VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Sufixové stromy

Suffix Trees

2018 Jiří Friml

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jiří Friml**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Suffixové stromy
Suffix Trees

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je návrh, implementace a experimenty s kompresními algoritmy založenými na sufixových stromech. Kompresní algoritmy budou využívat různé druhy vstupních abeced (znaky, slabiky, slova).

Práce bude mít následující strukturu:
1. Sufixové stromy, popis vybraných algoritmů nad těmito stromy.
2. Návrh variant kompresního algoritmu založeného na sufixových stromech pro různé abecedy.
3. Implementace kompresních algoritmů.
4. Experimenty, vyhodnocení experimentů.

Seznam doporučené odborné literatury:

[1] Senft M.: Sufixové grafy a bezeztrátová komprese dat, MFF UK, 2013, https://is.cuni.cz/webapps/zzp/detail/42521/
[2] Publikace na http://dblp.uni-trier.de/pers/hd/s/Senft:Martin
Dále podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Datum zadání: 01.09.2016
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 ....

Rád bych na tomto místě poděkoval zejména vedoucímu práce doc. Mgr. Jiřímu Dvorskému, PhD. za čas, který mi při tvorbě této práce věnoval, a za jeho dobré a užitečné rady, a také všem, kteří mě při psaní podporovali, protože bez nich by tato práce nevznikla.

Abstrakt

Sufixový strom je velice zajímavá struktura co se týče práce s řetězci a umožňuje efektivně vykonávat řadu operací nad nimi, např. vyhledávání. Cílem této práce je využití této vlastnosti v kompresním algoritmu, jenž je navržen a implementován. V práci je rozebráno chování sufixového stromu během konstrukce a při simulaci sufixových spojek a, jelikož nezbytnou součástí komprese je udržování textu v posuvném okně, také při posouvání okna. Je navržen a implementován způsob, jak uchovávat pohyby aktivního bodu ve stromu tak, aby bylo možné při dekompresi rekonstruovat původní řetězec. V závěru práce je experimentálně zjištěno a zhodnoceno, jaký vliv na výsledky má komprese nad znaky a nad celými slovy.

Klíčová slova: sufixový strom, posuvné okno, bezztrátová komprese dat

Abstract

Suffix tree is very interesting data structure regarding working with strings and enables effective execution of many operations with them, e.g. searching. Aim of this thesis is to use this feature in compression algorithm which is devised and implemented. We analyse behavior of suffix tree during construction and also during suffix links simulation and, because necessary part of compression is keeping of string in sliding window, also during sliding of the window. A way, how to store movement of active point through tree to be able to reconstruct original string at decompression, is devised and implemented. At the end we find out from experiments, what is the difference between results of compression on characters level and on word level.

Key Words: suffix tree, sliding window, lossless data compression

Obsah

Seznam použitých zkratek a symbolů **7**

Seznam obrázků **8**

Seznam tabulek **9**

Seznam výpisů zdrojového kódu **10**

1 Úvod **11**

2 Základy **13**
2.1 Teorie řetězců 13
2.2 Teorie grafů 14

3 Definice **16**
3.1 Sufixový trie 16

4 Algoritmus **20**
4.1 Konstrukce sufixového stromu 20
4.2 Posouvání stromu v posuvném okně 27
4.3 Vyhledání shod v textu a komprese 31
4.4 Komprese nad slovy 33

5 Implementace **35**
5.1 Diagram tříd 35
5.2 Popis tříd 37

6 Experimenty **42**

7 Závěr **48**

Literatura **49**

Seznam použitých zkratek a symbolů

GCC	–	GNU Compiler Collection
JDK	–	Java Development Kit
HTML	–	Hyper Text Markup Language
XML	–	Extensible Markup Language
RFC	–	Request For Comments
CD	–	Compact Disc

Seznam obrázků

1 Sufixový trie a sufixový strom nad slovem *be-be*. Vysvětlivky použitých značek jsou v Tabulce 2 17

2 Simulace sufixové spojky při zapracování symbolu *a* do sufixového stromu nad slovem *coco* (FTJ = factorToJump, viz Výpis 2) 24

3 Posouvání stromu s posuvným oknem délky 6 nad řetězcem "␣Bye␣bye␣by,". Červeně označené části stromů jsou určeny k odstranění. (Legenda značení je v Tabulce 2.) 30

4 Posuvné okno délky 6 nad řetězcem "␣Bye␣bye␣by,"– vyhledání shody a její překryv do nezpracované části řetězce. 32

5 Komprese řetězce "␣Bye␣bye␣by," 32

6 Komprese a dekomprese nad slovy 34

7 Třídní diagram (část 1.) 36

8 Třídní diagram (část 2.) 37

9 Kódování shody 39

10 Kompresní poměr při kompresi binárních souborů korpusu Calgary s velikostí posuvného okna v intervalu 2^4-2^{21}; komprese na úrovni znaků 43

11 Kompresní poměr při kompresi textových souborů korpusu Calgary s velikostí posuvného okna v intervalu 2^4-2^{21}; komprese na úrovni znaků 44

12 Kompresní poměr při kompresi textových souborů korpusu Calgary s velikostí posuvného okna v intervalu 2^4-2^{16}; komprese na úrovni slov 45

13 Kompresní poměr při kompresi textových souborů korpusu Lightweight s velikostí posuvného okna v intervalu 2^9-2^{22}; komprese na úrovni znaků 45

14 Kompresní poměr při kompresi textových souborů korpusu Lightweight s velikostí posuvného okna v intervalu 2^9-2^{16}; komprese na úrovni slov 46

Seznam tabulek

1 Určení množin UnikátníSufix(μ), Větvící$^L(\mu)$ a Větvící$^P(\mu)$ a množin Výskyt$^L_\mu(bye)$, Výskyt$^P_\mu(bye)$, Kontext$^L_\mu(bye)$ a Kontext$^P_\mu(bye)$ pro slovo "*bye-bye.*". 14
2 Význam symbolů použitých v grafech 17
3 Soubory datového korpusu Lightweight – mohutnost abecedy, velikost a popis . . 26
4 Výsledky simulace sufixové spojky 26
5 Soubory datového korpusu Calgary – mohutnost abecedy, velikost v bytech a popis 43
6 Velikost původního a komprimovaného souboru a slovníku při kompresi na úrovni slov a velikosti posuvného okna 2^{12}; korpus Calgary 44
7 Porovnání kompresního poměru použitého algoritmu s formáty zip a 7z; korpus Calgary 46
8 Porovnání kompresního poměru použitého algoritmu s formáty zip a 7z; korpus Lightweight 47

Seznam výpisů zdrojového kódu

1 Pseudokód konstrukčního algoritmu 21
2 Pseudokód simulace sufixové spojky – varianta horní sufixové spojky 24
3 Pseudokód simulace sufixové spojky – varianta spodní sufixové spojky 25
4 Posouvání sufixového stromu v okně 28
5 Zakódování symbolu do pole bajtů 38
6 Přečtení symbolu v binárním módu 41

1 Úvod

Sufixový strom je velice zajímavá a výhodná datová struktura, kterou lze velmi dobře uplatnit v oblastech, kde je potřeba rychle a efektivně vyhledat podřetězec ve kterékoli části řetězce. Příkladem takového uplatnění je komprese dat a využitím sufixového stromu v bezztrátové kompresi dat se zabývám v této práci.

Co se týče konstrukce sufixového stromu, jako první představil svůj algoritmus Weiner v roce 1973 [1]. Algoritmus pracoval zprava doleva a využíval prefixové spojky pro vytvoření stromu v lineárním čase vzhledem k délce vstupu. Weiner byl následován McCreightem, který představil algoritmus praktičtější ve směru zpracování vstupu – pracoval totiž zleva doprava, přičemž dosahoval stejné asymptotické složitosti díky sufixovým spojkám. Jeho metoda byla posléze upravena Ukkonenem, který jako první představil algoritmus, který zpracovával jeden symbol za druhým a byl asymptoticky stejně rychlý jako jeho předchůdci. Tento algoritmus staví na původním algoritmu s kvadratickou složitostí pro konstrukci sufixového trie.

Problému posuvného okna se věnovalo méně pozornosti než konstrukci grafu, proto byl první algoritmus pro posouvání sufixového stromu představen až v roce 1989 Fialou a Greenem [2]. Nicméně jejich práce měla několik problémů, které se snažil vyřešit ve své doktorské práci Larsson [3]. Navzdory jeho pokusu největší problém se správností algoritmu úpravy popisů hran stále nebyl vyřešen uspokojivě. Vyřešil jej až Senft ve své doktorské práci [4].

Tato doktorská práce se nezabývala pouze sufixovým stromem, ale obecnějším tématem využití sufixového grafu v bezztrátové kompresi dat. Senft ve své práci shrnul poznatky svého výzkumu, během něhož pátral po využití posouvání sufixového grafu a posuvného okna v kompresi dat. Senft ve své práci popisuje konstrukci čtyř různých sufixových grafů (které ale mají v mnoha ohledech stejné či velice podobné vlastnosti), konkrétně sufixového trie, sufixového stromu, DAWG (orientovaného acyklického grafu se slovy) a CDAWG (kompaktního orientovaného acyklického grafu se slovy) nad libovolným řetězcem s alespoň dvěma symboly (zde se autor odkazuje na univerzální Ukkonenův algoritmus [5], jímž – a jeho drobnými obměnami – lze zkonstruovat všechny čtyři druhy grafu), algoritmus posouvání okna a grafu po vstupním řetězci a nakonec také samotný kompresní algoritmus. Autor v práci uvádí, že posouvání sufixového grafu pro potřeby komprese dat lze použít pouze v případě sufixového stromu, jelikož v případě ostatních druhů sufixových grafů mají algoritmy konstrukce (v případě trie) či posuvného okna (v případě DAWG a CDAWG) kvadratickou složitost a tudíž tyto struktury nejsou pro kompresi použitelné. Zmíněná vědecká práce mi posloužila jako základní studijní materiál a zdroj mnoha užitečných informací.

V Kapitole 2: Základy uvádím nezbytné základní pojmy z teorie řetězců a teorie grafů, které jsou používány ve zbytku práce. V Kapitole 3: Definice je uvedena definice ústředního pojmu sufixového stromu a vysvětlena jeho návaznost na sufixový trie, ze kterého sufixový strom vychází. Definice sufixového stromu je doplněna všemi potřebných podklady, na kterých staví a které jsou specifické právě pro sufixový strom, takže je v teorii grafů nenajdeme.

Kapitola 4: Algoritmus vysvětluje kompletní algoritmus komprese pomocí sufixového stromu. Objasňuje základní algoritmus konstrukce doplněný algoritmem simulace sufixových spojek a také zmiňuje princip posouvání stromu a okna a změny, které se při posouvání dějí. V závěru kapitoly je pak ukázán princip kompresního algoritmu pracujícího se znaky a slovy.

V Kapitole 5: Implementace je nastíněna architektura implementační části práce a vysvětlena funkcionalita jednotlivých tříd a jejich návaznost na kompresní algoritmus. V předposlední Kapitole 6: Experimenty jsou uvedeny výsledky experimentální části práce a komentáře k nim a závěrečná Kapitola 7: Závěr shrnuje práci.

2 Základy

Dříve, než přistoupíme k popisu algoritmu, je potřeba zmínit několik základních kamenů, na kterých budeme v dalších částech textu stavět. Těmito základními kameny jsou pojmy z teorie řetězců a grafů, které zopakujeme, a také pojem sufixového stromu a další pojmy, na kterých definice sufixového stromu stojí, které zavedeme.

2.1 Teorie řetězců

Opakování začneme teorií řetězců. *Slovo* μ je konečná sekvence symbolů z neprázdné konečné množiny Σ zvané *abeceda*. *Délku* slova označujeme $|\mu|$ a jediné slovo s délkou nula nazýváme *prázdné slovo* a označujeme je ϵ. *Množinu všech slov nad abecedou* Σ a *množinu všech neprázdných slov nad abecedou* Σ označujeme Σ^* a Σ^+, v tomto pořadí. Jednotlivé symboly abecedy značíme malými latinskými písmeny $(a, b, c, ...)$, slova malými řeckými písmeny $(\alpha, \beta, \gamma, ...)$ a abecedy velkými řeckými písmeny (Σ).

Na slovech definujeme operaci *zřetězení*. Použitím této operace na dvojici řetězců vznikne jediný řetězec, který je spojením sekvencí symbolů obou řetězců. Je-li $\alpha = a_1 a_2 ... a_m$ a $\beta = b_1 b_2 ... b_n$, jejich zřetězení označíme $\alpha\beta$ a platí $\alpha\beta = a_1 a_2 ... a_m b_1 b_2 ... b_n$. Obdobně lze zřetězit slovo se znakem – máme-li slovo α a znak a, jejich zřetězením vznikne slovo αa – a také samostatné znaky.

Pomocí zřetězení lze každé slovo μ zapsat způsobem $\mu = \alpha\beta\gamma$, což je zřetězení tří slov (mohou být i prázdná). V tomto případě říkáme slovu α *prefix*, slovu β *podslovo* a slovu γ *sufix* slova μ. Pokud se α, β nebo γ nerovná celému slovu μ, hovoříme o *vlastním* prefixu, podslovu, sufixu. Množiny všech prefixů, podslov a sufixů slova μ označujeme $\mathrm{Prefix}(\mu)$, $\mathrm{Podslovo}(\mu)$ a $\mathrm{Sufix}(\mu)$, v tomto pořadí.

Konceptu výskytů lze využít pro třídění podslov podle jejich počtu. Podslovo α označíme jako *unikátní*, pokud se ve slově μ vyskytuje právě jednou (tj. platí $\mathrm{Výskyt}^L_\mu(\alpha) = \mathrm{Výskyt}^P_\mu(\alpha) = 1$). Jako *neunikátní* pak označíme takové podslovo α, které se ve slově μ vyskytuje alespoň jednou (tj. $\mathrm{Výskyt}^L_\mu(\alpha) = \mathrm{Výskyt}^P_\mu(\alpha) > 1$; Stojí za povšimnutí, že prázdné slovo je v neprázdném řetězci vždy neunikátní). Množinu všech unikátních podslov v řetězci μ označujeme $\mathrm{Unikátní}(\mu)$ a jejím průnikem s množinami $\mathrm{Prefix}(\mu)$ a $\mathrm{Sufix}(\mu)$ vzniknou množiny $\mathrm{UnikátníPrefix}(\mu)$ a $\mathrm{UnikátníSufix}(\mu)$, v tomto pořadí. Pokud si naopak vezmeme množinu neunikátních podslov slova μ, nalezneme průnik s množinami prefixů a sufixů a ze vzniklých množin vybereme jejich nejdelší členy, získáme nejdelší neunikátní prefix a nejdelší neunikátní sufix slova μ ($\mathrm{NNP}(\mu)$, $\mathrm{NNS}(\mu)$).

Dále je potřeba zavést značení týkající se okolí výskytu podslova. Je-li $\alpha = \mu[i\,..\,j]$ a symbol $\mu[i-1]$ ($\mu[j+1]$) existuje, nazveme jej *levým* (*pravým*) *kontextem* výskytu podslova α ve slově μ. Množinu všech symbolů, které se vyskytují jako levý a pravý kontext podslova α, označujeme $\mathrm{Kontext}^L_\mu(\alpha)$ a $\mathrm{Kontext}^P_\mu(\alpha)$, v tomto pořadí.

μ	*bye-bye.*
$\vert\mu\vert$	8
Výskyt$^L_\mu$(*bye*), Výskyt$^P_\mu$(*bye*)	$\{1,5\}$, $\{3,7\}$
UnikátníSufix(μ)	{-*bye*, *e-bye*, *ye-bye*, *bye-bye*, *bye-bye.*}
NNS(μ)	ϵ
Kontext$^L_\mu$(*bye*), Kontext$^P_\mu$(*bye*)	{-}, {-, .}
Větvící^L(μ), Větvící^P(μ)	$\{\emptyset\}$, $\{e, ye, bye\}$

Tabulka 1: Určení množin UnikátníSufix(μ), Větvící^L(μ) a Větvící^P(μ) a množin Výskyt$^L_\mu$(*bye*), Výskyt$^P_\mu$(*bye*), Kontext$^L_\mu$(*bye*) a Kontext$^P_\mu$(*bye*) pro slovo "*bye-bye.*".

Před závěrem podkapitoly se dostáváme k množině, která je pro definici sufixového stromu, jak zjistíme později, klíčová. Sjednotíme-li prázdné slovo, zprava větvící podslova ve slově μ a unikátní sufixy slova μ do jedné množiny, získáme speciální množinu slov zvanou Explicitní(μ). Platí tedy, že Explicitní(μ) = $\{\epsilon\} \cup$ Větvící^P(μ) $\cup$ UnikátníSufix(μ) a prvky této množiny se nazývají *explicitní* podslova v μ. Prázdné slovo ϵ je zde uvedeno v samostatné množině, neboť se může nacházet v množině Větvící^P(μ) nebo v množině UnikátníSufix(μ) (platí-li $\mu = \epsilon$), anebo ani v jedné z těchto množin (platí-li $\mu = a^k$). Pokud bychom na tyto poslední dva případy zapomněli a množinu $\{\epsilon\}$ z rovnice vynechali, bylo by snadné definici sufixového stromu vyvrátit.

Pro lepší představu o tom, jak vypadají množiny zmíněné v této kapitole pro určité slovo, uvádím v Tabulce 1 jejich prvky pro slovo "*bye-bye.*"(v případě množin, pro jejichž určení je potřeba podslovo α, jsem určil $\alpha = bye$).

2.2 Teorie grafů

Druhá část základních pojmů se týká oboru teorie grafů. Základní grafovou terminologii je možné nalézt v mnoha kvalitních knihách či skriptech (z publikací vydaných na mé alma mater je vhodnou literaturou např. Úvod do Teorie grafů, [6]). Co se týče značení, v textu této práce je používáno standardním způsobem – užívá se malých latinských písmen pro vrcholy $(u, v, ...)$ a hrany $(e, f, ...)$, pro grafy a množiny vrcholů a hran pak velká latinská písmena (G, V, E).

Grafy, které se objevují v této práci, jsou *souvislé*, *orientované* a *acyklické*. Protože jsou orientované, můžeme psát, že pro každý vrchol je *vstupní* hrana jakákoli s ním *incidentní* hrana, pro kterou je tento vrchol jejím *koncovým* vrcholem. Podobně *výstupní* hrana má dotyčný incidentní vrchol jako *počáteční*. Počet vstupních (výstupních) vrcholů udává *vstupní* (*výstupní*) *stupeň* vrcholu a řekneme, že vrchol je *větvící*, pokud je jeho výstupní stupeň větší než 1.

Ohledně stupňů vrchlů je potřeba podotknout, že vrcholy všech grafů mají vstupní stupeň roven 1 vyjma *kořene*, který má vstupní stupeň roven 0. Podobně mají všechny vrcholy výstupní stupeň alespoň 1 vyjma *listů*, jejichž výstupní stupeň je 0. V listech také končí všechny maximální cesty vedoucí z kořene. Každý vrchol, který není listem, se nazývá *vnitřní vrchol*.

Abychom mohli postoupit v diskuzi o sufixových stromech dále, je nutné zmínit se blíže i o hranách grafů. Grafy vyskytující se v tomto textu obsahují hrany, které jsou popsány neprázdnými řetězci. Tyto označené hrany jsou doplněny speciálními neoznačenými hranami zvanými *spojky*, které mají pomocný charakter a bez kterých by se, jak uvidíme později, konstrukce stromu nedostala do rozumných mezí složitosti. Tyto hrany, jak rovněž uvidíme později, nabourávají tvrzení, že graf je acyklický. Protože ale mají pouze pomocný charakter a v hotovém grafu nejsou potřeba, zůstaneme u původního tvrzení. Graf sufixového stromu $G = (V, E, L)$ tedy sestává z množiny vrcholů V, množiny popsaných hran $E \subseteq V \times \Sigma^+ \times V$ a množiny spojek $L \subseteq V \times V$.

Díky tomu, že každá hrana v grafu je popsaná neprázdným řetězcem, můžeme psát, že *a-hrana* vrcholu je ta z jeho výstupních hran, jejíž popis začíná symbolem a. Popisy hran lze také řetězit a vytvářet tak *popisy cest*. Říkáme pak, že cesty začínající v kořeni *reprezentují* své popisy a také koncové vrcholy těchto cest reprezentují jejich popisy (můžeme tedy označit kterýkoli vrchol slovem složeným z popisů všech hran na cestě od kořene až k němu). Cesta reprezentující nejdelší popis cesty se nazývá *páteř* a označíme ji i její koncový vrchol podslovem, které je zároveň nejdelším unikátním prefixem i nejdelším unikátním sufixem slova, nad kterým je strom postaven, tedy celým tímto slovem.

Pro konstrukci sufixového stromu je žádoucí, aby každý vrchol včetně kořene měl platnou sufixovou spojku. Doplníme tedy graf o nový pomocný vrchol *bot*, který bude koncovým vrcholem sufixové spojky vedoucí z kořene (bot sufixovou spojku nemá). Z botu ale nyní nevede žádná hrana, doplníme tedy jednu hranu z botu do kořene pro každý symbol z abecedy Σ.

Pro budoucí lepší představu grafů je vhodné přidat do každé hrany, jejíž popis je delší než 1 symbol, mezi každé dva sousední symboly myšlený *implicitní vrchol*. Tyto myšlené vrcholy rozdělují hranu na několik *implicitních hran*, kde každá implicitní hrana má popis o délce 1. Skutečné vrcholy v tomto kontextu nazýváme *explicitní*. Pomocí explicitních vrcholů a implicitních vrcholů a hran můžeme definovat *implicitní cesty* jako cesty, které propojují explicitní a implicitní vrcholy implicitními hranami. Podobně jako skutečné cesty i implicitní cesty mají své popisy cest a reprezentují je, pokud začínají v kořeni. Stejně tak je reprezentují vrcholy (implicitní i explicitní), ve kterých tyto cesty končí.

3 Definice

3.1 Sufixový trie

Tématem této práce je komprese dat pomocí sufixového stromu, nicméně sufixový strom velice úzce souvisí a jeho definice vlastně vychází ze sufixového trie (při definici stromu se na definici trie budeme odkazovat). Nejprve si tedy definujme sufixový trie.

Definice 1 (Sufixový trie)
Sufixový trie nad slovem μ je graf $G = (V, E, L)$ s následujícími komponentami:

- $V = \{\alpha; \alpha \in \text{Podslovo}(\mu)\}$
- $E = \{(\alpha, a, \alpha a); \alpha, \alpha a \in V \wedge a \in \Sigma\}$
- $L = \{(a\alpha, \alpha); a\alpha, \alpha \in V \wedge a \in \Sigma\}$

Množina vrcholů sufixového trie nad slovem μ je tedy rovna množině podslov slova μ. Každá hrana tohoto grafu je označena jediným symbolem, její počáteční vrchol je tedy nejdelším vlastním prefixem koncového vrcholu. U tohoto grafu také platí, že sufixové hrany se nacházejí mezi každým podslovem a jeho nejkratším vlastním sufixem.

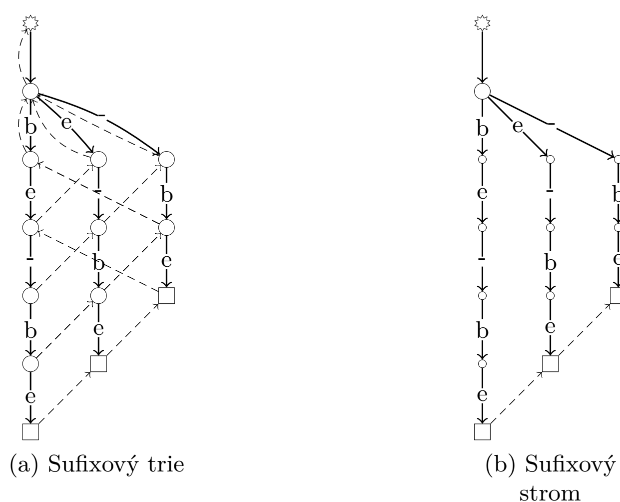
Z definice přímo vyplývají také následující vlastnosti sufixového trie:

- Sufixový trie nad slovem μ je orientovaný stromový graf, jehož kořenem je ϵ a listy jsou unikátní sufixy μ.
- Množina slov reprezentovaných cestami s počátkem v kořeni je rovna množině všech podslov μ.
- Z každého vrcholu vede nejvýše jedna výstupní hrana pro každé $a \in \Sigma$ (vyhledávání řetězců v grafu je tedy deterministické).
- Počet vrcholů je omezen $\Omega(|\mu|^2)$ v nejhorším případě.

Jednoduchost definice sufixového trie a manipulace s ním je velice lákavá, kvadratická prostorová složitost je však nepříjemným faktem. Na Obrázku 1 vidíme, že velikost trie způsobují z podstatné části vnitřní vrcholy, které mají pouze jednu výstupní hranu (v trie je takových vrcholů 9 z 10) a nemají tedy pro naše účely valnou hodnotu. Proto je vhodné hrany, se kterými jsou takové vrcholy incidentní (tj. hrany na takových cestách, kde je počátečním vrcholem kořen nebo větvící vrchol a koncovým vrcholem další větvící vrchol nebo list), jakýmsi způsobem zjednodušit. A to tak, že je vyměníme za jedinou delší hranu, jejíž popis vznikne zřetězením popisů všech hran na dané cestě. Z několika explicitních hran vznikne jedna implicitní cesta. Výsledek takové záměny hran vidíme na Obrázku 1a a Obrázku 1b, kde je poměr počtu vnitřních vrcholů

Tabulka 2: Význam symbolů použitých v grafech

Značka	**Význam**
✲ ○ ∘ □	Bot, explicitní a implicitní vrchol, list
✹ ● • ■	Aktivní vrchol (bot, explicitní a implicitní vrchol, list) – vrchol, přes který se rozkládá aktivní bod
—a⟶	a-hrana
—a⟶	Aktivní a-hrana
- - - - - - -›	Sufixová spojka
- - - - - - ->	Aktivní sufixová spojka
······a····› ⬚	Nová a-hrana, nový list



(a) Sufixový trie

(b) Sufixový strom

Obrázek 1: Sufixový trie a sufixový strom nad slovem *be-be*. Vysvětlivky použitých značek jsou v Tabulce 2

mezi stromem a trie $\frac{1}{10}$. Tato technika, která umožňuje v některých případech i poměrně výraznou redukci velikosti grafu, se nazývá *komprese cesty* a její aplikací na sufixový trie vznikne sufixový strom.

Formální definice komprese cesty využívá konceptu pravého rozšíření:

Definice 2 (Pravé rozšíření)
Nechť $\mu \in \Sigma$ a $\alpha \in \text{Podslovo}(\mu)$. *Pravé rozšíření* podslova α ve slově μ označujeme $\langle\alpha\rangle_\mu^R$ a je definováno jako:

$$\langle\alpha\rangle_\mu^R = \begin{cases} \alpha & \alpha \in \text{Explicitní}(\mu), \\ \langle\alpha a\rangle_\mu^R, \text{ kde } a \in \Sigma \text{ a } \alpha a \in \text{Podslovo}(\mu) & \text{v ostatních případech.} \end{cases}$$

Jinak řečeno pravé rozšíření podslova α ve slově μ je takové nejkratší podslovo $\alpha\beta$, kde $\alpha\beta \in \text{Explicitní}(\mu)$. Pravé rozšíření podslova α, které v množině Explicitní(μ) není, se tedy rovná pravému rozšíření podslova αa (podslovo prodlouženo o jeden symbol), a takto cyklicky připojujeme další symboly, až získáme podslovo $\alpha\beta$, které se již v množině Explicitní(μ) nachází. Ke každému podslovu existuje ve slově μ právě jedno pravé rozšíření.

Příklad 1
Definici pravého rozšíření si ilustrujme na jednoduchém příkladu slova *aaaabb*. Víme, že platí $\text{Explicitní}(\mu) = \{\epsilon\} \cup \text{Větvící}^P(\mu) \cup \text{UnikátníSufix}(\mu)$, takže potřebujeme zjistit obsah posledních dvou množin. Obsah množin je následující:

- $\text{Větvící}^P(aaaabb) = \{\epsilon, a, aa, aaa\}$
- $\text{UnikátníSufix}(aaaabb) = \{bb, abb, aabb, aaabb, aaaabb\}$

Odvození pravého rozšíření všech neexplicitních podslov slova *aaaabb* pak vypadá takto:

$$\begin{aligned} \langle aaaa\rangle_{aaaabb}^P = \langle aaaab\rangle_{aaaabb}^P &= aaaabb \\ \langle aaab\rangle_{aaaabb}^P &= aaabb \\ \langle aab\rangle_{aaaabb}^P &= aabb \\ \langle ab\rangle_{aaaabb}^P &= abb \\ \langle b\rangle_{aaaabb}^P &= bb \end{aligned}$$

Nyní, když známe formální definici pravého rozšíření, získáme pozměněním definice sufixového trie (zavedeme do ní kompresi cesty) definici sufixového stromu. Zní takto:

Definice 3 (Sufixový strom)
Sufixový strom nad slovem μ je graf $G = (V, E, L)$ s následujícími komponentami:

- $V = \{\alpha; \alpha \in \text{Explicitní}(\mu)\}$,
- $E = \{(\alpha, a\beta, \alpha a\beta); \alpha, \alpha a\beta \in V \wedge a \in \Sigma \wedge \beta \in \text{Podslovo}(\mu) \wedge \langle \alpha a \rangle_\mu^P = \alpha a\beta\}$,
- $L = \{(a\alpha, \alpha); a\alpha, \alpha \in V \wedge a \in \Sigma\}$

Pokud porovnáme tuto definici s definicí sufixového trie, vidíme podstatné rozdíly mezi množinami vrcholů a hran. Množina vrcholů v předchozí definici obsahovala všechna podslova, v této definici obsahuje pouze explicitní podslova slova μ (pravá rozšíření všech podslov μ). Důsledkem změny množiny vrcholů je i změna množiny hran, které nyní vedou pouze mezi explicitními vrcholy a jejich popisy již nemusí být pouze jednotlivé znaky, mohou jimi být i celá podslova.

Stejně jako u definice sufixového trie i zde uvádím několik vlastností, které vyplývají z definice. Některé jsou shodné s vlastnostmi sufixového trie:

- Sufixový trie nad slovem μ je orientovaný stromový graf, jehož kořenem je ϵ a listy jsou unikátní sufixy μ.
- Množina slov reprezentovaných implicitními cestami s počátkem v kořeni je rovna množině všech podslov μ.
- Z každého vrcholu vede nejvýše jedna výstupní hrana pro každé $a \in \Sigma$ (vyhledávání řetězců tedy probíhá deterministicky).
- V sufixovém stromu nad slovem μ s alespoň dvěma symboly existuje nejvýše $|\mu| - 1$ uzlů, nejvýše $|\mu|$ listů a nejvýše $2|\mu| - 2$ hran.
- Sufixové hrany vedou z každého vrcholu vyjma kořene a případně nejkratšího listu. Kořen žádnou sufixovou hranu nemá a nejkratší list ji má pouze tehdy, pokud nejdelší neunikátní sufix je zároveň větvící podslovo (a tedy jeho vrchol je explicitní – existuje v grafu). Ostatní sufixové hrany vedou mezi dvěma různými větvícími vrcholy nebo mezi dvěma různými listy.

4 Algoritmus

Konečně nyní, když jsme v předchozích sekcích položili všechny potřebné základní kameny, můžeme přistoupit k popisu použitého kompresního algoritmu. Algoritmus má tři části, které se vzájemně prolínají:

- Konstrukce sufixového stromu
- Posouvání stromu v posuvném okně
- Vyhledávání shod v již zpracovaném textu a komprese

Z poslední odrážky seznamu lze vypozorovat, že algoritmus pracuje na stejném principu jako některé slovníkové kompresní algoritmy, např. LZ77 a LZ78 (tedy na principu ukládání odkazu na předchozí výskyt aktuálně zpracovávaného symbolu). Rozdíl je pouze ve způsobu vyhledání symbolu ve zpracovaném textu, což se v případě použitého algoritmu děje právě pomocí sufixového stromu.

Ale pojďme popořádku, žádný z bodů seznamu totiž není triviální a proto je každému věnována zvláštní sekce.

4.1 Konstrukce sufixového stromu

Algoritmus konstrukce sufixového stromu je založen na obecném Ukkonenově online algoritmu, kterým je možné (jak je zmíněno v Sekci 1) zkonstruovat nejen sufixový strom, ale i sufixový trie, DAWG a CDAWG a který vytváří strukturu sufixového stromu postupně podle vstupních dat, přičemž ji udržuje validní po každém kroku. Tento vysokoúrovňový krok sestává z přidání dalšího symbolu k již zpracovanému řetězci a úpravy sufixového grafu odpovídajícím způsobem. Lze tedy říci, že přijímá jako vstupy graf nad slovem μ a nový symbol a a jeho výstupem je graf nad slovem μa. Ve Výpisu 1 je reprezentován funkcí `AppendSymbol`.

Stejně jako v případě definic uvádím rozbor algoritmu pro snadnější pochopení nejen pro sufixový strom, ale i pro jeho "předchůdce", sufixový trie.

`Build` – funkce, která vytvoří sufixový graf. Nejprve vytvoří nezbytné počáteční součásti grafu (kořen, bot, sufixovou spojku mezi kořenem a botem a hrany mezi botem a kořenem) funkcí `CreateEmptyGraph` a poté dotvoří graf pro celý vstupní řetězec funkcí `AppendString`.

`AppendString` – jednoduchá funkce, která projde vstupní řetězec a na každý jeho jednotlivý znak aplikuje funkci `AppendSymbol`.

`AppendSymbol` – jak již bylo řečeno výše, úkolem této funkce je vytvořit z grafu nad slovem μ graf nad slovem μa. Idea je pomocí tzv. *aktivního bodu* projít graf a vykonat lokální změny v jeho okolí tam, kde jsou potřeba, přičemž změny se odehrávají v okolí sufixů (tedy listů nebo vnitřních vrcholů reprezentujících sufix) a jejich pravděpodobnost klesá s

```
void Build(String stringToAppend)
{
   CreateEmptyGraph();
   AppendString(stringToAppend);
}

void AppendString(String stringToAppend)
{
   for (symbol symbolToAppend in stringToAppend)
   {
      AppendSymbol(symbolToAppend);
   }
}

void AppendSymbol(Symbol symbolToAppend)
{
   ResetActivePointPosition();
   MoveActivePointDown(symbolToAppend);
   AddLastSuffixLink();
}

void MoveActivePointDown(Symbol symbolToAppend)
{
   while (!MoveActivePointDownIfPossible(symbolToAppend))
   {
      CreateEdgeFromActivePoint();
      MoveActivePointSideways();
   }
}
```

Výpis 1: Pseudokód konstrukčního algoritmu

krátícím se sufixem, proto aktivní bod postupuje od nejdelšího sufixu k nejkratšímu. Vrcholy reprezentující sufixy jsou propojeny sufixovými spojkami a tvoří tzv. *hraniční cestu*, po které aktivní bod postupuje a v okolí jeho aktuálního výskytu se vykonávají potřebné změny.

`ResetActivePointPosition` – funkce přemístí aktivní bod do nejdelšího sufixu slova μ (tj. posledního vrcholu páteře), aby bylo možné projít po hraniční cestě všechny sufixy. Ovšem pouze v případě trie, jelikož v případě stromu bychom pravidelným procházením všech listů ztratili možnost dosáhnout lineární časové složitosti. Proto Ukkonen zavedl označení *otevřené hrany* [5]. Otevřená hrana je hrana s *otevřeným popisem*, tzn. popis má pevný pouze levý index a pravý index ukazuje na konec řetězce, nad kterým se strom tvoří. K popisům všech hran vedoucích do listů se tedy nový symbol připojí automaticky a není potřeba je zvlášť upravovat, úpravy začnou až ve vrcholu reprezentujícím NNS(μ) a aktivní bod postupuje po hraniční cestě, dokud je potřeba. (Pozn.: Od této chvíle se nebudou v grafickém zobrazení grafů objevovat sufixové spojky mezi listy.)

`MoveActivePointDown` – funkce, která posouvá aktivní bod po hraniční cestě a zkouší v každém vrcholu nalézt a-hranu podle parametru funkce. Zkouší to pomocí funkce `MoveActivePointDownIfPossi` (která zjistí, zda z aktuálního implicitního či explicitního vrcholu vede a-hrana, a vrátí o tom informaci) a pokud uspěje, cyklus končí. Pokud neuspěje, je potřeba vytvořit novou hranu funkcí `CreateEdgeFromActivePoint`,
poté se aktivní bod posune pomocí sufixové spojky na další kratší sufix pomocí funkce `MoveActivePointSideways`. Pokud tímto postupem aktivní bod dospěje až do kořene a ani tam se nepodaří vhodnou hranu nalézt, vytvoří novou hranu a posune se pomocí sufixové spojky do botu. Z tohoto speciálního vrcholu vedou do kořene hrany se všemi symboly abecedy, takže se aktivní bod posune po hraně dolů zpátky do kořene a cyklus končí.

`AddLastSuffixLink` – funkce, která se opět liší v závislosti na tom, zda ji použijeme na trie nebo na strom. V sufixovém trie slouží k doplnění sufixové spojky mezi nejkratším unikátním sufixem (poslední vytvořený list) a NNS (aktuální poloha aktivního bodu), čímž je dokončena nově vzniklá hraniční cesta (mezi všemi kratšími sufixy již sufixové spojky jsou). V případě sufixového stromu je tento krok vynechán úplně, jelikož by bylo potřeba spojku v průběhu přidávání dalšího symbolu upravit, jelikož po prodloužení otevřených hran se stává nevalidní – nachází se mezi nesprávnými vrcholy (měla by být mezi dvěma sufixy). Navíc tato sufixová spojka není při konstrukci nikdy použita – otevřené hrany vedoucí do listů se upravují automaticky a úprava zbytku grafu začíná v NNS. Z obou důvodů je vhodné tuto spojku přidat až na samém konci konstrukce.

`MoveActivePointSideways` – funkce posune aktivní bod po sufixové spojce na další kratší sufix. Některé sufixové spojky v sufixovém stromu ale mohou být implicitní, takže v grafu fyzicky neexistují. Budeme je proto simulovat pomocí objektů, které v grafu přítomny jsou (explicitních vrcholů, hran a sufixových spojek).

4.1.1 Simulace sufixových spojek

Jak bylo zmíněno, komprese cesty mění vrcholy, které mají výstupní stupeň 1, na implicitní. Potřebujeme-li pak z těchto vrcholů provést posun do vrcholu reprezentujícího kratší sufix, musíme potřebnou sufixovou spojku nasimulovat z důvodu jejího virtuálního charakteru.

Je pochopitelné, že cíl sufixové spojky by byl jednoduše nalezitelný vyhledáním od kořene. Čas strávený vyhledáváním by byl ale přímo úměrný délce vyhledávaného řetězce, což by v kontextu celého konstrukčního procesu vyústilo v kvadratickou složitost, nikoli lineární, o kterou usilujeme. Proto je nutné najít lepší řešení a tím bude určitá aproximace přesunu přes implicitní sufixovou spojku. Pokusíme se posunout aktivní bod přes nějakou blízkou explicitní (= v grafu fyzicky existující) sufixovou spojku a poté jej dostat do implicitního vrcholu, do kterého by implicitní spojka vedla.

V dalších dvou sekcích jsou popsány dva různé přístupy k problému. První přístup využívá sufixovou spojku vedoucí z nejbližšího explicitního vrcholu *nad* aktuální pozicí aktivního bodu. Druhý přístup navazuje na první a vydává se opačným směrem – vyhledá sufixovou spojku vedoucí z nejbližšího explicitního vrcholu *pod* aktuální pozicí aktivního bodu (může se stát, že tímto vrcholem bude list) a použije ji.

4.1.1.1 Simulace přes horní sufixovou spojku

Aktivní bod se nachází v explicitním vrcholu, který byl právě vytvořen z implicitního vrcholu funkcí `CreateEdgeFromActivePoint` po rozdělení původní hrany a je posledním z řady takto změněných vrcholů, takže sám sufixovou spojku postrádá. Tu ale má počáteční vrchol původní (nyní rozdělené) hrany, takže ji využijeme. Pseudokód simulace sufixové spojky vidíme ve Výpisu 2 a opět si stručně vysvětlíme význam jednotlivých funkcí (Pozn.: Nyní a nadále již nebudou existovat odchylky mezi popisovanými algoritmy aplikovanými na sufixový trie a na sufixový strom. Budeme se proto zabývat již pouze sufixovým stromem).

`GetLabelAboveActivePoint` – funkce vrací popis té části původní hrany, která se nyní nachází nad nově vytvořeným vrcholem.

`MoveActivePointToEdgeInitialVertex` – funkce přesune aktivní bod do počátečního vrcholu původní hrany.

`MoveActivePointSidewaysExplicit` – funkce posune aktivní bod po explicitní sufixové spojce, která vede z počátečního vrcholu původní hrany.

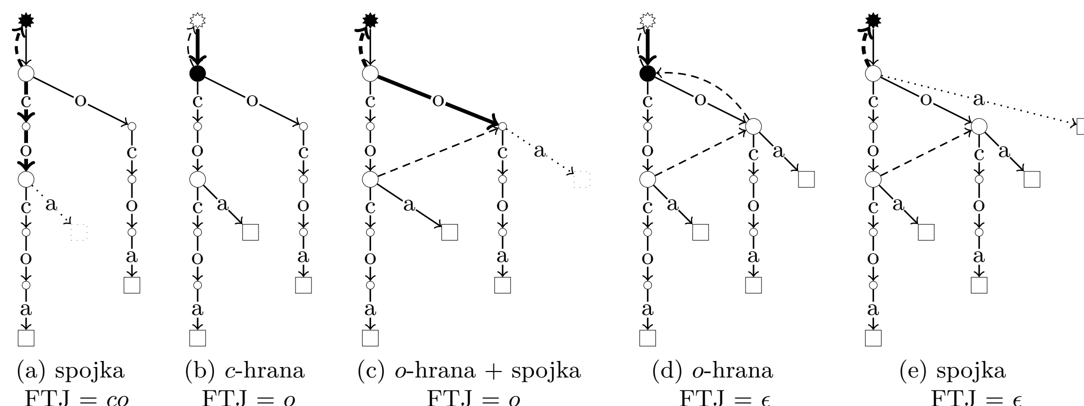
`ActivePointBranchAndJumpDownLoop` – po přesunu aktivního bodu po explicitní sufixové spojce je cílový vrchol někde pod aktuální pozicí aktivního bodu a bude nalezen pomocí uloženého popisu části původní hrany získaného funkcí `GetLabelAboveActivePoint`.

`ActivePointBranchAndJumpDown` – funkce nalezne podle prvního znaku poskytnutého řetězce odpovídající hranu (větvící operace). Pokud je délka této hrany menší než délka řetězce,

Výpis 2: Pseudokód simulace sufixové spojky – varianta horní sufixové spojky

```
void SimulateSuffixLinkAbove()
{
   string labelAbove = GetLabelAboveActivePoint();
   MoveActivePointToEdgeInitialVertex();
   MoveActivePointSidewaysExplicit();
   ActivePointBranchAndJumpDownLoop(labelAbove);
}

void ActivePointBranchAndJumpDownLoop(String factorToJump)
{
   do
   {
      ActivePointBranchAndJumpDown(factorToJump);
   } while (Length(factorToJump) > 0);
}
```



(a) spojka FTJ = *co* (b) *c*-hrana FTJ = *o* (c) *o*-hrana + spojka FTJ = *o* (d) *o*-hrana FTJ = ϵ (e) spojka FTJ = ϵ

Obrázek 2: Simulace sufixové spojky při zapracování symbolu a do sufixového stromu nad slovem *coco* (FTJ = factorToJump, viz Výpis 2)

skočí aktivní bod do cílového vrcholu hrany, jinak se provede skok pouze po takovém úseku hrany, jehož délka je rovna zbývající délce řetězce. Následně je z řetězce odebrán prefix, jehož délka je rovna délce skočené po hraně a otestuje se délka zbytku řetězce – v závislosti na ní je cyklus pokračuje dále nebo je ukončen.

Příklad simulace je Obrázku 2 (vysvětlivky značení jsou v Tabulce 2). V části (a) vidíme změnu implicitního vrcholu na explicitní, vytvoření nového listu a posun po sufixové spojce. V části (b) je první skok dolů do kořene a zkrácení podslova, které je potřeba přeskočit. V části (c) se vytváří opět explicitní vrchol z implicitního a nový list a posouváme aktivní bod po sufixové spojce. Část (d) ukazuje posun do kořene a další zkrácení přeskakovaného podslova. A konečně v části (e) je vytvořen poslední list v tomto kroku `AppendSymbol` a opět posuneme aktivní bod po sufixové spojce. (Pozn.: Podle definice vedou z botu do kořene hrany popsané

Výpis 3: Pseudokód simulace sufixové spojky – varianta spodní sufixové spojky

```
void SimulateSuffixLinkBelow()
{
   string labelBelow = GetLabelBelowActivePoint();
   MoveActivePointToEdgeTerminalVertex();
   MoveActivePointSidewaysExplicit();
   ActivePointJumpUpLoop(labelAbove);
}

void ActivePointBranchAndJumpDownLoop(String factorToJump)
{
   do
   {
      ActivePointJumpUp(factorToJump);
   } while (Length(factorToJump) > 0);
}
```

všemi symboly abecedy. V ukázkách a implementaci jsou tyto hrany z praktických důvodů nahrazeny jedinou hranou – při jejím průchodu se nekontroluje shoda symbolů, pouze se zkrátí přeskakovaný řetězec.)

4.1.1.2 Simulace přes spodní sufixovou spojku

Řešení popsané v předchozí sekci je jistě funkční, nicméně při bližším pohledu zjistíme, že je zbytečně komplikované. Komplikace je ve funkci `ActivePointBranchAndJumpDown` a spočívá v tom, že po každém skoku přes hranu je potřeba vykonat větvící operaci – vybrat mezi potomky vrcholu ten správný pro další postup. Pokud si dále uvědomíme, že ve zde použitých grafech mají vrcholy výstupní stupeň alespoň 2 (kromě listů) a vstupní stupeň právě 1 (kromě kořene), dojdeme k závěru, že postup přes spodní sufixovou spojku by měl být díky chybějící větvící operaci efektivnější a rychlejší.

Oba přístupy jsem nasimuloval. Simulace probíhala na stroji s procesorem Intel Pentium 2556 (1,7 GHz), operační pamětí 8 GB DDR3 RAM pod operačním systémem Windows 8.1. Simulace jsem prováděl přes vývojové prostředí Visual Studio Enterprise 2017 v konfiguraci x64/Release. K simulaci byly využity soubory z datového korpusu Lightweight (dostupný zde), jejichž krátké popisy jsou v Tabulce 3. Během simulace jsem z paměťových důvodů konstruoval sufixový strom nad slovem o délce pouze 2^{25} symbolů, po přečtení dalšího symbolu byla konstrukce zastavena. Simuloval jsem také rozdílné způsoby přístupu k procházení potomků. Ti jsou totiž uloženi ve spojových seznamech [1], proto jsem rovněž sledoval, zda bude rozdíl ve výsledku (a případně jak velký), když

[1]Pro uložení potomků se jiná datová struktura typu mapa či strom jeví vzhledem k režii okolo struktury a velice malému průměrnému počtu potomků, které je potřeba projít, než je nalezena správná hrana (viz Tabulku 4), oproti spojovému seznamu jako nevýhodná.

Tabulka 3: Soubory datového korpusu Lightweight – mohutnost abecedy, velikost a popis

Soubor	$\|\Sigma\|$	Velikost (B)	Popis
chr22	5	34 553 758	Lidský chromozom 22
etext99	146	105 277 340	Soubory projektu Gutenberg ETEXT99
gcc-3.0	150	86 630 400	Zdrojový kód GCC 3.0 (tar)
howto	197	39 422 105	Linuxové textové návody Howto
jdk13c	113	69 728 899	Dokumentace JDK 1.3 (HTML a java soubory)
linux-2.4.5	256	116 254 720	Zdrojový kód linuxu 2.4.5 (tar)
rctail96	93	114 711 151	Zprávy agentury Reuters (XML)
rfc	120	116 421 901	textové soubory RFC
sprot34	66	109 617 186	Databáze swiss prot
w3c	256	104 201 579	HTML stránky z www.w3c.org

Tabulka 4: Výsledky simulace sufixové spojky

Soubor	Horní spojka					Spodní spojka					
	Větvení	Základní		MTF		Větvení	Skoky	Základní		MTF	
		PH	Čas (s)	PH	Čas (s)			PH	Čas (s)	PH	Čas (s)
chr22	62 924 830	2,1	48,65	2,1	48,47	34 130 645	21 953 241	2,1	50,05	2,2	50,51
etext99	55 520 279	4,8	61,41	3,9	51,47	31 553 220	11 369 636	5,1	54,07	4,6	51,72
gcc-3.0	46 746 181	4,9	43,28	3,7	36,49	21 087 576	6 145 832	6	41,05	5,1	37,79
howto	52 416 037	6,4	61,43	4,7	49,67	28 370 242	9 268 409	6,8	54,40	5,8	49,79
jdk13c	37 150 606	2,8	21,39	2,6	20,08	12 554 499	1 644 089	4,5	23,65	4,3	23,04
linux-2.4.5	46 846 162	5,2	42,29	3,9	34,96	22 309 002	6 073 477	6,2	40,34	5,3	36,46
rctail96	43 181 363	4,4	36,83	3,8	33,47	21 140 144	4 430 462	5,3	35,6	5	35,07
rfc	48 775 890	5,1	47,65	3,9	40,78	23 197 033	7 166 722	6	43,27	5,2	40,77
sprot34	45 952 270	4,7	45,45	4	40,34	23 480 099	5 600 523	5,6	42,67	5,2	40,60
w3c	38 469 385	2,6	24,19	2,4	22,83	5 960 657	1 557 540	5,2	24,69	4,7	24,85

(a) potomci zůstávají stále ve stejném pořadí (Základní),

(b) nalezený potomek je vždy předřazen před prvního potomka (MTF = Move To Front).

Výsledky simulace vidíme v Tabulce 4. Hodnoty v tabulce jsou průměrem hodnot 3 nezávislých měření [2]. Zdrojový kód použitý při simulacích je uložen na přiloženém CD.

V tabulce můžeme vidět, že průměrný počet prohledaných hran (sloupec PH) se skutečně po použití přístupu MTF ve většině případů snížil a s ním i celkový čas konstrukce stromu, a to v obou variantách použitých sufixových spojek. Pokud dále porovnáme hodnotu PPH při MTF přístupu u obou variant spojek, zjistíme, že varianta horní spojky je lepší. Důvod je pravděpo-

[2] Větší část operací větvení připadne na postup aktivního bodu po hraně dolů z explicitního vrcholu ve funkci `MoveActivePointDownIfPossible`, proto jsem obě verze práce s potomky zařadil i do varianty posunu přes spodní spojku.

dobně ten, že aktivní bod si "předchystávhodnou hranu na počáteční pozici mezi potomky již při postupu po hraně dolů a proto ji posléze při simulaci spojky nalezne rychleji, a toto předchystávání má dokonce na výsledek ve většině případů větší vliv než postup přes spodní sufixovou spojku. Dále budeme tedy používat variantu horní spojky s MTF přístupem.

4.2 Posouvání stromu v posuvném okně

Poté, co jsme rozebrali první část kompresního algoritmu nad sufixovým stromem, totiž konstrukci stromu a simulaci implicitních sufixových spojek, přistoupíme ke druhému bodu – posouvání stromu v posuvném okně.

Posouvání sufixového stromu v posuvném okně je založeno na dvou základních operacích, a to smazání nejstaršího symbolu a připojení nového symbolu. Pomocí těchto dvou operací je také definováno Perfektní posouvání sufixového stromu po řetězci:

Definice 4 (Perfektní posouvání)
Algoritmus implementující perfektní posouvání sufixového stromu musí mít následující vstup a výstup:

Vstup – řetězec $\mu \in \Sigma^*$ a $w \in \mathbb{N}$ zvané *délka okna* takové, že $0 < w < |\mu|$.

Výstup – Posloupnost $G_1, G_2, ..., G_{|\mu|-w+1}$, kde G_1 je sufixový strom nad řetězcem $\mu[1 \mathinner{..} w]$ a $G_{i+1} =$ Smazání(Připojení(G_i), $\mu[i + w]$) je sufixový strom nad řetězcem $\mu[i+1 \mathinner{..} i+w]$.

Přirozeně lze navrhnout algoritmus, který operace smazání a připojení implementuje takovým způsobem, že celý stávající strom smaže a vytvoří úplně nový, nicméně nejen z výkonových důvodů je takový algoritmus nepřijatelný, proto hledáme jinou cestu.

Operace smazání a připojení je možné zapojit do algoritmu načrtnutého ve Výpisu 1 způsobem uvedeným ve Výpisu 4. Operace mazání musí být vždy provedena před operací připojení – před připojením dalšího symbolu je nejprve potřeba uvolnit mu místo v posuvném okně. Pro úplnost rovněž zde uvádím stručné vysvětlení činnosti jednotlivých funkcí z Výpisu 4:

`Slide` – funkce nejprve vytvoří sufixový strom nad první části poskytnutého řetězce o délce `windowSize` (postrádá smysl začít posouvat strom, dokud není celé posuvné okno zaplněno – posouvání přidává konstrukčnímu algoritmu na náročnosti, proto čím později se začne strom posouvat, tím lépe) a poté pomocí funkce `SlideOverString` provede posouvání stromu nad zbývající částí řetězce.

`SlideOverString` – funkce posouvá sufixový strom po řetězci. Činí tak voláním funkce `SlideOverSymbol` pro každý symbol řetězce.

`SlideOverSymbol` – funkce posune sufixový strom přes jeden symbol. Využívá k tomu zmíněných operací smazání a připojení a to prostřednictvím funkcí `DeleteOldestSymbol` a `AppendSymbol`. Po posunutí stromu po řetězci ještě nutné upravit popisy hran, což provede funkce `UpdateEdgeLabels`.

```
void Slide(String stringToSlide, int windowSize)
{
   Build(stringToSlide.substr(1, windowSize));
   SlideOverString(stringToSlide.substr(windowSize + 1,
      Length(string)));
}

void SlideOverString(String stringToSlide)
{
   for (symbol in stringToSlide)
   {
      SlideOverSymbol(symbol);
   }
}

void SlideOverSymbol(Symbol symbol)
{
   DeleteOldestSymbol();
   AppendSymbol(symbol);
   UpdateEdgeLabels();
}
```

Výpis 4: Posouvání sufixového stromu v okně

`DeleteOldestSymbol` – funkce smaže nejstarší symbol (první zleva) z posuvného okna a upraví sufixový strom odpovídajícím způsobem.

`AppendSymbol` – funkce je ekvivalentem stejnojmenné funkce ve Výpisu 1 – připojí nový symbol zprava do posuvného okna a upraví sufixový strom.

`UpdateEdgeLabels` – v sufixovém stromu jsou popisy hran zastoupeny dvěma indexy do posuvného okna. A jelikož se okno posouvá, je potřeba indexy aktualizovat, aby zůstaly platné.

4.2.1 Smazání nejstaršího symbolu

Abychom našli efektivní algoritmus pro odstranění nejstaršího symbolu, prozkoumejme nejprve, jaký vliv bude mít smazání nejstaršího symbolu na sufixový strom.

V první řadě se podívejme, co se stane s množinou všech podslov podléhajícího řetězce, smažeme-li z něj první symbol. (Pozn.: Symbol $\dot\cup$ označuje sjednocení množin s prázdným průnikem.)

Tvrzení 1 (Změna množiny podslov při mazání)
Nechť $\mu \in \Sigma^*$ a $c \in \Sigma$, potom

$$\text{Podslovo}(c\mu) = \text{Podslovo}(\mu) \,\dot\cup\, \text{UnikátníPrefix}(\mu).$$

Hledáme tedy obsah množiny UnikátníPrefix(μ). Zde si pomůžeme srovnáním s prvky množiny UnikátníSufix(μ) – unikátní sufixy jsou shluknuty do skupiny listů. Unikátní prefixy jsou také shluknuty, a to podle následujícího tvrzení:

Tvrzení 2 (Prefixový blok)
Nechť $\mu \in \Sigma^*$. Potom existuje takové $k \in \mathbb{N}$, kde $0 \leq k \leq |\mu|$, že

$$\text{UnikátníPrefix}(\mu) = \{\alpha; \alpha \in \text{Prefix}(\mu) \wedge |\alpha| \geq k\}.$$

Unikátní prefixy se tedy nachází na konci páteře (nejkratší unikátní prefix je vzdálen od kořene k implicitních hran) a ten by měl být odebrán. Jelikož se ale zabýváme takovým typem grafu, na němž je aplikována komprese cesty (vrcholy grafu jsou pouze explicitní podslova), toto zjištění nestačí – nevíme, kolik vrcholů je potřeba odebrat. Je tedy nutné se ještě podívat na změny v množině explicitních podslov při mazání nejstaršího symbolu. Připomeňme, že množina explicitních podslov se skládá z prázdného slova, zprava větvících podslov a unikátních sufixů. A v posledních dvou množinách mohou nastat změny.

Tvrzení 3 (Změna množiny zprava větvících podslov při mazání)
Nechť $\mu \in \Sigma^*$ a $c \in \Sigma$, potom

$$\begin{aligned}\text{Větvící}^P(c\mu) = \text{Větvící}^P(\mu)\ \dot\cup \\ \{\alpha; \text{Kontext}^P_{c\mu}(\alpha) = \{a, b\} \wedge a \neq b \wedge \alpha b \in \text{UnikátníSufix}(c\mu)\}.\end{aligned}$$

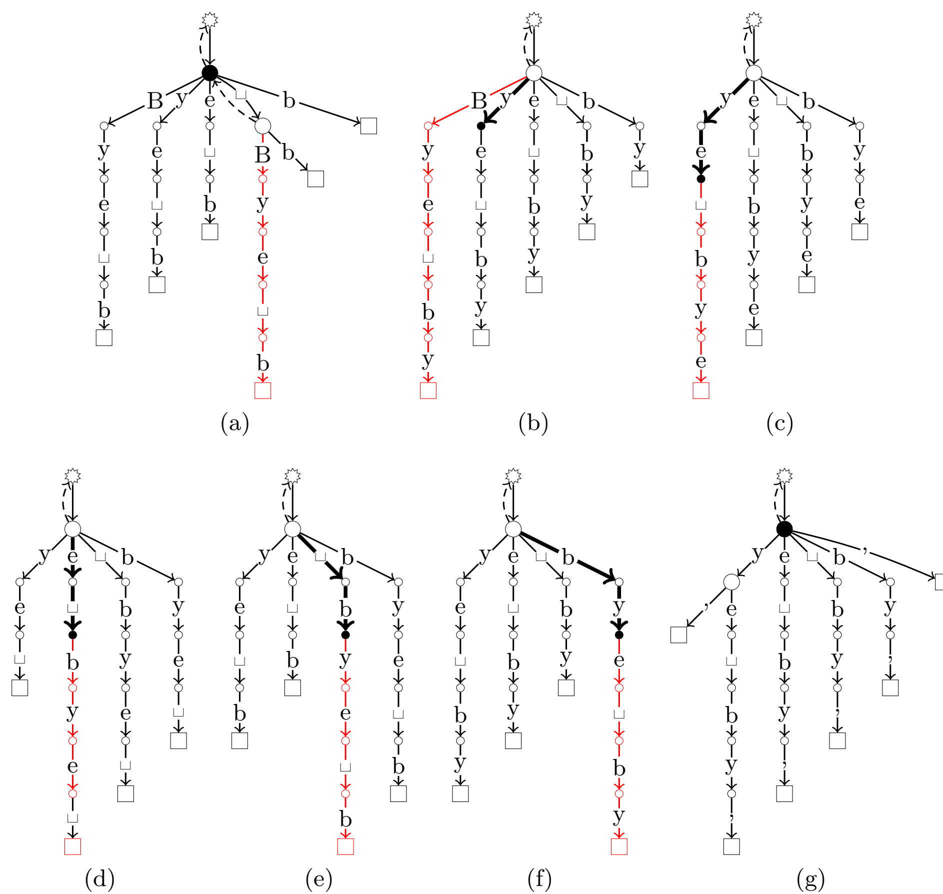
Z tvrzení plyne, že ubyde nejvýše jedno větvící podslovo. Toto podslovo je NNP($c\mu$) a má právě dva pravé kontexty – druhý výskyt se po odebrání symbolu c a připojení svého pravého kontextu b stane nejkratším unikátním sufixem.

Tvrzení 4 (Změna množiny unikátních sufixů při mazání)
Nechť $\mu \in \Sigma^*$ a $c \in \Sigma$, potom

$$\begin{aligned}\text{UnikátníSufix}(\mu)\ \dot\cup\ \{c\mu\} &= \text{UnikátníSufix}(c\mu)\ \dot\cup \\ &\quad \{\alpha; \alpha \in \text{Prefix}(c\mu) \cap \text{Sufix}(c\mu) \wedge |\text{Výskyt}^P_{c\mu}(\alpha)| = 2\} \\ &= \text{UnikátníSufix}(c\mu)\ \dot\cup \Big(\{\text{NNP}(c\mu)\} \cap \{\text{NNS}(c\mu)\}\Big)\end{aligned}$$

Stejně jako v případě Tvrzení 3, změny v množinách nejsou velké, opět ubyde nejvýše jeden sufix. Lépe řečeno jeden sufix ubyde jistě (nejdelší sufix $c\mu$) a jeden může přibýt (podslovo, které je zároveň NNP($c\mu$) a NNS($c\mu$) a po odstranění symbolu c se stane nejkratším unikátním sufixem).

Situace, kdy sufix přibyde, nastane právě tehdy, když se aktivní bod po připojení symbolu nachází na poslední hraně páteře (poslední větvení se musí nutně nacházet blíže ke kořeni, jelikož se toto podslovo v celém slově $c\mu$ vyskytuje právě dvakrát a onen druhý případ je právě zpracováván) právě v implicitním vrcholu reprezentujícím jak NNP($c\mu$), tak i NNS($c\mu$).



Obrázek 3: Posouvání stromu s posuvným oknem délky 6 nad řetězcem "␣Bye␣bye␣by,". Červeně označené části stromů jsou určeny k odstranění. (Legenda značení je v Tabulce 2.)

Nyní, když jsme získali informace o změnách, které nastanou při mazání nejstaršího symbolu, transformujme tyto poznatky do algoritmu. Víme, že začneme mazat od nejdelšího (nejstaršího) listu z konce páteře a že chceme smazat pouze unikátní prefixy, tedy vrcholy až do nejdelšího neunikátního prefixu. Tím je ale díky kompresi cesty vrchol právě nad listem – explicitní nebo implicitní s aktivním bodem.

V prvním případě je smazán nejdelší sufix i s jeho vstupní hranou. Pokud se rodičovský vrchol tímto krokem stane nevětvícím, stává se zároveň implicitním a obě s ním incidentní hrany se spojí do jediné. Ve druhém případě je vrchol změněn na explicitní a stane se novým koncový vrcholem (zde listem) hrany, která vedla do nejdelšího sufixu. Nyní je ještě potřeba posunout aktivní bod po sufixové spojce na nový nejdelší neunikátní sufix.

Příklad posouvání sufixového stromu je na Obrázku 3 (vysvětlivky značení se nacházejí v Tabulce 2).

4.2.2 Úprava popisů hran

Jakmile jsou dokončeny všechny strukturální změny ve stromu, je potřeba ještě upravit popisy hran. Důvodem je fakt, že díky kompresi cesty nejsou popisy hran dány pevně, ale jsou určeny indexy v posuvném okně.

Samozřejmě nebudeme celý komprimovaný řetězec nahrávat do paměti (mohli bychom v případě souboru o velikosti několika GB rychle skončit), ale budeme paměť určenou pro uložení řetězce využívat cyklicky znovu. Pak je ale jasné, že když jsou symboly přepisovány novými symboly a indexy směřující do řetězce zůstanou stejné, popisy hran se mohou nenadále měnit. Proto je potřeba zavést mechanismus na obnovování indexů.

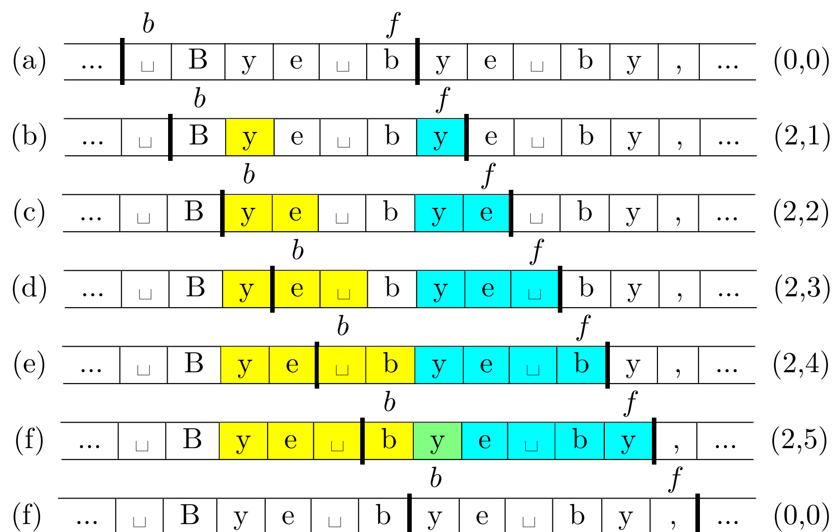
Nejjednodušší a nejpřímější cesta je jednorázová úprava hran celého grafu. Algoritmus provede průchod grafem do hloubky, přičemž hledá v každém podstromu nejnovější list a levý index podslova, které reprezentuje, pak vynáší do vyšších pater stromu. Pro minimalizaci časové náročnosti se tak děje pouze nezbytně často, totiž jednou za w znaků (w je délka posuvného okna), a proto je potřeba zvětšit velikost paměti pro uložení řetězce (konkrétně na $2w$ znaků), aby nedošlo k přepsání znaku, který je stále využíván.

4.3 Vyhledání shod v textu a komprese

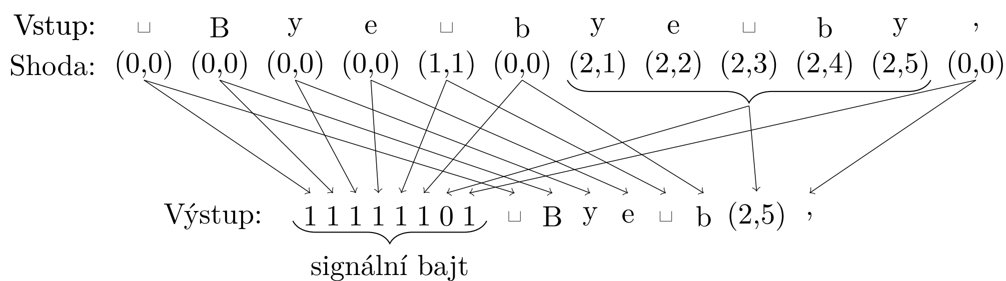
V minulých dvou sekcích, které se zabývaly algoritmem konstrukce sufixového stromu nad částí řetězce a jeho posouváním po zbytku řetězce, byl důvod uvažovat o algoritmu a jeho efektivitě. V této části pro to důvod není, protože myšlenka je velice jednoduchá – stejně jako v algoritmu LZ77 hledat prefix nezpracované části vstupního řetězce v již zpracované části (tento prefix samozřejmě může být delší než délka okna a může zasahovat do dosud nezpracovaného textu, viz Obrázek 4 – dvojice čísel v závorce znamená (pozice shody, délka shody)), což je v sufixovém stromu snadná věc – u každé hrany je možné okamžitě zjistit, jaký je levý index jejího popisu a podle tohoto indexu vybrat z paměti řetězce symbol a porovnat jej s prvním symbolem nezpracované části.

Při vyhledávání řetězce ve stromu de facto pouze využíváme skutečnosti, že aktivní bod při přidávání dalšího symbolu do stromu vyhledává podslovo začínající shodným symbolem a tím nalezne také index shody. My pouze tohoto nálezu využijeme a uložíme jej. Není potřeba vyhledávat shodu od kořene, i když by to samozřejmě bylo možné a nalezli bychom shodu v zaručeně lineárním čase.

Ze začátku tedy hledáme v sufixovém stromu shodné podslovo délky jedna a pokud existuje vhodná cesta po hranách dále dolů (další symboly v obou částech řetězce jsou stejné), prodlužujeme nalezené podslovo posouváním aktivního bodu po hranách této cesty během dalších konstrukčních kroků, dokud se aktivní bod z důvodu absence vhodné hrany neposune po sufixové spojce na kratší sufix nebo není dosažena maximální délka shodného prefixu. V takových případech shodu vhodným způsobem uložíme a začneme hledat další.



Obrázek 4: Posuvné okno délky 6 nad řetězcem "␣Bye␣bye␣by,"– vyhledání shody a její překryv do nezpracované části řetězce.



Obrázek 5: Komprese řetězce "␣Bye␣bye␣by,"

Hledání prefixu ve stromu samozřejmě nemůže začít, pokud je aktivní bod v botu (tzn. byl zpracován symbol, který se v posuvném okně nevyskytoval), jelikož z botu vedou hrany se všemi symboly abecedy, takže by hledání postrádalo smysl. Nejprve se tedy aktivní bod musí posunout do kořene a až poté můžeme začít hledat další shodu.

Na Obrázku 4 vidíme příklad posouvání okna s vyhledáním shody. Na Obrázku 3 je využito stejné slovo je pro ilustraci posouvání stromu. Můžeme si všimnout, že prodlužování shody pokračuje i tehdy, když se okno posune za první pozici shody – tedy první symbol shody je z posuvného okna vymazán. Shodu můžeme takto prodlužovat i dále, dokud se nezmění pozice shody. Jakmile se pozice změní, nemáme již možnost ověřit, zda se v původním okně na změněné pozici vyskytoval shodný symbol. Proto shodu raději ukončíme.

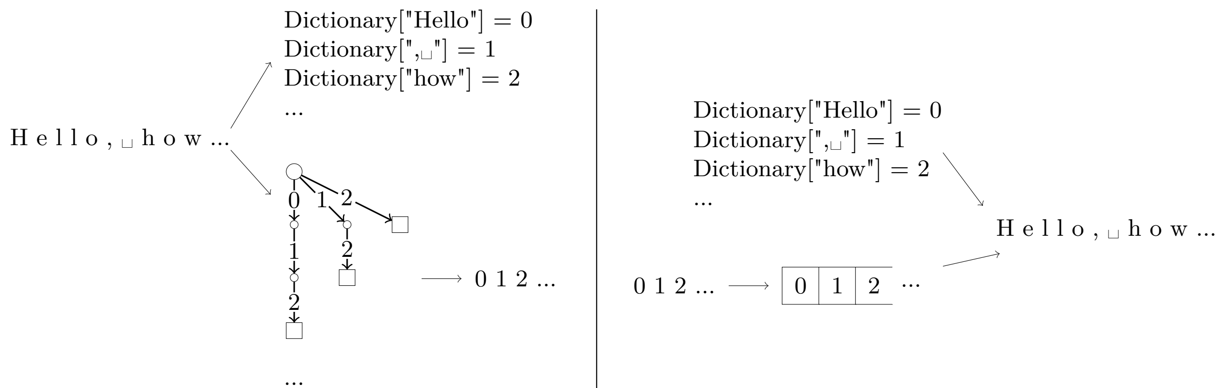
Co se týče komprese, je jasné, že potřebujeme v průběhu zpracování vstupního řetězce ukládat data, ze kterých bude možné posléze při dekompresi sestavit původní řetězec. V našem případě jsou těmito daty nalezené shody a samostatné symboly, a to v případech, pokud shoda nalezena není nebo pokud má shoda kratší délku, než kolik samostatných symbolů by bylo možné uložit na místo v paměti, které by shoda zabrala. Shody a symboly se posléze rozdělí do osmic a každé z nich se předřadí jeden signální bajt, ve kterém každý bit říká o úseku osmice, který mu pořadím odpovídá, zda se jedná o symbol (hodnota 1) či shodu (hodnota 0). Tato skupina (signální bajt + 8 shod či symbolů) se poté vypíše do kompresního souboru.

Příklad komprimovaného výstupu je na Obrázku 5. Všimněme si, že při druhém výskytu mezery na vstupu byla nalezena shoda na pozici 1. Další symbol na vstupu byl ale b, shoda se tedy neprodloužila. Protože ale zakódovaná shoda zabírá alespoň 1 bajt, tedy stejně jako jeden znak, je zbytečné vypisovat shodu, neboť neušetříme žádné místo a navíc přímé vypsání symbolu při dekompresi je jistě rychlejší než proces kolem vypisování symbolů shody, vypíšeme tedy symbol.

Při dekompresi není potřeba konstruovat sufixový strom, proto je podstatně rychlejší než komprese. Vystačíme si totiž pouze s posuvným oknem stejné velikosti, jakou mělo posuvné okno při kompresi, aby bylo možné zrcadlit shody na stejné pozice.

4.4 Komprese nad slovy

Komprese nad slovy funguje na jednoduchém principu – převodu slov na číselné identifikátory. Ze vstupního souboru se čtou symboly jeden po druhém tak dlouho, dokud se na vstupu neobjeví bílý znak. Lépe řečeno čteme znaky tak dlouho, dokud jsou alfanumerické – bílé znaky jsou se speciálními symboly v jedné skupině. Jakmile na vstup dorazí znak, který není alfanumerický, sekvence přijatých alfanumerických znaků utvoří jedno slovo (všechny nealfanumerické znaky příchozí za sebou také vytvoří samostatné slovo) a umístí se do slovníku s jedinečným číselným identifikátorem. Tento identifikátor je posléze použit v kompresi. Komprese probíhá klasickým způsobem nad sufixovým stromem, ve kterém vrcholy reprezentují číselné posloupnosti. Slovník je uložen zvlášť do souboru.



Obrázek 6: Komprese a dekomprese nad slovy

Při dekompresi se pořadí operací obrátí. Nejprve se ze souboru načte převodní slovník a poté se z druhého souboru začnou číst čísla, která se po zpracování v posuvném okně a vynětí z něj převedou pomocí slovníku zpět na slova a ta se vypíšou do dekomprimovaného souboru.

Postup komprese a dekomprese nad slovy je přehledně zobrazen na Obrázku 6.

5 Implementace

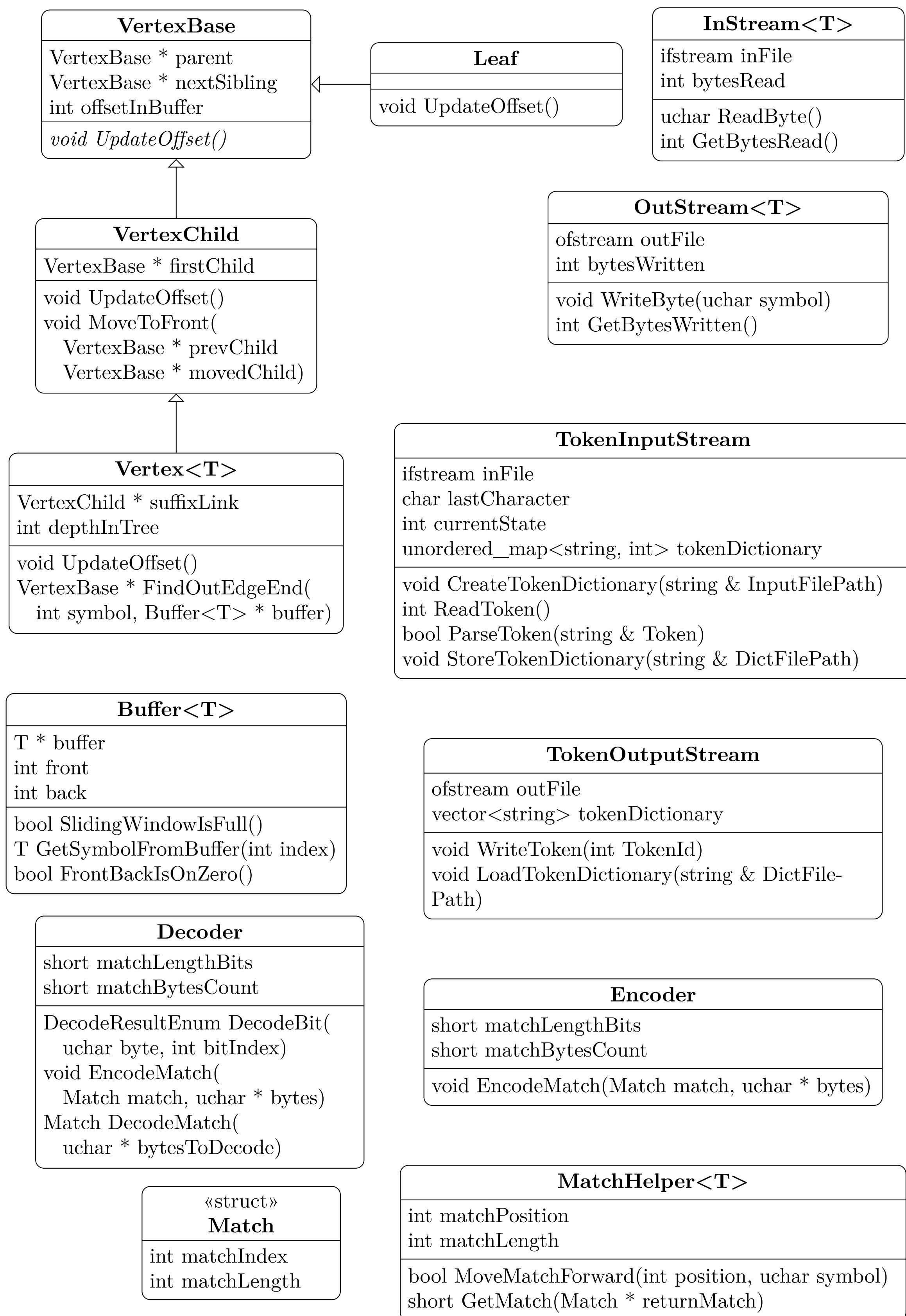
V této kapitole prezentuji implementaci kompletního algoritmu popsaného v Kapitole 4. Nejprve uvádím diagram tříd, do kterých jsem rozdělil kód, a poté podrobnější popis každé z tříd.

5.1 Diagram tříd

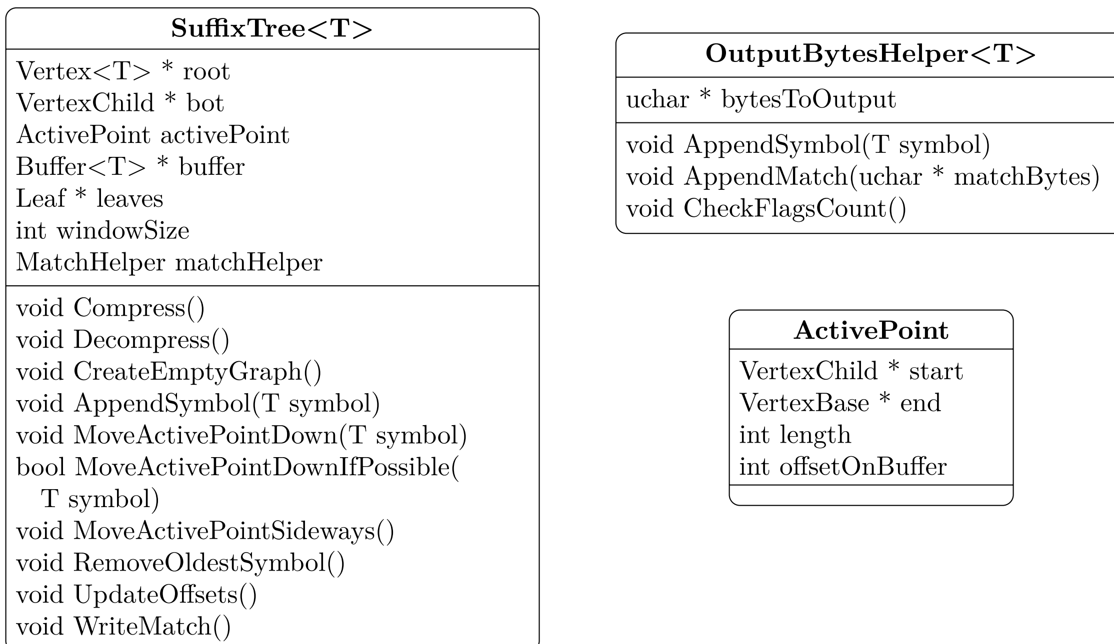
Diagram tříd, do kterých jsem kód aplikace rozdělil, je na Obrázku 6 a Obrázku 7. Jedná se celkem o 14 tříd a 1 strukturu. Některé třídy mají pouze podpůrný charakter (`InStream`, `OutStream`, `Buffer`, `OutputBytesHelper`), jiné naopak řeší klíčovou logiku aplikace (`SuffixTree`, `MatchHelper`). U každé třídy jsou vypsány význačné funkce.

Všimněme si, že některé třídy jsou šablony (mají v názvu `"<T>"`). Šablony jsou využívány z důvodu komprese nad slovy (viz Sekce 4.4). V šabloně se předpokládá vložení celočíselného datového typu, tedy některého z typů `char`, `short`, `int` nebo `long long` (či jejich neznaménkové varianty). Co se týče programátorského pohledu na věc, můžeme ke kompresi nad slovy přistoupit dvěma způsoby:

- Zadání datového typu svěříme aplikačnímu programátorovi a jeho znalostem vstupního souboru a velikosti vstupní abecedy (tj. jaká je šíře slovní zásoby obsažené v souboru). Potom můžeme číst ze souboru znak po znaku, vytvářet ze znaků slova a ta pak bez obav z přetečení datového typu převádět na čísla. Pokud aplikačního programátora nemáme a nemáme ponětí o možné různorodosti slov, musíme odhadnout datový typ podle odbornosti a délky zpracovávaného textu (všechny články encyklopedie Wikipedia budou pravděpodobně obsahovat větší množství unikátních slov než např. všechny články novin The New York Times za celou historii). V extrémním případě zvolíme datový typ `unsigned long long` (18 trilionů možných slov), v méně extrémním datový typ `unsigned int` (s velkou pravděpodobností nepřekročíme 4 miliardy unikátních slov). Na kratší soubory postačí datový typ `unsigned short` s možností 65 536 unikátních slov (všechny typy jsou neznaménkové pro zvětšení pozitivního rozsahu). Při použití tohoto přístupu je potřeba projít vstupní soubor pouze jednou.
- Nejprve zpracujeme celý vstupní soubor a zjistíme, kolik unikátních slov se v něm vyskytuje. Podle počtu unikátních slov pak vybereme vhodný datový typ, z jehož rozsahu nejvyšší hodnota je nejbližší vyšší od počtu unikátních slov a slova začneme zpracovávat a hodnoty jim přiřazovat až poté. Tento přístup vyžaduje dvojí přečtení vstupního souboru, ale ne jeho znalost, proto je použit v implementaci.



Obrázek 7: Třídní diagram (část 1.)



Obrázek 8: Třídní diagram (část 2.)

5.2 Popis tříd

5.2.1 TokenInputStream, TokenOutputStream

Dvě třídy, které jsou využívány, pokud se jedná o kompresi nad slovy. Třída `TokenInputStream` vytvoří podle obsahu souboru převodní slovník mezi slovy a čísly (`CreateTokenDictionary`) a poté čte slova ze souboru (`ParseToken`) a převádí je na číselné identifikátory. Kompresi nad slovy lze zatím smysluplně provádět nad soubory obsahujícími znaky, které lze zakódovat v kódování Unicode do 2 bajtů. Ve funkci `ReadChar` se totiž rozeznává konec slova podle toho, zda je široký znak (`wchar_t`) alfanumerický (zjistí se pomocí funkce `iswalnum`). Tato funkce rozpozná jako alfanumerické všechny zaky, které skutečně jsou alfanumerické a jejich kódy jsou v rozmezí hodnot 0x00 – 0xFF. Funkce `ParseToken` tedy správně rozpozná slova v angličtině, španělštině nebo němčině a ostatních jazycích využívajících kódování ISO 8859-1. Má ale problém rozpoznat např. české nebo polské speciální znaky. Na běh programu ale tento nedostatek nemá zásadní vliv – program bude fungovat, pouze slova nebudou správně rozpoznána. Navíc soubory ze standardních korpusů použité pro testování jsou převážně v angličtině, problém v rozpoznávání slov v nich by tedy neměl nastat a tento stav je pro experimentální zhodnocení postačující. Třída `TokenOutputStream` během dekomprese pouze načte převodní slovník a podle číselných identifikátorů (indexů do vektoru) vrátí odpovídající slova.

5.2.2 InStream, OutStream

Jsou to třídy pro znakový přístup k souboru – přistupují k souboru po jednotlivých bytech (pracují s datovým typem `unsigned char` přejmenovaným na `uchar`). Pokud tedy zvolíme znakovou kompresi, užívá se pouze těchto dvou tříd, a pokud kompresi nad slovy, přidávají se i výše zmíněné třídy ze skupiny `TokenStream` – při kompresi na úrovni slov čteme pomocí třídy `TokenInStream` a zapisujeme přes třídu `OutStream`, při dekompresi čte třída `InStream` a zapisuje třída `TokenOutStream`.

5.2.3 OutputBytesHelper

Tato třída se stará o předchystávání výstupu – shromažďování skupin po osmi objektech (shoda nebo symbol) a vytváření signálních bajtů – tak, jak je to naznačeno na Obrázku 5. Nové symboly a shody jsou do osmic připojovány pomocí funkcí `AppendSymbol` a `AppendMatch`. Po každém přidání je zavoláním funkce `CheckFlagsCount` zkontrolován počet objektů připravených k výpisu a pokud je jich 8, jsou jejich bajty vypsány.

Jednotlivé osmice jsou v bajtech uloženy v poli `bytesToOutput`. Shody do funkce `AppendMatch` přicházejí už jako sekvence bajtů, se symboly je to ale komplikovanější. Na Obrázku 8 vidíme, že funkce `AppendSymbol` je také šablona – dopředu nevíme, jaký datový typ budeme zpracovávat. Víme ale, že bude celočíselný a že bude mít 1, 2, 4 nebo 8 bajtů. Jazyk C++ zná způsob, jak v binárním módu vypsat číslo do souboru binárně (tedy po bajtech, nikoli číslicemi), nicméně v kontextu toho, že se zde k souboru nepřistupuje přímo, je vhodnějším přístupem rozdělit symbol na jednotlivé bajty a ty postupně uložit od nejvyššího po nejnižší na nejbližší volnou pozici do pole `bytesToOutput`, jak je to naznačeno v následujícím výpisu.

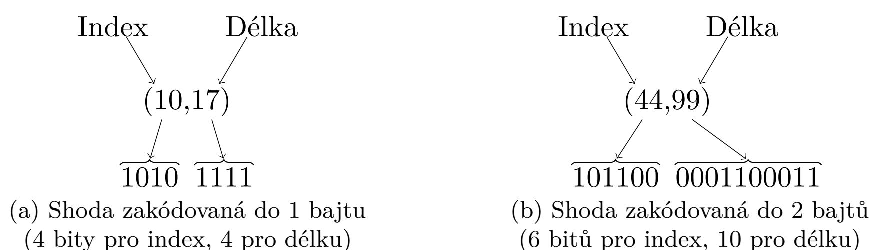
```
for (short i = sizeof(T) - 1; i >= 0; i--)
{
  bytesToOutput[bytesToOutputCount++] = (symbol >> i * 8) & 0xff;
}
```

Výpis 5: Zakódování symbolu do pole bajtů

5.2.4 VertexBase a odvozené třídy

Základní třída všech vrcholů v grafu je třída `VertexBase`. Obsahuje odkaz na rodiče, dalšího sourozence a levý index jedné z pozic podslova, které daný vrchol reprezentuje, v paměti řetězce (nikoli v posuvném okně).

`VertexChild` je třída odvozená z třídy `VertexBase` – tohoto typu jsou vrcholy, které mají potomka, tedy bot a vnitřní vrcholy. Proti základní třídě obsahuje třída ještě odkaz na prvního potomka (potomci jsou z výkonových důvodů uložení ve spojovém seznamu – rodič ukazuje na prvního potomka, první potomek na druhého atd.) a implementaci metody pro úpravu pozice



(a) Shoda zakódovaná do 1 bajtu (4 bity pro index, 4 pro délku)

(b) Shoda zakódovaná do 2 bajtů (6 bitů pro index, 10 pro délku)

Obrázek 9: Kódování shody

(vyhledá mezi listy podstromu, jemuž je tento vrchol kořenem, nejnovější pozici a tu propaguje do vyšších pater stromu).

Třída `Vertex` reprezentuje pouze vnitřní vrcholy stromu. Proti bázové třídě má navíc sufixovou spojku a hloubku vrcholu ve stromu, která udává, jak dlouhé je podslovo reprezentované vrcholem. Součet s proměnnou `offsetInBuffer` pak říká, jaký je pravý index popisu vstupní hrany v paměti řetězce.

`Leaf` – listový vrchol stromu, třída odvozená z `VertexBase`. Nemá potomka, hloubku ve stromu (v případě listu není hloubka v implementační části zapotřebí) ani sufixovou spojku (listy je nepotřebují, protože díky otevřeným hranám je možné při přidávání symbolu do stromu unikátní sufixy přeskočit a implicitní spojky se simulují pomocí horních spojek). Má pouze metodu, která vrací levý index jím reprezentovaného sufixu v paměti řetězce (nemá žádné potomky).

5.2.5 MatchHelper

`MatchHelper` je třída, která uchovává právě zpracovávanou shodu. Proměnné `matchPosition` a `matchLength` udávají, na jaké pozici v posuvném okně bylo nalezeno podslovo shodné s prefixem nezpracované části řetězce a jakou má aktuálně délku. Pokaždé, když se aktivní bod posune po hraně dolů, je zavolána funkce `MoveMatchForward`. Tato funkce zkontroluje, zda nedošlo k překročení maximální délky shody. Pokud k němu nedošlo, inkrementuje délku shody (případně nastaví pozici nové shody), pokud ano, vrátí shodu v podobě struktury `Match`.

5.2.6 Encoder

Třída `Encoder` má na starosti kódování shod do bajtů, aby bylo možné je vypsat. V metodě `EncodeMatch` se podle zadané maximální délky shody, resp. podle počtu bitů určených pro délku shody, index shody posune bitově doleva a doplní se délkou shody použitím logického součtu, viz Obrázek 9.

Pokud má tedy posuvné okno velikost např. 2^3 symbolů, vyhradíme pro index symbolu 4 bity (index může nabývat hodnot $0 - 2^4$-1 – kvůli opožděné úpravě hran musíme indexovat do dvojnásobně dlouhého pole než je délka posuvného okna), pro délku shody tedy zbývají také 4 bity, viz Obrázek 9a. V implementaci je rovněž zavedeno, že pokud na délku shody zbydou nejvýše 3 bity, automaticky je dodáno dalších 8 bitů, protože při kompresi na úrovni znaků je

maximální délka shody 7 znaků dosti nízká. Tedy např. pro 6 bitů vyhrazených pro pozici shody nepřipadají 2 bity pro délku shody, ale rovnou 10 bitů, viz Obrázek 9b.

5.2.7 Decoder

Tato třída má dva úkoly – dekódovat bity signálního bajtu a vracet informaci o tom, zda bit zastupuje symbol nebo shodu, a dekódovat shodu z bajtů.

5.2.8 Buffer

Třída, která se v aplikaci stará o správu paměti, kde je uložen řetězec (`buffer`), a má na starosti i posuvné okno – udržuje jeho začátek a konec (`front` a `back`) a změnami těchto dvou indexů s oknem pohybuje. Ověřuje, zda posuvné okno není plné (`SlidingWindowIsFull`) či zda se jeho okraje nenacházejí na pozicích, kdy je potřeba provést úpravu indexů popisů hran (`FrontBackIsOnZero`) a také se stará o čtení z paměti a zápis do ní.

Jak bylo nastíněno v Sekci 4.2.2, je kvůli hromadné úpravě popisů hran potřeba, aby velikost pole `buffer` byla dvojnásobná ku velikosti posuvného okna, aby bylo možné využívat paměť cyklicky a přitom za žádných okolností nedošlo k přepsání stále používaného symbolu.

5.2.9 ActivePoint

Tato třída reprezentuje aktivní bod, který při konstrukci sufixového stromu prochází grafem a v jehož blízkosti se dějí strukturální změny grafu. Při pohybu v grafu funguje aktivní bod následujícím způsobem: `start` a `end` jsou explicitní vrcholy (`start` je bot nebo vnitřní vrchol, `end` je vnitřní vrchol či list). Pokud aktivní bod právě putuje po implicitní cestě, jeho délka (proměnná `length`) se inkrementuje s každou překročenou implicitní hranou (každá implicitní hrana má délku 1). Po každém překročení hrany kontrolujeme, zda délka aktivního bodu není rovna délce hrany – rozdílu v hloubkách počátečního a cílového vrcholu (neplatí u listu – do listu aktivní bod nikdy dorazit nemůže). Pokud jsou si délky aktivního bodu a hrany rovny, nastavíme délku aktivního bodu na 0 a v dalším kroku se vrcholu `end` přiřadí vrchol na konci hrany, jejíž počáteční symbol je přečten na vstupu.

V případě, že odpovídající hrana neexistuje, přesune se aktivní bod po sufixové spojce vrcholu `start`. V případě, že je délka aktivního bodu nenulová, musíme spojku nasimulovat, viz Sekci 4.1.1 a Obrázek 2. Simulace spojek je ale oproti pseudokódu drobně zjednodušená – namísto zjišťování označení hrany pracujeme pouze s délkou aktivního bodu a tu také porovnáváme s délkou hrany při skocích po hranách dolů.

5.2.10 SuffixTree

Ústřední třída programu, která obstarává kompresi i dekompresi. Právě v této třídě je obsažen kód, který je uveden ve výpisech kódu Výpis 1, Výpis 2 a Výpis 4, obstarávající konstrukci

sufixového stromu včetně simulace sufixových spojek a jeho posouvání po vstupním řetězci, a to v metodě `Compress` a dalších metodách, které metoda `Compress` volá.

Metoda `Decompress`, jak napovídá její název, zařizuje nutné kroky ohledně dekomprese. Metoda cyklicky získává z komprimovaného souboru signální bajty a podle hodnoty bitů v nich obsažených získává ze souboru symboly nebo bajty shody. Zde se již, na rozdíl od třídy `OutputBytesHelper` (viz Sekci 5.2.3), k souboru přistupuje přímo bez shromažďujícího pole, takže se symboly čtou pomocí následující konstrukce (je potřeba mít otevřený soubor v binárním režimu).

```
T symbol;
inFile.read(reinterpret_cast<char *>(&symbol), sizeof(symbol));

if (!inFile.eof())
{
   this->bytesRead += sizeof(T);
}
```

Výpis 6: Přečtení symbolu v binárním módu

Pokud funkce právě zpracovává shodu, cyklicky vybírá z `Bufferu` z pozice určené indexem shody symbol a vkládá jej na konec `Bufferu` tolikrát, jak dlouhá je shoda. Mezi každými dvěma vloženými symboly se samozřejmě posuvné okno přemístí o jednu pozici vpravo.

V metodě `Decompress` je obsažen i kód pro hledání a úpravu shod. Shodu nalezneme, kdykoli se ve funkci `MoveActivePointDownIfPossible` posuneme po hraně dolů. Pokud má shoda před posunutím nenulovou délku, proběhne větvení a pozici shody je potřeba upravit na novou hodnotu, která se vypočítá z indexu nového vrcholu `end` aktivního bodu. Pozici shody musíme rovněž upravit ve funkci `RemoveLongestSuffix` při nenulové délce shody v případě, kdy odebíráme list, na jehož rodiče odkazuje ukazatel `start` nebo `end`.

6 Experimenty

Pro experimenty jsem zvolil datové korpusy Calgary a Lightweight. První jmenovaný protože je považován de facto za standard pro účely hodnocení bezdrátové komprese a druhý protože obsahuje poměrně velké soubory velikosti řádově desítek MB, což je vhodné vzhledem ke kompresi nad slovy. Pro testování komprese nad slovy jsem ale použil pouze soubory, které jsou textové.

Datový korpus Calgary se celkem skládá ze 14 souborů, jejich velikosti v bytech a mohutnosti množin abeced jsou uvedeny v tabulce Tabulce 5. V tabulce vidíme, že v korpusu jsou 4 soubory, u kterých je pravděpodobné, že jsou binární. Letmý průzkum souborů v textovém editoru nám potvrdí, že všechny 4 soubory opravdu binární jsou, z experimentů s kompresí nad slovy je tedy vyřadíme.

V případě korpusu Lightweight vidíme v Tabulce 3 také hned několik souborů, které by mohly být binární. Kontrola v textovém editoru ale tuto domněnku vyvrátí. Na druhou stranu nemá smysl zkoušet kompresi nad slovy nad souborem `chr22`, protože obsahuje pouze jedno dlouhé slovo. Kromě tohoto souboru jsem otestoval všechny.

Jelikož je vstupem algoritmu také velikost posuvného okna, vyzkoušel jsem, která velikost je pro který soubor z datového korpusu nejlepší. Kvalitu kompresního algoritmu lze samozřejmě hodnotit z mnoha hledisek, např. z hlediska kompresního poměru – velikosti vstupních dat ku velikosti výstupních dat.

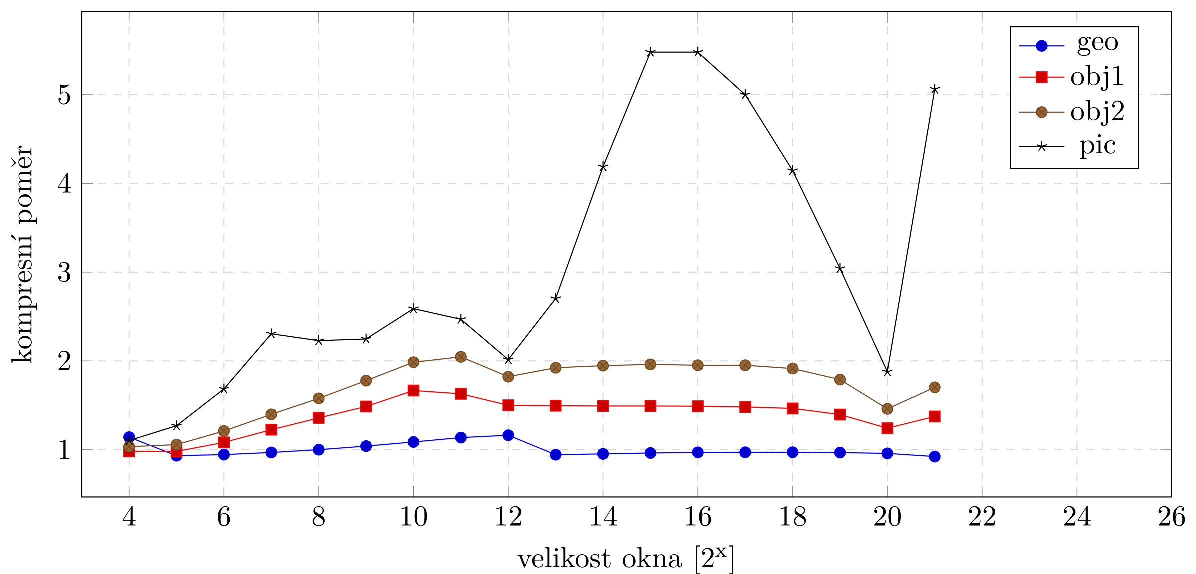
První tři grafy na Obrázku 10, Obrázku 11 a Obrázku 12 se věnují souborům z korpusu Calgary. První graf ukazuje výsledky simulace na binárních souborech, další dva pak výsledky simulace na ostatních souborech – ve druhém grafu vidíme výsledky simulace komprese na úrovni znaků a ve třetím grafu na úrovni slov. Z grafů je vidět, že pro většinu souborů vychází nejlepší kompresní poměr pro velikost posuvného okna 11 – 12, tedy hodnoty, kdy se ještě zakódovaná shoda vejde do 2 bajtů a zároveň délka shody se pohybuje nejvýše kolem 8 znaků. Když možnou délku shody zkrátíme na maximálně 4 znaky při udržování zakódované shody na 2 bytech, vidíme již mírný vzestup kompresního poměru. Zároveň je na těchto malých souborech vidět, jaký vliv má na kompresní poměr soubor se slovníkem, který se do výsledné velikosti samozřejmě také započítává. Z Tabulky 6 vyčteme, že v některých případech je slovník dokonce větší než komprimovaný soubor, v dalších případech se mu velikostně blíží.

Další dva grafy (Obrázek 13 a Obrázek 14) se věnují souborům z korpusu Lightweight. V prvním grafu jsou opět výsledky simulace komprese na úrovni znaků a ve druhém na úrovni slov. Simulaci jsem provedl nad 30 MB prefixy souborů, přičemž velikost posuvného okna byla v rozmezí 9 – 22 v případě komprese na úrovni znaků a v rozmezí 9 – 16 při kompresi na úrovni slov. Vidíme, že pro jediný soubor vychází kompresní poměr výborný, jinak je průměrný až podprůměrný.

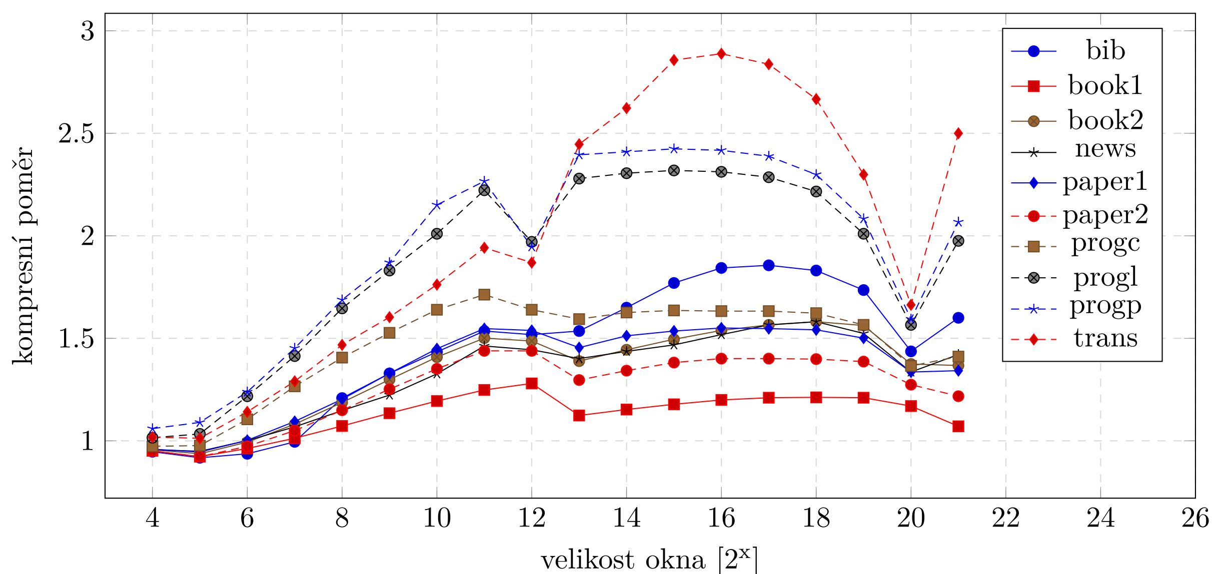
Na závěr experimentální části jsem porovnal nejlepší kompresní poměr každého komprimovaného souboru s kompresním poměrem souborů s příponou `zip` a `7z`. Výsledky jsou v Tabulce 7 a Tabulce 8 (sloupce s výsledky navrženého algoritmu jsou označeny "st_char"a "st_word").

Tabulka 5: Soubory datového korpusu Calgary – mohutnost abecedy, velikost v bytech a popis

Soubor	$\|\Sigma\|$	Velikost (B)	Popis
bib	81	111 261	bibliografie
book1	82	768 771	novela
book2	96	610 856	technická literatura
geo	256	102 400	geofyzikální data
news	98	377 109	dávkový soubor USENET
obj1	256	21 504	objektový soubor pro VAX
obj2	256	246 814	objektový soubor pro MAC
paper1	95	53 161	technický článek
paper2	91	82 199	technický článek
pic	159	513 216	černobílý FAX obrázek
progc	92	39 611	zdrojový kód jazyka C
progl	87	71 646	zdrojový kód jazyka LISP
progp	89	49 379	zdrojový kód jazyka Pascal
trans	99	93 695	přepis relace terminálu



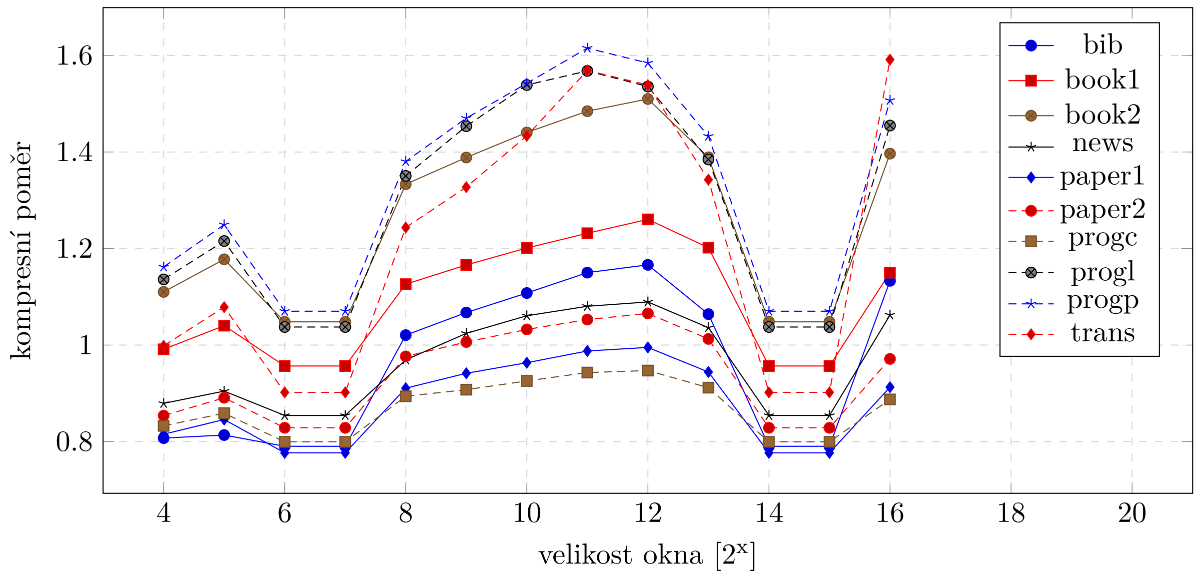
Obrázek 10: Kompresní poměr při kompresi binárních souborů korpusu Calgary s velikostí posuvného okna v intervalu 2^4-2^{21}; komprese na úrovni znaků



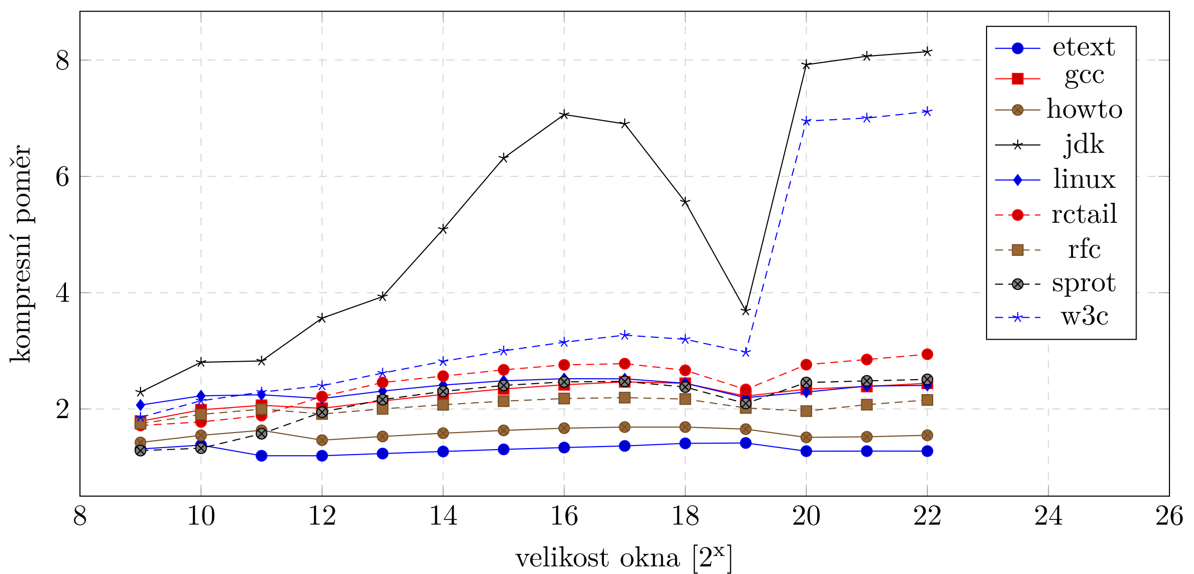
Obrázek 11: Kompresní poměr při kompresi textových souborů korpusu Calgary s velikostí posuvného okna v intervalu 2^4-2^{21}; komprese na úrovni znaků

Tabulka 6: Velikost původního a komprimovaného souboru a slovníku při kompresi na úrovni slov a velikosti posuvného okna 2^{12}; korpus Calgary

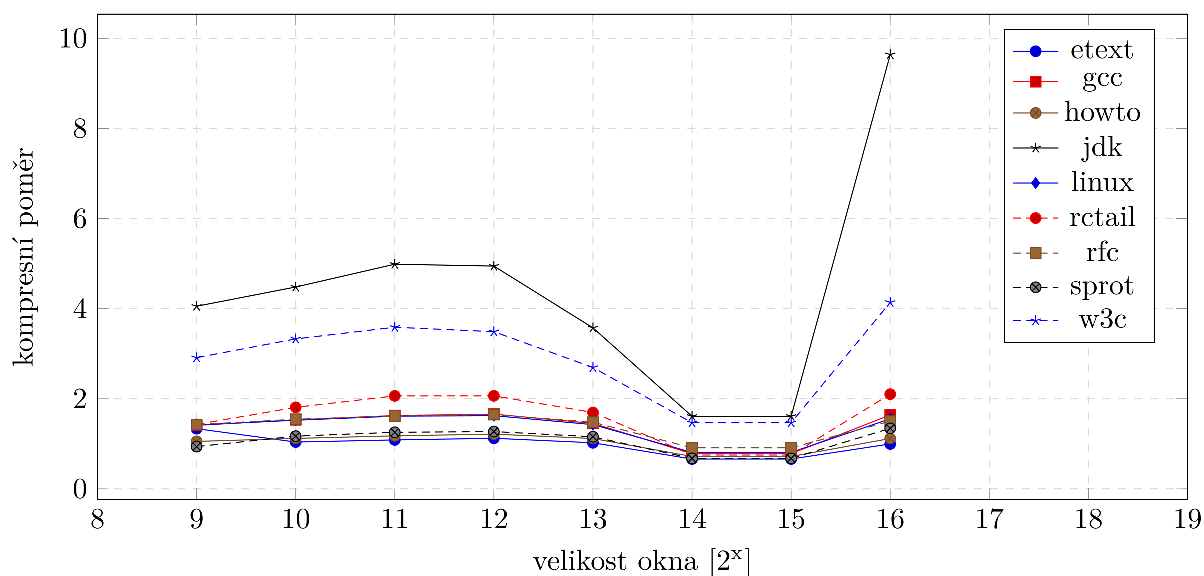
Soubor	Původní soubor	Komprimovaný soubor	Slovník
bib	111 261 B	41 818 B	53 588 B
book1	768 771 B	406 526 B	203 445 B
book2	610 856 B	272 064 B	132 492 B
news	377 109 B	171 575 B	174 622 B
paper1	53 161 B	23 900 B	29 523 B
paper2	82 199 B	38 571 B	38 575 B
progc	39 611 B	16 070 B	25 753 B
progl	71 646 B	22 902 B	23 752 B
progp	49 379 B	16 493 B	14 674 B
trans	93 695 B	30 566 B	30 345 B



Obrázek 12: Kompresní poměr při kompresi textových souborů korpusu Calgary s velikostí posuvného okna v intervalu 2^4-2^{16}; komprese na úrovni slov



Obrázek 13: Kompresní poměr při kompresi textových souborů korpusu Lightweight s velikostí posuvného okna v intervalu 2^9-2^{22}; komprese na úrovni znaků



Obrázek 14: Kompresní poměr při kompresi textových souborů korpusu Lightweight s velikostí posuvného okna v intervalu 2^9-2^{16}; komprese na úrovni slov

Tabulka 7: Porovnání kompresního poměru použitého algoritmu s formáty zip a 7z; korpus Calgary

Soubor	st_char	st_word	7z	zip
bib	1,81	1,17	3,62	3,25
book1	1,28	1,26	2,94	2,56
book2	1,58	1,51	3,59	3,08
geo	1,16	-	1,93	1,55
news	1,58	1,09	3,16	2,68
obj1	1,67	-	2,27	2,07
obj2	2,05	-	3,99	3,12
paper1	1,55	1,00	3,07	2,93
paper2	1,44	1,07	3,01	2,86
pic	5,48	-	11,77	9,74
progc	1,71	0,95	3,14	3,02
progl	2,32	1,57	4,75	4,53
progp	2,42	1,62	4,73	4,48
trans	2,89	1,59	5,54	5,02

Tabulka 8: Porovnání kompresního poměru použitého algoritmu s formáty zip a 7z; korpus Lightweight

Soubor	st_char	st_word	7z	zip
etext	1,41	1,33	3,82	2,86
gcc	2,47	1,66	6,61	4,39
howto	1,69	1,21	4,41	3,27
jdk	7,06	9,64	19,62	10,52
linux	2,52	1,62	6,94	4,71
rctail	2,94	2,10	8,06	4,90
rfc	2,20	1,65	6,33	4,23
sprot	2,51	1,34	5,96	4,26
w3c	7,11	4,14	23,46	5,51

7 Závěr

Tato práce si kladla za cíl navrhnout a implementovat kompresní algoritmus založený na sufixovém stromu. Bylo prozkoumáno chování sufixového stromu při jeho konstrukci i při pohybu posuvného okna. V těchto částech jsem využil poznatků Ukkonena [5] a Senfta [4], kteří již tuto problematiku podrobně zkoumali. Na základě získaných poznatků byl navržen slovníkový kompresní algoritmus, který pracuje na principu vyhledávání výskytů v části již zpracovaného řetězce. Je ukázáno, že k tomu lze využít pohyb aktivního bodu sufixovým stromem. Tento algoritmus byl implementován a je přiložen k této práci na CD.

Implementace byla experimentálně otestována na standardních datových korpusech Calgary a Lightweight. Výsledky napovídají, že navržený kompresní algoritmus postrádá potřebnou kvalitu, aby mohl soupeřit s dnes běžně používanými kompresními algoritmy. Tento výsledek bylo možné očekávat, jelikož algoritmus není na rozdíl od jiných algoritmů používaných v kompresních programech nijak dále optimalizován.

Komprese nad slovy byla implementována tak, aby bylo možné pracovat se slovy ve většině světových jazyků, které používají latinskou abecedu. Je ale vidět, že v menších souborech s reálným textem nejsou opakující se pasáže až tak časté, takže komprese na úrovni slov nedosahuje takového poměru jako komprese na úrovni znaků. U větších souborů sice pracujeme s celými slovy zastoupenými jednou hodnotou místo s jednotlivými znaky, nicméně pokud má tato hodnota 4 bajty, ušetříme pouze na slovech delších než 4 znaky a vzhledem k tomu, že za slova považujeme i mezery, nebude pravděpodobně úspora nijak značná.

Další rozvoj aplikace by mohl pokračovat hned několika směry. Bylo by vhodné zvýšit jazykové možnosti, aby bylo možné bez problémů komprimovat třeba i český, polský či slovenský text. Dále by bylo vhodné vyzkoušet jiný způsob kódování kroků aktivního bodu v sufixovém stromu, např. aritmetické kódování. Námětem na vylepšení je také úprava implementované architektury, jelikož za nynějšího stavu nejsme schopni v rozumných paměťových mezích využít větší posuvné okno než 2^{25}.

Literatura

[1] GIEGERICH, Robert a KURTZ, Stefan. *From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. Algorithmica.* New York: Springer-Verlag, 1997, strany 331-353. ISSN 0178-4617. Dostupné online: doi:10.1007/PL00009177.

[2] FIALA, Edward E. a GREENE, Daniel H. *Data compression with finite windows. Communications of the ACM.* New York: ACM, duben 1989, strany 490-505. ISSN 0001-0782. Dostupné online: doi:10.1145/63334.63341.

[3] LARSSON, N. Jesper. *Structures of String Matching and Data Compression.* Lund, Švédsko, září 1999. Dizertační práce. Lund University, Department of Computer Science. Dostupné online: http://www.larsson.dogma.net/thesis.pdf.

[4] SENFT, Martin. *Suffix Graphs and Lossless Data Compression.* Praha, 2013. Dizertační práce. Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwaru a výuky informatiky. Vedoucí práce Dvořák, Tomáš.

[5] UKKONEN, Esko. *On-Line Construction of Suffix Trees. Algorithmica.* New York: Springer-Verlag, září 1995, strany 249–260. ISSN 0178-4617. Dostupné online: doi:10.1007/BF01206331.

[6] KOVÁŘ, Petr. *Úvod do Teorie grafů.* FEI VŠB - TUO, 2012. Dostupné online: http://homel.vsb.cz/ kov16/files/uvod_do_teorie_grafu.pdf