FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Slovníkové kompresní metody
Student:	Matěj Javorka
Vedoucí:	Ing. Tomáš Pecka
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Seznamte se s principem slovníkových kompresních metod a algoritmy rodiny LZ.
Proveďte analýzu a implementaci alespoň čtyř algoritmů rodiny LZ.
Algoritmy implementujte tak, aby si uživatel mohl vybrat mezi zkomprimovanými daty v binární podobě a výstupem čitelným pro člověka, který je vhodný pro demonstraci výsledků daných algoritmů při výuce.
Implementaci vhodně otestujte.
Proveďte měření rychlosti a efektivity komprese na vhodných datových korpusech.

Seznam odborné literatury:
[1] SALOMON, David. Data Compression: The Complete Reference. 4th ed. London: Springer, 2007. ISBN 978-1-84628-603-2.
[2] SAYOOD, Khalid. Introduction to Data Compression. 3rd ed. San Francisco: Elsevier, 2006. ISBN 978-0-12-620862-7.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 2. února 2023 v Praze.

Bakalářská práce

SLOVNÍKOVÉ KOMPRESNÍ METODY

Matěj Javorka

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: Ing. Tomáš Pecka
10. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Rád bych poděkoval Ing. Tomášovi Peckovi za jeho čas, vstřícnost, cennou pomoc, ochotu a odborné vedení této bakalářské práce. Dále bych chtěl poděkovat středisku ELSA ČVUT za zastoupení studentů, kteří v životě neměli mnoho štěstí. Na závěr bych rád poděkoval své rodině za její soustavnou pomoc a oporu, díky které jsem mohl jít studovat a mohl si tak přese všechny překážky a nepřízně osudu jít plnit svůj sen.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 10. května 2023 Matěj Javorka

Abstrakt

Náplní této bakalářské práce je analýza, návrh a implementace slovníkových kompresních metod LZ77, LZ78, LZSS a LZW.

V literární rešerši jsou popsány základní pojmy, principy a metody komprese dat. Podrobně se literární rešerše věnuje slovníkovým metodám komprese dat, konkrétně analýzou algoritmů LZ77, LZ78, LZSS a LZW, společně s popisem vhodných datových struktur používaných v těchto algoritmech.

Praktická část práce se zaobírá návrhem a implementací těchto algoritmů v programovacím jazyce C++. Implementace umožňuje, aby bylo možné volit mezi komprimovaným a výukovým výstupem, tj. zda výstupem komprese mají být komprimovaná nečitelná binární data, nebo textová data čitelná pro člověka, která reprezentují běh algoritmu a usnadňují tak jeho pochopení. Dále jsou navrženy a implementovány testy těchto algoritmů a jsou zvoleny vhodné datové korpusy, na kterých je změřena doba běhu komprese, dekomprese a kompresní poměr implementovaných algoritmů. Výsledky detailních měření jsou uvedeny v příloze této práce.

Hlavním výstupem práce je implementace kompresních algoritmů v podobě konzolových aplikací, které lze použít pro reálnou kompresi dat i jako pomocný nástroj pro výuku kompresních algoritmů v předmětech zabývajících se kompresí dat, jako je magisterský předmět NI–KOD na FIT ČVUT.

Klíčová slova komprese dat, dekomprese dat, slovníkové metody komprese dat, konzolová aplikace, implementace slovníkových metod komprese dat, měření efektivity komprese, LZ77, LZ78, LZSS, LZW, C++

Abstract

The subject of this bachelor thesis is the analysis, design and implementation of dictionary compression methods LZ77, LZ78, LZSS and LZW.

The literature research describes the basic concepts, principles and methods of data compression. It then discusses dictionary-based data compression methods in greater detail, specifically analysis of the LZ77, LZ78, LZSS, and LZW algorithms, along with a description of the appropriate data structures used with these algorithms.

The practical part of the thesis deals with the design and implementation of these algorithms in the C++ programming language. The implementation makes it possible to choose between compressed and learning output, i.e. whether the compression output should be compressed unreadable binary data or human-readable textual data that represent the run of the algorithm and thus facilitate its understanding. Furthermore, tests of these algorithms are designed and implemented, and suitable data corpora are selected on which the compression time, decompression time and compression ratio of the implemented algorithms are measured. The results of the detailed measurements are presented in the appendix of this paper.

The main output of the thesis is the implementation of compression algorithms in the form of console applications, which can be used for real data compression as well as an auxiliary tool for teaching compression algorithms in courses dealing with data compression, such as the master's course NI–KOD at FIT CTU.

Keywords data compression, data decompression, dictionary methods of data compression, console application, implementation of dictionary compression methods, measurements of compression effectivity, LZ77, LZ78, LZSS, LZW, C++

Seznam zkratek

LZ77	Lempel–Ziv 77
LZ78	Lempel–Ziv 78
LZSS	Lempel–Ziv–Storer–Szymanski
LZW	Lempel–Ziv–Welch

Úvod

> *„For me, data compression is more than a manipulation of numbers; it is the process of discovering structures that exist in the data.“*
>
> *—Khalid Sayood, Introduction to Data Compression*

Teoretická informatika je významným a vlivným oborem počítačových věd, neboť tvoří stavební kameny pro studium výpočetních problémů a algoritmů, identifikuje klíčové problémy v nových oblastech informatiky a díky tomu její výzkum pokládá základy potřebné pro vývoj technologií budoucnosti. Totéž platí i pro kompresi dat jako jednu z disciplín teoretických počítačových věd.

Komprese dat je jednou z klíčových vědeckých oblastí, které umožnily probíhající digitální multimediální revoluci a stala se nedílnou, téměř všudypřítomnou součástí každodenního používání digitálních technologií. Používáme ji, když si pustíme oblíbenou písničku na Spotify, epizodu seriálu na Netflixu, video na Youtube, film na DVD nebo Blu-ray disku, rádio nebo digitální televizní stanici. Kompresi dat používáme, když telefonujeme přes mobilní síť, když se účastníme videokonference, při pořizování fotografií z digitální kamery nebo chytrého telefonu, nebo při zálohování většího množství dat na lokální nebo cloudové úložiště.

S narůstajícím objemem dat, která používáme a se kterými přicházíme do styku, bude potřeba použití efektivních kompresních algoritmů nadále růst v osobní i ve firemní sféře. Komprimovaná data zabírají na úložištích méně místa a vyžadují méně času pro přenos po síti, což může vést ke zvýšení produktivity i zásadním úsporám nákladů firmy, která nemusí investovat finanční prostředky do renovace datové a komunikační infrastruktury.

Tato práce přibližuje problematiku komprese, seznamuje čtenáře se základními pojmy a používanými metodami komprese dat. Detailně pak probírá slovníkové metody komprese dat a analyzuje kompresní algoritmy LZ77, LZ78, LZSS a LZW. V praktické části je popsána implementace těchto algoritmů v jazyce C++, jsou vylíčeny problémy, na které lze při implementaci narazit a jak lze tyto problémy řešit. Pro porovnání efektivity komprese, času komprese a dekomprese implementovaných algoritmů jsou zvoleny vhodné datové korpusy, na kterých jsou tyto údaje změřeny.

Komprese dat je rozsáhlý obor teoretické informatiky plný fascinujících algoritmů, které však nejsou triviální na pochopení. Proto řešení implementované v rámci této práce umožňuje zvolit, aby výstupem kompresních algoritmů místo komprimovaných nečitelných binárních dat byla data textová, která reprezentují běh kompresních algoritmů, včetně možnosti zvolit vypsání vnitřních stavů použité slovníkové datové struktury. Tento výukový textový výstup najde využití nejen mezi studenty a vyučujícími předmětů zabývajících se kompresí dat, jako je např. magisterský předmět NI–KOD na FIT ČVUT, ale i mezi čtenáři se zájmem o kompresní algoritmy z řad široké veřejnosti a snad bude díky tomu tato práce i jiskrou, která ve čtenářích zažehne nadšení pro kompresi dat i touhu věnovat se tomuto oboru nadále a detailněji.

Kapitola 1
Cíle práce

Cílem této práce je seznámení se slovníkovými kompresními metodami, analýza algoritmů LZ77, LZ78, LZSS a LZW, a návrh a implementace těchto algoritmů v binárním a čitelném režimu v programovacím jazyce C++. Cíle práce bude dosaženo řešením několika vzájemně souvisejících dílčích cílů.

Prvním dílčím cílem je popis slovníkových kompresních metod a analýza algoritmů LZ77, LZ78, LZSS a LZW.

Druhým dílčím cílem je implementace těchto algoritmů v programovacím jazyce C++ a implementace testů pro ověření korektnosti komprese, dekomprese a implementovaných datových struktur.

Dalším dílčím cílem je implementace textového výstupu čitelného pro člověka, který bude reprezentovat běh kompresních algoritmů s cílem usnadnění jejich pochopení.

Posledním dílčím cílem je změření efektivity komprese, délky komprese a délky dekomprese implementovaných algoritmů na vhodných datových korpusech.

Kapitola 2
Základní pojmy

Tato kapitola se zabývá popisem a vysvětlením základních pojmů používaných v teorii kompresi dat, způsoby měření kvality komprese a vysvětlením některých limitů bezztrátové komprese.

2.1 Teorie komprese dat

Komprese je proces kódování struktury vstupních dat do výstupních dat tak, aby výstupní data měla menší velikost. Existuje mnoho různých metod komprese, které jsou postaveny na různých myšlenkách, přístupech a poskytují různé výsledky, všechny z nich ale staví na stejném principu a tím je dosažení komprimace odstraněním nadbytečných informací neboli *redundance* ze vstupních dat. Veškerá data, která nejsou čistě náhodná, mají určitou strukturu, a této struktury lze využit k vytvoření zmenšené reprezentace vstupních dat, ve které tato struktura již není rozpoznatelná. [1]

Struktura vstupních dat však není to jediné, čeho lze využít k dosažení komprese. Využít lze také vlastností uživatele těchto dat. Například při zpracovávání, přenosu nebo ukládání řeči či obecně audiovizuálních dat jsou tato data určena pro člověka, který má omezené schopnosti vnímání. Nedává proto smysl uchovávat v datech informace, které uživatel není schopen vnímat, jako jsou například zvuky vysokých frekvencí. [2] Této myšlenky využívá např. ztrátový formát pro kódování audia MP3.

Kompresi lze tedy chápat také jako efektivní reprezentaci digitálních vstupních dat, jako jsou textová, zvuková a audiovizuální data. Cílem komprese je vytvoření digitální reprezentace vstupu za pomoci co nejmenšího možného počtu bitů tak, aby byly splněny minimální požadavky na rekonstrukci původních dat. [3]

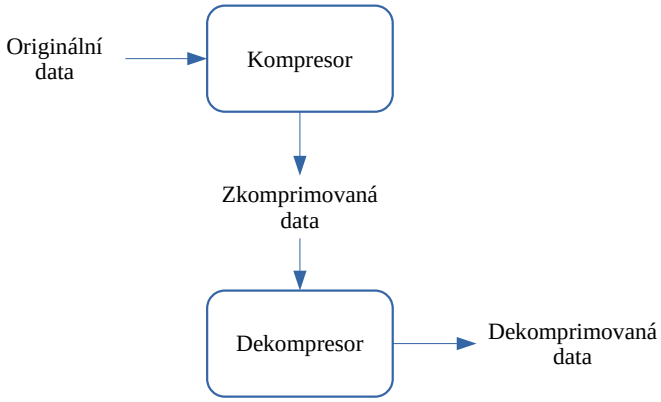
Obecné schéma komprese a dekomprese dat ukazuje obrázek 2.1. Aby bylo možné kódování, kód či redundanci definovat formálně, je potřeba nejprve uvést několik základních pojmů z teorie formálních jazyků, teorie množin, teorie pravděpodobnosti a teorie informace.

▶ **Definice 2.1** (Abeceda [4]). *Abeceda* Σ *je konečná množina prvků, které se nazývají symboly abecedy.*

Velikost abecedy je rovna počtu symbolů v abecedě a značíme ji $|\Sigma|$. Volba abecedy záleží na konkrétní problémové doméně.

▶ **Příklad 2.2** (Abeceda). Následuje několik příkladů, jak může abeceda vypadat.

- $\Sigma_1 = \{0, 1\}$ je abeceda, která obsahuje symboly 0, 1. Velikost této abecedy je 2.
- $\Sigma_2 = \{a, b, c\}$ je abeceda, která obsahuje symboly a, b, c. Platí, že $|\Sigma_2| = 3$.



Obrázek 2.1 Schéma komprese [3]

- $\Sigma_3 = \{if, then, exit, true, false\}$ je abeceda, která obsahuje pět symbolů: *if*, *then*, *exit*, *true* a *false*. Velikost této abecedy je 5.
- $\Sigma_4 = \{(r, g, b) \mid r, g, b \in \mathbb{Z} \wedge 0 \leq r, g, b \leq 255\}$ je abeceda, jejíž symboly jsou uspořádané trojice celých čísel nabývajících hodnot 0 až 255. Pro velikost abecedy platí, že $|\Sigma_4| = 256 \cdot 256 \cdot 256 = 256^3 = 16777216$.
- $\Sigma_5 = \{a \mid a \in \mathbb{Z} \wedge 0 \leq a \leq 2^{16} - 1\}$ je abeceda, jejíž symboly jsou všechna 16-bitová celá čísla bez znaménka. Je zřejmé, že pro velikost této abecedy platí $|\Sigma_5| = 2^{16}$.

▶ **Definice 2.3** (Řetězec nad abecedou [4]). *Nechť Σ je abeceda. Řetězcem nad abecedou Σ se rozumí konečná posloupnost symbolů abecedy Σ.*

Prázdná posloupnost symbolů se nazývá *prázdný řetězec* a značí se ϵ. Množina všech možných řetězců nad abecedou Σ se značí Σ^*. Tato množina vždy obsahuje prázdný řetězec a pro neprázdnou abecedu je vždy nekonečná. Množina všech možných neprázdných řetězců nad Σ se značí Σ^+, tato množina je pro $|\Sigma| \neq 0$ též nekonečná. Platí, že $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. [4]

Délka řetězce x je rovna počtu symbolů, kterými je řetězec tvořen a označuje se $|x|$. Pro libovolný řetězec x platí, že $|x| \geq 0$, pro libovolný neprázdný řetězec y platí $|y| > 0$, a pro prázdný řetězec ϵ platí $|\epsilon| = 0$. [4]

Pro účely tohoto textu bude potřeba definovat pojem délky řetězce vůči konkrétní abecedě.

▶ **Definice 2.4** (Délka řetězce vůči abecedě). *Nechť x je libovolný řetězec nad abecedou Σ, tj. $x \in \Sigma^*$. Délka řetězce vůči abecedě Σ je rovna počtu symbolů abecedy Σ, ze kterých se řetězec x skládá a značí se $|x|_\Sigma$.*

▶ **Příklad 2.5** (Délka řetězce vůči abecedě). Nechť Σ, Π jsou abecedy takové, že $\Sigma = \{a, b, c\}$ a $\Pi = \{aa, bb, cc\}$. Dále nechť $x = bbaacc$ je řetězec nad těmito abecedami. Pak platí, že

- $|x|_\Sigma = 6$;
- $|x|_\Pi = 3$.

Množina všech možných slov nad abecedou Σ, jejichž délka je vůči abecedě Σ rovna $n \in \mathbb{N}_0$, se značí Σ^n.

▶ **Definice 2.6** (Zřetězení [4]). *Nechť x a y jsou libovolné řetězce nad abecedou Σ, tj. $x, y \in \Sigma^*$. Výsledkem operace zřetězení řetězce x s řetězcem y je řetězec, který vznikne připojením řetězce y za řetězec x a značí se $x \cdot y$ nebo zkráceně xy.*

▶ **Definice 2.7** (Prefix [5]). *Nechť x a y jsou libovolné řetězce nad abecedou Σ, tj. $x, y \in \Sigma^*$. Řetězec x je prefixem řetězce y, pokud existuje neprázdný řetězec $z \in \Sigma^+$ takový, že $y = x \cdot z$.*

▶ **Definice 2.8** (Vlastní podmnožina [6]). *Nechť A, B jsou dvě množiny. Množina A se nazývá vlastní podmnožinou množiny B, pokud platí*

$$(A \subseteq B) \wedge (A \neq B)$$

Vlastní podmnožina se pak značí $A \subset B$.

Nyní je možné přistoupit k definici kódování a kódu. Formální definice kódování a kódu z [7], [8] a [9] bude pro potřeby tohoto textu modifikována a rozšířena, aby připouštěla i ztrátové kódování a lépe vyhovovala účelům komprese dat.

▶ **Definice 2.9** (Kódování a kód). *Nechť A, C jsou abecedy a $S \subset A^+$ je konečná vlastní podmnožina množiny všech možných neprázdných řetězců nad abecedou A. Množina S se nazývá množina zdrojových jednotek a každé zobrazení $\kappa : S \to C^*$ se nazývá kódování. Obor hodnot tohoto zobrazení $H_\kappa \subseteq C^*$ se nazývá množina kódových slov a libovolný řetězec z H_κ se nazývá kódové slovo. Každé zobrazení $\delta : H_\kappa \to A^+$ se nazývá dekódování. Uspořádaná pětice $(A, S, C, \kappa, \delta)$ se nazývá kód.*

Zobrazení κ lze přirozeně rozšířit na řetězce ze S^+ takto: nechť $u = u_1 u_2 \dots u_n \in S^n$ je řetězec rovný zřetězení prvků množiny zdrojových jednotek. Potom

$$\kappa(u) = \kappa(u_1)\kappa(u_2)\dots\kappa(u_n)$$

Stejným způsobem lze rozšířit i zobrazení δ na množinu H_κ^+, tj. na řetězce složené z kódových slov.

Tato obecná definice vyhovuje i ztrátovým kompresním algoritmům, neboť připouští ztrátu informace. Definice například připouští, aby pro nějaké kódové slovo $c \in H_\kappa$ a nějaké prvky $x_1, x_2 \dots x_n \in S$ platilo, že $\forall i \in \{1, 2, \dots n\} : \kappa(x_i) = c$. Dekódování pak může kódové slovo c dekódovat pouze na jeden z původních prvků x_i, nebo na takový prvek $y \in A^+$, který při procesu kódování nebyl vůbec použit. Definice dále připouští i situaci, kdy je prvek $x \in S$ při kódování zahozen, tj. je kódován na prázdný řetězec $\kappa(x) = \epsilon$.

Je zřejmé, že pro bezztrátovou kompresi bude κ prosté zobrazení, tj. $\forall x, y \in S : x \neq y \Rightarrow \kappa(x) \neq \kappa(y)$, a zobrazení δ bude zobrazením vůči κ inverzním, tedy $\delta = \kappa^{-1}$.

Kódy, jejichž všechna kódová slova mají stejnou délku, se označují jako kódy s *fixní délkou* (z angl. *fixed-length code*). Kódy, jejichž kódová slova mají různé délky, se označují jako kódy s *proměnnou délkou* (z angl. *variable-length code*).[2]

U kompresních algoritmů je na kód ještě vznášen požadavek, aby dekódování bylo jednoznačné. [3, 8]

▶ **Definice 2.10** (Jednoznačné dekódování [8]). *Nechť $(A, S, C, \kappa, \delta)$ je kód. Dekódování δ se označuje jako jednoznačné, pokud pro každý řetězec $c \in H_\kappa^+$ platí, že existuje právě jedna posloupnost kódových slov $c_1, c_2 \dots c_n \in H_\kappa$ taková, že $c = c_1 c_2 \dots c_n$.*

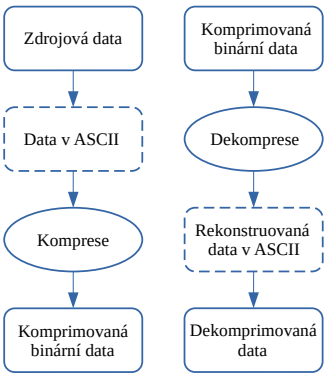
▶ **Příklad 2.11** ([3]). Nechť $A = \{a, b, c, d\}$, $C = \{0, 1\}$ jsou abecedy a kódování je definováno následovně: $\kappa(a) = 0$, $\kappa(b) = 10$, $\kappa(c) = 010$, $\kappa(d) = 101$. Dále nechť $y \in H_\kappa^+$ značí zakódovaný řetězec a platí $y = 0100101010$. Je zřejmé, že dekódování pro řetězec y může být interpretováno více než jedním způsobem, např. může být interpretováno jako:

$$\delta(y) = \delta(0100101010) = \delta(0)\delta(10)\delta(010)\delta(101)\delta(0) = abcda$$

Dekódování pro řetězec y může ale být interpretováno i takto:

$$\delta(y) = \delta(0100101010) = \delta(010)\delta(0)\delta(101)\delta(010) = cadc$$

Proto platí, že toto dekódování není jednoznačné.



Obrázek 2.2 Reprezentace vstupních dat [3]

Požadavek na jednoznačné dekódování automaticky splňují tzv. *prefixové kódy*. Pro prefixový kód platí, že žádné jeho kódové slovo není prefixem jiného kódového slova. Mezi nejznámější a nejpoužívanější prefixové kódy patří například Shannon-Fanovo kódování a Huffmanovo kódování. [1, 2]

Komprese tedy usiluje o konstrukci takového kódu $K = (A, S, C, \kappa, \delta)$, pro který bude možné vstupní řetězec $x \in A^+$ délky $|x|_A = n$ reprezentovat řetězcem $y \in C^+$, jehož délka $|y|_C = m$ je vůči abecedě C nejmenší možná.

V kontextu této práce bude platit, že abeceda A budou bajty, tj. všechna osmiciferná binární čísla v rozmezí 0–255, která lze také reprezentovat jako všechny symboly rozšířené ASCII tabulky. Prvky množiny S pak budou posloupnosti bajtů nebo také posloupnosti znaků rozšířené ASCII tabulky a abeceda $C = \{0, 1\}$ bude binární abeceda. Tuto situaci přibližuje obrázek 2.2.

Délka vstupního řetězce bude proto v tomto textu měřena v bajtech a ačkoliv výstupem komprese bude posloupnost bitů, bude pro účely měření efektivity komprese i délka výstupního binárního řetězce měřena v bajtech, protože velikost objektů uložených v počítačových pamětech se tradičně měří v bajtech. [10]

Komprese se tedy snaží najít ve vstupních datech redundanci a tu odstranit (společně se daty, které algoritmus klasifikuje jako postradatelné v případě ztrátové komprese). Výstupní zkomprimovaná data pak budou mít v optimálním případě redundanci nulovou. Pro definování pojmu nadbytečnosti informací neboli redundance je potřeba nejprve definovat samotný pojem informace. To není zcela triviální úloha, protože jde o veličinu, která je obtížně uchopitelná a měřitelná. [1]

V tomto textu bude použita definice podle Claude Shannona, jednoho ze zakladatelů teorie informace. Pro úplnost je nutné nejprve uvést několik základních pojmů z teorie pravděpodobnosti.

▶ **Definice 2.12** (Výběrový prostor [11]). *Množina všech možných výsledků daného pokusu se značí Ω a nazývá se výběrový prostor nebo také prostor elementárních jevů. Libovolný možný výsledek $\omega \in \Omega$ se nazývá elementární jev.*

▶ **Definice 2.13** (σ-algebra [11]). *Množina F podmnožin prostoru Ω se nazývá σ-algebra, jestliže jsou splněny následující podmínky:*

- $\emptyset \in F$, *kde* $\emptyset$ *značí nemožný jev;*
- $A \in F \Rightarrow A^c \in F$, *kde* $A^c = \Omega \setminus A$ *značí opačný jev;*
- $A_1, A_2, \ldots \in F \Rightarrow \bigcup_{i=1}^{\infty} A_i \in F$.

▶ Poznámka 2.14. Pro konečný či spočetný výběrový prostor Ω je nejpřímočařejší volbou pro σ-algebru množina všech podmnožin výběrového prostoru, tj. $F = 2^{\Omega}$. [11]

▶ **Definice 2.15** (Měřitelný prostor [11]). *Nechť Ω je výběrový prostor a F je σ-algebra. Uspořádaná dvojice (Ω, F) se pak nazývá měřitelný prostor.*

▶ **Definice 2.16** (Náhodný jev [11]). *Nechť (Ω, F) je měřitelný prostor. Prvky $A \in F$ se nazývají náhodné jevy.*

▶ **Definice 2.17** (Pravděpodobností míra [11]). *Nechť (Ω, F) je měřitelný prostor a P reálná funkce $P : F \to \mathbb{R}$ splňující následující podmínky:*

- $\forall A \in F : P(A) \geq 0$;
- $P(\Omega) = 1$;
- *pro každou posloupnost $A_1, A_2, \ldots \in F$ vzájemně disjunktních jevů platí*

$$P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i).$$

Pak P se nazývá pravděpodobností mírou na (Ω, F).

▶ **Definice 2.18** (Pravděpodobnostní prostor [11]). *Nechť (Ω, F) je měřitelný prostor a P pravděpodobnostní míra na (Ω, F). Uspořádaná trojici (Ω, F, P) se potom nazývá pravděpodobnostní prostor nebo také experiment.*

▶ **Definice 2.19** (Nezávislé jevy [11]). *Nechť (Ω, F, P) je pravděpodobnostní prostor a A, B jsou náhodné jevy. Jevy A a B se nazývají nezávislé, pokud platí*

$$P(A \cap B) = P(A) \cdot P(B)$$

S tímto aparátem je již nyní možné přistoupit k definici Shannonovy informace.

▶ **Definice 2.20** (Informace [2]). *Nechť (Ω, F, P) je pravděpodobnostní prostor, $A \in F$ je náhodný jev a $P(A)$ pravděpodobnost tohoto jevu. Pak množství informace spojené s tímto jevem se značí $i(A)$ a pro každé reálné číslo $b \in \mathbb{R}$ splňující $b > 1$ platí*

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A)$$

Jednotky informace závisí na zvoleném základu logaritmu v předchozí definici, pro některé se používají speciální názvy [2]:

- $b = 2$ představuje jednotku *bit*;
- $b = 10$ představuje jednotku *Hartley*;
- $b = e$, kde e značí Eulerovo číslo, představuje jednotku *nat*.

Vzhledem k povaze tohoto textu budou jako výchozí jednotky informace implicitně považovány bity.

Definice 2.20 odpovídá i přirozené intuici. Pokud je pravděpodobnost jevu A nízká, tak je s ním spojeno vysoké množství informace. Naopak pokud je pravděpodobnost takového jevu vysoká, je množství informace spojené s tímto jevem nízké. Další vlastnost této definice, která dává přirozený smysl, je, že informace získaná z výskytu dvou nezávislých jevů je rovna součtu informací získaných z obou jednotlivých jevů, jak ukazuje následující věta. [2]

▶ **Věta 2.21** ([2]). *Nechť (Ω, F, P) je pravděpodobnostní prostor, $A, B \in F$ jsou nezávislé náhodné jevy a $P(A), P(B)$ jsou pravděpodobnosti těchto jevů. Potom platí*

$$i(AB) = i(A) + i(B)$$

Důkaz. Z definice informace platí

$$i(AB) = \log_b \frac{1}{P(AB)}$$

Nyní stačí využít předpokladu o nezávislosti jevů A a B a ihned dostáváme

$$i(AB) = \log_b \frac{1}{P(AB)} = \log_b \frac{1}{P(A) \cdot P(B)} = \log_b \frac{1}{P(A)} + \log_b \frac{1}{P(B)} = i(A) + i(B)$$

◀

▶ **Definice 2.22** (Entropie [2])**.** *Nechť $E = (\Omega, F, P)$ je pravděpodobnostní prostor a posloupnost $A_1, A_2, \dots A_n$ je posloupnost vzájemně nezávislých jevů, které pokrývají výběrový prostor Ω, tj. $\bigcup_{i=1}^{n} A_i = \Omega$. Pak průměrné množství informace spojené s tímto experimentem E se nazývá entropie, značí se $H(E)$ a platí*

$$H(E) = \sum_{i=1}^{n} P(A_i) \cdot i(A_i) = -\sum_{i=1}^{n} P(A_i) \cdot \log_b P(A_i)$$

Předcházející definice není uměle zkonstruovaná, jde o přirozené odvození z požadovaných vlastností na měření průměrného množství informace. Ačkoliv odvození definice entropie není obzvlášť náročné, tak je technické a poměrně zdlouhavé, a proto bude v tomto textu vynecháno. Zájemce však toto odvození nalezne v [2] nebo [12].

Předchozí definice odpovídá takovému experimentu, pro který platí, že pokud experiment E k-krát opakujeme, jsou výsledky opakování experimentu E vzájemně nezávislé a mají stejné pravděpodobnostní rozdělení (z angl. *iid – indepentent and identically distributed*). Pokud toto není splněno, pak pro obecný experiment $E = (\Omega, F, P)$, který při opakování generuje výsledky $\{X_1, X_2, \dots\}$ a $\Omega = \{1, 2, \dots n\}$ platí, že entropie je dána vztahem

$$H(E) = \lim_{k \to \infty} \frac{1}{k} G_k$$

kde

$$G_k = -\sum_{i_1=1}^{i_1=n} \sum_{i_2=1}^{i_2=n} \dots \sum_{i_k=1}^{i_k=n} P(X_1 = i_1, X_2 = i_2, \dots X_k = i_k) \log_b P(X_1 = i_1, X_2 = i_2, \dots X_k = i_k)$$

a $\{X_1, X_2, \dots X_k\}$ je sekvence k výsledků při opakování experimentu. [2] Pokud každý výsledek experimentu E v sekvenci $\{X_1, X_2, \dots X_k\}$ je na ostatních výsledcích vzájemně nezávislý a všechny mají stejné pravděpodobnostní rozdělení (*iid*), pak lze ukázat, že

$$G_k = -k \sum_{i_1=1}^{i_1=n} P(X_1 = i_1) \log_b P(X_1 = i_1)$$

a vztah pro entropii takového experimentu je rovný

$$H(E) = -\sum_{i_1=1}^{i_1=n} P(X_1 = i_1) \log_b P(X_1 = i_1)$$

Tento vztah pak odpovídá vztahu z definice 2.22. [2]

Shannon ukázal, že pokud je experimentem zdroj, který vysílá symboly a_i abecedy A, pak pro $b = 2$ entropie odpovídá průměrnému počtu bitů nezbytných k zakódování výstupu tohoto zdroje, tj. výsledku experimentu E. Shannon dále ukázal, že bezztrátový kompresní algoritmus (více viz. 3.1) nemůže při komprimaci výstupu zdroje dosáhnout lepšího výsledku, než je zakódování tohoto výstupu zdroje pomocí průměrného počtu bitů rovnému entropii zdroje. Z toho plyne, že nejlepší možný bezztrátový kompresní algoritmus dokáže řetězec o n symbolech zkomprimovat na $H(E) \cdot n$ bitů, lépe to není bez ztráty informace možné. [1, 2]

▶ **Příklad 2.23.** Nechť experimentem E je zdroj vysílající symboly abecedy $\Omega = \{a, b, c\}$ s pravděpodobnostmi $P(a) = \frac{1}{6}, P(b) = \frac{1}{2}, P(c) = \frac{1}{3}$. Pak entropie v bitech takového zdroje je podle definice

$$H(E) = -(\frac{1}{6}\log_2\frac{1}{6} + \frac{1}{2}\log_2\frac{1}{2} + \frac{1}{3}\log_2\frac{1}{3}) \approx 1.46$$

Tento výsledek znamená, že při nejlepší možné bezztrátové kompresi bude potřeba na zakódování výstupního symbolu z tohoto zdroje alespoň 1.46 bitů, v praxi tedy alespoň 2 bity. Pro předem neznámý libovolný řetězec délky $|x|_\Omega = 4$ to pak znamená, že nejlepší možný bezztrátový kompresní algoritmus bude potřebovat na zakódování řetězce x ze zdroje E minimálně $H(E) \cdot |x|_\Omega \approx 1.46 \cdot 4 = 5.84$ bitů, v praxi tedy alespoň 6 bitů.

Lze ukázat, že entropie je maximální pro takový experiment, jehož výsledky mají všechny stejnou pravděpodobnost. Pokud možných výsledků experimentu je n a všechny jevy mají stejnou pravděpodobnost $\frac{1}{n}$, pak takový experiment dosahuje maximální entropie $\log_2 n$. [1]

▶ **Příklad 2.24.** Nechť experimentem E je zdroj vysílající symboly abecedy $\Omega = \{0, 1, 2, 3\}$ s pravděpodobnostmi $P(0) = P(1) = P(2) = P(3) = \frac{1}{4}$. Pak entropie v bitech takového zdroje je podle definice

$$H(E) = -(\frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4}) = \log_2 4 = 2$$

Tento výsledek znamená, že při nejlepší možné bezztrátové kompresi budou potřeba na zakódování výstupního symbolu z tohoto zdroje alespoň 2 bity. Pro předem neznámý arbitrární řetězec x délky $|x|_\Omega = 5$ to pak znamená, že nejlepší možný bezztrátový kompresní algoritmus bude potřebovat na zakódování řetězce x ze zdroje E minimálně $H(E) \cdot |x|_\Omega = 2 \cdot 5 = 10$ bitů.

V praxi však často nejsou pravděpodobnosti jevů známy předem, je tedy nemožné určit entropii zdroje a je potřeba entropii aproximovat. Pravděpodobnosti jednotlivých jevů se odhadují na základě pozorovaných výsledků experimentu E, ten se pak modeluje na základě těchto pozorování a získá se tím informace o nejlepším možném kódování konkrétní posloupnosti výsledků, v kontextu této práce tedy informace o nejlepším možném kódování konkrétního řetězce, avšak pouze v rámci zvoleného modelování. Odhad entropie se totiž odráží od předpokladů o struktuře sekvence výsledků zdroje. [2]

▶ **Příklad 2.25** ([2])**.** Nechť výsledkem pozorování je následující řetězec:

$$x = 12123333123333123312$$

Pokud se jako abeceda zvolí jednotlivé symboly, pak pravděpodobnosti těchto symbolů lze odhadnout jako $P(1) = P(2) = \frac{1}{4}$ a $P(3) = \frac{1}{2}$. Entropii zdroje těchto dat pak lze odhadnout jako

$$H(E) = -(\frac{1}{4}\log_2\frac{1}{4} + \frac{1}{4}\log_2\frac{1}{4} + \frac{1}{2}\log_2\frac{1}{2}) = 1.5$$

Entropie tohoto zdroje tedy je 1.5 bitů na jeden symbol. Pro tento konkrétní řetězec x to navíc znamená, že nejlepší možný bezztrátový kompresní algoritmus bude schopný řetězec x zkomprimovat na $H(E) \cdot |x|_\Omega = 1.5 \cdot 20 = 30$ bitů, tj. 30 bitů je vyžadované minimum pro bezztrátovou reprezentaci řetězce x.

Lze však pro experiment zvolit jiné modelování, např. lze řetězec x rozdělit na symboly o dvou cifrách. Pak má taková abeceda pouze dva symboly, 12 a 33. Pravděpodobnosti pak lze odhadnout jako $P(1\,2) = \frac{1}{2}$ a $P(3\,3) = \frac{1}{2}$, odhad entropie potom je

$$H(E) = -(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}) = \log_2 2 = 1$$

Entropie tohoto zdroje tedy je 1 bit na jeden symbol. Podle nového modelu je v řetězci x pak 10 těchto symbolů a pro řetězec x to tedy znamená, že nejlepší možný bezztrátový kompresní algoritmus bude schopný řetězec x zkomprimovat na $H(E) \cdot |x|_\Omega = 1 \cdot 10 = 10$ bitů, tj. 10 bitů je vyžadované minimum pro bezztrátovou reprezentaci řetězce x.

Proces z předchozího příkladu se nazývá *modelování* a je součástí každého kompresního procesu. Před provedením komprese je potřeba, aby byly o zdroji učiněny určité předpoklady, jako je např. určení symbolů zdrojové abecedy nebo rozdělení pravděpodobnosti jednotlivých symbolů abecedy. Vytvořený model potom reprezentuje získané znalosti o zdroji a umožňuje manipulaci s redundancí zdrojových dat. [3]

Pozorování, že maximální entropie dosahuje takový experiment, jehož výsledky mají všechny stejnou pravděpodobnost, lze pak využít k definici redundance.

▶ **Definice 2.26** (Redundance [1]). *Nechť $E = (\Omega, F, P)$ je pravděpodobnostní prostor a posloupnost $A_1, A_2, \dots A_n$ je posloupnost vzájemně nezávislých jevů, pro které platí $\bigcup_{i=1}^{n} A_i = \Omega$. Pak redundance R je rovna rozdílu maximální možné entropie experimentu a jeho skutečné entropie, tj.*

$$R = \left[-\sum_{i=1}^{n} P \cdot \log_b P \right] - \left[-\sum_{i=1}^{n} P(A_i) \cdot \log_b P(A_i) \right] = \log_2 n + \sum_{i=1}^{n} P(A_i) \cdot \log_b P(A_i)$$

Z této definice plyne, že pro dokonale komprimovaná data platí, že jejich redundance je nulová, tj. $R = \log_2 n + \sum_{i=1}^{n} P(A_i) \cdot \log_b P(A_i) = 0$. [1]

Pro úplnost bude na závěr této kapitoly ještě uvedena definice komprese, dekomprese, kompresoru a dekompresoru.

▶ **Definice 2.27** (Komprese a dekomprese). *Komprese je proces, který usiluje o konstrukci takového kódu $K = (A, S, C, \kappa, \delta)$, pro který bude možné vstupní řetězec $x \in A^+$ délky $|x|_A = n$ zakódovat do takového řetězce $y \in C^+$, jehož redundance je nejnižší možná. Proces, který tento řetězec dekóduje zpět, se nazývá dekomprese.*

▶ **Definice 2.28** (Kompresor a dekompresor [1]). *Program, který komprimuje data na svém vstupu a vytváří výstupní zkomprimovaná data s nízkou úrovní redundance, se nazývá kompresor nebo také kodér. Program, který data konvertuje v opačném směru, se nazývá dekompresor nebo také dekodér.*

2.2 Měření kvality komprese

Není jednoduché změřit kvalitu výstupu kompresního algoritmu obecně, protože jeho výstup značně závisí na tom, zda vstupní data obsahují konkrétní druh redundance, který se daný algoritmus snaží najít. [3] Je proto důležité se uvědomit, že kompresní algoritmus či formát nemusí být vhodný na všechny možné druhy dat, např. použití Huffmanova kódování na obrazová data nepřinese tak významnou úroveň úspory objemu, jakou by přineslo použití ztrátového kompresního algoritmu pro obrazová data, dále např. kompresní algoritmy, které jsou určeny pro práci s bloky textu, nemusí poskytnout dobrou kompresi při použití na číselných datech a naopak. [13]

Na kompresní algoritmus jsou také kladeny různé požadavky v závislosti na povaze aplikace. Kromě úspory místa určují kvalitu a vhodnost použití daného kompresního algoritmu i další faktory. [3] Příkladem může být situace, kdy jsou data komprimována mimo zařízení koncového uživatele, poté jsou distribuována ke klientovi, kde jsou nakonec dekomprimována. Tento scénář je nejběžnější pro distribuování aplikací, jejichž data jsou dekomprimována při instalaci aplikace na cílovém zařízení, nebo pro tvorbu a sdílení děl umělců, jako jsou videa, hudba či fotografie a obrázky, které se vytvářejí pomocí nástrojů podporující vysoké rozlišení a poté jsou exportována a komprimována pro distribuci. Cílem komprese v těchto situacích jsou soubory o co nejmenší možné velikosti, kompromisem je pak příliš dlouhá doba dekomprese v případě aplikací a kvalita a věrnost vůči originálu v případě audiovizuálních dat. [13]

Dalším příkladem je komprese na straně klienta a dekomprese na straně serveru. Většina moderních aplikací pro sociální sítě generuje velké množství obsahu na straně klienta a pak jej odesílají na server ke zpracování a distribuci dalším klientům dané služby. Většina takových

klientů jsou mobilní aplikace, proto je na místě snaha o dobrou kompresi pro snížení odchozí komunikace a šetření nákladů na datový tarif uživatele. Příkladem může být serializace odchozího paketu do binárního formátu a následná komprese před odesláním na server. Klient se také může se serverem pomocí HTTP hlavičky *Accept-Encoding* dohodnout, aby celý obsah dat v HTTP odpovědi zaslané serverem byl komprimován ve formátu *gzip*, *compress* nebo *deflate*. V praxi se však v těchto případech používá komprese pouze mírná kvůli snížení režie odchozí komunikace, zachování požadované odezvy aplikace a šetření baterie v zařízení uživatele. [13, 14]

Opačným scénářem je odesílání dat ze serveru ke klientovi, kdy komprese probíhá na straně serveru a dekomprese na straně klienta. Klient může například čekat na výsledek nějaké databázové operace, popř. server odesílá dynamická data a klient čeká na vygenerování obsahu. Doba, kterou server potřebuje k vygenerování a komprimaci těchto dat, je kriticky důležitá, aby klient nečekal na síťovou odpověď příliš dlouho. Zásadní je v tomto případě najít přijatelný kompromis mezi velikostí dat a dobou komprese. Speciálním případem je nakládání s vysokými objemy audiovizuálních dat, příkladem mohou být gigabajty obrázků formátu PNG, které je potřeba převést do formátu WebP v několika různých rozlišeních, nebo tisíce hodin videa, které je zapotřebí převést do formátu H.264 předtím, než budou přenášeny do koncového zařízení. Využití výpočetních zdrojů serveru je v tomto případě ideálním řešením, protože je potřeba zkomprimovat velké objemy dat co nejefektivnějším způsobem, od toho se však také odvíjí požadavek na efektivní využití výpočetních prostředků kompresním algoritmem kvůli finanční úspoře zákazníka výpočetních služeb serveru. [13]

Existuje také mnoho klientských aplikací, které potřebují komunikovat mezi sebou navzájem. Mohou např. posílat pakety přes P2P síť, fotografie nebo GPS data. V těchto případech vygeneruje klient data, která zkomprimuje a poté odešle dalšímu klientovi k dekompresi. Klientské zařízení je často mobilní zařízení, které obvykle nedisponuje významnými výpočetními prostředky, které jsou potřebné k optimální kompresi dat. Pro obecná data to proto znamená, že úspora místa dosažená kompresí může být nižší, kvůli menšímu množství času nutnému na optimalizaci, a stejně i doba dekomprese může být pomalejší, kvůli menšímu výkonu klientských zařízení. V těchto případech je potřeba najít vyvážení mezi výpočetními schopnostmi zařízení, časem potřebným ke kompresi a dekompresi a dobou, do které musejí být data odeslána. Na druhou stranu však tato mobilní zařízení často disponují speciálním grafickým hardwarem, který lze využít k efektivní kompresi audiovizuálních dat ve formátech JPG nebo H.264. [13]

Kvalita kompresního algoritmu se proto silně odvíjí od konkrétních požadavků daného použití. Příkladem může být archivační program ZPAQ, který obvykle dosahuje nejlepší komprese pro 1 GB textových dat. Nicméně pro účely komprese však potřebuje alespoň 2 GB paměti a doba komprese 1 GB textových dat dosahuje na běžném počítači až 3 hodin a podobných hodnot dosahuje i při dekompresi. Pokud je hlavním požadavkem co nejvyšší úspora místa při kompresi textových dat, klientská zařízení disponují dostatečným výpočetním výkonem a doba komprese i dekomprese nehraje roli, pak je tento program ideální volbou, pro mobilní zařízení však není příliš použitelný. [13]

Ačkoliv je z předchozích příkladů užití zřejmé, že posouzení kvality kompresního algoritmu je závislé na problémové doméně, vždy mezi klíčová kritéria patří doba běhu komprese a dekomprese společně se zmenšením objemu dat, tj. jak efektivně dokáže algoritmus kompresí ušetřit místo v paměti. Výstup kompresního algoritmu také značně závisí na požadavku, zda mají být zrekonstruovaná data identická se zdrojovými, metriky se proto liší podle toho, zda je o bezztrátovou či ztrátovou kompresi.

2.2.1 Měření kvality bezztrátové komprese

Přirozeným a logickým způsobem jak změřit, jak dobře kompresní algoritmus komprimuje daná data na svém vstupu, je poměr počtu bajtů potřebných k reprezentaci dat po kompresi k počtu bajtů potřebných k reprezentaci dat předtím, než se data zkomprimují. Tento poměr se nazývá *kompresní poměr*. [1]

$$\textit{kompresní poměr} = \frac{\textit{velikost výstupních dat}}{\textit{velikost vstupních dat}}$$

Hodnota 0.6 znamená, že data po kompresi zabírají 60 % své původní velikosti. Hodnoty větší než 1 znamenají, že výstupní data kompresního algoritmu zabírají více místa než vstupní data, tento jev se nazývá *záporná komprese*. [1]

Převrácená hodnota kompresního poměru se nazývá *kompresní faktor*. [1]

$$\textit{kompresní faktor} = \frac{\textit{velikost vstupních dat}}{\textit{velikost výstupních dat}}$$

V tomto případě hodnoty větší než 1 znamenají kompresi a hodnoty menší než 1 naopak expanzi. Tato metrika se může zdát přirozenější, protože platí, že čím vyšší je kompresní faktor, tím lepší je komprese. [1]

Je potřeba poznamenat, že definice kompresního poměru a kompresního faktoru se v odborné literatuře občas zaměňuje, příkladem může být porovnání z [1, 3] vůči [2].

Poslední nejčastěji používanou metrikou je *kompresní úspora*:

$$\textit{kompresní úspora} = \frac{\textit{velikost vstupních dat} - \textit{velikost výstupních dat}}{\textit{velikost vstupních dat}} \cdot 100$$

V tomto případě hodnota 60 znamená, že komprese přinesla úsporu 60 % nebo ekvivalentně že výstup zabírá 40 % své původní velikosti. [1]

▶ **Příklad 2.29** ([2, 3])**.** Zdrojový černobílý obrázek s rozlišením 256×256 pixelů zabírá v paměti 65 536 bajtů. Po bezztrátové kompresi tento obrázek zabírá 16 384 bajtů. Pak platí, že kompresní poměr je 16384/65536 = 1/4, kompresní faktor je 4 a kompresní úspora je

$$\frac{65536 - 16384}{65536} \cdot 100 = 75\ \%$$

2.2.2 Měření kvality ztrátové komprese

Metriky měření kvality bezztrátové komprese se používají i pro kompresi ztrátovou, nicméně kromě nich je ještě potřeba změřit kvalitu dekomprimovaných dat. Při ztrátové kompresi se dekomprimovaná data liší od původních, je proto žádoucí tento rozdíl vhodným způsobem kvantifikovat. Rozdíl mezi originálními a rekonstruovanými daty se nazývá *zkreslení* (z angl. *distortion*). Ztrátové techniky se obvykle používají pro kompresi audiovizuálních dat, která vznikají jako analogová data, jako je řeč, video nebo obrazová data. Při kompresi těchto dat je konečným arbitrem kvality člověk, nicméně lidské reakce je obtížné matematicky modelovat, pro určení kvality zrekonstruovaných dat se proto používá několik měr zkreslení. [2, 3]

Dvě často používané míry zkreslení nebo-li rozdílu originální a zrekonstruované posloupnosti dat jsou čtvercová chyba a absolutní rozdíl. Tyto metriky se nazývají *metriky rozdílového zkreslení* (z angl. *difference distortion measures*). Pokud je $(x_i)_{i=1}^n$ posloupnost originálních dat a $(y_i)_{i=1}^n$ je posloupnost dat po kompresi, pak čtvercová chyba i-tého prvku [2] je dána vztahem

$$d(x_i, y_i) = (x_i - y_i)^2.$$

Absolutní rozdíl i-tého prvku [2] je pak daný vztahem

$$d(x_i, y_i) = |x_i - y_i|.$$

V obecných případech je nepraktické i obtížné zkoumat zkreslení po jednotlivých prvcích, proto se k shrnutí informací z posloupnosti rozdílového zkreslení používají průměrné metriky.

Nejčastěji používanou průměrnou metrikou je MSE (z ang. *mean squared error*) a označuje se σ^2 [2]:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2$$

Další často používanou metrikou je průměr z absolutních rozdílů, který je obzvlášť užitečný při měření kvality komprese obrázků [2]:

$$d_1 = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|$$

V některých případech není zkreslení patrné, pokud je pod hranicí určité prahové hodnoty. V těchto situacích se pak používá maximální hodnota chyby [2]:

$$d_\infty = \max_i |x_i - y_i|.$$

Pokud je potřeba porovnat průměrnou chybu vzhledem k původním datům, lze použít poměr průměrné čtvercové hodnoty vstupu a MSE. Tento poměr se nazývá *poměr signálu a šumu* (z angl. *signal-to-noise ratio*) a značí se SNR.

$$\mathrm{SNR} = \frac{\sigma_x^2}{\sigma_d^2}$$

σ_x^2 značí průměrnou čtvercovou hodnotu původních dat a σ_d^2 značí MSE. Hodnota SNR se často měří na logaritmické stupnici, jednotky jsou potom decibely. [2]

$$\mathrm{SNR(dB)} = 10\log_{10}\frac{\sigma_x^2}{\sigma_d^2}$$

Místo měření velikosti chyby vzhledem k průměrné čtvercové hodnotě vstupu je někdy užitečnější změřit velikost chyby vzhledem k maximální hodnotě vstupních dat. Tento poměr se nazývá *špičkový poměr signálu k šumu* (z angl. *peak signal-to-noise ratio*) a značí se PSNR. [2]

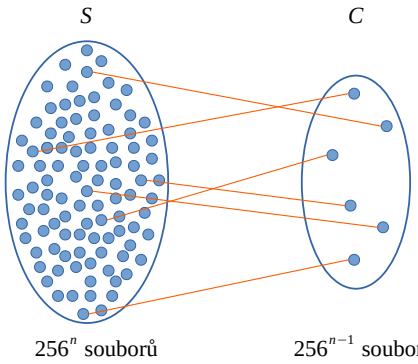
$$\mathrm{PSNR(dB)} = 10\log_{10}\frac{x_{peak}^2}{\sigma_d^2}$$

Další pojem, který se při posuzování rozdílu originálních a rekonstruovaných dat používá, je *věrnost originálu* (z ang. *fidelity*). O rekonstruovaných datech se říká, že mají vysokou věrnost vůči originálu, pokud je rozdíl mezi originálními a rekonstruovanými daty malý. Z kontextu by vždy mělo být zřejmé, zda se jedná o rozdíl kvantifikovaný matematicky, nebo o rozdíl percepční, tj. vnímaný člověkem. [2]

2.3 Limity bezztrátové komprese

Tato práce se zabývá bezztrátovými kompresními algoritmy, nabízí se proto otázka, čeho je možné pomocí bezztrátové komprese dosáhnout. Např. jestli je možné najít univerzální kompresní algoritmus, který by dokázal pro jakýkoliv libovolný vstup vždy vygenerovat výstup menší velikosti tak, aby data po dekompresi byla identická s původními. Pokud by to nebylo možné, pak by bylo vhodné vědět, jak velký podíl souborů je možné bezztrátově zkomprimovat. Další častou otázkou je, zda je možné již jednou zkomprimovaný soubor komprimovat opakovaně dál tak dlouho, dokud není výsledek komprese dostatečně uspokojující. Na tyto otázky odpovídají následující tvrzení. [3]

▶ **Tvrzení 2.30** ([1]). *Neexistuje algoritmus, který by dokázal bezztrátově zkomprimovat jakýkoliv soubor alespoň o polovinu jeho původní velikosti.*



Obrázek 2.3 Mapování souborů na jejich komprimované reprezentace [3]

Důkaz. Důkaz bude vedený sporem, tj. bude se předpokládat, že existuje algoritmus, který je schopný bezztrátově zkomprimovat jakýkoliv soubor alespoň o polovinu jeho velikosti. Nechť vstupní soubor, který se má zkomprimovat, má velikost n bitů. Existuje celkem 2^n různých n-bitových souborů. Je zřejmé, že pro dva různé soubory na vstupu musí bezztrátový kompresní algoritmus vygenerovat dva různé výstupy, jinak by neexistoval způsob, jak je dekomprimovat a získat zpět původní soubory. Proto pro 2^n různých n-bitových vstupních souborů musí algoritmus vygenerovat 2^n různých výstupních souborů velikosti $n/2$ nebo nižší. Celkový počet souborů s takovou velikostí je:

$$N = 1 + 2 + 4 + 8 + \ldots + 2^{n/2} = 1 \cdot \frac{2^{1+n/2} - 1}{2 - 1} = 2^{1+n/2} - 1 \approx 2^{1+n/2}$$

Je zřejmé, že N je mnohem menší než 2^n, což je spor s předpokladem, že pro m různých vstupních souborů musí bezztrátový kompresní algoritmus vygenerovat m různých výstupních souborů. Původní tvrzení je tedy pravdivé. [1] ◀

▶ **Tvrzení 2.31** ([3]). *Neexistuje algoritmus, který by dokázal bezztrátově zkomprimovat jakýkoliv soubor alespoň o 1 bajt.*

Důkaz. Důkaz bude opět vedený sporem, tj. bude se předpokládat, že existuje takový kompresní algoritmus, který je schopný zkomprimovat jakýkoliv soubor alespoň o 1 bajt. Pak by bylo možné takový algoritmus použít k opakované kompresi daného souboru. Nechť *big.file* je velký vstupní soubor a *compress* je bezztrátový kompresní program, který dokáže bezztrátově zkomprimovat jakýkoliv soubor alespoň o 1 bajt. To znamená, že musí být možné zkomprimovat data, která už jednou tímto programem zkomprimována byla. Z toho plyne, že když bude dostávat program *compress* na svůj vstup data, která v předchozím kroku zkomprimoval, bude i nadále kompresní poměr menší než 1. To znamená, že po dostatečném počtu opakování bude mít výstupní soubor *compress(compress(compress(…compress(big.file) …)))* nulovou velikost. Soubor s nulovou velikostí zřejmě nikdy nepůjde dekomprimovat na původní data, což je spor. [3] ◀

▶ **Tvrzení 2.32** ([3]). *Neexistuje algoritmus, který by dokázal bezztrátově zkomprimovat alespoň 1 % všech možných souborů alespoň o 1 bajt.*

Důkaz. K důkazu tohoto tvrzení je potřeba najít podíl souborů, které je možné zkomprimovat o 1 bajt, a ukázat, že těchto souborů je méně než 1 %. Na kompresi zdrojového souboru lze nahlížet jako na mapování vstupního souboru na jeho zkomprimovanou reprezentaci. Soubor o délce n bajtů zkomprimovaný o 1 bajt je proto ekvivalentní mapování souboru o délce n bajtů na soubor o velikosti $n - 1$ bajtů.

Na obrázku 2.3 představuje S množinu všech souborů velikosti n bajtů a C představuje množinu všech souborů velikosti $n - 1$ bajtů. Každá tečka v množině reprezentuje soubor o dané

velikosti, každá úsečka pak reprezentuje mapování souboru velikosti n bajtů na soubor velikosti $n-1$ bajtů. Celkem existuje $(2^8)^n = 256^n$ různých souborů velikosti n bajtů a 256^{n-1} různých souborů velikosti $n-1$ bajtů. V nejlepším možném případě by bylo možné každý z 256^n souborů komprimovat o 1 bajt. Počet mapování je však roven počtu zkomprimovaných souborů, kterých je nejvýše 256^{n-1}. Z toho plyne, že podíl souborů zkomprimovaných o 1 bajt je

$$\frac{256^{n-1}}{256^n} = \frac{1}{256}$$

Protože $1/256 < 1/100 = 1\ \%$, tak je zřejmé, že takto zkomprimovaných souborů je méně než 1 %. To znamená, že existuje méně než 1 % všech možných souborů, které lze bezztrátově zkomprimovat o 1 bajt, tedy neexistuje bezztrátový kompresní algoritmus, který by dokázal zkomprimovat alespoň 1 % všech možných souborů alespoň o 1 bajt. [3] ◀

Z předchozích tvrzení a uvedené teorie komprese je tedy evidentní, že aby bylo možné bezztrátově zkomprimovat určitá data, musí kompresní algoritmus data prozkoumat, najít v nich redundanci a tu odstranit. Redundance ve vstupních datech však silně závisí na tom, o jaký druh dat se jedná (text, video, zvuk, obrázky, atd.), proto je žádoucí, aby pro určitý druh dat byly vyvinuty specifické kompresní metody, které budou na těchto datech poskytovat nejlepší výsledky. Jak je však vidět z předchozích tvrzení, tak skutečně univerzální a efektivní bezztrátový kompresní algoritmus neexistuje a nikdy existovat nebude. [1]

Kapitola 3

Charakterizace kompresních metod

Tato kapitola se zabývá popisem a vysvětlením vlastností, které kompresní algoritmus charakterizují.

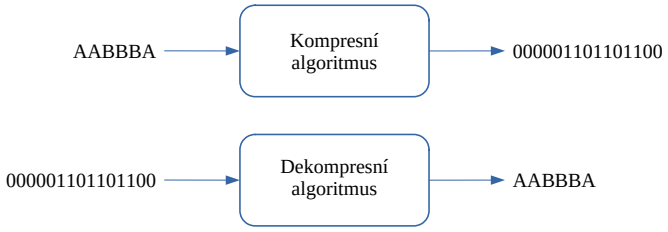
Ačkoliv se v literatuře algoritmy jako LZ77, Huffmanovo kódování nebo ACB zkráceně označují jako kompresní algoritmy, vždy se jedná o algoritmy dva. Jedním je kompresní algoritmus (implementovaný v programu nebo jeho části nazývané kompresor), který na svém vstupu přijímá vstupní data χ a vytváří jejich reprezentaci χ_c. Druhým je dekompresní algoritmus (implementovaný v programu nebo jeho části nazývané dekompresor), který na svém vstupu přijímá komprimovanou reprezentaci χ_c a generuje rekonstruovaná data γ. [2]

I tento text se bude řídit zavedenou konvencí a označovat oba algoritmy, kompresní i dekompresní, souhrnně jako kompresní algoritmus. Kompresní algoritmy lze charakterizovat na základě požadavků na rekonstrukci dat, modifikaci vnitřních struktur a parametrů v závislosti na aktuálně zpracovávaných datech, odlišné složitosti komprese a dekomprese, počtu zpracování vstupních dat a způsobu modelování dat.

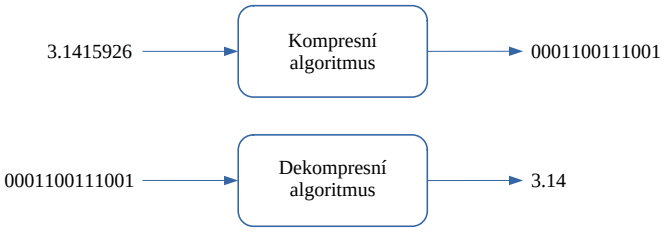
3.1 Ztrátové a bezztrátové metody

Bezztrátové kompresní metody při kompresi nezahrnují žádnou ztrátu informace. Původní data lze z komprimovaných dat přesně zrekonstruovat, tj. dekomprimovaná data γ a původní data χ jsou identická. Ukázku bezztrátové komprese ilustruje obrázek 3.1.

Bezztrátové metody se obvykle používají tam, kde je nežádoucí a nepřípustná ztráta jakékoliv informace. Například soubory obsahující počítačové aplikace se mohou stát bezcennými, pokud



Obrázek 3.1 Ilustrace bezztrátové komprese [3]



Obrázek 3.2 Ilustrace ztrátové komprese [3]

dojde ke změně jediného bitu. Další důležitou oblastí bezztrátové komprese je komprese textu, u které je velmi důležité, aby rekonstruovaná data byla identická s původními, protože i malé změny mohou mít za následek text s odlišným významem, a podobné je to např. i u archivace bankovních záznamů. Bezztrátová komprese však může být vyžadována i u obrazových dat, jako jsou například radiologické snímky, fotografie pořizované a archivované pro právní účely, nebo snímky pořízené z oběžných satelitů. [1, 2, 3]

Mezi bezztrátové kompresní algoritmy patří algoritmy pro kompresi obecných dat, jako je například aritmetické a Huffmanovo kódování, algoritmy rodiny LZ jako LZ77, LZ78, LZMA či LZSS, nebo algoritmus DEFLATE. Dalším příkladem může být kontextová metoda PPM, která je optimalizována speciálně pro zpracování textových dat, algoritmus Adam7 používaný ve formátu PNG pro bezztrátovou kompresi obrázků, nebo kontextově-adaptivní binární aritmetické kódování (z angl. *context-adaptive binary arithmetic coding*) pro bezztrátovou kompresi videa ve formátu H.264. [1, 2]

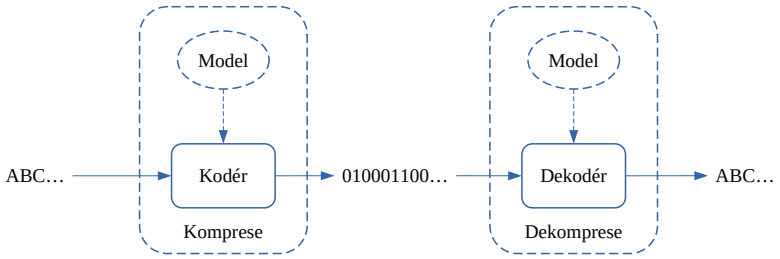
Existují však i situace, kdy je určitá ztráta informace akceptovatelná, obzvlášť pokud tyto ztráty povedou k výrazně lepšímu kompresnímu poměru. Kompresní metoda je *ztrátová*, pokud je nemožné z komprimovaných dat χ_c zrekonstruovat přesná původní data χ. Výměnou za akceptování takového zkreslení jsou mnohem vyšší kompresní faktory, než jakých by bylo možné dosáhnout na stejných datech s bezztrátovými kompresními metodami. Měření kvality zrekonstruovaných dat popisuje kapitola 2.2.2 a ukázku ztrátové komprese ilustruje obrázek 3.2. Použití těchto metod dává smysl zejména při kompresi obrázků, videa či zvuků, protože v některých situacích tato ztráta nepředstavuje problém, případně člověk ani nemusí rozdíl poznat, pokud je ztráta informace dostatečně malá. [1, 2, 3]

Například při ukládání či přenosu řeči není přesná hodnota každého vzorku řeči nutná. V závislosti na požadované kvalitě zrekonstruované řeči lze tolerovat různě velkou ztrátu informace každého vzorku. Má-li být kvalita zrekonstruované řeči podobná kvalitě řeči z telefonního rozhovoru, lze tolerovat velmi značnou ztrátu informace. Má-li být kvalita zrekonstruované řeči podobná řeči z kompaktního audio disku, pak je tolerovaná ztráta informace naopak velice nízká. Podobně při komprimaci videa není obecně důležité, že se zrekonstruovaná data liší od originálu, pokud tyto rozdíly nevedou k výskytu artefaktů. [2]

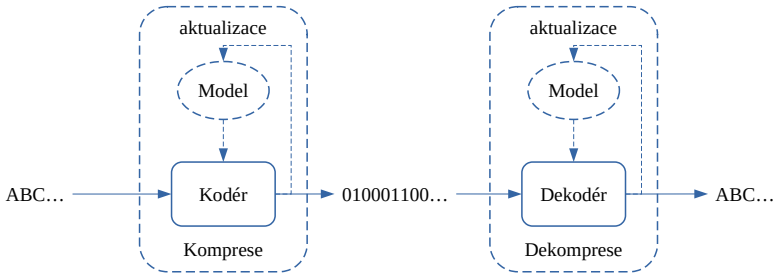
Mezi ztrátové kompresní metody patří např. diskrétní kosinová transformace (z angl. *discrete cosine transform*, zkratka DCT) používaná ve formátech JPEG a WebP pro kompresi obrázků, nebo ve formátech H.264 a H.265 pro kompresi videa. Modifikovaná verze diskrétní kosinové transformace (zkratka MDCT) se pak používá pro kompresi audia ve formátech MP3, AAC nebo WMA. Dalším příkladem může být vlnková komprese (z angl. *wavelet compression*), která se používá např. ve formátech JPEG 2000 a PGF pro kompresi obrázků, nebo ve formátu DjVu pro skenování dokumentů. [1, 2]

3.2 Statické, semi-adaptivní a adaptivní metody

Kompresní metodu lze charakterizovat podle toho, jestli při procesu komprese upravuje či aktualizuje svůj model zdrojových dat. *Statická* (někdy také *neadaptivní*) metoda je rigidní a nemění



Obrázek 3.3 Statická kompresní metoda [3]



Obrázek 3.4 Adaptivní kompresní metoda [3]

své operace, parametry ani vnitřní datové struktury v závislosti na aktuálně zpracovávaných datech. Tyto metody poskytují nejlepší výsledky při kompresi dat, která jsou všechna jednoho stejného druhu. Příkladem mohou být metody GROUP 3 (zkr. G3) a GROUP 4 (zkr. G4), které se používají ve faxových přístrojích při přenosu statického obrazu přes telefonní linku. Obě metody byly speciálně navrženy jako kompresní metody pro faxování a poskytovaly by velmi špatné výsledky při použití na jakýchkoliv jiných datech. [1]

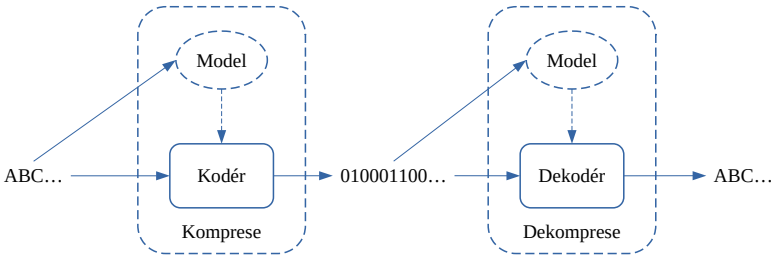
Mezi další statické metody patří např. Morseovo a Braillovo kódování. Schéma statické kompresní metody ilustruje obrázek 3.3.

Opačný přístup využívají *adaptivní* metody, které v závislosti na tom, jaká aktuálně zpracovávají data, aktualizují model zdrojových dat. Své parametry a vnitřní datové struktury tedy adaptivní metody aktualizují za běhu při kompresi či dekompresi dat a některé adaptivní metody dokonce začínají s prázdným modelem.

Mezi adaptivní kompresní metody patří například adaptivní Huffmanovo kódování, kontextová metoda PPMC, nebo slovníkové algoritmy LZ77, LZ78, LZSS a LZW. [1, 3] Schéma adaptivní kompresní metody ukazuje obrázek 3.4.

Některé metody nejprve při prvním průchodu vstupními daty vytvoří model zdroje a teprve v dalším průchodu komprimují samotná data na základě modelu vytvořeném při prvním čtení dat. Takové metody se nazývají *semi-adaptivní*. Příkladem mohou být některé statistické metody, které před zahájením kódování vyžadují počáteční průchod daty pro výpočet jednotlivých pravděpodobností. Tento přístup nemusí být ideálním řešením pro velmi objemná data, protože vždy je potřeba provést počáteční průchod daty pro výpočet pravděpodobností a vypočtená čísla pak zůstávají po celou dobu komprese neměnná. Nicméně pro relativně malá vstupní data ani jedno nepředstavuje značný problém. [1, 13]

Mezi semi-adaptivní metody patří například aritmetické kódování, Shannonovo–Fanovo kódování a statické Huffmanovo kódování. Schéma semi-adaptivní kompresní metody je k vidění na obrázku 3.5.



Obrázek 3.5 Semi-adaptivní kompresní metoda [3]

3.3 Statistické, kontextové a slovníkové metody

Statistické kompresní metody využívají statistické modelování zdroje dat, jmenovitě modelují abecedu a pravděpodobnostní rozdělení zdroje, přičemž se předpokládá, že zdroj generuje posloupnost na sobě vzájemně pravděpodobnostně nezávislých symbolů. Tyto metody v modelovací fázi přiřazují symbolům abecedy pravděpodobnosti, které jsou vypočítané na základě četnosti výskytu symbolu, a v kódovací fázi přiřazují symbolům abecedy číselné kódy na základě vypočtených pravděpodobností. Pravděpodobnostní model lze získat pomocí statického přístupu, kdy jsou všechny pravděpodobnosti předem určené a za běhu algoritmu neměnné, adaptivního (někdy také dynamického) přístupu, kdy jsou pravděpodobnosti modifikovány za běhu algoritmu při čtení a komprimování vstupních dat, nebo semi-adaptivního přístupu, kdy jsou pravděpodobnosti vypočítány při samostatném průchodu vstupními daty a v dalším průchodu při kompresi dat již zůstávají pravděpodobnosti neměnné. [1, 2]

Statistické metody používají kódová slova proměnné délky a dosahují komprese pomocí přiřazování kratších kódových slov těm symbolům, které se v datech vyskytují často (jejich modelovaná pravděpodobnost je vysoká), a přiřazování delších kódových slov symbolům, jejichž četnost výskytu je malá (jejich modelovaná pravděpodobnost je nízká). Dále efektivita komprese závisí na velikosti abecedy a také na tom, jak je pravděpodobnostní rozdělení získané při procesu modelování blízké skutečnému rozdělení zdroje. [1, 3]

Mezi statistické metody patří Morseovo kódování, statické i adaptivní Huffmanovo kódování, aritmetické kódování a Shannonovo–Fanovo kódování.

Dále existuje řada technik, které využívají minimum předběžných předpokladů o pravděpodobnostním rozdělení zdrojových dat. Místo toho využívají kontext zpracovávaných dat a jejich nedávnou historii. Takové metody se nazývají *kontextové* a používají se primárně pro kompresi textu. [2]

Z principu statistických metod plyne, že lepší komprese lze dosáhnout pro takovou zdrojovou zprávu, která má sešikmené (někdy také *zkosené*, z angl. *skewed*) pravděpodobnostní rozdělení, čímž je míněno, že některé symboly se v kódované posloupnosti vyskytují s mnohem vyšší pravděpodobností než jiné. Pak dává přirozený smysl hledat jiné reprezentace zdrojové zprávy, které by vedly k ještě vyššímu sešikmení pravděpodobnostního rozdělení. Jeden z účinných způsobů je určování pravděpodobnosti výskytu symbolu v závislosti na aktuálním kontextu, ve kterém se symbol vyskytuje. Existuje proto řada technik, které využívají minimum předběžných předpokladů o pravděpodobnostním rozdělení zdrojových dat a místo toho využívají kontext zpracovávaných dat a jejich nedávnou historii. Takové metody se nazývají *kontextové* a používají se primárně pro kompresi textu. [1, 2]

Protože kodér i dekodér nemají přístup k dosud nezpracované části vstupu, musí kodér i dekodér omezit kontext na předcházející data, tj. na symboly, které byly již zpracovány. V praxi je kontextem aktuálně zpracovávaného symbolu N symbolů, které mu předcházejí. Kontextové kompresní metody tedy na každý symbol v posloupnosti komprimovaných dat nenahlíží jako na samostatný jev nezávislý na ostatních, ale místo toho se zkoumá historie posloupnosti symbolů

a teprve poté se určí pravděpodobnost zpracovávaného symbolu. Někdy se také říká, že kontextové kompresní metody využívají kontext aktuálně zpracovávaného symbolu k tomu, aby jej tzv. předpověděly, čímž je jednoduše myšleno, že symbolu přiřazují jeho pravděpodobnost. Z matematického pohledu se pak říká, že taková kompresní metoda využívá Markovův statistický model řádu N. [1, 2]

Kontextové metody předpokládají, že se kódovaná posloupnost symbolů neskládá ze vzájemně nezávislých jevů, proto znalost toho, které symboly se vyskytly v okolí právě kódovaného symbolu, pak přináší mnohem lepší aproximaci pravděpodobnosti daného symbolu. Pokud je znám kontext, ve kterém se symbol vyskytuje, pak lze s mnohem vyšší pravděpodobností úspěchu odhadnout hodnotu kódovaného symbolu. Z toho plyne, že vzhledem ke kontextu se některé symboly budou vyskytovat s mnohem vyšší pravděpodobností než jiné, což znamená, že pravděpodobností rozdělení výskytu symbolů je více sešikmené. Pokud se navíc podaří vhodným způsobem seskupit podobné kontexty, je velmi pravděpodobné, že symboly následující po těchto kontextech budou stejné, což umožňuje použít některé jednoduché a přitom účinné kompresní strategie. [2]

Mezi kontextové kompresní metody patří například PPM a jeho varianty jako PPMA, PPMB, PPMC či PPMZ, nebo metoda ACB.

Statistické metody komprese dat používají statistický model dat, a proto kvalita komprese, které tyto metody dosáhnou, značně závisí na tom, jak kvalitní je tento model. *Slovníkové* kompresní metody nepoužívají statistický model dat a rovněž jako kontextové metody nepředpokládají, že by se symboly ve vstupní posloupnosti vyskytovaly nezávisle na ostatních. Slovníkové metody postupně čtou řetězce ze vstupu (někdy označované též jako slova, fráze nebo vzory) a pro přečtený řetězec hledají shodu ve své vnitřní datové struktuře, která se nazývá slovník. V případě nalezení shody je pak na výstup zapsán index (někdy také reference nebo ukazatel) nalezeného řetězce ve slovníku. Slovník jako své záznamy tedy obsahuje řetězce symbolů a může být statický nebo adaptivní (někdy také dynamický). Statický slovník je neměnný po celou dobu komprese, zatímco adaptivní slovník udržuje řetězce, které se objevily dříve ve vstupních datech a umožňuje přidávání i mazání řetězců při čtení další části vstupu, tj. aktuální obsah slovníku se mění podle zpracovávané části vstupní posloupnosti symbolů. [1, 2]

Statický slovník není pro univerzální kompresi vhodným řešením, může však být dobrou volbou pro specifické problémové domény. K použití statického slovníku je potřeba mít poměrně dobrou představu o struktuře vstupních dat. Pokud tyto informace nejsou k dispozici před kódováním konkrétních zdrojových dat, tak je nutné tyto informace nějakým způsobem získat během kódování. Pokud máme dostatek informací před zahájením kódování, je možné použít statický slovník, v opačném případě je potřeba použít adaptivní slovník. [2]

Obecně je vhodnější metoda založená na adaptivním slovníku. Taková metoda může začít s prázdným slovníkem nebo s malým výchozím slovníkem, přidávat do něj řetězce podle toho, jak jsou nalezeny ve vstupních datech, a mazat staré záznamy, pokud je počet záznamů ve slovníku příliš velký. Algoritmus takové metody se vždy skládá z hlavní smyčky, kdy každá iterace začíná čtením vstupních dat a jeho rozdělením (někdy také parsováním) na řetězce. Poté se pro přečtený řetězec prohledá slovník a v případě nalezení shody je na výstup zapsán index řetězce ve slovníku. V opačném případě se na výstup zapíše nekomprimované slovo (často je však zakódováno nějakou jinou, méně efektivní metodou) a současně se přidá do slovníku. V posledním kroku může nastat kontrola, zda má být starý či dlouho nepoužívaný záznam ze slovníku odstraněn. [1, 2]

Protože slovníkové metody nepoužívají statistický model a tím pádem jejich kompresní efekt nezávisí na kvalitě modelu jako je tomu u statistických metod, dosahují slovníkové metody lepších kompresních poměrů než metody statistické. [3] Některé kompresní metody navíc kombinují slovníkový a statistický přístup tak, že indexy frází nalezených ve slovníku před zápisem na výstup kódují nějakou statistickou metodou, např. statickým nebo dynamickým Huffmanovým kódováním, a tím dosahují ještě lepších kompresních výsledků. [1] Z výše uvedeného dále plyne, že slovníkové kompresní metody jsou obzvlášť efektivní pro takové zdroje, které často generují relativně malý počet řetězců. [2]

Slovníkové metody komprese poskytují dobré výsledky na obecných datech, tj. fungují srovnatelně dobře na obrazových a zvukových datech stejně jako na textu. Díky svým dobrým vlastnostem začaly slovníkové metody v praktickém použití postupně převažovat a všechny dnes běžně používané archivační programy, jako je gzip, 7-Zip, zlib, PKZIP nebo XZ Utils, používají slovníkové kompresní metody jako svůj hlavní transformační krok. [1, 13]

Slovníkové kompresní metody založené na principech algoritmů LZ77 a LZ78 se také někdy souhrnně označují jako algoritmy rodiny LZ podle svých objevitelů, kterými jsou vědci Abraham Lempel a Jacob Ziv. Mezi slovníkové kompresní algoritmy proto patří všechny algoritmy rodiny LZ, jako je LZ77, LZ78, LZSS, LZW, LZMA nebo algoritmus DEFLATE.

3.4 Další charakterizace

Kompresní metody, které čtou vstupní data právě jednou, se nazývají *jedno-průchodové*. Metody, které čtou vstupní data dvakrát, se nazývají *dvou-průchodové*. Příkladem jednoprůchodové metody je dynamické Huffmanovo kódování, algoritmy LZ77 a LZ78 nebo metoda PPM. Mezi dvou-průchodové metody pak patří např. statické Huffmanovo kódování a Shannonovo–Fanovo kódování. [1]

Pokud kompresor i dekompresor používají v podstatě stejný algoritmus, akorát v opačném směru, označuje se taková metoda jako *symetrická*. Symetrická metoda se používá v situacích, kdy se komprimuje obdobný počet souborů, jako se dekomprimuje. U *asymetrické* metody se může stát, že kompresor nebo dekompresor bude muset provádět intenzivnější výpočty. Asymetrická metoda je přirozenou volbou v situacích, kdy jsou soubory komprimovány do archivu, ve kterém pak budou často používány a dekomprimovány. Opačný případ je naopak užitečný v prostředí, ve kterém se soubory neustále aktualizují a provádějí zálohy. V této situaci je malá pravděpodobnost, že se soubor se zálohou použije, takže dekompresor není používán příliš často. Mnoho moderních kompresních metod je asymetrických. [1]

Kapitola 4

Datové struktury používané v kompresních algoritmech

Tato kapitola se zabývá popisem a analýzou datových struktur, které jsou používané v algoritmech LZ77, LZ78, LZSS a LZW. Při následné analýze a popisu implementace těchto algoritmů v dalších kapitolách budou dané datové struktury považovány již za známé. Pro potřeby popisu datových struktur jsou také uvedeny základní pojmy z teorie složitosti a teorie grafů.

4.1 Základní pojmy z teorie složitosti

Algoritmy se porovnávají na základě různých, a někdy i protichůdných, kritérií. Kromě přehlednosti algoritmu, která může vést k lepšímu pochopení a k tomu, že takový algoritmus může být méně náchylný k chybám při implementaci, je vhodné zkoumat jeho *časovou* (někdy také *operační*) a *paměťovou* (někdy také *prostorovou*) složitost. [15]

Časová složitost algoritmu reprezentuje počet elementárních operací, které algoritmus musí vykonat v závislosti na velikosti vstupních dat. Pojem velikosti vstupních dat je závislý na konkrétním problému, který algoritmus řeší, např. při řazení posloupnosti n čísel je za velikost vstupu považováno právě n. Ale např. Euklidův algoritmus na svém vstupu pokaždé dostává dvě čísla a běží různě dlouho v závislosti na jejich hodnotách (někdy se také říká, že je takový algoritmus citlivý na vstupní data). V tomto případě se pak považuje za velikost vstupu maximum ze zadaných dvou čísel. [16]

Časovou složitost je možné uvažovat v tzv. nejlepším, průměrném a nejhorším případě. Zpravidla se však časová složitost uvažuje v nejhorším možném případě pro konkrétní vstup, tj. vyjadřuje počet operací potřebných k vyřešení nejhorší možné úlohy při dané konkrétní velikosti vstupu. [15] Příkladem může být řazení posloupnosti n čísel, kdy nejlepší případ zpravidla je, že všechna čísla na vstupu jsou již vzestupně seřazená, průměrným případem může být rovnoměrně náhodné rozložení čísel, a nejhorším případem může být situace, kdy je všech n čísel na vstupu v sestupném pořadí.

Prostorová složitost algoritmu analogicky reprezentuje počet paměťových buněk, které algoritmus potřebuje k vyřešení instance dané úlohy, a platí pro ni stejné úvahy, jaké byly uvedeny u složitosti časové.

Složitost algoritmu lze zřídkakdy určit přesně. Nejprve by bylo nutné definovat pojem elementární operace počítače, což by mohlo vést k tomu, že by výsledek byl svázán s konkrétním počítačem, a dále by bylo potřeba algoritmus popsat do nejmenších detailů, včetně různých rutinních operací, tj. vycházet z konkrétního naprogramování v konkrétním programovacím jazyce. V takovém případě i u jednoduchých algoritmů může jít o velmi náročnou úlohu, u těch těžších

to zpravidla bývá i nemožné. Navíc, z praktického hlediska je důležité, jak se algoritmus bude chovat pro velké instance daného problému. Z těchto důvodu se v teoretické analýze algoritmů složitost odhaduje pouze asymptoticky. [15]

Asymptotickou složitost lze neformálně chápat jako složitost vyjádřenou pro dostatečně velké instance problému, pro které se zřetelně projeví řád růstu složitosti v závislosti na velikosti vstupních dat. Pro dostatečně velké vstupy lze v takovém případě zanedbat multiplikativní konstanty a členy nižších řádů v přesné složitosti. Asymptotická složitost tedy vyjadřuje, jak roste složitost algoritmu s velikostí vstupu v limitě, když velikost vstupních dat roste nade všechny meze. Pak platí, že asymptoticky lepší algoritmus bude lepší pro všechny instance daného problému až na konečný počet výjimek. [17]

▶ **Definice 4.1** (Asymptotická horní mez [16]). *Nechť $f, g : \mathbb{N} \to \mathbb{R}$ jsou dvě reálné funkce. Funkce $f(n)$ je nejvýše řádu $g(n)$, psáno $f(n) = O(g(n))$ právě tehdy, když platí*

$$\exists c \in \mathbb{R}^+ \wedge \exists n_0 \in \mathbb{N} \wedge \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

Značení asymptotické horní meze $f(n) = O(g(n))$ je nepřesné, neboť je zřejmé, že $O(g(n))$ označuje množinu všech funkcí, které splňují uvedenou definici, a bylo by proto vhodnější množinové značení $f(n) \in g(n)$. Ve většině literatury se však používá původní značení a tato konvence bude proto dodržována i v tomto textu. [16]

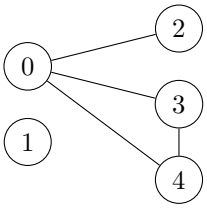
Asymptotická horní mez tedy vyjadřuje, že dostatečně velký násobek funkce $g(n)$ shora omezuje funkci $f(n)$ až na konečný počet výjimek. Často může být přínosné odhadnout složitost algoritmu v nejhorším možném případě pomocí dalších mezí, konkrétně pomocí asymptotické dolní meze $\Omega(g(n))$ a asymptotické těsné meze $\Theta(g(n))$. To však není triviální úloha, protože analýzou algoritmu je často možné dostat pouze horní odhad počtu provedených instrukcí či paměťových míst. [16] Více o těchto dalších asymptotických mezích je možné se dočíst v [16] nebo [17].

Při reálném posuzování a porovnávání algoritmů je navíc vhodné doplnit asymptotickou analýzu testováním algoritmů pro konkrétní data na konkrétním počítači. Pokud je např. výsledkem asymptotické analýzy časová složitost $O(n^2)$ pro jeden algoritmus a časová složitost $O(n \log^4 n)$ pro druhý algoritmus, pak z této analýzy vychází druhý algoritmus jako lepší z hlediska operační složitosti. Pokud by však přesná složitost prvního algoritmu byla $n^2 - 5n$ a přesná složitost druhého algoritmu byla $20n(log_2 n)^4$, pak by se převaha druhého algoritmu projevila až zhruba od $n = 5 \cdot 10^6$, což jsou hodnoty, kterých reálná data v dané problémové doméně ani nemusejí dosáhnout. [15]

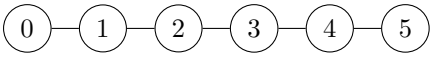
Asymptotická horní mez operace nad datovou strukturou, jejíž velikost a složení se v závislosti na operacích nad takovou strukturou mění, nemusí nejlépe vystihovat skutečné chování operace v rámci celého životního cyklu této datové struktury. Z toho důvodu se u algoritmů a operací, které se volají mnohokrát nad dynamicky se měnící datovou strukturou, zavadí tzv. *amortizovaná složitost*. Při amortizované analýze se zprůměruje čas, který je potřebný k provedení posloupnosti operací nad datovou strukturou, přese všechny provedené operace. Pomocí amortizované analýzy je pak možné ukázat, že průměrné náklady na operaci jsou malé, pokud jsou zprůměrovány přes celou posloupnost operací, ačkoliv jednotlivá operace v této posloupnosti může být velmi drahá. Je vhodné poznamenat, že amortizovaná analýza nezahrnuje pravděpodobnost, tj. zaručuje průměrný výkon dané operace v nejhorším možném případě. [16, 17]

▶ **Definice 4.2** (Amortizovaná časová složitost [16]). *Operace A nad dynamickou datovou strukturou má v daném kontextu svého provádění amortizovanou časovou složitost $O^*(f(n))$, pokud posloupnost $k \in \mathbb{N}$ operací A má celkovou časovou složitost $O(k \cdot f(n))$. Parametr $n \in \mathbb{N}_0$ je velikost dynamické množiny po provedení této posloupnosti operací.*

Pro odhad časové složitosti jednotlivých provedení takové operace se stále bere ten nehorší možný případ. Amortizovanou časovou složitost je třeba stanovovat přes dostatečně dlouhé posloupnosti operací. Obvyklý kontext amortizované analýzy je posloupnost všech operací od ini-



Obrázek 4.1 Grafické znázornění grafu G



Obrázek 4.2 Cesta P_5

cializace datové struktury až po vybudování struktury velikosti n. Časová složitost jednoho konkrétního volání takové operace v nejhorším možném případě však může být řádově vyšší. [16]

4.2 Základní pojmy z teorie grafů

Některé datové struktury používané v kompresních algoritmech, jako je trie nebo binární vyhledávací strom, mají stromovou strukturu. Před jejich popisem je proto vhodné uvést alespoň základní pojmy z teorie grafů.

▶ **Definice 4.3** (Neorientovaný graf [16]). *Neorientovaný graf (zkráceně graf) je uspořádaná dvojice (V, E), kde*

- *V je neprázdná konečná množina vrcholů (někdy také uzlů);*
- *E je množina hran.*

Hrana je dvouprvková podmnožina V, tedy neuspořádaná dvojice vrcholů, a značí se $\{u, v\}$, kde $u, v \in V$. Platí, že $E \subseteq 2^V$, kde 2^V značí potenční množinu množiny V, tedy množinu všech podmnožin množiny V. Množina všech dvouprvkových podmnožin množiny V, tj. množina všech možných hran, se značí $\binom{V}{2}$. Platí, že $E \subseteq \binom{V}{2} \subseteq 2^V$.

Množina vrcholů grafu G se značí $V(G)$, množina hran se pak analogicky značí $E(G)$. Počet vrcholů grafu G se zpravidla označuje písmenem $n = |V(G)|$, počet hran grafu G se pak značí písmenem $m = |E(G)|$. [16]

Grafické znázornění grafu $G = (\{0, 1, 2, 3, 4\}, \{\{0, 2\}, \{0, 3\}, \{0, 4\}, \{3, 4\}\})$ je možné vidět na obrázku 4.1.

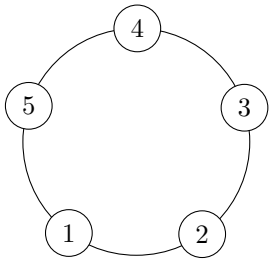
▶ **Definice 4.4** (Stupeň vrcholu [16]). *Nechť $G = (V, E)$ je graf a v je jeho vrchol. Symbolem $deg_G(v)$ se značí počet hran grafu G obsahujících vrchol v. Toto číslo se pak nazývá stupněm vrcholu v.*

Ačkoliv existuje několik elementárních typů grafů, pro účely tohoto textu budou postačující dva základní druhy, kterými jsou cesta a kružnice.

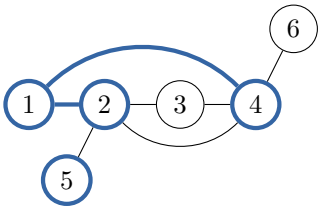
▶ **Definice 4.5** (Cesta [16]). *Nechť $m \geq 0$. Cesta P_m délky m je graf*

$$P_m = (\{0, \ldots, m\}, \{\{i, i+1\} \mid i \in \{0, \ldots, m-1\}\})$$

U cesty P_m index m koresponduje s počtem hran grafu, nikoliv však s počtem vrcholů. Graf cesty délky 5 ilustruje obrázek 4.2.



Obrázek 4.3 Kružnice C_5



Obrázek 4.4 Podgraf

▶ **Definice 4.6** (Kružnice [16]). *Nechť $n \geq 3$. Kružnice C_n délky n je graf*

$$C_n = (\{1, \ldots, n\}, \{\{i, i+1\} \mid i \in \{1, \ldots, n-1\}\} \cup \{\{1, n\}\}).$$

Kružnici délky 5, tedy graf C_5, ukazuje obrázek 4.3.

▶ **Definice 4.7** (Podgraf [16]). *Graf H je podgrafem grafu G právě tehdy, když $V(H) \subseteq V(G)$ a současně $E(H) \subseteq E(G)$. Podgraf H grafu G se pak značí $H \subseteq G$.*

Příklad podgrafu ilustruje obrázek 4.4.

▶ **Definice 4.8** (Izomorfismus grafů [16]). *Nechť G a H jsou dva grafy. Funkce $f : V(G) \to V(H)$ je izomorfismus grafů G a H, pokud platí tyto podmínky:*

- *f je bijektivní zobrazení;*
- *$\forall u, v \in V(G) : \{u, v\} \in E(G) \Leftrightarrow \{f(u), f(v)\} \in E(H)$.*

Dva grafy G a H jsou izomorfní, pokus existuje izomorfismus grafů G a H.

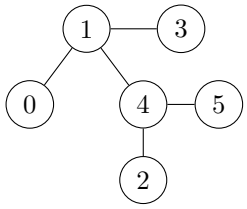
Podgraf grafu G izomorfní s nějakou cestou P se nazývá *cesta v grafu*. Pokud má cesta P v grafu G koncové vrcholy u a v, pak se P označuje také jako *cesta z u do v* nebo zjednodušeně *u-v cesta*. Podgraf grafu G izomorfní s nějakou kružnicí C se nazývá *kružnice v grafu* nebo také *cyklus*. [16]

▶ **Definice 4.9** (Souvislost grafu [16]). *Graf G je souvislý, pokud v G pro každé jeho dva vrcholy $u, v \in V(G)$ existuje u-v cesta. V opačném případě je graf nesouvislý.*

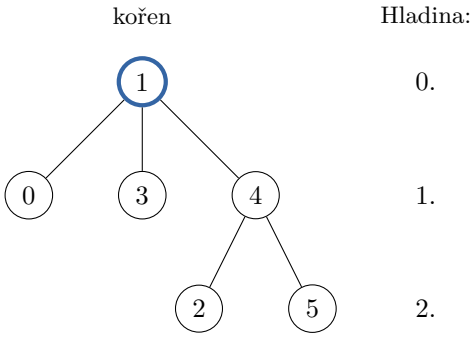
Nyní je možné přistoupit k definici stromu.

▶ **Definice 4.10** (Strom [16]). *Graf G je strom, pokud je souvislý a neobsahuje žádnou kružnici. Pokud pro vrchol $v \in V(G)$ platí, že $deg_G(v) = 1$, pak se takový vrchol nazývá list.*

Strom $G = (\{0, 1, 2, 3, 4, 5\}, \{\{0, 1\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{4, 5\}\})$, který má právě 4 listy, ukazuje obrázek 4.5.



Obrázek 4.5 Strom se 4 listy



Obrázek 4.6 Zakořeněný strom

▶ **Definice 4.11** (Zakořeněný strom [16]). *Zakořeněný strom G je takový strom, ve kterém je jeden vrchol označen jako kořen. Leží-li $u \in V(G)$ na cestě z vrcholu $v \in V(G)$ do kořene, pak vrchol u se nazývá předek vrcholu v a vrchol v se nazývá potomek vrcholu u. Pokud je navíc $\{u, v\}$ hranou stromu, pak vrchol u se nazývá otec vrcholu v a vrchol v syn vrcholu u. Vrcholy jsou rozděleny podle vzdálenosti od kořene do hladin, v nulté hladině leží kořen, v první hladině jeho synové, atd. Hloubka zakořeněného stromu je číslo poslední hladiny, tedy maximální délka cesty mezi kořenem a listem.*

Strom G z předchozího příkladu 4.5, ve kterém byl vrchol 1 označen jako kořen, je možné vidět na obrázku 4.6.

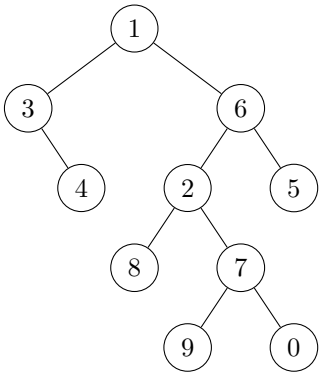
▶ **Definice 4.12** (Binární strom [16]). *Strom G se nazývá binární, pokud platí následující podmínky:*

- *G je zakořeněný;*
- *každý vrchol stromu G má nejvýše dva syny;*
- *u každého syna je rozlišováno, který je levý a pravý.*

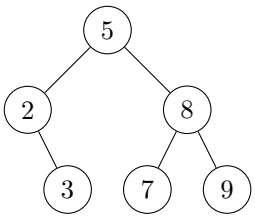
Binární strom ukazuje obrázek 4.7.

4.3 Samo-vyvažovací binární vyhledávací strom

Jedno z nejčastějších použití binárních stromů je při vyhledávání. Před definicí binárního vyhledávacího stromu je potřeba zavést některá potřebná značení. Pro vrchol v binárního stromu T značí $T(v)$ podstrom obsahující vrchol v a všechny jeho potomky. Levý syn vrcholu v se značí $l(v)$, pravý syn pak $r(v)$. Levý podstrom vrcholu v se značí $L(v)$, odpovídá tedy stromu $T(l(v))$. Analogicky, pravý podstrom vrcholu v se značí $R(v)$ a odpovídá stromu $T(r(v))$. Počet vrcholů stromu T, $T(v)$, $L(v)$ a $R(v)$ se značí jako $|T|$, $|T(v)|$, $|L(v)|$ a $|R(v)|$. [16]



Obrázek 4.7 Binární strom



Obrázek 4.8 Vyvážený binární vyhledávací strom

Hloubka stromu T, resp. podstromu $T(v)$, se značí $h(T)$, resp. $h(T(v))$, a podle definice 4.11 odpovídá maximu z délek cest z vrcholu v do listů. Pokud vrchol v nemá levého, resp. pravého, syna, pak $l(v) = \emptyset$, resp. $r(v) = \emptyset$. $T(\emptyset)$ pak odpovídá prázdnému stromu, tedy grafu bez vrcholů a hran. Hloubku prázdného stromu lze dodefinovat jako $h(\emptyset) = -1$. [16]

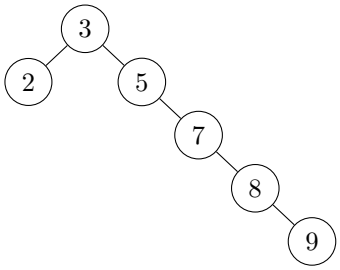
▶ **Definice 4.13** (Binární vyhledávací strom [16])**.** *Binární vyhledávací strom (zkráceně BVS) je binární strom T, jehož každý vrchol $v \in V(T)$ obsahuje unikátní klíč $k(v)$ a pro jehož každý vrchol $v \in V(T)$ platí:*

- $a \in L(v) \Rightarrow k(a) < k(v)$;
- $b \in R(v) \Rightarrow k(b) > k(v)$.

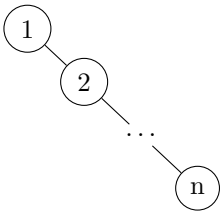
Z předchozí definice je tedy plyne, že vlastnost, která binární vyhledávací strom odlišuje od binárního stromu, spočívá v tom, že pro každý vrchol $v \in V(T)$ stromu jsou klíče všech prvků v jeho levém podstromu $L(v)$ menší než $k(v)$ a hodnoty všech klíčů v pravém podstromu $R(v)$ větší než $k(v)$. Je zřejmé, že z této vlastnosti vyplývá předpoklad, že všechny klíče, které se mají do binárního vyhledávacího stromu vkládat, lze konzistentním způsobem mezi sebou porovnat a uspořádat. [18]

Binární strom na obrázku 4.7 není binární vyhledávací strom, protože např. levý syn kořene obsahuje klíč, jehož hodnota je větší než hodnota klíče uloženého v kořeni. Obrázky 4.8 a 4.9 naopak ukazují dva binární vyhledávací stromy, které sice obsahují stejné prvky, ale liší se svým tvarem.

Mezi základní operace, které se nad binárním vyhledávacím stromem T provádějí, patří vložení, smazání a vyhledání prvku, dále pak také nalezení minima, maxima, předchůdce a následníka prvku. Pro účely analýzy operační složitosti se bude dále předpokládat, že klíče lze mezi sebou porovnávat v konstantním čase. Z obrázků 4.8 a 4.9 je zřejmé, že operační složitost těchto operací se odvíjí od hloubky stromu, tj. pro obecný binární vyhledávací strom je složitost těchto operací $O(h(T))$. Pokud jsou do prázdného binárního vyhledávacího stromu postupně vloženy prvky $1, 2, \ldots n$, pak vznikne zdegenerovaný strom, který odpovídá spojovému seznamu,



Obrázek 4.9 Nevyvážený binární vyhledávací strom



Obrázek 4.10 Zdegenerovaný binární vyhledávací strom

a operační složitost operací nad takovým stromem je proto $O(n)$. Zdegenerovaný strom ilustruje obrázek 4.10. [16, 17]

Řešením tohoto problému je trvat na dodatečné strukturální podmínce, které se říká *vyváženost*. Nabízí se otázka, jak přísně má být podmínka vyváženosti nastavena. Například binární vyhledávací strom, pro jehož každý vrchol v platí $||L(v)| - |R(v)|| \leq 1$, se nazývá *dokonale vyvážený*. Lze však ukázat, že pokud má tento strom zůstat dokonale vyvážený i po provedení operací pro vložení či smazání prvku, pak alespoň jedna z těchto operací nutně musí mít minimálně lineární operační složitost vůči počtu prvků ve stromu. Důkaz tohoto tvrzení lze nalézt např. v [16].

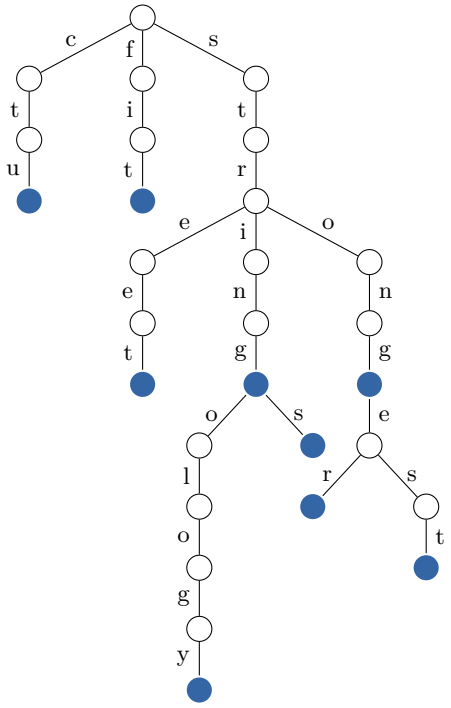
V praxi je proto vhodné nastavit podmínku pro vyváženost méně restriktivně, postačující je udržovat hloubku $O(\log(n))$ vůči počtu prvků ve stromu. Binární vyhledávací stromy, které po vložení či smazání prvku kontrolují vyváženost struktury a v případě potřeby provedou restrukturalizaci k zachování požadované hloubky stromu, se nazývají *samo-vyvažovací* (z angl. *self-balancing binary search trees*). Mezi nejpoužívanější takové struktury patří AVL stromy a červeno-černé stromy, které udržují logaritmickou hloubku stromu. Všechny uvedené základní operace nad těmito strukturami proto mají časovou složitost $O(\log(n))$. [16, 18]

Více o AVL stromech, červeno-černých stromech a o principu udržování podmínky vyvážení při modifikaci prvků stromu se lze dočíst v jakékoliv literatuře zabývající se datovými strukturami, např. v [16] nebo [17].

4.4 Trie

U binárních vyhledávacích stromů je při použití jeho operací pro vyhledávání, vložení či smazání prvku potřeba porovnávat klíče, které jsou ve vrcholech uloženy. V nejhorším případě má takové porovnání lineární složitost vůči délce kratšího klíče. Navíc, u binárních vyhledávacích stromů je operační složitost jejich operací úměrná počtu prvků, které jsou ve stromu uloženy. Stromová datová struktura, která používá pouze části klíče k navigaci při vyhledávání, a nabízí operace s časovou složitostí, která není úměrná celkovému počtu prvků ve struktuře, se nazývá *trie* [1] nebo také *prefixový* či *písmenkový strom*.

[1] V českém jazyce vyslovováno jako „trije“ a skloňováno podle vzoru růže. [16]



Obrázek 4.11 Trie

V prefixovém stromu se na klíče nahlíží jako na posloupnost znaků nějaké abecedy a trie si udržuje vnitřní organizaci struktury vůči těmto znakům, nikoliv vůči celým vkládaným klíčům. [19]. Trie je tedy struktura, která je vhodná pro zpracovávání takových klíčů, které lze jednoduše reprezentovat jako řetězce. Čísla v plovoucí řádové čárce mohou být pro tuto strukturu problematická, protože stejné číslo v plovoucí řádové čárce je možné reprezentovat jako řetězec různými způsoby, např. číslo 1 lze reprezentovat jako 1.0, 1.00, +1.0, atd. [20]

Trie je zakořeněný strom, na rozdíl od binárních vyhledávacích stromů však uzly trie nejsou binární. Místo toho každý vrchol může mít potenciálně jednoho syna pro každý možný znak dané abecedy Σ, tedy počet synů každého vrcholu je maximálně roven $|\Sigma|$. Z každého vrcholu vedou hrany, které jsou označeny vzájemně různými znaky abecedy. V kořeni odpovídají hrany do jeho synů prvnímu symbolu daného řetězce, o hladinu níž druhému, atd. Vrcholům lze přiřadit řetězce tak, že se zřetězí všechny symboly na cestě z kořene do daného vrcholu. Kořen bude odpovídat prázdnému řetězci a čím hlouběji se trie bude procházet, tím delší bude vznikat výsledný řetězec. Vrcholy odpovídající vloženým klíčům jsou označené a kromě listů mohou být označeny i vnitřní vrcholy stromu v případě, že jeden vložený řetězec je prefixem jiného. Další rozdíl oproti binárním vyhledávacím stromům tedy je, že zatímco v BVS jsou celé klíče uložené v samotných vrcholech, v prefixovém stromu je vložený klíč reprezentován pozicí označeného vrcholu ve stromu. Je zřejmé, že hloubku stromu určuje nejdelší vložený řetězec. [16, 20, 21]

Každý uzel u trie odpovídá prefixu nějakého řetězce z množiny vložených řetězců a všichni potomci vrcholu u mají společný prefix, který odpovídá uzlu u. Jedna z významných výhod této struktury proto je, že pokud se stejný prefix vyskytuje v množině vložených řetězců vícekrát, pak je takový prefix reprezentovaný vždy pouze jedním uzlem stromu. Kořen trie je uzel odpovídající prázdnému prefixu. Uzel reprezentující prefix σ_1 obsahuje pro každý symbol $a \in \Sigma$ ukazatel na uzel, který reprezentuje prefix $\sigma_1 a$, pokud takový uzel ve stromu existuje, tj. pokud existuje $\sigma_2 \in \Sigma^*$ takové, že klíč odpovídající řetězci $\sigma_1 a \sigma_2$ byl do trie vložen. [20, 21]

Prefixový strom po vložení množiny řetězců *string*, *strings*, *strong*, *stronger*, *strongest*, *stringology*, *street*, *ctu* a *fit* ukazuje obrázek 4.11. Mezi základní operace, které trie nabízí, patří vložení,

smazání a vyhledání řetězce. Zásadní výhodou této struktury je, že složitost všech uvedených operací je $O(l)$, kde l značí délku vkládaného, mazaného či vyhledávaného řetězce. Operační složitost těchto operací tedy vůbec nezávisí na celkovém počtu klíčů, které jsou ve struktuře uložené. Trie proto nabízí rychlejší vyhledávání oproti binárnímu vyhledávacímu stromu, který provede v nejhorším případě $O(\log n)$ porovnání (pokud je hloubkově vyvážený) a celková složitost operace vyhledávání podle řetězce délky l proto je $O(l \cdot log(n))$, kde n odpovídá počtu uložených klíčů. Vzhledem k tomu, že klíče nejsou v prefixovém stromu uloženy explicitně a uzly jsou sdíleny mezi klíči se stejným prefixem, vyžaduje trie ve srovnání s binárním vyhledávacím stromem méně místa v paměti, obzvlášť pokud obsahuje velké množství kratších řetězců. [20]

4.5 Kruhová fronta

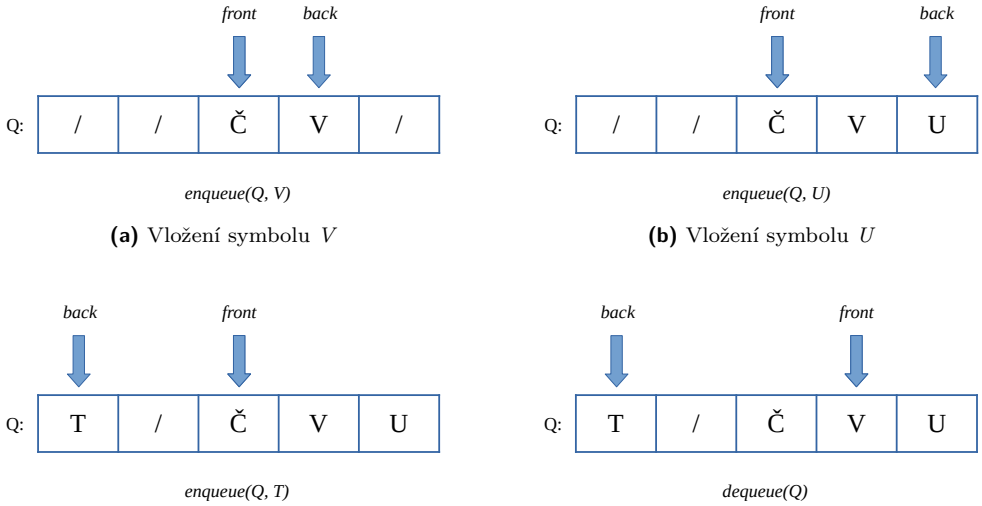
Fronta je jednou ze základních datových struktur. Jde o posloupnost prvků, která roste přidáváním prvku na její konec (v angl. literatuře *back* nebo *tail*) a zmenšuje se odebíráním prvku z jejího začátku (v angl. literatuře *front* nebo *head*). Poslední prvek proto musí čekat, dokud nejsou z fronty vyzvednuty všechny prvky, které mu ve frontě předcházejí. Tato vlastnost datové struktury se nazývá *FIFO* (z angl. *first-in-first-out*), která způsobuje, že se struktura při modifikaci prvků chová jako běžná fronta zákazníků. Pokud se vkládá nový prvek, pak je zařazen na konec fronty, stejně jako nově příchozí zákazník. Analogicky, odbavený prvek je vždy ten, který se nachází na začátku fronty, stejně jako zákazník, který je na začátku řady a ve frontě čeká nejdéle. Operace vložení prvku x do fronty Q se obvykle označuje *enqueue(Q, x)*, operace odebrání prvku se označuje jako *dequeue(Q)*. [17, 19]

Jedním ze způsobů, jak frontu implementovat, je pomocí spojového seznamu, tedy jako dynamický uspořádaný soubor uzlů, kde každý uzel obsahuje data a ukazatel na další uzel v seznamu. Je zřejmé, že v této implementaci mají obě operace *enqueue(Q, x)* a *dequeue(Q)* konstantní operační složitost, tedy $O(1)$. Implementace pomocí spojového seznamu však s sebou nese dvě zásadní nevýhody. Pokud je potřeba přečíst data z uzlu, který se nachází uvnitř fronty místo na jejím začátku (jako to umožňuje např. kontejner *std::deque* ve standardní knihovně jazyka C++), pak má taková operace lineární složitost vůči pozici prvku ve frontě, protože je potřeba nejprve přistoupit na všechny prvky, které mu v seznamu předcházejí. Druhou nevýhodou je, že uzly se mohou nacházet kdekoliv v paměti, tedy nesousedí na po sobě jdoucích adresách a tím porušují princip prostorové lokality pro efektivní práci s vyrovnávací pamětí procesoru. Princip prostorové lokality se předpokládá, že po přístupu k informacím na konkrétní adrese v paměti je pravděpodobné, že budou brzy požadována data na sousedních adresách. [10, 19, 22]

Oba tyto nedostatky řeší implementace pomocí pole, tj. struktury fixní velikosti uspořádaných prvků uložených na po sobě jdoucích adresách v paměti. V této implementaci je atribut fronty *front* reprezentovaný indexem, na kterém se nachází první prvek ve frontě, a atribut *back* je reprezentovaný indexem, na kterém se nachází poslední prvek fronty. Kdykoliv se některý z těchto indexů dostane na konec pole, je při další operaci vrácen na jeho začátek. Na pole se tedy v tomto případě nahlíží jako na cyklickou strukturu, ve které prvek na nulté pozici následuje za prvkem na poslední pozici. Indexy *head* a *tail* ohraničují platný rozsah fronty a cyklicky se pohybují po poli podle toho, jak jsou prvky do fronty vkládány a odebírány. Cyklickou sémantiku indexů lze realizovat pomocí zbytku po celočíselném dělení velikostí pole. Této datové struktuře se říká *kruhová fronta*. Operace náhodného přístupu k prvkům uvnitř kruhové fronty pak má operační složitost $O(1)$. [18, 22]

▶ **Příklad 4.14.** Fronta Q je na začátku prázdná, atributy *front* a *back* ukazují na nultý index v poli. Následně jsou do fronty Q byly postupně přidávány a odebírány prvky tak, že fronta Q bude obsahovat obsahuje pouze prvek $\check{C}$ na indexu 2. Na ostatních pozicích v poli se nacházejí data z předchozích operací, které byly nad frontou vykonány.

Nyní jsou do fronty postupně vloženy prvky V, U a T. Během těchto operací atribut *back* reprezentující konec fronty přeteče přes velikost pole a vrátí se zpět na jeho začátek. Na závěr



(a) Vložení symbolu V **(b)** Vložení symbolu U

(c) Vložení symbolu T **(d)** Odstranění prvního prvku fronty

Obrázek 4.12 Operace nad kruhovou frontou

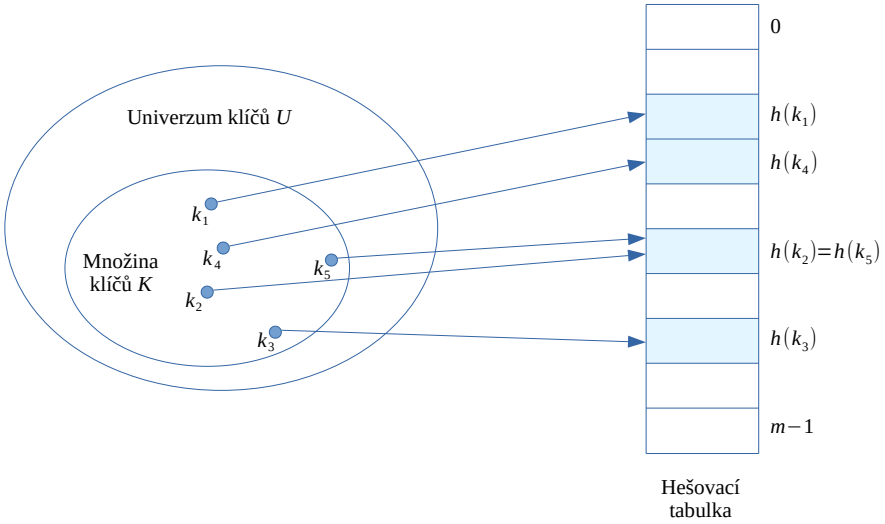
je ještě z fronty odebrán prvek *Č*. Posloupnost těchto čtyř operací nad kruhovou frontou Q ilustruje obrázek 4.12. Data, která se v poli nacházela před provedením těchto čtyř operací, jsou pro odlišení od ostatních prvků v obrázku reprezentována symbolem dopředného lomítka.

Nevýhodou kruhové fronty je fixní velikost pole a je proto ideální volbou v situacích, kdy je maximální možný počet prvků ve frontě známý dopředu. Pokud toto není zajištěno, je možné při vkládání prvků do fronty provést podle potřeby realokaci pole, v takovém případě však již bude operační složitost vložení prvku do fronty $O(n)$, kde n reprezentuje počet prvků ve frontě po provedením operace. Amortizovaná složitost operace $enqueue(Q, x)$ by i v tomto případě zůstala konstantní, tedy $O^*(1)$, protože jde pouze o modifikaci techniky nafukovacího pole. Analýzu amortizované složitosti nafukovacího pole lze nalézt např. v [16].

4.6 Hešovací tabulka

Binární vyhledávací strom a trie jsou datové struktury, které nabízejí všechny tři základní operace abstraktního datového typu *množina*, mezi které patří vložení, vyhledání a smazání klíčů z nějakého univerza U, kde univerzum může být např. množina celých čísel nebo množina všech možných řetězců nad abecedou Σ. Pokud je potřeba ukládat kromě klíče i nějakou jeho hodnotu, pak je vhodné použít abstraktní datový typ *slovník*. Slovník umožňuje ukládat konečnou množinu klíčů a každému z nich přiřazuje hodnotu, což může být prvek ze stejného či nějakého jiného univerza. Slovník je proto konečná množina dvojic *(klíč, hodnota)*, ve které se neopakují klíče. [16]

Pokud jsou klíče malá celá čísla, pak je možné použít pole k implementaci slovníku tak, že klíč je interpretován jako index pole a hodnota spojená s klíčem k je uložena do pole na index k v konstantním čase. V praxi je však velikost univerza U velmi vysoká, nebo je dokonce univerzum U nekonečné, a tuto techniku proto nelze kvůli paměťovým nárokům použít. *Hešování* je metoda, která zvládá i složitější typy klíčů, které pomocí aritmetických operací transformuje na indexy pole, ve kterém jsou uložené samotné klíče, pokud je implementována množina, nebo uspořádané



Obrázek 4.13 Hešování [17]

dvojice *(klíč, hodnota)*, pokud je implementován slovník. Tato struktura, která využívá hešování k transformaci klíčů na indexy pole, se nazývá *hešovací tabulka*. [17, 23]

Hešování je klasickým příkladem kompromisu mezi paměťovou náročností a časovou složitostí. Pokud by dostupné paměti bylo neomezeně mnoho, pak by vyhledávání bylo možné realizovat jediným přístupem do paměti jednoduchým použitím klíče jako indexu do velmi velkého pole. Tento ideál je však očividně nedosažitelný kvůli obrovskému počtu možných hodnot klíče. Na druhou stranu, pokud by neexistovalo žádné časové omezení, pak je možné vyhledávání realizovat s minimálním množstvím paměti jako jednoduché sekvenční vyhledávání v neseřazeném poli. Hešování poskytuje způsob, jak využít rozumné množství paměti i času a najít rovnováhu mezi těmito dvěma extrémy. [23]

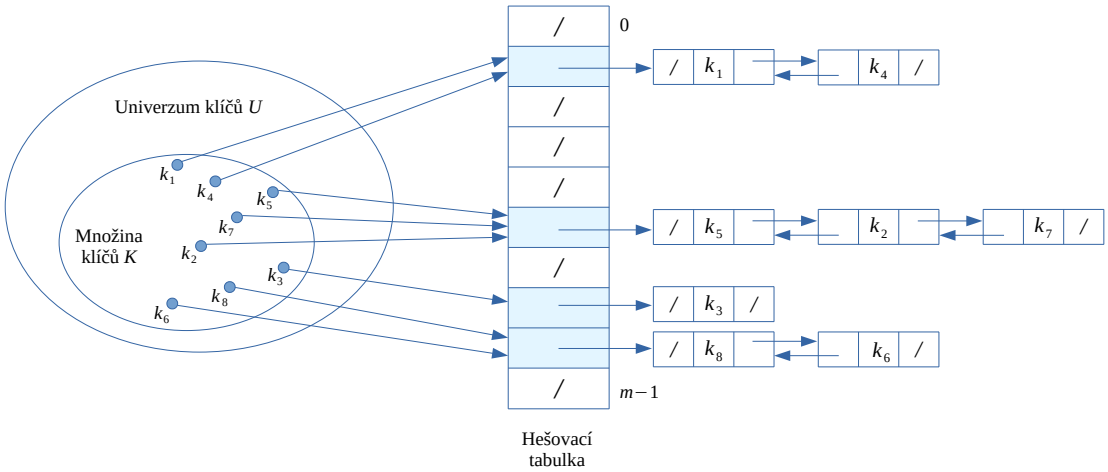
Ačkoliv vyhledání, vložení či smazání prvku v hešovací tabulce může v nejhorším případě trvat $O(n)$, tak v praxi poskytují hešovací tabulky výborné výsledky a všechny uvedené operace dosahují konstantní amortizované operační složitosti $O^*(1)$ při paměťové náročnosti $O(|K|)$, kde K reprezentuje množinu ukládaných klíčů. [17]

Proces hešování se skládá ze dvou základních částí. Tou první je výpočet hešovací funkce $h : U \to P$, která mapuje prvky univerza U na konečnou množinu přihrádek $P = \{0, \ldots, m-1\}$ hešovací tabulky, kde m je velikost hešovací tabulky a je typicky výrazně menší než $|U|$. Číslo přihrádky, na kterou má být klíč $k \in U$ uložen, že vypočítá z klíče k jako hodnota hešovací funkce $h(k)$. Hešovací funkce tedy zmenšuje rozsah indexů pole a tím i jeho velikost, tj. místo velikosti $|U|$ si vystačí s velikostí m. Situaci přibližuje obrázek 4.13. [17, 23]

Ideální hešovací funkci je taková, kterou lze spočítat v konstantním čase a posloupnost n vkládaných prvků rozprostře mezi m přihrádek dokonale rovnoměrně. V ideálním případě by tedy hešovací funkce měla chovat náhodně. Ideální hešovací funkce je v praxi nedosažitelná, existuje však řada hešovacích funkcí, které poskytují dobré výsledky a jejich chování se na reálných datech jeví jako prakticky náhodné, jako je např. lineární kongruence, vyšší bity součinu nebo skalární součin. [16] Více o těchto hešovacích funkcích se lze dočíst např. v [16] nebo [17].

Pro zachování dobrých vlastností struktury se sleduje zaplnění hešovací tabulky, které se nazývá *faktor naplnění*, a v případě překročení stanovené hranice se hodnota m zvýší, zvolí se nová hešovací funkce a všechny prvky se přehešují, tj. pro každý prvek z původní hešovací tabulky se spočte hodnota nové hešovací funkce a vloží se do nové hešovací tabulky. [16].

Protože velikost hešovací tabulky m je výrazně menší než $|U|$, tak ani s ideální hešovací funkcí není dosažitelné, aby se různé klíče vždy mapovaly na různé přihrádky a je proto nutné se



Obrázek 4.14 Řešení kolizí řetězením [17]

vypořádat se situací, kdy se dva a více klíčů namapují na stejnou přihrádku hešovací tabulky. Tato situace se nazývá *kolize* a druhou podstatnou částí hešovacího procesu je proto řešení kolizí. Mezi způsoby řešení kolizí patří *zřetězení* prvků (z angl. *chaining* nebo *separate chaining*) a *otevřená adresace* (z angl. *open addressing*). [17, 23]

Řešení kolizí pomocí zřetězení spočívá v tom, že do hešovací tabulky nejsou ukládány přímo klíče, ale spojové seznamy klíčů, tj. v každém z m indexů hešovací tabulky je uchováván spojový seznam klíčů, které se hešují na stejnou přihrádku. Každý index i hešovací tabulky tedy obsahuje ukazatel na začátek spojového seznamu prvků, které se hešovací funkcí h hešují na hodnotu i, nebo nulový ukazatel pokud žádné takové prvky nejsou. Základní myšlenkou této metody je zvolit hodnotu m tak, aby byla velikost hešovací tabulky dostatečně velká a délka spojových seznamů v přihrádkách tabulky dostatečně malá. [17, 23] Hešovací tabulku s řetězením ukazuje obrázek 4.14.

Při otevřeném adresování se na rozdíl od řetězení prvky vkládají přímo dovnitř hešovací tabulky. Každá položka tabulky tedy obsahuje buď prvek univerza U, nebo symbolickou hodnotu reprezentující prázdnou přihrádku. Při hledání či mazání prvku se systematicky procházejí přihrádky tabulky dokud není požadovaný prvek nalezen, nebo dokud není jisté, že hledaný prvek se v tabulce nenachází. Analogicky, při vkládání se systematicky procházejí přihrádky tabulky dokud není nalezena prázdná přihrádka, na kterou je prvek vložen. Na rozdíl od řetězení tedy může dojít ke kompletnímu zaplnění hešovací tabulky. [17]

Při vyhledávání, vkládání či mazání prvku je nejprve vypočítána posloupnost přihrádek, která se má prozkoumat. Důležité je, že tato posloupnost zkoumaných přihrádek není fixní pro všechny klíče, ale je vždy vypočítána v závislosti na konkrétním klíči k. Aby bylo možné určit, které přihrádky prozkoumat, je hešovací funkce rozšířena o druhý argument reprezentující číslo zkoumané přihrádky počínaje 0. V otevřeném adresování je proto hešovací funkce zobrazení $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$. V otevřeném adresování je vyžadováno, aby pro každý klíč k z univerza U byla posloupnost $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ permutací posloupnosti $0, 1, \ldots, m-1$. V ideálním případě má vyhledávací posloupnost každého klíče stejnou pravděpodobnost, že bude některou z $m!$ permutací posloupnosti $0, 1, \ldots, m-1$. Toto obdobně jako v případě ideální hešovací funkce obtížně dosažitelné, existují však vhodné aproximace, které mají dobré vlastnosti, jako je např. *lineární přidávání*, *kvadratické přidávání* či technika *dvojitého hešování*. [17] Více o těchto technikách a otevřeném adresování se lze dočíst např. v [17].

Více o hešovacích tabulkách, hešovacích funkcích, řešení kolizí a detailní analýze časové složitosti jejích operací se lze dočíst např. v [17] nebo [23].

Kapitola 5

Algoritmy slovníkových metod komprese dat

Tato kapitola se zabývá popisem a analýzou slovníkových kompresních metod LZ77, LZ78, LZW a LZSS. U každé metody jsou uvedeny její charakterizace, které byly popsány v sekci 3, a pseudokód kompresního i dekompresního algoritmu.

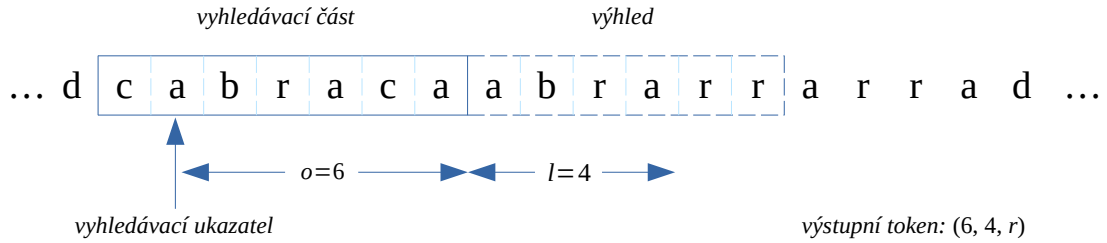
Většina slovníkových kompresních metod je založena na principech, které jsou popsány ve dvou přelomových pracích vědců Abrahama Lempela a Jacoba Ziva z let 1977 [24] a 1978 [25]. Tyto vědecké články prezentují dva různé přístupy k vytváření slovníků a každý z těchto přístupů dal vzniknout řadě dalších variant. O přístupech založených na článku z roku 1977 se říká, že patří do rodiny algoritmů LZ77 (v některé literatuře také LZ1), zatímco o přístupech založených na článku z roku 1978 se říká, že se řadí do rodiny algoritmů LZ78 (v některé literatuře také pod označením LZ2). [2]

LZ77 je historicky první algoritmus založený na slovníkové metodě komprese dat a jako efektivní alternativa Huffmanova kódování znovu nastartoval vědecký výzkum v oblasti komprese dat. [9]

5.1 Algoritmus LZ77

Slovníkový kompresní algoritmus LZ77 patří mezi bezztrátové, adaptivní, jedno-průchodové a asymetrické metody. Princip této metody spočívá v tom, že se jako slovník použije část dříve zpracované části vstupu. Kompresor si udržuje *klouzavé okénko* (z angl. *sliding window*) do proudu vstupních dat a při kódování řetězců posouvá vstup v okénku zprava doleva. Ekvivalentně lze tento přístup popsat tak, že klouzavé okénko je při běhu algoritmu posouváno po vstupním proudu dat zleva doprava. Okénko je rozděleno do dvou částí. Levá část okénka se nazývá *vyhledávací část* (z angl. *search buffer*), reprezentuje aktuální slovník a obsahuje symboly, které byly nedávno přečteny a zakódovány. Pravá část se nazývá *výhled* (z angl. *look-ahead buffer*) a obsahuje symboly, které teprve mají být zakódovány. V praktických implementacích mívá vyhledávací část několik tisíc bajtů, zatímco výhled bývá velký několik desítek bajtů. [1, 9]

Pro zakódování symbolů ve výhledu kodér prohledává vyhledávací část okénka pozpátku (zprava doleva) pomocí vyhledávacího indexu nebo ukazatele a hledá shodu s prvním symbolem ve výhledu. Vzdálenost vyhledávacího ukazatele od konce vyhledávací části se nazývá *offset*. Kodér poté zkoumá symboly následující za symbolem na místě vyhledávacího ukazatele, aby zjistil, jestli se budou shodovat s po sobě jdoucími symboly ve výhledu. Počet po sobě jdoucích symbolů ve vyhledávací části, které se shodují s po sobě jdoucími symboly ve výhledu se nazývá



Obrázek 5.1 Klouzavé okénko [2]

délka shody (z angl. *match length*). Kodér tímto způsobem hledá ve vyhledávací části nejdelší možnou shodu. Pokud existuje více shodných nálezů stejné délky, pak kodér vybere poslední nalezenou shodu. Po nalezení nejdelší shody kodér zapíše na výstup uspořádanou trojici (o, l, c), kde o je offset, l je délka shody a c reprezentuje symbol ve výhledu, který následuje za shodou. Tato uspořádaná trojice se nazývá *token*. Na závěr kodér posune klouzavé okénko po vstupním proudu dat doprava o $l+1$ pozic, l pozic za zakódování řetězce této délky a 1 pozici za zakódování symbolu c. Kodér tyto kroky opakuje tak dlouho, dokud výhled není prázdný. [1, 2]

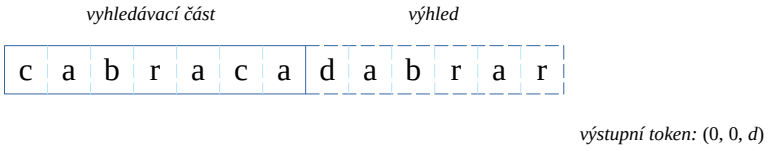
Příklad klouzavého okénka společně s vyznačenými hodnotami jednotlivých složek výstupního tokenu ilustruje obrázek 5.1. Pseudokód komprese LZ77 ukazuje algoritmus 1.

Vstup : posloupnost symbolů délky n
Výstup: posloupnost tokenů (o, l, c)
Naplň výhled symboly vstupu.
while *výhled není prázdný* **do**
  Najdi ve vyhledávací části nejdelší možnou shodu s řetězcem začínajícím ve výhledu.
  if *shoda existuje* **then**
    $o \leftarrow$ offset shody ve vyhledávací části okénka
    $l \leftarrow$ délka shody
    $c \leftarrow$ symbol ve výhledu následující za shodou
  else
    $o \leftarrow 0$
    $l \leftarrow 0$
    $c \leftarrow$ první symbol výhledu
  end
  Zapiš na výstup token (o, l, c).
  Posuň klouzavé okénko o právě $l + 1$ znaků.
end

Algoritmus 1: Komprese LZ77

Pro zakódování první složky tokenu, offsetu o, je potřeba $\lceil \log_2 |S| \rceil$ bitů, kde $|S|$ je velikost vyhledávací části klouzavého okénka. V praxi je velikost vyhledávací části několik nižších tisíc bajtů, takže velikost offsetu bývá 10 až 12 bitů. Analogicky, pro zakódování druhé složky tokenu, délky shody l, je potřeba $\lceil \log_2(|L| - 1) \rceil$ bitů, kde $|L|$ je velikost výhledu. Vzhledem k tomu, že token musí obsahovat platnou třetí složku, tedy symbol c, tak může být délka shody l nejvýše $|L| - 1$, aby bylo možné i v případě nalezení reálné shody délky $|L|$ zapsat platný symbol do třetí složky tokenu, což by v takovém případě byl poslední symbol výhledu. Takovou situaci zachycuje blíže příklad 5.1. V praktických implementacích je velikost výhledu několik nižších desítek bajtů, takže velikost druhé složky bývá pouze několik jednotek bitů. Na zakódování poslední složky tokenu, tedy symbolu c, je pak potřeba $\lceil \log_2 |\Sigma| \rceil$ bitů, kde $|\Sigma|$ značí velikost abecedy. V praxi je jako abeceda Σ volena množina všech možných hodnot jednoho bajtu, velikost třetí složky tokenu potom tedy je 8 bitů. [1]

Obrázek 5.2 Ukázka běhu komprese LZ77 [1]



Obrázek 5.3 LZ77: Žádná shoda [2]

Hlavním důvodem přítomnosti třetí složky tokenu je ošetření situace, kdy ve vyhledávací části nelze nalézt žádnou shodu s prvním symbolem ve výhledu. V tomto případě jsou hodnoty prvních dvou složek tokenu nastaveny na nulu a třetí složkou trojice je samotný první symbol výhledu. Tokeny s nulovým offset a nulovou délkou shody jsou běžné na začátku každé komprese, kdy vyhledávací část je prázdná nebo téměř prázdná. [1, 2]

Prvních sedm kroků komprese textu „*sir sid eastman easily teases sea sick seals*“ algoritmem LZ77 s klouzavým okénkem velikosti 40 symbolů, vyhledávací částí velikosti 24 symbolů a výhledem o 16 symbolech lze vidět na obrázku 5.2. Během procesu kódování algoritmem LZ77 mohou nastat tři základní situace [2]:

- Pro další znak výhledu, který má být zakódován, neexistuje žádná shoda.
- Existuje shoda délky l.
- Řetězec způsobující shodu přetéká z vyhledávací části do výhledu.

Všechny tyto možné situace detailně demonstruje následující příklad.

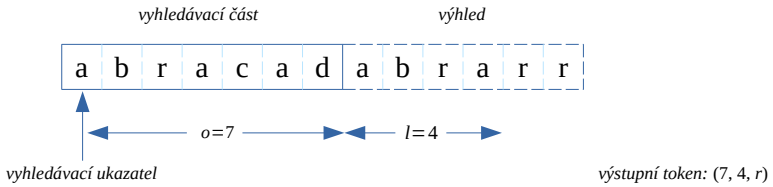
▶ **Příklad 5.1** ([2])**.** Nechť je velikost klouzavého okénka 13, délka vyhledávací části 7, délka výhledu 6 symbolů a řetězec, který má být zakódován, je

$$\ldots cabracadabrarrarrad\ldots$$

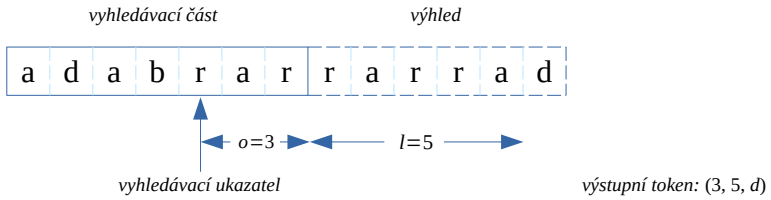
V aktuálním stavu klouzavého okénka je obsahem vyhledávací části řetězec *cabraca* a obsahem výhledu řetězec *dabrar*. Tento aktuální stav klouzavého okénka ukazuje obrázek 5.3.

Kodér prohledává pozpátku vyhledávací část okénka, aby našel shodu s prvním symbolem výhledu *d*. Je zřejmé, že žádná shoda neexistuje, takže kodér připraví výstupní token (0, 0, *d*) a pošle jej na výstup. První dvě složky tohoto tokenu odpovídají stavu, kdy nebyla pro symbol nalezena žádná shoda. Protože byl zakódován právě jeden symbol, je klouzavé okénko posunuto o jeden symbol doprava po vstupním proudu dat. Tento nový stav po posunu klouzavého okénka zachycuje obrázek 5.4.

Kodér opět začne pozpátku procházet vyhledávací část klouzavého okénka pro nalezení shody, tentokrát se symbolem *a*. První shodu nalezne na offsetu 2 a celková délka této shody je 1. Kodér



Obrázek 5.4 LZ77: Nalezena shoda [2]



Obrázek 5.5 LZ77: Shoda přetéká do výhledu [2]

pokračuje v prohledávání vyhledávací části a další shodu nalezne na offsetu 4. Délka této shoda je také 1. Kodér dále pokračuje ve vyhledávání a nalezne poslední shodu na offsetu 7. Protože délka této shody je 4, připraví kodér token (7, 4, r) a pošle jej na výstup. Protože bylo zakódováno celkem 5 symbolů, je klouzavé okénko posunuto o 5 symbolů doprava po toku vstupních dat. Tento aktuální stav klouzavého okénka demonstruje obrázek 5.5.

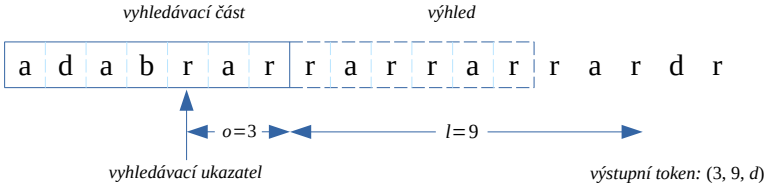
Kodér znovu začne prohledávat vyhledávací část okénka pro nalezení shody, tentokrát pro symbol r. První shodu nalezne na offsetu 1, délka shody je též 1. Další shodu nalezne na offsetu 3. Mohlo by se zdát, že délka této shody je 3. Kodér se však po porovnání symbolů *rar* na offsetech 3, 2 a 1 ve vyhledávací části nezastaví na konci vyhledávací části, ale plynule přeteče do výhledu a zde pokračuje v procesu porovnávání symbolů pro nalezení nejdelší možné shody. Kodér proto připraví výstupní token (3, 5, d) a zapíše jej na výstup. Zakódováno bylo celkem 6 symbolů, proto kodér posune klouzavé okénko o 6 symbolů doprava po proudu vstupních dat.

Dekodér je v případě algoritmu LZ77 výrazně jednodušší než kodér, proto se LZ77 řadí mezi asymetrické kompresní metody. Dekodér si udržuje vyrovnávací paměť o velikosti klouzavého okénka kodéru, tedy velikosti $|S|+|L|$. Dekodér přečte ze vstupu token, nalezne ve své vyrovnávací paměti řetězec R na offsetu o délky l, zapíše tento řetězec na výstup a poté na výstup zapíše symbol c. Na závěr dekodér vloží řetězec R společně se symbolem c do své vyrovnávací paměti, čímž zároveň z vyrovnávací paměti odstraní nejstarších $l+1$ symbolů. Tyto kroky dekodér opakuje tak dlouho, dokud jsou na vstupu přítomny tokeny. [1, 3] Pseudokód dekomprese LZ77 ukazuje algoritmus 2.

Vstup : posloupnost tokenů (o, l, c)
Výstup: dekomprimovaná data jako posloupnost symbolů délky n
1 **while** *vstup není prázdný* **do**
2 Přečti ze vstupu token (o, l, c).
3 Zapiš na výstup a na konec vyrovnávací paměti posloupnost l symbolů, která ve vyrovnávací paměti začíná na offsetu o.
4 Zapiš na výstup a na konec vyrovnávací paměti symbol c.
5 **end**

Algoritmus 2: Dekomprese LZ77

Na první pohled se může zdát, že tato metoda nevytváří žádné předpoklady o vstupních datech, resp. nevěnuje pozornost četnosti výskytu jednotlivých symbolů jako to dělají statistické



Obrázek 5.6 Modifikace LZ77: Výhled přetéká za klouzavé okénko [2]

metody. Nicméně z povahy klouzavého okénka plyne, že algoritmus vždy porovnává symboly ve výhledu s nedávno přečteným obsahem ve vyhledávací části okénka a nikdy se symboly, které byly přečteny dávno a už byly z vyhledávací části okénka odstraněny. Metoda proto implicitně předpokládá, že se shody ve vstupních datech vyskytují blízko u sebe. Data, která tento předpoklad splňují, pak budou tímto algoritmem dobře komprimovatelná. [1]

Kompresor potřebuje v každém kroku provést nejvýše S porovnání řetězců, což je konstantní vůči počtu vstupních symbolů. Operační složitost pro kompresi dat o n symbolech proto je $O(n)$. Analogicky, dekompresor stráví na dekódování každého tokenu konstantní čas vůči celkovému počtu tokenů. Složitost dekomprese komprimovaných dat o n tokenech proto je také $O(n)$. [26]

Autoři tohoto algoritmu ukázali, že efektivita algoritmu LZ77 se asymptoticky blíží tomu nejlepšímu, čeho lze dosáhnout pomocí metody, která má úplné informace o statistikách zdrojových dat. Protože jde pouze o asymptotický odhad, tak existuje několik způsobů jak algoritmus modifikovat, aby se zlepšily některé jeho vlastnosti. [2] Například je možné zvětšit velikost vyhledávací části klouzavého okénka i výhledu, což umožňuje najít lepší shody, ale kompromisem je nárůst doby nalezení shody a tím přípravy výstupního tokenu. Zvětšení klouzavého okénka proto vyžaduje použití sofistikovanější datové struktury, jako to dělá například algoritmus LZSS nebo algoritmus DEFLATE. Další možné vylepšení se týká klouzavého okénka. Nejjednodušší přístup je po každé nalezené shodě posunout celý obsah okénka doleva. Tento přístup je však silně neefektivní, mnohem rychlejší metoda je nahrazení lineárního okénka kruhovou frontou, která umožňuje realizovat posuv okénka v konstantním čase. Dalším možným vylepšením je použití jednobitového příznaku k rozlišení mezi tokeny a nekomprimovanými symboly, což je jedna z technik použitá v algoritmu LZSS. Dalším možným vylepšením je použití proměnlivé délky offsetu o a délky shody l v tokenu za pomocí Huffmanova kódování nebo nějaké jiné statistické metody. Tato technika je použita např. v algoritmu LZH. [1]

Zajímavým přístupem může být prodloužení maximální délky shody z původních $|L|-1$ symbolů až na $|S|+|L|$ symbolů, tedy délku celého klouzavého okénka. Tento postup je prezentován při popisu algoritmu LZ77 v [2] a odlišuje se tím od zbylé literatury jako [1] nebo [26]. Tento přístup spočívá v tom, že kodér se při hledání shody nezastavuje na konci výhledu, ale přetéká i na data, která se nacházejí za výhledem, tedy na data za klouzavým okénkem. Vyhledávací část sice i v tomto případě může přetéct do výhledu, ale na rozdíl od výhledu nemůže přetéct za hranice klouzavého okénka, proto je v takovém případě maximální délka shody $|S|+|L|$, které je dosaženo v případě, kdy je nalezena shoda, která začíná symbolem na levém konci vyhledávací části (na offsetu $|S|$) a končí posledním symbolem výhledu. Na druhou složku tokenu je v takovém případě potřeba $\lceil \log_2(|S|+|L|) \rceil$ bitů místo původních $\lceil \log_2(|L|-1) \rceil$. [2] Tuto modifikaci LZ77 přibližuje obrázek 5.6

Je zřejmé, že tento postup bude efektivní na datech, která obsahují shody velké délky, v praxi tedy několik tisíc bajtů. Na ostatních datech bude naopak tento postup velmi neefektivní, neboť pokud taková data běžně neobsahují shody délek několik tisíc bajtů, tak nebude využita celá datová šířka druhé složky tokenu o velikosti $\lceil \log_2(|S|+|L|) \rceil$ bitů, tedy bude ve výstupních tokenech zbytečně plýtváno místem a nebude proto dosaženo dobrého kompresního poměru. Možným řešením je tento přístup zkombinovat s nějakou statistickou metodou pro kódování složek tokenu, jako je např. Huffmanovo kódování, a tím snížit výstupní bitovou šířku druhé složky tokenu i pro data, ve kterých se často nevyskytují shody velkých délek.

5.2 Algoritmus LZ78

Algoritmus LZ77 předpokládá, že se shodné řetězce budou vyskytovat blízko u sebe. Této vlastnosti využívá tak, že jako slovník používá posloupnost nedávno zakódovaných symbolů. To však znamená, že žádný řetězec, který se opakuje po době delší, než je doba, na kterou se vztahuje klouzavé okénko, nebude zachycen. Nejhorším případem pak je situace, kdy se posloupnost symbolů ve vstupních datech opakuje s periodou delší než je délka vyhledávací části klouzavého okénka. [2]

Algoritmus LZ78 tento problém řeší tak, že nepoužívá vyhledávací paměť, výhled ani klouzavé okénko. LZ78 místo toho uchovává explicitní slovník dříve přečtených řetězců. Tento slovník je na začátku komprese i dekomprese prázdný, resp. téměř prázdný, a musí být postupně sestaven nezávisle v kodéru i dekodéru. Jeho velikost je reálně omezena pouze množstvím dostupné paměti a je nutné dbát na to, aby slovníky byly v kodéru a dekodéru stavěny stejným způsobem. [1, 2]

Výstupní tokeny komprese jsou v případě LZ78 uspořádané dvojice (i, c). První složka i reprezentuje ukazatel (někdy také index či referenci) na takový záznam slovníku, který má nejdelší shodu se symboly na vstupu, a c reprezentuje symbol na vstupu následující za nalezenou shodou. Tokeny neobsahují délku shody, protože ta je dána implicitně ze záznamů ve slovníku. Stejně jako v případě LZ77 se i zde pro hodnotu první složky tokenu i použije 0, pokud není nalezena žádná shoda. Každý token tedy kóduje nějaký řetězec R uložený ve slovníku následovaný symbolem c. Poté, co je sestavený token zapsán na výstup, je do slovníku jako nový záznam vložen řetězec R zřetězený se symbolem c. Každý nový záznam ve slovníku je tedy jeden nový symbol zřetězený s nějakým již existujícím záznamem ve slovníku. Ze slovníku se nikdy nic neodstraní, což je oproti algoritmu LZ77 jako výhoda, protože řetězce na vstupu mohou být porovnávány i s řetězci z velmi dávno zpracované části vstupu, tak i přítěž, neboť slovník má tendenci růst rychle a zaplnit tak celou dostupnou paměť. Možná řešení neomezeného růstu slovníku budou popsána později. [1, 2]

Do slovníku je na začátku komprese i dekomprese vložen na index 0 prázdný řetězec. Při čtení a kódování symbolů na vstupu se do slovníku postupně přidávají další záznamy na indexy 1, 2, atd. Při čtení dalšího symbolu x ze vstupu se ve slovníku hledá řetězec skládající se pouze ze symbolu x. Pokud ve slovíku takový řetězec neexistuje, přidá se řetězec skládající se ze symbolu x na další volnou pozici ve slovníku a na výstup se zapíše token $(0, x)$. Tento token reprezentuje zřetězení prázdného řetězce na indexu 0 se symbolem x. Pokud je ve slovníku nalezena shoda, např. na indexu 37, přečte se ze vstupu další symbol y a ve slovníku se následně hledá záznam odpovídající řetězci xy. Pokud žádný takový záznam neexistuje, je řetězec xy vložen na další volnou pozici do slovníku a na výstup je zapsán token $(37, y)$. Tento token reprezentuje řetězec xy, protože číslo 37 odpovídá indexu, na kterém je ve slovníku uložen řetězec x. Tento proces pokračuje tak dlouho, dokud jsou na vstupu kompresoru přítomna data. [1]

Obecně je vždy přečten jeden symbol ze vstupu a stane se z něj jednosymbolový řetězec. Kodér se jej pak pokusí najít ve slovníku. Pokud je symbol ve slovníku nalezen, přečte se další symbol a spojí se s prvním symbolem, čímž vznikne dvousymbolový řetězec, který se kodér pokusí najít ve slovníku. Dokud jsou tyto řetězce nalézány ve slovníku, jsou ze vstupu čteny další symboly a jsou spojovány do nového řetězce I. Jednou pak nastane situace, kdy takový řetězec I ve slovníku nebude nalezen, proto jej kodér přidá do slovníku a na výstup zapíše token, jehož první složkou je ukazatel na poslední shodu se slovníkem a druhou složkou je poslední symbol řetězce I, tedy takový symbol, který způsobil neshodu s řetězci ve slovníku. [1] Pseudokód komprese ukazuje algoritmus 3.

Komprese algoritmem LZ78 typicky začíná se slovníkem, který obsahuje krátké řetězce. Protože se poté v průběhu komprese zpracovává stále více vstupních symbolů, přidávají se do slovníku stále delší řetězce. Protože každý řetězec, který se má přidat do slovníku, je vždy pouze o jeden symbol delší, než nějaký již existující řetězec ve slovníku, je potřeba pro slovník zvolit vhodnou datovou strukturu. V [1] je jako vhodná datová struktura doporučována trie, která nabízí základní množinové operace s lineární časovou složitostí vůči délce řetězce. Více informací

Vstup : posloupnost symbolů délky n
Výstup: posloupnost tokenů (i, c)
$S \leftarrow$ prázdný slovník
Vlož do slovníku S prázdný řetězec.
while *vstup není prázdný* **do**
  $I \leftarrow$ prázdný řetězec
  $c \leftarrow$ další symbol vstupu
  while *řetězec* $I \cdot c$ *je nalezen ve slovníku* S **do**
    $I \leftarrow I \cdot c$
    $c \leftarrow$ další symbol vstupu
  end
  $i \leftarrow$ index řetězce I ve slovníku S.
  Zapiš na výstup token (i, c).
  Přidej řetězec $I \cdot c$ do slovníku S.
end

Algoritmus 3: Komprese LZ78

Slovník		Token	Slovník		Token
0					
1	s	(0, s)	8	a	(0, a)
2	i	(0, i)	9	st	(1, t)
3	r	(0, r)	10	m	(0, m)
4	␣	(0, ␣)	11	an	(8, n)
5	si	(1, i)	12	␣ea	(7, a)
6	d	(0, d)	13	sil	(5, l)
7	␣e	(4, e)	14	y	(0, y)

Obrázek 5.7 Ukázka běhu komprese LZ78 [1]

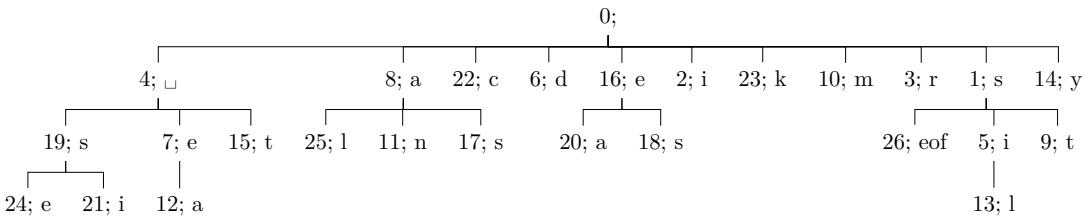
o této struktuře je napsáno v kapitole 4.4. Protože jsou do trie v případě LZ78 přidávány nové řetězce vždy právě po jednom symbolu, jsou implicitně označené všechny uzly trie, tj. každý uzel trie odpovídá nějakému platnému záznamu ve slovníku.

Prvních 14 kroků komprese textu „*sir sid eastman easily teases sea sick seals*“ ukazuje obrázek 5.7. Výsledná trie k tomuto příkladu je k vidění na obrázku 5.8. První složka uzlu na tomto obrázku reprezentuje index řetězce ve slovníku, druhá složka pak reprezentuje symbol vloženého řetězce. Kořen odpovídá prázdnému řetězci na indexu 0.

Dekodér LZ78 při dekompresi vytváří a udržuje slovník stejným způsobem jako kodér, proto je dekodér LZ78 složitější než dekodér algoritmu LZ77. Psedokód dekomprese LZ77 ukazuje algoritmus 4.

Z výše uvedeného popisu je zřejmé, že algoritmus LZ78 patří mezi bezztrátové, adaptivní, jedno-průchodové a symetrické metody.

Maximální velikost slovníku je fixní a nezávislá na velikosti vstupních dat. Algoritmus komprese přečte každý vstupní symbol c pouze jednou a při vhodné implementaci slovníku (např. pomocí trie) stráví kompresor ve slovníku nad vyhledáním řetězce $I \cdot c$ konstantní čas vůči velikosti slovníku i vůči délce hledaného řetězce. Jak je totiž patrné z pseudokódu 3, akumuluje kompresor vstupní symboly do řetězce I vždy právě po jednom symbolu. Pokud je pro slovník použita trie, pak je možné realizovat vyhledání řetězce $I \cdot c$ jako nalezení potomka uzlu u, který reprezentuje řetězec I, pro symbol c. Takové nalezení potomka je možné pak realizovat v konstantním čase vůči velikosti slovníku i vůči velikosti hledaného řetězce. Operační složitost komprese v tomto



Obrázek 5.8 Slovník LZ78 v podobě trie [1]

Vstup : posloupnost tokenů (i, c)
Výstup: dekomprimovaná data jako posloupnost symbolů délky n

```
1 S ← prázdný slovník
2 Vlož do slovníku S prázdný řetězec.
3 while vstup není prázdný do
4 |   Přečti ze vstupu token (i, c).
5 |   I ← řetězec ve slovníku S na indexu i
6 |   Zapiš na výstup řetězec I · c.
7 |   Přidej řetězec I · c do slovníku S.
8 end
```

Algoritmus 4: Dekomprese LZ78

případě proto je $O(n)$, kde n značí velikost vstupních dat. Protože LZ78 je symetrická metoda, platí i pro dekompresi, že její časová složitost je lineární vůči velikosti vstupu.

Algoritmus LZ78 je schopný zachytit ze vstupních dat řetězce a uchovávat je ve slovníku po neomezenou dobu. Velký slovník může obsahovat více řetězců a tím umožnit delší shody, ale kompromisem jsou pak větší ukazatele a tím větší bitová šířka výstupních tokenů, a také pomalejší vyhledávání ve slovníku. Zásadní problém pak je růst velikost slovníku, který teoreticky může neomezeně růst nade všechny meze. Jedním řešením je omezit velikost slovníku množstvím dostupné paměti při každém nezávislém spuštění kompresní programu. Problémem je, že dekodér nemusí mít k dispozici stejné množství dostupné paměti jako měl kodér, pokud je dekompresní program spouštěn na jiném zařízení. Dalším způsobem je fixní velikost slovníku, kdy je obsah slovníku při dosažení určité velikosti tzv. zmrazen. Zmrazený slovník se pak sice stává statickým, ale stále jej lze používat ke kódování dalších řetězců. Další možností je odstranění některých nejméně používaných položek při zaplnění slovníku, aby se uvolnilo místo pro nové záznamy. Bohužel však není znám žádný dobrý algoritmus, který by dokázal rozhodnout o tom, které položky a kolik jich odstranit. Další možností po zaplnění slovníku jeho celý obsah smazat a začít od začátku s novým, prázdným slovníkem. Toto řešení reálně rozdělí vstup na samostatné bloky, z nichž každý má svůj vlastní slovník. Tento přístup poskytuje dobré výsledky pouze v případě, kdy se obsah vstupu v jednotlivých blocích liší, protože se předpokládá, že obsah smazaného slovníku nebude pro další vstupní data potřeba. Obecně lze říci, že toto řešení stejně jako LZ77 předpokládá, že budoucí data na vstupu budou mít více shod s novějšími zpracovanými daty než se staršími. Poslední možností je poměrně komplexní řešení používané v unixovém programu *compress*. Ten po zaplnění slovníku dynamicky sleduje kompresní poměr a pokud klesne pod určitou mez, je slovník vyprázdněn a postupně budován znovu od začátku. [1, 2]

5.3 Algoritmus LZSS

LZSS je efektivní varianta algoritmu LZ77, kterou v roce 1982 ve vědeckém článku [27] představili James A. Storer a Thomas Szymanski. LZSS patří mezi bezztrátové, adaptivní, jedno-průchodové a asymetrické metody. Původní algoritmus LZ77 vylepšuje ve třech následujících směrech. [1]

- Udržuje výhled L v kruhové frontě.
- Udržuje vyhledávací část S klouzavého okénka v binárním vyhledávacím stromu.
- Výstupní tokeny mají dvě složky místo tří.

Protože v tomto případě jsou jako klíče do binárního vyhledávacího stromu ukládány řetězce, je potřeba zvolit způsob, kterým lze porovnat dva řetězce a rozhodnout, který z nich má být označen jako „menší“. Pro tyto účely se běžně používá lexikografické uspořádání, tj. řetězce se porovnávají lexikograficky podle dané abecedy. [1]

Kodér si během komprese udržuje v binárním vyhledávacím stromu T celkem $|T|$ řetězců, kde $|T| = |S| - |L| + 1$. Každý z těchto řetězců má délku $|L|$. Vyhledávací část klouzavého okénka je proto reálně rozdělena na právě $|T|$ řetězců délek $|L|$. Kodér se následně stejně jako LZ77 snaží najít v binárním vyhledávacím stromu nejdelší shodu se symboly ve výhledu. [1]

Výstupem kodéru jsou tokeny (o, l). Pokud je však bitová délka takového tokenu delší něž bitová délka shody, pak kodér zapíše na výstup nekomprimovaný první symbol shody, tj. první symbol ve výhledu. Pro rozlišení mezi tokeny a nekomprimovanými symboly je před každým z nich uveden jednobitový příznak, který určuje, zda následuje token nebo nekomprimovaný symbol. Poté je klouzavé okénko posunuto o k symbolů doprava, kde $k = l$, pokud byl na výstup zapsán token, nebo $k = 1$, pokud byl na výstup zapsán nekomprimovaný symbol. Na závěr je aktualizován obsah binárního vyhledávací stromu odstraněním k řetězců a vložením k nových řetězců. Řetězce, které mají být ze stromu odstraněny, odpovídají prvním k řetězcům ve vyhledávací části před posunem, a řetězce, které mají být do stromu přidány, odpovídají k posledním řetězcům ve vyhledávací části po posunu. [1, 3]

Možný přístup pro aktualizaci stromu je připravit si řetězec složený s prvních $|L|$ písmen ve vyhledávací části, najít jej společně s jeho offsetem ve stromu a následně odstranit. Poté posunout klouzavé okénko o jednu pozici doprava, připravit řetězec složený z posledních $|L|$ písmen ve vyhledávací části a přidat jej společně s jeho offsetem do stromu. Tento postup je potřeba provést celkem k-krát. [1] Pseudokód komprese LZSS ukazuje algoritmus 5.

Vstup : posloupnost symbolů délky n
Výstup: posloupnost tokenů (o, l) nebo nekomprimovaný symbol c

```
1  Naplň výhled symboly vstupu.
2  while výhled není prázdný do
3  |  Najdi ve vyhledávací části S nejdelší možnou shodu s řetězcem začínajícím ve
   |    výhledu L.
4  |  o ← offset shody ve vyhledávací části klouzavého okénka
5  |  l ← délka shody
6  |  if délka tokenu (o, l) ≤ délka shody then
7  |  |  Zapiš na výstup bitový příznak 0.
8  |  |  Zapiš na výstup token (o, l).
9  |  |  Posuň klouzavé okénko o právě l symbolů.
10 |  else
11 |  |  c ← první symbol výhledu
12 |  |  Zapiš na výstup bitový příznak 1.
13 |  |  Zapiš c na výstup. Posuň klouzavé okénko o 1 symbol.
14 |  end
15 end
```

Algoritmus 5: Komprese LZSS

▶ **Příklad 5.2** ([1])**.** Nechť velikost klouzavého okénka je 21 symbolů, velikost vyhledávací části je 16 symbolů a velikost výhledu je 5 symbolů. Kompresor má na vstupu řetězec „*sid eastman*

Obrázek 5.9 Počáteční stav okénka v příkladu 5.2

Řetězec	Offset
sid␣e	16
id␣ea	15
d␣eas	14
␣east	13
eastm	12
astma	11
stman	10
tman␣	9
man␣c	8
an␣cl	7
n␣clu	6
␣clum	5

Obrázek 5.10 Obsah binárního vyhledávacího stromu pro příklad 5.2

clumsily teases sea sick seals“. Obsah klouzavého okénka po zpracování prvních 21 symbolů ukazuje obrázek 5.9.

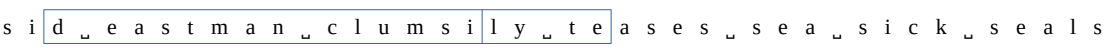
Kodér prohledá vyhledávací část klouzavého okénka a vytvoří 12 řetězců skládající se z 5 symbolů. Těchto 12 řetězců je následně vloženo do binárního vyhledávacího stromu, každý se svým offsetem. Obsah toto stromu ukazuje obrázek 5.10.

První symbol ve výhledu je symbol s, takže kodér hledá ve stromu řetězce, které začínají symbolem s. Ve stromu jsou nalezeny dva takové řetězce, na offsetech 16 a 10. Řetězec na offsetu 16 má se symboly ve výhledu delší shodu, konkrétně je délka shody 2. Na výstup je proto zapsán token (16, 2). Kodér nyní musí posunout klouzavé okénko o dvě pozice doprava a aktualizovat binární vyhledávací strom. Obsah aktualizovaného klouzavého okénka ukazuje obrázek 5.11. Strom by měl být aktualizován odstraněním řetězců *sid␣e* a *id␣ea*, a vložením dvou nových řetězců *clums* a *lumsi*.

Dekodér je stejně jako v případě algoritmu LZ77 jednodušší než kodér a proto se LZSS řadí mezi asymetrické kompresní metody. Dekodér nepotřebuje pro účely dekomprese udržovat nad vyhledávací částí klouzavého okénka binární vyhledávací strom a vystačí si s postupem analogickým s dekodérem LZ77. Pseudokód dekomprese LZSS ukazuje algoritmus 6.

Maximální délka shody je omezena konstantou $|L|$, počet řetězů v binárním vyhledávacím stromě je shora omezen konstantou $|T| = |S| - |L| + 1$, každý řetězec v tomto stromě pak má délku právě $|L|$. Kompresor proto v každém kroku vykoná konstantní počet operací vůči celkové délce vstupních dat. Operační složitost kompresoru proto je $O(n)$ vůči velikosti vstupu. Dekompresor obdobně jako LZ77 stráví dekompresí každého tokenu konstantní čas vůči celkovému počtu tokenů na svém vstupu. Časová složitost dekomprese je proto také $O(n)$ vůči velikosti vstupu.

V praktických implementacích má vyhledávací část klouzavého okénka několik tisíc bajtů, takže offset o zabírá typicky 11 až 13 bitů. Velikost výhledu by měla být volena tak, aby celková



Obrázek 5.11 Klouzavé okénko v příkladu 5.2 po provedení posuvu

Vstup : posloupnost tokenů (o, l) nebo nekomprimovaný symbol c
Výstup: dekomprimovaná data jako posloupnost symbolů délky n

```
1  while vstup není prázdný do
2  |  Přečti ze vstupu bitový příznak b.
3  |  if b = 0 then
4  |  |  Přečti ze vstupu token (o, l).
5  |  |  Zapiš na výstup a na konec vyrovnávací paměti posloupnost l symbolů, která ve
   |  |    vyrovnávací paměti začíná na offsetu o.
6  |  else
7  |  |  Přečti ze vstupu symbol c.
8  |  |  Zapiš na výstup a na konec vyrovnávací paměti symbol c.
9  |  end
10 end
```

Algoritmus 6: Dekomprese LZSS

velikost výstupního tokenu byla právě 16 bitů, tj. 2 bajty. Například pokud je velikost vyhledávací části okénka $2048 = 2^{11}$ bajtů, pak by výhled měl být velký $32 = 2^5$ bajtů. Bitová šířka offsetu o by takovém případě byla 11 bitů a bitová šířka délky shody l by byla 5 bitů. Při této volbě velikostí vyhledávací části a výhledu klouzavého okénka pak výstupem komprese jsou tokeny velikosti 2 bajtů, nebo nekomprimovaný symbol rozšířené ASCII tabulky délky 1 bajtu. Bitové příznaky pro rozlišení výstupního tokenu a nekomprimovaného symbolu mohou být na výstup zapisovány vždy před samotným tokenem či nekomprimovaným symbolem, nebo alternativně mohou být shromažďovány do malého bufferu o délce 8 bitů. V takovém případě je pak na výstup zapsán nejprve jeden bajt obsahující bitové příznaky a následně 8 výstupních položek, každá o délce 1 nebo 2 bajty. [1]

5.4 Algoritmus LZW

LZW jako variantu algoritmu LZ78 publikoval v článku [28] v roce 1984 Terry A. Welch. Tento algoritmus patří mezi bezztrátové, adaptivní, jedno-průchodové a symetrické kompresní metody. LZW využívá jednu z myšlenek vylepšení algoritmu LZSS oproti LZ77, a to je eliminace zapsání explicitního nekomprimovaného symbolu na výstup za každým zakódovaným řetězcem. LZW dokáže tyto nekomprimované symboly zcela eliminovat, takže jeho výstupní token nepotřebuje oproti algoritmu LZ78 druhou složku tokenu. Výstupní tokeny algoritmu LZW se tedy skládají z jediné složky a tou je reference do slovníku. [1, 26]

Eliminace druhé složky tokenu je dosaženo inicializací slovníku před zahájením komprese všemi symboly vstupní abecedy Σ. V praxi je tedy slovník inicializován všemi 256 možnými hodnotami jednoho bajtu, tj. všemi hodnotami rozšířené ASCII tabulky. Těchto 256 hodnot je zapsáno do slovníku ještě předtím, než se začnou zpracovávat vstupní data. Protože je slovník tímto způsobem inicializován, bude další symbol vstupní abecedy vždy ve slovníku nalezen. Proto může být token algoritmu LZW tvořen pouze ukazatelem do slovníku a nemusí obsahovat nekomprimovaný symbol, který způsobil neshodu s řetězci uloženými ve slovníku, jako je tomu v případě algoritmu LZ78. [1, 26]

Kompresor algoritmu LZW čte vstupní symboly jeden po druhém a shromažďuje je v řetězci I. Po přečtení symbolu c a jeho zřetězení s řetězcem I se ve slovníku hledá řetězec $I \cdot c$. Pokud je řetězec $I \cdot c$ ve slovníku nalezen, je I inicializován řetězcem $I \cdot c$ a kodér pokračuje přečtením dalšího symbolu. Tento proces pokračuje tak dlouho, dokud je řetězec $I \cdot c$ ve slovníku nalezen. V určitém okamžiku přidání dalšího symbolu c způsobí selhání hledání, tj. řetězec I je ve slovníku, ale řetězec $I \cdot c$ ve slovníku není. V tomto okamžiku kodér zapíše na výstup token (i), kde i reprezentuje index řetězce I ve slovníku, přidá řetězec $I \cdot c$ na další volnou pozici do slovníku

a inicializuje řetězec I symbolem c. Tento proces je opakován tak dlouho, dokud jsou na vstupu kodéru data. [1, 3] Pseudokód komprese ukazuje algoritmus 7.

```
   Vstup  : posloupnost symbolů délky n
   Výstup: posloupnost tokenů (i)
 1 S ← prázdný slovník
 2 Inicializuj slovník S všemi symboly vstupní abecedy Σ.
 3 I ← prázdný řetězec
 4 while vstup není prázdný do
 5 |   c ← další symbol vstupu
 6 |   if řetězec I · c je nalezen ve slovníku S then
 7 |   |   I ← I · c
 8 |   else
 9 |   |   i ← index řetězce I ve slovníku S
10 |   |   Zapiš na výstup token (i).
11 |   |   Přidej řetězec I · c do slovníku S.
12 |   |   I ← c
13 |   end
14 end
15 i ← index řetězce I ve slovníku S
16 Zapiš na výstup token (i).
```

Algoritmus 7: Komprese LZW

▶ **Příklad 5.3** ([1])**.** Kodér má na vstupu ke kompresi text „*sir sid eastman easily teases sea sick seals*“. Prvním krokem je inicializace slovníku na indexech 0 až 255 všemi symboly rozšířené ASCII tabulky.

Nyní kodér přečte ze vstupu symbol s, vyhledá jej ve slovníku a najde jej na indexu 115, který odpovídá kódu symbolu s rozšířené ASCII tabulky. Následně přečte ze vstupu symbol i, řetězec si však kodér nenalézá ve slovníku. Proto kodér zapíše na výstup index 115, vloží řetězec si do slovníku na další volnou pozici (v tomto případě pozice č. 256), a inicializuje akumulační řetězec I symbolem i.

V dalším kroku kodér přečte ze vstupu symbol r, řetězec ir však není uložený ve slovníku. Kodér proto na výstup zapíše index 105 (kód symbolu i v rozšířené ASCII tabulce), vloží řetězec ir do slovníku na další volnou pozici (v tomto případě pozice 257) a inicializuje akumulační řetězec I symbolem r.

Nyní kodér přečte ze vstupu symbol mezery $␣$, řetězec $r␣$ však není nalezený ve slovníku. Kodér proto zapíše na výstup index 114 (kód symbolu r v rozšířené ASCII tabulce), vloží řetězec $r␣$ do slovníku na další dostupnou pozici (v tomto případě 258) a inicializuje akumulační řetězec I symbolem $␣$.

Dále kodér přečte ze vstupu symbol s, řetězec $␣s$ však není nalezený ve slovníku. Kodér proto zapíše na výstup index 32 (kód symbolu $␣$ v rozšířené ASCII tabulce), vloží řetězec $␣s$ do slovníku na další dostupnou pozici (v tomto případě na index 259) a inicializuje akumulační řetězec I symbolem s.

Nyní kodér přečte ze vstupu symbol i a řetězec si je nalezen na slovníku na indexu 256. Kodér proto pokračuje čtením dalšího symbolu ze vstupu, tentokrát je to symbol d. Řetězec sid však není nalezený ve slovníku, kodér proto na výstup zapíše index 256 (ukazatel do slovníku na řetězec si), vloží řetězec sid do slovníku na další volnou pozici (v tomto případě 260) a inicializuje akumulační řetězec I symbolem d.

Tímto způsobem kodér pokračuje tak dlouho, dokud má na vstupu data. Výsledný slovník po komprimaci celého vstupního textu „*sir sid eastman easily teases sea sick seals*“ je k vidění na obrázku 5.12.

0	NULL	110	n	262	␣e	276	te
1	SOH	…		263	ea	277	eas
…		115	s	264	as	278	se
32	SP	116	t	265	st	279	es
…		…		266	tm	280	s␣
97	a	121	y	267	ma	281	␣se
98	b	…		268	an	282	ea␣
99	c	255	NBSP	269	n␣	283	␣si
100	d	256	si	270	␣ea	284	ic
101	e	257	ir	271	asi	285	ck
…		258	r␣	272	il	286	k␣
107	k	259	␣s	273	ly	287	␣sea
108	l	260	sid	274	y␣	288	al
109	m	261	d␣	275	␣t	289	ls

Obrázek 5.12 Výsledný LZW slovník pro příklad 5.3 [1]

Dekodér začíná stejně jako kodér inicializací svého slovníku všemi symboly abecedy Σ. Poté čte ze svého vstupu tokeny (i), kde i reprezentuje ukazatel do slovníku, a zapisuje na výstup řetězce nacházející se v jeho slovníku na indexu i. Z toho plyne, že dekodér algoritmu LZW musí stejně jako dekodér algoritmu LZ78 postupně budovat svůj slovník a to identickým způsobem, jakým byl slovník budován v kodéru při kompresi (tato skutečnost se někdy vyjadřuje slovy, že kodér a dekodér jsou tzv. *synchronizovány*). [1]

Na začátku dekomprese přečte dekodér ze vstupu první token a vyhledá odpovídající řetězec ve svém slovníku. Díky inicializaci slovníku před zahájením dekomprese je zaručené, že toto počáteční vyhledávání ve slovníku neselže. Nalezený řetězec dekodér zapíše na výstup a současně tímto řetězcem inicializuje řetězce I a J.

Nyní dekodér přečte ze vstupu token (i) a pokusí se ve slovníku vyhledat řetězec na indexu i. Pokud je řetězec ve slovníku nalezen, pak je jím inicializován řetězec J, který je následně zapsán na výstup. Do slovníku je poté uložen řetězec I zřetězený s prvním symbolem řetězce J. Na závěr je řetězec I inicializován řetězcem J. Kodér pak dále pokračuje přečtením dalšího tokenu ze vstupu. [1, 3]

Pokud však hledání řetězce ve slovníku na indexu i vstupního tokenu (i) selže, pak je řetězec J inicializován řetězcem I zřetězeným s prvním symbolem řetězce I. Tento nový řetězec J je zapsán na výstup a současně uložen do slovníku na další volnou pozici. Na závěr je opět řetězec I inicializován řetězcem J a kodér dál pokračuje přečtením dalšího tokenu ze svého vstupu. [3] Pseudokód dekomprese ukazuje algoritmus 8. Z předchozího popisu a pseudokódu dekompresoru je zřejmé, že ačkoliv dekompresor buduje slovník stejným způsobem jako kompresor, je budování slovníku dekompresoru o jeden krok zpožděné.

Protože prvních 256 položek slovníku se obsaženo hned od začátku komprese, musí být položka i výstupního tokenu delší než 8 bitů. Obvyklé implementace typicky používají 16-bitové indexy, které umožňují použít slovník o $2^{16} = 65536$ položkách. Takový slovník se kromě malých kompresních úloh poměrně rychle zaplní. Nastává tedy stejný problém se zaplněním slovníku jako u algoritmu LZ78 a všechna možná řešení použitá u LZ78, která jsou popsána v kapitole 5.2, lze použít i u LZW. Další zajímavou skutečností u algoritmu LZW je, že se řetězce ve slovníku prodlužují vždy pouze o jeden symbol. Může tedy trvat poměrně dlouho, než se do slovníku začnou ukládat dlouhé řetězce a lze proto říct, že se algoritmus LZW pomalu přizpůsobuje vstupním datům. [1]

Vstup : posloupnost tokenů (i)
Výstup: dekomprimovaná data jako posloupnost symbolů délky n

```
1  S ← prázdný slovník
2  Inicializuj slovník S všemi symboly vstupní abecedy Σ.
3  Přečti ze vstupu token (i).
4  J ← řetězec ve slovníku S na indexu i
5  Zapiš na výstup řetězec J.
6  I ← J
7  while vstup není prázdný do
8  |  Přečti ze vstupu token (i).
9  |  if ve slovníku S existuje řetězec na indexu i then
10 |  |  J ← řetězec ve slovníku S na indexu i
11 |  else
12 |  |  J ← I · I[0]
13 |  end
14 |  Zapiš na výstup řetězec J.
15 |  Přidej řetězec I · J[0] do slovníku S.
16 |  I ← J
17 end
```

Algoritmus 8: Dekomprese LZW

Z popisu algoritmu je zřejmé, že vzhledem k tomu, že se řetězce ve slovníku prodlužují vždy jen o jeden symbol, tak každý řetězec, který je uložený do slovníku, ve skutečnosti přidává pouze jeden symbol. To ekvivalentně znamená, že pro každý řetězec, který je uloženy ve slovníku a je delší než jeden symbol, existuje ve slovníku další řetězec, který je pouze o jeden symbol kratší. Z tohoto důvodu je stejně jako u algoritmu LZ78 nejvhodnější volbou pro slovník algoritmu LZW trie. [1]

V [1] je jako datová struktura pro slovník LZW popsána modifikace trie, která k ukládání uzlů trie využívá hešovací tabulku s otevřeným adresováním. Každý uzel takové trie je ukládán do pole hešovací tabulky. Uzel trie je strukturou o dvou položkách, tou první je ukazatel na rodičovský uzel a druhou položkou je aktuální symbol řetězce. Pohyb prefixovým stromem směrem dolů, od uzlu k některému z jeho potomků, je pak realizován za pomoci hešování. Hešovací funkce je v tomto případě funkce o dvou proměnných a dostává na vstup ukazatel na aktuální uzel a symbol potomka. Výstupem hešovací funkce je pak ukazatel na tohoto potomka.

Protože LZW může využívat stejnou datovou strukturu pro slovník jako LZ78 a jeho vnitřní logika akumulace a vyhledávání řetězců ve slovníku je stejná jako u LZ78, je operační složitost kompresoru i dekompresoru algoritmu LZW také lineární vůči velikosti vstupních dat.

Publikování algoritmu LZW silně ovlivnilo komunitu zabývající se kompresí dat a motivovalo mnoho dalších, kteří přišli s modifikacemi a variantami této metody. LZMW místo aktuálního řetězce I a jednoho následujícího vstupního symbolu přidává řetězec I zřetězený s celým řetězcem z následujícího kompresního kroku. Algoritmus LZAP je další variantou založenou na této myšlence. Namísto zřetězení posledních dvou řetězců a uložení výsledku do slovníku jsou do slovníku vkládány všechny prefixy tohoto zřetězení. Pokud jsou tedy I a J dvě poslední shody, pak LZAP do slovníku přidává řetězec $S \cdot j$ pro každý neprázdný prefix j řetězce J, včetně samotného řetězce J. Metoda LZY pak do slovníku přidává jeden řetězec na každý vstupní symbol a zvětšuje řetězce vždy o jeden symbol. [1]

Kapitola 6
Implementace

Tato kapitola se zabývá popisem implementace slovníkových kompresních metod LZ77, LZ78, LZW a LZSS. Nejprve je představen obecný návrh aplikace včetně logických celků, které jsou pro všechny algoritmy společné, v podkapitolách jsou pak vylíčeny detaily implementace dílčích algoritmů a potřebných datových struktur. U každého algoritmu jsou také popsány problémy, které implementace daného kompresního algoritmu představuje, a zvolená řešení těchto problémů.

Každý z kompresních algoritmů je implementovaný jako samostatná konzolová aplikace. Projekt byl implementován v programovacím jazyce C++ a byl zvolen objektově orientovaný návrh s důrazem na rozšiřitelnost, znovupoužitelnost a udržitelnost.

Zdrojové kódy projektu jsou rozděleny na prezentační vrstvu a aplikační vrstvu. V prezentační vrstvě se nacházejí třídy pro *parsování* (syntaktickou analýzu) argumentů dílčích kompresních algoritmů. Tyto třídy zároveň kontrolují, jestli jsou uživatelem zadané argumenty validní. Pro parsování argumentů byla využita knihovna *argparse*. [29]

Algoritmy LZ77 a LZSS spadají do algoritmů typu LZ1 využívající pro kompresi klouzavé okénko a sdílejí proto argumenty pro zvolení velikosti vyhledávací části klouzavého okénka i výhledu. Sdílení společných argumentů bylo hlavní motivací pro navržení samostatné třídy *LZ1ArgumentParser*, která se stará o parsování argumentů pro kompresní algoritmy typu LZ1. Třídy pro parsování argumentů pro LZ77 a LZSS jsou potomci třídy *LZ1ArgumentParser* a mohou v případě potřeby rozšíření přidávat své vlastní specifické argumenty.

Analogicky byla navržena třída *LZ2ArgumentParser*, která se stará o parsování argumentů pro algoritmy typu LZ2, pro které lze nastavit velikost slovníku, strategii zaplněného slovníku a mezní kompresní poměr rozhodující o vyprázdnění slovníku, pokud byla zvolena odpovídající strategie. Potomci *LZ2ArgumentParser* jsou třídy pro parsování argumentů algoritmů LZ78 a LZW. Přínos tohoto hierarchického objektově orientovaného návrhu je patrný např. při nastavení minimální požadované velikosti slovníku. LZ78 začíná se slovníkem obsahující pouze prázdný řetězec, LZW na rozdíl od LZ78 začíná se slovníkem o 256 položkách. Stačí proto v konstruktoru třídy *LZ78ArgumentParser* či *LZWArgumentParser* pouze předat rodičovské třídě jinou hodnotu vyžadované minimální velikosti slovníku a dále není potřeba přidávat žádný další kód, o veškerou zbylou logiku se totiž postará právě rodičovská třída *LZ2ArgumentParser*.

O parsování argumentů, které jsou společné pro všechny algoritmy, jako je například zdroj vstupních a výstupních dat nebo akce určující spuštění komprese či dekomprese, se stará třída *ArgumentParser*, která je rodičovskou třídou obou tříd *LZ1ArgumentParser* a *LZ2ArgumentParser*.

Pokud argumenty úspěšně prošly procesem parsování, jsou před předáním fasádě zaobaleny do samostatné struktury. Tyto struktury s argumenty pro konkrétní algoritmus mají podobný hierarchický návrh jako třídy pro parsování argumentů a díky tomu je nesmírně jednoduché

rozšířit množinu argumentů pro konkrétní algoritmus. Tento návrh také dále díky polymorfismu umožňuje přenechat zodpovědnost inicializace proměnných dané struktury odpovídajícím třídám pro parsing argumentů. Např. pro strukturu *LZ77Setup* proměnné odpovídající argumentům, které jsou společné pro všechny kompresní algoritmy, inicializuje třída *ArgumentParser*. Analogicky, proměnné odpovídající argumentům, jež jsou společné pro algoritmy LZ1, inicializuje třída *LZ1ArgumentParser*. A v případě potřeby vytvoření nového argumentu specifického pro algoritmus LZ77 pak pouze stačí vytvořit odpovídající proměnnou ve struktuře *LZ77Setup* a inicializovat ji ve třídě *LZ77ArgumentParser*. Rozšíření konkrétního algoritmu o další argument má proto díky zvolenému návrhu zcela minimální dopad na okolní zdrojový kód.

Vzhledem k rozdělení aplikace do samostatných vrstev slouží fasáda prezentační vrstvě jako vstupní brána do aplikační vrstvy. Fasáda na základě předaných argumentů inicializuje binární či čitelný kompresor, resp. dekompresor, odpovídajícími argumenty a spustí proces komprese, resp. dekomprese, pro zvolený vstup a výstup. Celý tento proces je zapouzdřen do metody *run*, prezentační vrstvě proto stačí zavolat na fasádě pouze tuto metodu a tím je od popsaného procesu zcela odstíněna. Díky tomuto návrhu a jednoduššímu rozhraní fasády je minimalizována závislost prezentační vrstvy na aplikační vrstvě, takže je možné v případě potřeby prezentační vrstvu snadno modifikovat či dokonce zcela nahradit jinou implementací.

6.1 Komprese a dekomprese

Pro všechny kompresory bylo navrženo jednotné rozhraní reprezentované abstraktní třídou *Compressor*. Ta obsahuje virtuální metodu *compress*, která má obstarávat logiku čtení dat ze vstupního proudu, jejich odpovídající kompresi podle implementovaného algoritmu a zápis komprimovaných dat do výstupního proudu. Tuto metodu proto musí implementovat každá třída, která realizuje konkrétní kompresní algoritmus.

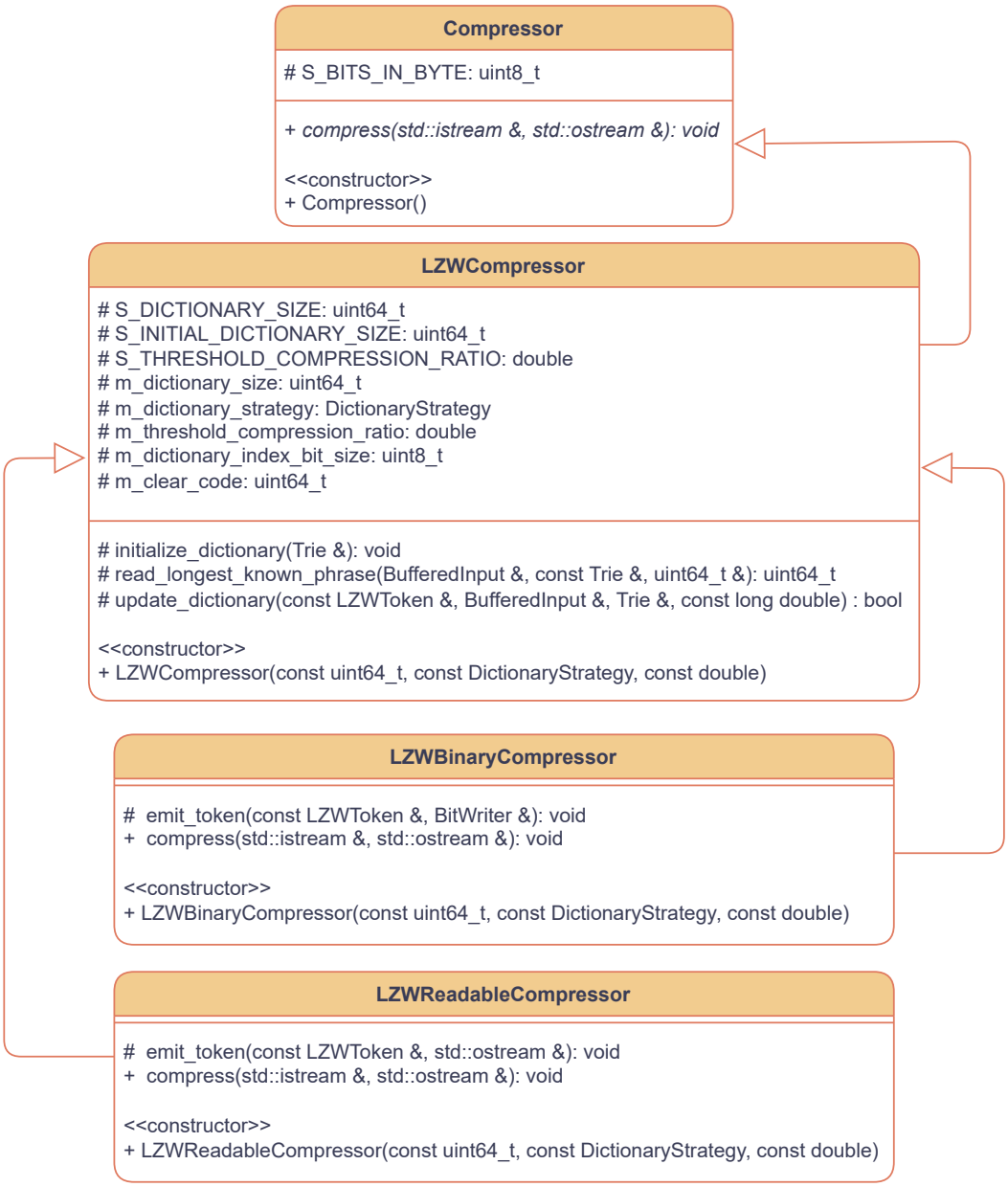
Každý algoritmus byl implementovaný ve dvou režimech: binárním a čitelném. Binární kompresor posílá na výstup tokeny v binárním formátu, čitelný kompresor zapisuje tokeny na výstup ve formátované podobě čitelné pro člověka. Analogicky binární dekompresor očekává na vstupu tokeny v binárním zápisu, zatímco čitelný dekompresor čte tokeny ze vstupu ve formátovaném režimu. Binární kompresor, resp. dekompresor, je reprezentovaný samostatnou třídou a stejné je tomu u čitelného kompresoru, resp. dekompresoru.

Protože binární i čitelný kompresor mají značnou část své vnitřní logiky společnou, je tato sdílená logika rozdělena do samostatných metod, které jsou extrahovány do společné rodičovské třídy. Např. společná logika binárního a čitelného kompresoru LZW je implementována ve třídě *LZWCompressor*, jejíž potomci jsou třídy *LZWBinaryCompressor* a *LZWReadableCompressor*, které pak implementují jiný způsob zápisu výstupních tokenů. Třída *LZWCompressor* je zároveň potomkem abstraktní třídy *Compressor* a slouží proto jako spojovací článek mezi velmi obecnou abstraktní třídou *Compressor* a konkrétními třídami *LZWBinaryCompressor* a *LZWReadableCompressor*. Analogicky jsou navrženy i třídy pro dekompresory jednotlivých kompresních algoritmů.

Tento hierarchický návrh demonstruje na příkladu kompresorů algoritmu LZW UML diagram tříd na obrázku 6.1. Diagramy ostatních tříd je možné nalézt v dokumentaci projektu vygenerované nástrojem Doxygen.

Díky výše uvedenému návrhu je možné využít polymorfismus k jednotnému volání metod pro kompresi a dekompresi v jiných modulech projektu. Např. fasáda implementuje metodu pro spuštění procesu komprese, resp. dekomprese a přijímá jako argument referenci na abstraktní třídu *Compressor*, resp. *Decompressor*. Díky tomu je možné použít tuto metodu fasády jednotně pro všechny algoritmy a to v obou režimech, binárním i čitelném.

Podobné je to u třídy *CompressionTester*, která testuje korektnost dat po kompresi a následné dekompresi. Tato třída očekává jako parametry svých metod reference na abstraktní třídy *Compressor* a *Decompressor*. Díky tomu je možné používat třídu *CompressionTester* pro testování



Obrázek 6.1 UML diagram tříd pro kompresory LZW

Výpis kódu 6.1 BitWriter

```
void write_bit(const uint64_t bit);
void write_bits(const uint64_t bits, const uint8_t count);
```

všech kompresorů a dekompresorů v binárním i čitelném režimu. Více o třídě *CompressionTester* je možné se dočíst v kapitole 7 o testování.

Aby proces měření kompresního poměru nebyl zkreslený dalšími informacemi, nejsou parametry komprese zapisovány na výstup společně s komprimovanými daty. Dekompresor je proto nutné spustit se stejnými parametry, s jakými byla data zkomprimována.

6.2 Vstup a výstup

Pro maximální využití cache paměti procesoru je načítání vstupních dat realizováno pomocí vyrovnávací paměti. Postupné načítání dat ze vstupu po blocích, jejich uložení do vnitřní vyrovnávací paměti a vystavení rozhraní pro postupné čtení dat z této vyrovnávací paměti realizuje třída *BufferedInput*. Velikost vyrovnávací paměti, do které se načítají vstupní data, je parametrem konstruktoru této třídy.

Analogicky byla navržena třída *BufferedOutput*, která data, jež mají být zapsána na výstup, nejprve ukládá do své vnitřní vyrovnávací paměti a po jejím zaplnění zapíše celý blok dat na výstup.

Kompresory pracující v binárním režimu potřebují zapisovat na výstup binární data v proměnlivé bitové šířce. Pro tyto účely byla navržena třída *BitWriter*, která je inicializována referencí na výstupní proud *std::ostream*. Tato třída umožňuje na výstup zapisovat jednotlivé bity nebo zapsat až 64 bitů dat, které jsou uložené v proměnné typu *uint64_t*. Pokud je počet bitů *count* určených k zápisů menší 64, pak metoda *write_bits* zapíše na výstup požadovaný počet bitů na nejméně významných pozicích proměnné *bits*. Toto rozhraní ukazuje kód 6.1.

Třída *BitWriter* akumuluje bity určené k zápisu na výstup ve svém vnitřním bufferu o velikosti jednoho bajtu a teprve po zaplnění tohoto bajtu posílá jeho obsah na výstup. Pokud celkový počet bitů určený k zápisu na výstup není celočíselným násobkem jednoho bajtu, jsou poslední bity doplněny zprava nulami na velikost jednoho bajtu, který je při volání destruktoru třídy zapsán na výstup.

Z důvodu efektivity nezapisuje *BitWriter* data na výstup po jednotlivých bajtech, ale využívá vyrovnávací paměť prostřednictvím třídy *BufferedOutput*.

Analogicky byla navržena i třída *BitReader* pro čtení požadovaného počtu bitů ze vstupu.

6.3 Datové struktury

6.3.1 Kruhová fronta

Kruhová fronta byla implementována pomocí dynamicky alokovaného pole fixní velikosti. Velikost vnitřního pole použitého pro ukládání prvků fronty lze nastavit ručně pomocí argumentu konstruktoru této třídy. Implementace dále z důvodu znovupoužitelnosti používá šablony, aby bylo možné do kruhové fronty ukládat obecný datový typ T.

Implementovány byly dvě standardní operace pro kruhovou frontu: *push* pro vložení prvku na konec fronty a *pop* pro odstranění prvku, který se nachází na začátku fronty. V rámci této práce je kruhová fronta použita jako datová struktura pro klouzavé okénko, jak bylo popsáno v kapitole 5.1. Kromě operací pro vkládání a odstraňování prvků bylo proto nutné implementovat i operátor náhodného přístupu pro čtení prvků uvnitř kruhové fronty pomocí indexů, aby bylo možné číst obsah klouzavého okénka.

Operátor náhodného přístupu byl implementovaný ve dvou verzích. První verze operátoru náhodného přístupu provádí kontrolu mezí (z angl. *bounds checking*) a pro návratovou hodnotu využívá třídu *std::optional*. Pokud tato verze operátoru náhodného přístupu obdrží požadavek na přístup k prvku na indexu, který se nachází za hranicí aktuální velikosti fronty, je návratovou hodnotou takového volání objekt *std::nullopt*. Pokud je index v mezích aktuální velikosti fronty, pak je kopie prvku nacházející se na tomto indexu zaobalena do instance *std::optional*, která je pak návratovou hodnotou volání této metody. [30]

Kontrola mezí a kopírování prvku fronty do instance třídy *std::optional* má však určitou režii a proto byl operátor náhodného přístupu implementován ve druhé verzi. Ta neprovádí kontrolu mezí a vždy vrací referenci na prvek uvnitř fronty na požadovaném indexu. Je zřejmé, že tato verze operátoru náhodného přístupu je díky nižší režii rychlejší, ale pokud se požadovaný index nachází za hranicí velikosti fronty, pak návratová hodnota samozřejmě odkazuje na neplatná data. Proto je při používání této verze operátoru náhodného přístupu na místě určitá obezřetnost, aby volající nepřistupoval na neplatná data v kruhové frontě.

6.3.2 Trie

Trie byla implementována jako posloupnost uzlů, které jsou ukládány do kontejneru *std::vector*. Protože je trie využívána jako struktura pro uložení množiny řetězců, bylo nutné implementovat logiku překladu indexů na vnitřní uzly trie a naopak.

Pro překlad indexů na vnitřní uzly trie byl použit kontejner *std::unordered_map*, který je implementovaný jako hešovací tabulka s řetězením [18]. Díky tomuto řešení má překlad indexu na číslo vnitřního uzlu trie amortizovaně konstantní složitost.

Překlad vnitřního uzlu trie na index má jednodušší řešení: uvnitř struktury reprezentující uzel trie je proměnná *dictionary_index* typu *std::optional<uint64_t>*. Pokud tento uzel reprezentuje řetězec vložený do trie, pak je odpovídající index uložený právě v proměnné *dictionary_index*. Pokud uzel netvoří vložený řetězec, obsahuje tato proměnná *std::nullopt*.

Aby byla datová struktura znovupoužitelná i v jiných problémových doménách, byly implementovány metody pro vložení celého řetězce a vyhledání celého řetězce. Obě tyto operace mají lineární operační složitost vůči délce řetězce. Nicméně vzhledem k tomu, že obě tyto operace jsou používány v algoritmech LZ78 a LZW velice často, je lineární operační složitost nežádoucí a tyto operace by proto zbytečně zpomalovaly kompresor i dekompresor. Navíc kompresor i dekompresor přidávají a vyhledávají řetězce ve struktuře vždy po jednom symbolu, jak již bylo zmíněno v kapitolách 5.2 a 5.4, a bylo by proto vhodné této vlastnosti využít.

Z těchto důvodů byly implementovány metody *append_to* a *find_from*. Metoda *append_to* vloží nový řetězec $I \cdot c$ tak, že pouze připojí za uzel reprezentující řetězec I uzel pro symbol c. Je zřejmé, že tato metoda očekává jako svůj argument kromě symbolu c také index, který odpovídá dříve vloženému řetězci I. Analogicky byla implementována i metoda *find_from*, která vyhledává řetězec $I \cdot c$. Obě tyto metody mají konstantní operační složitost vůči délce vkládaného či vyhledávaného řetězce.

6.3.3 Hešovací tabulka s otevřeným adresováním

Hešovací tabulka s otevřeným adresováním je implementována ve třídě *FlatHashTable*. Stejně jako v případě kruhové fronty byly i pro implementaci hešovací tabulky s otevřeným adresováním použity šablony z důvodu znovupoužitelnosti, aby bylo možné strukturu použít s generickým datovým typem T. Pro generování posloupnosti navštěvovaných přihrádek byla implementována technika lineárního přidávání.

Kromě operací pro vložení, vyhledání a smazání prvku byl implementován i operátor náhodného přístupu. Obdobně jako u kruhové fronty byl i v případě hešovací tabulky s otevřeným adresováním implementován operátor náhodného přístupu k prvkům v tabulce ve dvou verzích.

Jeden z těchto operátorů provádí kontrolu mezí a pokud je požadovaná přihrádka v mezích kapacity tabulky a současně obsahuje tato přihrádka platná data, pak je prvek v této přihrádce zkopírován do instance *std::optional*, která je pak návratovou hodnotou tohoto operátoru. Pokud přihrádka není v mezích kapacity tabulky nebo přihrádka neobsahuje platná data, pak je návratovou hodnotou *std::nullopt*.

Druhý operátor náhodného přístupu stejně jako v případě kruhové fronty neprovádí žádné kontroly a rovnou vrací referenci na data v požadované přihrádce.

Z teorie popsané v kapitole 4.6 je zřejmé, že pro efektivní práci s hešovací tabulkou je kritická kvalitní hešovací funkce. Z toho důvodu je jedním z parametrů konstruktoru této třídy i funkce, která se má použít pro hešování. Pokud volající nedodá vlastní hešovací funkci, pak je implicitně použita funkce *std::hash<T>* ze standardní knihovny jazyka C++. V takovém případě se samozřejmě předpokládá, že ve standardní knihovně již existuje funkce *std::hash* pro datový typ T, nebo je vytvořena specializace pro datový typ T.

6.4 Implementace LZ77

6.4.1 Kompresor

Všechny kritické operace pro kompresi algoritmem LZ77, jako je nalezení shody co nejdelšího prefixu ve výhledu s obsahem vyhledávací části klouzavého okénka a posun klouzavého okénka po vstupních datech, jsou delegovány na samostatnou třídu *LZ77SlidingWindow*.

Tento návrh přináší dvě zásadní výhody. První výhodou je, že tato třída je zcela abstrahována od konkrétního způsobu zápisu výstupních tokenů a lze ji proto využít jak pro binární tak pro čitelný kompresor. Po znovupoužitelnosti je další výhodou slabá provázanost: veškerá logika pro nalezení shody a posun klouzavého okénka je zapouzdřená uvnitř třídy *LZ77SlidingWindow*. Tento návrh umožňuje snadno implementovat jinou, např. efektivnější, logiku pro nalezení shody. Stačí totiž ve třídě pro binární či čitelný kompresor pouze vyměnit instanci třídy *LZ77SlidingWindow* za novou implementaci s novým algoritmem pro nalezení shody, a dále již není potřeba měnit žádný další kód. Tato možnost byla v rámci této bakalářské práce skutečně využita, jak je popsáno níže.

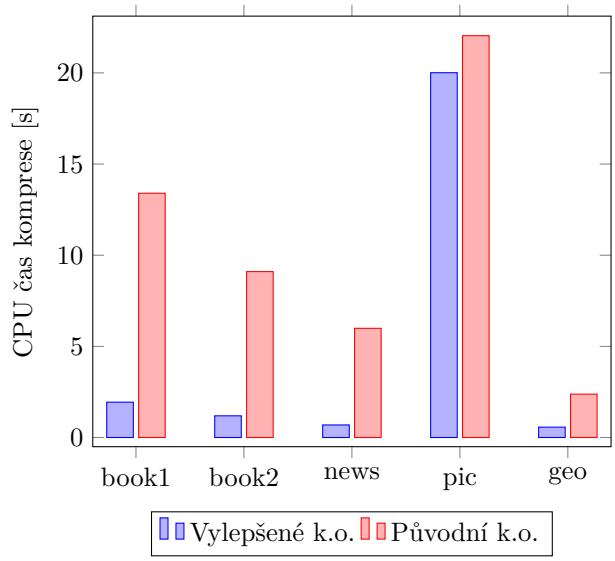
Třída *LZ77SlidingWindow* používá pro vyhledávací část klouzavého okénka i pro výhled kruhovou frontu pro efektivní vkládání i odstraňování prvků z obou částí klouzavého okénka.

Původní algoritmus popsaný v kapitole 5.1 pro nalezení shody je silně neefektivní. Algoritmus postupně prochází po jednom všechny offsety vyhledávací části okénka a teprve pokud se symbol na tomto offsetu shoduje s prvním symbolem ve výhledu, pak zkoumá, jestli se shodují i symboly na dalších po sobě jdoucích pozicích. Implementace tohoto postupu vedla k velmi pomalé kompresi a to i pro relativně malé soubory o velikostech několika stovek kilobajtů.

Je zřejmé, že úzkým hrdlem předchozího postupu je zkoumání každého offsetu vyhledávací části klouzavého okénka. Toto prohledání všech offsetů je navíc opakováno vždy pro každý výstupní token. Je přitom evidentní, že algoritmus po jednom navštívení všech offsetů vyhledávací části klouzavého okénka získá úplnou znalost o jeho obsahu a bylo by proto výhodné tuto znalost využít i pro další iterace.

Jednou z možností jak tuto myšlenku využít, je poznamenat si pro každý symbol rozšířené ASCII tabulky všechny offsety, na kterých se daný symbol nachází v aktuálním stavu vyhledávací části okénka. Pak je možné při hledání shody s prefixem ve výhledu navštívit pouze ty offsety, na kterých se vyskytuje první symbol ve výhledu a tím zcela eliminovat zbytečné navštěvování offsetů, na kterých se první symbol výhledu nevyskytuje.

Zbývá však vyřešit problém, jak aktualizovat offsety symbolů ve vyhledávací části okénka poté, co se provede posuv. Ruční aktualizace offsetů všech symbolů uvnitř vyhledávací části má lineární operační složitost vůči velikosti okénka a je proto nevhodným řešením. Tento problém lze vyřešit tím, že se pro každý symbol uvnitř vyhledávací části okénka nebudou ukládat jeho



Obrázek 6.2 Porovnání časů komprese LZ77 s původním a vylepšeným klouzavým okénkem na vybraných souborech z Calgary korpusu

offsety, ale absolutní pozice vůči začátku vstupních dat. Pak není potřeba při posuvu okénka aktualizaci těchto pozic provádět vůbec, protože absolutní pozice symbolů vůči začátku vstupu se s posuvem okénka nikdy nezmění. Zbývá akorát nalézt způsob pro přepočet absolutní pozice na offset. Toho lze elegantně dosáhnout tak, že je pro okénko udržována absolutní pozice nejlevějšího symbolu uvnitř vyhledávací části klouzavého okénka. Z absolutní pozice p konkrétního symbolu a absolutní pozice t nejlevějšího symbolu uvnitř vyhledávací části velikosti s lze pak offset pro daný symbol snadno vypočítat jako $s - (p - t)$.

Pro uložení absolutních pozic každého symbolu uvnitř vyhledávací části okénka byl použit kontejner *std::vector* o 256 položkách. Indexy 0 až 255 pro tento kontejner reprezentují dané symboly rozšířené ASCII tabulky a na každém indexu se pak nachází instance kruhové fronty, uvnitř které jsou uložené absolutní pozice pro daný symbol. Tato struktura je ve zdrojovém kódu uložená v proměnné s názvem *bookkeeping* a umožňuje $O(1)$ přístup ke kruhové frontě s absolutními pozicemi daného symbolu a také $O(1)$ vkládání a odstraňování absolutních pozic daného symbolu při aktualizaci klouzavého okénka.

Pokud má být symbol c při posuvu odstraněn z klouzavého okénka, je potřeba odstranit absolutní pozici tohoto konkrétního symbolu c ze struktury *bookkeeping*. Díky FIFO vlastnosti fronty a faktu, že jsou ukládány absolutní pozice vůči začátku vstupu, které s posuvem okénka vždy rostou, je tato operace ekvivalentní odstranění počátečního prvku fronty s absolutními pozicemi pro symbol c. Analogicky, pokud má být symbol c vložen do vyhledávací části klouzavého okénka, stačí jeho absolutní pozici vložit na konec kruhové fronty, která obsahuje absolutní pozice symbolu c.

Implementace tohoto postupu přinesla významné zrychlení komprese oproti původnímu naivnímu řešení. Porovnání s původním řešením pro $|S| = 4096$ a $|L| = 16$ na vybraných souborech z Calgary korpusu ukazuje graf 6.2.

Z grafu je zřejmé, že pro soubor *pic* z Canterbury korpusu není zlepšení u vylepšené verze okénka tak dramatické jako u ostatních souborů. Soubor *pic* je černobílý faxový snímek a z jeho hexadecimálního výpisu je zřejmé, že obsahuje vysoký počet stejných bajtů, konkrétně jsou to bajty `0xFF` a `0x00`. Z těchto důvodů nepřináší vylepšená verze klouzavého okénka pro tento soubor tak dramatické zrychlení jako u ostatních souborů, protože pokud vyhledávací část okénka obsahuje z majoritní části bajty shodné s prvním symbolem výhledu, pak i vylepšená verze okénka musí většinovou část vyhledávací části projít.

Výpis kódu 6.2 Dekomprese řetězce algoritmem LZ77

```
uint64_t remaining = token.m_length;
while (remaining) {
    for (uint64_t i = offset_to_index(token.m_offset);
         i < m_decompression_buffer.size() && remaining;
         ++i, --remaining) {
         phrase.push_back(m_decompression_buffer[i]);
    }
}
phrase.push_back(token.m_symbol);
```

Třída *LZ77OriginalSlidingWindow* implementující původní neefektivní klouzavé okénko byla v projektu pro studijní účely ponechána. Pokud je potřeba použít toto původní okénko, např. pro účely měření či testování, stačí díky výše popsanému návrhu pouze vyměnit v požadovaném LZ77 kompresoru instanci třídy *LZ77SlidingWindow* za instanci *LZ77OriginalSlidingWindow* a překompilovat požadované spustitelné soubory.

Z důvodů úspory bitové šířky tokenů byla podle [1] implementována maximální délka shody jako $|L| - 1$, kde $|L|$ značí velikost výhledu. Jak již bylo popsáno v kapitole 5.1, postup pro navýšení maximální délky shody představený v [2] by byl výhodný pouze pro taková data, která obsahují shody délek o velikostech vyhledávací části klouzavého okénka, tj. několik tisíc bajtů.

Velikost vyhledávací části klouzavého okénka i velikost výhledu je nastavitelná pomocí argumentů konzolové aplikace.

6.4.2 Dekompresor

Podobně jako kompresor deleguje veškeré klíčové operace na klouzavé okénko, byl dekompresor navržen tak, aby klíčové operace pro dekompresi řetězce a posuv okénka delegoval na samostatnou třídu *LZ77DecompressionBuffer*. Z tohoto návrhu plynou stejné výhody, jaké byly popsány u kompresoru.

Zajímavostí je, že dekompresní buffer je možné implementovat tak, aby si vystačil pouze s bufferem o velikosti vyhledávací části klouzavého okénka, tj. vůbec není potřeba udržovat obsah výhledu. Pokud má však dekompresní buffer udržovat pouze vyhledávací část klouzavého okénka, pak je potřeba ošetřit situaci, kdy shoda přetéká do výhledu. V tomto případě si stačí uvědomit, že pokud shoda přetéká do výhledu, pak shoda od pozice začínající ve výhledu periodicky opakuje symboly, které se vyskytují na jejím začátku z vyhledávací části okénka. Díky tomu je možné, že si dekompresní buffer nemusí udržovat separátní paměť pro výhled a vystačí si pouze s obsahem vyhledávací části okénka. Tento přístup ukazuje kód 6.2.

6.5 Implementace LZ78

Kompresor i dekompresor LZ78 využívá pro svůj slovník instanci třídy *Trie*. Pro efektivnější běh používá kompresor i dekompresor metody trie *append_to* a *find_from*, které nevkládají, resp. nevyhledávají, celý řetězec, ale svou operaci začínají od vnitřního uzlu, který reprezentuje nějaký již dříve do trie vložený řetězec.

Jak již bylo popsáno v kapitole 5.2, kompresní algoritmus má v teoretické rovině neomezeně velký slovník. V praxi však není reálné nechat slovník růst do nekonečna a je potřeba předem stanovit maximální možnou velikost slovníku v počtu vložených řetězců. To představuje otázku, jak postupovat v momentě, kdy dojde k zaplnění slovníku. V kapitole 5.2 byly představeny celkem tři různé strategie: zamrznutí slovníku, vyprázdnění slovníku a sledování kompresního poměru. V rámci této práce byly implementovány všechny tři tyto strategie.

Strategie pro zamrznutí slovníku funguje přesně tak, jak napovídá její název: ve chvíli, kdy je do slovníku vložen stanovený maximální počet řetězců, stává se slovník statickým a do slovníku již nejsou vkládány žádné další řetězce.

Při strategii vyprázdnění slovníku je ve chvíli, kdy již není možné do slovníku vložit žádný nový řetězec z důvodu zaplnění slovníku, slovník vyprázdněn a opětovně inicializován prázdným řetězcem jako před začátkem komprese, resp. dekomprese.

Při strategii sledování kompresního poměru je po celou dobu komprese udržován aktuální kompresní poměr a pokud stoupne nad stanovený mezní limit, pak je slovník vyprázdněn. Je zřejmé, že u této strategie potřebuje dekompresor vědět přesně v jakém okamžiku má vyprázdnit svůj slovník, aby udržel synchronizaci se slovníkem kompresoru. Z hlediska implementace je proto nutné přijít na způsob, jakým bude kompresor signalizovat vyprázdnění slovníku. Pro signalizaci vyprázdnění slovníku se nabízí, aby kompresor zapsal na výstup speciální kód, který bude jednoznačný, tj. bude zaručené, že v žádném stavu komprese nebude tento kód odpovídat indexu nějakého řetězce ve slovníku. Toho lze dosáhnout tím, že pro tuto strategii bude jeden z indexů slovníku vyhrazen pouze pro signalizaci vyprázdnění slovníku a tím je zaručené, že nemůže dojít k záměně za platný index řetězce uloženého ve slovníku. V implementaci je pro tento kód vyhrazeno nejvyšší možné číslo v dané bitové šířce maximální velikosti slovníku, tj. pokud je maximální velikost slovníku s, pak unikátní kód signalizující vyprázdnění slovníku je roven $2^{\lceil log_2(s) \rceil} - 1$. Kompresor i dekompresor pak při této strategii zaplněného slovníku ošetřují, aby na tento index nikdy nebyl vložen žádný řetězec.

Je důležité poznamenat, že binární i čitelný kompresor musejí až na formát výstupu pracovat stejně, resp. lišit se pouze způsobem zápisu výstupních tokenů, aby bylo možné čitelný kompresor využít k výukovým a studijním účelům. Z toho důvodu čitelný kompresor při strategii sledování kompresního poměru počítá délku výstupního tokenu v bitech stejným způsobem, jakým to dělá binární kompresor, tj. jako $\lceil log_2(s) \rceil + b$, kde s značí maximální velikost slovníku a b značí počet bitů v jednom bajtu. Pokud by čitelný kompresor místo toho počítal délku výstupního tokenu podle počtu zapsaných bajtů na výstup ve formátovaném režimu, pak by binární a čitelné kompresory vyprazdňovaly slovník v různých chvílích při strategii sledování kompresního poměru a poskytovaly by odlišný výstup při kompresi stejných dat, resp. posílaly by na výstup jiné tokeny, což se z výukových důvodů nesmí stát.

Maximální velikost slovníku v počtu vložených řetězců, strategie pro zaplněný slovník a případný mezní kompresní poměr samozřejmě patří mezi parametry, které může uživatel ručně nastavit pomocí argumentů konzolové aplikace.

6.6 Implementace LZSS

6.6.1 Datová struktura slovníku

Implementace kompresoru LZSS stejně jako LZ77 deleguje veškeré kritické operace na samostatnou třídu reprezentující klouzavé okénko. Výhody takového návrhu byly popsány v kapitole 6.4 o implementaci algoritmu LZ77. I v tomto případě byly obě části klouzavého okénka implementovány pomocí kruhové fronty.

Kompresor LZSS využívá pro vyhledávání nad vyhledávací částí okénka binární vyhledávací strom. Kompresor si navíc potřebuje pro každý řetězec v binárním vyhledávacím stromě navíc udržovat seznam offsetů, na kterých se takový řetězec ve vyhledávací části okénka nachází. Z těchto důvodů byl použit kontejner *std::map*, který je ve standardní knihovně jazyka C++ implementován pomocí samo-vyvažovacího binárního vyhledávacího stromu, typická implementace pak využívá červeno-černý strom [18]. Klíče v *std::map* jsou tedy řetězce z vyhledávací části klouzavého okénka, hodnotami jsou pak kruhové fronty, které jsou použity pro ukládání offsetů, na kterých se dané řetězce nacházejí. Implementace využívá implicitního lexikografického řazení řetězců v *std::map*.

Pro nalezení shody s řetězcem ve výhledu je využita metoda *lower_bound*, která vrací první prvek v kontejneru, který není menší než její argument. Pro délku shody pak stačí vypočítat, v kolika symbolech se shoduje řetězec ve výhledu a řetězec nalezený metodou *lower_bound*. Je však důležité si uvědomit, že pokud metoda *lower_bound* vrátí řetězec J, pak je vzhledem k principu fungování této metody možné, že delší shodu s argumentem metody *lower_bound* poskytuje takový řetězec I, který v binárním vyhledávacím stromu lexikograficky předchází řetězec J. Proto je nutné na délku shody kontrolovat i předchůdce řetězce nalezeného metodou *lower_bound*.

6.6.2 Bitové příznaky

V implementaci jsou prohozena čísla pro bitový příznak, který signalizuje, jestli následuje token nebo nekomprimovaný symbol. Původní hodnoty tohoto příznaku podle algoritmu z kapitoly 5.3, tj. 0 signalizující token a 1 signalizující nekomprimovaný symbol, totiž mohou vést k těžko odhalitelné chybě, byť pouze v případě binárního kompresoru a pro malé hodnoty velikostí klouzavého okénka, jak ukazuje následující příklad.

▶ **Příklad 6.1.** Nechť velikost vyhledávací části okénka $|S|$ je 16 a velikost výhledu $|L|$ je 4. Protože algoritmus LZSS na rozdíl od algoritmu LZ77 nepotřebuje nulu pro signalizaci nenalezení shody, je postačující použít pro zakódování všech offsetů vyhledávací části okénka $log_2|S| = 4$ bity. Způsob zakódování těchto offsetů závisí na implementaci, v této práci byl zvolen aditivní kód s konstantou -1, tj. offset 1 je zakódován jako 0, offset 2 je zakódován jako 1, atd. Analogicky, pro zakódování délky shody jsou postačující $log_2|L| = 2$ bity. Velikost tokenu v bitech proto je $4 + 2 = 6$ bitů, velikost nekomprimovaného symbolu je 8 bitů. Dále nechť bitový příznak 0 signalizuje token a 1 signalizuje nekomprimovaný symbol.

Nechť binární kompresor zapsal na výstup taková data, pro jejichž délku $|r|$ v bitech platí, že $|r|\ mod\ 8 = 1$. Protože výstupní data netvoří celočíselný násobek jednoho bajtu, je nutné provést zarovnání a poslední bit zapsaný kompresorem doplnit zprava sedmi nulami na velikost jednoho bajtu. Bez újmy na obecnosti lze předpokládat, že poslední bit zapsaný kompresorem je roven 1. Pak poslední bajt komprimovaných dat je 1000 0000.

Nyní nechť dekompresor správně dekomprimuje všechna data až do posledního bitu zapsaného kompresorem, pro který bylo potřeba provést zarovnání. Dekompresor pak má v takovém stavu na vstupu následující bity: 000 0000. Ačkoliv je z předchozího popisu zřejmé, že se jedná o zarovnání a nikoliv o platná komprimovaná data, jsou tato data dekompresorem považována za platný token. Dekompresor totiž přečte první ze sedmi nul, kterou interpretuje jako bitový příznak signalizující, že následuje token. Dekompresor z parametrů velikosti vyhledávací části okénka a výhledu ví, že platný token má bitovou šířku $4 + 2 = 6$. Dekompresor proto přečte zbývajících šest nul, které kvůli použití aditivního kódu interpretuje jako shodu na offsetu 1 délky 1 a tato data zapíše na výstup. Kvůli tomuto poslednímu kroku se pak dekomprimovaná data neshodují s původními.

Je zřejmé, že pokud se prohodí hodnoty bitového příznaku, tj. 0 bude použita pro signalizaci symbolu a 1 pro signalizaci tokenu, tak předchozí situace nikdy nenastane vzhledem k tomu, že symbol rozšíření ASCII tabulky má vždy právě 8 bitů a implementace *BitWriter* provádí zarovnání pouze nulami.

Stejně jako v případě implementace algoritmu LZ77 je velikost vyhledávací části klouzavého okénka i velikost výhledu nastavitelná pomocí argumentů konzolové aplikace.

6.7 Implementace LZW

6.7.1 Datová struktura slovníku

Původní implementace algoritmu LZW v rámci této práce používala pro slovník instanci třídy *FlatHashTable*. Jak bylo popsáno v kapitole 5.4, jsou do této struktury ukládány uzly trie o dvou

Výpis kódu 6.3 Tělo hešovací funkce pro uzly trie uložené v hešovací tabulce

```
uint64_t seed = 0;
if (node.first.has_value()) {
    boost::hash_combine(seed, node.first.value());
}
boost::hash_combine(seed, node.second);
return seed;
```

položkách: ukazatel na rodičovský uzel a aktuální symbol vloženého řetězce. Jak plyne z teorie popsané v kapitole 5.4, bylo v tomto případě nutné navrhnout hešovací funkci, která bude funkcí o dvou proměnných.

Reálná situace je ještě o něco složitější, může se totiž stát, že první složka uzlu trie, tj. ukazatel na rodičovský uzel, nemusí existovat, neboť slovník LZW je inicializovaný všemi symboly rozšířené ASCII tabulky, tj. řetězci, které mají délku jednoho symbolu a nemají platný ukazatel na rodičovský uzel. Z toho důvodu byla hešovací funkce pro hešování vnitřních uzlů trie implementována za použití třídy *std::optional* ze standardní knihovny jazyka C++ a za pomocí knihovny *boost* pro kombinování hešovacích kódů. Tělo této hešovací funkce ukazuje kód 6.3.

Algoritmus LZW, který pro svůj slovník využíval hešovací tabulku s otevřeným adresováním společně s hešovací funkcí popsanou výše, byl z hlediska času komprese a dekomprese výrazně pomalejší, než algoritmus LZW využívající pro slovník třídu *Trie*.

Úzkým hrdlem této implementace je hešovací funkce, která by měla být navržena speciálně pro uzly trie místo kombinování hešovacích kódů složek uzlu, aby se minimalizovaly kolize hešovací funkce. Navíc vzhledem k tomu, že kompresor i dekompresor prochází přes uzly trie velmi často, není režie pro výpočet hešovacích kódů a jejich kombinování zanedbatelná. Dalším důvodem může být i technika pro navštěvování přihrádek: implementace používá lineární přidávání a technika dvojitého hešování by mohla přinést příznivější výsledky. Pro techniku dvojitého hešování jsou však nutné dokonce dvě hešovací funkce, které budou kvalitně a efektivně hešovat vnitřní uzly trie různými způsoby. Z těchto důvodů bylo od použití *FlatHashTable* upuštěno a výsledná implementace algoritmu LZW používá pro svůj slovník instanci třídy *Trie*.

Ačkoliv se ukázalo, že implementace algoritmu LZW, která pro slovník používá instanci *Trie*, je výrazně rychlejší než implementace využívající třídu *FlatHashTable*, je ve zdrojových kódech projektu pro studijní účely implementace hešovací tabulky s otevřeným adresováním ponechána.

6.7.2 Synchronizace kompresoru a dekompresoru

Pro algoritmus LZW je specifické, že dekompresor buduje svůj slovník o jeden krok pozadu oproti kompresoru, jak je vidět z jeho pseudokódu v kapitole 5.4. Z toho důvodu byla potřeba při implementaci strategií zaplněného slovníku velká obezřetnost, neboť tato vlastnost dekompresoru může vést k těžko odhalitelným chybám.

Příkladem může být implementace strategie vyprázdnění slovníku. Původní implementace používala následující optimalizaci: pokud potřebuje kompresor, resp. dekompresor vložit frázi r do slovníku, který je plně obsazený a strategie pro zaplněný slovník je vyprázdnění slovníku, pak je slovník vyprázdněn, inicializován všemi symboly rozšířené ASCII tabulky a zároveň je do slovníku hned po inicializaci rovnou vložen řetězec r. Tato implementace vzhledem ke zpoždění dekompresoru při budování slovníku vedla k nezvyklé chybě, kterou demonstruje následující příklad.

▶ **Příklad 6.2.** Nechť je velikost slovníku 257, strategie pro plný slovník je vyprázdnění slovníku a vstupními daty pro kompresor je řetězec *afff*.

Kompresor přečte ze vstupu symbol *a*, který díky inicializaci nalezne ve slovníku. Řetězec *af* není ve slovníku, na výstup proto zapíše token 97 a vloží řetězec *af* na index 256 do slovníku.

Dále kompresor ze vstupu přečte symbol *f*, který je ve slovníku. Řetězec *ff* však není ve slovníku, kompresor proto zapíše na výstup index 102 a chce vložit řetězec *ff* do slovníku na další volnou pozici. Slovník je však plný, kompresor proto slovník vyprázdní a rovnou vloží na další volnou pozici, tj. na index 256, řetězec *ff*.

V dalším kroku kompresor přečte ze vstupu symbol *f*. Řetězec obsahující pouze symbol *f* má kompresor díky počáteční inicializaci vždy ve slovníku. Po přečtení dalšího symbolu kompresor zjistí, že řetězec *ff* také má ve slovníku. Další symbol však již na vstupu není, kompresor proto posílá na výstup token 256 a ukončuje proces komprese.

Dekompresor tedy bude dekomprimovat posloupnost tokenů s indexy 97, 102 a 256. Dekompresor přečte token 97, na výstup zapíše symbol *a*, ale zatím nepřidává nový řetězec do slovníku, protože k tomu potřebuje znát první symbol fráze následující.

Dekompresor proto pokračuje přečtením tokenu 102 a na výstup zapíše symbol *f*. Teprve nyní dekompresor vkládá do slovníku na index 256 řetězec *af*.

Nyní dekompresor přečte ze vstupu token 256. Řetězec s tímto indexem dekompresor má ve slovníku, konkrétně jde o řetězec *af*. Dekompresor proto zapíše řetězec *af* na výstup a chce do slovníku vložit řetězec *fa*. Slovník je však plný, dekompresor proto slovník vyprázdní a poté rovnou vloží na index 256 řetězec *fa*. Protože na vstupu již nejsou další tokeny, ukončuje dekompresor svoji činnost.

Dekompresor tedy původní data *afff* chybně dekomprimoval jako *afaf*.

Na předchozím příkladu je vidět, jak je dekompresor v budování slovníku o jeden krok zpožděný oproti kompresoru, což je jedním z důvodů popsané chyby. Této chybě je však možné se elegantně vyhnout tím, že kompresor, resp. dekompresor, nepoužívá výše popsanou optimalizaci, tj. po vyprázdnění slovníku je slovník pouze opětovně inicializován symboly rozšířené ASCII tabulky a další řetězec je do slovníku vložen až při další iteraci komprese, resp. dekomprese.

Obdobně jako u implementace algoritmu LZ78 jsou i v tomto případě parametry jako velikost slovníku, strategie pro zaplněný slovník a mezní kompresní poměr nastavitelné pomocí argumentů konzolové aplikace.

Kapitola 7
Testování

Tato kapitola popisuje testování implementovaných tříd pro vstup a výstup, datových struktur a kompresních algoritmů. Dále popisuje třídu, na kterou je testování kompresních algoritmů delegováno.

Pro testování byl použit testovací framework *doctest*. [31] Pro datové struktury *CircularQueue*, *Trie* a *FlatHashTable* a třídy pro vstup a výstup, mezi které patří *BitReader*, *BitWriter*, *BufferedInput* a *BufferedOutput*, byly implementovány jak jednotkové testy testující veřejné rozhraní těchto tříd konkrétními daty, tak testy náhodnými daty.

Testy náhodnými daty byly implementovány tak, aby co nejvíce využívaly prvek náhodnosti. Např. pro třídu *BitWriter* jsou vygenerována náhodná data náhodných délek, které jsou následně prostřednictvím této třídy zapsána na výstup a porovnána s očekávaným výsledkem. U datových struktur jako je např. *CircularQueue* jsou pak součástí testu náhodnými daty i operace, které se mají provést, tj. jestli se bude do fronty vkládat nový prvek nebo odstraňovat prvek ze začátku fronty je také generováno náhodně.

Testování kompresních algoritmů je delegováno na třídu *CompressionTester*. Díky hierarchickému návrhu a polymorfismu tříd pro kompresory a dekompresory je možné tuto testovací třídu používat jednotně pro všechny kompresní algoritmy v binárním i čitelném režimu, protože třída *CompressionTester* pracuje s referencemi na abstraktní třídy *Compressor*, resp. *Decompressor*.

Metoda *test_files* třídy *CompressionTester* testuje pro soubor či složku souborů specifikované parametrem *path*, jestli jsou data po kompresi a následné dekompresi totožná s původními daty. Složka souborů může obsahovat další podsložky v libovolné hloubce zanoření, metoda *test_files* rekurzivně projde všechny podsložky a spustí test pro soubory, které se v těchto složkách nacházejí.

Metoda *test_random_files* třídy *CompressionTester* pak generuje náhodné soubory, na kterých testuje, jestli se data po kompresi a následné dekompresi shodují s původními. Počet náhodně vygenerovaných souborů je nastavitelný parametrem *iterations*, velikosti náhodných souborů jsou také generované náhodně, nicméně velikosti jsou shora omezené hodnotou parametru *file_size_max*. Za pozornost stojí uchování vygenerovaného náhodného souboru v případě neúspěchu testu. Pokud se pro vygenerovaný náhodný soubor stane, že se data po dekompresi neshodují s původními daty, je tento soubor uložen do složky *failed_rnd_files*, aby bylo možné na tomto souboru implementovaný kompresní algoritmus odladit. Jméno tohoto souboru je též generováno náhodně a to tak dlouho, dokud není zajištěno, že v dané složce neexistuje žádný jiný soubor se stejným jménem.

Prostřednictvím třídy *CompressionTester* jsou otestovány všechny kompresní algoritmy na vhodných datových korpusech. Seznam těchto korpusů a popis souborů, které obsahují, je k dispozici v příloze B. Kompresní algoritmy jsou na těchto korpusech otestovány v binárním i v čitelném

režimu, v případě algoritmů typu LZ2 jsou algoritmy otestovány rovněž pro všechny tři možné strategie zacházení se zaplněným slovníkem. Pro strategii sledování kompresního poměru byly separátně otestovány mezní kompresní poměry 0.0, 0.3, 0.5 a 1.0.

V rámci této práce byl použit nástroj GitLab CI/CD k důkladnému otestování aplikace. Po každém odeslání kódu je prostřednictvím systému GitLab CI/CD ověřeno, zda byl veškerý kód úspěšně zkompilován, a zda úspěšně procházejí rozsáhlé sady testů pro zjištění funkčnosti aplikace.

Kapitola 8
Měření

Tato kapitola se zabývá popisem implementace aplikace pro měření kompresních algoritmů a prezentací konkrétních výsledků měření pro dílčí algoritmy.

Cílem měření bylo změřit pro kompresní algoritmy dobu komprese, dekomprese a kompresní poměr. Z důvodu automatizace těchto měření byla implementována samostatná konzolová aplikace *measurements*. Tato aplikace byla navržena analogickým způsobem jako aplikace kompresních algoritmů, včetně rozdělení aplikace na prezentační a aplikační vrstvu.

Aplikace *measurements* umožňuje pro soubor či složku souborů a kompresní algoritmus změřit dobu komprese, dekomprese a kompresní poměr. Tato data jsou měřena pro binární kompresory, resp. dekompresory, protože varianty kompresorů, resp. dekompresorů, s čitelným výstupem jsou určené pouze pro výukové a studijní účely. Všechny parametry specifické pro daný kompresní algoritmus, jako je velikost vyhledávací části klouzavého okénka a velikost výhledu pro algoritmy typu LZ1, nebo velikost slovníku, strategie pro zaplněný slovník a mezní kompresní poměr pro algoritmy typu LZ2, je možné nastavit ručně pomocí argumentů aplikace *measurements*.

Kromě reálného času měří aplikace i procesorový čas, aby byly výsledky měření co nejpřesnější. Procesorový čas zahrnuje součet uživatelského a systémového času, tj. součet času, po který procesor program skutečně vykonával, a času stráveného uvnitř operačního systému. Procesorový čas tedy nezahrnuje čekání na vstupně/výstupní zařízení a čas, po který program na daném procesoru nebyl vykonáván.

Měření je pro každý soubor implicitně opakováno desetkrát, dílčí časy jsou sečtené a výsledek je pak zprůměrovaný celkovým počtem iterací. Postup opakovaného měření je doporučovaný i autory knihovny *Boost Timer*. [32] Počet celkových iterací pro měření jednoho souboru je možné nastavit ručně pomocí příslušného argumentu aplikace.

Volba vhodné knihovny pro měření reálného i procesorového času není zcela triviální úloha. Knihovna *std::chrono* jazyka C++ měří pouze reálný čas, hodiny *std::clock* ze standardní knihovny *ctime* naopak měří procesorový čas. Navíc, při použití *std::clock* by nebyla aplikace *measurements* přenositelná: zatímco na operačních systémech Linux měří *std::clock* procesorový čas, na platformě Windows tyto hodiny měří reálný čas. [33, 34]. Z těchto důvodů byla pro měření zvolena knihovna *Boost Timer*, která prostřednictvím třídy *cpu_timer* dokáže změřit reálný, uživatelský a systémový čas. [32]

Rozlišení měřeného reálného a procesorového času prostřednictvím knihovny *Boost Timer* se odvíjí od operačního systému i procesoru, na kterém je program pro měření spouštěn. Obzvlášť uživatelský a systémový čas může mít výrazně nižší rozlišení oproti reálnému času. [32] Pro velmi malé soubory s velmi krátkou dobou komprese, resp. dekomprese, se proto může stát, že na daném operačním systému a procesoru bude změřený procesorový čas rovný nule. Pro vyřešení tohoto problému byla v aplikaci *measurements* implementována druhá metoda pro měření časů.

Implicitní metoda používá diskrétní měření, tj. postupně sčítá dílčí časy komprese, resp. dekomprese, podle počtu zvolených iterací a výsledný akumulovaný čas vydělí celkovým počtem iterací. Alternativní metoda využívá spojité měření, tj. tato metoda spustí měření a následně sekvenčně opakuje kompresi, resp. dekompresi, v nastaveném počtu iterací a měření zastaví až po ukončení všech iterací komprese, resp. dekomprese. Tento celkový čas je následně podělen celkovým počtem zvolených iterací. V kombinaci s velmi vysokým počtem iterací může tato metoda pro velmi malé soubory kompenzovat nízké rozlišení procesoru a změřit procesorový čas pro takto malé soubory tak, aby výsledek nebyl nulový. Metodu měření lze také nastavit ručně pomocí odpovídajícího argumentu konzolové aplikace.

Pro účely porovnávání měření se může stát, že je potřeba pro určitý soubor či složku souborů spustit měření pro všechny implementované algoritmy. Z těchto důvodů byl napsán shellový skript *measurements.sh*, který pro zvolený soubor či složku souborů spustí měření pro všechny čtyři implementované algoritmy s parametry, které jsou skriptu předány na příkazové řádce.

8.1 Parametry měření

Pro měření kompresních algoritmů v rámci této práce byl zvolen počet iterací rovný 10, pro velmi malé soubory byl z důvodu nízkého rozlišení procesoru navýšen počet iterací na 10000 až 100000.

Jako výchozí metoda pro měření souborů byla zvolena metoda diskrétních měření, pouze pro velmi malé soubory byla zvolena metoda spojitého měření.

Z důvodu přesnějších informací o tom, jak efektivně algoritmus využívá výpočetní zdroje, je ve vykreslených grafech a výsledcích měření v příloze C uváděn pouze procesorový čas.

Pro zajištění významnosti srovnání algoritmů jsou pro parametry algoritmů zvolené hodnoty, které používají typické implementace, jak je popsáno v [1]. Pro algoritmy typu LZ1 jsou to $|S| = 4096$ a $|L| = 16$, pro algoritmy typu LZ2 je pro velikost slovníku zvolena hodnota 65536. Jako strategie zaplněného slovníku byla zvolena strategie zamrznutí slovníku, aby nedocházelo k diskriminaci algoritmu LZW, které musí slovník inicializovat 256 hodnotami, zatímco LZ78 jej inicializuje pouze jednou hodnotou. Při strategii vyprázdnění slovníku by algoritmus LZW byl v porovnání s LZ78 penalizován dobou inicializace slovníku při každém jeho vyprázdnění, a obdobné by to bylo při strategii sledování kompresního poměru, pokud by docházelo k častému vyprazdňování slovníku. Navíc, pokud při strategii sledování kompresního poměru nestoupne aktuální kompresní poměr nad mezní stanovenou hodnotu, chová se tato strategie identicky, jako strategie zamrznutí slovníku.

Pro měření byl použit počítač Dell Latitude 7280 s procesorem Intel Core i5-6300U o frekvenci 2.4 GHz, velikostí operační paměti 8 GB a operačním systémem Ubuntu 22.04.2 LTS.

8.2 Výsledky měření

Detailní výsledky měření všech implementovaných algoritmů na datový korpusech, které jsou popsány v příloze B, lze nalézt v příloze C.

Zajímavým výsledkem měření je, že algoritmus LZ77, pro který bylo implementováno vylepšení popsané v kapitole 6.4.1, je často v době komprese rychlejší, než algoritmus LZSS. Toto samozřejmě neplatí pro původní algoritmus LZ77 bez implementovaného vylepšení. Pokud se použije původní neefektivní klouzavé okénko, které je implementováno ve třídě *LZ77OriginalSlidingWindow*, pak je algoritmus LZSS konzistentně rychlejší. Toto pozorování ukazuje, že použití důmyslné techniky pro zrychlení kompresoru LZ77 může přinést lepší výsledky, než použití složitější datové struktury jako je binární vyhledávací strom, jehož režie zpomaluje kompresor.

Podle výsledků měření jsou v rychlosti komprese konzistentně nejlepší algoritmy typu LZ2, tj. algoritmy LZ78 a LZW.

V kompresním poměru již tyto algoritmy nemají tak dominantní převahu, nicméně rozdíl oproti algoritmům typu LZ1 není výrazný a drobně vyšší kompresní poměr pro některé soubory

je silně kompenzován významnou rychlostí komprese algoritmů LZ2. Dále je možné si všimnout, že existují soubory, pro které kompresní algoritmy vytváří data s kompresním poměrem vyšším než 1, tj. dosahují záporné komprese. Z výsledků měření je zřejmé, že zápornou kompresi lze pozorovat u dvou typů souborů.

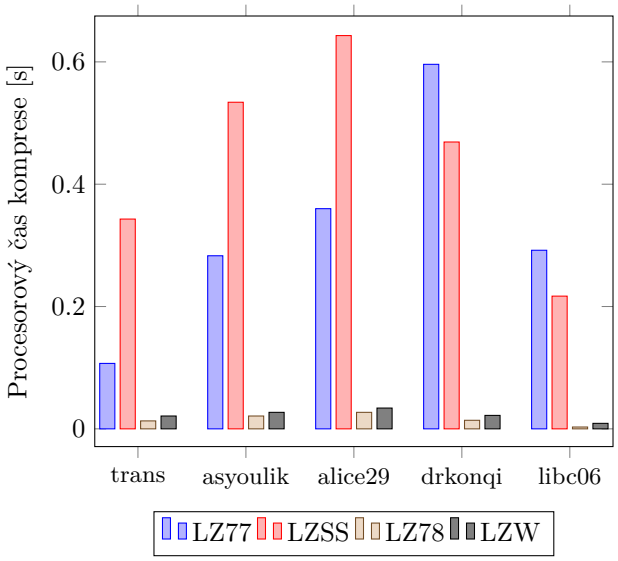
Prvním typem jsou soubory, které mají velmi malou velikost, konkrétně jsou to desítky až stovky bajtů. Pro tyto soubory si kompresní algoritmy nestačí vybudovat dostatečně velký slovník a v kombinaci s režií bitové šířky výstupních tokenů pak kompresní algoritmy na těchto malých souborech dosahují záporné komprese.

Druhým typem jsou soubory, které obsahují náhodná nebo téměř náhodná data a mají proto velmi malou redundanci. Mezi tyto soubory patří například náhodně vygenerovaná data, obrázky nebo zvuky hromů či ohňostrojů. V těchto souborech proto existuje jen málo vzorů, pokud vůbec nějaké, které může kompresní algoritmus využít k dosažení zmenšení velikosti.

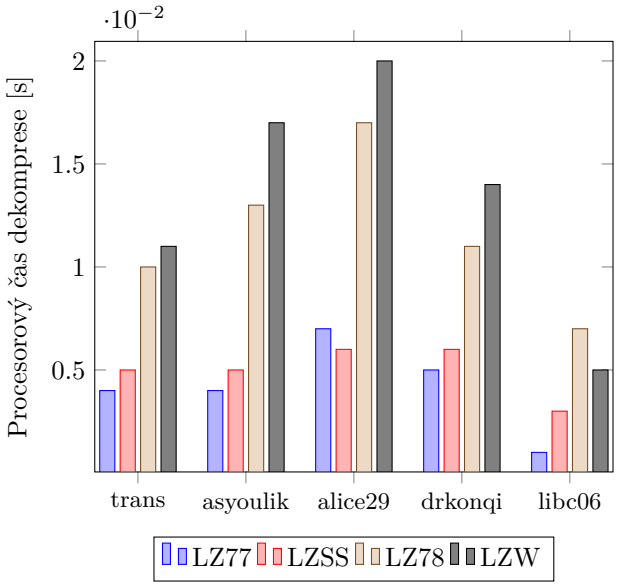
Nejzajímavějších výsledků v oblasti kompresního poměru pak dosahuje soubor *ultima* z korpusu Prague. Jedná se o formátovaný textový soubor, na kterém algoritmy LZ77 a LZW dosahují kompresních poměrů 1.000329 a 1.038009, zatímco algoritmy LZSS a LZ78 dosahují kompresních poměrů 0.773459 a 0.834623. Z těchto výsledků plyne, že mohou existovat konkrétní data, na kterých některé kompresní algoritmy poskytují lepší výsledky než jiné algoritmy a to i přesto, že jsou takové kompresní algoritmy stejného typu. Například lepšího kompresního poměru než LZ77 může algoritmus LZSS dosahovat díky své strategii zápisu výstupních tokenů, kdy na výstup zapisuje token reprezentující shodu pouze v případě, že je bitová délka takového tokenu menší, než bitová délka shody. U algoritmů LZ78 a LZW to pak může být odlišný způsob zpracovávání vstupních symbolů způsobujících neshodu, kdy LZ78 takový symbol v aktuální iteraci ze vstupu odstraní, zatímco LZW symbol způsobující neshodu na vstupu ponechává pro další iteraci.

V době dekomprese jsou však algoritmy typu LZ2 dokonce horší než algoritmy typu LZ1, což plyne z vlastnosti asymetrie algoritmů LZ1 a symetrie algoritmů LZ2. Pro soubory, které se zkomprimují jednou, ale jsou dekomprimovány velice často proto může být vhodnější volbou některý z algoritmů typu LZ1. Pro obecné použití lze doporučit kompresní algoritmus typu LZ2, například algoritmus LZW.

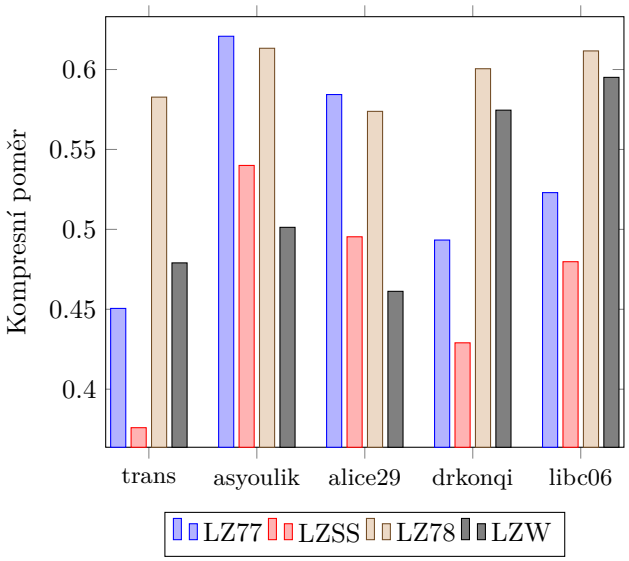
Porovnání procesorové doby komprese, dekomprese a kompresního poměru na vybraných souborech z Calgary, Canterbury a Prague korpusů pro všechny implementované kompresní algoritmy ukazují grafy 8.1, 8.2 a 8.3.



Obrázek 8.1 Porovnání časů komprese na vybraných souborech z korpusů Calgary, Canterbury a Prague



Obrázek 8.2 Porovnání časů dekomprese na vybraných souborech z korpusů Calgary, Canterbury a Prague



Obrázek 8.3 Porovnání kompresních poměrů na vybraných souborech z korpusů Calgary, Canterbury a Prague

Závěr

Cílem této práce bylo seznámení se slovníkovými kompresními metodami, dále pak analýza, implementace a otestování kompresních algoritmů LZ77, LZ78, LZSS a LZW v jazyce C++.

V textu této práce byla představena teorie komprese dat, základní charakterizace kompresních metod a detailní analýza kompresních algoritmů LZ77, LZ78, LZSS a LZW. Součástí byla i teoretická analýza datových struktur, které se v těchto kompresních algoritmech používají.

Všechny algoritmy byly implementovány v jazyce C++ jak v binárním režimu, tak v režimu s čitelným textovým výstupem usnadňující pochopení implementovaných algoritmů. Výstupem jsou konzolové aplikace, které umožňují kromě binárního a čitelného režimu nastavit i další parametry kompresního algoritmu, jako je velikost vyhledávací části klouzavého okénka a velikost výhledu, velikost slovníku, strategii zaplněného slovníku a mezní kompresní poměr pro vyprázdnění slovníku.

Pro kompresní algoritmy byly navržena a implementována testovací třída, která umožňuje ověřit korektnost dekomprimovaných dat s původními daty, a to jak pro konkrétní datové soubory, tak pro náhodně vygenerovaná data. Dále byly implementovány jednotkové testy datových struktur, které byly v rámci této práce naprogramovány pro potřeby kompresních algoritmů.

Pro účely měření a automatizace měření byla implementována samostatná konzolová aplikace, pomocí které lze změřit dobu komprese, dobu dekomprese a kompresní poměr na zvolených datových souborech. V aplikaci pro měření lze také nastavit všechny příslušné parametry daných kompresních algoritmů.

Prostřednictvím implementované aplikace pro měření byly zaznamenány výsledky měření všech implementovaných algoritmů na zvolených datových korpusech. Pro lepší názornost jsou součástí práce i grafy, které vybrané výsledky měření vizualizují. Kompletní výsledky detailních měření jsou k dispozici v příloze této práce, včetně seznamu datových korpusů a popisu souborů v těchto korpusech, které byly pro měření použity.

Možnosti rozšíření

Práci je možné rozšířit implementací dalších slovníkových kompresních algoritmů, jako je např. LZMA, LZX, QIC-122, LZFG, LZRW1, LZP a další. Zajímavou výzvou by také mohla být implementace komplexního slovníkového kompresního algoritmu DEFLATE.

Mezi další možnosti rozšíření patří i implementace kompresních metod, které nejsou slovníkové, např. některých kontextových či statistických metod, jako je Huffmanovo kódování, MNP5, PPM či ACB. Další možností je vytvoření grafického uživatelského rozhraní pro kompresní aplikace a pro aplikaci provádějící měření implementovaných kompresních metod.

Příloha A

Uživatelská příručka

A.1 Závislosti

1. CMake
   - `https://cmake.org/`
   - Pro distribuce založené na Debianu: `sudo apt-get install cmake`
2. Doctest
   - `https://github.com/doctest/doctest`
   - Pro distribuce založené na Debianu: `sudo apt-get install doctest-dev`
3. Boost Timer
   - `https://www.boost.org/`
   - Pro distribuce založené na Debianu: `sudo apt-get install libboost-timer-dev`
4. Doxygen
   - `https://www.doxygen.nl/`
   - Pro distribuce založené na Debianu: `sudo apt-get install doxygen graphviz`

A.2 Použití

Projekt používá standardní nástroj CMake, pro kompilaci proto lze použít například následující příkazy.

```
mkdir build
cd build
cmake ..
make -j $(nproc)
```

A.2.1 LZ77 a LZSS

Příručka ukazuje použití pro LZ77, nicméně stejné argumenty má i aplikace algoritmu LZSS.

```
./lz77 [-h] [--act VAR] [--mode VAR] [--in VAR] [--out VAR]
       [--sbs VAR] [--lbs VAR]

  -h, --help    Zobrazí nápovědu a ukončí program.
  -v, --version Vypíše informace o verzi a ukončí program.
  --act         Určuje, zda se mají vstupní data komprimovat ("compress") nebo
                dekomprimovat ("decompress"). [výchozí: "compress"]
  --mode        Určuje, zda je výstup komprese nebo vstup dekomprese binární
                ("binary") nebo čitelný ("readable"). [výchozí: "binary"]
  --in          Nastavuje cestu ke vstupnímu souboru, který se má zpracovat.
                Pokud není zadán, použije se standardní vstup.
  --out         Nastavuje cestu k výstupnímu souboru. Pokud není zadán,
                použije se standardní výstup.
  --sbs         Nastavuje velikost vyhledávací části okénka,
                musí být alespoň 2. [výchozí: 4096]
  --lbs         Nastavuje velikost výhledu, musí být alespoň 2. [výchozí: 16]
```

A.2.2 LZ78 a LZW

Příručka ukazuje použití pro LZ78, nicméně vyjma minimální velikosti slovníku (256) má stejné argumenty i aplikace pro algoritmus LZW.

```
./lz78 [-h] [--act VAR] [--mode VAR] [--in VAR] [--out VAR]
       [--ds VAR] [--stg VAR] [--thr VAR]

  -h, --help    Zobrazí nápovědu a ukončí program.
  -v, --version Vypíše informace o verzi a ukončí program.
  --act         Určuje, zda se mají vstupní data komprimovat ("compress") nebo
                dekomprimovat ("decompress"). [výchozí: "compress"]
  --mode        Určuje, zda je výstup komprese nebo vstup dekomprese binární
                ("binary") nebo čitelný ("readable"). [výchozí: "binary"]
  --in          Nastavuje cestu ke vstupnímu souboru, který se má zpracovat.
                Pokud není zadán, použije se standardní vstup.
  --out         Nastavuje cestu k výstupnímu souboru. Pokud není zadán,
                použije se standardní výstup.
  --ds          Nastaví velikost slovníku, musí být alespoň 1. [výchozí: 65536]
  --stg         Určuje, zda je strategie slovníku "freeze" (zamrznutí),
                "flush" (vyprázdnění) nebo "ratio" (sledování kompresního
                poměru). [výchozí: "freeze"]
  --thr         Určuje mezní kompresní poměr pro slovníkovou
                strategii "ratio". [výchozí: 0,75]
```

A.2.3 Measurements

Použití:

```
./measurements [-h] [--alg VAR] [--sbs VAR] [--lbs VAR] [--ds VAR] [--stg VAR]
               [--thr VAR] [--in VAR] [--itr VAR] [--out VAR] [--mth VAR]
```

```
-h, --help    Zobrazí nápovědu a ukončí program.
-v, --version Vypíše informace o verzi a ukončí program.
--alg         Určuje, který algoritmus se má použít pro měření.
              Podporované algoritmy jsou "lz77", "lz78", "lzss", "lzw".
--sbs         Nastavuje velikost vyhledávací části okénka pro algoritmy LZ1,
              musí být alespoň 2. [výchozí: 4096]
--lbs         Nastavuje velikost výhledu pro algoritmy LZ1,
              musí být alespoň 2. [výchozí: 16]
--ds          Nastavuje velikost slovníku pro algoritmy LZ2,
              musí být alespoň 256. [výchozí: 65536]
--stg         Určuje, zda je strategie slovníku "freeze" (zamrznutí),
              "flush" (vyprázdnění) nebo "ratio" (sledování kompresního
              poměru). [výchozí: "freeze"]
--thr         Určuje mezní kompresní poměr pro slovníkovou
              strategii "ratio". [výchozí: 0,75]
--in          Nastavuje cestu k souboru nebo složce souborů, nad kterými
              se provede měření.
--itr         Nastaví počet opakování měření pro každý jednotlivý
              soubor. [výchozí: 10]
--out         Nastaví cestu k výstupnímu souboru, do kterého se budou
              zapisovat výsledky měření. Pokud není zadán, použije se ve
              výchozím nastavení standardní výstup.
--mth         Určuje, zda je metoda měření "parts" (diskrétní)
              nebo "whole" (spojitá).
```

A.2.4 Testy

Všechny testy lze spustit ze složky se zkompilovanými binárními soubory.

```
cd build
ctest . --output-on-failure -j $(nproc)
```

Příloha B

Datové korpusy

Všechny uvedené datové korpusy se v projektu nacházejí ve složce *src/test/resources*.

B.1 Artificial Corpus

Tabulka B.1 Artificial Corpus

Název	Popis	Velikost[B]
a.txt	Písmeno 'a'.	1
aaa.txt	Písmeno 'a', které se opakuje 100 000krát.	100000
alphabet.txt	Tolik opakování abecedy, aby bylo zaplněno 100 000 znaků.	100000
random.txt	100 000 náhodně vybraných znaků.	100000

B.2 Calgary Corpus

Tabulka B.2 Calgary Corpus

Název	Popis	Velikost[B]
bib	Bibliografie.	111261
book1	Beletrie.	768771
book2	Literarura faktu.	610856
geo	Geofyzikální údaje.	102400
news	Dávkový soubor USENET.	377109
obj1	Objektový kód pro VAX.	21504
obj2	Objektový kód pro Apple Mac.	246814
paper1	Technický dokument.	53161
paper2	Technický dokument.	82199
paper3	Technický dokument.	46526
paper4	Technický dokument.	13286
paper5	Technický dokument.	11954
paper6	Technický dokument.	38105
pic	Černobílý faxový snímek.	513216

Název	Popis	Velikost[B]
progc	Zdrojový kód v jazyce C.	39611
progl	Zdrojový kód v jazyce LISP.	71646
progp	Zdrojový kód v jazyce PASCAL.	49379
trans	Přepis terminálové relace.	93695

B.3 Canterbury Corpus

Tabulka B.3 Canterbury Corpus

Název	Popis	Velikost[B]
alice29.txt	Anglický text.	152089
asyoulik.txt	Shakespeare.	125179
cp.html	Zdrojový kód HTML.	24603
fields.c	Zdrojový kód v jazyce C.	11150
grammar.lsp	Zdrojový kód v jazyce LISP.	3721
kennedy.xls	Tabulka aplikace Excel.	1029744
lcet10.txt	Technický text.	426754
plrabn12.txt	Poezie.	481861
ptt5	Testovací sada CCITT.	513216
sum	Spustitelný soubor SPARC.	38240
xargs.1	Manuálová stránka GNU.	4227

B.4 Large Corpus

Tabulka B.4 Large Corpus

Název	Popis	Velikost[B]
E.coli	Kompletní genom bakterie E. Coli.	4638690
bible.txt	Bible ve verzi krále Jakuba.	4047392
world192.txt	Světová kniha faktů CIA.	2473400

B.5 Miscellaneous Corpus

Tabulka B.5 Miscellaneous Corpus

Název	Popis	Velikost[B]
bible.txt	Bible z projektu Gutenberg.	4791960
crlf.txt	Text obsahující kontrolní znaky CRLF.	5
eastman_clumsy.in	Anglický jazykolam.	46
eastman_easy.in	Anglický jazykolam.	44
green_eggs_and_ham.txt	Dětská říkanka.	3595
havran.txt	Poezie.	5828
laws_of_robotics.txt	Zákony robotiky.	257
numbers.txt	Náhodně vygenerovaná čísla.	1048575
pangram_cze.txt	Pangramy v českém jazyce.	470

Název	Popis	Velikost[B]
pangram_eng.txt	Pangramy v anglickém jazyce.	351
pi.txt	Prvních milion číslic čísla pí.	1000000
raven.txt	Poezie.	6625
rnd_file	Náhodně vygenerovaný soubor.	100000
rnd_file_big	Náhodně vygenerovaný soubor.	1000000
XLS_10_rows.xls	Tabulka aplikace Excel.	8704
XLS_100_rows.xls	Tabulka aplikace Excel.	20480
XLS_1000_rows.xls	Tabulka aplikace Excel.	140288
XLS_5000_rows.xls	Tabulka aplikace Excel.	672256
XLSX_10_rows.xlsx	Tabulka aplikace Excel.	5425
XLSX_100_rows.xlsx	Tabulka aplikace Excel.	9299
XLSX_1000_rows.xlsx	Tabulka aplikace Excel.	42669
XLSX_5000_rows.xlsx	Tabulka aplikace Excel.	188887

B.6 Prague Corpus

Tabulka B.6 Prague Corpus

Název	Popis	Velikost[B]
abbot	Součást aplikace interiérového designu.	349055
age	Věková struktura obyvatelstva ve světě.	137216
bovary	Gustave Flaubert: Madame Bovary, v němčině.	2202291
collapse	Zdrojový kód JavaScriptu.	2871
compress	Stránka Wikipedie o kompresi dat.	111646
corilis	Údaje o půdním pokryvu CORILIS.	1262483
cyprus	Monitorování kvality ovzduší na Kypru.	555986
drkonqi	Obsluha při pádu prostředí KDE.	111056
emission	Údaje o emisích Waterbase.	2498560
firewrks	Zvuky ohňostrojů.	1440054
flower	Fotografie květiny.	10287665
gtkprint	Nástroj příkazového řádku.	3756
handler	Zdrojový kód Java ze systému sledování GPS.	11873
higrowth	Finanční výpočty.	129536
hungary	Monitorování kvality ovzduší v Maďarsku.	3705107
libc06	Dynamicky linkovaná knihovna.	4812
lusiadas	Luís Vaz de Camoes: Os Lusíadas, v portugalštině.	625664
lzfindmt	Zdrojový kód jazyka C z archivačního programu.	22922
mailfilder	Zdrojový kód jazyka Python z ECM frameworku.	43732
mirror	Součást softwarového balíčku.	90968
modern	Modern. En berättelse, ve švédštině.	388909
nightsht	Fotografie nočního města.	14751763
render	Zdrojový kód z akční hry v jazyce C++.	15984
thunder	Zvuky hromu.	3172048
ultima	Mack Reynolds: Ultima Thule, v angličtině.	1073079
usstate	Zdrojový kód Java ze systému sledování GPS.	8251
venus	Ultrafialový snímek mraků Venuše.	13432142
w01vett	Databázový soubor.	1381141

Název	**Popis**	**Velikost[B]**
wnvcrdt	Databázový soubor.	32855
xmlevent	Zdrojový kód PHP z generátoru událostí.	7542

Příloha C

Výsledky měření

C.1 LZ77

C.1.1 Artificial Corpus

Tabulka C.1 Výsledky měření LZ77 pro Artificial Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
a.txt	0.005	0.000001	4
aaa.txt	6.459	0.004	0.19535
alphabet.txt	0.253	0.002	0.1961
random.txt	0.117	0.005	1.18735

C.1.2 Calgary Corpus

Tabulka C.2 Výsledky měření LZ77 pro Calgary Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bib	0.178	0.003	0.550957
book1	1.997	0.036	0.653617
book2	1.199	0.027	0.566567
geo	0.536	0.004	0.848574
news	0.687	0.017	0.615955
obj1	0.074	0.000011	0.699730
obj2	0.351	0.01	0.503594
paper1	0.095	0.004	0.561445
paper2	0.176	0.002	0.585287
paper3	0.097	0.002	0.609487
paper4	0.028	0.001	0.619374
paper5	0.025	0.001	0.616697
paper6	0.073	0.001	0.561947
pic	20.486	0.018	0.284939
progc	0.071	0.000013	0.548560

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
progl	0.123	0.005	0.404377
progp	0.094	0.003	0.407886
trans	0.107	0.004	0.450536

C.1.3 Canterbury Corpus

Tabulka C.3 Výsledky měření LZ77 pro Canterbury Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
alice29.txt	0.36	0.007	0.584342
asyoulik.txt	0.283	0.004	0.620815
cp.html	0.033	0.001	0.540219
fields.c	0.019	0.001	0.446816
grammar.lsp	0.01	0.000003	0.534265
kennedy.xls	8.699	0.038	0.262284
lcet10.txt	0.909	0.018	0.572367
plrabn12.txt	1.346	0.023	0.640454
ptt5	20.831	0.02	0.284939
sum	0.203	0.001	0.530779
xargs.1	0.011	0.000005	0.634493

C.1.4 Large Corpus

Tabulka C.4 Výsledky měření LZ77 pro Large Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
E.coli	36.945	0.186	0.439053
bible.txt	10.597	0.175	0.496585
world192.txt	5.088	0.107	0.625197

C.1.5 Miscellaneous Corpus

Tabulka C.5 Výsledky měření LZ77 pro Miscellaneous Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bible.txt	11.936	0.219	0.487631
crlf.txt	0.007	0.000001	3.2
eastman_clumsy.in	0.002	0.000001	1.630435
eastman_easy.in	0.007	0.000001	1.5
green_eggs_and_ham.txt	0.004	0.001	0.354659
havran.txt	0.015	0.001	0.720144
laws_of_robotics.txt	0.007	0.000003	1.120623
numbers.txt	4.806	0.052	0.723113
pangram_cze.txt	0.007	0.000002	1.231915
pangram_eng.txt	0.007	0.000002	1.068376
pi.txt	4.606	0.05	0.716454

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
raven.txt	0.019	0.000005	0.625962
rnd_file	0.07	0.01	1.52032
rnd_file_big	0.551	0.079	1.516854
XLS_10_rows.xls	0.094	0.000005	0.396369
XLS_100_rows.xls	0.207	0.001	0.387598
XLS_1000_rows.xls	0.886	0.004	0.353002
XLS_5000_rows.xls	3.444	0.027	0.348998
XLSX_10_rows.xlsx	0.009	0.000006	1.347465
XLSX_100_rows.xlsx	0.012	0.000007	1.366814
XLSX_1000_rows.xlsx	0.024	0.001	1.017437
XLSX_5000_rows.xlsx	0.093	0.011	0.902227

C.1.6 Prague Corpus

Tabulka C.6 Výsledky měření LZ77 pro Prague Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
abbot	0.226	0.023	1.369781
age	1.4	0.006	0.602226
bovary	4.736	0.112	0.533261
collapse	0.009	0.000011	0.611982
compress	0.125	0.006	0.386973
corilis	3.436	0.065	0.813608
cyprus	0.881	0.024	0.246607
drkonqi	0.596	0.005	0.493310
emission	23.909	0.11	0.304182
firewrks	2.364	0.108	1.363066
flower	13.128	0.618	0.961568
gtkprint	0.392	0.004	0.441134
handler	0.031	0.000047	0.420366
higrowth	0.696	0.008	0.570691
hungary	5.905	0.155	0.256118
libc06	0.292	0.001	0.522984
lusiadas	6.639	0.029	0.529818
lzfindmt	0.031	0.000087	0.378894
mailflder	0.231	0.000177	0.396163
mirror	0.556	0.004	0.557339
modern	0.985	0.018	0.63827
nightsht	17.565	1.099	1.385368
render	0.028	0.003	0.413914
thunder	22.782	0.191	1.019021
ultima	1.091	0.066	1.000329
usstate	0.022	0.000340	0.425767
venus	25.031	0.783	1.106041
w01vett	20.117	0.054	0.233206
wnvcrdt	4.086	0.013	0.231663
xmlevent	0.013	0.000267	0.472023

C.2 LZ78

C.2.1 Artificial Corpus

Tabulka C.7 Výsledky měření LZ78 pro Artificial Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
a.txt	0.000011	0.000003	3
aaa.txt	0.005	0.000881	0.013400
alphabet.txt	0.01	0.001762	0.06803
random.txt	0.021	0.016	1.02566

C.2.2 Calgary Corpus

Tabulka C.8 Výsledky měření LZ78 pro Calgary Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bib	0.02	0.01	0.578603
book1	0.148	0.075	0.526093
book2	0.115	0.063	0.516588
geo	0.016	0.014	0.771328
news	0.077	0.051	0.585801
obj1	0.003	0.002	0.851656
obj2	0.045	0.032	0.618741
paper1	0.008	0.007	0.686594
paper2	0.015	0.007	0.632733
paper3	0.007	0.005	0.703134
paper4	0.001938	0.004	0.823875
paper5	0.001	0.002	0.855697
paper6	0.005	0.005	0.720273
pic	0.041	0.018	0.155757
progc	0.005792	0.01	0.716367
progl	0.01	0.006	0.570458
progp	0.006	0.005	0.596104
trans	0.013	0.01	0.582731

C.2.3 Canterbury Corpus

Tabulka C.9 Výsledky měření LZ78 pro Canterbury Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
alice29.txt	0.027	0.017	0.573828
asyoulik.txt	0.021	0.013	0.613298
cp.html	0.004	0.002	0.693167
fields.c	0.001	0.001	0.749238
grammar.lsp	0.001	0.000408	0.863209
kennedy.xls	0.109	0.045	0.309265

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
lcet10.txt	0.086	0.056	0.509718
plrabn12.txt	0.098	0.058	0.529545
ptt5	0.042	0.017	0.155757
sum	0.006	0.003	0.691763
xargs.1	0.001	0.000517	0.953631

C.2.4 Large Corpus

Tabulka C.10 Výsledky měření LZ78 pro Large Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
E.coli	0.688	0.26	0.349425
bible.txt	0.662	0.247	0.443934
world192.txt	0.462	0.174	0.461291

C.2.5 Miscellaneous Corpus

Tabulka C.11 Výsledky měření LZ78 pro Miscellaneous Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bible.txt	0.762	0.292	0.43845
crlf.txt	0.000008	0.000005	3
eastman_clumsy.in	0.00002	0.000012	1.869565
eastman_easy.in	0.000015	0.000013	1.75
green_eggs_and_ham.txt	0.000471	0.000339	0.724896
havran.txt	0.000882	0.000697	0.951613
laws_of_robotics.txt	0.000055	0.000046	1.435798
numbers.txt	0.16	0.087	0.57255
pangram_cze.txt	0.000153	0.000154	1.459574
pangram_eng.txt	0.000075	0.000063	1.39886
pi.txt	0.151	0.085	0.567857
raven.txt	0.001	0.001	0.867925
rnd_file	0.028	0.025	1.32452
rnd_file_big	0.18	0.116	1.16
XLS_10_rows.xls	0.002	0.000557	0.475873
XLS_100_rows.xls	0.002016	0.001263	0.453906
XLS_1000_rows.xls	0.014	0.008	0.342873
XLS_5000_rows.xls	0.084	0.042	0.303137
XLSX_10_rows.xlsx	0.001283	0.001188	1.419908
XLSX_100_rows.xlsx	0.002121	0.00196	1.41327
XLSX_1000_rows.xlsx	0.009	0.006	1.125829
XLSX_5000_rows.xlsx	0.043	0.039	0.923706

C.2.6 Prague Corpus

Tabulka C.12 Výsledky měření LZ78 pro Prague Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
abbot	0.085	0.064	1.138468
age	0.03	0.021	0.537415
bovary	0.358	0.143	0.45289
collapse	0.000448	0.000356	0.996517
compress	0.015	0.01	0.493175
corilis	0.236	0.109	0.636588
cyprus	0.097	0.028	0.15706
drkonqi	0.014	0.011	0.600499
emission	0.277	0.105	0.337934
firewrks	0.237	0.143	1.093051
flower	0.926	0.548	1.079197
gtkprint	0.004	0.003	0.559159
handler	0.001	0.002	0.653078
higrowth	0.02	0.014	0.654042
hungary	0.963	0.184	0.100811
libc06	0.003	0.007	0.611638
lusiadas	0.103	0.069	0.557042
lzfindmt	0.003	0.003	0.646366
mailflder	0.004	0.004	0.523941
mirror	0.013	0.011	0.648613
modern	0.081	0.05	0.55775
nightsht	2.311	1.247	1.181549
render	0.003	0.001	0.686874
thunder	0.437	0.244	1.003816
ultima	0.198	0.119	0.834623
usstate	0.001024	0.000719	0.655799
venus	1.799	0.86	1.078051
w01vett	0.196	0.065	0.132165
wnvcrdt	0.033	0.013	0.127204
xmlevent	0.000994	0.000742	0.766773

C.3 LZSS

C.3.1 Artificial Corpus

Tabulka C.13 Výsledky měření LZSS pro Artificial Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
a.txt	0.000177	0.000003	2
aaa.txt	0.063	0.002	0.13298
alphabet.txt	0.08	0.002	0.13309
random.txt	0.542	0.009	1.06752

C.3.2 Calgary Corpus

Tabulka C.14 Výsledky měření LZSS pro Calgary Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bib	0.45	0.003	0.487484
book1	3.549	0.039	0.56818
book2	2.55	0.029	0.484237
geo	0.512	0.006	0.835166
news	1.536	0.019	0.539417
obj1	0.085	0.001	0.583659
obj2	0.92	0.013	0.432403
paper1	0.212	0.003	0.47533
paper2	0.343	0.004	0.496137
paper3	0.196	0.002	0.517754
paper4	0.056	0.000028	0.525290
paper5	0.047	11954	0.001
paper6	0.191	0.001	0.480147
pic	0.817	0.02	0.220198
progc	0.162	0.000831	0.459721
progl	0.214	0.007	0.329704
progp	0.149	0.001	0.328743
trans	0.343	0.005	0.375911

C.3.3 Canterbury Corpus

Tabulka C.15 Výsledky měření LZSS pro Canterbury Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
alice29.txt	0.643	0.006	0.495355
asyoulik.txt	0.534	0.005	0.539995
cp.html	0.088	0.000121	0.467992
fields.c	0.037	0.000164	0.357309
grammar.lsp	0.011	0.000159	0.432948
kennedy.xls	4.006	0.039	0.340664
lcet10.txt	1.785	0.02	0.483332
plrabn12.txt	2.14	0.022	0.562735
ptt5	0.677	0.018	0.220198
sum	0.135	0.003	0.484702
xargs.1	0.015	0.000189	0.511474

C.3.4 Large Corpus

Tabulka C.16 Výsledky měření LZSS pro Large Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
E.coli	21.746	0.185	0.347782

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bible.txt	18.873	0.181	0.417522
world192.txt	10.695	0.119	0.550169

C.3.5 Miscellaneous Corpus

Tabulka C.17 Výsledky měření LZSS pro Miscellaneous Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bible.txt	23.134	0.216	0.41446
crlf.txt	0.000008	0.000003	1.2
eastman_clumsy.in	0.000152	0.000005	1.065217
eastman_easy.in	0.000146	0.000005	1.068182
green_eggs_and_ham.txt	0.009	0.000135	0.278999
havran.txt	0.03	0.000282	0.612045
laws_of_robotics.txt	0.001	0.000018	0.863813
numbers.txt	5.351	0.05	0.641334
pangram_cze.txt	0.003	0.000033	0.912766
pangram_eng.txt	0.001	0.000026	0.783476
pi.txt	4.834	0.049	0.63261
raven.txt	0.036	0.001	0.546566
rnd_file	0.564	0.007	1.117830
rnd_file_big	5.515	0.063	1.117668
XLS_10_rows.xls	0.02	0.000362	0.34869
XLS_100_rows.xls	0.085	0.000868	0.393164
XLS_1000_rows.xls	0.596	0.009	0.396021
XLS_5000_rows.xls	2.683	0.029	0.39622
XLSX_10_rows.xlsx	0.036	0.000297	0.965714
XLSX_100_rows.xlsx	0.055	0.0005	1.01398
XLSX_1000_rows.xlsx	0.233	0.002	0.812393
XLSX_5000_rows.xlsx	2.683	0.029	0.39622

C.3.6 Prague Corpus

Tabulka C.18 Výsledky měření LZSS pro Prague Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
abbot	1.689	0.02	1.015768
age	0.629	0.007	0.543625
bovary	10.264	0.1	0.460318
collapse	0.01	0.000114	0.493905
compress	0.33	0.006	0.316886
corilis	4.784	0.067	0.640298
cyprus	0.807	0.021	0.179233
drkonqi	0.469	0.006	0.429009
emission	5.167	0.099	0.238470
firewrks	6.977	0.083	1.072004
flower	52.196	0.602	0.934162

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
gtkprint	0.129	0.002	0.384824
handler	0.031	0.001	0.335635
higrowth	0.541	0.006	0.514467
hungary	6.18	0.143	0.187283
libc06	0.217	0.003	0.479759
lusiadas	2.472	0.031	0.466858
lzfindmt	0.072	0.000914	0.29897
mailflder	0.133	0.003	0.316496
mirror	0.349	0.005	0.484863
modern	1.764	0.018	0.542227
nightsht	76.439	0.88	1.078812
render	0.047	0.001	0.340528
thunder	15.434	0.186	1.036578
ultima	5.844	0.06	0.773459
usstate	0.021	0.001	0.348321
venus	65.287	0.758	0.899153
w01vett	2.189	0.05	0.165294
wnvcrdt	0.409	0.011	0.166821
xmlevent	0.021	0.000314	0.377751

C.4 LZW

C.4.1 Artificial Corpus

Tabulka C.19 Výsledky měření LZW pro Artificial Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
a.txt	0.000056	0.000051	2
aaa.txt	0.005	0.001	0.00894
alphabet.txt	0.007	0.004	0.04536
random.txt	0.031	0.026	1.00278

C.4.2 Calgary Corpus

Tabulka C.20 Výsledky měření LZW pro Calgary Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bib	0.029	0.022	0.482847
book1	0.174	0.085	0.422797
book2	0.128	0.07	0.418256
geo	0.031	0.026	0.836699
news	0.1	0.069	0.495093
obj1	0.004	0.004	0.843378
obj2	0.073	0.051	0.553291
paper1	0.011	0.007	0.578243
paper2	0.015	0.012	0.519033

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
paper3	0.009	0.007	0.589133
paper4	0.004	0.001	0.72121
paper5	0.002	0.002	0.762757
paper6	0.008	0.004	0.6153
pic	0.054	0.025	0.136621
progc	0.009	0.006	0.604832
progl	0.014	0.01	0.461296
progp	0.009	0.005	0.486725
trans	0.021	0.011	0.479022

C.4.3 Canterbury Corpus

Tabulka C.21 Výsledky měření LZW pro Canterbury Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
alice29.txt	0.034	0.02	0.46123
asyoulik.txt	0.027	0.017	0.501266
cp.html	0.005	0.003	0.607568
fields.c	0.001919	0.001401	0.635336
grammar.lsp	0.000701	0.000551	0.757323
kennedy.xls	0.12	0.051	0.34145
lcet10.txt	0.1	0.06	0.402124
plrabn12.txt	0.114	0.065	0.425152
ptt5	0.049	0.023	0.136621
sum	0.007	0.005	0.655178
xargs.1	0.001	0.001	0.847883

C.4.4 Large Corpus

Tabulka C.22 Výsledky měření LZW pro Large Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
E.coli	0.752	0.275	0.263324
bible.txt	0.774	0.295	0.352236
world192.txt	0.518	0.189	0.377497

C.4.5 Miscellaneous Corpus

Tabulka C.23 Výsledky měření LZW pro Miscellaneous Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
bible.txt	0.86	0.311	0.351866
crlf.txt	0.000057	0.000053	2
eastman_clumsy.in	0.000071	0.000063	1.652174
eastman_easy.in	0.000067	0.000062	1.590909
green_eggs_and_ham.txt	0.000595	0.000456	0.599722

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
havran.txt	0.001	0.001	0.854839
laws_of_robotics.txt	0.000118	0.000113	1.439689
numbers.txt	0.181	0.099	0.469784
pangram_cze.txt	0.000176	0.000163	1.370213
pangram_eng.txt	0.000137	0.000122	1.287749
pi.txt	0.174	0.095	0.464154
raven.txt	0.001	0.001	0.778566
rnd_file	0.046	0.043	1.43638
rnd_file_big	0.24	0.157	1.246278
XLS_10_rows.xls	0.001115	0.000779	0.458869
XLS_100_rows.xls	0.003	0.002	0.460742
XLS_1000_rows.xls	0.018	0.011	0.329308
XLS_5000_rows.xls	0.085	0.046	0.354787
XLSX_10_rows.xlsx	0.003	0.001	1.681106
XLSX_100_rows.xlsx	0.003	0.003	1.667061
XLSX_1000_rows.xlsx	0.014	0.01	1.167686
XLSX_5000_rows.xlsx	0.058	0.045	0.894334

C.4.6 Prague Corpus

Tabulka C.24 Výsledky měření LZW pro Prague Corpus

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
abbot	0.099	0.071	1.213923
age	0.029	0.021	0.593109
bovary	0.323	0.124	0.422624
collapse	0.000604	0.000507	0.888889
compress	0.018	0.012	0.402827
corilis	0.239	0.112	0.673731
cyprus	0.091	0.03	0.121532
drkonqi	0.022	0.014	0.574575
emission	0.295	0.112	0.320877
firewrks	0.305	0.194	1.17975
flower	1.174	0.728	1.293914
gtkprint	0.004	0.005	0.540202
handler	0.001906	0.001382	0.566327
higrowth	0.029	0.023	0.646971
hungary	0.775	0.145	0.078705
libc06	0.009	0.005	0.595096
lusiadas	0.103	0.065	0.573461
lzfindmt	0.004	0.002	0.530146
mailflder	0.009	0.002	0.430348
mirror	0.02	0.01	0.630595
modern	0.096	0.056	0.447261
nightsht	2.365	1.223	1.49158
render	0.003	0.001	0.580706
thunder	0.507	0.286	1.14565
ultima	0.186	0.124	1.038009

Název	Komprese [s]	Dekomprese [s]	Kompresní poměr
usstate	0.001	0.001	0.58102
venus	1.92	1.029	1.276411
w01vett	0.191	0.063	0.105604
wnvcrdt	0.035	0.013	0.101714
xmlevent	0.001	0.001	0.666667

Bibliografie

1. SALOMON, David. *Data Compression: The Complete Reference*. 4. vyd. London: Springer, 2007. ISBN 978-1-84628-603-2.
2. SAYOOD, Khalid. *Introduction to Data Compression*. San Francisco: Morgan Kaufmann Publishers – Elsevier, 2006. ISBN 978-0-12-620862-7.
3. PU, Ida Mengyi. *Fundamental Data Compression*. Oxford: Butterworth-Heinemann – Elsevier, 2006. ISBN 978-0-7506-6310-6.
4. MELICHAR, Bořivoj. *Jazyky a překlady*. 2. roz. vyd. Praha: České vysoké učení technické v Praze, 2003. ISBN 80-01-02776-7.
5. RICH, Elaine. *Automata, Computability and Complexity: Theory and Applications*. New Jersey: Prentice Hall – Pearson, 2007. ISBN 978-0-13-228806-4.
6. ROSEN, Kenneth H. *Discrete Mathematics and Its Applications*. 7. vyd. New York: McGraw-Hill, 2012. ISBN 978-0-07-338309-5.
7. OLŠÁK, Petr. *Úvod do algebry, zejména lineární*. 2. vyd. Praha: Česká technika – nakladatelství ČVUT, 2013. ISBN 978-80-01-05291-4.
8. ADÁMEK, Jiří. *Foundations of Coding: Theory and Applications of Error-Correcting Codes with an Introduction to Cryptography and Information Theory*. Chichester: John Wiley & Sons, Inc., 1991. ISBN 0-471-62187-0.
9. DUMAS, Jean-Guillaume; ROCH, Jean-Louis; TANNIER, Éric; VARRETTE, Sébastien. *Foundations of Coding: Compression, Encryption, Error Correction*. New Jersey: John Wiley & Sons, Inc., 2015. ISBN 978-1-118-88144-6.
10. HARRIS, David Money; HARRIS, Sarah L. *Digital Design and Computer Architecture*. San Francisco: Morgan Kaufmann Publishers – Elsevier, 2007. ISBN 978-0-12-370497-9.
11. HOBZA, Tomáš. *Matematická statistika* [online]. Praha: České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská, 2011. Dostupné také z: `https://people.fjfi.cvut.cz/hobzatom/mast/mast.pdf`.
12. SHANNON, Claude Elwood. A mathematical theory of communication. *Bell System Technical Journal*. 1948, roč. 27, č. 4, s. 623–656. ISSN 0005-8580.
13. MCANLIS, Colt; HAECKY, Aleks. *Understanding Compression: Data Compression for Modern Developers*. California: Morgan Kaufmann Publishers – Elsevier, 2016. ISBN 978-1-491-96153-7.
14. GOURLEY, David; TOTTY, Brian. *HTTP: The Definitive Guide*. Sebastopol: O'Reilly Media, Inc., 2002. ISBN 978-1-56592-509-0.

15. MATOUŠEK, Jiří; NEŠETŘIL, Jaroslav. *Kapitoly z diskrétní matematiky*. 4. upr. a dop. vyd. Praha: Univerzita Karlova, nakladatelství Karolinum, 2019. ISBN 978-80-246-1740-4.
16. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. 2. vyd. Praha: CZ.NIC, 2022. ISBN 978-80-88168-66-9.
17. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3. vyd. Cambridge: MIT Press, 2009. ISBN 978-0-262-53305-8.
18. WEISS, Mark Allen. *Data structures and algorithm analysis in C++*. 4. vyd. New Jersey: Pearson, 2014. ISBN 978-0-13-284737-7.
19. DROZDEK, Adam. *Data Structures and Algorithms in C++*. 4. vyd. Boston: Cengage Learning, 2013. ISBN 978-1-133-60842-4.
20. THAREJA, Reema. *Data Structures Using C*. 2. vyd. New Delhi: Oxford University Press, 2014. ISBN 978-0-19-809930-7.
21. BRASS, Peter. *Advanced Data Structures*. New York: Cambridge University Press, 2008. ISBN 978-0-511-43388-7.
22. MEHLHORN, Kurt; SANDERS, Peter. *Algorithms and Data Structures – The Basic Toolbox*. Berlin: Springer, 2008. ISBN 978-3-540-77977-3.
23. SEDGEWICK, Robert; WAYNE, Kevin. *Algorithms*. 4. vyd. Boston: Pearson Education, 2011. ISBN 978-0-321-57351-3.
24. LEMPEL, Abraham; ZIV, Jacob. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*. 1977, roč. 23, č. 3, s. 337–343. ISSN 0018-9448.
25. LEMPEL, Abraham; ZIV, Jacob. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*. 1978, roč. 24, č. 5, s. 530–536. ISSN 0018-9448.
26. BELL, Timothy C.; CLEAR, John G.; WITTEN, Ian H. *Text Compression*. New Jersey: Prentice Hall, 1990. ISBN 0-13-911991-4.
27. STORER, James A.; SZYMANSKI, Thomas. Data Compression via Textual Substitution. *Journal of the ACM*. 1982, roč. 29, č. 4, s. 928–951. ISSN 0004-5411.
28. WELCH, Terry A. A Technique for High-Performance Data Compression. *IEEE Computer*. 1984, roč. 17, č. 6, s. 8–19. ISSN 0018-9162.
29. *argparse – Argument Parser for Modern C++* [online]. 2022. Dostupné také z: `https://github.com/p-ranav/argparse`.
30. *std::optional – cppreference* [online]. 2023. Dostupné také z: `https://en.cppreference.com/w/cpp/utility/optional`.
31. *doctest – C++ Testing Framework* [online]. 2023. Dostupné také z: `https://github.com/doctest/doctest`.
32. *Boost Timer Library* [online]. 2011. Dostupné také z: `https://www.boost.org/doc/libs/1_82_0/libs/timer/doc/index.html`.
33. *std::clock – cppreference* [online]. 2021. Dostupné také z: `https://en.cppreference.com/w/cpp/chrono/c/clock`.
34. *clock – Microsoft Learn* [online]. 2022. Dostupné také z: `https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/clock?view=msvc-170`.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `compression_algorithms` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `bi_bap_javormat.pdf` ..... text práce ve formátu PDF