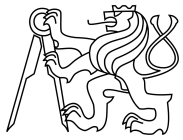
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace indexu stromu pro stromové vzorky
Student:	Lukáš Vozáb
Vedoucí:	doc. Ing. Jan Janoušek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Seznamte se s implementací Automatové knihovny, která je vyvíjena na katedře teoretické informatiky FIT ČVUT a slouží k experimentům s programátorem definovovanými algoritmy a modely výpočtů. Seznamte se s metodou indexování stromových datových struktur pro stromové vzorky [1]. Navrhněte vhodné datové struktury pro začlenění uvedené metody indexování do Automatové knihovny. Váš návrh datových struktur a metodu indexování implementujte v rámci Automatové knihovny a otestujte.

Seznam odborné literatury

[1] Jan Janousek, Borivoj Melichar, Radomir Polach, Martin Poliak, Jan Travnicek: A Full and Linear Index of a Tree for Tree Patterns. In: LNCS 8614 (DCFS 2014), Springer, pp. 198-209, 2014.

L.S.

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

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 18. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Implementace indexu stromu pro stromové vzorky

Lukáš Vozáb

Vedoucí práce: doc. Ing. Jan Janoušek, Ph.D.

12. května 2016

Poděkování

Chtěl bych poděkovat především Ing. Trávníčkovi za ochotné a pohotové odpovídání na všechny mé nejasnosti a také mé rodině za poskytnutí příjemného prostředí a podpory během psaní této bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2016 .....

Odkaz na tuto práci

Vozáb, Lukáš. *Implementace indexu stromu pro stromové vzorky*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá implementací indexace a vyhledávání vzorků ve stromových strukturách. Je použita nová metoda využívající kompaktní suffixový automat, který představuje především drastickou paměťovou úsporu. Vyhledávací fáze pak dokáže poskytnout množinu výskytů stromových vzorků v čase nezávislém na celkovém množství dat.

Klíčová slova implementace indexu stromových struktur, kompaktní suffixový automat, hledání stromových vzorků, knihovna, Katedra teoretické informatiky, Fakulta informačních technologii ČVUT

Abstract

This bachelor thesis deals with implementation of indexing and matching patterns in tree structures. This new method uses compact suffix automaton, which comes with significant drop in memory cost. Matching phase is able to provide a set of occurrences in time not dependent on total amount of data.

Keywords implementation of tree structure index, compact suffix automaton, tree pattern matching, library, Department of theoretical computer science, Faculty of information technology CTU

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vyhledávání v datech je velice běžným problémem. Pokud bychom se např. snažili nalézt konkrétní slovo v textu a pokaždé bychom tento text procházeli od začátku do konce, stoupala by časová složitost s celkovým objemem dat. Od jisté hranice by už vyhledávání trvalo tak dlouho, že by se velká část aplikací jevila jako nepoužitelná.

Na Katedře teoretické informatiky FIT ČVUT v Praze je v současné době vyvíjena knihovna týkající se algoritmů mnoha odvětví. Jedním z nich je arbologie, neboli zpracování stromových struktur a na tomto poli také vzniká tato bakalářská práce. Krom výzkumných účelů katedry je plánováno vydání knihovny studentům.

Tato práce nalezne užitek v jejím rozšíření o další funkcionalitu, a sice možnost indexace stromových datových struktur. Indexací dat se rozumí takové předzpracování, jež dokáže efektivněji odpovědět na konkrétní typ dotazu a to v čase typicky nezávislém na celkovém množství dat.

V analytické části řeším výhody volby kompaktního suffixového automatu jako indexující strukturu a následně vysvětluji princip a postup konstrukce. Dále popisuji, co více je nutné k nalezení výskytů stromového vzorku v indexovaném stromu.

V praktické části provázím čtenáře vývojem knihovny (v rámci této práce) chronologicky, jak jsem jednotlivé funkcionality implementoval. Těmi jsou kompaktní suffixový automat a mechanismus, jež vyhodnocuje výskyty stromových vzorků. Na závěr předvádím, jak různé části knihovny spouštět a jaké jsou možnosti.

Kapitola 1

Cíl práce

Cílem analytické části je představení a porovnání současných možností řešení, stanovení všech potřebných pojmů a konvencí, seznámení čtenáře s kompaktním suffixovým automatem a veškerých technik použitých při předzpracování indexovaného stromu a nacházení výskytů stromových vzorků. Toto všechno je doplněno o příslušné obrázky a pseudokódy.

Cílem praktické části je rozšíření knihovny vyvíjené na Katedře informačních technologií o implementaci kompaktního suffixového automatu, který bude následně použit v implementaci algoritmu vyhledávajícího výskyty stromových vzorků v indexovaném stromu. V této kapitole provázím čtenáře postupem vývoje knihovny (v rámci práce), ukazuji kusy vytvořeného zdrojového kódu a vše náležitě komentuji.

Kapitola 2

Analýza a návrh

2.1 Úvodní poznámky

Tato práce se zabývá zejména implementací toho, co je zpracováno ve výzkumu nesoucí název *A Full and Linear Index of a Tree for Tree Patterns* [1]. Je mým prvním a hlavním zdrojem.

Pokud čtenář nenarazil v této práci na předchozí výskyt základních pojmů *index stromu* a *stromový vzorek*, rád bych je ujasnil ještě před tím, než se vrhnu na sekci, která je pojmům věnována.

Stromový index je takové předzpracování dat, které umožňuje poskytovat odpověď na konkrétní skupinu dotazů v čase typicky nezávislém na celkovém množství dat.

Stromový vzorek je strom, jehož listy však mohou být nahrazeny speciálním symbolem S, který slouží jako zástupný znak pro libovolně vypadající podstrom.

Píši tady o *stromovém* indexu a vzorku, ale to stejné se dá aplikovat i na prostý text. Vyhledávání podstromů a vzorků ve stromu je analogické k hledání podřetězců v textu. Toto je zapříčiněno následující skutečností:

Lemma 1. *Lineární notace podstromu je podřetězcem lineární notace stromu. [1]*

Hledání stromových vzorků ve stromu tedy odpovídá hledání textových vzorků se zástupnými znaky (wildcardy), v tomto případě nahrazující libovolný počet znaků. Pojem *lineární notace* bude vysvětlen v následující sekci. V této práci je využívána prefixová varianta.

Celkový počet podřetězců v textu je kvadratický jeho délce. Velikost indexovací struktury je ale typicky lineární. [1]

2.2 Pojmy a konvence

V této práci nebudu definovat základní z teorie grafů a stringologie, jako jsou *strom*, *uzel*, *hrana*, *řetězec*, atp. od základů. Jejich obecně známé definice postačí. Potřebuji ale stanovit jejich konvenci, proto tedy následuje i zmínka o stromu. Konvence definic 1–9 je přejata z [1].

Definice 1. Nechť $T = (N, R)$ je *strom*. N je množina uzlů stromu s kořenem $r \in N$. R je množina hran mezi uzly. Pokud $g \in N$ je přímým potomkem $f \in N$, pak $(f, g) \in R$.

Následující 4 definice spolu velice úzce souvisí a jsou na sobě závislé. Po definici 5 je vysvětlím jako celek.

Definice 2. *Arita*, značena $arity(a)$, je vyjádření, kolik symbolů přímo následuje za symbolem a.

V této definici úmyslně nebylo specifikováno, o jaké symboly se jedná. K tomu je potřeba definovat následující pojem.

Definice 3. *Ohodnocená abeceda* je konečná množina $\mathcal{A}$ symbolů nezáporné arity. Množina symbolů arity p je značena $\mathcal{A}_p$.

Symboly ohodnocené abecedy $\mathcal{A}$ budu denotovat například $a2$, kde $a \in \mathcal{A}$ a $arity(a) = 2$. Jedna ohodnocená abeceda může obsahovat i více stejných symbolů, ale s rozdílnou aritou. Jedna z možných abeced je například $\mathcal{A} = \{a2, a0, b3\}$.

Definice 4. *Označený strom* je takový strom, ve kterém je každý uzel označen symbolem a z ohodnocené abecedy $\mathcal{A}$.

Po srovnání definice 2 a této začíná vyplývat dopad arity symbolů na uzly stromu. Aby arita měla nějaký efekt, musí být strom ještě *ohodnocený* (viz následující definice).

Definice 5. *Ohodnocený strom* je takový strom, kde je každý uzel označen aritou svého příslušného symbolu $a \in \mathcal{A}$, neboli $arity(a)$.

U každého uzlu tedy arita jeho symbolu znamená stupeň uzlu, neboli počet přímých potomků. Uzly arity 0 jsou nazývány *listy*. Zástupný znak S ve stromovém vzorku je chápan jako list, tedy $arity(S) = 0$. Arita $0, 1, 2, \ldots, p$ je ve stejném pořadí nazývána jako nulární, unární, binární, ..., p-ární.

Definice 6. Seřazený strom je takový strom, kde jsou potomci všech uzlů nějakým způsobem seřazeni. Pokud je strom zároveň ohodnocený (definice 5), pak jsou v této práci řazeni podle arity.

Definice 7. *Podstrom* stromu T zakořeněný v libovolném uzlu $f \in N$ je strom $T_f = (N_f, R_f)$, kde N_f a R_f jsou největší možné podmnožiny N a R.

Je to tedy strom obsahující uzel z původního stromu a všechny potomky tohoto uzlu, včetně nepřímých.

Definice 8. *Prefixová notace* $pref(T)$ je řetězec nad abecedou $\mathcal{A}$. Je to způsob, kterým lze jednoduše reprezentovat strom T textovou formou. Je definována následovně:

(1) $pref(a) = a_0$, pokud a je list,

(2) $pref(T) = a_n\ pref(b_1)\ pref(b_2) \ldots pref(b_n)$, kde a_n je kořenem stromu T a $b_1, b_2, \ldots, b_n$ jsou podstromy, jejichž kořeny jsou přímými potomky a_n.

V prefixové notaci může být analogicky zapsán i stromový vzorek P s tím rozdílem, že abeceda $\mathcal{A}$, nad kterou je tento řetězec postaven, je obohacena o zástupný znak S, tedy $pref(P) \in (\mathcal{A} \cup \{S\ \})$.

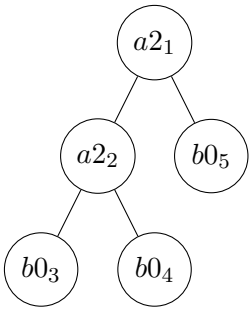
Symboly ohodnocené abecedy jsou denotovány například $a2$, ale přidání subskriptu, tedy $a2_1$ nebo $a2_2$, jen poukazuje na to, že se jedná o dva rozdílné uzly. V prefixové lineární notaci nad ohodnocenou abecedou $\mathcal{A} = \{a2, a0\}$ by tedy $pref(T) = a2_1 a0_2 a0_3$ znamenalo, že strom T s kořenem $a2_1$ má dva přímé potomky – listy $a0_2$ a $a0_3$.

Definice 9. *Aritní kontrolní součet* (zkratka *ac* z anglického *arity checksum*) je prostředek pro kontrolu validity prefixové notace vůči stromu. Nechť $w = a_1 a_2 \ldots a_m, m \geq 1$ je řetězec nad ohodnocenou abecedou A. Pak aritní kontrolní součet tohoto řetězce je $ac(w) = \sum_{i=1}^{m} arity(a_i) - m + 1$. Nechť $pref(T)$ je strom T v prefixové notaci a w je podřetězcem $pref(T)$. Řetězec w je prefixovou notací podstromu T, právě když $ac(w) = 0$. Jakýkoli prefix w_1 řetězce w rozdílný od w (tj. $w = w_1 x$, $x \neq \varepsilon$) je zároveň prefixovou notací, pokud $ac(w_1) \geq 1$.

Příklad 1. Předpokládejme ohodnocenou abecedu $\mathcal{A} = \{a0, b0\}$ a označený, ohodnocený, seřazený strom $T' = (N_1, R_1)$ nad abecedou $\mathcal{A}$, kde

(1) $N_1 = \{a2_1, a2_2, b0_3, b0_4, b0_5\}$ je seřazená množina vrcholů,

(2) $R_1 = \{(a2_1, a2_2), (a2_1, b0_5), (a2_2, b0_3), (a2_2, b0_4)\}$ je seřazená množina hran.



Obrázek 2.1: Strom T'

Vrcholy jsou zde záměrně očíslovány $1, 2, \ldots, 5$ tak, aby obrázek 1 poukazoval na pořadí čtení uzlů, pokud by tento strom byl zapsán v prefixové notaci. Pomocí definice 8 je správný postup k sestrojení prefixového zápisu následující:

(1) $pref(T') = a2_1\ pref(a2_2)\ pref(b0_5)$

(2) $pref(a2_2) = a2_2\ pref(b0_3)\ pref(b0_4)$
$pref(T') = a2_1\ a2_2\ pref(b0_3)\ pref(b0_4)\ pref(b0_5)$

(3) $pref(b0_3) = b0_3$
$pref(T') = a2_1\ a2_2\ b0_3\ pref(b0_4)\ pref(b0_5)$

(4) $pref(b0_4) = b0_4$
$pref(T1) = a2_1\ a2_2\ b0_3\ b0_4\ pref(b0_5)$

(5) $pref(b0_5) = b0_5$
$pref(T') = a2_1\ a2_2\ b0_3\ b0_4\ b0_5$

Strom T' v prefixovém zápisu je tedy $pref(T') = a2_1\ a2_2\ b0_3\ b0_4\ b0_5$. Podle definice 9 nyní provedu kontrolní aritní součet.

(1) $ac(pref(T')) = arity(a2_1)+arity(a2_2)+arity(b0_3)+arity(b0_4)+arity(b0_5)-|N_1|+1$

(2) $ac(pref(T')) = 2+2+0+0+0-5+1$

(3) $ac(pref(T')) = 0$

Zápis $pref(T')$ je tedy validní prefixovou notací stromu T'.

2.3 Současná řešení a nová metoda

2.3.1 Současná řešení

Jelikož vyhledávací fáze klade důraz na podporu zástupných znaků, velice se tím zužuje výběr použitelných řešení. V [1] je prezentována následující analýza, kterou tato práce cituje.

V [2] je index konstruován se zástupným symbolem nahrazujícím jen jeden znak, což nemůže být použito pro lineární notaci stromu. Zástupný symbol variabilní délky je implementován v [3], nicméně časová složitost vyhledávání záleží na délce těchto mezer, a to není efektivní pro stromové vzorky, kde můžou být mezery libovolně velké.

Zásobníkový automat pro stromové vzorky je použit v [4], jenže velikost tohoto automatu není lineární vůči indexovanému stromu. Také lze vytvořit konečný automat přijímající všechny možné vzorky stromu, ovšem velikost by byla exponenciální [5, 6].

V [7] jsou použity mezery variabilních délek, ovšem reprezentace mezer je zde nekompatibilní s hledáním stromů. Dochází zde navíc k předzpracování vzorku, nikoliv indexovaného stromu. Další metody využívající taková předzpracování jsou navrženy v [5, 8].

2.3.2 Nová metoda

Ze stromu T s n uzly je vytvořen index využívající kompaktní suffixový automat (sekce 2.4) a podstromovou přechodovou tabulku (sekce 2.5). Velikost indexu je $\mathcal{O}(n)$.

Vyhledávací fáze (sekce 2.6) přečte vstupní stromový vzorek P velikosti m v lineární prefixové notaci $pref(P) = P_1SP_2S \ldots SP_k, k \geq 1$ v čase $\mathcal{O}(m + \sum_{i=1}^{k} |occ(P_i)|)$, kde $occ(P_i)$ je množina všech výskytů P_i ve stromě T. [1]

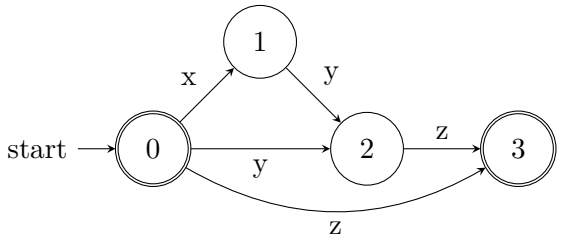
2.4 Kompaktní suffixový automat

Tato sekce se zaobírá konstrukcí kompaktního suffixového automatu. Je to efektivní a elegantní struktura implementující index nad textem. Když mluvím o textu, nevylučuji ale možnost použití prefixové notace stromu.

Definice 10. *Kompaktní suffixový automat* (CSA z anglického „compact suffix automaton“) pro text je konečný automat, který přijímá všechny suffixy textu. Liší se od klasického suffixového automatu (SA) tím, že neobsahuje stavy, které mají jen jednoho následníka a zároveň nejsou konečné. [9]

Notace $CSA(pref(T))$ značí CSA nad prefixovou notací stromu T, tedy řetězcem nad abecedou $\mathcal{A} \cup S$. V následujícím příkladu ukážu rozdíl mezi SA a CSA.

Příklad 2. Předpokládejme jednoduchý $SA(xyz)$, který vypadá následovně:



Obrázek 2.2: Suffixový automat $SA(xyz)$

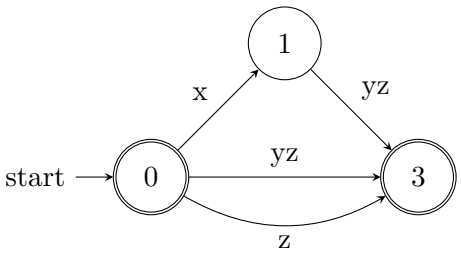
Transformace SA do CSA probíhá následujícím postupem:

(1) najdi všechny stavy, které nejsou konečné a mají pouze jednoho přímého následníka,

(2) každý takový stav odstraň a příchozí hrany vyveď do onoho přímého následníka,

(3) spoj řetězce na příchozích a odchozích hranách tohoto uzlu.

Kroku (1) vyhovuje stav 2. Lze na něj tedy aplikovat kroky (2) a (3). Výsledkem je následující CSA obsahující o 1 stav a přechod méně:



Obrázek 2.3: Kompaktní suffixový automat $CSA(xyz)$

Velikost CSA je lineární ke vstupnímu textu a lze ji i v lineárním čase postavit za předpokladu, že text je nad fixně definovanou abecedu [10]. Konstrukci lze provést přímo z indexovaného textu, a nebo, jak jsem ukázal na příkladu 2.2, transformací z SA. Podle statistiky uvedené v [10] má kompaktní verze automatu nad sekvencemi DNA (řetězce nad abecedou $\{a, c, g, t\}$) přibližně o 2/3 méně stavů a 1/2 přechodů než klasická. Následkem toho je úspora paměti okolo 50 %.

2.4.1 Konstrukce naivním algoritmem

Ještě než se dostanu k efektivní lineární verzi, chtěl bych poukázat na nejjednodušší řešení. Díky němu bude později patrnější, kde je klíč k dosažení lineární časové složitosti.

Nejprve ze potřeba zavést několik notací, které budou následně použity k vysvětlení algoritmu. Pro řetězec délky n popisuje notace suf_i takový suffix, kde suf_0 je celý řetězec a suf_{n-1} jen poslední znak. Tedy např. pro $w = abc$ je $suf_0 = abc$, $suf_1 = bc$ a $suf_2 = c$. Notace $head_i$ popisuje nejdelší společný prefix suf_i a suf_j pro všechny j takové, že $0 \leq j < i$. Notace $tail_i$ je jen rozdíl mezi suf_i a $head_i$, neboli $head_i + tail_i = suf_i$. Přechod α ze stavu p do q značím (p, α, q). Přechod s prefixem β ze stavu p do neurčeného stavu značím $(p, \beta]$. [10]

Následuje znění algoritmu. Konstrukce vždy začíná s počátečním stavem I (initial) a koncovým F (final). Jelikož prázdný řetězec je suffixem libovolného textu, i stav I je inicializován jako koncový. Pseudokód je s drobnými změnami přebrán z [10].

Algoritmus 1 Naivní konstrukce CSA

Vstup: Stavy I a F
Výstup: CSA (naivní)

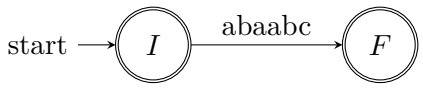
```
1:  for all suf_i (i ∈ {0, ..., n − 1}) do
2:      (q, γ) ← SlowFind(I, suf_i)
3:      if γ = ε then
4:          if tail_i ≠ ε then
5:              vytvoř přechod (q, tail_i, F)
6:      else
7:          vytvoř nový stav v
8:          rozděl přechod (q, γ] na (q, γ, v) a jeho zbytek
9:          if tail_i ≠ ε then
10:             vytvoř přechod (v, tail_i, F)
11: označ koncové stavy
```

Na řádku 2 je použito volání $SlowFind(I, suf_i)$. Tato funkce začíná na stavu I a následuje cestu danou suf_i. Má toho za úkol ještě o něco více, ale to rozeberu podrobně později. Její návratovou hodnotou je dvojice (q, γ), kde q je poslední stav, kterým funkce prošla a γ je text od stavu q, který se shoduje s prefixem nějakého z odchozích hran q.

Řádek 3 řeší případ, kdy žádná z odchozích hran takový prefix neobsahuje. V takovém případě stačí ze zbytku textu vytvořit přechod do koncového stavu F. Důležité je poznamenat, že pokud $tail_i$ je prázdný řetězec, není potřeba vyvádět žádnou hranu do F, protože stav, do které suf_i vede, bude na konci algoritmu označen jako koncový.

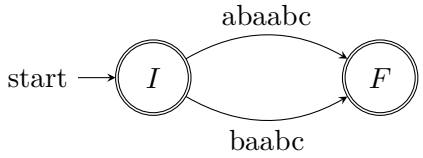
Pokud γ není prázdný řetězec, pak by suf_i končil „uprostřed“ hrany. Zde, jak praví řádky 8–12, je potřeba vytvořit nový stav v a k němu náležité přechody.

Příklad 3. V tomto příkladu ukážu postupné sestavení $CSA(abaabc)$ naivním algoritmem.



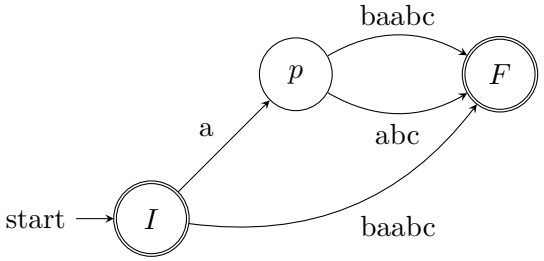
Obrázek 2.4: $CSA(abaabc)$ po vložení $suf_0 = abaabc$ (1. krok)

První krok je triviální. Mezi stavy I a F zatím neexistují žádné přechody, proto je vytvořen $(I, abaabc, F)$.



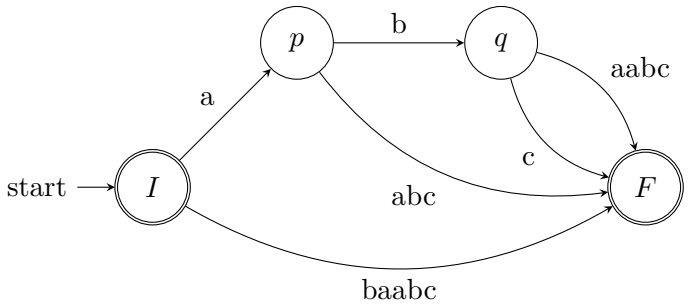
Obrázek 2.5: $CSA(abaabc)$ po vložení $suf_1 = baabc$ (2. krok)

V druhém kroku jediný existující přechod $(I, abaabc, F)$ nesdílí se $suf_1 = baabc$ žádný prefix, proto je potřeba vytvořit $(I, baabc, F)$.



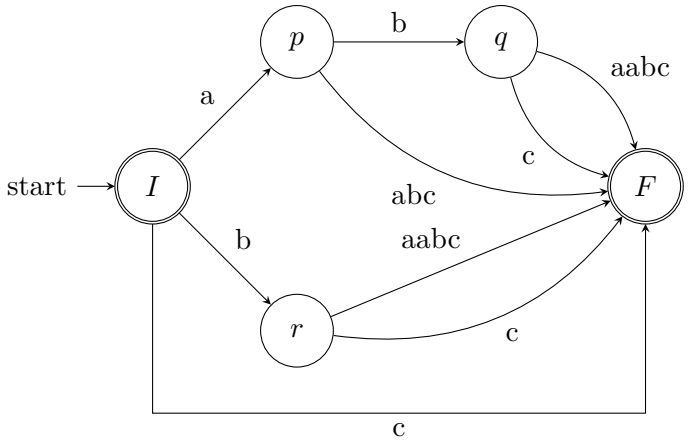
Obrázek 2.6: $CSA(abaabc)$ po vložení $suf_2 = aabc$ (3. krok)

Třetí krok už je zajímavější. Přechod $(I, abaabc, F)$ a $suf_2 = aabc$ spolu sdílí prefix a, aneb tvoří $head_2 = a$ a $tail_2 = abc$. Je tedy vytvořen nový stav p a přechod $(I, abaabc, F)$ je rozdělen na (I, a, p) a $(p, baabc, F)$. Pak už jen stačí vložit $tail_2$, neboli (p, abc, F).



Obrázek 2.7: $CSA(abaabc)$ po vložení $suf_3 = abc$ (4. krok)

Ve čtvrtém kroku lze následovat přechod (I, a, p). Z vkládaného textu tedy lze odebrat prefix a. Zbytek (bc) sdílí s přechodem $(p, baabc, F)$ prefix b, proto $tail_3 = c$ a analogicky ke kroku 3 vzniká stav q a přechody (p, b, q), $(q, aabc, F)$ a (q, c, F).

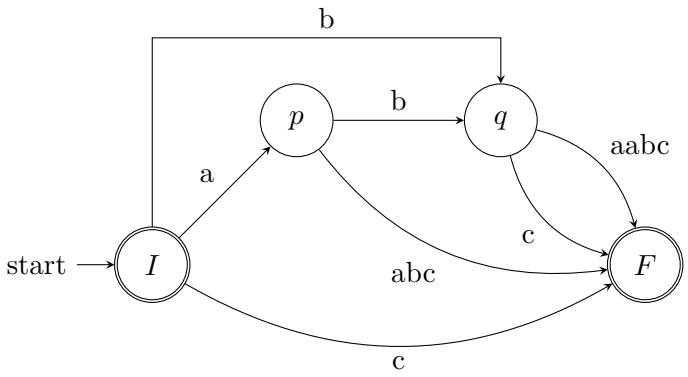


Obrázek 2.8: $CSA(abaabc)$ po vložení $suf_4 = bc$ a $suf_5 = c$ (5. a 6. krok)

Konečně vložení $suf_4 = bc$ zapříčiní rozpojení $(I, baabc, F)$ a $suf_5 = c$ nesdílí žádný prefix, proto jde rovnou do F.

2.4.2 Konstrukce lineárním algoritmem

Na obrázku 2.8 je dobře vidět, proč je tento způsob konstrukce naivní a proč by se dal zlepšit. Při porovnání odchozích hran stavů q a r vychází najevo, že se v automatu vyskytuje to samé vícekrát, než je potřeba. To vede k větší paměťové náročnosti. Intuitivně bych například úplně vymazal stav r a místo toho vyvedl „zkratku“ (I, b, q).



Obrázek 2.9: $CSA(abaabc)$ po odebrání stavu r a vyvedení „zkratky“ (I, b, q)

Automat po takové změně stále přijímá naprosto identickou množinu řetězců, jen je odlehčen o jeden stav a dva přechody.

Definice 11. Nechť p a q jsou stavy CSA a s je řetězec, kterým se lze od počátečního stavu dostat do p. *Suffix link* stavu p je stav q, pokud se lze z počátečního stavu dostat do q nejdelším možným suffixem řetězce s, který mu není roven. [10]

Pokud by někdy nastala v konstrukci situace, kdy by algoritmus po využití „zkratky“ chtěl nějaký stav označit jako konečný, znamenalo by to, že všechny řetězce, jejichž cesta vede do tohoto stavu bez využití takového přechodu, jsou nutně suffixy vkládaného textu. Konečné stavy jsou proto označeny až na úplném konci konstrukce pomocí tzv. *suffixové cesty*, která začíná ve stavu F a končí v počátečním [10].

Definice 12. Délka stavu p zapsaná jako $legth(p)$ reprezentuje délku nejdelšího řetězce, ze kterého se lze z počátečního stavu dostat do p. Nechť r je stav, ze kterého vede hrana do p a l je délka řetězce této hrany. Pokud $length(r)+l = length(p)$, pak je tato hrana *pevná*. V opačném případě, pokud $length(r) + l < length(p)$, je hrana *nepevná*.

Po nastudování literatury [10] prezentující konstrukci CSA ovšem docházím k závěru, že pro funkční implementaci nemám dostatečné množství informací. Zdá se, že některé instrukce si navíc vzájemně odporují. Mé problémy s kapitolou lineární konstrukce vysvětlím v následujících bodech.

- Suffix link má být vyhledáván u každého stavu v moment, kdy je vytvořen. Není ale patrné, jak přesně postupovat, pokud předcházející stav je počáteční.
- V lineární verzi má být *SlowFind* rozšířen o duplikování stavu v případě, že projde přes nepevnou hranu. V jednom odstavci je psáno, na kterou

hodnotu má být nastaven suffix link tohoto stavu. V dalším je ale psáno, že pro zjištění suffix linku tohoto stavu má být spuštěna metoda *FastFind* (patrně rychlejší varianta *SlowFind*, která toho nemá mít tolik na práci).

- *FastFind* má prý porovnávat hrany jen podle prvního znaku. Stále se ale může stát, že hledání skončí „uprostřed“ nějaké hrany stejným způsobem jako u *SlowFind*.

- Jelikož duplikace stavů se dějí ve *SlowFind*, není v literatuře specifikováno, co se má stát v případě, že přes nepevnou hranu projde *FastFind*. Pokud by stavy byly duplikovány stejným způsobem, jakým to dělá *SlowFind*, pak by úplně pozbývalo smyslu odlišovat tyto dva algoritmy. Pokud k duplikaci docházet nemá, pak se stává, že se nastavují suffix linky na sebe sama, což podle definice 11 nemá smysl.

- Při hledání suffix linků občas dojde k vložení kratšího suffixu než ten, kterého se iterace týka. V jedné iteraci je např. vytvořena nepevná hrana a v iteraci tohoto kratšího suffixu je ihned zničena duplikací ze *SlowFind*. Dávalo by smysl, že takové iterace mají být nějak přeskakovány, aby vůbec vytváření „zkratek“ v automatu mělo nějaký význam. To ale vůbec není prodiskutováno.

- Suffix link stavu F se podle pseudokódu kapitoly nastavuje v každé iteraci na stav získaný vložením suf_i. Není ale specifikováno, zda má v nějakém případě tento suffix link ponechat na původní hodnotě. Tento bod úzce souvisí s vynecháváním iterací (viz minulý bod) týkajících se vkládání suffixů, které se v automatu již nachází. Pokud by se suffix link F nastavil skutečně v každé iteraci, byl by pomocí suffixové cesty označen jen stav, do kterého vede poslední vkládaný suffix.

2.5 Indexace stromu pro stromové vzorky

V této sekci se zaměřím na dvě různé tabulky, které algoritmům v následující sekci poskytnou potřebné informace o indexovaném stromu.

Definice 13. Nechť T je strom a $pref(T) = a_1 a_2 \dots a_n$, kde $n \geq 1$ jeho prefixová notace. *Podstromová přechodová tabulka* (subtree jump table) $SJT(T)$ je mapování z množiny $\{1, \dots, n\}$ do $\{2, \dots, n+1\}$. Pokud je možné $pref(T)$ zároveň zapsat jako $a_i a_{i+1} \dots a_{j-1}$, kde $1 \leq i < j \leq n+1$, pak $SJT[i] = j$. [1]

Tato tabulka, jak už její název napovídá, má na starosti správu podstromů. Lineární prefixová notace je přeci jen text a není časově úplně triviální rozhodnout, od jaké pozice kam se rozmáhá konkrétní podstrom. Pozice v tabulce odpovídá pozici uzlu stromu v lineární notaci a hodnota udává konec podstromu

zakořeněného v tomto uzlu. V definici 13 uvádím, že se jedná o mapování do množiny, ve které je spodní i horní hranice o 1 větší než hranice pozic prefixové notace. Tato hodnota značí horní hranici výlučně, tedy u řetězce délky n je hranice $n + 1$ úplným koncem. Pro hodnoty menší než $n + 1$ se tedy jedná o pozice, které se týkají jiného podstromu.

Příklad 4. V tomto příkladu ukážu, jak by vypadala SJT stromu T' z příkladu 1. Prefixová notace tohoto stromu je $pref(T') = a2_1a2_2b0_3b0_4b0_5$.

i	1	2	3	4	5
$SJT[i]$	6	5	4	5	6

Tabulka 2.1: SJT nad $pref(T')$

Jelikož kořen celého stromu je vždy na první pozici prefixové notace, hned první dvojice v tabulce ($SJT[1] = 6$) vlastně říká, že „tento strom končí na úplném konci notace.“ Listy, v tomto příkladu $b0_3$, $b0_4$ a $b0_5$, mají hodnotu v tabulce přesně o 1 vyšší než je jejich pozice, neboli každý z těchto uzlů tvoří podstrom o 1 prvku.

Algoritmus 2 Konstrukce podstromové přechodové tabulky [1]

Název: ConstructSubtreeJumpTable

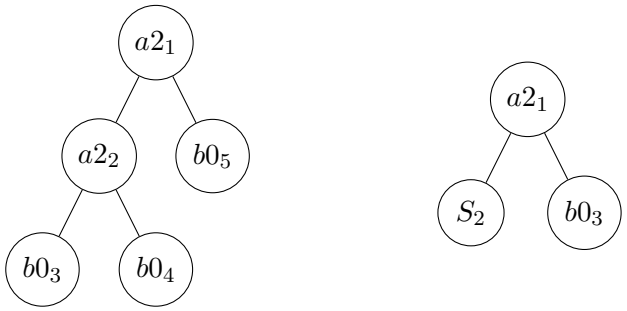
Vstup: Strom T v prefixové notaci $pref(T)$, pozice aktuálního uzlu $rootIndex$, reference na prázdnou tabulku $SJT(T)$.

Výstup: Pozice $exitIndex$, $SJT(T)$.

1: $index \leftarrow rootIndex + 1$
2: **for** $i = 1$ $arity(pref(T)[rootIndex])$ **do**
3: $\quad index \leftarrow ConstructSubtreeJumpTable(pref(T), index, SJT(T))$
4: $SJT(T)[rootIndex] \leftarrow index$
5: **return** $index$

Definice 14. Nechť $occ = \{(first_1, last_1), \ldots, (first_n, last_n)\}$ je množina dvojic přirozených čísel, kde každé číslo $last_i, 1 \leq i \leq n$ je unikátní. Pro tabulku Rev^n_{occ} platí, že $Rev^n_{occ}[last_i] = first_i$, pokud $(first_i, last_i) \in occ$, jinak $Rev^n_{occ}[last_i] = -1$. [1]

Příklad 5. Druhá tabulka se netýká už jen indexovaného stromu, ale i vyhledávaného vzorku. Jako u minulého příkladu zde použiju stejný strom T' s prefixovou notací $pref(T') = a2_1a2_2b0_3b0_4b0_5$. Ještě musí existovat nějaký vzorek, který má ve stromu být vyhledáván. Nechť tedy existuje stromový vzorek P' s prefixovou notací $pref(P') = a2_1S_2b0_3$. Zde je porovnání těchto dvou stromů:



Obrázek 2.10: Srovnání stromu T' (nalevo) a vzorku P' (napravo)

Podle obrázku 2.10 je zřetelné, že vzorek P' sedí na dva různé podstromy stromu T'. První je zakořeněn v $a2_1$ a druhý v $a2_2$. Takový výsledek by vyprodukoval množinu $occ = \{(1,6),(2,5)\}$. Dvojice $(1,6)$ praví, že v subjektu $pref(T') = a2_1a2_2b0_3b0_4b0_5$ je shoda se vzorkem $pref(P') = a2_1S_2b0_3$ od 1. do 5. pozice. Takže se dá říci, že zástupný znak S_2 zde zastoupil řetězec $a2_2b0_3b0_4$. U dvojice $(2,5)$ je shoda od 2. do 4. pozice, takže S_2 zastoupil jen $b0_3$. Tabulka Rev^5_{occ} vypadá takto:

i	1	2	3	4	5	6
$Rev^5_{\{(1,6),(2,5)\}}[i]$	-1	-1	-1	-1	2	1

Tabulka 2.2: Rev^5_{occ} nad $pref(T')$

Tabulka 2.2 neslouží k ničemu jinému než získání dvojic z množiny occ klíčem $last_i$, proto např. pro dvojici $(1,6)$ platí $Rev^5_{occ}[5] = 1$. Použití najde v následující sekci.

2.6 Výpočet pozic všech výskytů vzorku

Sekce 2.5 představila tabulku Rev^n_{occ}, ale před tím, než bude možné ji sestavit, je potřeba ustanovit několik dalších pojmů.

Definice 15. Nechť $pref(P) = P_1S\ P_2S\ \ldots S\ P_k$ je prefixová notace vzorku P nad abecedou $A \cup \{S\}$, kde žádný podřetězec $P_i, 1 \leq i \leq k$ neobsahuje symbol S. Pak podřetězec P_i je zván *podvzorkem.* [1]

Příklad 6. Vzorek $pref(P') = a2_1S_2b0_3$ z příkladu 5 by se podle definice 15 dal přepsat do $pref(P') = P_1SP_2$, kde $P_1 = a2$ a $P_2 = b0$ jsou podvzorky P'.

Příklad 7. Nechť existuje nový vzorek P'' s prefixovou notací $pref(P'') = a4SSb0b0$. Takový vzorek lze vyjádřit jako $pref(P'') = P_1SP_2S\ P_3$, kde $P_1 = a4$, $P_2 = \varepsilon$ a $P_3 = b0b0$.

Následující definice už byla nastíněna v definici 14, ale nikoliv v kontextu podvzorků.

Definice 16. Nechť $pref(T) = a_1 a_2 \dots a_n$ je prefixová notace stromu T a $pref(P) = P_1 S\ P_2 S\ \dots S\ P_k$ prefixová notace vzorku P. *Výskyt* podvzorku $P_i, 1 \leq i \leq k$ v $pref(T)$ je dvojice $(first, last)$, kde:

- pokud $P_i = \varepsilon$, pak $1 < (first = last) \leq n + 1$,
- pokud $P_i \neq \varepsilon$, pak $1 \leq (first < last) \leq n + 1$

Ve druhém případě se dá P_i zapsat jako $a_{first} a_{first+1} \dots a_{last-1}$. Množina všech výskytů P_i v $pref(T)$ je denotována $occ^T(P_i)$, případně jen $occ(P_i)$, pokud je strom T zřejmý z kontextu. [1]

Příklad 8. V tomto příkladu opět využiju stejného stromu T' a vzorku P', jako v příkladu 5. Podvzorek $P_2 = b0$ vzorku P' se ve stromě T' vyskytuje celkem třikrát. To znamená, že $occ^{T'}(P') = \{(3,4),(4,5),(5,6)\}$.

Definice 17. *Prefix stromového vzorku*, označený $TPP(P)$ z anglického „tree pattern prefix“, je libovolný prefix řetězce $pref(P) = P_1 S\ P_2 S\ \dots S\ P_k$ končící symbolem $P_i, 1 \leq i \leq k$ nebo S. [1]

Příklad 9. Možné prefixy stromového vzorku $pref(P') = a4_1 S_2 b0_3$ jsou $TPP(P') = \{a4, a4S, a4S\ b0\}$.

Výskyt celého $TPP(P) = P_1 S\ P_2 S\ \dots S\ P_k$ je dvojice $(first, last)$, kterou lze získat spojením hodnot $first_1$ z $occ(P_1)$ a $last_k$ z $occ(P_k)$. Je to tedy rozsah všech jednotlivých podvzorků. P_1 končí tam, kde začíná první S a to končí tam, kde začíná P_2. Proto dotázáním se SJT tabulky indexem $last_1$ je stejné jako dotázání se pozicí podstromu, který S zastupuje, takže platí $SJT[last_1] = first_2$ z P_2.

Příklad 10. Nechť $TPP_1(P') = a2S$ je prefix stromového vzorku P' z příkladu 5 a T' strom z příkladu z příkladu 1. Pak $occ^{T'} = \{(1,5),(2,4)\}$. Uzel $a2$ nacházející se na první pozici končí hned na druhé, kde začíná podstrom zastoupený S. $SJT[2] = 5$. Analogicky lze zjistit dvojici $(2,4)$.

Následuje popis algoritmů potřebných k dokončení vyhledávací fáze.

Algoritmus 3 Kontrola aritního kontrolního součtu [4]

Název: VerifyArityChecksum
Vstup: Řetězec nad ohodnocenou abecedou $str = a_1 a_2 \dots a_n, n \geq 1$.
Výstup: Rozhodnutí, zda $str = pref(T)$ pro strom T.

1: $ac(T) \leftarrow 1$
2: **for** $i = 1$ n **do**
3: $\quad ac(str) \leftarrow ac(str) + arity(a_i) - 1$
4: $\quad$ **if** $i < n$ and $ac(str) = 0$ **then**
5: $\quad\quad$ **return** $false$
6: **return** rozhodnutí, zda $ac(str) = 0$

Výpočet aritního kontrolního součtu je již popsán v definici 9. Rekurzivní implementace je takto triviální.

Algoritmus 4 Hledání výskytů podvzorku [9]

Název: FindOccurrences
Vstup: Kompaktní suffixový automat $CSA(pref(T))$, podvzorek P_i.
Výstup: Množina $occ^T(P_i)$.

1: q je stav CSA dosažený po zpracování P_i.
2: najdi všechny cesty od q do finálního stavu ($length$ je délka každé cesty)
3: přidej výskyt $(n - length - |P_i|, n - length)$ do $occ^T(P_i)$
4: **return** $occ^T(P_i)$

Celý algoritmus `FindOccurrences` 4 je popsán jen takto slovně, jelikož pracuje s vnitřkem CSA a tudíž je velmi závislá na jeho reprezentaci.

Algoritmus 5 Sjednocování výskytů [1]

Název: MergeOccurencies
Vstup: Množiny výskytů $prevOcc = occ^T(TPP(P))$ a $subOcc = occ^T(P_k)$, tabulka $Rev_{\{\}}^{|pref(T)|}$.
Výstup: Množina výskytů $mergedOcc$ jako sjednocení $prevOcc$ a $subOcc$.

1: $mergedOcc = \{\}$
2: **for all** $(first, last) \in prevOcc$ **do** $Rev_{prevOcc}^{|pref(T)|}[last] \leftarrow first$
3: **for all** $(first', last') \in subOcc$ **do**
4: $\quad$ **if** $Rev_{prevOcc}^{|pref(T)|}[first'] \neq -1$ **then**
5: $\quad\quad$ vlož $(Rev_{prevOcc}^{|pref(T)|}[first'], last')$ do $mergedOcc$
6: **for all** $(first, last) \in prevOcc$ **do** $Rev_{\{\}}^{|pref(T)|}[last] \leftarrow -1$
7: **return** $mergedOcc$

Množina $subOcc$ je výstupem z posledního volání `FindOccurrences` 4 a $prevOcc$ sjednocením výskytů ze všech předešlých. Návratová hodnota `Mer-`

`geOccurrences` 5 je *mergedOcc*, což je sjednocení výskytů *prevOcc* a *subOcc*. Na řádku 6 nemá tabulka Rev dolní index, protože se jedná o přípravu na příští běh této metody a tou dobou nemusí být množina *prevOcc* stejná.

Konečně se dostávám v hlavnímu algoritmu, který všechny 3 předešlé spojuje do jednoho celku.

Algoritmus 6 Hledání výskytů stromového vzorku [1]

Název: MatchPattern

Vstup: Stromový vzorek $pref(P) = P_1 S\ P_2 S\ \ldots P_k$, kompaktní suffixový automat $CSA(pref(T))$, podstromová přechodová tabulka $SJT(T)$, tabulka $Rev_{\{\}}^{|pref(T)|}$.

Výstup: Množina výskytů stromového vzorku P.

1: **if** not $VerifyArityChecksum(pref(P))$ **then**
2: chyba – nevalidní vzorek P
3: $prevOcc = \{\}$
4: **for** $i \leftarrow 1\ k$ **do**
5: **if** $P_i \neq \varepsilon$ **then**
6: $occ \leftarrow FindOccurrences(CSA, P_i)$
7: **if** $i = 1$ **then**
8: $prevOcc \leftarrow occ$
9: **else**
10: $prevOcc \leftarrow MergeOccurrences(prevOcc, occ, Rev_{\{\}}^{|pref(T)|})$
11: **if** $i \neq k$ **then**
12: **for all** $(first, last) \in prevOcc$ **do**
13: $(first, last) \leftarrow (first, SJT[last])$
14: **return** *prevOcc*

Příklad 11. V tomto příkladu zopakuji příklad 5, tentokrát ale čistě strojovým postupem – spuštěním algoritmu 6. Nechť existuje strom T' s prefixovou notací $pref(T') = a2_1a2_2b0_3b0_4b0_5$ a stromový vzorek P' s prefixovou notací $pref(P') = P_1SP_2$, kde $P_1 = a2$ a $P_2 = b0$.

Parametry ke spuštění MatchPattern 6 jsou $CSA(pref(T'))$ a tabulka $SJT(T')$ vyobrazená v příkladu 4. Aritní kontrolní součet je v pořádku, protože $ac(pref(P')) = 0$. Podvzorky jsou dva, proto běh skončí po dvou iteracích.

i = 1: Metoda `FindOccurrences` vrátí množinu výskytů $\{(1,2),(2,3)\}$. Jelikož $i = 1$, okamžitě je tato množina přiřazena *prevOcc*. Po nahlédnutí do SJT se přemění do $\{(1,6),(2,5)\}$.

i = 2: Výskyt $b0$ je objeven na $\{(3,4),(4,5).(5,6)\}$. Nyní je potřeba tuto množinu sjednotit s *prevOcc*. Výsledkem bude nezměněná $\{(1,6),(2,5)\}$, jelikož množinu nalezenou v této iteraci kompletně pokrývá. Jedná se o poslední iteraci, proto se již hodnoty *last* nemění podle SJT.

Kapitola 3

Realizace

Při náhledu na implementaci budu v této kapitole soustředit svou pozornost na soubory tak, jak vznikaly chronologicky. Jestli tedy např. nějaká metoda vznikla čistě kvůli konkrétní funkcionalitě, zmíním ji až po představení oné funkcionality. Všechen relevantní kód k této práci je možné najít v příloze.

Většina dění knihovny, které tato práce pokrývá, se odehrává v podsložkách `alib2data` a `alib2algo`. Složka `alib2data` obsahuje veškeré datové struktury knihovny a je to místo, kde jsou nakonec drženy všechny *data*. Složka `alib2algo` má na starosti různé možnosti sestrojení a jiné operace nad strukturami v `alib2data`, jako např. determinizace či minimalizace automatů.

3.1 Konstrukce kompaktního suffixového automatu

Před realizací konstrukce s lineární paměťovou složitostí jsem implementoval naivní algoritmus podle sekce 2.4.1. Efektivnější varianta využívá stejný stavební kámen, tudíž implementace naivní verze není zbytečná. Z důvodů popsaných v sekci 2.4.2 ale tato práce lineární řešení implementovat nebude. Algoritmus konstrukce je v knihovně vždy oddělen od příslušné datové struktury. To znamená, že vyvíjený index se při změnách v takovém algoritmu konstrukce buď nemění, nebo jen minimálně.

V `alib2data/src/indexes` jsem vytvořil složku `compactSuffixAutomaton`, kde je index definován.

Soubory následují stejnou konvenci jako již existující automat `CompactNFA`. První znatelnou změnou je způsob uchovávání přechodů mezi stavy automatu - mapa `transitions`.

```
1 std::map <
2    std::pair <automaton::State, alphabet::Symbol>,
3    std::pair <automaton::State, string::LinearString>
4 > transitions;
```

Protože CSA je deterministický, k identifikaci každého přechodu stačí stav, ze kterého vychází a první symbol tohoto přechodu. Tento klíč získá hodnotu v podobě stavu, do kterého přechod míří a celý řetězec tohoto přechodu.

Obvyklé metody jako `addTransition` nebo `removeTransition` se od implementací v ostatních datových strukturách liší jen rozdílnou reprezentací přechodů. První úplně novou metodou je ale `matchTransition`.

```
1 std::pair <automaton::State, string::LinearString>
2 CompactSuffixAutomaton::matchTransition (const std::pair <automaton::State,
3       alphabet::Symbol> & key ) const {
4    auto it = transitions.find(key);
5    if (it == transitions.end())
6       return std::make_pair(key.first,
7          std::move(string::LinearString()));
8    return it -> second;
9 }
```

Tato metoda se pokusí v mapě nalézt hledaný přechod. Pokud jej nenalezne, vrátí stav z klíče zpět a místo textu přechodu vrátí prázdný řetězec. Vytváření prázdného řetězce má smysl kvůli konstrukci automatu – zaznamenávají se do něj symboly, po kterých algoritmus „ušel“ mezi dvěma stavy.

Další metodou indexu je `slowFind`. Než se k ní ale dostanu, dává smysl přednostně představit algoritmus konstrukce, který ji využívá. Implementace konstrukce leží ve složce `alib2algo/src/stringology/indexing`. Důležité je poznamenat, že namespace konstrukce je `indexing`, ale samotná datová struktura indexu leží v namespace `indexes`. Toto od sebe rozlišuje dvě třídy, obě nesoucí název `CompactSuffixAutomaton`.

```
1  indexes::CompactSuffixAutomaton
2  CompactSuffixAutomaton::naiveConstruct(const string::LinearString& text) {
3     automaton::State initialState ('I');
4     indexes::CompactSuffixAutomaton csa(initialState);
5     csa.addFinalState(initialState);
6
7     automaton::State finalState ('F');
8     csa.addState(finalState);
9     csa.addFinalState(finalState);
10
11    csa.setInputAlphabet(text.getAlphabet());
12    int stateNumber = 0;
13 ...
```

Řádky 4–11 inicializují automat a nastavují počáteční stav I a konečný F. Stav I je také konečný, aby byl schopen přijímat prázdný řetězec jako validní suffix libovolného textu. Metoda `setInputAlphabet` jen přenese definici vstupní abecedy ze vstupního textu na automat. Vnitřní stavy (rozdílné od I a F) jsou číslovány od nuly v pořadí jejich vytvoření. K tomu zde slouží

proměnná `stateNumber`.

```
13 ...
14 for (unsigned i = 0; i < text.size(); i++) {
15    unsigned tailStart = i;
16    std::pair<automaton::State, string::LinearString> lastStateAndSymbolsFound
          = csa.slowFind(initialState, text, tailStart);
17 ...
```

K implementaci metody `slowFind` se dostanu za chvíli, ale pro zatím stačí znát skutečnost, že parametr `tailStart` přijímá jako nekonstantní referenci. Od řádku 4 je v něm uložena počáteční pozice suf_i, ze které se po dokončení běhu slowFind stane pozice začátku $tail_i$. Návratovou hodnotou je dvojice posledního nalezeného stavu a textu shodujícího se s prefixem některé z vycházejících hran. Jakmile běh `slowFind` skončí, stačí už jen rozhodnout, jestli je třeba vytvářet nové přechody, případně jakým způsobem.

```
17 ...
18    // if slowFind finished exactly on a state
19    if (lastStateAndSymbolsFound.second.isEmpty()) {
20       if (tailStart < text.size())
21          csa.addTransition(lastStateAndSymbolsFound.first, text.getTail(
                tailStart), finalState);
22       else
23          csa.addFinalState(lastStateAndSymbolsFound.first);
24    }
25 ...
```

První možností je, že `slowFind` skončil přímo na stavu. Pokud nešlo už dále jít po žádném přechodu, ale v řetězci, který má být vložen, stále ještě zbývají nějaké znaky ($tail_i$ je neprázdný řetězec), pak stačí vyvést přechod do stavu F. Naivní konstrukce, narozdíl od lineární, neobsahuje žádné přepojování hran či duplikování stavů, proto nemusím s označováním koncových hran čekat až na úplný konec. Pokud je tedy $tail_i$ prázdný řetězec, mohu poslední nalezený stav označit jako koncový ihned, tedy tak, jak to dělá `else` větev na řádku 26.

```
25 ...
26    // if slowFind finished on the "middle" of a transition
27    else {
28       automaton::State newState = automaton::State(stateNumber++);
29       csa.addState(newState);
30
31       std::pair<automaton::State, alphabet::Symbol> originalTransitionKey =
             std::make_pair(lastStateAndSymbolsFound.first,
             lastStateAndSymbolsFound.second.getFirstSymbol());
32       std::pair<automaton::State, string::LinearString> originalTransition =
             csa.matchTransition(originalTransitionKey);
33       string::LinearString originalTransitionRemainder = originalTransition.
             second.getTail(lastStateAndSymbolsFound.second.size());
```

```
34
35      csa.removeTransition(originalTransitionKey);
36      csa.addTransition(lastStateAndSymbolsFound.first,
            lastStateAndSymbolsFound.second, newState);
37      csa.addTransition(newState, originalTransitionRemainder,
            originalTransition.first);
38
39      if (tailStart < text.size())
40         csa.addTransition(newState, text.getTail(tailStart), finalState);
41      else
42         csa.addFinalState(newState);
43   }
44 }
45 return csa;
```

Druhá možnost představuje případ, kdy vkládaný suffix skončil „uprostřed“ nějakého přechodu. Kdykoli taková situace nastane, zaručeně to v naivní verzi vede k vytvoření nového stavu. Pomocí návratové hodnoty `slowFind` sestavím na řádku 34 klíč z získání řetězce přechodu a cílového stavu. Ty představuje dvojice na řádku 35. Nyní je potřeba tento přechod rozdělit na dvě části, které jsou určené přesně tím, kam `slowFind` došel. První část už mám uloženou v `lastStateAndSymbolsFound.second`, proto pomocí velikosti tohoto řetězce vytvořím rozdíl a vznikne mi druhá část – `originalTransitionRemainder`.

Teď už stačí původní přechod odebrat a nahradit nově vytvořenými dvěmi částmi (řádky 38–40). Opět, jako v prvním případě, pokud je $tail_i$ neprázdný řetězec, mohu vyvést hranu do koncového stavu F, jinak mohu současný stav ihned označit jako koncový.

V této metodě byl použit `slowFind` a `getTail`, jejichž implementace jsou stále ještě neznámé. I volání `getFirstSymbol` a `size` nad typem `LinearString` jsou představeny touto bakalářskou prací, ale nejedná se o nic víc než triviální gettery dělající přesně to, o čem jejich pojmenování vypovídá. Nejprve tedy ukážu metodu `getTail`, protože je velice přímočará. Týká se výhradně datového typu `LinearString`, který se nachází ve složce `string` v `alib2data`.

```
1 LinearString LinearString::getTail (int tailStart) const {
2    std::vector < alphabet::Symbol > newData;
3    newData.insert(newData.end(), m_Data.begin() + tailStart, m_Data.end());
4    return LinearString(newData);
5 }
```

Nejde o nic více než o zkopírování intervalu `tailStart`–konec do nového řetězce. Nyní se ze složky `string` přesouvám zpět do souboru s indexem (`indexes`, stále v `alib2data`).

```
1 std::pair < automaton::State, string::LinearString >
2 CompactSuffixAutomaton::slowFind ( automaton::State from,
3      const string::LinearString & text, unsigned & tailStart) {
```

```
4     while (tailStart < text.size()) {
5       std::pair < automaton::State, string::LinearString > transition =
            matchTransition(std::make_pair(from, text.getContent().at(tailStart)
            ));
6
7       if (transition.second.isEmpty())
8         return transition;
9
10      string::LinearString symbolsSinceP = text.getCommonPrefix(transition.
            second, tailStart);
11      if (symbolsSinceP.size() < transition.second.size())
12        return std::make_pair(std::move(from), std::move(symbolsSinceP));
13
14      from = transition.first;
15    }
16    return std::make_pair(std::move(from), string::LinearString());
17  }
```

Naivní konstrukce ještě nevyužívá `slowFind` v plné své kráse. Pro lineární verzi bude mít navíc za úkol duplikovat stavy při nalezení nepevné hrany. Zatím stačí chození po hranách, dokud nevyčerpá všechny znaky vstupního textu. Jak jsem uvedl již dříve v této sekci, na začátku metody představuje proměnná `tailStart` začátek $suffix_i$. Tato pozice se posová a postupně se z ní stane $tail_i$. V zásadě tato metoda může skončit dvěma způsoby.

Může skončit na nějakém stavu tak, že už není dále kam jít (tento stav je bod, kde vstupní text skončil, nebo žádná hrana se zbylými znake textu nesdílí žádný prefix). V takovém případě na řádku 8 jednoduše předá to, co vrátila metoda `matchTransition`.

Zbylé symboly vstupního textu se také můžou rovnat prefixu některé z hran. Pokud se tímto prefixem rozumí celá délka hrany, pak metoda musí pokračovat hlouběji do automatu. Jestli nakonec skončí „uprostřed“ nějaké hrany, vrátí poslední potkaný stav a společný prefix této hrany a zbytku textu.

Nyní se dostávám k implementaci metody `getCommonPrefix`, použité na řádku 10. Týká se opět třídy `LinearString`. Jejím posláním je vytvořit nový řetězec – společný prefix – daný startovní pozicí `tailStart` v řetězci, nad kterým je volána a druhým řetězcem `s`.

```
1   LinearString LinearString::getCommonPrefix (const LinearString & s,
2       unsigned & tailStart) const {
3     const std::vector < alphabet::Symbol > & sData = s.getContent();
4     std::vector < alphabet::Symbol > newData;
5     size_t tailSize = size() - tailStart;
6     size_t lowerSize = (tailSize < sData.size() ? tailSize : sData.size());
7     unsigned i = 0;
8
9     for (; i < lowerSize && m_Data[tailStart + i] == sData[i]; i++);
10
11    newData.insert(newData.end(), sData.begin(), sData.begin() + i);
12    tailStart += i;
```

```
13
14      return LinearString(newData);
15  }
```

s

Jelikož řetězec, nad kterým je metoda volána, v tomto kontextu skutečně začíná až od indexu `tailStart`, odečtením této pozice od celkové velikosti získám relevantní velikost – `tailSize`. Cyklus na řádku 8 je tu čistě z důvodu obdžení hodnoty `i`, která představuje pozici, kde se buď symboly dvou řetězců poprvé nerovnají, nebo jeden z nich končí. Nový řetězec, který chci vrátit, je tedy interval od začátku `sData` do `i`-té pozice. Před ukončením metody si ještě zvýším `tailStart` o `i`, protože shodná část dvou řetězců je od teď zaručeně součást *head*, ovšem `tailStart` musí ukazovat na začátek *tail*.

3.2 Hledání výskytů stromových vzorků

Algoritmus`VerifyArityChecksum` 3 již v knihovně existuje, proto není třeba jej znovu implementovat. Je volán při sestavení objektu reprezentující prefixovou notaci stromu (např. `PrefixRankedTree`) nebo zavolání jeho setteru.

Metoda `findOccurrences`, popsaná algoritmem 4, se prakticky celá týká dění uvnitř CSA, proto dává smysl umístit ji do stejné třídy (`indexes::CompactSuffixAutomaton`).

bd::map < unsigned, unsigned > CompactSuffixAutomaton::findOccurrences (const string::LinearString subpattern, unsigned subjectSize) std::map < unsigned, unsigned > occ; unsigned i = 0; automaton::State currentState = getInitialState(); // process subpattern while (i < subpattern.size()) std::pair < automaton::State, string::LinearString > transition = matchTransition(currentState, subpattern.getContent().at(i)); i += transition.second.size(); currentState = transition.first; // find paths from current state to all final states ahead findAllPaths(currentState, occ, i - subpattern.size(), subjectSize, subpattern.size()); return occ;

Do metody si posílám jako parametr podvzorek (již zpracovaný ze stromové struktury `PrefixRankedPattern` do textové `LinearString`) a velikost indexovaného stromu. Cílem zde je vyčerpat celou délku vzorku od počátečního stavu CSA. Volání `findAllPaths` na řádku 12 pak od tohoto bodu rekurzivně vyhledá všechny možné cesty do finálních stavů. Kromě předání množiny výskytů, délky subjektu a vzorku ještě potřebuje stav, od kterého má hledat cesty a napočítanou délku od tohoto stavu, do které pak bude rekurzivně přičítat. Pro zavolání této metody z `findOccurrences` je potřeba zvážit dva případy dokončení běhu cyklu na řádku 6:

Cyklus skončil přesně na nějakém stavu. V takovém případě metodě `findAllPaths` můžu předat tento stav a délku uražené cesty inicializovat na 0.

Cyklus skončil „uprostřed“ nějaké hrany. Stačí si uvědomit, že všechny cesty nalezené `findAllPaths` musí jít přes stav na druhém konci této hrany. Proto stav, od kterého půjde, bude ten, do kterého tato hrana směřuje a délka uražené cesty rozdíl mezi délkou do tohoto stavu a skutečnou délkou vzorku.

Obě varianty lze zobecnit na jedno volání `findAllPaths`, protože pokud je rozdíl `i - subpattern.size()` roven 0, jde přesně o první případ a pokud větší než 0, jedná se o druhý. Návratovou hodnotou je mapa, kde klíč je *first* a hodnota *last* výskytu $(first, last)$. Takto budou reprezentovány výskyty až do finální transformace v hlavním algoritmu, o kterém budu mluvit později.

```
1  void CompactSuffixAutomaton::findAllPaths (automaton::State from,
2       std::map < unsigned, unsigned > & occ, unsigned pathLength,
3       unsigned subjectSize, unsigned subpatternSize) {
4     // if from is final, add path to occurencies
5     if (getFinalStates().find(from) != getFinalStates().end())
6        occ.insert( {subjectSize - pathLength - subpatternSize, subjectSize -
             pathLength} );
7
8     // look for further transitions from current state
9     for (const alphabet::Symbol & symbol : getInputAlphabet()) {
10       auto it = transitions.find({from, symbol});
11       if (it != transitions.end())
12          findAllPaths(it->second.first, occ, pathLength + it->second.second.
                size(), subjectSize, subpatternSize);
13    }
14 }
```

Metoda `findAllPaths` zaznamená výskyt podvzorku v subjektu, pokud stav `from` je konečný. Cesta ale končí až poté, co z tohoto stavu nevycházejí žádné hrany – to je přesně stav, který v analytické části označuji jako F. Na každou odchozí hranu je jinak zavolána tato metoda rekurzivně, kde stav, do kterého směruje, je nový `from` a k uražené délce cesty je přičtena délka hrany.

Implementací těchto dvou metod je završena diskuze o algoritmu `FindOccurrences` 4. Nyní přichází na řadu `MergeOccurrences` 5. Tento a také hlavní algoritmus 6 už ale nejsou součástí CSA, nýbrž arbologické části knihovny. Třída, ve které se oba nacházejí, je umístěna v `alib2algo/src/arbology/exact` a jmenuje se `LinearTreePatternMatch`.

```
1  std::map < unsigned, unsigned > LinearTreePatternMatch::mergeOccurrences (
2       const std::map < unsigned, unsigned > & prevOcc,
3       const std::map < unsigned, unsigned > & subOcc,
4       std::vector < int > & rev) {
5     std::map < unsigned, unsigned > mergedOcc;
6
7     for (const std::pair<unsigned, unsigned> & occ : prevOcc)
8        rev[occ.second] = occ.first;
9     for (const std::pair<unsigned, unsigned> & occ : subOcc) {
```

```
10        if (rev[occ.first] != -1)
11           mergedOcc.insert( { rev[occ.first], occ.second } );
12     }
13     for (const std::pair<unsigned, unsigned> & occ : prevOcc)
14        rev[occ.second] = -1;
15     return mergedOcc;
16  }
```

Zdrojový kód této metody snad nejlépe z celé práce odráží znění algoritmu v analytické části 5. Každý řádek metody přímo koresponduje s nějakým v pseudokódu. Nově je tu zmínka o poli `rev`, jež je reprezentováno vektorem kvůli jeho požadavku na dynamičnost. Způsob inicializace tohoto pole je vidět v následující metodě.

Algoritmus `MatchPattern` 6 je reprezentován metodou `match`. Rozdíl v pojmenování je kvůli zachování konvence vůči ostatním algoritmům v knihovně – všechny používají právě volání `match`.

```
1  std::set < unsigned > LinearTreePatternMatch::match (
2        const tree::PrefixRankedTree & subject,
3        const tree::PrefixRankedPattern & pattern ) {
4     std::vector < int > sjt = SubtreeJumpTable::compute ( subject );
5
6     // prepare LinearString constructor parameters with distinct ranked symbols
7     std::set < alphabet::Symbol > alphabet;
8     std::vector < alphabet::Symbol > subjectData;
9     for (const alphabet::RankedSymbol & rSymbol : subject.getContent()) {
10       alphabet::Symbol symbol = alphabet::symbolFrom( (std::string) rSymbol );
11       alphabet.insert(symbol);
12       subjectData.push_back(symbol);
13    }
14    string::LinearString subjectString(alphabet, std::move(subjectData));
15    indexes::CompactSuffixAutomaton csa = stringology::indexing::
          CompactSuffixAutomaton::naiveConstruct(std::move(subjectString));
16 ...
```

Podstromová přechodová tabulka v knihovně byla již před touto prací, proto se její konkrétní implementací nebudu zabývat.

Jelikož CSA je datová struktura stringologie pracující nad řetězcovým typem `LinearString`, musím nejprve z prefixové notace typu `PrefixRankedTree` takový řetězec vyrobit. V cyklu začínajícím na řádku 9 postupně přetypuji ohodnocené symboly typu `RankedSymbol` na `Symbol` takovým způsobem, aby se rovnaly jen symboly se stejným původním názvem a aritou. Toto za mě dělá přetypování ohodnoceného symbolu do `std::string`, které vyprodukuje formát `<jméno>_<arita>`, tedy např. `a_0` pro symbol a arity 0.

Rozdělování vzorku na podvzorky se v této implementaci děje rovnou za běhu algoritmu. Jejich datový typ bude `LinearString`, a to stejným způsobem jako subjekt. Pole *Rev* 14 je reprezentováno vektorem z důvodu požadavku

```
16 ...
17    std::vector<alphabet::Symbol> subpatternData;
18    std::map < unsigned, unsigned > prevOcc;
19    std::vector < int > rev;
20    rev.resize(subject.getContent().size() + 1);
21    std::fill(rev.begin(), rev.end(), -1);
22
23    for (const alphabet::RankedSymbol & rSymbol : pattern.getContent()) {
24       if (rSymbol != pattern.getSubtreeWildcard())
25          subpatternData.push_back(alphabet::symbolFrom((std::string)rSymbol));
26       // append symbols to subpattern until wildcard is found
27       if (rSymbol == pattern.getSubtreeWildcard() || rSymbol == pattern.
             getContent().back()) {
28          string::LinearString subpattern(alphabet, subpatternData);
29          subpatternData.clear();
30
31          if (subpattern.size() > 0) {
32             std::map < unsigned, unsigned > occ = csa.findOccurrences(
                   subpattern, subject.getContent().size());
33             prevOcc = prevOcc.empty() ? occ : mergeOccurrences(prevOcc, occ,
                   rev);
34          }
35          // look for subtree end in SJT
36          for (std::pair<const unsigned, unsigned> & occ : prevOcc)
37             occ.second = sjt[occ.second];
38       }
39    }
40 ...
```

na jeho proměnnou velikost. Hodnoty jsou inicializovány na -1.

Od prvního do posledního jsou procházeny znaky vzorku a při nalezení zástupného symbolu S, případně při doražení na konec vzorku je vytvořen podvzorek ze symbolů nahromaděných ve vektoru `subpatternData`. Samotné vyhledávání výskytů začíná na řádku 32 a na následujícím případné sjednocování výskytů. Řádek 36 řeší upravování hodnoty *last* ze SJT.

V analytické části mluvím o výskytech jako o dvojicích $(first, last)$, kde $first$ je pozice, na které je vzorek zakořeněn a *last* pozice, kde už začíná jiný podstrom. V této knihovně jsou výskyty ale reprezentovány jen pozicemi $first$ s tím, že se lze podívat do SJT na příslušný index a zjistit tak *last*. Přepínání mezi těmito variantami výstupu ukážu v sekci 4.2.

```
40 ...
41    // return set of first only
42    std::set<unsigned> occResult;
43    for (const std::pair<unsigned, unsigned> & occ : prevOcc)
44       occResult.insert(occ.first);
45    return occResult;
46 }
```

Před vrácením množiny výskytů tedy vytvořím novou množinu skládající se čistě z hodnot $first$ každé dvojice $(first, last) \in prevOcc$.

3.3 Testování

V této sekci popíšu způsob testování implementace knihovny v rámci této práce. Vzhledem k počtu řádků, na který se unit testy většinou rozmáhají, sem nebudu vkládat zdrojové kódy. Jsou v nich navíc použity operace, o kterých jsem se v praktické části již zmínil. Kódy jsou samozřejmě k dispozici v příloze.

3.3.1 Kompaktní suffixový automat

Ve složce `alib2algo/test-src/stringology/indexing` lze nalézt unit test `CompactSuffixAutomatonTest`. Jedná se o vytvoření automatu $CSA(abaabc)$ z příkladu 3 dvakrát – poprvé zavoláním algoritmu naivní konstrukce a podruhé ručním sestavením. Výsledkem tohoto testu je skutečnost, zda jsou tyto dva automaty identické.

3.3.2 Hledání výskytů stromových vzorků

Pro tento test se přesouvám ze stringologie do arbologie, tedy do složky `alib2algo/test-src/arbology/exact`. Reprezentuje jej třída `LinearTreePatternMatchTest`, která obsahuje 2 unit testy:

(1) První test replikuje příklad 5, tj. hledání vzorku $pref(P') = a2_1S_2b0_3$ ve stromu $pref(T') = a2_1a2_2b0_3b0_4b0_5$.

(2) Druhý test je o trochu větší varianta prvního. Replikuje příklad použitý v [1]. Subjektem je strom $pref(T'') = a4_1a4_2a4_3a0_4b0_5a0_6a0_7a0_8b0_9a0_{10}$-$a0_{11}a0_{12}b0_{13}$ a vzorkem $pref(P'') = a4_1S_2a0_3S_4S_5$.

Kapitola 4

Ukázky spuštění

Po kompilaci knihovny příkazem `make release`, resp. `make debug` se v kořenovém adresáři vytvoří složky `bin-release`, resp. `bin-debug`. Všechny volané konzolové příkazy v této sekci předpokládají, že se čtenář nachází v jedné z těchto složek.

4.1 Kompaktní suffixový automat

CSA pracuje s řetězci typu `LinearString` a proto patří do oblasti stringologie. Před konstrukcí je tedy nutné od někud tento řetězec získat, např. vygenerovat náhodný. Veškeré náhodné generování struktur v této knihovně se děje v části `arand2`. Všechny příkazy knihovny podporují přepínač `-h`, který vypíše seznam všech přepínačů a možností daného příkazu. Nahlédnu tedy na možnosti `arand2`:

```
$ ./arand2 -h
```

Pro specifikování generování řetězce slouží `-t ST`, dále zajímavý je přepínač `--length <číslo>` ovládající délku řetězce. Řetězec délky 10 je vytvořen následujícím příkazem:

```
$ ./arand2 -t ST --length 10
```

Vygenerovaný řetězec je XML formát obsahující nejprve definici abecedy a poté posloupnost znaků. Řetězec potřebný pro konstrukci CSA mám. Pojďme se podívat na nápovědu od stringologie.

```
$ ./astringology2 -h
```

Zajímavé jsou přepínače `-a` a `-s`. První je určen k výběru algoritmu (tato práce implementuje `-a compactSuffixAutomaton`) a druhý k načtení řetězce ze souboru. CSA lze tedy zkonstruovat takto:

```
$ ./astringology2 -a compactSuffixAutomaton -s str
```

Soubor `str` zde představuje řetězec, např. vygenerovaný výše zmíněným příkazem. Řetězec ale není nutné načítat ze souboru příkazem `-s` – lze použít i přesměrování konzolového výstupu.

```
$ ./arand2 -t ST --length 10 | ./astringology2 -a compactSuffixAutomaton
```

Na závěr bych rád poznamenal, že takovéhle přímé konstruování CSA je určeno jen k porovnání, co konkrétní řetězec vytvoří. K použití CSA pro vyhledávání vzorků si jej hledací algoritmus vytvoří sám. Použití je popsáno v následující sekci.

4.2 Hledání výskytů stromových vzorků

Pro tuto ukázku využiju příklad 5, aneb hledání vzorku $pref(P') = a2_1S_2b0_3$ ve stromu $pref(T') = a2_1a2_2b0_3b0_4b0_5$. Stejnou věc také dělá první unit test 3.3.2 po kompilaci `alib2algo`. Nejprve se podívám, co arbologická část knihovny nabízí:

```
$ ./aarbology2 -h
```

Zajímá mě především to, že požadovaný algoritmus zadám po přepínači `-a`, vstupní subjekt po `-s` a vzorek po `-p`. Hledání tedy můžu zahájit takto:

```
$ ./aarbology2 -a linearTreePatternMatch -s t2_pre -p p2_pre
```

Soubor `t2_pre` zde reprezentuje strom T' v datovém typu `PrefixRankedTree`. Obdobně `p2_pre` reprezentuje vzorek P' typu `PrefixRankedTree`. Nyní stojí otázka, jak přesně vypadá obsah těchto dvou souborů, případně zda je možné je nějakým způsobem vygenerovat.

```
$ ./arand2 -t RT | ./acast2 -t PrefixRankedTree
$ ./arand2 -t RP | ./acast2 -t PrefixRankedPattern
```

Stejně jak v prvním příkladu spuštění, poslouží příkaz `arand2`. Volby `-t RT` a `RP` vygenerují datové typy `RankedTree` a `RankedPattern`. Při nahlédnutí do nápovědy přepínačem `-h` lze vidět, že hloubka stromu, počet uzlů,

apod. jsou volitelné. Tyto vygenerované objekty ale ještě nejsou v prefixové notaci, s čímž pomůže `acast2`.

Takto vygenerovaný vzorek bude mít jen velice nepravděpodobně výskyt v takto vygenerovaném stromu. Jedná se ale o XML formát obsahující nejprve definici abecedy a pak posloupnost samotných dat a není vůbec těžké se v něm zorientovat. Doporučuji výstup ještě před tím prolít nějakým pretty-printem, např. `xmllint`.

Po spuštění algoritmu (viz výše), kde `t2_pre` obsahuje $pref(T')$ a `p2_pre` obsahuje $pref(P')$, dostanu následující výstup (zformátován přes `xmllint`):

```
<?xml version="1.0"?>
<Set>
  <Unsigned>0</Unsigned>
  <Unsigned>1</Unsigned>
</Set>
```

Výskyt P' v T' je množina $\{(1,6),(2,5)\}$, v analytické části ale pro přehlednost indexovaná od jedničky. V implementaci jsou výskyty samozřejmě indexovány od 0, proto výstup odpovídá hodnotám $first$ z této množiny. Jak ale zjistit hodnoty $last$? Stačí algoritmus spustit znovu s přepínačem `-e`.

```
$ ./aarbology2 -a linearTreePatternMatch -s t2_pre -p p2_pre -e |
  xmllint --format -
```

Výstup pak odpovídá hodnotám $last$ z množiny.

```
<?xml version="1.0"?>
<Set>
  <Unsigned>4</Unsigned>
  <Unsigned>5</Unsigned>
</Set>
```

Závěr

V této práci jsem se zabýval novým přístupem indexace stromových struktur, a sice s využitím kompaktního suffixového automatu. Stromové vzorky hledané v indexovaném stromu mohou obsahovat mezery variabilních délek, s čímž využitý algoritmus počítá.

Literatura prezentující algoritmus konstrukce kompaktního suffixového automatu mi bohužel neposkytla odpovědi na všechny otázky, které jsem potřeboval k úspěšnému dokončení implementace efektivní konstrukce. Naivní konstrukci jsem však implementoval úspěšně. Mechanismus vyhledávající výskyty vzorků v indexovaném stromu by se tedy stále dal rozšířit o implementaci kompaktního suffixového automatu s lineární časovou i paměťovou složitostí.

Literatura

[1] Janoušek, J.; Melichar, B.; Polách, R.; aj.: *Descriptional Complexity of Formal Systems*, kapitola A Full and Linear Index of a Tree for Tree Patterns. Springer International Publishing, 2014, s. 198–209.

[2] Bille, P.; Gørtz, I. L.; Vildhøj, H. W.; aj.: *Algorithm Theory – SWAT 2012: 13th Scandinavian Symposium and Workshops*, kapitola String Indexing for Patterns with Wildcards. Springer Berlin Heidelberg, 2012, s. 283–294.

[3] Grossi, R.; Silvestri, F.; Sebastiani, F.: *String Processing and Information Retrieval*, ročník 7024. Springer-Verlag Berlin Heidelberg, 2011.

[4] Melichar, B.; Janoušek, J.; Flouri, T.: Arbology: Trees and pushdown automata. In *Kybernetika*, ročník 48, 2012, s. 402–428.

[5] Cleophas, L. G. W. A.: *Tree Algorithms: Two Taxonomies and a Toolkit.* Dizertační práce, Technische Universiteit Eindhoven, 2008.

[6] Comon, H.; Dauchet, M.; Gilleron, R.; aj.: Tree Automata Techniques and Applications. Dostupné na: `http://www.grappa.univ-lille3.fr/tata`, 2008, vydáno 12. října.

[7] Bille, P.; Li Gørtz, I.; Vildhøj, H. W.; aj.: *String Processing and Information Retrieval: 17th International Symposium*, kapitola String Matching with Variable Length Gaps. Springer Berlin Heidelberg, 2010, s. 385–394.

[8] Hoffmann, C. M.; O'Donnell, M. J.: Pattern Matching in Trees. *J. ACM*, ročník 29, č. 1, 1982: s. 68–95.

[9] Crochemore, M.; Hancart, C.; Lecroq, T.: *Algorighms on Strings.* Cambridge University Press, 2007.

[10] Crochemore, M.; Vérin, R.: On Compact Directed Acyclic Word Graphs. In *Structures in Logic and Computer Science*, Springer-Verlag, 1997, s. 192–211.

Příloha A

Seznam použitých zkratek

SA Suffixový automat (suffix automaton)

CSA Kompaktní suffixový automat (compact suffix automaton)

SJT Podstromová přechodová tabulka (subtree jump table)

TPP Prefix stromového vzorku (tree pattern prefix)

XML Extensible Markup Language

Příloha **B**

Obsah přiloženého disku

- `readme.txt` ..... stručný popis obsahu disku
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `task.pdf` ..... zadání práce