vhodné zmínit, že délky kódů pro různé znaky se liší podle toho jak často se ve zprávě vyskytují - čím více výskytů, tím kratší kód - tímto se zajistí, že zakódovaná zpráva bude pro nejčastější znaky používat co nejméně kódu (pokud bude kód v bitech, tak budou nejčastěji vyskytující se znaky používat nejmenší počet bitů).
Příklad Pro znázornění algoritmu bude uveden ilustrační příklad.
Mějme vstupní zprávu: "*abrakadabra*". Celkově tato zpráva disponuje délkou o 11 znacích, z nichž je pouze 5 unikátních. Předpokladem bude, že hledanou hodnotou pro každý znak bude počet výskytů. Pokud bychom každému znaku přiřadili binární kód, tak by zakódovaná zpráva podle seřazeného seznamu znaků zobrazeného v tabulce 1 vypadala následně:

000, 001, 010, 000, 011, 000, 100, 000, 001, 010, 000

Tato zpráva má velikost 36 bitů.

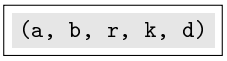
Znak	Četnost	Binární kód
a	5	000
b	2	001
r	2	010
k	1	011
d	1	100

Tabulka 1: Unikátní znaky ze zprávy "*abrakadabra*" s jejich četnostmi a binárními kódy.

Shannon-Fano kódování Použitím Shannon-Fano kódování se nám tato velikost značně redukuje - podle postupu algoritmu popsaného v 2.2.
První a druhý krok algoritmu byl již proveden konstrukcí a seřazení seznamu do tabulky 1.
Třetí krok spočívá v inicializaci hodnot kódů, viz. tabulka 2 a obrázek 2. Pro lepší představení rekurze bude zobrazena stromová struktura reprezentující hodnoty kódů v tabulkách.

Znak	Kód
a	-
b	-
r	-
k	-
d	-

Tabulka 2: Iniciální prázdné řetězce jako hodnoty kódů znaků.



Obrázek 2: Grafické znázornění tabulky 2. Seznam `(a, b, r, k, d)` je kořen stromu.

Následně se seznam `(a, b, r, k, d)` rozdělí na dva seznamy, součet výskytů jejich členů bude přibližně stejný:

1. Seznam `(a)` - součet výsktů: 5
2. Seznam `(b, r, k, d)` - součet výskytů: $2 + 2 + 1 + 1 = 6$

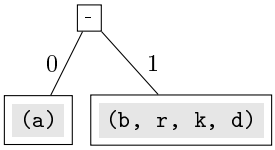
Na konec kódů znaků v 1. seznamu `(a)` se přiřadí `0` a na konec kódů znaků ve 2. seznamu `(b, r, k, d)` se přiřadí `1`, viz. tabulka 3 a obrázek 3.

Znak	Kód
a	0
b	1
r	1
k	1
d	1

Tabulka 3: Hodnoty kódů znaků po 1. dělení seznamu.

Seznam `(a)` obsahuje pouze jeden znak, a tedy pro něj algoritmus končí. Seznam `(b, r, k, d)` obsahuje více než jeden prvek, tedy bude opět rozdělen na dva seznamy:

1. Seznam `(b, r)` - součet výsktů: $2 + 2 = 4$
2. Seznam `(k, d)` - součet výskytů: $1 + 1 = 3$

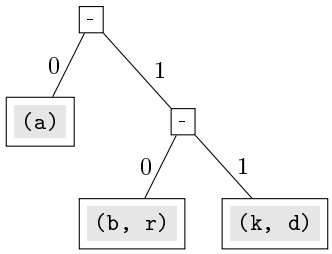


Obrázek 3: Grafické znázornění tabulky 3. Pohyb po hranách tvoří dosavadní kód pro znaky v uzlech.

Na konec kódů znaků seznamu `(b, r)` se přiřadí `0` a na konec kódů znaků seznamu `(k, d)` se přiřadí `1`, viz. tabulka 4 a obrázek 4.

Znak	Kód
a	0
b	10
r	10
k	11
d	11

Tabulka 4: Hodnoty kódů znaků po 2. dělení seznamu.



Obrázek 4: Grafické znázornění tabulky 4. Pohyb po hranách tvoří dosavadní kód pro znaky v uzlech.

Seznam `(b, r)` obsahuje více než jeden prvek, a bude rozdělen na dva seznamy. Seznam `(k, d)` také obsahuje více než jeden prvek, tedy bude také rozdělen na dva seznamy:

1. Seznam `(b, r)` - součet výsktů: $2 + 2 = 4$
   (a) Seznam `(b)` - součet výsktů: 2
   (b) Seznam `(r)` - součet výsktů: 2
2. Seznam `(k, d)` - součet výskytů: $1 + 1 = 2$
   (a) Seznam `(k)` - součet výsktů: 1
   (b) Seznam `(d)` - součet výsktů: 1

Na konec kódů znaků 1. seznamů `(b)` a `(k)` se přiřadí `0` a na konec kódů znaků ve 2. seznamech `(r)` a `(d)` se přiřadí `1`, viz. tabulka 5 a obrázek 5.

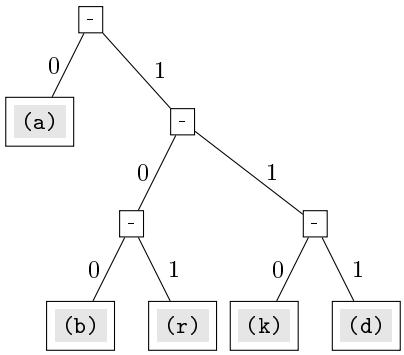
Stromová struktura 5 reprezentuje tabulku takovým způsobem, že binární kód bude posloupnost hodnot hran v cestě od kořene do uzlu toho daného znaku. Porovnání starých a nových kódů pro znaky je zobrazeno v tabulce 6. Podle této tabulky bude zakódovaná zpráva "*abrakadabra*"vypadat takto:

0, 100, 101, 0, 110, 0, 111, 0, 100, 101, 0

Tato zpráva má velikost 23 bitů, což je podstatně méně než velikost původní zprávy: 36 bitů. Jak bylo zmíněno na začátku, tak se jedná o prefixový kód, což je v tabulce 6 ve sloupci *Nový binární kód* také vidět - žádný binární kód není prefixem jiného.

Znak	Kód
a	0
b	100
r	101
k	110
d	111

Tabulka 5: Hodnoty kódů znaků po 3. dělení seznamu.



Obrázek 5: Grafické znázornění tabulky 5. Pohyb po hranách tvoří finální kód pro znaky v uzlech.

Znak	Četnost	Původní Binární kód	Nový binární kód
a	5	000	0
b	2	001	100
r	2	010	101
k	1	011	110
d	1	100	111

Tabulka 6: Unikátní znaky ze zprávy "*abrakadabra*" s jejich četnostmi, starými binárními kódy a novými binárními kódy.

2.3 Huffmanovo kódování

Huffmanovo kódování je příkladem grafového algoritmu, tedy používá ve svém postupu uzly a hrany[**?**]. Princip algoritmu je založen na redukci počtu bitů co zaberou nejčastěji vyskytující se znaky v nějaké zprávě. Tohoto se docílí pomocí tzv. Huffmanova stromu, který obsahuje znaky ze vstupní zprávy a pomocí kterého se zakódují znaky podle toho kde se v něm nachází.
V této rešerši budou popsána tři různá Huffmanova kódování:

- Statické Huffmanovo kódování (viz. sekce 2.3.1)
- Adaptivní Huffmanovo kódování - FGK (viz. sekce 2.3.2)
- Adaptivní Huffmanovo kódování - Vitterův algoritmus (Λ) (viz. sekce 2.3.3)

Prvně bude představen obecný a detailní popis algoritmu v rámci Statického Huffmanova kódování a následně budou pro ostatní druhy Huffmanova kódování popsány pouze rozdíly.

2.3.1 Statické Huffmanovo kódování

Algoritmus Konkrétně je algoritmus následující:

Znak	Četnost	Binární kód
a	5	000
b	2	001
d	1	010
k	1	011
r	2	100

Tabulka 7: Unikátní znaky ze zprávy "*abrakadabra*" s jejich četnostmi a binárními kódy.

1. Načte se vstupní zpráva a udělá se analýza obsažených znaků - tj. vytvoří se množina znaků a počet jejich výskytů (případně jejich váhy - pokud celá zpráva má váhu 1, tak z toho jsou například 0,1 znaky "a"). Nechť velikost této množiny je n.

2. Pro každý prvek z této množiny se vytvoří kořenový strom - tedy máme n kořenových stromů. Přičemž každý strom T_i $(i \in \hat{n})$ reprezentuje právě jeden znak Z_i. Další důležitá informace je váha stromu, v tomto případě je váha stromu T_i definována následovně: $w(T_i) = \#Z_i$. Jinými slovy je váha stromu T_i rovna počtu výskytů znaku Z_i ve zprávě. Nyní tedy máme plně nesouvislý graf (mezi žádnými dvěma uzly neexistuje hrana).

3. Dokud graf není souvislý (tj. dokud neexistuje sled mezi všemi uzly ve stromu) tak se opakuje následující:

   (a) Z grafu se vyjmou dva kořenové stromy s nejmenší váhou: T_1 a T_2

   (b) Vytvoří se nový kořenový strom T_3, který bude mít jako levého potomka T_1 a jeho pravý potomek bude T_2. Váha nového kořenu se bude rovnat součtu vah jeho potomků, tj. $w(T_3) = w(T_1) + w(T_2)$

   (c) T_1 a T_2 se odeberou z grafu a vloží se do něj T_3

4. Binární strom, který z tohoto postupu vznikne se nazývá Huffmanův strom, kterému se ohodnotí hrany jež ústí v levých potomcích nulou a hrany ústící v pravých potomcích jedničkou.

5. Následně se zkonstruuje binární kód pro každý znak z množiny znaků. Začínaje v kořenu Huffmanova stromu jdeme postupně k uzlu obsahující znak, pro který chceme určit binární kód. Hodnotu každé navštívené hrany (0 nebo 1) zapíšeme. Výsledný zápis je binární kód pro daný znak.

Výsledné kódy pro znaky se nazývají prefixové, tzn., že kód žádného ze znaků není prefix jiného, což zaručuje jednoznačnost kódování a dekódování. Také je nutné podotknout, že délky kódů pro různé znaky se liší podle toho jak často se ve zprávě vyskytují - čím více výskytů, tím kratší kód - tímto se zajistí, že zakódovaná zpráva bude pro nejčastější znaky používat co nejméně kódu.

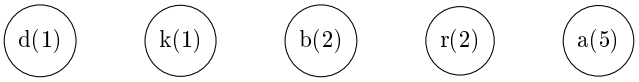
Příklad Pro znázornění algoritmu bude uveden ilustrační příklad.
Nechť máme vstupní zprávu: "*abrakadabra*". Celkově tato zpráva disponuje délkou o 11 znacích, z nichž je pouze 5 unikátních. Pokud bychom každému znaku přiřadili binární kód, tak by zakódovaná zpráva podle tabulky 7 vypadala takto:

000, 001, 100, 000, 011, 000, 010, 000, 001, 100, 000

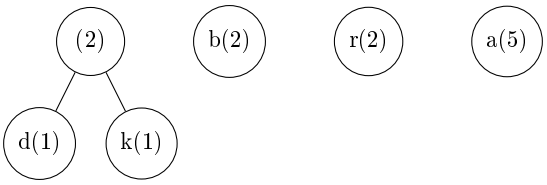
Tato zpráva má velikost 36 bitů.

Statické Huffmanovo kódování Použitím statického Huffmanova kódování se nám tato velikost značně redukuje - podle postupu popsaného v 2.3.1.
V prvním kroku je potřeba sestavit Huffmanův strom, počínaje s kořenovými uzly stromu pro každý znak:

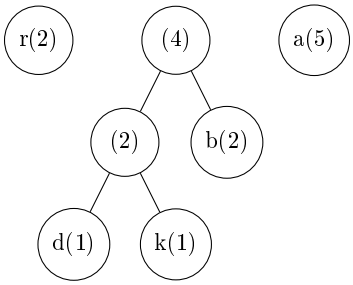
Následně je potřeba vzít dva uzly jejichž znaky mají nejmenší váhu (četnost), spojit je pod nový kořenový uzel s váhou která bude rovna součtu jejich vah, odebrat je z grafu a zařadit do grafu nový kořen.



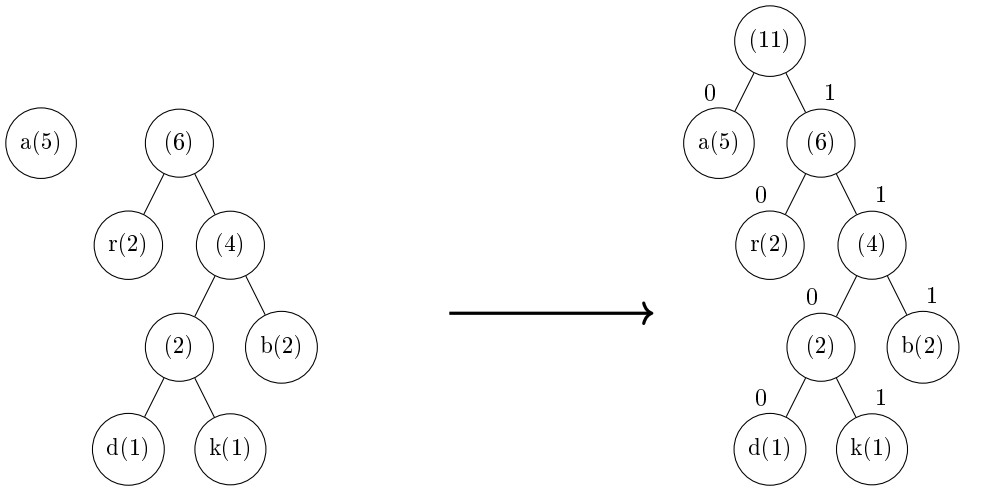
Obrázek 6: Kořenové uzly se znakem a odpovídající četností v závorkách uvnitř uzlu. Uzly jsou seřazeny podle velikostí četností pro přehlednost.



Obrázek 7: Spojení d(1) a k(1) do (2).



Obrázek 8: Spojení (2) a b(2) do (4).



(a) Spojení r(2) a (4) do (6).

(b) Spojení a(5) a (6) do (11).

Obrázek 9: Výsledný Huffmanův strom v (b) s ohodnocenými hranami.

Tento postup se opakuje dokud nebude graf souvislý.

Z Huffmanova stromu v obrázku 9 se sestaví tabulka binárních kódů pro každý znak, s tím, že tento binární kód bude posloupnost hodnot hran v cestě od kořene do uzlu toho daného znaku. Výslednou tabulku můžeme vidět v tabulce 8. Podle této nové tabulky bude zakódovaná zpráva "abrakadabra"vypadat takto:

0, 111, 10, 0, 1101, 0, 1100, 0, 111, 10, 0

Tato zpráva má velikost 23 bitů, což je podstatně méně než velikost původní zprávy: 36 bitů. Jak bylo zmíněno na začátku, tak se jedná o prefixový kód, což je v tabulce 8 ve sloupci *Nový binární kód* také vidět - žádný binární kód není prefixem jiného.

Znak	Četnost	Původní Binární kód	Nový binární kód
a	5	000	0
b	2	001	111
d	1	010	1100
k	1	011	1101
r	2	100	10

Tabulka 8: Unikátní znaky ze zprávy "*abrakadabra*" s jejich četnostmi, starými binárními kódy a novými binárními kódy.

2.3.2 Adaptivní Huffmanovo kódování - FGK

Adaptivní Huffmanovo kódování (typu FGK) je modifikace Huffmanova kódování. Tuto modifikaci nezávisle na sobě přišli publikovali pánové Faller v roce 1973, Gallager v roce 1978 a Knuth v roce 1985. Algoritmus získal svůj doplňkový název "*FGK*" z prvních písmen příjmení publikujících.

Před popisem algoritmus FGK algoritmu je nutné uvést modifikaci pro definici Huffmanova binárního stromu - představena v *Statické Huffmanovo kódování* (sekce 2.3.1). Strom v FGK algoritmu se nazývá sourozenecký (také se říká, že strom má sourozeneckou vlastnost), pokud jsou splněny následující podmínky:

i) Kromě kořene, každý uzel má sourozence.

ii) Uzly lze uvést v pořadí nezvyšujících se četností přičemž každý uzel má jako souseda v posloupnosti svého sourozence.

Ze sourozeneckého stromu lze poté vytvořit binární prefixový kód, který je zároveň Huffmanovým kódem - dokázáno panem Gallagerem.

Algoritmus Konkrétně je algoritmus následující:

1. Načte se vstupní zpráva a udělá se analýza obsažených znaků - tj. vytvoří se množina znaků a počet jejich výskytů (případně jejich váhy - pokud celá zpráva má váhu 1, tak z toho jsou například 0,1 znaky "a"). Nechť velikost této množiny je n.

2. Pro každý prvek z této množiny se vytvoří kořenový strom - tedy máme n kořenových stromů. Přičemž každý strom T_i $(i \in \hat{n})$ reprezentuje právě jeden znak Z_i. Další důležitá informace je váha stromu, v tomto případě je váha stromu T_i definována následovně: $w(T_i) = \#Z_i$. Jinými slovy je váha stromu T_i rovna počtu výskytů znaku Z_i ve zprávě. Nyní tedy máme plně nesouvislý graf (mezi žádnými dvěma uzly neexistuje hrana).

3. Dokud graf není souvislý (tj. dokud neexistuje sled mezi všemi uzly ve stromu) tak se opakuje následující:

   (a) Z grafu se vyjmou dva kořenové stromy s nejmenší váhou: T_1 a T_2

   (b) Vytvoří se nový kořenový strom T_3, který bude mít jako levého potomka T_1 a jeho pravý potomek bude T_2. Váha nového kořenu se bude rovnat součtu vah jeho potomků, tj. $w(T_3) = w(T_1) + w(T_2)$

   (c) T_1 a T_2 se odeberou z grafu a vloží se do něj T_3

   (d) Zkontrolovat, jestli nedošlo k porušení sourozenecké vlastnosti. Pokud ne, tak se pokračuje na další krok.

   (e) Pokud došlo k porušení sourozenecké vlastnosti tak se strom přestaví takovým způsobem, aby sourozenecká vlastnost nebyla porušována.

4. Binární strom, který z tohoto postupu vznikne se nazývá modifikovaný Huffmanův strom, kterému se ohodnotí hrany jež ústí v levých potomcích nulou a hrany ústící v pravých potomcích jedničkou.

5. Následně se zkonstruuje binární kód pro každý znak z množiny znaků. Začínaje v kořenu Huffmanova stromu jdeme postupně k uzlu obsahující znak, pro který chceme určit binární kód. Hodnotu každé navštívené hrany (0 nebo 1) zapíšeme. Výsledný zápis je binární kód pro daný znak.

Pozn. Krok 1 ve výše popsaném algoritmu má dvě možná znění:

i) Pro každý znak ze vstupní abecedy se vytvoří kořenový strom, který bude obsahovat onen znak a četnost tohoto znaku. Tedy bude n kořenových stromů

ii) Při inicializaci obsahuje strom pouze tzv. uzel `ZERO`. Poté se místo kroků 2 a 3 provede následující: načítají se znaky ze vstupního textu, přičemž pokud se načte znak, který se ve stromu nenachází (tj. nebyl dosud zakódován), tak se jako kód daného znaku zapíše kód uzlu `ZERO`. Následně se uzel `ZERO` rozštěpí na dva uzly: nový uzel `ZERO` a uzel obsahující nový znak s četností.

2.3.3 Adaptivní Huffmanovo kódování - Vitterův algoritmus (Λ)

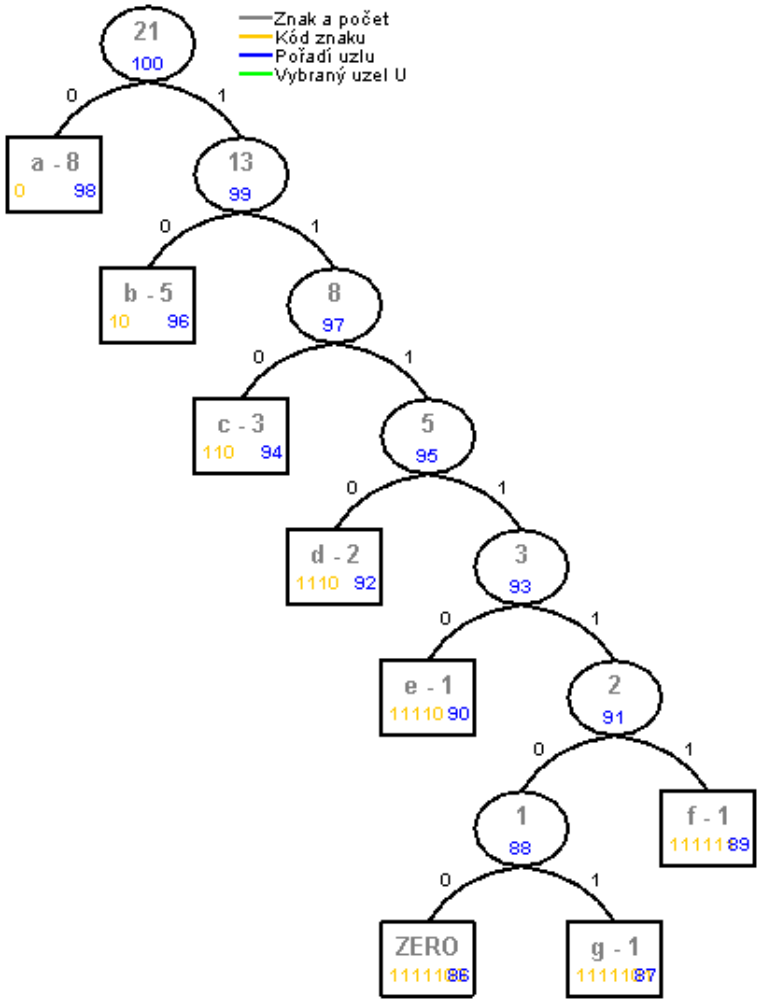
Další modifikaci Huffmanova kódování v roce 1987 publikoval pan Vitter. V celku se jedná o algoritmus podobná algoritmu FGK s tím, že úprava stromu po porušení sourozenecké vlastnosti se trochu liší. Tato mírná úprava má za následek efektivnější kompresi v určitých případech.

Algoritmus Konkrétně je algoritmus následující:

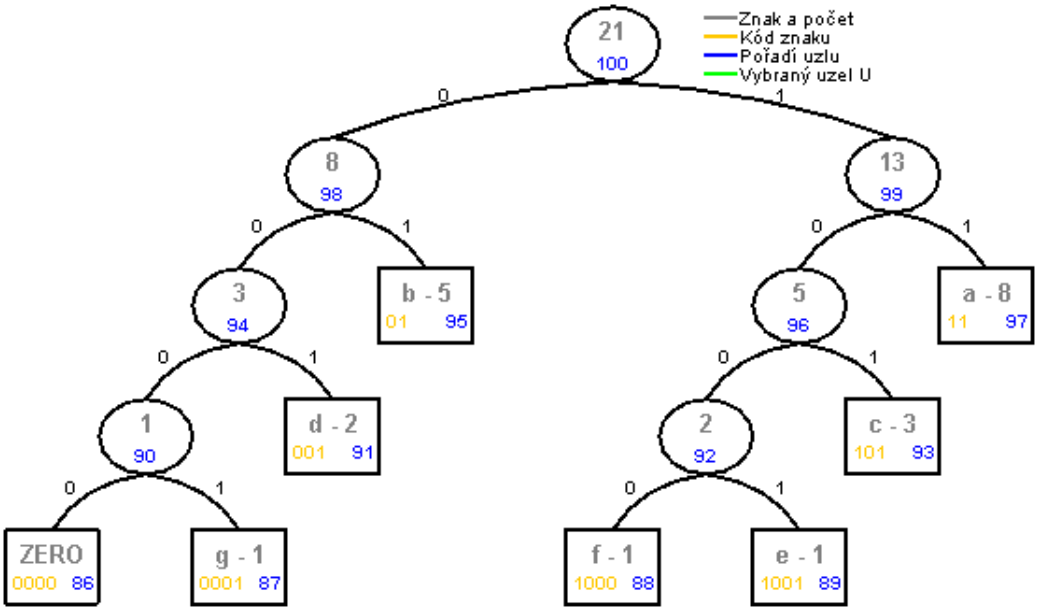
1. Inicializuje se strom pouze s uzlem `ZERO`.
2. Dokud nejsou načteny všechny znaky ze vstupu tak se opakuje následující:
   (a) Pokud se načtený znak ve stromu již nachází, tak se pokračuje na další znak
   (b) Pokud se načtený znak ve stromu nenachází (tj. nebyl dosud zakódován), tak se jako kód daného znaku bude uvažovat kód uzlu `ZERO`. Následně se uzel `ZERO` rozštěpí na dva uzly: nový uzel `ZERO` a uzel obsahující nový znak s četností. Původní uzel bude obsahovat součet četností výskytů svých potomků.
   (c) Proběhne aktualizace stromu:
      i. Začínaje zespoda u posledního nově vytvořeného uzlu (alternativně u nejméně vyskytujícího se znaku). Tento uzel bude označen jako uzel u.
      ii. Určí se blok b (skupina uzlů, které mají stejnou četnost - případně pravděpodobnost - a v rámci stromu se jedná o stejný typ uzlu: vnitřní uzly nebo listové uzly) předcházející uzlu u.
      iii. Pokud platí jedno z následujících:
         - u je listový uzel a b je blok vnitřních uzlů.
         - u je vnitřní uzel a b je blok listových uzlů.

      tak se pomocí levé rotace uzel u přemístí na pozici před blokem. Pokud je nyní uzel u listovým uzlem, tak se přesune označení uzlu u na jeho předka. Pokud není listovým uzlem, tak se přesune označení uzlu u na jeho předka před rotací. Pokračuje se na další znak.
      iv. Pokud ani jedna z podmínek v kroku iii neplatí, tak se zvýší ohodnocení uzlu u o 1 a přesune se označení uzlu u na jeho předka.
3. Binární strom, který z tohoto postupu vznikne se nazývá modifikovaný Huffmanův strom, kterému se ohodnotí hrany jež ústí v levých potomcích nulou a hrany ústící v pravých potomcích jedničkou.
4. Následně se zkonstruuje binární kód pro každý znak z množiny znaků. Začínaje v kořenu Huffmanova stromu jdeme postupně k uzlu obsahující znak, pro který chceme určit binární kód. Hodnotu každé navštívené hrany (0 nebo 1) zapíšeme. Výsledný zápis je binární kód pro daný znak.

Příklad Pro znázornění rozdílů mezi FGK a Vitterovou modifikací Huffmanova kódování je znázorněn příklad z *Adaptivní Huffmanovo kódování - Vitterův algoritmus (Λ)*[20]. Mějme vstupní zprávu "*abacabdabaceabacabdf*". FGK i Vitterův algoritmus zkonstruují stejný strom. Pokud by se ale na konec vstupní zprávy přidal znak "G", tak se stromy již nebudou shodovat. Konkrétně bude strom zkonstruovaný v rámci Vitterova algoritmu vyváženější. Stromy po načtení znaku "G" jsou vidět v obrázku 10 a v obrázku 11.



Obrázek 10: Strom vytvořený v rámci algoritmu FGK po vložení znaku "G". Převzatý z [20].



Obrázek 11: Strom vytvořený v rámci Vitterova algoritmu po vložení znaku "G". Převzatý z [20].

2.4 LZ77

LZ77 je kompresní algoritmus, se kterým v roce 1977 přišli pánové Abraham Lempel a Jacob Ziv. Jedná se o slovníkovou kompresní metodu, tedy vyhledává opakující se řetězce ve vstupním souboru. Každému řetězci je následně přiřazen kód, který se ukládá do výstupního souboru. LZ77 postupně prochází vstupní data a zapisuje do slovníku nové řetězce (ty také vypisuje do výstupu), pokud narazí na řetězec který se již nachází ve slovníku, tak jej ve výstupu nahradí ukazatelem na poslední výskyt onoho řetězce ve vstupních datech a následně jej zapíše do slovníku s neshodným znakem, který následuje onen shodný řetězec.
Popis algoritmu bude v rámci této práce předpokládat, že vstupní data jsou v podobě řetězce.

Algoritmus komprese

1. Na vstupní text se posadí okno, které se dělí na dvě (po sobě jdoucí) okna: prohlížecí okno (tzv. search buffer) a aktuální okno (tzv. look-ahead buffer). Prohlížecí okno bude koukat na m znaků a aktuální okno bude koukat na n znaků, přičemž platí $m \geq n$.

2. Začně se s prázdným prohlížejícím oknem a načtením n znaků z počátku textu do aktuálního okna.

3. Hledá se nejdelší podřetězec (tedy může být i celý řetězec v prohlížecím okně nebo prázdný řetězec) v prohlížejícím okně, který je shodný s nějakým prefixem v aktuálním okně.

4. Nalezený řetězec je zakódován trojicí údajů: `(i, j, z)`, kde:
   - `i` - vzdálenost začátku aktuálního okna k začátku podřetězce v prohlížecím okně (tj. počínaje na prvním znaku v aktuálním okně, o kolik znaků se musíme vrátit zpět aby jsem se dostali na počátek podřetězce v prohlížecím okně). Pokud je více podřetězců v prohlížecím okně, tak se použije nejkratší vzdálenost.
   - `j` - délka shodného řetězce.
   - `z` - znak který se nachází v aktuálním okně po shodném vybraném prefixu (*pozn.* pokud je shodný podřetězec prázdný, tak prefix je také prázdný a vezme se první znak v aktuálním okně).

5. Celé okno se posune v textu o `j+1` znaků doprava.

6. Pokud není aktuální okno prázdné, tak se přejde na krok 3. Jinak končí komprese.

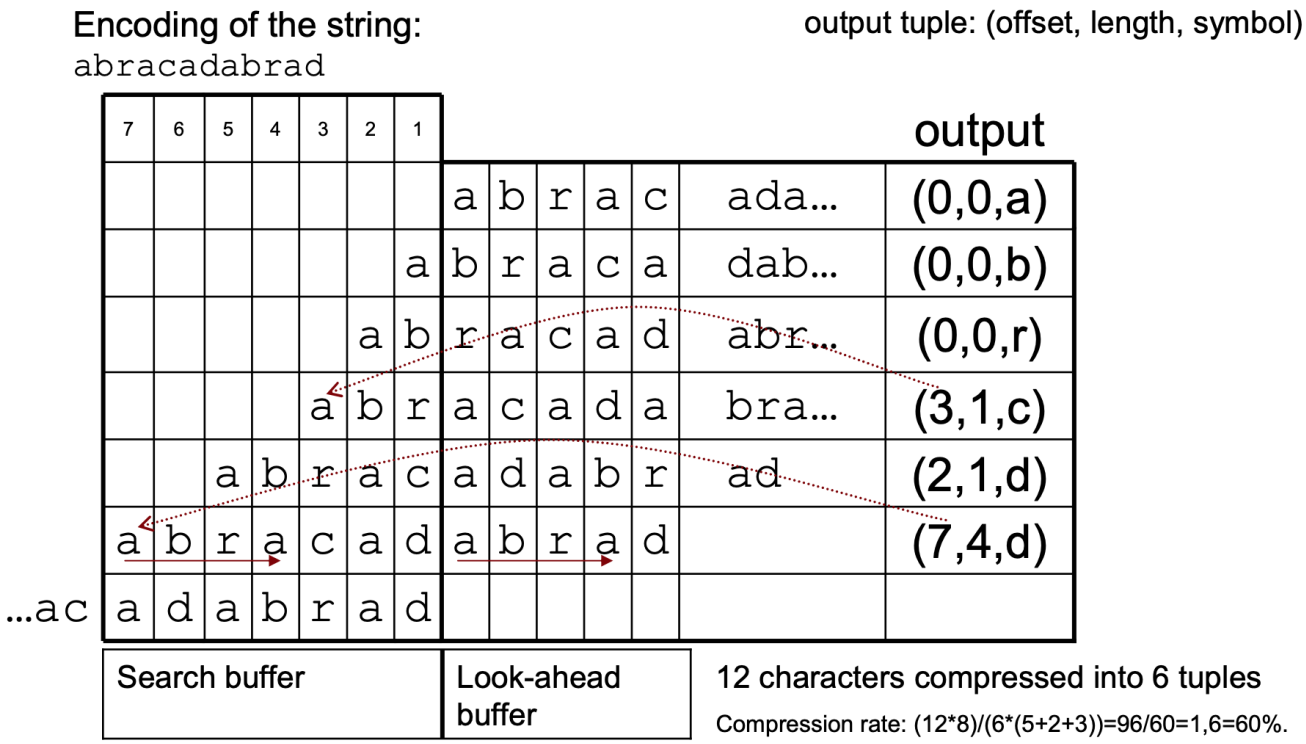
Pozn. - Slovník je v každém kroku tvořen všemi řetězci, které začínají v prohlížecím okně. Např. mějme v prohlížecím okně řetězec `AB` a v aktuálním okně řetězec `CD` (viz. obrázek 12).

Obrázek 12: Příklad rozdělení celkového okna na prohlížecí okno (ozn. 1) a aktuální okno (ozn. 2).

Slovník je tvořen daty v podobě: `"řetězec": (i, j)` (kde `i` značí kolik znaků zpátky z první pozice v aktuálním okně bychom se museli posunout, aby jsme se dostali na první znak řetězce a `j` značí délku řetězce). Slovník je tedy následující:

```
{"A": (2, 1), "AB": (2, 2), "ABC": (2, 3), "ABCD": (2, 4),"B": (1, 1), "BC": (1, 2), "BCD": (1, 3)}
```

Příklad Pro znázornění algoritmu bude uveden ilustrační příklad z *LZ77 compression algorithm (general code efficiency)*[22]. Nechť máme vstupní zprávu: "*abracadabrad*". Celkově tato zpráva disponuje délkou o 12 znacích, z nichž je pouze 5 unikátních, a také se v ní opakují podřetězce s více než 1 znakem, např. "*abra*". Postup algoritmu je zobrazen v obrázku 13.
V tomto příkladě je $m = 7$ a $n = 5$ a řádky v tabulce značí iterace algoritmu.



Obrázek 13: Postup komprese dle algoritmu LZ77. Každý řádek představuje iteraci algoritmu. Převzatý z [22].

V **první iteraci** je prohlížecí okno prázdné a aktuální okno obsahuje $n = 5$ znaků ze vstupního textu: `abrac`. Nejdelší podřetězec z prohlížecího okna, který se vyskytuje jako prefix v aktuálním okně je prázdný řetězec `""`, tedy neexistuje potenciální předchozí výskyt, nejde se posouvat k žádnému začátku řetězce `(i=0)` a potenciální předchozí výskyt nemá žádnou délku `(j=0)`. Následující znak v aktuálním okně po prefixu `""` je `a`. Na výstup zapíše tedy `(0, 0, a)`. Celkové okno se posune o `j+1=0+1=1`.

Ve **druhé iteraci** obsahuje prohlížecí okno: `a`, aktuální okno: `braca`. Nejdelší podřetězec z prohlížecího okna, který se vyskytuje jako prefix v aktuálním okně je prázdný řetězec `""`, tedy neexistuje potenciální předchozí výskyt, nejde se posouvat k žádnému začátku řetězce `(i=0)` a potenciální předchozí výskyt nemá žádnou délku `(j=0)`. Následující znak v aktuálním okně po prefixu `""` je `b`. Na výstup se tedy zapíše `(0, 0, b)`. Celkové okno se posune o `j+1=0+1=1`.

Ve **třetí iteraci** obsahuje prohlížecí okno: `ab`, aktuální okno: `racad`. Nejdelší podřetězec z prohlížecího okna, který se vyskytuje jako prefix v aktuálním okně je prázdný řetězec `""`, tedy neexistuje potenciální předchozí výskyt, nejde se posouvat k žádnému začátku řetězce `(i=0)` a potenciální předchozí výskyt nemá žádnou délku `(j=0)`. Následující znak v aktuálním okně po prefixu `""` je `r`. Na výstup se tedy zapíše `(0, 0, r)`. Celkové okno se posune o `j+1=0+1=1`.

Ve **čtvrté iteraci** obsahuje prohlížecí okno: `abr`, aktuální okno: `acada`. Nejdelší podřetězec z prohlížecího okna, který se vyskytuje jako prefix v aktuálním okně je `a`. Poslední předchozí výskyt `a` v prohlížejícím okně je od aktuálního okna vzdálen o `(i=3)` znaky a má délku `(j=1)`. Následující znak v aktuálním okně po prefixu `a` je `c`. Na výstup se tedy zapíše `(3, 1, c)`. Celkové okno se posune o `j+1=1+1=2`.

V **páté iteraci** obsahuje prohlížecí okno: `abrac`, aktuální okno: `adabr`. Nejdelší podřetězec z prohlížecího okna, který se vyskytuje jako prefix v aktuálním okně je `a`. Poslední předchozí výskyt `a` v prohlížejícím okně je od aktuálního okna vzdálen o `(i=2)` znaky a má délku `(j=1)`. Následující znak v aktuálním okně po prefixu `a` je `d`. Na výstup se tedy zapíše `(2, 1, d)`. Celkové okno se posune o `j+1=1+1=2`.

V **šesté iteraci** obsahuje prohlížecí okno: `abracad`, aktuální okno: `abrad`. Nejdelší podřetězec z prohlížecího

okna, který se vyskytuje jako prefix v aktuálním okně je `abra`. Poslední předchozí výskyt `abra` v prohlížejícím okně je od aktuálního okna vzdálen o `(i=7)` znaků a má délku `(j=4)`. Následující znak v aktuálním okně po prefixu `abra` je `d`. Na výstup se tedy zapíše `(7, 4, d)`. Celkové okno se posune o `j+1=4+1=5`.

V **sedmé iteraci** obsahuje prohlížecí okno: `adabrad`, aktuální okno je prázdné. Končí algoritmus a celkový výstup je:

`(0, 0, a), (0, 0, b), (0, 0, r), (3, 1, c), (2, 1, d), (7, 4, d)`

Algoritmus dekomprese Postup dekomprese pro soubory komprimované pomocí algoritmu LZ77 spočívá v načtení zakódovaného vstupu a postupném sestavování dekódovaného textu podle "instrukcí"na vstupu. Postup bude předveden na předem zakódovaném textu z příkladu.

Příklad Konkrétní iterace dekomprese se nacházejí v tabulce 9.

Krok	Vstup	Buffer	Přidaný řetězec	Výstup
1	`(0, 0, a)`		`a`	a
2	`(0, 0, b)`	a	`b`	ab
3	`(0, 0, r)`	ab	`r`	abr
4	`(3, 1, c)`	abr	`ac`	abrac
5	`(2, 1, d)`	abrac	`ad`	abracad
6	`(7, 4, d)`	abracad	`abrad`	abracadabrad

Tabulka 9: Iterace postupu dekomprese vstupního textu "*abracadabrad*" komprimovaného pomocí algoritmu LZ77.

V **prvním kroku** ze vstupu načítáme `(0, 0, a)`. Buffer je prázdný a na konec výstupu se zapisuje znak `a` bez žádného předchozího řetězce, protože `i=0` a `j=0`. Toto se opakuje ve druhém a třetím kroku.

Ve **čtvrtém kroku** ze vstupu načítáme `(3, 1, c)`. Na konec výstupu `abr` se bude zapisovat znak `c`, ale před tímto znakem se ještě zapíše řetězec, který je od konce bufferu vzdálen `i=3` znaky zpátky a má délku `j=1`: řetězec `a`. Tedy na konec výstupu se zapíše řetězec `ac`.

V **pátém kroku** ze vstupu načítáme `(2, 1, d)`. Na konec výstupu `abrac` se bude zapisovat znak `d`, ale před tímto znakem se ještě zapíše řetězec, který je od konce bufferu vzdálen `i=2` znaky zpátky a má délku `j=1`: řetězec `a`. Tedy na konec výstupu se zapíše řetězec `ad`.

V **šestém kroku** ze vstupu načítáme `(7, 4, d)`. Na konec výstupu `abracad` se bude zapisovat znak `d`, ale před tímto znakem se ještě zapíše řetězec, který je od konce bufferu vzdálen `i=7` znaky zpátky a má délku `j=4`: řetězec `abra`. Tedy na konec výstupu se zapíše řetězec `abrad`.
Další data na vstupu už nejsou, tedy je konec algoritmu a máme dekódovaný text:

`abracadabrad`

2.5 LZ78

Podobně jako LZ77, LZ78 je kompresní algoritmus se kterým v roce 1978 přišli pánové Abraham Lempel a Jacob Ziv. Opět se jedná o slovníkovou kompresní metodu, tedy vyhledává opakující se řetězce ve vstupním souboru. Každému řetězci je následně přiřazen kód, který se ukládá do výstupního souboru. LZ78 postupně prochází vstupní data a zapisuje do slovníku nové řetězce (ty také vypisuje do výstupu), pokud narazí na řetězec který se již nachází ve slovníku, tak jej zapíše do slovníku jako ukazatel na onen původní výskyt ve slovníku a přidá k němu jeho následující znak ve vstupních datech, který způsobuje neshodu. Do výstupu se pak vypíše místo řetězce ukazatel na existující řetězec ve slovníku a následující neshodný znak.
Popis algoritmu bude v rámci této práce předpokládat, že vstupní data jsou v podobě textového řetězce.

Algoritmus komprese

1. Inicializuje se prázdný dynamický slovník (tj. bude se za běhu algoritmu měnit).
2. Načte se první znak ze vstupního textu.
3. Počínaje od aktuálního znaku (včetně) se hledá nejdelší prefix, který se vyskytuje ve slovníku.
4. Pokud existuje znak ve vstupním text, který následuje po nalezeném nejdelšímu prefixu, tak se tento znak přidá na konec onoho prefixu (tj. přidá se na konec prefixu ten znak, který způsobuje neshodu).
5. Poté je tento výsledný řetězec zakódován jedním z následujících způsobů:
   (a) Pokud je délka řetězce `1`, tak se do slovníku přidá záznam v podobě n-tice: `(0, znak)`.
   (b) Pokud je délka nalezeného řetězce větší než `1`, tak se do slovníku přidá záznam v podobě n-tice: `(index_prefixu, nasledujici_znak)` (kde `index_prefixu` ukazuje na index ve slovníku, kde se nalezený nejdelší prefix (prefix bez neshodného znaku) vyskytuje, a `nasledujici_znak` je onen přidaný neshodný znak - pokud neexistuje, tak algoritmus narazil na konec vstupního textu, n-tice `(index_prefixu, "")` se vypíše na výstup a algoritmus končí).
6. Výsledná n-tice se zapíše na výstup.
7. Pokud existuje další znak ve vstupním textu (tj. znak, který následuje poslednímu znaku řetězce - přidaný znak), tak se na něj algoritmus přesune a pokračuje se na krok 3.
8. Pokud tento znak neexistuje, tak je konec vstupního textu a algoritmus končí.

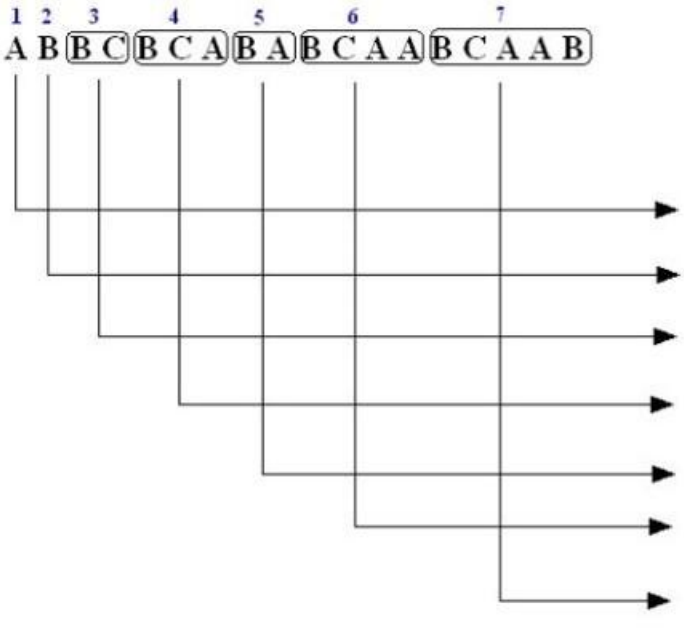
Příklad Pro znázornění algoritmu bude uveden ilustrační příklad z *LZ78 encoding algorithm*[23]. Nechť máme vstupní zprávu: "*ABBCBCABABCAABCAAB*". Celkově tato zpráva disponuje délkou o 18 znacích, z nichž jsou pouze 3 unikátní, a také se v ní opakují podřetězce s více než 1 znakem, např. "*BCAA*". Postup algoritmu je zobrazen v obrázku 14.

V **první iteraci** se načte první znak `A`. Od tohoto znaku (včetně) se hledá nejdelší prefix. Jelikož je slovník inicializován prázdný tak nejdelší prefix nacházející se ve slovníku je prázdný řetězec `""`. Znak, který následuje po prefixu je `A`. Tento znak se přidá na konec nalezeného prefixu. Výsledný řetězec `A` má délku `1` a tedy se zakóduje do následující n-tice: `(0, A)` (nalezený prefix `""` se ve slovníku nenachází a tedy mu bude stanovena prohibitivní hodnota: `0`). Tato n-tice se uloží na pozici `1` do slovníku a zapíše se na výstup:

`(0, A)`

Algoritmus se přesune na další znak po výsledném řetězci: `B`.

Ve **druhé iteraci** je aktuální znak `B` (index v textu: `1`). Od tohoto znaku (včetně) je nejdelší prefix, který se nachází ve slovníku, prázdný řetězec `""`. Znak, který následuje po prefixu je `B`. Tento znak se přidá na konec nalezeného prefixu. Výsledný řetězec `B` má délku `1` a tedy se zakóduje do následující n-tice: `(0, B)` (nalezený



	Dictionary	
output	index	string
(0, A)	1	A
(0, B)	2	B
(2, C)	3	BC
(3, A)	4	BCA
(2, A)	5	BA
(4, A)	6	BCAA
(6, B)	7	BCAAB

Obrázek 14: Postup komprese dle algoritmu LZ78. Každá šipka představuje iteraci algoritmu. Převzatý z [23].

prefix `""` se ve slovníku nenachází a tedy mu bude stanovena prohibitivní hodnota: `0`). Tato n-tice se uloží na pozici `2` do slovníku a zapíše se na výstup:

```
(0, A), (0, B)
```

Algoritmus se přesune na další znak po výsledném řetězci: `B`.

Ve **třetí iteraci** je aktuální znak `B` (index v textu: `2`). Od tohoto znaku (včetně) je nejdelší prefix, který se nachází ve slovníku, znak `B`. Znak, který následuje po prefixu je `C`. Tento znak se přidá na konec nalezeného prefixu. Výsledný řetězec `BC` má délku `2` a tedy se zakóduje do následující n-tice: `(2, C)` (nalezený prefix `B` se ve slovníku nachází na indexu `2`). Tato n-tice se uloží na pozici `3` do slovníku a zapíše se na výstup:

```
(0, A), (0, B), (2, C)
```

Algoritmus se přesune na další znak po výsledném řetězci: `B`.

Ve **čtvrté iteraci** je aktuální znak `B` (index v textu: `4`). Od tohoto znaku (včetně) je nejdelší prefix, který se nachází ve slovníku, podřetězec `BC`. Znak, který následuje po prefixu je `A`. Tento znak se přidá na konec nalezeného prefixu. Výsledný řetězec `BCA` má délku `3` a tedy se zakóduje do následující n-tice: `(3, A)` (nalezený prefix `BC` se ve slovníku nachází na indexu `3`). Tato n-tice se uloží na pozici `4` do slovníku a zapíše se na výstup:

```
(0, A), (0, B), (2, C), (3, A)
```

Algoritmus se přesune na další znak po výsledném řetězci: `B`.

V **páté iteraci** je aktuální znak `B` (index v textu: `7`). Od tohoto znaku (včetně) je nejdelší prefix, který se nachází ve slovníku, znak `B`. Znak, který následuje po prefixu je `A`. Tento znak se přidá na konec nalezeného prefixu. Výsledný řetězec `BA` má délku `2` a tedy se zakóduje do následující n-tice: `(2, A)` (nalezený prefix `B` se ve slovníku nachází na indexu `2`). Tato n-tice se uloží na pozici `5` do slovníku a zapíše se na výstup:

```
(0, A), (0, B), (2, C), (3, A), (2, A)
```

Algoritmus se přesune na další znak po výsledném řetězci: `B`.

V **šesté iteraci** je aktuální znak `B` (index v textu: `9`). Od tohoto znaku (včetně) je nejdelší prefix, který se nachází ve slovníku, podřetězec `BCA`. Znak, který následuje po prefixu je `A`. Tento znak se přidá na konec

nalezeného prefixu. Výsledný řetězec `BCAA` má délku `4` a tedy se zakóduje do následující n-tice: `(4, A)` (nalezený prefix `BCA` se ve slovníku nachází na indexu `4`). Tato n-tice se uloží na pozici `6` do slovníku a zapíše se na výstup:

`(0, A), (0, B), (2, C), (3, A), (2, A), (4, A)`

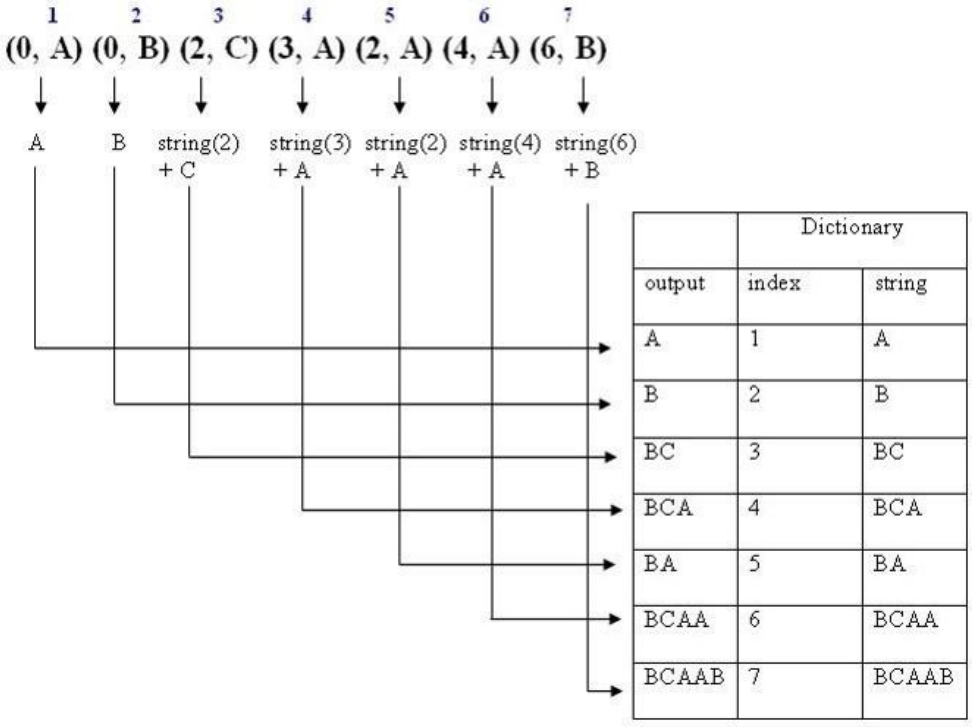
Algoritmus se přesune na další znak po výsledném řetězci: `B`.

V **sedmé iteraci** je aktuální znak `B` (index v textu: `13`). Od tohoto znaku (včetně) je nejdelší prefix, který se nachází ve slovníku, podřetězec `BCAA`. Znak, který následuje po prefixu je `B`. Tento znak se přidá na konec nalezeného prefixu. Výsledný řetězec `BCAAB` má délku `5` a tedy se zakóduje do následující n-tice: `(6, B)` (nalezený prefix `BCAA` se ve slovníku nachází na indexu `6`). Tato n-tice se uloží na pozici `7` do slovníku a zapíše se na výstup:

`(0, A), (0, B), (2, C), (3, A), (2, A), (4, A), (6, B)`

Algoritmus se přesune na další znak, což je ale prázdný znak (konec souboru) a tedy algoritmus končí. Výstup v poslední iteraci je zakódovaný vstupní text.

Algoritmus dekomprese Postup dekomprese pro soubory komprimované pomocí algoritmu LZ78 spočívá v načtení zakódovaného vstupu a postupném sestavování slovníku pro text podle "instrukcí"na vstupu. Postup bude předveden na předem zakódovaném textu z příkladu v obrázku 15.



Obrázek 15: Postup dekomprese dle algoritmu LZ78. Každá šipka představuje krok algoritmu. Převzatý z [23].

V **prvním kroku** ze vstupu načítáme `(0, A)`. Index prefixu je prohibitivní hodnota `0`, tedy se žádný předchozí prefix nenačítá. Do slovníku se na pozici `1` zapisuje znak `A`. Na konec výstupu se zapisuje znak `A`. Toto se opakuje ve druhém kroku pro znak `B` (do slovníku se zapisuje na pozici `2`).

Ve **třetím kroku** ze vstupu načítáme `(2, C)`. Index prefixu je `2`, tedy použijeme prefix ze slovníku s indexem `2`: `B`. Na konec prefixu ze slovníku se zapíše znak `C` a výsledný řetězec `BC` se zapíše do slovníku na pozici `3` a na konec výstupu.

Ve **čtvrtém kroku** ze vstupu načítáme `(3, A)`. Index prefixu je `3`, tedy použijeme prefix ze slovníku s indexem `3`: `BC`. Na konec prefixu ze slovníku se zapíše znak `A` a výsledný řetězec `BCA` se zapíše do slovníku na pozici `4` a na konec výstupu.

V **pátém kroku** ze vstupu načítáme `(2, A)`. Index prefixu je `2`, tedy použijeme prefix ze slovníku s indexem `2`: `B`. Na konec prefixu ze slovníku se zapíše znak `A` a výsledný řetězec `BA` se zapíše do slovníku na pozici `5` a na konec výstupu.

V **šestém kroku** ze vstupu načítáme `(4, A)`. Index prefixu je `4`, tedy použijeme prefix ze slovníku s indexem `4`: `BCA`. Na konec prefixu ze slovníku se zapíše znak `A` a výsledný řetězec `BCAA` se zapíše do slovníku na pozici `6` a na konec výstupu.

V **sedmém kroku** ze vstupu načítáme `(6, B)`. Index prefixu je `6`, tedy použijeme prefix ze slovníku s indexem `6`: `BCAA`. Na konec prefixu ze slovníku se zapíše znak `B` a výsledný řetězec `BCAAB` se zapíše do slovníku na pozici `7` a na konec výstupu. Další data na vstupu už nejsou, tedy je konec algoritmu a máme dekódovaný text:

`ABBCBCABABCAABCAAB`

2.6 LZW

LZW je kompresní algoritmus, který byl v roce 1984 představen panem Welchem jako modifikace algoritmu LZ78. Název tohoto algoritmu je akronym jmen tvůrců Lempel a Ziv, a modifikátora Welche. Podobně jako algoritmus LZ78 se jedná o slovníkovou kompresní metodu, tedy se ve vstupním textu hledají opakující se řetězce. Rozdíl ale tkví v tom, že je slovník inicializován před jakýmkoliv načítání dat. Díky této změně se každý znak (případně řetězec, pokud se opakují znaky) ze vstupního souboru dá reprezentovat pouze jedním číslem.
Popis algoritmu bude v rámci této práce předpokládat, že vstupní data jsou v podobě textového řetězce.

Algoritmus komprese

1. Inicializuje se slovník s množinou o 256 základních znacích (jedná-li se o 8 bitovou implementaci). Např. jako součást této množiny může být abeceda *a, b, c, d* atd.
2. Ze vstupního textu se načte 8 prvních bitů (tj. nějaký znak, např. 'a', 'b').
3. Počínaje od aktuálního znaku (včetně) se hledá nejdelší prefix, který se vyskytuje ve slovníku.
4. Na výstup se vypíše index prefixu ve slovníku.
5. Načte se další znak ze vstupního textu. Tento znak se přidá na konec onoho prefixu. Pokud neexistuje další znak, tak je konec souboru a algoritmus končí.
6. Tento výsledný řetězec se nevyskytuje ve slovníku, a tak se přidá na konec slovníku s indexem o 1 vyšší než dosud nejvyšší index ve slovníku.
7. Algoritmus se přesune na poslední znak řetězce a pokračuje na krok 3.

Příklad Pro znázornění algoritmu bude uveden ilustrační příklad z *LZW Data Compression*[25]. Nechť máme vstupní zprávu: "*banana_bandana*". Celkově tato zpráva disponuje délkou o 14 znacích, z nichž je pouze 5 unikátní, a také se v ní opakují podřetězce s více než 1 znakem, např. "*ana*". Postup algoritmu je zobrazen v tabulce 11.
Nechť dále máme inicializovaný slovník, viz tabulka 10.

Index	Znak
0	a
1	b
2	d
3	n
4	_

Tabulka 10: Iniciální slovník obsahující pouze individuální znaky ze vstupního textu (pro jednoduchost).

Algoritmus dekomprese Postup dekomprese pro soubory komprimované pomocí algoritmu LZW spočívá v inicializování identického počátečního slovníku, jako při kompresi (tj. slovník s množinou o 256 základních znacích). Následně se načítá zakódovaný vstup a postupně se sestavuje slovník pro text podle "instrukcí"na vstupu. Postup bude předveden na předem zakódovaném textu z příkladu v tabulce 12.

Zajímavosti V praxi se často určuje maximální velikost slovníku, aby algoritmus nezabíral příliš mnoho místa v paměti. Roku 1987 se LZW stal populárnější, jelikož se stal součástí formátu *GIF*. Případně je také občas používán při kompresi *TIFF* nebo *PDF* souborů, avšak *Adobe Acrobat* spíše používá *DEFLATE* na většinu kompresních činností.

Krok	Vstup	Řetězec	Je ve slovníku	Výstup	Nově do slovníku "záznam: index"
1	ba	ba	ne	1	ba: 5
2	ban	an	ne	1,0	an: 6
3	bana	na	ne	1,0,3	na: 7
4	banana	ana	ne	1,0,3,6	ana: 8
5	banana_	a_	ne	1,0,3,6,0	a_: 9
6	banana_b	_b	ne	1,0,3,6,0,4	_b: 10
7	banana_ban	ban	ne	1,0,3,6,0,4,5	ban: 11
8	banana_band	nd	ne	1,0,3,6,0,4,5,3	nd: 12
9	banana_banda	da	ne	1,0,3,6,0,4,5,3,2	da: 13
10	banana_bandana	ana	ano	1,0,3,6,0,4,5,3,2,8	žádný

Tabulka 11: Iterace postupu komprese vstupního textu "*abracadabrad*" pomocí algoritmu LZW.

Krok	Zakódovaný vstup	Předklad	Dekódovaný výstup	Aktuální řetězec	Nově do slovníku "záznam: index"
1	1	1 = b	b	1	nic
2	1,0	0 = a	ba	1,0	ba: 5
3	1,0,3	3 = n	ban	1,0,3	an: 6
4	1,0,3,6	6 = an	banan	1,0,3,6	na: 7
5	1,0,3,6,0	0 = a	banana	1,0,3,6,0	ana: 8
6	1,0,3,6,0,4	4 = _	bananab_	1,0,3,6,0,4	a_: 9
7	1,0,3,6,0,4,5	5 = ba	banana_ba	1,0,3,6,0,4,5	_b: 10
8	1,0,3,6,0,4,5,3	3 = n	banana_ban	1,0,3,6,0,4,5,3	ban: 11
9	1,0,3,6,0,4,5,3,2	2 = d	banana_band	1,0,3,6,0,4,5,3,2	nd: 12
10	1,0,3,6,0,4,5,3,2,8	8 = ana	banana_bandana	1,0,3,6,0,4,5,3,2,8	da: 13

Tabulka 12: Iterace postupu dekomprese vstupního textu "*abracadabrad*" komprimovaného pomocí algoritmu LZW.

3 Implementace a výsledky benchmarku

Pro účel porovnání časů běhů a kompresních poměrů byly algoritmy zmíněné na začátku sekce 2 (kromě algoritmu LZW), spolu s generátorem datových souborů typu CSV implementovány. Implementace byla provedena v programovacím jazyce Python verze 3.9.0 (benchmark spouštěn pomocí *pypy3*). Generovaná data obsahovala náhodně generovaný text Lorem Ipsum a náhodně generované souřadnice v okolí České republiky. Kompresní algoritmy byli benchmarkovány na vygenerovaném CSV souboru o velikosti ∼100 MB (∼360 tis. řádků). Výsledné časy a kompresní poměry ve vteřinách jsou zaznamenány v tabulce 13.

Algoritmus	Čas komprese [s]	Kompresní poměr [%]	Vstupní text [MB]	Výstupní text [MB]
RLE	27.86	393.83	99.66	392.49
SHF	14.74	71.19	99.66	70.94
HUFF	13.26	58.68	99.66	58.48
FGK	60.90	58.70	99.66	58.50
VTTR	158.45	58.69	99.66	58.49
LZ77	101.45	48.63	99.66	48.47
LZ78	47.56	33.93	99.66	33.82

Tabulka 13: Výsledky benchmarku algoritmů na souboru o velikosti 100 MB (velikost souboru podle Windows je přesně 100MB, vstupní text podle UTF-8 v Pythonu má 99.66MB). Nejlepší (resp. nejhorší) kompresní poměr a čas v benchmarku je zvýrazněn zeleně (resp. červeně).

3.1 Shrnutí výsledků benchmarku

Ze všech implementovaných algoritmů se na vygenerovaných datech celkově jako nejefektivnější kompresní algoritmus ukázal LZ78. Patrně horšího kompresního poměru dosáhl algoritmus LZ77, dokonce s horší časovou náročností. Oba tyto algoritmy dosáhly velice dobrých výsledků z důvodu nedokonalosti náhodného generátoru slov Lorem Ipsum, který ve vstupním textu zdánlivě používá poměrně malou množinu slov, které se vyskytují pouze v zaměněném pořadí a případně s velkým prvním písmenem. Tento fakt je výhodou obzvlášť pro LZ78, jelikož jakmile ve svém slovníku obsahuje většinu slov používaných generátorem Lorem Ipsum, tak už je jen potřeba na jejich výskyt ve slovníku odkazovat. Huffmanovo kódování a jeho adaptivní modifikace dosáhli poměrně podobného kompresního poměru, avšak za zhoršujících se časů. Nejhorší algoritmus v rámci kompresního poměru bylo RLE kódování, z důvodu přítomnosti velice malého množství po sobě jdoucích znaků v generovaném souboru. V rámci RLE je kompresní poměr zhoršen od očekávaného, jelikož původní implementace RLE algoritmu nepočítala s číslicemi ve vstupních datech a tedy bylo nutné přidat dělicí znak do zakódovaného výstupu, který odděloval četnosti znaků a samotné znaky. Obecně pomalé implementace nastaly kvůli nedostatečným zkušenostem programátora a následným nevhodným zvolením programovacího jazyka. Vyřešení tohoto problému by spočívalo buď v implementaci všech algoritmů s použitím podobných datových struktur, nebo v použití jiného programovacího jazyka.

4 Shrnutí

Pro efektivní práci s daty je vhodné používat metody kompresí pro zmenšení místa, které tyto data zabírají na úložištích a při přenosu sítí.
Pro kompresi dat, která obsahují velké množství částí, kde se po sobě opakují stejné znaky je vhodné použít *RLE algoritmus*, který ukládá počet opakování určitých znaků po sobě jdoucích. Pro kompresi dat, ve kterých se unikátní znaky vyskytují jak ve větším, tak v menším množství je vhodné použít Shannon-Fano algoritmus kódování, jelikož pro znaky vytvoří slovník kódů, ve kterém nejčastěji vyskytující se znaky v datech budou reprezentovány nejkratším binárním kódem a tedy i komprimovaná data budou kratší než původní. Velice podobná alternativa k Shannon-Fano kódování je algoritmus Huffmanova kódování nebo jedna z jeho modifikací. Pokud je nutné komprimovat soubory, které jsou poměrně velké a obsahují více výskytů i delších slov, tak je vhodné použít algoritmu LZ78 nebo LZW, jelikož jakmile si dané slovo uloží do slovníku, tak bude při dalším výskytu reprezentováno indexem z onoho slovníku a ušetří se tak značně místa.

Reference

[1] Hilbert, M., López, P. (2011). *The World's Technological Capacity to Store, Communicate, and Compute Information*. Science, 332(6025), 60 – 65. URL: http://www.martinhilbert.net/WorldInfoCapacity.html/

[2] Večerka, A. 2008. *Komprese dat*. [online]. Učební text. Olomouc: Katedra Informatiky, Přírodovědecká fakulta, Univerzita Palackého [cit. 2021-04-12]. URL: https://phoenix.inf.upol.cz/esf/ucebni/komprese.pdf

[3] Silwa, C., Crocetti, P. (ed.). 2017. *Data Compression*. SearchStorage [online] [cit. 2021-04-10]. URL: https://searchstorage.techtarget.com/definition/compression

[4] *Bezeztrátová komprese*. 2001-. Wikipedia: the free encyclopedia [online] [cit. 2021-04-10]. San Francisco (CA): Wikimedia Foundation. URL: https://cs.wikipedia.org/wiki/Bezeztr%C3%A1tov%C3%A1_komprese

[5] *Ztrátová komprese*. 2001-. Wikipedia: the free encyclopedia [online] [cit. 2021-04-10]. San Francisco (CA): Wikimedia Foundation. URL: https://cs.wikipedia.org/wiki/Ztr%C3%A1tov%C3%A1_komprese

[6] Cvrček, V. 2013. *Zipfovy zákony*. Wiki: Český národní korpus [online] cit. 2015-01-03]. URL: http://wiki.korpus.cz/doku.php/pojmy:zipf

[7] Svetelska, H., Hrychová, T. (ed.). 2015. *Zipfovy zákony*. Wikisofia [online] [cit. 2021-04-11]. URL: https://wikisofia.cz/wiki/Zipfovy_z%C3%A1kony

[8] Radboud University. 2017. *Unzipping Zipf's Law: Solution to a century-old linguistic problem*. Phys Org [online] [cit. 2021-04-10]. URL: https://phys.org/news/2017-08-unzipping-zipf-law-solution-century-old.html

[9] E. M. 2012. *George Kingsley Zipf*. WikiKnihovna [online] [cit. 2021-04-11]. URL: https://wiki.knihovna.cz/index.php/George_Kingsley_Zipf

[10] Tvorogov, I., P. Rajmic. 2016. *RLE – kódování délkou běhu*. Ústav telekomunikací, FEKT, VUT [online] [cit. 2021-04-11]. Brno. URL: https://www.utko.fekt.vut.cz/~rajmic/applets/rle_kod/RLE_zakladni.html

[11] *RLE algoritmus: popis, vlastnosti, pravidla a příklady*. 2021. CZE Kagutech [online] [cit. 2021-04-11]. URL: https://cze.kagutech.com/4241004-rle-algorithm-description-features-rules-and-examples

[12] *Run-length encoding*. 2001-. Wikipedia: the free encyclopedia [online] [cit. 2021-04-11]. San Francisco (CA): Wikimedia Foundation. URL: https://cs.wikipedia.org/wiki/Run-length_encoding

[13] Tišnovský, P. 2006. *PCX prakticky - implementace komprimace RLE*. ROOT CZ [online] [cit. 2021-04-11]. URL: https://www.root.cz/clanky/pcx-prakticky-implementace-komprimace-rle/

[14] Cerna, M. 2018. *Shannon-Fanovo kódování*. Wikisofia [online] [cit. 2021-04-11]. URL: https://wikisofia.cz/wiki/Shannon-Fanovo_k%C3%B3dov%C3%A1n%C3%AD.

[15] *Shannonovo–Fanovo kódování*. 2001-. Wikipedia: the free encyclopedia [online] [cit. 2021-04-12]. San Francisco (CA): Wikimedia Foundation. URL: https://cs.wikipedia.org/wiki/Shannonovo%E2%80%93Fanovo_k%C3%B3dov%C3%A1n%C3%AD

[16] Hordějčuk, V. 2021. *Shannon-Fanovo kódování*. Voho eu [online] [cit. 2021-04-12]. URL: http://voho.eu/wiki/kodovani-shannon-fano/

[17] Hordějčuk, V. *Huffmanovo kódování*. URL: http://voho.eu/wiki/kodovani-huffman/

[18] Huffman, D. (1952). *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the IRE. 40 (9): 1098–1101. URL: http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

[19] Van Leeuwen, J. (1976). *On the construction of Huffman trees*. ICALP: 382–410. URL: https://webspace.science.uu.nl/~leeuw112/huffman.pdf

[20] Balík, M., Hanuš, M., Holub, J., Paulíček, M. *Přehled algoritmů - statické metody*. Knihovna vizualizačních appletů pro kompresi dat [online] [cit. 2021-04-19]. URL: http://www.stringology.org/DataCompression/obsah.html#statistic_compress_methods

[21] Hordějčuk, P. 2008 - 2021. *Kompresní algoritmus LZ77*. Voho eu [online] [cit. 2021-4-29]. URL: http://voho.eu/wiki/algoritmus-lz77/

[22] asymmetriq. *LZ77 compression algorithm (general code efficiency)*. StackExchange [online] [cit. 2021-4-29]. URL: https://codereview.stackexchange.com/questions/233865/lz77-compression-algorithm-general-code-efficiency

[23] *LZ78 encoding algorithm*. 2021. Programmer Sought [online] [cit. 2021-5-3]. URL: https://www.programmersought.com/article/2333222976/

[24] *LZ77 and LZ78*. 2001-. Wikipedia: the free encyclopedia [online] [cit. 2021-5-3]. San Francisco (CA): Wikimedia Foundation. URL: https://en.wikipedia.org/wiki/LZ77_and_LZ78

[25] Bhat, S. *LZW Data Compression*. CS Duke University [online] [cit. 2021-5-3]. URL: https://www2.cs.duke.edu/csed/curious/compression/lzw.html

[26] *Lempel–Ziv–Welch*. 2001-. Wikipedia: the free encyclopedia [online] [cit. 2021-5-3]. San Francisco (CA): Wikimedia Foundation. URL: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch.