FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Quick search vyhledávání ve stromech
Student:	Michal Cvach
Vedoucí:	Ing. Jan Trávníček
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte návrh a implementaci algoritmu protisměrného vyhledání ve stromech [1]. Nastudujte algoritmus Quick Search pro vyhledávání v řetězcích [2].
2) Navrhněte adaptaci Quick Search algoritmu pro linearizované stromy.
3) Implementujte tento algoritmus se zaměřením na efektivitu.
4) Zvolte vhodné množiny stromů a stromových vzorků pro experimenty. Proveďte měření rychlosti běhu implementovaného algoritmu a porovnejte ji s implementací algoritmu protisměrného vyhledávání ve stromech [1].

Seznam odborné literatury

[1] Trávníček, J., Janoušek, J., Melichar, B., Cleophas, L.: Linearised Backward Tree Pattern Matching. In: LATA 2015 Proceedings, LNCS, to appear, 2015.
[2] Sunday D. M., 1990, A very fast substring search algorithm, Communications of the ACM . 33(8):132-142.

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

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 29. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Quick search vyhledávání ve stromech

Michal Cvach

Katedra teoretické informatiky
Vedoucí práce: Ing. Jan Trávníček

13. května 2018

Poděkování

Děkuji Ing. Janu Trávníčkovi za jeho trpělivost a ochotu při vedení této práce. Dále bych rád poděkoval svému okolí, především své blízké rodině, za podporu během studia.

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 13. května 2018 ……

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Cvach, Michal. *Quick search vyhledávání ve stromech.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práce se zabývá problémem hledání stromových vzorů ve stromech. V práci je navržena a následně implementována adaptace algoritmu Quick Search pro hledání stromových vzorů ve stromech. Algoritmus je implementován v projektu Automatová knihovna, dále je též implementován v souboru nástrojů Forest fire & Fire wood, ve kterém byl extenzivně testován a následně porovnán s ostatními algoritmy pro řešení stejného problému, především s algoritmem protisměrného vyhledávání ve stromech. Prezentovaný algoritmus poskytuje o 30 % lepší výsledky než algoritmus protisměrného vyhledávání ve stromech.

Klíčová slova návrh algoritmu, algoritmus pro vyhledávání ve stromech, vyhledávání vzorů ve stromech, zpracovávání stromů, linearizace stromů

Abstract

This thesis deals with the problem of tree pattern matching. Adaptation of the Quick Search algorithm for tree pattern matching is designed and then implemented in the work. The algorithm is implemented in the Algorithm library project, and it is also implemented in the Forest fire & Fire wood toolkit, where it was extensively tested and then compared with other algoritms which can solve the same problem, mainly the backward linearised tree pattern matching algorithm. The presented algorithm achieves results which are by 30 % better than the results of the backward linearised tree pattern matching algorithm.

Keywords algorithm design, tree pattern matching algorithm, tree pattern matching, tree processing, tree linearisation

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Úvod

Stromy jsou v informatice fundamentální datovou strukturou s širokými možnostmi použití. Častým problémem je pak problém hledání podstromů ve větších stromech. V našem případě se budeme zabývat hledáním takzvaných vzorů. Vzory jsou stromy, které mohou ve svých vrcholech kromě znaků z abecedy navíc obsahovat ještě speciální zástupný symbol, na jehož místo může být při hledání výskytů dosazen libovolný podstrom. Tento problém je potřeba řešit například při interpretaci neprocedurálních programovacích jazyků, výběru kódu kompilátorem, nebo při zpracování XML (Extensible Markup Language).

Výsledkem práce je nový algoritmus pro řešení daného problému, který v určitých případech dosahuje lepších výsledků než existující algoritmy řešící stejný problém. Díky tomu bude výsledek práce prospěšný všude tam, kde je třeba problém hledání vzorů ve stromech (nebo nějakou jeho část) řešit.

Široké spektrum aplikací, kde je tento problém potřeba řešit pro mě byl i motivací, proč se právě tímto tématem zabývat. Výsledný algoritmus by totiž mohl přispět k nezanedbatelnému zrychlení všech aplikací, kde je potřeba tento problém řešit.

Cíle práce

Hlavním cílem této práce je návrh adaptace algoritmu Quick Search pro hledání podřetězců v řetězcích, která by byla schopná řešit problém hledání stromových vzorů ve stromech. Dále implementovat tento algoritmus (nebo jeho varianty) v rámci projektu *Algoritmová knihovna* a dále v souboru nástrojů *Forest fire & Fire wood*, a následně změřit rychlost této implementace a porovnat jí s ostatními existujícími algoritmy řešícími problém hledání stromových vzorů ve stromech, především s algoritmem protisměrného vyhledávání ve stromech.

Implementace v rámci projektu Algoritmová knihovna by měla být relativně pochopitelná a přehledná, neboť se jedná o projekt, který slouží mimo jiné i studentům, kteří si knihovnu mohou stáhnout a využít ji pro své expe-

rimenty. U implementace v rámci souboru nástrojů Forest fire & Fire wood je pak cílem algoritmus implementovat s důrazem na efektivitu, protože tento soubor nástrojů bude využit pro měření rychlosti algoritmu a následné porovnání s ostatními algoritmy.

V rámci teoretické části práce je pak tedy cílem nastudovat a následně vysvětlit teorii týkající se stromů, abeced (speciálně ohodnocených abeced) a též teorii linearizace stromů. Dále získat přehled o existujících algoritmech řešících stejný problém, především o již zmíněném algoritmu protisměrného vyhledávání ve stromech.

Struktura práce

V části 1 nejprve definujeme základní pojmy z teorie stromů a abeced, se kterými budeme dále v rámci práce pracovat. Dále stručně shrneme existující algoritmy řešící stejný problém, více detailně pak v této části popíšeme algoritmus protisměrného vyhledávání ve stromech, protože s tímto algoritmem budeme námi prezentovaný algoritmus především porovnávat v dalších částech práce. Dále v této části ukážeme principy linearizace stromů a představíme dvě notace, se kterými bude námi prezentovaný algoritmus pracovat. Nakonec představíme algoritmus Quick Search pro hledání podřetězců v řetězcích, ze kterého vychází algoritmus prezentovaný v této práci, a představíme i projekt Algoritmové knihovny, v rámci kterého byl prezentovaný algoritmus implementován.

V části 2 se pak zabýváme konkrétním návrhem našeho algoritmu. Ukážeme, jak zkonstruovat potřebné pomocné struktury, a jak funguje samotný algoritmus. Kromě pseudokódu algoritmu ukážeme běh algoritmu i na příkladech pro konkrétní vstupy. Prezentujeme několik variant algoritmu, tedy popíšeme i to, jak tyto jednotlivé varianty fungují a v čem se liší.

A nakonec v části 3 se budeme zabývat implementací a testováním prezentovaného algoritmu. Nejprve nastíníme některé konkrétní implementační detaily, budeme mluvit o rozdílech v implementaci v rámci projektu Algoritmová knihovna, v rámci kterého se indexuje od nuly, a v implementaci v rámci souboru nástrojů Forest fire & Fire wood, ve kterém se indexuje od jedničky. Dále shrneme všechny nově vzniklé algoritmy v rámci Algoritmové knihovny. Poté se budeme věnovat testování prezentovaného algoritmu. Popíšeme, jak byl implentovaný algoritmus zahrnut do testovacích skriptů v rámci projektu Algoritmová knihovna, a dále se budeme věnovat testování v rámci souboru nástrojů Forest fire & Fire wood. V tomto souboru nástrojů byla změřena rychlost námi prezentovaného algoritmu a dalších algoritmů řešících stejný problém, výsledky měření prezentujeme především formou grafů a vyvozujeme z nich nějaké závěry.

Kapitola 1

Teorie

1.1 Stromy

Než začneme mluvit o existujících algoritmech, řešících problém hledání vzorů ve stromech, speciálně pak o algoritmu protisměrného vyhledání ve stromech [1], se kterým budeme námi prezentovaný algoritmus porovnávat v praktické části práce, bude potřeba definovat samotný problém. K tomu bude ovšem nejprve potřeba zadefinovat několik důležitých pojmů a struktur.

Definice 1. Strom je acyklický orientovaný souvislý graf $t = (V, E)$ s tím, že implicitně pracujeme se zakořeněnými stromy, tedy existuje vrchol $r \in V$, který nazýváme kořen. Tento kořen r má vstupní stupeň 0, všechny ostatní vrcholy stromu t pak mají vstupní stupeň 1. Pro každý vrchol $v \in V$, kde $v \neq r$ existuje pouze jedna cesta z r do v. Vrcholy s výstupním stupněm 0 nazýváme *listy*.

Definice 2. Abeceda je konečnou neprázdnou množinou symbolů. Ohodnocenou abecedou nazýváme konečnou neprázdnou množinu symbolů, kde navíc každý symbol má přiřazenou jedinečnou aritu a, $a \in \mathbb{N}_0$.

Definice 3. Ohodnocený označený zakořeněný strom je pak stromem, pro který navíc platí, že každý vrchol $v \in V$ je označený symbolem c z ohodnocené abecedy $\mathcal{A}$. Dále platí, že pokud symbol c má aritu a, pak výstupní stupeň vrcholu v je též a. Z toho ihned vyplývá, že listy mohou být ohodnoceny pouze symboly s aritou 0. A konečně, pro ohodnocený označený uspořádaný zakořeněný strom ještě navíc platí, že přímí následníci (tedy synové) $x_{v1}, x_{v2}, ..., x_{vn}$ vrcholu v s aritou n jsou uspořádaní.

Definice 4. Stromovým vzorem nazýváme strom, ve kterém se navíc kdekoliv může vyskytovat speciální zástupný znak $*$, který zastupuje libovolný podstrom. Arita zástupného symbolu $*$ je 0.

1.2 Definice problému

Definice 5. Problém vyhledávání vzorů ve stromech je následující. Máme zadaný stromový vzor $p = (V_1, E_1)$ a dále strom $t = (V_2, E_2)$. Chceme najít veškeré výskyty vzoru p ve stromu t. Vzor p se vyskytuje ve vrcholu $v \in V_2$ stromu t pokud platí:

- Buď $p = *$
- Nebo pro symbol s_1 v kořenu vzoru p a symbol s_2 ve vrcholu v platí $s_1 = s_2$ a pro všechny podstromy vzoru p, které označíme $x_{p1}, x_{p2}, ..., x_{pn}$ a pro všechny podstromy vrcholu v, které označíme $y_{v1}, y_{v2}, ..., y_{vn}$ platí, že strom x_{pi} se vyskytuje ve stromu y_{vi} pro všechny $i \in n$, kde $n \in \mathbb{N}_0$ je arita kořene vzoru p.

Můžeme si povšimnout, že problém hledání podstromů ve stromech je vlastně speciálním případem hledání vzorů ve stromech. Podstrom je vzor, který neobsahuje žádný výskyt zástupného znaku $*$.

1.3 Existující algoritmy

Existuje velká řada algoritmů, které mohou být použity k řešení tohoto problému. Algoritmy poskytující nejlepší praktické výsledky se dají rozdělit do několika kategorií.

- Algoritmy založené na deterministických konečných stromových automatech založených na principu zezdola-nahoru (anglicky deterministic frontier-to-root tree automata, zkráceně DFRTAs) [2, 3, 4].
- Stringpath vyhledávače ve stylu Hoffmanna a O'Donella [5, 6]. Některé z těchto algoritmů využívají principu protisměrného vyhledávání.
- Algoritmy založené na linearizaci. Tyto algoritmy využívají toho, že stromy mohou být převedeny na řetězce a následně porovnávají tyto jejich řetězcové reprezentace. Principem linearizace stromů se budeme více zabývat v části 1.4. Typickým představitelem tohoto přístupu je algoritmus protisměrného vyhledávání ve stromech [1].

Do poslední zmíněné kategorie zapadá i v této práci prezentovaný algoritmus.

1.4 Princip linearizace stromů

Stromy mohou být převedeny na řetězce pomocí linearizace. Protože řetězec reprezentující podstrom stromu je podřetězcem řetězce reprezentující tento

strom (uvidíme dále), problém hledání podstromů ve stromech se dá vlastně linearizací zredukovat na problém hledání podřetězců v řetězcích, který umíme řešit relativně efektivně. Několik algoritmů pro tento problém dosahuje v praktickém čase lineární časové složitosti, i když časová složitost v nejhorším případě může být horší. Mezi tyto algoritmy patří například algoritmy Boyer-Moore-Horspool [7, 8], Knuth-Morris-Pratt [9], nebo Quick Search [10], na kterém je založený prezentovaný algoritmus.

Náš problém je o něco obecnější, protože musíme navíc ještě nějak řešit možné výskyty zástupného symbolu $*$, místo kterého se může v původním stromě vyskytovat libovolný podstrom. Budeme tedy muset do našeho algoritmu přidat nějaké podpůrné struktury, abychom s těmito podstromy mohli efektivně pracovat. Nejprve se ale pojďme podívat na to, jak bude vypadat ona řetězcová reprezentace stromu.

Existuje mnoho různých notací, které je možné při linearizaci stromů použít. My budeme používat dvě, a to konkrétně *prefixovou ohodnocenou notaci* a *prefixovou ohodnocenou zarážkovou notaci*.

Definice 6. Prefixová ohodnocená notace $pref\_ranked(t)$ je definována následovně:

1. $pref\_ranked(*) = *$
2. $pref\_ranked(v) = v0$, pokud v je list.
3. $pref\_ranked(t) =$
   $rn\ pref\_ranked(x_1)\ pref\_ranked(x_2)\ ...\ pref\_ranked(x_n)$, kde r je kořenem stromu t, n je aritou kořene r a $x_1, x_2, ..., x_n$ jsou přímí následníci kořene r.

Příklad 1. Mějme ohodnocenou abecedu $A = \{a2, a1, a0\}$. Dále mějme ohodnocený označený zakořeněný strom $t_1 = (\{a2_1, a0_2, a1_3, a0_4\}, R_1)$ nad abecedou $\mathcal{A}$, kde $R_1 = \{(a2_1, a0_2), (a2_1, a1_3), (a1_3, a0_4)\}$. Tento strom bychom v prefixové ohodnocené notaci zapsali jako $pref\_ranked(t_1) = a2\ a0\ a1\ a0$.

Příklad 2. Mějme ohodnocenou abecedu $A = \{a2, a1, a0\}$. Dále mějme stromový vzor $p_1 = (\{a2_1, *_2, a1_3, *_4\}, R_2)$ nad abecedou $\mathcal{A} \cup \{*\}$, kde $R_2 = \{(a2_1, *_2), (a2_1, a1_3), (a1_3, *_4)\}$. Tento strom bychom v prefixové ohodnocené notaci zapsali jako $pref\_ranked(p_1) = a2 * a1 *$.

Definice 7. Prefixová ohodnocená zařážková notace $pref\_ranked\_bar(t)$ je definována následovně:

1. $pref\_ranked\_bar(*) = * \uparrow *$.
2. $pref\_ranked\_bar(v) = v0 \uparrow 0$, pokud v je list.

3. $pref\_ranked\_bar(t) = rn\, pref\_ranked\_bar(x_1)\, pref\_ranked\_bar(x_2)$ ... $pref\_ranked\_bar(x_n) \uparrow n$, kde r je kořenem stromu t, n je aritou kořene r a $x_1, x_2, ..., x_n$ jsou přímí následníci kořene r.

Definice 8. Symbol $\uparrow n$, kde $n \geq 0$ nazveme zarážkou arity n. Množinu všech zarážek značíme symbolem $\uparrow_n$.

Příklad 3. Mějme ohodnocenou abecedu $A = \{a2, a1, a0\}$. Dále mějme ohodnocený označený zakořeněný strom $t_1 = (\{a2_1, a0_2, a1_3, a0_4\}, R_1)$ nad abecedou $\mathcal{A}$, kde $R_1 = \{(a2_1, a0_2), (a2_1, a1_3), (a1_3, a0_4)\}$. Tento strom bychom v prefixové ohodnocené zarážkové notaci zapsali jako
$pref\_ranked\_bar(t_1) = a2\ a0 \uparrow 0\ a1\ a0 \uparrow 0 \uparrow 1 \uparrow 2$.

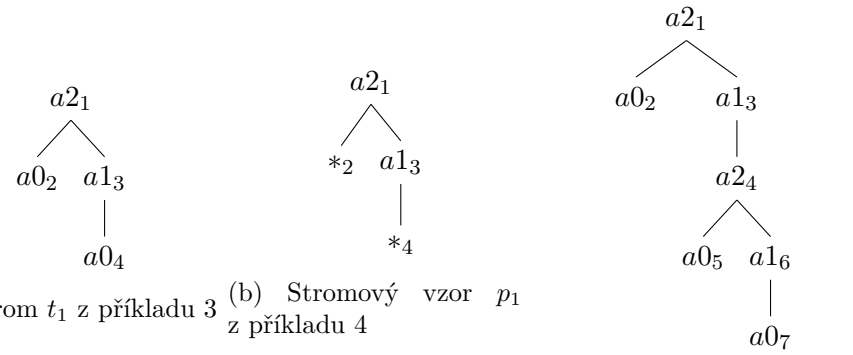
Příklad 4. Mějme ohodnocenou abecedu $A = \{a2, a1, a0\}$. Dále mějme stromový vzor $p_1 = (\{a2_1, *_2, a1_3, *_4\}, R_2)$ nad abecedou $\mathcal{A} \cup \{*\}$, kde $R_2 = \{(a2_1, *_2), (a2_1, a1_3), (a1_3, *_4)\}$. Tento strom bychom v prefixové ohodnocené zarážkové notaci zapsali jako $pref\_ranked\_bar(p_1) = a2 * \uparrow * a1 * \uparrow * \uparrow 1 \uparrow 2$.

Příklad 5. Mějme opět ohodnocenou abecedu $\mathcal{A} = \{a2, a1, a0\}$ a ohodnocený označený zakořeněný strom t_2 nad touto abecedou. Platí, že $t_2 = (\{a2_1, a0_2, a1_3, a2_4, a0_5, a1_6, a0_7\}, R_3)$ a $R_3 = \{(a2_1, a0_2), (a2_1, a1_3), (a1_3, a2_4), (a2_4, a0_5), (a2_4, a1_6), (a1_6, a0_7)\}$. Chceme najít veškeré výskyty t_1 z příkladu 3 a p_1 z příkladu 4 ve stromu t_2. Snadno nahlédneme, že t_1 se v t_2 vyskytuje pouze jednou, a sice ve vrcholu $a2_4$, zatímco p_1 se v t_2 vyskytuje hned dvakrát, a sice ve vrcholech $a2_1$ a $a2_4$.

Pokud bychom linearizovali strom t_2, dostaneme následující řetězcovou reprezentaci $pref\_ranked\_bar(t_2) = a2\ a0 \uparrow 0\ a1\ a2\ a0 \uparrow 0\ a1\ a0 \uparrow 0 \uparrow 1 \uparrow 2 \uparrow 1 \uparrow 2$. Pokud se znovu podíváme na linearizaci t_1 z příkladu 3, můžeme si všimnout, že daný řetězec je podřetězcem linearizovaného stromu t_2. Bohužel pro vzor p_1 z příkladu 4 už toho neplatí, protože znak $*$ (a stejně tak $\uparrow *$) se nikdy nebude vyskytovat v prohledávaném stromě. Při hledání výskytu vzorů ve stromech je proto nutné nějak speciálně pracovat se zástupným znakem $*$. V našem případě k tomu poslouží speciální struktura SJT, ke které se dostaneme v části 1.6.

Obrázek 1.1 demonstruje strom t_2 z příkladu 5 a dále strom t_1 z příkladu 3 a stromový vzor p_1 z příkladu 4.

Pro účely této práce budeme znaky v řetězcové reprezentaci indexovat vždy od nuly. Tedy pokud bude mít řetězcová reprezentace stromu t délku n, pak se tato reprezentace skládá ze znaků $t_0, t_1, ..., t_{n-1}$.



(a) Strom t_1 z příkladu 3 (b) Stromový vzor p_1 z příkladu 4

(c) Strom t_2 z příkladu 5

Obrázek 1.1: Stromy t_1 z příkladu 3 a t_2 z příkladu 5 a dále stromový vzor p_1 z příkladu 4

1.5 Algoritmus Quick Search pro hledání podřetězců v řetězcích

Protože náš algoritmus bude v určitých ohledech vycházet z algoritmu Quick Search, je vhodné tento algoritmus stručně představit. V celé této části vycházíme z [10].

Nejprve si pojďme formálně zavést problém hledání podřetězců v řetězcích.

Definice 9. Problém hledání podřetězců v řetězcích je následující. Je dán řetězec p délky m a delší řetězec t délky n, tedy platí $n > m > 0$. Chceme zjistit, zda se řetězec p vyskytuje někde v řetězci t jako podřetězec. Znak na pozici i v řetězci p budeme značit $p[i]$ a budeme indexovat od nuly, tedy řetězec $p = p[0]...p[m-1]$ má délku m, znaky na pozici j v řetězci t budeme značit $t[j]$, tedy řetězec $t = t[0]...t[n-1]$ má délku n. Řetězec p se vyskytuje v řetězci t jako jeho podřetězec na pozici k, jestliže platí $t[k+i] = p[i]$ pro všechna $0 \leq i < m$.

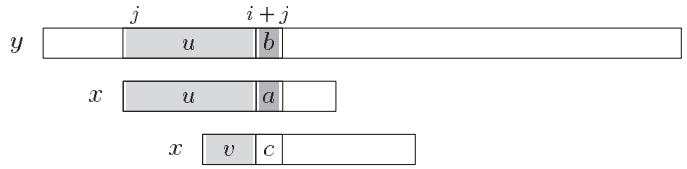
Snadno si dovedeme představit naivní algoritmus řešící tento problém. Na počátku zarovnáme hledaný podřetězec na nejlevější pozici prohledávaného řetězce, poté začneme porovnávat znaky v hledaném podřetězci se znaky v delším řetězci zleva, tedy v pořadí $p[0], p[1], ..., p[m-1]$. Pokud se všechny shodují, nalezli jsme výskyt podřetězce, pokud se některý liší, posuneme hledaný podřetězec o znak doprava a opět porovnáme všechny znaky hledaného podřetězce s korespondujícími znaky prohledávaného řetězce zleva doprava.

Tento algoritmus je velmi přímočarý, snadno se implementuje, a pro krátké řetězce je i velmi dobře použitelný. Časová složitost tohoto algoritmu v nejhorším případě je sice $O(mn)$, tedy se jedná o kvadratický algoritmus, při použití algoritmu v praxi s reálnými jazyky (například s angličtinou) se ale velmi často

liší hned první znak podřetězce od znaku na aktuální pozici prohledávaného řetězce, a tedy můžeme očekávat, že algoritmus poběží v průměru v lineárním čase $O(n)$ [10].

Protože problém hledání podřetězců v řetězcích je už od počátku informatiky jedním z fundamentálních problémů, v průběhu let bylo vyvinuto mnoho algoritmů poskytujících znatelně lepší praktické výsledky než zmíněný naivní algoritmus. Za zmínku stojí především algoritmus Knuth-Moriss-Pratt [9] a z něj vycházející algoritmus Boyer-Moore [7]. Hlavní myšlenkou algoritmu Knuth-Moriss-Pratt je možnost posunout pozici hledaného podřetězce v delším řetězci o více než jeden znak při nalezení neshody při porovnávání znaků. Algoritmus Knuth-Morris-Pratt prochází delší řetězec i hledaný podřetězec zleva, ale pokud narazí na neshodu, například znak $p[i]$ se neshoduje se znakem $t[k+i]$, pak je hledaný podřetězec posunut doprava tak, aby bylo možné zarovnat již oskenovaný text s nejbližší shodující se předponou hledaného podřetězce. Tyto posuny na základě předpony hledaného podřetězce se dají předpočítat v čase $O(m)$, a jsou určeny pouze hledaným podřetězcem bez ohledu na delší prohledávaný řetězec.

Poté, co je hledaný podřetězec posunut o nějaké $\delta \geq 1$, skenování pokračuje opět na pozici $t[k+i]$, ale znak na ní je nyní porovnáván s novým znakem z hledaného podřetězce $p[i-\delta]$, pokud $i - \delta \geq 0$. Pokud $i - \delta < 0$, pak je hledaný podřetězec posunut na pozici $k+i+1$ v delším řetězci t. Protože se v delším řetězci t algoritmus nikdy nevrací, jeho časová složitost v nejhorším případě je $O(n)$, tedy lineární.

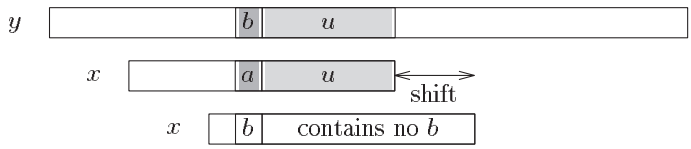


Obrázek 1.2: Posun algoritmu Knuth-Morris-Pratt – y je zde prohledávaný řetězec, x hledaný podřetězec a pro znaky a, b a c platí, že jsou všechny tři rozdílné [11]

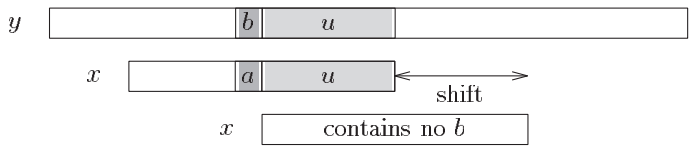
Při použití s reálnými jazyky (například s angličtinou) jsou ovšem naivní algoritmus a algoritmus Knuth-Moriss-Pratt zhruba stejně výkonné, protože velmi často dochází k tomu, že je neshoda odhalena hned prvním znakem z hledaného podřetězce p, kvůli čemuž se výhoda možných větších posunů příliš neprojeví [10].

Algoritmus Boyer-Moore též využívá možnosti posouvat pozici hledaného podřetězce o více než jeden znak při nalezení neshody. Tento algoritmus ale mění směr skenování hledaného podřetězce. Ten je při algoritmu Boyer-Moore skenován zprava doleva, tedy v pořadí znaků $p[m-1], ..., p[1], p[0]$. Stejně jako

v případě algoritmu Knuth-Moriss-Pratt, když se při porovnávání hledaného podřetězce s delším řetězcem narazí na neshodu, využijeme již oskenované znaky k tomu, abychom mohli posunout pozici hledaného podřetězce co nejvíce doprava. Posuny v algoritu Boyer-Moore jsou většinou větší než u algoritmu Khuth-Moriss-Pratt. Pokud se liší znak v prohledávaném textu na pozici $t[k+m-i]$ se znakem v hledaném podřetězci na odpovídající pozici, algoritmus Boyer-Moore nalezne v předpočítané tabulce index nejlevějšího výskytu daného znaku od konce prohledávaného podřetězce. Pokud se tento výskyt nachází vlevo od pozice aktuální neshody, pak můžeme vzdálenost onoho nejlevějšího výskytu a pozici našeho aktuálního lišícího se znaku v hledaném podřetězci označit δ_1. Protože velmi často dojde k neshodě ihned na pozici $p[m-1]$, δ_1 bude většinou pozitivní, a pro krátké hledané podřetězce se bude velmi často blížit $(m-1)$. Použitím tohoto δ_1 pro posuny dostaneme algoritmus, který je při použití obvykle více než trojnásobně tak rychlý, jako naivní algoritmus, případně algoritmus Knuth-Moriss-Pratt [10].



(a) Posun, pokud se b vyskytuje v x nalevo od a



(b) Posun, pokud se b nevyskytuje v x

Obrázek 1.3: Posuny algoritmu Boyer-Moore na základě δ_1 – y je zde prohledávaný řetězec, x hledaný podřetězec [11]

Protože δ_1 může být i negativní, je časová složitost výsledného algoritmu v nejhorším případě $O(mn)$, ale pokud si kromě δ_1 předpočítáme ještě δ_2 pomocí myšlenky z algoritmu Knuth-Moriss-Pratt a pro výsledný posun pak budeme používat maximum z δ_1 a δ_2, dostaneme algoritmus s časovou složitostí $O(m+n)$ v nejhorším případě. Hodnoty δ_2 jsou počítány tak, že se vezme již zkontrolovaná část hledaného podřetězce, tedy vše vpravo od prvního neshodujícího se znaku, a následně je nalezen nejbližší výskyt tohoto nového podřetězce v hledaném podřetězci vlevo od neshodujícího se znaku. Znak, který se bude následně nacházet vlevo od tohoto výskytu musí být na-

víc pochopitelně jiný, než aktuální neshodující se znak. Jak δ_1 tak δ_2 dokážeme předpočítat v lineárním čase $O(m)$.

Knuth-Moriss-Pratt i Boyer-Moore využívají znaky z aktuálního porovnávaného rozsahu pro výpočet posunu hledaného podřetězce pro další krok. Algoritmus Quick Search na to jde trochu jinak. Snadno si můžeme povšimnout, že znak ihned za koncem aktuálního porovnávaného rozsahu, tedy na pozici $t[k+m]$, bude určitě muset být nějakým způsobem zahrnut do procesu porovnávání v dalším kroku. Díky tomu můžeme tento znak použít pro výpočet nového Δ_1, které budeme počítat pro každý znak c z abecedy jako vzdálenost jeho nejlevějšího výskytu v hledaném podřetězci od jeho konce (hodnota Δ_1 pro nejpravější znak hledaného podřetězce bude 1, hodnota Δ_1 pro nejlevější znak bude m, za předpokladu že se daný znak nebude v hledaném podřetězci vyskytovat vícekrát, a konečně hodnota Δ_1 pro znaky, které se v hledaném podřetězci vůbec nevyskytují, bude $m+1$).

Algoritmus Quick Search tedy v případě nalezení neshody ve znaku hledaného podřetězce se znakem prohledávaného řetězce na odpovídající pozici posune hledaný podřetězec doprava na základě hodnoty Δ_1 pro znak za koncem aktuálního porovnávaného rozsahu. To buď zarovná nejpravější výskyt tohoto znaku v hledaném podřetězci s jeho výskytem v prohledávaném textu na pozici $t[k+m]$, a nebo, v případě, že se daný znak v hledaném podřetězci vůbec nevyskytuje, posune hledaný podřetězec rovnou na pozici za tento znak, tedy na pozici $t[k+m+1]$.

Použití této Δ_1 má hned několik výhod. Jednou z nich je fakt, že $\Delta_1 \geq 1$ vždy, což například pro δ_1 algoritmu Boyer-Moore neplatí. Velkou výhodou je i fakt, že o posunu rozhodujeme na základě znaku, který se nachází mimo aktuální porovnávací rozsah. To nám umožňuje skenovat prohledávaný podřetězec v libovolném směru. Můžeme ho prohledávat zepředu, odzadu, nebo například v pořadí určeném relativní četností výskytu jednotlivých znaků hledaného podřetězce v daném jazyce [10].

Algoritmus Quick Search prochází prohledávaný text zleva doprava, a ve stejném směru provádí i skenování hledaného podřetězce. Stejného Δ_1 pro výpočet posunu hledaného podřetězce, pouze s jiným pořadím skenování samotného hledaného podřetězce využívají algoritmy Maximal Shift a Optimal Mismatch, těmi se ale nebudeme dále zabývat, neboť se dále budeme inspirovat právě algoritmem Quick Search.

Algoritmus 1 ukazuje, jak je možné zkonstruovat tabulku posunů (BCS) založenou na Δ_1 pro algoritmus Quick Search pro hledání podřetězců v řetězcích. Algoritmus 2 pak ukazuje samotný pseudokód algoritmu Quick Search pro hledání podřetězců v řetězcích.

Zde je ještě vhodné zmínit, že řádka 11 v algoritmu 2 řeší situaci, kdy je hledaný podřetězec již zarovnán s m nejpravějšími znaky v prohledávaném řetězci t, a tedy bychom se na řádce 12 dotazovali na pozici n. Této podmínky se dá zbavit například tak, že na konec prohledávaného řetězce přidáme libovolný znak, ale nezvýšíme délku řetězce, tedy bude existovat znak $t[n]$, na

Název algoritmu: Konstrukce BCS pro algoritmus Quick Search nad řetězci.
Vstup: Hledaný podřetězec p délky m nad abecedou $\mathcal{A}$ prohledávaného řetězce t.
Výstup: Tabulka posunů $BCS(p)$.

```
1  begin
2  |  foreach x ∈ A do BCS[x] = m;
3  |  for i := 0 to m − 1 do
4  |  |  BCS[p[i]] := m − i
5  |  end
6  end
```

Algoritmus 1: Konstrukce tabulky posunů pro algoritmus Quick Search pro hledání podřetězců v řetězcích

Název algoritmu: Algoritmus Quick Search nad řetězci.
Vstup: Prohledávaný řetězec t délky n, hledaný podřetězec p délky m a tabulka posunů $BCS(p)$.
Výstup: Všechny výskyty hledaného podřetězce p v prohledávaném řetězci t.

```
1  begin
2  |  i := 0
3  |  while i + m ≤ n do
4  |  |  found := true
5  |  |  for j := 0 to m − 1 do
6  |  |  |  if p[j] ≠ t[i + j] then
7  |  |  |  |  found := false
8  |  |  |  |  break
9  |  |  end
10 |  |  if found then output(i);
11 |  |  if i + m = n then break;
12 |  |  i := i + BCS[t[i + m]]
13 |  end
14 end
```

Algoritmus 2: Algoritmus Quick Search pro hledání podřetězců v řetězcích

jehož základě učiníme nějaký posun, ale podmínka $i + m \leq n$ v dalším kroku while cyklu už se vyhodnotí jako nepravda a další iterace tím pádem neproběhne. Toto řešení s přidaným znakem bude mírně efektivnější, avšak varianta s podmínkou navíc je podle nás pochopitelnější.

1.6 Algoritmus protisměrného vyhledávání ve stromech

Algoritmem, který je v mnoha ohledech podobný algoritmu, který bude prezentován dále, je algoritmus protisměrného vyhledávání ve stromech [1], který staví na principech algoritmu Boyer-Moore z části 1.5 a využívá linearizaci jak prohledávaného stromu, tak hledaného stromového vzoru. Existuje několik různých variant algoritmu, které se liší ve směru procházení prohledávaného stromu a dále v použité notaci. My si nyní představíme variantu algoritmu, která využívá prefixovou ohodnocenou zarážkovou notaci tak, jak byla definována v části 1.4, a prohledávaný strom (respektive jeho řetězcovou reprezentaci) prochází zleva doprava, přičemž při kontrole výskytu skenuje řetězcovou reprezentaci hledaného vzoru zprava doleva. V této části budeme vycházet z [1].

Algoritmus využívá speciální tabulku posunů (BCS, Bad Character Shift Table), což je rozšířením δ_1 z algoritmu Boyer-Moore. Bohužel, tato struktura sama o sobě nám nestačí, neboť se v hledaném stromovém vzoru mohou vyskytovat zástupné symboly $*$, které mohou být při hledání výskytů nahrazeny libovolným podstromem. Výskyty tohoto zástupného symbolu musí být řešeny speciálním způsobem, algoritmus protisměrného vyhledávání ve stromech využívá druhou pomocnou strukturu, kterou nazýváme podstromová skoková tabulka (SJT, Subtree Jump Table).

Pojďme si nejprve říci, jak vygenerovat první ze zmiňovaných struktur, tedy tabulku posunů. Ta obsahuje pro každý znak z abecedy hodnotu posunu reprezentace hledaného vzoru doprava v reprezentaci prohledávaného stromu. Algoritmus protisměrného vyhledávání ve stromech se snaží o to, aby posuny byly co největší, ale zároveň algoritmus nesmí přeskočit žádný výskyt hledaného vzoru. Jinými slovy se algoritmus snaží otestovat existenci výskytu vzoru na co nejmenším počtu indexů, ale tak, aby byly skutečně nalezeny všechny výskyty.

Algoritmus protisměrného vyhledávání ve stromech se o posunu rozhoduje vždy na základě znaku na aktuální pozici $i+m-1$, tedy na základě nejpravějšího znaku v rámci aktuálního porovnávaného rozsahu. Podle znaku na této pozici se algoritmus v každé iteraci snaží učinit co největší bezpečný posun (tedy takový, který nepřeskočí žádný výskyt vzoru v prohledávaném stromu) řetězcové reprezentace vzoru doprava. Hodnotu tohoto největšího možného bezpečného posunu pro všechny znaky, které by se potenciálně mohli někdy v průběhu algoritmu vyskytovat na aktuální pozici $i + m - 1$ tedy určuje právě tabulka posunů. Poznamenejme, že nepotřebujeme hodnotu posunu pro zástupný znak $*$, neboť ten se v prohledávaném stromu nikdy nebude vyskytovat. Pro každý znak a z abecedy $\mathcal{A}$ vypočteme hodnotu v tabulce posunů $bcs[a]$ jako minimum ze tří čísel:

1. m
2. $j : p[m-j-1] = a$ a platí $m > j > 0$
3. $j + arita(a) * 2 : p[m-j-1] = *$ a platí $m > j > 0$ pro $a \notin \uparrow_n$ nebo $j-1 : p[m-j-1] = *$ a platí $m > j > 1$ pro $a \in \uparrow_n$

Délku řetězcové reprezentace hledaného stromového vzoru označujeme m, znak na pozici $0 \leq i < m$ pak označujeme $p[i]$. Připomeňme, že $\uparrow_n$ označuje množinu všech zarážek všech arit.

První bod zajišťuje, že posun nebude nikdy větší než délka řetězcové reprezentace hledaného vzoru. Druhý bod v podstatě odpovídá výpočtu δ_1 algoritmu Boyer-Moore z části 1.5. V tomto bodě tedy chceme nalézt vzdálenost nejpravějšího výskytu daného znaku od konce reprezentace hledaného vzoru (pro pozici $p[m-1]$ je tato vzdálenost 0). Poznamenejme, že v obou bodech počítáme vždy s tím, že za reprezentaci zástupného symbolu $* \uparrow *$ bude dosazen nejmenší možný podstrom, tedy například $a0 \uparrow 0$.

Třetí bod je pak trochu zajímavější, protože řeší situaci, kdy se daný znak bude nacházet někde v podstromu, který bude dosazen na místo zástupného symbolu $*$. Vzdálenost nejpravějšího výskytu zástupného znaku $*$ od konce řetězce je využita jako výchozí velikost posunu, pro konkrétní (nezarážkové) znaky pak ovšem může být ještě navýšena na základě jejich arity. Předpokládejme, že chceme vypočítat skutečnou možnou velikost posunu pro znak a určité arity. Pak nejmenší možný podstrom, který obsahuje daný znak a je podstrom, jehož kořenem je právě náš znak a, a všichni jeho přímí následníci jsou znaky s aritou 0, tedy například $b0$. Za každý znak $b0$ musí ovšem řetězcová reprezentace nejmenšího možného podstromu obsahovat ještě zarážku $\uparrow 0$, proto je výchozí posun možné zvýšit o dvojnásobek arity znaku a. Kterákoliv zarážka, tedy znak z množiny $\uparrow_n$, se může vyskytovat na konci podstromu dosazeného na místo zástupného znaku $*$, tedy tento znak v prohledávaném stromě může korespondovat s nejpravějším výskytem znaku $\uparrow *$. Proto je pro všechny znaky z množiny $\uparrow_n$ potřeba výchozí posun snížit o jedna. Dodatečná podmínka $m > j > 1$ při výpočtu posunu pro zarážkové symboly nám zajistí, že minimální posun nebude 0 v případě, že by se znak $\uparrow *$ vyskytoval v reprezentaci hledaného vzoru na nejpravější pozici. Rozmysleme si, že toto by mohlo nastat pouze pro vzor $* \uparrow *$.

Samotnou konstrukci tvorby tabulky posunů pak popisuje algoritmus 3.

Řádky 2 až 5 jsou zodpovědné za nalezení nejpravějšího výskytu zástupného znaku $*$ v našem vzoru, respektive jeho vzdálenosti od konce vzoru. Pokud se ve vzoru nevyskytuje žádný zástupný znak $*$, pak je tato vzdálenost položena m, tedy délce řetězcové reprezentace vzoru, což nijak nevadí. Na řádce 6 se pak inicializuje tabulka posunů hodnotou m, tedy délkou řetězcové reprezentace, pro všechny znaky z abecedy $\mathcal{A}$. Řádky 7 až 12 odpovídají třetímu popisovanému bodu a tedy slouží k určení velikosti posunu v případě,

Název algoritmu: Konstrukce BCS pro algoritmus zpětného vyhledávání ve stromech.
Vstup: Stromový vzor *pattern* v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(pattern)$ o velikosti m nad abecedou $\mathcal{A}$ prohledávaného stromu *subject*.
Výstup: Tabulka posunů $BCS(pref\_ranked\_bar(pattern))$.

```
1  begin
2  |  s := m
3  |  for i := 0 to m − 1 do
4  |  |  if pref_ranked_bar(pattern)[i] = ∗ then s = m − i − 1;
5  |  end
6  |  foreach x ∈ A do BCS[x] = m;
7  |  foreach x ∈ A do
8  |  |  if x ∉ A↑ then shift := s + Arity(x) ∗ 2;
9  |  |  else if s ≥ 2 then shift := s − 1;
10 |  |  else shift := s;
11 |  |  if BCS[x] > shift then BCS[x] := shift;
12 |  end
13 |  for i := 0 to m − 2 do
14 |  |  if pref_ranked_bar(pattern)[i] ∉ {∗, ↑∗} and
   |  |    BCS[pref_ranked_bar(pattern)[i]] > (m − i − 1) then
   |  |    BCS[pref_ranked_bar(pattern)[i]] := m − i − 1 ;
15 |  end
16 end
```

Algoritmus 3: Konstrukce tabulky posunů pro algoritmus zpětného vyhledávání ve stromech

že by se daný znak z abecedy vyskytoval v podstromu, jehož řetězcová reprezentace bude dosazená na místo zástupného řetězce $* \uparrow *$. A konečně, řádky 13 až 15 řeší situaci, kdy se znak nachází přímo v řetězcové reprezentaci našeho vzoru.

Kromě tabulky posunů budeme potřebovat i druhou zmiňovanou strukturu, podstromovou skokovou tabulku (SJT). Ta, jak její název napovídá, slouží pro přeskakování podstromů, které budou dosazeny na místo zástupného symbolu $*$ v našem hledaném vzoru. Zatímco tabulka posunů se konstruuje z řetězcové reprezentace hledaného vzoru, podstromová skoková tabulka se konstruuje z řetězcové reprezentace prohledávaného stromu. Pro každou pozici řetězcové reprezentace prohledávaného stromu obsahuje podstromová skoková tabulka jedno číslo. Význam tohoto čísla se liší podle toho, zda se jedná o zarážkový nebo nezarážkový znak. Pro nezarážkové znaky, tedy pro $a \notin \uparrow_n$, určuje číslo v podstromové skokové tabulce index o jeden znak doprava od konce podřetězce reprezentujícího podstrom začínající tímto znakem a. Pro zarážkové znaky, tedy $a \in \uparrow_n$, určuje číslo v podstromové skokové tabulce

prozměnu index o jeden znak doleva od začátku podřetězce reprezentujícího podstrom končící tímto znakem a.

Podstromová skoková tabulka se dá zkonstruovat již v průběhu procesu linearizace stromu, nebo se dá zkonstruovat dodatečně algoritmem 4.

Název algoritmu: Konstrukce SJT pro algoritmus zpětného vyhledávání ve stromech.
Vstup: Prohledávaný strom t v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(t)$ o délce n, index aktuálního vrcholu $root$, což je implicitně 0 (index kořene) a reference na prázdnou podstromovou skokovou abulku $SJT(pref\_ranked\_bar(t))$ délky n
Výstup: index $koncovyIndex$, podstromová skoková tabulka $SJT(pref\_ranked\_bar(t))$

```
1  begin
2  |  index := root + 1
3  |  for i = 1 to Arity(pref_ranked_bar(t)[root]) do
4  |  |  index := ConstructSJT(pref_ranked_bar(t), index,
   |  |    SJT(pref_ranked_bar(t)))
5  |  end
6  |  index := index + 1
7  |  SJT(pref_ranked_bar(t))[root] = index
8  |  SJT(pref_ranked_bar(t))[index − 1] = root − 1
9  |  return index
10 end
```

Algoritmus 4: Konstrukce podstromové skokové tabulky pro algoritmus zpětného vyhledávání ve stromech

Všimněme si, že algoritmus 4 funguje rekurzivně. Při zpracovávání vrcholu x se rekurzivně zavolá na všechny jeho následníky, hodnotu pro daný vrchol x pak určí z návratové hodnoty rekurzivního volání algoritmu pro nejpravějšího následníka vrcholu x.

Tabulka 1.1: Podstromová skoková tabulka $SJT(pref\_ranked\_bar(t))$ pro strom z příkladu 6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$
14	7	4	1	6	3	0	13	10	7	12	9	6	-1

Příklad 6. Mějme si následující strom t, respektive jeho řetězcovou reprezentaci v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(t) = a2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$ nad abecedou $\mathcal{A} = \{a3, a2, a1,$

$a0, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0\}$. Tabulka 1.1 ukazuje, jak vypadá hotová podstromová skoková tabulka $SJT(pref\_ranked\_bar(t))$.

V tuto chvíli už víme, jak zkonstruovat obě struktury potřebné pro algoritmus protisměrného vyhledávání ve stromech. Můžeme tedy představit pseudokód algoritmu.

Název algoritmu: Algoritmus protisměrného vyhledávání ve stromech.
Vstup: Řetězcová reprezentace stromu *subject* v $pref\_ranked\_bar(subject)$ notaci délky n, řetězcová reprezentace stromového vzoru *pattern* v $pref\_ranked\_bar(pattern)$ notaci délky m, $SJT(pref\_ranked\_bar(subject))$ a $BCS(pref\_ranked\_bar(pattern))$.
Výstup: Všechny výskyty stromového vzoru *pattern* ve stromu *subjekc*.

```
1  begin
2      i := 0
3      while i ≤ (n − m) do
4          j := m − 1
5          position := i + j
6          while j ≥ 0 and position ≥ 0 do
7              if pref_ranked_bar(subject)[position] =
                pref_ranked_bar(pattern)[j] then
8                  position := position − 1
9              else if pref_ranked_bar(pattern)[j] = ↑∗ and
                pref_ranked_bar(subject)[position] ∈ A↑ then
10                 position := SJT(pref_ranked_bar(subject))[position]
11                 j = j − 1 {Subtree skip}
12             else break;
13             j := j − 1
14         end
15         if j = −1 then output(position + 1);
16         i := i + BCS[pref_ranked_bar(subject)[i + m − 1]]
17     end
18 end
```

Algoritmus 5: Algoritmus protisměrného vyhledávání ve stromech

Algoritmus 5 v každé iteraci while cyklu na řádcích 3 až 17 zkontroluje, zda existuje výskyt vzoru, končící na pozici $i+m-1$. Pokud ano, je tento výskyt na řádce 15 přidán do výstupu. Poté se na řádce 16 učiní posun na základě tabulky posunů (BCS), a to podle symbolu na pozici $i+m-1$. Řetězcová reprezentace hledaného vzoru je v řetězcové reprezentaci prohledávaného stromu posunuta

doprava, a pokud po tomto posunu nekončí reprezentace vzoru až za koncem reprezentace prohledávaného stromu, algoritmus pokračuje další iterací.

Podstromová skoková tabulka (SJT) je použita na řádce 10. K jejímu použití dochází, když při kontrole výskytu algoritmus v řetězcové reprezentaci vzoru narazí na zástupný zarážkový znak $\uparrow *$. V takovém případě se podíváme, zda na odpovídající pozici v reprezentaci prohledávaného stromu končí nějaký podstrom (tedy zda na dané pozici je nějaký zarážkový symbol), pokud ano, pak na místo zástupného podřetězce $* \uparrow *$ v podstatě dosadíme celý podstrom končící daným zarážkovým znakem. Abychom nemuseli dlouze zjišťovat, kde v reprezentaci prohledávaného stromu začíná podstrom, když známe pouze pozici jeho konce, máme tyto informace již předpočítané v podstromové skokové tabulce, ve které rovnou najdeme index, na který musíme skočit, abychom v následujícím kroku mohli pokračovat s porovnáváním dalších znaků v reprezentaci našeho vzoru.

1.7 Algoritmová knihovna

Algoritmová knihovna je projekt, s jehož myšlenkou přišel *Jan Trávníček*. Knihovna vznikla (tehdy ještě pod názvem Automatová knihovna) jako sada aplikací pro manipulaci a práci s automaty, gramatikami a regulárními výrazy, později ovšem do projektu přibyla i rozšíření pro práci s grafy, řetězci, nebo například stromy. Dnes je tedy Algoritmová knihovna sadou mnoha aplikací, které vycházejí z myšlenek unixové filozofie v tom smyslu, že každá z nich má jasně daný vstup a výstup a provádí dobře jeden úkon. Pokud uživatel potřebuje úkon složitější, několik programů lze snadno spojit pomocí rour. Automatová knihovna je napsána v jazyce C++ a je open source, tedy si ji kdokoliv může stáhnout a použít pro své experimentování. Dále pak knihovna slouží jako podpůrný výukový prostředek v některých předmětech, například BI-AAG (Automaty a gramatiky).

Základní kámen projektu položil ve své bakalářské práci *Martin Žák* [12], na jeho práci pak navázalo mnoho dalších prací. Pro tuto práci je z nich nejzajímavější práce *Štěpána Plachého* [13], který se zabýval stromovými automaty a algoritmy nad stromy, mimo jiné implementoval i generování náhodných stromů, které v této práci používáme pro testování našich algoritmů. Stromy a algoritmy nad nimi se ve své bakalářské práci věnoval ještě například *Aleksandr Shatrovskii* [14].

Kapitola 2

Návrh

2.1 Návaznost na řetězcový Quick Search

V části 1.5 jsme popsali algoritmus Quick Search pro hledání podřetězců v řetězcích. Dále jsme popsali i algoritmus Boyer-Moore pro stejný problém. V části 1.6 jsme pak popsali algoritmus protisměrného vyhledávání ve stromech, který částečně vychází z algoritmu Boyer-Moore a řeší problém hledání vzorů ve stromech.

V této části se zaměříme na to, jak využít myšlenky algoritmu Quick Search pro nový algoritmus pro hledání vzorů ve stromech. Nejprve připomeneme několik faktů. Řetězcová varianta algoritmu Quick Search prohledává dlouhý řetězec intuitivně zleva doprava, stejně tak hledaný podřetězec je skenován zleva doprava, dokud nenarazíme na neshodu, nebo nedojdeme na konec podřetězce. V případě, že narazíme na neshodu, podíváme se na znak, který je v prohledávaném textu na indexu ihned za aktuální pozicí posledního znaku hledaného podřetězce, a na základě něj posuneme hledaný podřetězec doprava o minimálně jeden znak.

Příklad 7. Předpokládejme, že hledáme řetězec $p = abc$ v delším řetězci $t = cdefabcd$. Na začátku máme hledaný podřetězec p zarovnán vlevo na pozici 0 v řetězci t (indexujeme od nuly, tak jak bylo zavedeno v části 1.5). Porovnáme tedy znak na pozici $p[0]$ se znakem na pozici $t[0]$. Protože $a \neq c$, podíváme se na znak na pozici $t[0+m]$ kde m je délka hledaného podřetězce, tedy v našem případě 3, a na jeho základě učiníme posun. Na pozici $t[3]$ nalezneme znak f, a protože se ten nikde v hledaném podřetězci p nevyskytuje, posuneme pozici hledaného podřetězce v delším textu doprava o $m + 1$ znaků, tedy na pozici $t[4]$. V dalším kroku bychom tedy rovnou nalezli výskyt hledaného podřetězce na pozici $t[4]$.

Jak vidíme z předchozího příkladu, když rozhodujeme o posunu pozice hledaného podřetězce v delším řetězci, činíme tak na základě znaku, který je

mimo aktuální porovnávaný rozsah. Například v uvedeném příkladě je na začátku porovnávaný rozsah $t[0], ..., t[2]$, ale o posunu se rozhodneme na základě znaku na pozici $t[3]$. Tedy znaku, který žádným způsobem nefiguruje při kontrole výskytu na aktuální pozici. Toto bychom rádi zachovali i v naší adaptaci algoritmu Quick Search pro problém hledání vzorů ve stromech. Bohužel se ale ukáže, že při procházení řetězce zleva doprava a kontrole výskytů též zleva doprava narazíme na problém. Tento problém nám způsobí náš zástupný znak $*$, který může být při kontrole výskytu nahrazen libovolným podstromem.

(a) Stromový vzor p z příkladu 8

(b) Výskyt vzoru p ve stromu t uvedený v příkladu 8

Obrázek 2.1: Znázornění vzoru p a jeho výskytu ve stromě t z příkladu 8

Příklad 8. Představme si linearizovaný stromový vzor (v prefixové ohodnocené zarážkové notaci z části 1.4) $p = a1 * \uparrow* \uparrow1$ a dále linearizovaný strom (ve stejné notaci) $t = ... \ a1 \ a2 \ a0 \uparrow0 \ a0 \uparrow0 \uparrow2 \uparrow1 \ ...$. Předpokládejme, že bychom chtěli najít všechny výskyty vzoru p ve stromu t. Pak ale celý řetězec $a1 \ a2 \ a0 \uparrow0 \ a0 \uparrow0 \uparrow2 \uparrow1$ je vlastně jeden výskyt vzoru p. Když pak budeme chtít posunout vzor doprava, abychom se mohli pokusit najít další výskyt, budeme se rozhodovat podle znaku $a0$ která se bude vyskytovat ve stromu t na pozici $t[i+m]$ kde m je délka reprezentace našeho vzoru, tedy v tomto případě 4. To ale znamená, že se rozhodujeme na základě znaku, který jsme už jednou viděli při kontrole našeho prvního výskytu. Může za to fakt, že za podřetězec $* \uparrow*$ v našem vzoru p se nám dosadil celý podstrom $a2 \ a0 \uparrow0 \ a0 \uparrow0 \uparrow2$ ze stromu t. Kvůli tomu, že za podřetězec $* \uparrow*$ se nám může dosadit libovolný podstrom, jehož reprezentace bude mít délku $l \geq 2$, znak, podle kterého budeme dělat posun, se najednou může ocitnout v našem porovnávacím rozsahu. Pokud bychom chtěli zachovat tu myšlenku algoritmu Quick Search, tedy bychom chtěli skenovat hledaný vzor zleva doprava a posouvat vzor v prohledávaném stromě ve stejném směru, zjistíme, že velmi často budeme moci dělat pouze malé posuny.

Příklad 9. Mějme linearizovaný stromový vzor (opět v prefixové ohodnocené zarážkové notaci) $p = a2 \ a1 \ a0 \uparrow0 \uparrow1 * \uparrow* \uparrow2$ a dále libovolný prohledávaný strom t. Předpokládejme, že jsme momentálně ověřili výskyt hledaného vzoru na pozici i ve stromu t, tedy $t[i]$. Nyní se tedy chceme posunout na další pozici, kde by se vzor mohl vyskytovat. O tomto posunu budeme rozhodovat

na základu znaku na pozici $t[i + m]$, kde m je délka reprezentace našeho vzoru, tedy v našem případě 8. Předpokládejme, že na této pozici nalezneme libovolný nezarážkový znak libovolné arity. Pak na základě tohoto znaku nikdy nemůžeme udělat posun větší než 1. Může za to fakt, že tento znak může být součástí výskytu vzoru na pozici $i + 1$, neboť může být součástí podstromu, který bude dosazen na místo zástupného řetězce $* \uparrow *$.

Předpokládejme například, že na pozici $t[i + m]$ nalezneme znak $a1$. Pak může výskyt vzoru p na pozici $i+1$ (tedy podřetězec začínající na pozici $i+1$ a končící na pozici $i+14$) vypadat následovně: $a2$ $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $a1$ $a1$ $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $\uparrow 1$ $\uparrow 1$ $\uparrow 2$. Všimněme si, že v tomto případě se na pozici $t[i+8]$ nachází právě znak $a1$, podle kterého jsme tedy určovali posun. Obdobný příklad můžeme učinit pro jakýkoliv nezarážkový znak libovolné arity. V podstatě se dá říct, že pokud bude náš vzor obsahovat alespoň jeden zástupný symbol $*$, pak posuny učiněné na základě libovolných nezarážkových znaků budou mít vždy velikost pouze 1. Větší posuny si nemůžeme dovolit, protože bychom mohli přeskočit nějaký výskyt hledaného vzoru.

Příklad 10. Mějme opět linearizovaný stromový vzor (v prefixové ohodnocené zarážkové notaci) $p = a2$ $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $*$ $\uparrow *$ $\uparrow 2$ a dále libovolný prohledávaný strom t. Opět předpokládejme, že jsme právě ověřili výskyt hledaného vzoru na pozici i ve stromu t, tedy $t[i]$. Nyní se tedy rozhodujeme o posunu na základu znaku na pozici $t[i + m]$, kde m je délka reprezentace našeho vzoru, tedy pro tento příklad 8. Předpokládejme, že na této pozici nalezneme znak $\uparrow 0$, pak můžeme podobně jako v předchozím příkladu udělat posun pouze o 1 znak doprava. To proto, že se na pozici $t[i+1]$ může v takové situaci nacházet výskyt vzoru začínající na pozici $i + 1$ a končící na pozici $i + 10$ vypadající následovně: $a2$ $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $\uparrow 2$. Povšimněme si, že v tomto případě se znak $\uparrow 0$ nachází na pozici $i + 8$. Pro tento konkrétní vzor už se na pozici $i + 8$ nemůže nacházet jiný zarážkový symbol, který by mohl být součástí podřetězce dosazeného na místo zástupného $* \uparrow *$. Pokud bychom totiž chtěli, aby se na daném místě vyskytoval například znak $\uparrow 1$, pak by takový znak musel být součástí minimálně nejmenšího možného podstromu, který obsahuje nějaký znak arity 1. Jenomže nejmenší takový podstrom bude mít řetězcovou reprezentaci $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ (snadno si uvědomíme, že menší strom, který by obsahoval znak arity 1 neexistuje). Pak ale při dosazení tohoto nejmenšího podstromu na místo $* \uparrow *$ se bude najednou znak $\uparrow 1$ vyskytovat až na pozici $i + 9$, i když my budeme o posunu rozhodovat na základě pozice $i + 8$.

Co kdyby se ale zástupný podřetězec $* \uparrow *$ vyskytoval v reprezentaci našeho vzoru někde více vlevo? Představme si nyní jiný vzor p_2, jehož řetězcová reprezentace bude následující: $p_2 = a2 * \uparrow * a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $\uparrow 2$. Nyní ještě jednou předpokládejme, že jsme právě ověřili výskyt vzoru p_2 na pozici i. Nyní už na pozici $i + m$ (kde m je opět délka reprezentace našeho vzoru, tedy 8) může být například i znak $\uparrow 2$, a přitom si stále budeme moci dovolit pouze posun doprava o velikosti 1. Na pozici $i+1$ se teď totiž může nacházet výskyt

vzoru končící na pozici $i + 14$ a vypadající následovně: $a2$ $a1$ $a2$ $a0$ $\uparrow 0$ $a0$ $\uparrow 0$ $\uparrow 2$ $\uparrow 1$ $a1$ $a0$ $\uparrow 0$ $\uparrow 1$ $\uparrow 2$, s tím, že podřetězec od pozice $i + 2$ do pozice $i + 9$ je reprezentací podstromu, který byl doplněn na místo zástupného podřetězce $*$ $\uparrow *$ a opět vidíme, že na pozici $i + m$ máme nyní znak $\uparrow 2$. Na tomto místě by mohl být i libovolný jiný zarážkový znak nižší arity, a stejně nebudeme moci udělat posun větší velikosti než 1.

Příklady 9 a 10 demonstrují, že při skenování vzoru ve stejném směru, ve kterém vzor posouváme v prohledávaném stromu, budou posuny velmi často poměrně malé. Příklad 9 ukazuje, že pokud se kdekoliv v našem vzoru nachází zástupný symbol $*$, pak velikost posunu pro jakýkoliv nezarážkový symbol bude vždy pouze 1. Na příkladu 10 pak vidíme, že i pro zarážkové symboly může být velikost maximálního posunu pouze 1. Čím více vlevo v řetězcové reprezentaci našeho vzoru se bude nacházet podřetězec $* \uparrow *$, tím větší aritu budou mít zarážkové symboly, pro které je velikost maximálního posunu stále pouze 1.

Dostáváme se tedy do situace, kdy pro velkou část vzorů bude velikost posunů 1 pro relativně velký počet znaků z abecedy. My bychom ideálně chtěli, aby posuny byly často větší než 1, protože pokud se budeme většinou posouvat v reprezentaci prohledávaného stromu pouze o 1, výkon našeho algoritmu bude degradovat téměř na úroveň naivního algoritmu, který v každém kroku posune reprezentaci hledaného vzoru o 1 a poté zkontroluje, zda se na dané pozici nachází výskyt.

V následující části ukážeme, že při posouvání vzoru v prohledávaném stromu jedním směrem a skenování výskytu v opačném směru bude velikost posunů často větší než 1, a algoritmus založený na těchto posunech bude mít velmi dobré vlastnosti.

2.2 Tabulka posunů při skenování vzoru zprava doleva

Nyní se podíváme na to, jak bude vypadat tabulka posunů (BCS) v případě, že budeme posouvat vzor v hledaném podstromu směrem zleva doprava a při kontrole výskytu budeme skenovat vzor v opačném směru (tedy zprava doleva).

Budeme tedy počítat s tím, že jsme ověřili výskyt vzoru končící na pozici $i + m - 1$ v reprezentaci prohledávaného stromu a nyní se podíváme na znak na pozici $i + m$, kde m je délka řetězcové reprezentace našeho vzoru. Chceme pro všechny znaky z abecedy znát délku, o kterou můžeme posunout pozici vzoru doprava v reprezentaci prohledávaného stromu, pokud se daný znak nachází na pozici $i + m$. Chceme vždy učinit co největší posun, avšak takový, aby nehrozilo že přeskočíme nějaký výskyt vzoru v prohledávaném stromu. Pro účely této části budeme pracovat s prefixovou ohodnocenou zarážkovou

notací. V části 2.5 pak sestavíme podobnou tabulku pro variantu algoritmu, kde budeme vzor posouvat zprava doleva a skenování provádět zleva doprava.

Hodnota v tabulce posunů (BCS) pro znak x bude určena jako minimum ze tří hodnot:

1. $m + 1$
2. $j : vzor[m - j] = x$ a platí $m \geq j > 0$
3. $j + arita(x) : vzor[m - j] = *$ a platí $m \geq j > 0$ pro $x \notin \uparrow_n$ nebo $j - 1 : vzor[m - j] = *$ a platí $m > j > 1$ pro $x \in \uparrow_n$

První bod nám slouží jako omezení posunu shora délkou vzoru plus jedna. Skutečně, pokud se daný znak x ve vzoru nebude nikde vyskytovat, a zároveň by se ve vzoru nevyskytoval ani žádný zástupný znak $*$, pak můžeme na základě tohoto znaku x posunout vzor v prohledávaném podstromu doprava rovnou za tento znak x, tedy se posuneme na základě znaku na pozici $i + m$ z pozice i na pozici $i + m + 1$.

Druhý bod definuje minimální bezpečnou vzdálenost posunu pro znaky, které se přímo vyskytují v našem vzoru (tedy nejsou součástí případného podstromu, který je dosazen na místo zástupného symbolu $*$). Minimální bezpečná vzdálenost musí být taková, že posun zarovná nejpravější výskyt znaku x v našem vzoru s jeho výskytem v prohledávaném stromě na pozici $i + m$. V tomto případě předpokládáme, že na místo zástupného podřetězce $* \uparrow *$ by byl dosazen nejmenší možný podstrom, tedy podstrom skládající se z jednoho vrcholu, který by obsahoval znak arity 0, tedy jeho řetězcová reprezentace by byla například $a0 \uparrow 0$. Poznamenejme, že nejmenší možná velikost posunu z tohoto bodu je 1, což bude velikost posunu pro nejpravější znak z reprezentace našeho vzoru.

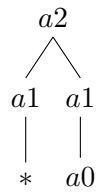
Třetí bod pak řeší případ, kdy by se znak vyskytoval v řetězcové reprezentaci podstromu, který by byl dosazen na místo zástupného symbolu $*$. Zde budeme hodnoty pro zarážkové symboly a nezarážkové symboly počítat trochu jinak. Pro nezarážkové symboly si nejprve nalezneme nejpravější výskyt zástupného znaku $*$ v řetězcové reprezentaci našeho vzoru. Základní velikost posunu pak vypočteme jako vzdálenost tohoto zástupného znaku od konce řetězcové reprezentace vzoru plus jedna. Skutečná velikost posunu pro jednotlivé nezarážkové znaky se pak bude lišit na základě jejich arity. K základní velikosti připočteme dvojnásobek arity znaku, a to bude naše skutečná velikost posunu. Zde předpokládáme, že daný znak bude součástí nejmenšího možného podřetězce, který tento znak obsahuje. To znamená, že pro znak arity 0 je délka tohoto nejmenšího možného podřetězce 2, například pro podřetězec $a0 \uparrow 0$. Tento podřetězec se tedy přesně dosadí na místo podřetězce $* \uparrow *$ a délka potenciální řetězcové reprezentace výskytu vzoru, jejíž součástí by byl tento podřetězec, se tím nemění. Pro znaky arity 1 už je ale délka tohoto nejmenšího možného podřetězce 4, například pro podřetězec $a1\ a0 \uparrow 0 \uparrow 1$.

V tomto případě už se při dosazení tohoto podřetězce na místo zástupného podřetězce $* \uparrow *$ objeví náš znak s aritou 1 na místě, které bude od v řetězcové reprezentaci potenciálního výskytu o dva znaky více vzdálené od konce této reprezentace, než je vzdálený nejpravější výskyt zástupného symbolu $*$ od konce reprezentace našeho vzoru. Díky tomu můžeme učinit o dva znaky větší posun, protože víme, že od výskytu tohoto znaku do konce řetězcové repreztace potenciálního výskytu bude muset být minimálně o dva znaky více (přibyde alespoň $a0 \uparrow 0$, s tím že $\uparrow 1$ nahradí $\uparrow *$), než kolik je znaků od nejpravějšího výskytu zástupného znaku $*$ do konce řetězcové reprezentace našeho vzoru. Pro znaky s aritou 2 je už minimální délka tohoto nejmenšího možného podřetězce 6 (například $a2\ a0 \uparrow 0\ a0 \uparrow 0 \uparrow 2$) a tak dále.

Pro zarážkové symboly je situace trochu horší v tom smyslu, že velikost posunu bude stejná pro všechny zarážkové znaky bez ohledu na aritu. Na místě nejpravějšího výskytu zástupného zarážkové znaku $\uparrow *$ v řetězcové reprezentaci našeho vzoru se totiž po dosazení může nacházet skutečně libovolný zarážkový znak. V tomto případě se tedy nemůžeme bezpečně posunout o větší vzdálenost, než je vzdálenost nejpravějšího výskytu znaku $\uparrow *$ od konce řetězcové reprezentace našeho vzoru plus jedna (což je ekvivalentní vzdálenosti nejpravějšího výskytu znaku $*$ od konce řetězcové reprezentace našeho vzoru).

Můžeme si všimnout, že velikost posunů získaných třetím bodem bude poměrně silně ovlivněná pozicí zástupného symbolu $*$ v našem vzoru. V podstatě platí, že čím vzdálenější bude nejpravější výskyt zástupného podřetězce $* \uparrow *$ od konce řetězcové reprezentace našeho vzoru, tím větší budou posuny jak pro zarážkové, tak pro nezarážkové znaky (alespoň tedy velikosti posunů získané třetím bodem, může se stát, že velikosti posunů získané ostatními body budou menší). V části 2.5 si ukážeme obrácenou variantu algoritmu, kde budeme posouvat reprezentaci hledaného vzoru zprava doleva a skenování vzoru provádět zleva doprava, tam to pak bude přesně naopak a posuny budou tím větší, čím bude nejlevější zástupný podřetězec $* \uparrow *$ vzdálenější od začátku řetězcové reprezentace vzoru.

Příklad 11 ukazuje, jak bude vypadat tabulka posunů pro konkrétní vzor. Na obrázku 2.2 pak můžeme vidět vzor z příkladu 11. Poznamenejme, že pro znaky $*$ a $\uparrow *$ nepotřebujeme v tabulce posunů žádný záznam, neboť se tyto znaky nemohou vyskytovat v řetězcové reprezentaci prohledávaného stromu. Tabulka posunů se dá zkonstruovat algoritmem 6.



Obrázek 2.2: Vzor p z příkladu 11

Příklad 11. Mějme stromový vzor p s následující řetězcovou reprezentací v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(p) = a2\ a1\ *\ \uparrow *\ \uparrow 1\ a1\ a0\ \uparrow 0\ \uparrow 1\ \uparrow 2$ nad abecedou $\mathcal{A} = \{a3, a2, a1, a0, *, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow *\}$.

Pak hodnoty v tabulce posunů (BCS) pro tento vzor budou následující:
$BCS[a3] = \min(\{11\} \cup \emptyset \cup \{14\}) = 11,$
$BCS[a2] = \min(\{11\} \cup \{10\} \cup \{12\}) = 10,$
$BCS[a1] = \min(\{11\} \cup \{5, 9\} \cup \{10\}) = 5,$
$BCS[a0] = \min(\{11\} \cup \{4\} \cup \{8\}) = 4,$
$BCS[\uparrow 3] = \min(\{11\} \cup \emptyset \cup \{7\}) = 7,$
$BCS[\uparrow 2] = \min(\{11\} \cup \{1\} \cup \{7\}) = 1,$
$BCS[\uparrow 1] = \min(\{11\} \cup \{2, 6\} \cup \{7\}) = 2,$
$BCS[\uparrow 0] = \min(\{11\} \cup \{3\} \cup \{7\}) = 3.$

Název algoritmu: Konstrukce BCS pro algoritmus Quick Search nad stromy.
Vstup: Stromový vzor *pattern* v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(pattern)$ o velikosti m nad abecedou $\mathcal{A}$ prohledávaného stromu *subject*.
Výstup: Tabulka posunů $BCS(pref\_ranked\_bar(pattern))$.

```
1  begin
2      s := m + 1
3      for i := 0 to m − 1 do
4          if pref_ranked_bar(pattern)[i] = ∗ then s = m − i;
5      end
6      foreach x ∈ A do BCS[x] = m + 1;
7      foreach x ∈ A do
8          if x ∉ A↑ then shift := s + Arity(x) ∗ 2;
9          else shift := s − 1;
10         if BCS[x] > shift then BCS[x] := shift;
11     end
12     for i := 0 to m − 1 do
13         if pref_ranked_bar(pattern)[i] ∉ {∗, ↑∗} and
             BCS[pref_ranked_bar(pattern)[i]] > (m − i) then
             BCS[pref_ranked_bar(pattern)[i]] := m − i ;
14     end
15 end
```

Algoritmus 6: Konstrukce tabulky posunů pro algoritmus Quick Search pro hledání stromových vzorů

Pokud bychom chtěli porovnat proces konstrukce tabulky posunů pro algoritmus Quick Search s procesem konstrukce tabulky posunů pro algoritmus protisměrného vyhledávání ve stromech (algoritmus 3), můžeme si všimnout několika rozdílů. Především výsledné hodnoty v tabulce posunů (a tedy i po-

tenciální posuny při procesu hledání vzoru) jsou pro většinu symbolů větší o 1. Pokud bychom si například zkonstruovali tabulku posunů pro algoritmus protisměrného vyhledávání ve stromech pro vzor p z příkladu 11, pak by hodnoty pro sedm z osmi znaků v tabulce posunů byly o 1 menší než v tabulce posunů pro algoritmus Quick Search nad stromy. Pouze pro znak $\uparrow 2$ by hodnota v tabulce posunů pro algoritmus protisměrného vyhledávání ve stromech byla vyšší než v tabulce pro algoritmus Quick Search nad stromy. Intuitivně bychom tedy mohli tvrdit, že pokud algoritmus nebude narážet na zmíněný znak $\uparrow 2$ příliš často, budou posuny učiněné algoritmem Quick Search nad stromy průměrně větší než posuny učiněné algoritmem protisměrného vyhledávání ve stromech při stejném hledaném vzoru a stejném prohledávaném stromu.

Dalším rozdílem, kterého si můžeme povšimnout, je fakt, že v algoritmu 3 pro konstrukci tabulky posunů pro algoritmus protisměrného vyhledávání ve stromech na řádce 9 musíme provádět kontrolu, zda není velikost základního posunu pouze 1. Pak bychom bez této podmínky pro zarážkové symboly totiž získali hodnotu posunu 0, což samozřejmě nechceme. V algoritmu 6 pro konstrukci tabulky posunů pro algoritmus Quick Search nad stromy ale tuto podmínku nepotřebujeme. To je dáno faktem, že pro velikost základního posunu s bude vždy platit $s \geq 2$. Pro nejmenší možný vzor $* \uparrow *$ bude velikost tohoto základního posunu rovna právě 2, pro všechny ostatní možné vzory bude tato velikost dokonce ještě větší.

2.3 Pseudokód základní varianty algoritmu

Obě pomocné struktury potřebné pro náš algoritmus již umíme zkonstruovat (algoritmus 6 pro konstrukci tabulky posunů a algoritmus 4 pro konstrukci podstromové skokové tabulky – ten jsme sice představili v části 1.6 o algoritmu protisměrného vyhledávání ve stromech, ale náš algoritmus bude využívat stejnou podstromovou skokovou tabulku, tedy může využít stejný algoritmus pro její konstrukci). Můžeme tedy již představit pseudokód celého algoritmu.

Ihned si můžeme povšimnout, že podobně jako v algoritmu 2, i zde potřebujeme podmínku, která zajistí že se nedotážeme na znak za koncem řetězcové reprezentace prohledávaného stromu. Zde jí vidíme na řádce 16. V případě, že by řetězcová reprezentace hledaného vzoru délky m byla již zarovnána s m nejpravějšími znaky v řetězcové reprezentaci prohledávaného podstromu, pak na řádce 16 musíme vyskočit z while cyklu, abychom se následně na řádce 17 nedotazovali na znak na pozici za koncem řetězcové reprezentace prohledávaného stromu. Této podmínky se dá zbavit například přidáním nějakého znaku na konec řetězcové reprezentace prohledávaného stromu bez zvýšení jeho délky tak, aby ještě existoval znak na pozici n (kde n je délka řetězcové reprezentace prohledávaného stromu), který nám už posune index i tak, že se podmínka na řádce 3 $i \leq (n - m)$ v další iteraci vyhodnotí jako nepravda

Název algoritmu: Algoritmus Quick Search nad stromy.
Vstup: Řetězcová reprezentace stromu *subject*
v $pref\_ranked\_bar(subject)$ notaci délky n, řetězcová
reprezentace stromového vzoru *pattern*
v $pref\_ranked\_bar(pattern)$ notaci délky m,
$SJT(pref\_ranked\_bar(subject))$
a $BCS(pref\_ranked\_bar(pattern))$.
Výstup: Všechny výskyty stromového vzoru *pattern* ve stromu
subject.

```
1  begin
2      i := 0
3      while i ≤ (n − m) do
4          j := m − 1
5          position := i + j
6          while j ≥ 0 and position ≥ 0 do
7              if pref_ranked_bar(subject)[position] =
                pref_ranked_bar(pattern)[j] then
8                  position := position − 1
9              else if pref_ranked_bar(pattern)[j] = ↑∗ and
                pref_ranked_bar(subject)[position] ∈ A↑ then
10                 position := SJT(pref_ranked_bar(subject))[position]
11                 j = j − 1 {Subtree skip}
12             else break;
13             j := j − 1
14         end
15         if j = −1 then output(position + 1);
16         if i = (n − m) then break;
17         i := i + BCS[pref_ranked_bar(subject)[i + m]]
18     end
19 end
```

Algoritmus 7: Algoritmus Quick Search pro hledání stromových vzorů ve stromech

a další iterace se již neprovede. Zde jsme zvolili variantu s podmínkou navíc, především kvůli pochopitelnosti algoritmu, protože řešení s přidaným znakem není příliš přímočaré. Při testování algoritmu se ale ukázalo, že pokud se podmínky zbavíme, algoritmus bude nezanedbatelně rychlejší. Jak se konkrétně podmínky zbavit ještě popíšeme v části 3.3.

Řádky 4 až 14 jsou zodpovědné za ověření toho, zda existuje výskyt vzoru končící na pozici $i + m - 1$. Znaky řetězcové reprezentace vzoru a řetězcové reprezentace prohledávaného stromu jsou porovnávány odzadu, tedy zprava doleva. Pokud se znak na pozici ve vzoru přímo shoduje se znakem na odpovídající pozici v prohledávaném stromu, pak se pokračuje porovnáním znaku na pozici vlevo (řádky 7 a 8). Pokud je na pozici ve vzoru zástupný znak $\uparrow *$ (protože procházíme řetězcovou reprezentaci vzoru zprava doleva, narazíme dříve na zarážkový zástupný znak $\uparrow *$ než na nezarážkový zástupný znak $*$), a na odpovídající pozici v prohledávaném stromu nějaký zarážkový znak libovolné arity, pak se ve vzoru posuneme doleva rovnou o dva znaky (poprvé na řádce 11 a podruhé na řádce 13). Který znak řetězcové reprezentace prohledávaného stromu máme v dalším kroku porovnávat s novým aktuálním znakem reprezentace vzoru pak zjistíme za pomoci podstromové skokové tabulky. Pro zarážkový znak v prohledávaném stromu máme totiž v podstromové skokové tabulce index znaku, který se nachází o jeden znak vlevo od začátku reprezentace podstromu, která končí právě daným zarážkovým znakem. A přesně znak nacházející se na této pozici pozici určené podstromovou skokovou tabulkou v dalším kroku potřebujeme.

Na řádce 15 pak pouze zkontrolujeme, zda jsme našli výskyt vzoru končící na pozici $i + m - 1$, pokud ano, tak přidáme do výstupu počáteční pozici tohoto výskytu. A konečně na řádce 17 učiníme posun na základě znaku na pozici $i + m$ podle naší tabulky posunů sestrojené algoritmem 6.

Příklad 12 společně s tabulkou 2.1 demonstrují běh algoritmu na příkladu s konkrétními vstupy. Strom a vzor z příkladu 12 si pak můžeme prohlédnout v rámci obrázku 2.3 (snadno nahlédneme, že daný vzor se ve stromu skutečně vyskytuje právě třikrát).

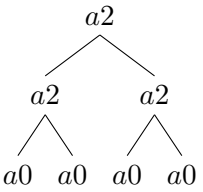
Příklad 12. Mějme stromový vzor p s řetězcovou reprezentací v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(p) = a2\ *\ \uparrow *\ *\ \uparrow *\ \uparrow 2$ nad abecedou $\mathcal{A}_1 = \{a2, a1, a0, *, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow *\}$ a strom t s řetězcovou reprezentací v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(t) = a2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$ nad abecedou $\mathcal{A}_2 = \{a2,\ a1,\ a0,\ \uparrow 2,\ \uparrow 1,\ \uparrow 0\}$. Pro vzor p dostaneme následující hodnoty v tabulce posunů (algoritmem 6): $BCS[a2] = 6$, $BCS[a1] = 5$, $BCS[a0] = 3$, $BCS[\uparrow 2] = 1$, $BCS[\uparrow 1] = 2$, $BCS[\uparrow 0] = 2$. Tabulka 2.1 ukazuje běh algoritmu 7 nad vzorem p a prohledávaným stromem t. Reprezentace delších podstromů dosazených na místo zástupných podřetězců $* \uparrow *$ jsou v tabulce naznačeny jako $*\rightarrow \leftarrow *$.

Běh algoritmu 7 pro strom a vzor z příkladu 12 začne kontrolou, zda exis-

Tabulka 2.1: Běh algoritmu 7 nad stromem t a vzorem p z příkladu 12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t)$
14	7	4	1	6	3	0	13	10	7	12	9	6	-1	$SJT(t)$
					$\uparrow 2$									$\uparrow 0 \neq \uparrow 2$, $posun = 1$
	$a2$	$*\rightarrow$	$\leftarrow *$	$*\rightarrow$	$\leftarrow *$	$\uparrow 2$								nález, $posun = 6$
							$a2$	$*\rightarrow$	$\leftarrow *$	$*\rightarrow$	$\leftarrow *$	$\uparrow 2$		nález, $posun = 1$
$a2$	$*\rightarrow$					$\leftarrow *$	$*\rightarrow$					$\leftarrow *$	$\uparrow 2$	nález, *break*

(a) Vzor p z příkladu 12



(b) Strom t z příkladu 12

Obrázek 2.3: Strom t a vzor p z příkladu 12

tuje výskyt vzoru končící na pozici 5 ($i + m - 1$, i je v tu chvíli 0 a m je 6). Protože $\uparrow 0 \neq \uparrow 2$, hned první porovnávaný znak se neshoduje a algoritmus se tedy posune na další možnou pozici výskytu. To udělá na základě znaku na pozici 6 ($i + m$). Tam se nachází znak $\uparrow 2$, tedy se provede posun pouze o 1 znak doprava. Nyní algoritmus zkontroluje, zda existuje výskyt vzoru končící na pozici 6. Tam už skutečně výskyt existuje (za oba výskyty zástupného podřetězce $* \uparrow *$ je v tomto případě dosazen podřetězec $a0 \uparrow 0$), takže ho algoritmus přidá do výstupu a na základě znaku na pozici 7 udělá další posun, tentokrát o velikosti 6 znaků doprava. Poté algoritmus zkontroluje existenci výskytu končícího na pozici 12, kde je výskyt též nalezen a přidán do výstupu. Další posun učiní algoritmus na základě znaku na pozici 13 a bude pouze o 1 znak doprava. Poté tedy algoritmus provede kontrolu existence výskytu končícího na pozici 13, který existuje (a začíná na pozici 0, zde se za výskyty zástupného podřetězce $* \uparrow *$ v reprezentaci vzoru dosadí delší podřetězce z reprezentace prohledávaného stromu, jak je vidět v tabulce 2.1), a protože v tuto chvíli máme reprezentaci vzoru již zarovnanou s m nejpravějšími znaky reprezentace prohledávaného stromu, po přidání nalezeného výskytu do výstupu se uplatní podmínka na řádce 16 algoritmu 7 a algoritmus skončí.

Pro porovnání – tabulka 2.2 ukazuje průběh algoritmu protisměrného vyhledávání ve stromech nad stejným párem prohledávaného stromu a hledaného vzoru. Můžeme si povšimnout, že námi prezentovaný algoritmus pro tyto konkrétní vstupy otestuje o jednu možnou pozici konce výskytu méně než algoritmus protisměrného vyhledávání ve stromech. Existují samozřejmě i případy

Tabulka 2.2: Běh algoritmu 5 nad stromem t a vzorem p z příkladu 12

0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t)$
14	7	4	1	6	3	0	13	10	7	12	9	6	-1	$SJT(t)$
					$\uparrow 2$									$\uparrow 0 \neq \uparrow 2$, $posun = 1$
	$a2$	$*\rightarrow$	$\leftarrow *$	$*\rightarrow$	$\leftarrow *$	$\uparrow 2$								nález, $posun = 1$
							$\uparrow 2$							$a2 \neq \uparrow 0$, $posun = 5$
							$a2$	$*\rightarrow$	$\leftarrow *$	$*\rightarrow$	$\leftarrow *$	$\uparrow 2$		nález, $posun = 1$
$a2$	$*\rightarrow$					$\leftarrow *$	$*\rightarrow$					$\leftarrow *$	$\uparrow 2$	nález, $posun = 1$

stromů a vzorů, na kterých by námi prezentovaný algoritmus otestoval naopak více možných pozic, než algoritmus protisměrného vyhledávání ve stromech, v průměru bychom ale očekávali, že jich otestuje méně, neboť bude činit větší posuny, jak už jsme popisovali v části 2.2 a potvrzují to i výsledky testování.

2.4 Důkaz korektnosti algoritmu

V této části ukážeme, že prezentovaný algoritmus skutečně nalezne všechny výskyty hledaného vzoru v prohledávaném stromě. Přímo v důkazu se sice odvoláváme na algoritmus 7, tedy již představenou standardní variantu algoritmu, obdobný důkaz ovšem funguje pro všechny varianty algoritmu, jak pro standardní variantu, tak pro obrácené varianty, které představíme dále v částech 2.5 a 2.6.

Věta 10. *Pro daný strom t a stromový vzor p (oba v prefixové ohodnocené zarážkové notaci) nalezne algoritmus 7 indexy všechny výskytů vzoru p v prohledávaném stromě t, za předpokladu že využívá pomocnou tabulku posunů $BCS(pref\_ranked\_bar\_(p))$ zkonstruovanou algoritmem 6.*

Důkaz. Potřebujeme dokázat, že při posouvání řetězcové reprezentace hledaného vzoru p v reprezentaci prohledávaného stromu t na základě hodnot v tabulce posunů $BCS(pref\_ranked\_bar\_(p))$ nemůže algoritmus přeskočit žádný výskyt vzoru p v prohledávaném stromě t.

Mějme $c \in \mathcal{A}$. Předpokládejme, že existuje výskyt vzoru p na pozici $0 < i < BCS(pref\_ranked\_bar\_(p))[c]$. Znak c se pak musí nacházet na nějaké pozici i buď přímo, nebo jako součást nějakého podstromu, který byl dosazen na místo zástupného symbolu $*$. Z části 2.2 vyplývá, že hodnota $BCS(pref\_ranked\_bar\_(p))[c]$ je vypočtena za pomoci vzdálenosti nejpravějšího výskytu daného znaku c od konce řetězcové reprezentace vzoru p, tedy dostáváme spor s tím, že by hodnota v tabulce posunů pro znak c byla větší než tato vzdálenost, a dále z vzdálenosti nejpravějšího výskytu zástupného podřetězce $* \uparrow *$ od konce řetězcové reprezentace vzoru p. Pokud se znak c

nachází v podstromu, který byl dosazen na místo zástupného symbolu $*$, pak byla ovšem velikost posunu vypočtena pro případ, že by byl na místo tohoto symbolu dosazen nejmenší možný podstrom obsahující znak c. Skutečný dosazený podstrom nemůže být menší, tedy dostáváme opět spor. Algoritmus tedy nemůže přeskočit žádný výskyt vzoru p v prohledávaném stromě t. □

2.5 Obrácená varianta algoritmu se skenováním vzoru zleva doprava

Doposud jsme pracovali s variantou algoritmu, která procházela řetězcovou reprezentaci prohledávaného stromu zleva doprava a při kontrole existence výskytu vzoru na určité pozici skenovala řetězcovou reprezentaci vzoru zprava doleva. My ovšem můžeme postupovat i opačně, tedy reprezentaci prohledávaného stromu procházet od konce zprava doleva, a při kontrole existence výskytů skenovat reprezentaci vzoru zleva doprava. Této alternativní variantě algoritmu budeme říkat *obrácená* (anglicky reversed). Prozatím budeme opět pracovat s prefixovou ohodnocenou zarážkovou notací, ale v části 2.6 ukážeme i variantu algoritmu, která umí pracovat s prefixovou ohodnocenou notací.

Podstromová skoková tabulka pro tuto obrácenou variantu algoritmu může stále zůstat stejná. Tabulka posunů (BCS) ale musí být opět konstruována jiným způsobem. Podobně jako v případě standardní varianty algoritmu, i zde bude hodnota v tabulce posunů určena jako minimum ze tří hodnot:

1. $m + 1$
2. $j : vzor[j-1] = x$ a platí $m \geq j > 0$
3. $j : vzor[j] = \uparrow *$ a platí $m > j > 0$ pro $x \notin \uparrow_n$ nebo $j + arita(x) : vzor[j-1] = \uparrow *$ a platí $m > j \geq 1$ pro $x \in \uparrow_n$

Podobně jako v případě tabulky posunů pro standardní variantu algoritmu, i zde první bod slouží jako omezení posunu shora délkou vzoru plus jedna.

Druhý bod opět definuje minimální bezpečnou vzdálenost posunu pro znaky, které se přímo vyskytují v reprezentaci našeho vzoru. Zatímco ve standardní variantě jsme ovšem chtěli zarovnat nejpravější výskyt znaku x ve vzoru s jeho výskytem v prohledávaném stromě na pozici $i+m$, zde chceme zarovnat pro změnu nejlevější výskyt znaku x s jeho výskytem v prohledávaném stromě na pozici $i-1$. Obdobně jako při konstrukci tabulky posunů pro standardní variantu algoritmu, i zde předpokládáme, že na místo zástupného podřetězce $* \uparrow *$ bude dosazen nejmenší možný podstrom. Nejmenší možný posun získaný tímto bodem je 1, což bude velikost posunu pro nejlevější znak z reprezentace našeho vzoru.

Třetí bod nám opět řeší případ, kdy se znak vyskytuje v řetězcové reprezentaci podstromu, který je dosazen na místo zástupného symbolu $*$ v našem

vzoru. Zde budeme opět hodnoty pro zarážkové a nezarážkové symboly počítat rozdílně. V tomto případě jsou horší nezarážkové znaky. Pro všechny nezarážkové znaky se hodnota v tabulce posunů bude rovnat vzdálenosti nejlevějšího výskytu zástupného znaku $*$ od začátku řetězcové reprezentace vzoru plus jedna (což je ekvivalentní vzdálenosti nejlevějšího výskytu zástupného zarážkového znaku $\uparrow*$ od začátku řetězcové reprezentace vzoru). Na místě nejlevějšího výskytu zástupného znaku $*$ v řetězcové reprezentaci vzoru se totiž po dosazení může nacházet skutečně libovolný nezarážkový znak. Větší posun si tedy nemůžeme bezpečně dovolit.

Pro zarážkové symboly vyjdeme z nějaké základní velikosti posunu, kterou budeme následně zvyšovat podle arity příslušné zarážky. Základní velikost určíme jako vzdálenost nejlevějšího výskytu zástupného zarážkového symbolu $\uparrow*$ v reprezentaci našeho vzoru plus jedna. K této základní velikosti pak připočteme dvojnásobek arity konkrétního zarážkového znaku a tím dostaneme skutečnou velikost posunu pro daný znak. Podobně jako když jsme určovali velikost posunů pro nezarážkové znaky u standardní varianty algoritmu v části 2.2, i zde předpokládáme, že na místo zástupného podřetězce $* \uparrow*$ bude dosazen nejkratší možný podřetězec obsahující daný zarážkový znak. Pro zarážkový znak arity 0 se může jednat například o podřetězec $a0 \uparrow 0$, pro zarážkový znak arity 1 například podřetězec $a1\ a0 \uparrow 0 \uparrow 1$ a tak dále.

Stejně jako u standardní varianty algoritmu, i u obrácené varianty algoritmu platí, že velikost posunů získaných třetím bodem bude značně ovlivněná pozicí zástupného symbolu $*$ v našem vzoru. V tomto případě platí, že čím vzdálenější bude nejlevější výskyt zástupného podřetězce $* \uparrow*$ od začátku řetězcové reprezentace vzoru, tím větší budou posuny jak pro zarážkové, tak pro nezarážkové znaky (samozřejmě zde máme na mysli posuny získané třetím bodem, posuny získané ostatními body mohou být menší).

Příklad 13 ukazuje, jak bude vypadat tabulka posunů pro obrácenou variantu algoritmu pro konkrétní vzor. Úmyslně jsme zvolili stejný vzor jako v příkladu 11 v části 2.2, aby bylo možné hodnoty porovnat. Opět připomeneme, že v tabulce posunů nepotřebujeme mít hodnotu posunu pro zástupné znaky $*$ a $\uparrow*$, protože ty se nemohou vyskytovat v reprezentaci prohledávaného stromu. Pro sestrojení tabulky posunů pro obrácenou variantu algoritmu pak lze využít algoritmus 8.

Příklad 13. Mějme stromový vzor p s následující řetězcovou reprezentací v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(p) = a2\ a1\ *\ \uparrow* \uparrow 1\ a1\ a0\ \uparrow 0\ \uparrow 1\ \uparrow 2$ nad abecedou $\mathcal{A} = \{a3, a2, a1, a0, *, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow *\}$.

Pak hodnoty v tabulce posunů (BCS) pro obrácenou variantu algoritmu pro tento vzor budou následující:
$BCS[a3] = \min(\{11\} \cup \emptyset \cup \{3\}) = 3$, $BCS[a2] = \min(\{11\} \cup \{1\} \cup \{3\}) = 1$,
$BCS[a1] = \min(\{11\} \cup \{2, 6\} \cup \{3\}) = 2$, $BCS[a0] = \min(\{11\} \cup \{7\} \cup \{3\}) = 3$,
$BCS[\uparrow 3] = \min(\{11\} \cup \emptyset \cup \{10\}) = 10$, $BCS[\uparrow 2] = \min(\{11\} \cup \{10\} \cup \{8\}) = 8$,
$BCS[\uparrow 1] = \min(\{11\} \cup \{5, 9\} \cup \{6\}) = 5$, $BCS[\uparrow 0] = \min(\{11\} \cup \{8\} \cup \{4\}) = 4$.

Průměrná velikost posunu pro tento konkrétní vzor je při obrácené variantě algoritmu o něco menší než pro standardní variantu algoritmu. To je ale dáno tím, že se zástupný podřetězec $* \uparrow *$ nachází poměrně vlevo v řetězcové reprezentaci tohoto konkrétního vzoru. Pokud bychom si pro náš příklad zvolili jiný vzor, ve kterém by se zástupný podřetězec $* \uparrow *$ nacházel výrazně více vpravo, pak by průměrná velikost posunu byla pravděpodobně větší pro obrácenou variantu algoritmu.

Název algoritmu: Konstrukce BCS pro obrácenou variantu algoritmu Quick Search nad stromy.
Vstup: Stromový vzor *pattern* v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(pattern)$ o velikosti m nad abecedou $\mathcal{A}$ prohledávaného stromu *subject*.
Výstup: Tabulka posunů $BCS(pref\_ranked\_bar(pattern))$.

```
1  begin
2      s := m + 1
3      for i := m − 1 to 0 do
4          if pref_ranked_bar(pattern)[i] = ↑∗ then s = i + 1;
5      end
6      foreach x ∈ A do BCS[x] = m + 1;
7      foreach x ∈ A do
8          if x ∉ A↑ then shift := s − 1;
9          else shift := s + Arity(x) ∗ 2;
10         if BCS[x] > shift then BCS[x] := shift;
11     end
12     for i := m to 1 do
13         if pref_ranked_bar(pattern)[i − 1] ∉ {∗, ↑∗} and
             BCS[pref_ranked_bar(pattern)[i − 1]] > i then
             BCS[pref_ranked_bar(pattern)[i − 1]] := i ;
14     end
15 end
```

Algoritmus 8: Konstrukce tabulky posunů pro obrácenou variantu algoritmu Quick Search pro hledání stromových vzorů

Nyní tedy umíme sestrojit tabulku posunů pro obrácenou variantu našeho algoritmu, podstromová skoková tabulka se konstruuje stále stejně (lze použít algoritmus 4), nic nám tedy nebrání představit pseudokód pro obrácenou variantu algoritmu.

Obrácená varianta algoritmu (algoritmus 9 pracuje tak, že na začátku zarovná řetězcovou reprezentaci vzoru co možná nejvíce vpravo v rámci řetězcové reprezentace prohledávaného stromu. Na řádkách 4 až 14 pak algoritmus ověří, zda existuje výskyt vzoru začínající na aktuální pozici i. Pokud ano, pak je tento výskyt na řádce 15 přidán do výstupu. V každém případě je pak reprezentace vzoru na řádce 17 posunuta doleva v rámci reprezentace pro-

Název algoritmu: Obrácená varianta algoritmu Quick Search nad stromy.

Vstup: Řetězcová reprezentace stromu *subject* v $pref\_ranked\_bar(subject)$ notaci délky n, řetězcová reprezentace stromového vzoru *pattern* v $pref\_ranked\_bar(pattern)$ notaci délky m, $SJT(pref\_ranked\_bar(subject))$ a $BCS(pref\_ranked\_bar(pattern))$.

Výstup: Všechny výskyty stromového vzoru *pattern* ve stromu *subject*.

```
1  begin
2      i := n − m
3      while i ≥ 0 do
4          j := 0
5          position := i
6          while j < m and position < n do
7              if pref_ranked_bar(subject)[position] =
                pref_ranked_bar(pattern)[j] then
8                  position := position + 1
9              else if pref_ranked_bar(pattern)[j] = ∗ and
                pref_ranked_bar(subject)[position] ∈ A↑ then
10                 position := SJT(pref_ranked_bar(subject))[position]
11                 j = j + 1 {Subtree skip}
12             else break;
13             j := j + 1
14         end
15         if j = m then output(i);
16         if i = 0 then break;
17         i := i − BCS[pref_ranked_bar(subject)[i − 1]]
18     end
19 end
```

Algoritmus 9: Obrácená varianta algoritmu Quick Search pro hledání stromových vzorů ve stromech

Tabulka 2.3: Běh algoritmu 9 nad stromem t a vzorem p z příkladů 12 a 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$pref\_ranked\_bar(t)$
14	7	4	1	6	3	0	13	10	7	12	9	6	-1	$SJT(t)$
								a2						$a0 \neq a2$, $posun = 1$
							$a2$	$*\rightarrow$	$\leftarrow *$	$*\rightarrow$	$\leftarrow *$	$\uparrow 2$		nález, $posun = 6$
	$a2$	$*\rightarrow$	$\leftarrow *$	$*\rightarrow$	$\leftarrow *$	$\uparrow 2$								nález, $posun = 1$
$a2$	$*\rightarrow$					$\leftarrow *$	$*\rightarrow$					$\leftarrow *$	$\uparrow 2$	nález, *break*

hledávaného podstromu. Velikost tohoto posunu je určena znakem na pozici $i - 1$ (respektive hodnotou pro tento znak v tabulce posunů). Pouze pokud bychom již měli reprezentaci vzoru zarovnanou s m nejlevějšími znaky reprezentace prohledávaného stromu, nemůžeme vzor dále posouvat. Na řádce 17 bychom se v takovém případě dotazovali na znak na pozici -1. Z tohoto důvodu máme na řádce 16 podmínku, která přesně této situaci předchází. Pokud je tedy reprezentace vzoru již zarovnána s m nejlevějšími znaky reprezentace prohledávaného stromu, podmínka na řádce 16 zajistí, že algoritmus již nebude dále pokračovat.

Podívejme se nyní na příklad 14 společně s tabulkou 2.3. Ty demonstrují běh obrácené varianty algoritmu pro konkrétní strom a vzor. Záměrně jsme zvolili stejný vzor a strom jako v příkladu 12 v části 2.3 pro demonstraci běhu standardní varianty algoritmu.

Příklad 14. Mějme stromový vzor p s řetězcovou reprezentací v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(p) = a2 \ * \ \uparrow * \ * \uparrow * \ \uparrow 2$ nad abecedou $\mathcal{A}_1 = \{a2, a1, a0, *, \uparrow 2, \uparrow 1, \uparrow 0, \uparrow *\}$ a strom t s řetězcovou reprezentací v prefixové ohodnocené zarážkové notaci $pref\_ranked\_bar(t) = a2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ a2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2\ \uparrow 2$ nad abecedou $\mathcal{A}_2 = \{a2, a1, a0, \uparrow 2, \uparrow 1, \uparrow 0\}$. Chceme najít všechny výskyty vzoru p ve stromu t pomocí obrácené varianty algoritmu Quick Search pro hledání stromových vzorů ve stromech. Pro vzor p dostaneme následující hodnoty v tabulce posunů (algoritmem 8): $BCS[a2] = 1$, $BCS[a1] = 2$, $BCS[a0] = 2$, $BCS[\uparrow 2] = 6$, $BCS[\uparrow 1] = 5$, $BCS[\uparrow 0] = 3$. Tabulka 2.3 ukazuje běh algoritmu 9 nad vzorem p a prohledávaným stromem t. Reprezentace delších podstromů dosazených na místo zástupných podřetězců $* \uparrow *$ jsou v tabulce naznačeny jako $*\rightarrow \leftarrow *$.

Běh algoritmu 9 pro strom a vzor z příkladu 14 začíná kontrolou, zda existuje výskyt vzoru začínající na pozici 8 ($n - m$, n je pro tento prohledávaný strom 14 a m je pro tento vzor 6). Protože $a0 \neq a2$, hned první porovnávaný znak se neshoduje a algoritmus se posune na další možnou pozici výskytu. To učiní na základě znaku na pozici 7 ($i - 1$), kde se nachází znak $a2$, pro který

algoritmus nalezne v tabulce posunů hodnotu 1, tedy se provede posun pouze o 1 znak doleva. Poté se tedy zkontroluje existence výskytu vzoru začínající na pozici 7. Tam už výskyt skutečně existuje, tedy je přidán do výstupu a na základě znaku na pozici 6 se učiní další posun. Na pozici 6 se nachází znak $\uparrow 2$, na jehož základě algoritmus učiní posun doleva o velikosti 6. V následujícím kroku je zkontrolována existence výskytu začínajícího na pozici 1, kde opět výskyt existuje, takže je opět přidán do výstupu. Dále algoritmus učiní další posun na základě znaku na pozici 0, tam se nachází znak $a2$, tedy se učiní posun o velikosti 1. Po tomto posunu je zkontrolována existence výskytu začínajícího na pozici 0, tam výskyt opět existuje, takže je přidán do výstupu, a protože v tuto chvíli má algoritmus již zarovnanou reprezentaci vzoru s m nejlevějšími znaky reprezentace prohledávaného stromu, podmínka na řádce 16 algoritmu 9 se vyhodnotí jako pravda a algoritmus skončí. V tabulce 2.3 můžeme vidět, jaké podstromy byly v kterém kroku dosazeny na místo zástupného podřetězce $* \uparrow *$.

Pojďme se nyní krátce zamyslet nad tím, v čem by skenování reprezentace vzoru zleva doprava mohlo být výhodnější, než jeho skenování zprava doleva. Představme si, že máme nějaký strom t a jeho řetězcová reprezentace obsahuje tři následující podřetězce: $a2\ a0\ \uparrow 0\ d0\ \uparrow 0\ \uparrow 2$, $b2\ a0\ \uparrow 0\ a0\ \uparrow 0\ \uparrow 2$ a $c2\ d0\ \uparrow 0\ d0\ \uparrow 0\ \uparrow 2$. Dále si představme vzor p, který nebude obsahovat žádný výskyt zástupného symbolu $*$ a jeho řetězcová reprezentace bude následující: $d2\ c0\ \uparrow 0\ b0\ \uparrow 0\ \uparrow 2$. Představme si nyní, že máme standardní variantu našeho algoritmu a chceme ověřit existenci vzoru na třech konkrétních místech ve stromu tak, že na každém z těchto míst končí jeden ze tří zmíněných podřetězců. Při skenování vzoru zprava doleva tedy začneme porovnávat znaky od konce reprezentace vzoru. Všimněme si, že pro všechny tři podřetězce se dva nejpravější znaky rovnají dvěma nejpravějším znaků reprezentace našeho vzoru. Tedy tyto dva znaky nám nevyvrátí existenci výskytu vzoru na dané pozici a algoritmus musí pro každý z tří zmíněných podřetězců ověřit tři znaky, než nalezne neshodu. Pokud budeme používat obrácenou variantu algoritmu, ve všech třech případech narazíme na neshodu hned při porovnávání prvního znaku. Zde se jedná samozřejmě o konkrétní případ, který nelze nějak zobecňovat. Je ale dobré si uvědomit, že zarážkových znaků budeme mít typicky v naší abecedě méně, než těch nezarážkových. Pokud totiž budeme mít několik nezarážkových znaků se stejnou aritou, pro všechny nám bude stačit jeden zarážkový znak stejné arity. Zjednodušeně řečeno, když při porovnávání odzadu narazíme na znak $\uparrow 2$, nemáme jednoduchý způsob jak zjistit, zda tento znak slouží jako zarážka pro znak $a2$, $b2$ nebo třeba $c2$. Zdá se tedy, že při skenování vzoru zleva doprava budeme neshody identifikovat typicky dříve, protože u nezarážkových znaků bude docházet k neshodě v průměru častěji, než u znaků zarážkových.

2.6 Rozšíření obrácené varianty algoritmu na prefixovou ohodnocenou notaci

Doposud jsme pracovali pouze s prefixovou ohodnocenou zarážkovou notací. Obrácená varianta algoritmu je ale vhodná i pro použití s prefixovou ohodnocenou notací. Celkem očividnou výhodou prefixové ohodnocené notace v porovnání s prefixovou ohodnocenou zarážkovou notací je fakt, že řetězcová reprezentace stromu v prefixové ohodnocené notaci bude mít poloviční délku oproti reprezentaci v druhé zmíněné notaci. Poloviční délka reprezentace pro nás znamená, že budeme muset prozkoumat a porovnat méně znaků a algoritmus schopný pracovat s prefixovou ohodnocenou notací by díky tomu mohl být rychlejší. Pojďme si tedy představit variantu algoritmu, která pracuje s prefixovou ohodnocenou notací tak, jak byla definována v části 1.4.

Nejprve si opět musíme říci, jak zkonstruovat tabulku posunů pro tuto variantu algoritmu. Tento proces se nám mírně zjednoduší, protože v prefixové ohodnocené notaci nepracujeme s žádnými zarážkovými znaky. Hodnoty v tabulce posunů pro tuto variantu algoritmu opět získáme jako minimum ze tří hodnot:

1. $m + 1$
2. $j : vzor[j - 1] = x$ a platí $m \geq j > 0$
3. $j : vzor[j - 1] = *$ a platí $m \geq j > 0$

První bod slouží podobně jako u předchozích variant algoritmu jako omezení posunu shora délkou vzoru plus jedna. Zde je ale třeba si uvědomit, že pro stejný vzor bude délka jeho řetězcové reprezentace v prefixové ohodnocené notaci poloviční oproti délce jeho řetězcové reprezentace v prefixové ohodnocené zarážkové notaci.

Druhý bod opět určuje minimální bezpečnou vzdálenost posunu pro znaky, které se přímo vyskytují v reprezentaci vzoru. Podobně jako u obrácené varianty algoritmu nad prefixovou ohodnocenou zarážkovou notací, i tady chceme zarovnat nejlevější výskyt znaku x s jeho výskytem v prohledávaném stromě na pozici $i-1$. I v tomto případě předpokládáme, že na místo zástupného symbolu $*$ by byl dosazen nejmenší možný podstrom, ovšem tento nejmenší možný podstrom bude v tomto případě mít velikost pouze 1, například podstrom s řetězcovou reprezentací $a0$. Nejmenší možný posun získaný tímto bodem je 1, což bude velikost posunu pro nejlevější znak z reprezentace našeho vzoru.

Třetí bod řeší případ, kdy se znak vyskytuje v řetězcové reprezentaci podstromu, který je dosazen na místo zástupného symbolu $*$ v daném vzoru. Zde je ovšem naše situace jednoduší, než v případě prefixové ohodnocené zarážkové notace. Zde totiž bude hodnota pro všechny znaky stejná a bude odpovídat vzdálenosti nejlevějšího výskytu zástupného znaku $*$ plus jedna. Na místě nejlevějšího výskytu zástupného znaku $*$ v řetězcové reprezentaci vzoru se totiž

po dosazení může nacházet skutečně jakýkoliv znak z abecedy prohledávaného stromu.

Na první pohled je vidět, že algoritmus pro konstrukci tabulky posunů pro obrácenou variantu algoritmu nad prefixovou ohodnocenou notací bude jednoduší než algoritmy pro předchozí dvě varianty, neboť nemusíme řešit žádné zarážkové znaky. V příkladu 15 ukazujeme, jak bude vypadat tabulka posunů pro obrácenou variantu algoritmu nad prefixovou ohodnocenou notací pro konkrétní vzor. Opět jsme úmyslně použili stejný vzor, jako v příkladech 11 v části 2.2 a 13 v části 2.5. V tabulce nepotřebujeme hodnotu posunu pro zástupný znak $*$, protože ten se nikdy nebude vyskytovat v řetězové reprezentaci prohledávaného stromu. Podotýkáme, že abeceda v příkladu 15 je jiná než v příkladech v předchozích částech, protože používáme notaci, která nepotřebuje zarážkové znaky. Tabulku posunů pro tuto variantu algoritmu lze sestrojit například algoritmem 10.

Název algoritmu: Konstrukce BCS pro obrácenou variantu algoritmu Quick Search nad stromy při využití prefixové ohodnocené notace.
Vstup: Stromový vzor *pattern* v prefixové ohodnocené notaci $pref\_ranked(pattern)$ o velikosti m nad abecedou $\mathcal{A}$ prohledávaného stromu *subject*.
Výstup: Tabulka posunů $BCS(pref\_ranked(pattern))$.

```
1  begin
2  |  s := m + 1
3  |  for i := m − 1 to 0 do
4  |  |  if pref_ranked(pattern)[i] = ↑∗ then s = i + 1;
5  |  end
6  |  foreach x ∈ A do BCS[x] = m + 1;
7  |  foreach x ∈ A do
8  |  |  if BCS[x] > s + 1 then BCS[x] := S + 1;
9  |  end
10 |  for i := m to 1 do
11 |  |  if pref_ranked(pattern)[i − 1] ∉ {∗} and
   |  |  BCS[pref_ranked(pattern)[i − 1]] > i then
   |  |  BCS[pref_ranked(pattern)[i − 1]] := i ;
12 |  end
13 end
```

Algoritmus 10: Konstrukce tabulky posunů pro obrácenou variantu algoritmu Quick Search pro hledání stromových vzorů při využití prefixové ohodnocené notace

Příklad 15. Mějme stromový vzor p s následující řetězcovou reprezentací v prefixové ohodnocené notaci $pref\_ranked\_bar(p) = a2\ a1 * a1\ a0$ nad abecedou $\mathcal{A} = \{a3, a2, a1, a0, *\}$.

Pak hodnoty v tabulce posunů (BCS) pro obrácenou variantu algoritmu nad prefixovou ohodnocenou notací pro tento vzor budou následující:
$BCS[a3] = \min(\{6\} \cup \emptyset \cup \{3\}) = 3$, $BCS[a2] = \min(\{6\} \cup \{1\} \cup \{3\}) = 1$,
$BCS[a1] = \min(\{6\} \cup \{2, 4\} \cup \{3\}) = 2$, $BCS[a0] = \min(\{6\} \cup \{5\} \cup \{3\}) = 3$,

Vzor v příkladu 15 je skutečně stejný jako vzor v příkladech 11 a 13, pouze je zapsaný v jiné notaci. Hodnoty v tabulce posunů pro tuto varianta algoritmu se mohou zdát menší než pro předchozí dvě varianty algoritmu, je ale nutné si uvědomit, že i délka reprezentace prohledávaného stromu bude pouze poloviční.

Nyní tedy již umíme sestrojit tabulku posunů. Protože ale zde pracujeme s jinou notací než s prefixovou ohodnocenou zarážkovou notací, nemůžeme již využít algoritmus 4 pro konstrukci podstromové skokové tabulky. Tu budeme muset pro účely této varianty algoritmu konstruovat opět jinak. Využít můžeme algoritmus 11.

Název algoritmu: Konstrukce SJT pro obrácenou variantu algoritmu Quick Search nad prefixovou ohodnocenou notací.
Vstup: Prohledávaný strom t v prefixové ohodnocené notaci $pref\_ranked(t)$ o délce n, index aktuálního vrcholu $root$, což je implicitně 0 (index kořene) a reference na prázdnou podstromovou skokovou abulku $SJT(pref\_ranked(t))$ délky n
Výstup: index $koncovyIndex$, podstromová skoková tabulka $SJT(pref\_ranked(t))$

```
1 begin
2   index := root + 1
3   for i = 1 to Arity(pref_ranked(t)[root]) do
4     index :=
        ConstructSJT(pref_ranked(t), index, SJT(pref_ranked(t)))
5   end
6   SJT(pref_ranked_bar(t))[root] = index
7   return index
8 end
```

Algoritmus 11: Konstrukce podstromové skokové tabulky pro obrácenou variantu algoritmu Quick Search nad prefixovou ohodnocenou notací pro hledání stromových vzorů ve stromech

Jak bude vypadat podstromová skoková tabulka pro konkrétní strom v prefixové ohodnocené notaci ukazuje příklad 16 společně s tabulkou 2.4. Strom v příkladu 16 je záměrně stejný jako v příkladu 6, pouze je zapsaný v jiné

notaci.

Příklad 16. Mějme následující strom t, respektive jeho řetězcovou reprezentaci v prefixové ohodnocené notaci $pref\_ranked\ (t) = a2\ a2\ a0\ a0\ a2\ a0\ a0$ nad abecedou $\mathcal{A} = \{a3, a2, a1, a0\}$. Tabulka 2.4 ukazuje, jak vypadá hotová podstromová skoková tabulka $SJT(pref\_ranked(t))$.

Tabulka 2.4: Podstromová skoková tabulka $SJT(pref\_ranked(t))$ pro strom z příkladu 16.

0	1	2	3	4	5	6
$a2$	$a2$	$a0$	$a0$	$a2$	$a0$	$a0$
7	4	3	4	7	6	7

Nyní už tedy umíme sestrojit jak tabulku posunů tak podstromovou skokovou tabulku i pro variantu algoritmu pracující s prefixovou ohodnocenou notací. Můžeme tedy představit pseudokód pro samotnou variantu algoritmu.

Algoritmus 12 je velmi podobný algoritmu 9 (tedy obrácené variantě prezentovaného algoritmu pracující nad prefixovou ohodnocenou zarážkovou notací). Řádky 4 až 13 jsou zodpovědné za kontrolu existence výskytu začínajícího na aktuální pozici i. Za zmínku stojí, že v případě, že narazíme ve vzoru na výskyt zástupného znaku $*$, nemusíme se v reprezentaci vzoru posouvat o dva znaky, jako jsme to dělali u předchozích dvou variant algoritmu, ale stačí nám pouze posun ve vzoru o jednu pozici doprava. To je způsobeno tím, že tentokrát nepracujeme s podřetězcem $* \uparrow *$, na jehož místo bychom dosazovali reprezentaci podstromu, ale pracujeme pouze s podřetězcem $*$ délky jedna, na jehož místo dosazujeme případný podstrom.

Na řádce 14 standardně přidáme případný nalezený výskyt do výstupu. Na řádce 15 algoritmus skončí, pokud už máme řetězcovou reprezentaci vzoru zarovnanou s m nejlevějšími znaky řetězcové reprezentace prohledávaného stromu, abychom se na řádce 16 nedotazovali na znak na pozici -1. Pokud algoritmus není ukončen podmínkou na řádce 15, pak je reprezentace vzoru posunuta doleva v reprezentaci prohledávaného stromu na řádce 16 na základě hodnoty v tabulce posunů pro znak na aktuální pozici $i - 1$. Poté algoritmus pokračuje kontrolou existence výskytu na další možné pozici.

Pojďme se nyní podívat na příklad 17 a tabulku 2.5, které společně ukazují běh algoritmu 12 pro konkrétní strom a vzor. Záměrně jsme i zde zvolili stejný vzor a strom jako v příkladech 12 v části 2.3 a 14 v části 2.5, ačkoliv to tak na první pohled nemusí vypadat. I v tomto případě se ale jedná o stejný vzor a strom jako v předchozích dvou zmíněných příkladech, jenom používáme jinou řetězcovou notaci.

Název algoritmu: Obrácená varianta algoritmu Quick Search nad stromy s použitím prefixové ohodnocené notace.
Vstup: Řetězcová reprezentace stromu *subject* v $pref\_ranked(subject)$ notaci délky n, řetězcová reprezentace stromového vzoru *pattern* v $pref\_ranked(pattern)$ notaci délky m, $SJT(pref\_ranked(subject))$ a $BCS(pref\_ranked(pattern))$.
Výstup: Všechny výskyty stromového vzoru *pattern* ve stromu *subject*.

```
1  begin
2      i := n − m
3      while i ≥ 0 do
4          j := 0
5          position := i
6          while j < m and position < n do
7              if pref_ranked(subject)[position] = pref_ranked(pattern)[j]
                then
8                  position := position + 1
9              else if pref_ranked(pattern)[j] = ∗ and
                pref_ranked(subject)[position] ∈ A↑ then
10                 position := SJT(pref_ranked(subject))[position]
                    {Subtree skip}
11             else break;
12             j := j + 1
13         end
14         if j = m then output(i);
15         if i = 0 then break;
16         i := i − BCS[pref_ranked(subject)[i − 1]]
17     end
18 end
```

Algoritmus 12: Obrácená varianta algoritmu Quick Search nad prefixovou ohodnocenou notací pro hledání stromových vzorů ve stromech

Tabulka 2.5: Běh algoritmu 12 nad stromem t a vzorem p z příkladu 17

0	1	2	3	4	5	6	
$a2$	$a2$	$a0$	$a0$	$a2$	$a0$	$a0$	$pref\_ranked(t)$
7	4	3	4	7	6	7	$SJT(t)$
				$a2$	$*$	$*$	nález, $posun = 2$
		$a2$					$a0 \neq a2$, $posun = 1$
	$a2$	$*$	$*$				nález, $posun = 1$
$a2$	$*\rightarrow$		$\leftarrow*$	$*\rightarrow$		$\leftarrow*$	nález, $break$

Příklad 17. Mějme stromový vzor p s řetězcovou reprezentací v prefixové ohodnocené notaci $pref\_ranked(p) = a2 * *$ nad abecedou $\mathcal{A}_1 = \{a2, a1, a0, *\}$ a strom t s reprezentací v prefixové ohodnocené notaci $pref\_ranked(t) = a2\ a2\ a0\ a0\ a2\ a0\ a0$ nad abecedou $\mathcal{A}_2 = \{a2, a1, a0\}$. Chceme najít všechny výskyty vzoru p ve stromu t pomocí obrácené varianty algoritmu Quick Search nad prefixovou ohodnocenou notací pro hledání stromových vzorů ve stromech. Pro vzor p dostaneme následující hodnoty v tabulce posunů (algoritmem 10): $BCS[a2] = 1$, $BCS[a1] = 2$, $BCS[a0] = 2$. Tabulka 2.5 ukazuje běh algoritmu 12 nad vzorem p a prohledávaným stromem t. Reprezentace delších podstromů dosazených na místo zástupných znaků $*$ jsou v tabulce naznačeny jako $*\rightarrow\ \leftarrow*$.

Běh algoritmu 12 pro strom a vzor z příkladu 17 začíná kontrolou, zda existuje výskyt vzoru na pozici 4 ($n - m$, kde n je v tu chvíli 7 a m je 3) v řetězcové reprezentaci prohledávaného stromu. Tam je ihned výskyt nalezen, takže je přidán do výstupu a na základě znaku na pozici 3 ($i - 1$) je učiněn posun o velikosti 2 doleva v reprezentaci prohledávaného stromu (na pozici 3 se nachází znak $a0$, pro který máme v tabulce posunů hodnotu 2). Následně algoritmus kontroluje výskyt vzoru na pozici 2. Zde se ale neshoduje hned první porovnávaný znak, protože $a0 \neq a2$, tedy algoritmus zjistí, že na pozici 2 není výskyt vzoru a podle znaku na pozici 1 se posune o 1 znak doleva a zkontroluje pro změnu výskyt vzoru na pozici 1. Tam už se výskyt nachází, takže je tento též algoritmem přidán do výstupu, a následně algoritmus učiní další posun doleva, tentokrát podle znaku na pozici 0. Tam se nachází znak $a2$, pro který je v tabulce posunů hodnota 1, tedy algoritmus učiní posun o 1 znak doleva na pozici 0. Pokračuje ověřením výskytu vzoru na této pozici, kde opět výskyt nalezne, takže ho přidá do výstupu, a protože v tu chvíli už máme řetězcovou reprezentaci vzoru zarovnanou s m nejlevějšími znaky řetězcové reprezentace prohledávaného stromu, algoritmus skončí.

2.7 Shrnutí variant navrženého algoritmu

V celé části 2 jsme se věnovali představení hned několika variant algoritmu. Pro každou z těchto variant umíme sestrojit tabulku posunů (BCS) a podstromovou skokovou tabulku (SJT), tedy obě potřebné struktury, a pro každou z těchto variant algoritmu jsme představili pseudokód a ukázali jsme běh dané varianty algoritmu na příkladu konkrétního vzoru a prohledávaného stromu.

Pojďme si nyní ještě stručně připomenout všechny tři prezentované varianty algoritmu:

1. **Standardní variantou algoritmu** jsme se zabývali v částech 2.2 a 2.3. Tato varianta používá *prefixovou ohodnocenou zarážkovou notaci* jak pro prohledávaný strom, tak pro hledaný vzor. Reprezentace prohledávaného stromu je procházena zleva doprava, při kontrole výskytu vzoru na konkrétní pozici je pak reprezentace vzoru skenována zprava doleva. Tabulku posunů pro tuto variantu algoritmu můžeme zkonstruovat algoritmem 6, podstromovou skokovou tabulku algoritmem 4, samotný pseudokód této varianty algoritmu si pak můžeme prohlednout v rámci algoritmu 7.

2. **Obrácenou variantu algoritmu** jsme představili v části 2.5. Tato varianta též používá *prefixovou ohodnocenou zarážkovou notaci* jak pro prohledávaný strom, tak pro hledaný vzor. Reprezentace prohledávaného stromu je ale procházena zprava doleva a při kontrole výskytu vzoru na konkrétní pozici je reprezentace vzoru skenována zleva doprava. Tabulku posunů pro tuto variantu algoritmu lze zkonstruovat za pomoci algoritmu 8, podstromovou skokovou tabulku algoritmem 4, samotný pseudokód obrácené varianty algoritmu pak můžeme vidět v rámci algoritmu 9.

3. A konečně **obrácenou variantu algoritmu nad prefixovou ohodnocenou notací** jsme představili v části 2.6. Tato varianta používá *prefixovou ohodnocenou notaci* jak pro prohledávaný strom, tak pro hledaný vzor. Stejně jako obrácená varianta algoritmu nad prefixovou ohodnocenou zarážkovou notací, i tato varianta algoritmu prochází reprezentaci prohledávaného stromu zprava doleva a při kontrole výskytu vzoru na konkrétní pozici je reprezentace vzoru skenována zleva doprava. Tabulku posunů pro tuto variantu můžeme zkonstruovat algoritmem 10, podstromovou skokovou tabulku algoritmem 11, samotný pseudokód algoritmu pak můžeme vidět v rámci algoritmu 12.

V následující části 3 se nejprve podíváme na některé konkrétní implementační specifika zmíněných variant algoritmu v jazyce C++ v projektu *Algoritmová knihovna* a též v jazyce Java pro účely souboru nástrojů Forest fire & Fire wood. Následně všechny tři varianty algoritmu otestujeme, abychom

mohli porovnat jejich rychlost jednak mezi sebou, ale také s ostatními algoritmy, řešícími problém hledání stromových vzorů ve stromech.

Kapitola 3

Implementace a testování

V následující části práce se nejprve zaměříme na několik implementačních detailů prezentovaného algoritmu (a jeho variant) a následně se budeme zabývat testováním implementovaných algoritmů. Nejprve budeme mluvit o implementaci algoritmů v jazyce C++ v rámci projektu *Algoritmová knihovna* a o implementaci v jazyce Java pro účely souboru nástrojů Forest fire & Fire wood. Poté ukážeme, jak byly naimplementované algoritmy zahrnuty do testovacích skriptů Algoritmové knihovny, a nakonec popíšeme, jak byly algoritmy testovány v souboru nástrojů Forest fire & Fire wood a okomentujeme výsledky tohoto testování.

Jak v rámci projektu Algoritmová knihovna tak v rámci zmíněného souboru nástrojů již bylo dříve implementováno několik algoritmů pro práci se stromy i přímo pro problém hledání stromových vzorů ve stromech. Protože se v obou případech jedná o implementaci algoritmů do již existujícího projektu, snažili jsme se v obou případech dodržovat konvence a zvyklosti příslušných projektů. Vzhledem k tomu, že některé algoritmy například pro konstrukci podpůrných struktur již v knihovně existovali (například algoritmus pro konstrukci podstromové skokové tabulky, neboť ten je shodný pro algoritmus protisměrného vyhledávání ve stromech, který je v projektu již implementován, a námi prezentovaný algoritmus), snažili jsme se tyto algoritmy v rámci minimalizace duplikování kódu využít.

3.1 Implementace algoritmů v projektu Algoritmová knihovna

V rámci projektu *Algoritmová knihovna* byly algoritmy implementovány v jazyce C++. Protože se jedná o projekt, který by měl sloužit především studentům pro účely experimentování, případně jako pomocný výukový prostředek, při této implementaci jsme se snažili o co nejpochopitelnější a nejpřímočařejší zápis algoritmů.

Struktury potřebné pro práci se stromy a vzory v prefixové ohodnocené zarážkové notaci a v prefixové ohodnocené notaci již byly v Algoritmové knihovně k dispozici před touto prací. Stejně tak algoritmy pro převod stromů případně stromových vzorů do jejich řetězcové reprezentace v prefixové ohodnocené zařážkové notaci. Algoritmus tedy dostane strom a vzor již připravené v této notaci a vrátí set indexů všech výskytů vzoru v prohledávaném stromu.

V rámci samotné implementace pak tedy vždy nejprve voláme pomocné funkce pro zkonstruování tabulky posunů (BCS) a podstromové skokové tabulky (SJT). Následně začneme s prohledáváním na příslušné pozici podle varianty algoritmu (u standardní začínáme z levého konce řetězcové reprezentace prohledávaného stromu, u obrácené z pravého). O celou kontrolu výskytu vzoru na konkrétní pozici se starají speciální třídy `BackwardOccurrenceTest` a `ForwardOccurrenceTest` svými metodami `occurrence`. Metoda `occurrence` třídy `BackwardOccurrenceTest` dostane index a ověří, zda existuje výskyt vzoru končící na daném indexu, třída `ForwardOccurrenceTest` naopak prostřednictvím svojí metody `occurrence` ověřuje, zda existuje výskyt vzoru začínající daným indexem.

Tyto testy existence výskytu vzoru na konkrétním indexu využívá v Algoritmové knihovně více algoritmů. Pro algoritmy založené na linearizaci stromů je typické, že nějakým způsobem vybírají indexy, na kterých by se mohl výskyt vzoru nacházet, a pak tyto indexy testují. Podobně funguje například i algoritmus protisměrného vyhledávání ve stromech, ale například i naivní algoritmus, který se akorát dotáže postupně na všechny možné indexy. V knihovně se nacházejí i další algoritmy, které potřebují tento test často dělat, proto dává smysl mít tento test implementován v rámci vlastní třídy, aby ho mohly snadno volat všechny algoritmy, které ho využívají.

Po ověření existence výskytu metodou `occurrence` příslušné třídy se pak případný výskyt přidá do setu výskytů, následně dojde k ověření, zda již nemáme reprezentaci vzoru zarovnanou s m nejkrajnějšími znaky reprezentace prohledávaného stromu (opět záleží na variantě algoritmu zda s m nejpravějšími nebo m nejlevějšími), pokud ano, algoritmus skončí a vrátí set výskytů (který může být v tu chvíli prázdný), pokud ne, pak se provede posun na základě tabulky posunů a algoritmus pokračuje kontrolou existence výskytu vzoru na další možné pozici.

3.2 Specifika souboru nástrojů Forest fire & Fire wood

Kromě implementace v jazyce C++ v Algoritmové knihovně byly prezentované algoritmy implementovány i v jazyce Java v rámci souboru nástrojů Forest fire & Fire wood. Protože tyto implementace později sloužily pro testování rychlosti algoritmu, aby bylo možné porovnat jednotlivé varianty algoritmu mezi sebou, ale také s jinými algoritmy pro řešení stejného problému,

snažili jsme se v rámci tohoto souboru nástrojů o co nejvíce efektivní implementaci algoritmu, tedy se bylo potřeba zbavit i podmínky na testování, zda už je řetězcová reprezentace vzoru zarovnána s m nejlevějšími (respektive nejpravějšími pro obrácenou variantu algoritmu) znaky řetězcové reprezentace prohledávaného stromu. O této úpravě se konkrétně zmíníme v následující části 3.3.

Nyní si pojďme stručně shrnout specifika souboru nástrojů Forest fire & Fire wood. Kromě toho, že je tento soubor nástrojů implementovaný v Javě, se od Algoritmové knihovny liší ještě dalšími věcmi, které bylo třeba vzít v potaz. Jedním z hlavních rozdílů je fakt, že veškeré již implementované algoritmy a struktury v rámci tohoto souboru nástrojů indexují od jedničky. To znamená, že zatímco doposud jsme se pro řetězcovou reprezentaci vzoru délky m mohli dotazovat na znaky 0 až $m - 1$, v rámci souboru nástrojů Forest fire & Fire wood se budeme muset prozměnu dotazovat na znaky 1 až m. Zmíněný soubor nástrojů navíc pro účely měření efektivity algoritmů při jejich průběhu sleduje i některé další statistiky, což se projeví i na výsledném kódu. Například se sledují `attempts` (pokusy), což je počet indexů, pro které se algoritmus pokusil ověřit existenci výskytu vzoru. Algoritmy, které učiní menší počet `attempts` jsou teoreticky lepší v tom smyslu, že testují méně zbytečných indexů (samozřejmě za předpokladu že fungují korektně a naleznou všechny výskyty vzoru v prohledávaném stromu). Algoritmus ovšem může vyzkoušet méně indexů, ale strávit kontrolou každého indexu více času, tedy se sleduje i skutečný čas, jak dlouho algoritmus běžel pro konkrétní vzor a prohledávaný strom.

3.3 Implementace algoritmů v souboru nástrojů Forest fire & Fire wood a odstranění breaku před posunem indexu

Pojďme nejprve nastínit způsob, kterým se lze zbavit kontroly, zda už je řetězcová reprezentace vzoru zarovnána s m nejpravějšími znaky řetězcové reprezentace prohledávaného stromu. Nejprve ukážeme, jak se této kontroly zbavit u standardní varianty algoritmu. V algoritmu 7 se tato kontrola provádí na řádce 16 formou podmínky, která se pro všechny iterace kromě té poslední vyhodnotí jako nepravda. Tato podmínka je sice relativně dobře pochopitelná, má ovšem negativní vliv na rychlost výsledného algoritmu.

Odstranění této podmínky je naštěstí velmi jednoduché. Ukázka kódu 1 ukazuje začátek implementace algoritmu, kde je na konec řetězcové reprezentace vzoru přidán znak tak, že podmínka na řádce 16 algoritmu 7 už není dále potřeba.

Zde je potřeba si uvědomit několik věcí. Hlavní myšlenkou je zde fakt, že do reprezentace stromu přidáme znak na pozici $n + 1$, přitom ale nezvýšíme n. Tím docílíme toho, že v situaci, kdy budeme mít řetězcovou reprezentaci

```
int n = bt.size();
int m = patternTree.size();
int i = 0;

subject.add(subject.get(1));
```

Ukázka kódu 1: Nahrazení podmínky znakem přidaným na konec řetězcové reprezentace prohledávaného stromu v souboru nástrojů Forest fire & Fire wood

hledaného vzoru již zarovnanou s m nejpravějšími znaky reprezentace prohledávaného stromu, tak se stále budeme moci bezpečně dotázat na znak na pozici $i+m+1$, protože se tam bude nyní nacházet námi přidaný znak. V dalším kroku se ovšem při porovnávání podmínky $i \leq n-m$ v hlavičce while cyklu vyhodnotí tato podmínka jako nepravda, protože už v předchozím kroce muselo platit, že $i = n - m$, a my jsme zvýšili hodnotu i minimálně o jedna (protože hodnoty v tabulce posunů jsou pro všechny znaky alespoň jedna). Tedy další iterace while cyklu už se neprovede. Stejný postup bychom mohli zvolit v implementaci algoritmu v Algoritmové knihovně, pouze bychom tam přidávali libovolný znak na pozici n místo pozice $n + 1$, neboť bychom indexovali od nuly. Varianta s podmínkou a breakem nám ale přišla přímočařejší a pochopitelnější.

Kromě tohoto odstranění zmíněné podmínky je implementace v jazyce Java v souboru nástrojů Forest fire & Fire wood velice podobná implementaci v Algoritmové knihovně, až na zmiňované indexování od jedničky, a samozřejmě různé pomocné struktury v rámci zmiňovaného souboru nástrojů se jmenují (případně i fungují) trochu jinak než ty v rámci Algoritmové knihovny. I v této implementaci ale začneme zkonstruováním tabulky posunů a podstromové skokové tabulky pomocnými funkcemi, následně procházíme řetězcovou reprezentaci zleva doprava (v případě standardní varianty), a v každé iteraci vnějšího while cyklu ověříme existenci výskytu vzoru na jednom konkrétním indexu. Nalezené výskyty (respektive jejich počáteční indexy) opět přidáváme do výsledného setu, který na konci běhu vrátíme. Posuny zde děláme vždy na základě znaku na aktuální pozici $i+m+1$, respektive na základě hodnoty pro znak na dané pozici v tabulce posunů.

U obrácené varianty algoritmu můžeme pro odstranění podmínky na kontrolu zarovnání řetězcové reprezentace hledaného vzoru s m nejlevějšími znaky řetězcové reprezentace prohledávaného stromu použít podobné řešení, jako u standardní varianty algoritmu. Protože zde ale procházíme reprezentaci prohledávaného stromu zprava doleva, nepřidáváme znak na konec této reprezentace, ale na začátek. Zde nám přišel vhod fakt, že v souboru nástrojů Forest fire & Fire wood se indexuje od jedničky, tedy můžeme snadno znak

přidat na pozici 0. V případě, že bychom indexovali od nuly jako v případě implementace v projektu Algoritmová knihovna, situace by se trochu zkomplikovala, neboť tam bychom znak na začátek museli přidat na pozici -1, což ve většině jazyků nebude fungovat tak, jak bychom potřebovali. V případě potřeby implementace prezentovaného algoritmu s důrazem na efektivitu tedy rovnou doporučujeme zvolit indexaci od jedničky, při které se dá využít řešení s pouhým přidáním jednoho znaku do reprezentace prohledávaného stromu. Pokud bychom z nějakého důvodu přece jenom chtěli indexovat od nuly, dalším způsobem, jak se zmiňované podmínky zbavit, by bylo změnit podmínku ve vnějším while cyklu z $i \geq 0$ (tuto podmínku bychom ve vnějším while cyklu měli při indexování od nuly) na $i > 0$ a poté za konec tohoto while cyklu natvrdo přidat ještě jednu iteraci, v rámci které by se ověřilo, zda náhodou není výskyt začínající na nejlevější pozici řetězcové reprezentace prohledávaného stromu s tím, že po této kontrole by se už samozřejmě neprováděl další posun.

Efektivitě by toto druhé navrhované řešení zajisté pomohlo, neboť by odpadla kontrola podmínky v každé iteraci algoritmu (která by se vyhodnotila vždy jako nepravda s výjimkou poslední iterace). Zaplatíme ale tím, že budeme muset přidat navíc nemalý počet řádek kódu, které navíc budou stejné jako řádky uvnitř vnějšího while cyklu, takže navíc budeme mít pak v implementaci duplicitní kód, čemuž bychom se rádi vyhnuli. Platí ale, že v případě, že bychom z nějakých blíže nespecifikovaných důvodů museli indexovat od nuly a zároveň bychom chtěli co nejefektivněji implementovat prezentovaný algoritmus, pak by bylo možné výše zmiňovanou podmínku odstranit i tímto druhým způsobem.

Implementace obou obrácených variant prezentovaného algoritmu v rámci souboru nástrojů Forest fire & Fire wood už jsou pak kromě zmíněného odstranění podmínky opět velmi podobné jejich implementacím v Algoritmové knihovně.

3.4 Souhrn implementovaných algoritmů v projektu Algoritmová knihovna a příklady použití

Pojďme si nyní shrnout algoritmy, které díky této práci přibily v projektu Algoritmová knihovna, a již nyní je uživatelé mohou používat, v případě že si stáhnou a zkompilují nejnovější verzi projektu.

Začněme implementacemi samotné adaptace algoritmu Quick Search pro hledání stromových vzorů ve stromech. Pro tento problém byly v rámci Algoritmové knihovny implementovány tři algoritmy, standardní varianta prezentovaného algoritmu, obrácená varianta prezentovaného algoritmu, a obrácená varianta nad prefixovou ohodnocenou notací. Tyto algoritmy byly zařazeny do jmenného prostoru `arbology` a dají se zavolat buď prostřednictvím programu `aarbology2` kde se dají tyto implementace zavolat jako `quickSearch` případně

pro obrácenou variantu algoritmu `reversedQuickSearch`. Zde podotýkáme, že v rámci Algoritmové knihovny se obrácená varianta algoritmu nad prefixovou ohodnocenou zarážkovou notací a nad prefixovou ohodnocenou notací volají stejně, v projektu funguje přetěžování funkcí, tedy o tom, která varianta algoritmu se použije rozhoduje dodaný vzor a strom. Kromě volání přímo z programu `aarbology2` se implementace prezentovaného algoritmu dají použít též v rámci konzolového rozhraní Algoritmové knihovny (CLI), tedy program `aql2`. V rámci tohoto programu lze spustit libovolný algoritmus z knihovny, proto je zde při volání potřeba specifikovat i konkrétní jmenný prostor. Pokud bychom totiž například volali algoritmus `QuickSearch`, není jasné, zda myslíme implementaci algoritmu Quick Search pro hledání podřetězců v řetězcích, nebo implementaci adaptace algoritmu Quick Search pro hledání stromových vzorů ve stromech. Například standardní variantu prezentovaného algoritmu pro hledání stromových vzorů ve stromech je tedy z tohoto konzolového rozhraní možné volat příkazem `execute arbology::exact::QuickSearch`, za kterým by následoval prohledávaný strom a hledaný vzor v nějaké z reprezentací vhodných pro Algoritmovou knihovnu.

Kromě samotných algoritmů pro hledání stromových vzorů ve stromech se v rámci knihovny dají zavolat samostatně i pomocné funkce, které naše algoritmy využívají. Uživatel si tak může zavolat například algoritmus pro zkonstruování tabulky posunů a jako vstup mu předložit konkrétní vzor. Díky tomu uživatel může snadno zjistit, jaké hodnoty budou v tabulce posunů pro konkrétní vzor, a poté si třeba odsimulovat běh algoritmu na papíru. To by se mohlo hodit například studentům pro účely lepšího pochopení daného algoritmu.

Protože před touto prací v rámci projektu Algoritmová knihovna neexistovala implementace algoritmu Quick Search pro hledání podřetězců v řetězcích, byl tento algoritmus v rámci této práce též implementován. Protože v tomto případě se jedná o algoritmus pracující s řetězci a ne se stromy, byl tento algoritmus zařazen do jmenného prostoru `stringology` a dá se volat buď jako `quickSearch` prostřednictvím programu `astringology2`, nebo opět z konzolového rozhraní v rámci programu `aql2`, tam lze algoritmus volat příkazem `execute stringology::exact::QuickSearch`. Algoritmus pro konstrukci tabulky posunů lze opět volat samostatně.

Dohromady tedy v rámci Algoritmové knihovny vzniklo šest nových algoritmů, které je možné používat. Některé z těchto algoritmů mají navíc více variant, respektive jsou přetížené pro různé varianty vstupů. Například obrácená varianta adaptace algoritmu Quick Search pro hledání stromových vzorů ve stromech umí pracovat jak s prefixovou ohodnocenou zarážkovou notací, tak s prefixovou ohodnocenou notací. V rámci knihovny byla navíc implementována i varianta algoritmu, která požaduje, aby za všechny výskyty zástupného znaku $*$ v hledaném vzoru byl dosazen stejný podstrom. Tedy na místo zástupného znaku $*$ může být stále dosazen libovolný podstrom, ale pokud je v našem vzoru více výskytů zástupného znaku $*$, pak za všechny výskyty musí být do-

sazen stejný podstrom. Tuto variantu algoritmu dává smysl implementovat, protože je potřeba při některých praktických použitích zmiňovaných v úvodu práce. V tomto případě se ovšem mění pouze proces kontroly výskytu vzoru na konkrétním indexu, kde se navíc ještě musí ověřovat rovnost dosazených podstromů.

3.5 Testování v rámci projektu Algoritmová knihovna

Nyní se tedy dostáváme k testování implementovaných algoritmů. Nejrpve popíšeme, jak byly algoritmy zahrnuty do testovacích skriptů v projektu Algoritmová knihovna.

Implementaci v rámci algoritmové knihovny si může kdokoliv snadno vyzkoušet ručně. Stačí ve zkompilované Algoritmové knihovně spustit program `aql2` a poté požadovaný algoritmus zavolat příkazem `execute` s příslušnými vstupy.

Implementované algoritmy bylo ale samozřejmě potřeba podrobit rozsáhlejšímu testování, mimo jiné aby bylo možné odhalit případné chyby v implementaci, ale také aby bylo možné v budoucnu otestovat, že případný refaktoring struktur, se kterými algoritmy pracují, nerozbil jejich funkčnost.

Algoritmová knihovna má pro tyto účely k dispozici testovací skripty, které vždy otestují nějakou sadu algoritmů. Každý z těchto skriptů se snaží testovat nějakou oddělenou část knihovny. Existuje tedy například skript pro testování algoritmů pro práci s řetězci, pro testování algoritmů pro determinizaci automatů, a samozřejmě i pro algoritmy, které pracují se stromy. O ty se stará testovací skript `tests.aarbology.sh`.

Testování probíhá velice jednoduše. Pro každý algoritmus uvedený v tomto skriptu se nejprve provede několik předem definovaných testů, kde se testuje hledání předem definovaných vzorů v předem definovaných stromech (v případě našich algoritmů jsou takové testy 3), poté se pokračuje 100 náhodnými testy, kdy je pokaždé vygenerován nový strom a vzor. Ve skriptu je předem definovaná velikost generovaných stromů a vzorů, jejich hloubka a velikost použité abecedy. Všechny tyto hodnoty si ale uživatel může snadno upravit pro jeho potřeby. Pro generování náhodných stromů a vzorů se používá program `arand2` v rámci Algoritmové knihovny. Generování náhodných stromů pro účely Algoritmové knihovny implementoval v rámci své závěrečné práce *Štěpán Plachý* [13].

Nad každou dvojicí náhodně vygenerovaného vzoru a stromu se pak spustí ve skriptu zadaný algoritmus (takže například některá varianta námi prezentovaného algoritmu) a dále též naivní algoritmus, který hledá výskyty tím způsobem, že zkusí ověřit existenci výskytu postupně na všech indexech reprezentace prohledávaného stromu. Následně se porovná výstup obou těchto algoritmů. Pokud algoritmus zadaný ve skriptu (tedy například náš algorit-

mus) nalezne ve stromu jiný počet výskytů vzoru než naivní algoritmus, skript nás o tom informuje a zároveň do logovacího souboru vypíše jak vzor, tak strom, pro který test selhal, aby bylo možné algoritmus debugovat s konkrétními daty.

Všechny varianty prezentovaného algoritmu byly zahrnuty do testovacího skriptu `tests.aarbology.sh`. Tento skript byl následně několikrát spuštěn pro různé velikosti vygenerovaných stromů a vzorů a nebyly odhaleny žádné nedostatky dané implementace.

Protože v rámci Algoritmové knihovny nebyl před touto prací implementován ani algoritmus Quick Search pro hledání podřetězců v řetězcích, byl i tento algoritmus v rámci práce implementován. Jeho implementace byla též přidána do příslušného testovacího skriptu (konkrétně `tests.astringology.sh`) a následně několikrát otestována pro náhodné hledané podřetězce a prohledávané řetězce různých délek, též nebyly odhaleny žádné nedostatky.

3.6 Testování prezentovaného algoritmu v souboru nástrojů Forest fire & Fire wood

Nyní se dostáváme k zajímavější části našeho testování. Budeme se totiž bavit o testování implementace v rámci souboru nástrojů Forest fire & Fire wood, v rámci kterého byl prezentovaný algoritmus (a jeho varianty) porovnán s ostatními algoritmy pro řešení daného problému.

Pro účely tohoto porovnání jsme především měřili rychlost prezentovaných (ale i ostatních algoritmů), dále jsme měřili i počet indexů, které algoritmus musel vyzkoušet, aby nalezl všechny výskyty vzoru (tedy algoritmus, kterému stačilo vyzkoušet méně indexů, aby nalezl všechny výskyty vzoru, byl v tomto ohledu lepší, neboť zkoušel méně zbytečných indexů).

V rámci souboru nástrojů Forest fire & Fire wood byly pro účely tohoto měření implementovány všechny tři varianty prezentovaného algoritmu (to znamená standardní varianta nad prefixovou ohodnocenou zarážkovou notací, obrácená varianta nad prefixovou ohodnocenou zarážkovou notací a obrácená varianta nad prefixovou ohodnocenou notací). Ve všech případech jsme v rámci co největší efektivity implementace odstranily podmínku na testování, zda je řetězcová reprezentace vzoru již zarovnána s m nejpravějšími (respektive nejlevějšími pro obrácenou variantu algoritmu) znaky řetězcové reprezentace prohledávaného stromu, přidáním symbolu na konec (nebo na začátek pro obrácenou variantu) řetězcové reprezentace prohledávaného stromu, bez toho, abychom změnily n nebo nějak přeindexovali znaky dané reprezentace. Jak velký rozdíl v rychlosti je mezi implementací s podmínkou a implementací se speciálním znakem ještě uvedeme na příkladu standardní varianty algoritmu v následující části 3.9.

Pro účely samotného měření byl použit soubor testovacích dat již dříve používaný v rámci souboru nástrojů Forest fire (s tím že soubor nástrojů

Forest fire & Fire wood je jeho rozšířením). Tento soubor testovacích dat byl získán z gramatiky instrukčního setu Mono projektu X86, což je open source vývojová platforma založená na frameworku .NET. Pro každé pravidlo z této gramatiky byl pak využit strom získaný z pravé části pravidla s tím, že veškeré neterminály byly nahrazeny zástupným symbolem $*$. Výsledkem byla sada 460 stromových vzorů různých velikostí. Protože prezentovaný algoritmus (stejně jako algoritmus protisměrného vyhledávání ve stromech) je algoritmem pro hledání výskytů jednoho vzoru (tedy neumí vyhledávat sady vzorů rychlejším způsobem, než že bude vzory v sadě hledat postupně po jednom), pro účely testování jsme spustili každý z 18 testovaných algoritmů postupně individuálně nad každým vzorem, a sekvenčně jsme ho spustili pro každý strom z dvou sad stromů: sady 150 stromů, ve které každý strom měl přibližně 500 vrcholů, a sady 500 stromů, kde každý strom měl přibližně 150 vrcholů. Obě tyto sady stromů byly již dříve využity právě v rámci souboru nástrojů Forest fire.

Každý jednotlivý test (testy vyhledání jednoho ze 460 vzorů v jednom z prohledávaných stromů) byl pak opakován 21 krát s tím, že výsledný čas běhu konkrétního algoritmu pro konkrétní vzor byl určen jako průměr 20 běhů algoritmu pro tento konkrétní vzor (první běh se do průměru nezapočítával, neboť mohl být ještě ovlivněn nějakými nepříznivými vlivy v rámci systému, na kterém testy běžely).

Poznamenejme, že při testování nás zajímala především rychlost prezentovaného algoritmu, neřešili jsme využití paměti.

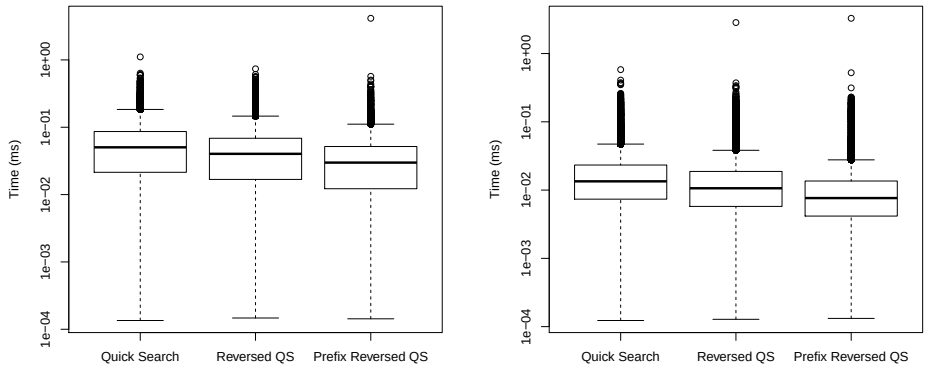
3.7 Srovnání variant algoritmu

Pojďme se nejprve podívat na výsledky jednotlivých variant prezentovaného algoritmu, tedy standardní varianty algoritmu nad prefixovou ohodnocenou zarážkovou notací, obrácené varianty algoritmu nad stejnou notací a obrácené varianty algoritmu nad prefixovou ohodnocenou notací.

Z obou grafů na obrázku 3.1 můžeme vidět, že nejlépe si vedla obrácená varianta algoritmu nad prefixovou ohodnocenou notací. To není příliš překvapivé, protože tato varianta pracuje s řetězcovou reprezentací stromu i vzoru poloviční délky oproti zbývajícím dvěma variantám algoritmu.

Obrácená varianta algoritmu nad prefixovou ohodnocenou zarážkovou notací byla v průměru o 18,5 % rychlejší než standardní varianta nad stejnou notací, tedy se potvrdila naše úvaha z části 2.5, že při skenování řetězcové reprezentace vzoru zleva doprava bude v průměru docházet k nalezení neshody ve znaku z reprezentace vzoru a znaku z reprezentace prohledávaného stromu na odpovídající pozici dříve. Platí totiž, že pro více různých nezarážkových znaků stejné arity budeme mít v abecedě pouze jeden zarážkový znak dané arity.

Obrácená varianta algoritmu nad prefixovou ohodnocenou notací byla rychlejší o 25,5 % než obrácená varianta algoritmu nad prefixovou ohodnocenou



(a) Srovnání variant algoritmu nad testovacím souborem 150 stromů

(b) Srovnání variant algoritmu nad testovacím souborem 500 stromů

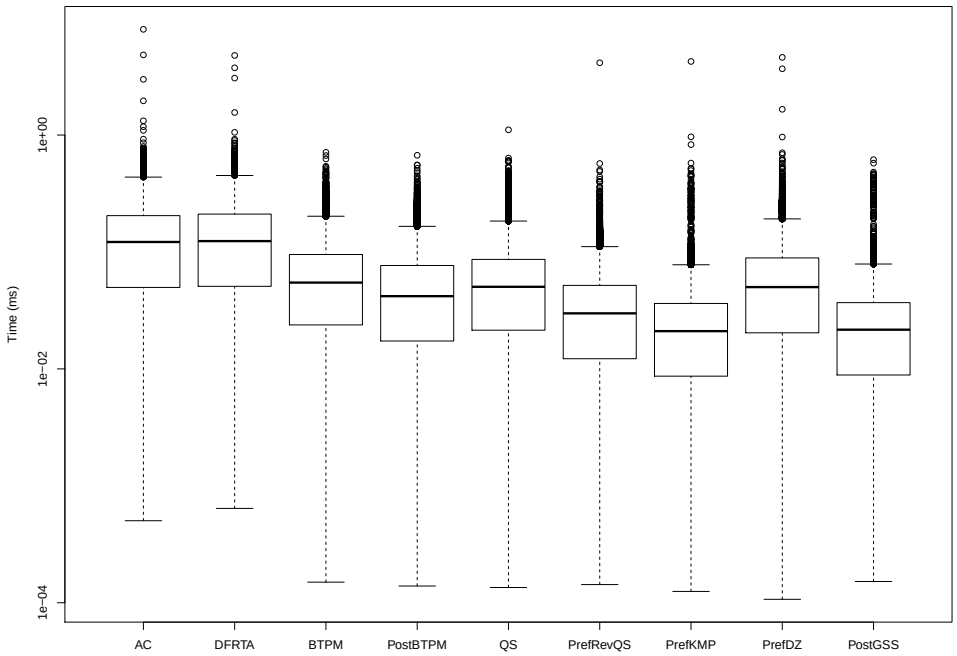
Obrázek 3.1: Srovnání všech tří variant algoritmu

zarážkovou notací, která byla o dalších již zmíněných 18,5 % rychlejší než standardní varianta alogritmu nad prefixovou ohodnocenou zarážkovou notací (obrácená varianta nad prefixovou ohodnocenou notací byla dohromady téměř o 40 % rychlejší než standardní varianta nad prefixovou ohodnocenou zarážkovou notací). Poznamenejme, že měřítko osy y v rámci obrázku 3.1 je logaritmické.

3.8 Porovnání algoritmu s ostatními existujícími algoritmy

Kromě prezentovaného algoritmu (a jeho variant) bylo součástí měření ještě 15 dalších algoritmů. Z toho 8 byly různé varianty algoritmu protisměrného vyhledávání ve stromech. Podobně jako u prezentovaného algoritmu, i algoritmus protisměrného vyhledávání ve stromech má mnoho variant, které se liší v tom, s jakou řetězcovou notací stromů pracují a v jakém směru procházejí řetězcovou reprezentaci prohledávaného stromu (a s tím souvisí i směr, v jakém skenují řetězcovou reprezentaci hledaného vzoru při ověřování výskytu na konkrétním indexu). Dále jeden stringpath vyhledávač, vyhledávač založený na deterministických konečných stromových automatech založených na principu zezdola-nahoru, dvě varianty adaptace algoritmu Knuth-Morris-Pratt pro hledání stromových vzorů ve stromech (jedna varianta pracující s prefixovou ohodnocenou zarážkovou notací, druhá s prefixovou ohodnocenou notací), dále dvě varianty algoritmu Dead Zone pro hledání stromových vzorů ve stromech, což je algoritmus, který vychází právě z algoritmu Knuth-Morris-Pratt (opět

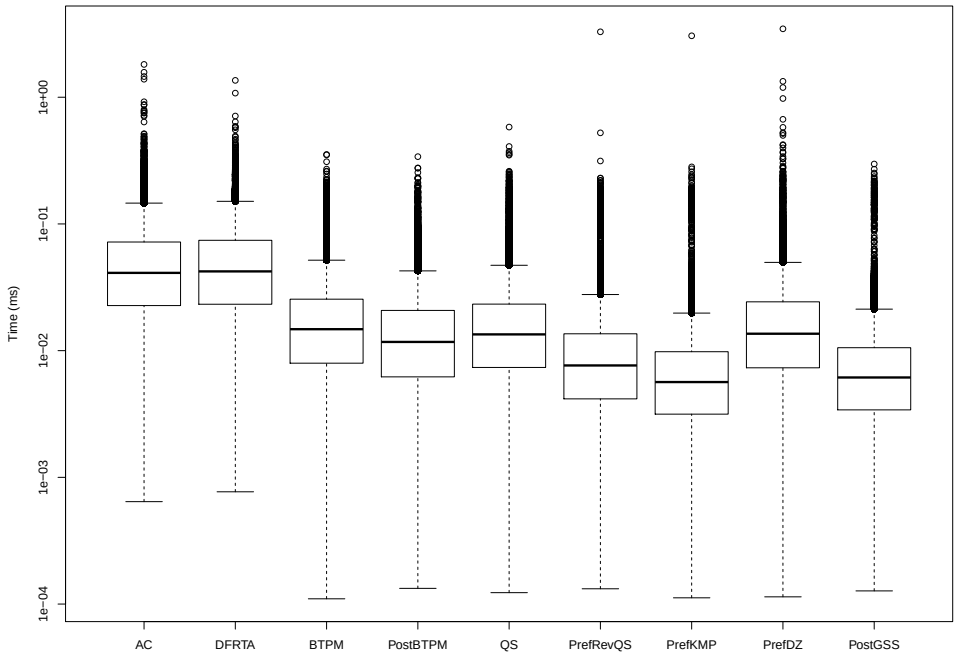
jedna varianta pracující s prefixovou ohodnocenou zarážkovou notací a druhá s prefixovou ohodnocenou notací), a nakonec algoritmus založený na heuristice good suffix shift.



Obrázek 3.2: Výsledky měření pro testovací soubor 150 stromů

Graf na obrázku 3.2 ukazuje výsledky měření pro testovací sadu 150 stromů, graf na obrázku 3.3 pak ukazuje výsledky měření pro testovací sadu 500 stromů. V obou grafech prezentujeme výsledky pouze některých algoritmů. V pořadí zleva doprava jsme zvolili následující algoritmy: stringpath vyhledávač (AC), vyhledávač založený na deterministických konečných stromových automatech založených na principu zezdola-nahoru (DFRTA), základní variantu algoritmu protisměrného vyhledávání ve stromech tak, jak byla prezentována v rámci algoritmu 5 (BTPM), variantu algoritmu protisměrného vyhledávání ve stromech pracující nad postfixovou ohodnocenou notací, jakožto nejrychlejší variantu algoritmu protisměrného vyhledávání v rámci našeho testování (PostBTPM), standardní variantu algoritmu prezentovaného v této práci (QS), variantu prezentovaného algoritmu pracující nad prefixovou ohodnocenou notací, jakožto nejrychlejší variantu prezentovaného algoritmu v rámci našeho testování (PrefRevQS), rychlejší variantu adaptace algoritmu Knuth-Morris-Pratt, tedy variantu pracující s prefixovou ohodnocenou notací (PrefKMP), rychlejší variantu algoritmu Dead Zone, tedy opět variantu pracu-

jící s prefixovou ohodnocenou notací (PrefDZ), a konečně algoritmus založený na heuristice good suffix shift, který pracuje s postfixovou ohodnocenou notací (PostGSS).



Obrázek 3.3: Výsledky měření pro testovací soubor 500 stromů

Z obou grafů můžeme vidět, že obrácená varianta algoritmu nad prefixovou ohodnocenou notací si vedla relativně dobře. V našich testech vyšla jako třetí nejrychlejší algoritmus, ihned po rychlejší variantě adaptace algoritmu Knuth-Morris-Pratt pro hledání sromových vzorů ve stromech, tedy variantě pracující nad prefixovou ohodnocenou notací, a algoritmu založeném na heuristice good suffix shift nad postfixovou ohodnocenou notací.

Nejrychlejší varianta prezentovaného algoritmu (tedy obrácená varianta nad prefixovou ohodnocenou notací) pak byla o 30 % rychlejší než nejrychlejší varianta algoritmu protisměrného vyhledávání ve stromech a zároveň více než 5 krát tak rychlá jako algoritmus založený na stromových automatech. Pro srovnání, algoritmus protisměrného vyhledávání je zhruba 3,63 krát tak rychlý.

Z výsledků testu se dá usuzovat, že prezentovaný algoritmus dosahuje výrazně lepších výsledků než algoritmus protisměrného vyhledávání ve stromech, přitom využívá stejné pomocné struktury (a tedy je i stejně náročný na paměť), pouze dané struktury konstruuje jinak a posuny provádí na základě znaku mimo aktuální porovnávací rozsah, zatímco algoritmus protisměrného

vyhledávání ve stromech provádí posuny na základě znaku, který je součástí aktuálního porovnávaného rozsahu.

Nejrychlejší varianta algoritmu protisměrného vyhledávání ve stromech vyzkouší v průměru o 72,35 % možných indexů, na kterých by se vzor mohl vyskytovat, než nejrychlejší varianta námi prezentovaného algoritmu. Standardní varianta algoritmu protisměrného vyhledávání ve stromech (nad prefixovou ohodnocenou zarážkovou notací) vyzkouší v průměru o 33 % více možných indexů než standardní varianta prezentovaného algoritmu nad stejnou notací. Tyto výsledky potvrzují naše tvrzení z části 2.2, že bychom očekávali, že posuny učiněné prezentovaným algoritmem budou v průměru větší než posuny učiněné algoritmem protisměrného vyhledávání ve stromech.

Jak jsme již zmínili, lepších výsledků než prezentovaný algoritmus dosahovaly v našich testech dva algoritmy. Rychlejší varianta algoritmu Knuth-Moriss-Pratt (která byla nejrychlejší ze všech testovaných algoritmů) byla o 27,66 % rychlejší než nejrychlejší varianta prezentovaného algoritmu. Tento algoritmus byl ještě o 4 % rychlejší než algoritmus založený na heuristice good suffix shift. Myšlenka námi prezentovaného algoritmu lze ovšem zkombinovat s myšlenkou algoritmu založeném na good suffix shift heuristice tím způsobem, že se velikost posunu bude určovat jako maximum z hodnoty v tabulce posunů prezentované v této práci a z hodnoty získané good suffix shift heuristikou. Kombinací těchto dvou myšlenek může vzniknout nový algoritmus, u kterého očekáváme, že bude dosahovat lepších výsledků než adapatace algoritmu Knuth-Moriss-Pratt.

3.9 Rozdíl v efektivitě algoritmu před a po odstranění nepotřebné podmínky

V části 3.3 jsme popsali, jak je možné se zbavit podmínky na testování, zda je řetězcová reprezentace vzoru již zarovnána s m nejpravějšími (respektive nejlevějšími pro obrácenou variantu algoritmu) znaky řetězcové reprezentace prohledávaného stromu. V algoritmu 7 můžeme tuto podmínku vidět na řádce 16.

V rámci implementace standardní varianty prezentovaného algoritmu v souboru nástrojů Forest fire & Fire wood byly vyzkoušeny obě varianty (jak s podmínkou, tak s jejím odstraněním přidáním znaku na konec řetězcové reprezentace prohledávaného stromu), aby bylo možné vyhodnotit vliv této podmínky na rychlost algoritmu.

Nad testovacím souborem se 150 stromy o velikosti 500 vrcholů jsme změřili nejprve celkovou dobu běhu (tedy součet dob běhu pro jednotlivé dvojice stromů a vzorů) standardní varianty algoritmu s podmínkou, a následně po odstranění této podmínky a přidání znaku na konec řetězcové reprezentace prohledávaného stromu tak, jak to bylo ukázáno v části 3.3. Po odstranění podmínky se algoritmus urychlil zhruba o 2 %. Toto zrychlení v řádu jedno-

tek procent nám přišlo nezanedbatelné, proto byly ostatní varianty algoritmu implementovány rovnou s odstraněnou podmínkou. V případě, že by prezentovaný algoritmus měl být nasazen někde, kde je důležitá rychlost, pak se určitě nahrazení podmínky přidáním symbolu vyplatí udělat, neboť tato úprava je relativně jednoduchá a výsledné zrychlení může být znát především u aplikací, které by algoritmus využívaly často.

Závěr

Cílem této práce bylo navrhnout adaptaci algoritmu Quick Search pro hledání podřetězců v řetězcích, která by byla schopná řešit problém hledání stromových vzorů ve stromech. Dále implementovat výsledné algoritmy (respektive varianty algoritmu pro různé notace či směry procházení prohledávaného stromu), integrovat je do projektu *Algoritmová knihovna*, změřit jejich rychlost a porovnat ji s ostatními existujícími algoritmy řešícími stejný problém.

Algoritmus byl navržen a úspěšně implementován. V projektu *Algoritmová knihovna* vzniklo šest nových algoritmů a několik jejich variant. Tyto algoritmy už je nyní možné v rámci knihovny používat.

Měřením v rámci souboru nástrojů Forest fire & Fire wood se ukázalo, že výsledný algoritmus, konkrétně jeho obrácená varianta na prefixové ohodnocené notaci (která vychází v testech nejlépe), poskytuje v průměru o 30 % lepší výsledky než algoritmus protisměrného vyhledávání ve stromech (respektive jeho nejrychlejší varianta). V případě, že je potřeba vyhledat pouze jeden vzor, je algoritmus více než pětinásobně tak rychlý jako algoritmy založené na stromových automatech (algoritmus protisměrného vyhledávání je zhruba 3,6 krát tak rychlý). Jedná se tedy o velmi rychlý algoritmus pro řešení daného problému, který může být použit v praxi.

Navazující práce by se mohly zabývat například tím, zda by bylo možné algoritmus rozšířit tak, aby dokázal pracovat s množinami vzorů efektivněji, než je pouze vyhledávat po jednom. Dále je možné prozkoumat další heuristiky posunů, které by umožnili ještě větší posuny reprezentace vzoru v reprezentaci prohledávaného stromu, než umožňuje tabulka posunů (BCS) prezentovaná v této práci. Velmi nadějnou heuristikou by mohla být například heuristika good suffix shift. Kombinací myšlenek tabulky posunů prezentované v této práci a heuristiky good suffix shift bude možné získat nový algoritmus, který bude dosahovat ještě lepších výsledků než algoritmus prezentovaný v této práci. V neposlední řadě může být algoritmus rozšířen i na jiné notace pro linearizované stromy než je prefixová ohodnocená notace a prefixová ohodnocená zarážková notace.

Bibliografie

1. TRÁVNÍČEK, Jan et. al. Backward Linearised Tree Pattern Matching. In: *Language and Automata Theory and Applications*. Cham: Springer International Publishing, 2015, s. 599–610. ISBN 978-3-319-15579-1.
2. CHASE, David R. An Improvement to Bottom-up Tree Pattern Matching. In: *POPL*. Mnichov: ACM Press, 1987, s. 168–177. ISBN 0-89791-215-2. Dostupné z DOI: 10.1145/41625.41640.
3. CLEOPHAS, Loek. *Tree Algorithms: Two Taxonomies and a Toolkit*. 2008. Disertační práce. Department of Mathematics a Computer Science, Eindhoven University of Technology.
4. HOFFMANN, Christoph M.; O'DONNELL, Michael J. Pattern Matching in Trees. *Journal of the ACM*. 1982, roč. 29, č. 1, s. 68–95. ISSN 0004-5411. Dostupné z DOI: 10.1145/322290.322295.
5. AHO, Alfred V.; GANAPATHI, Mahadevan; TJIANG, Steven W. K. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.* 1989, roč. 11, č. 4, s. 491–516. ISSN 0164-0925. Dostupné z DOI: 10.1145/69558.75700.
6. HOFFMANN, Christoph M.; O'DONNELL, Michael J. Pattern Matching in Trees. *J. ACM*. 1982, roč. 29, č. 1, s. 68–95. ISSN 0004-5411. Dostupné z DOI: 10.1145/322290.322295.
7. BOYER, Robert S.; MOORE, J. Strother. A Fast String Searching Algorithm. *Commun. ACM*. 1977, roč. 20, č. 10, s. 762–772. ISSN 0001-0782. Dostupné z DOI: 10.1145/359842.359859.
8. NIGEL, Horspool R. Practical fast searching in strings. *Software: Practice and Experience*. 1980, roč. 10, č. 6, s. 501–506. Dostupné z DOI: 10.1002/spe.4380100608.

9. KNUTH, Donald E.; MORRIS, James H.; PRATT, Vaughan R. Fast Pattern Matching in Strings. *SIAM Journal on Computing*. 1977, roč. 6, č. 2, s. 323–350. Dostupné z DOI: 10.1137/0206024.

10. SUNDAY, Daniel M. A Very Fast Substring Search Algorithm. *Commun. ACM*. 1990, roč. 33, č. 8, s. 132–142. ISSN 0001-0782. Dostupné z DOI: 10.1145/79173.79184.

11. CHARRAS, Christian; LECROQ, Thierry. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004. ISBN 0954300645.

12. ŽÁK, Martin. *Automatová knihovna - vnitřní a komunikační formát*. Praha, 2014. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

13. PLACHÝ, Štěpán. *Automatová knihovna - Stromové automaty a algoritmy nad stromy*. Praha, 2015. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

14. SHATROVSKII, Aleksandr. *Implementation of a repetition searching algorithm in trees*. Praha, 2017. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

Příloha **A**

Seznam použitých zkratek

BCS Bad Character Shift Table, tabulka posunů

CLI Command Line Interface, konzolové rozhraní, rozhraní příkazové rádky

SJT Subtree Jump Table, podstromová skoková tabulka

XML Extensible Markup Language, rozšiřitelný značkovací jazyk

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `alib` ..... projekt *Algoritmová knihovna*
- `forestfire` ..... soubor nástrojů *Forest fire & Fire wood*
- `text`
  - `BP_Cvach_Michal_2018.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojové soubory práce ve formátu XeLaTeX