FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Quantum leap vyhledávání v linearizovaných stromech
Student:	Bc. Josef Erik Sedláček
Vedoucí:	Ing. Jan Trávníček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Nastudujte Quantum leap vyhledávací algoritmus pro řetězce popsaný v [1].
Nastudujte vyhledávání v linearizovaných stromech na principu mrtvých zón [2].
Navrhněte adaptaci algoritmu Quantum leap pro vyhledávání v linearizovaných stromech.
Implementujte navržený algoritmus v programovacím jazyce Java do nástroje ForestFIRE [3] s důrazem na efektivitu.
Důkladně otestujte algoritmus na vedoucím dodaných stromech a stromových vzorcích.

Seznam odborné literatury

[1] WATSON, Bruce W.; KOURIE, Derrick G.; CLEOPHAS, Loek G. Quantum Leap Pattern Matching. In: *Stringology*. 2015. p. 104-117.
[2] OBŮRKA, Robin. Vyhledávání ve stromech na principu mrtvých zón. 2016.
[3] CLEOPHAS, Loek. Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms. In: *Proceedings of the 2009 conference on Finite-State Methods and Natural Language Processing: Post-proceedings of the 7th International Workshop FSMNLP 2008*. 2009. p. 191-198.

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

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

V Praze dne 4. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Quantum leap vyhledávání v linearizovaných stromech

Bc. Josef Erik Sedláček

Katedra Teoretické Informatiky
Vedoucí práce: Ing. Jan Trávníček Ph.D.

28. srpna 2020

Poděkování

Myslím, že slovy nedokážu popsat, jak moc jsem vděčný svému vedoucímu Ing. Janu Trávníčkovi Ph.D. Děkuji mu, že i když věděl, co obnáší práce se mnou, protože mi vedl bakalářskou práci, tak se obětoval a rozhodl se vést i mou diplomovou práci a projít si tím ještě jednou. Zároveň bych se mu tímto chtěl omluvit, že jsem nebyl schopný nijak pokročit s původním tématem práce, na kterém jsme se dohodli, kvůli čemuž jsme na poslední chvíli měnili zadání. Ještě jednou děkuji a přeji mu, aby budoucí studenti, kterým povede práce, byli mnohem schopnější, než jsem byl já.

Dále bych chtěl poděkovat všem lidem, kteří mi pomohli projít celým vysokoškolským studiem. Hlavně bych chtěl poděkovat svým spolužákům, kteří mi celé ty roky radili a pomáhali s domácími úkoly, projekty a přípravou na zkoušky. Jmenovitě musím poděkovat několika lidem.

Miroslav Sochor mi pomohl projít celým studiem a myslím, že bez něj bych jen velmi těžko prošel bakalářským studiem. Opravdu nejsem schopen vyjmenovat vše, s čím mi pomohl. Prvák na magistru pro mě byl mnohem jednodušší hlavně díky němu, zvláště ve zkouškovém období. Chtěl bych hlavně zmínit předměty PJP, APS, ADM a PSI, ve kterých si nedovedu představit, jak bych bez něho zvládl semestrální práce.

Jan Matyáš Křišťan mi pomohl projít hlavně bakalářským studiem, ale i na magistru mi velmi pomohl například s předmětem MVI. Při pohledu na bakalářské studium také nejsem schopen, jako v případě Míry, vyjmenovat vše. Rozhodně bych mu chtěl ale poděkovat za pomoc s úlohami na Progtest. Také musím zmínit předměty ZDM a hlavně AG2, protože díky němu pro mě byla zkouška mnohem snazší. I když na tu dobu velmi nerad vzpomínám, tak musím poděkovat i za psychickou podporu během pátého semestru na bakaláři. Stejně jako bez Míry bych i bez Matyáše neměl ani bakalářský titul.

Minh Quang Trieu byl první člověk, na kterého jsem se během třetího semestru magisterského studia obrátil, když jsem měl nějaký dotaz. Hlavně mi pomohl se semestrální prací do předmětu KOP a při přípravě na zkoušku

z MVI a PDP. Také byl součástí naší skupiny na domácí úkoly z GAK, takže rozhodně přispěl k tomu, že jsem dostal známku už v semestru.

Martin Bobek mi na magistru několikrát poradil s úlohami na KOP a kromě Minha jsem řešení probíral často i s ním. Dále si minimálně vzpomenu na několik rad, které mi dal s první úlohou na OSY ještě během bakalářského studia.

Jan Píro, Adam Platkevič, Kamil Červený a Lukáš Renc jsou lidé, které jsem před magisterským studiem znal nanejvýš od vidění. Musím je ale zmínit aspoň kvůli tomu, že se mnou byli ve skupince, ve které jsme řešili úkoly na GAK. Také bych jim rád poděkoval za sraz před zkouškou na NON, kde jsme prošli příklady, díky čemuž jsem nakonec zkoušku udělal.

I přesto, že jsem se s následujícími lidmi během magisterského studia příliš nebo vůbec nevídal, bych jim chtěl ještě jednou poděkovat za pomoc během bakalářského studia. Přece jen je pravda, že jen těžko bych v tuto chvíli psal diplomovou práci, kdybych se nedostal ani k té bakalářské, a protože je toto velmi pravděpodobně má poslední takováto práce, tak bych je rád zmínil i zde.

Jan Uhlík mi asi nejvíce pomohl těsně před státnicemi na bakaláři, kdy mi pomohl s otázkou k matematické logice, které jsem vůbec nerozuměl. Jelikož jsem v komisi měl přednášejícího matematické logiky, tak jsem se této otázky opravdu bál, ale díky Honzovi se můj strach značně zmírnil. Dále mu musím poděkovat za skvělou přípravu na zkoušku před APS. Nemám vůbec představu, jak bych zkoušku bez něj udělal. Také mu ale musím poděkovat za organizaci týmu v SI1 a za rady v AG2.

Giang Chau Nguyenová pro mě byla, stejně jako Matyáš a Míra, nemalou psychickou podporou během studia. Co se týče předmětů, tak musím zmínit přípravu na zápočtové testy na AAG a rady s PST před zkouškou. Rád bych poděkoval i za pomoc s LaTeXem, když jsem začínal psát bakalářskou práci.

Michal Cvach mi hodně poradil s druhou úlohou na APS. Díky jeho radě jsem úlohu vyřešil opravdu rychle, což jsem zrovna v pátém semestru bakaláře opravdu potřeboval. Také si vzpomínám, že se mnou procházel příklady na PPA před testem a zkouškou.

Hoang Huy Vu v předmětu APS napsal překladač pro překlad assembleru do strojového kódu, který mi značně ulehčil práci. Jak se znám, tak by mě taková věc nenapadla a nebýt jeho, tak bych ty instrukce překládal nejspíš ručně. Také bych chtěl poděkovat za shrnutí VZD před státnicemi.

Tung Anh Vu byl často se mnou a Matyášem na přípravě na zkoušku na AG2, jednoho z nejtěžších předmětů, který jsem měl. Také bych řekl, že díky němu se celá naše skupina sblížila s několika doktorandy, takže jsme před státnicemi mohli okupovat jejich kancl a využívat tiskárnu.

Ještě jednou všem moc děkuji.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, 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 zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). 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 zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 28. srpna 2020 . . . . . .

Odkaz na tuto práci

Sedláček, Josef Erik. *Quantum leap vyhledávání v linearizovaných stromech.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Práce se zabývá vyhledáváním stromových vzorů ve stromech. Konkrétně se pracuje s linearizovanými stromy v prefixové notaci. Práce přejímá myšlenku algoritmu Quantum Leap pro vyhledávání v řetězcích a využívá ji pro vyhledávání vzorů ve stromech. Několik verzí algoritmu je implementováno do souboru nástrojů Forest FIRE a nejlepší z nich jsou srovnány s již existujícími algoritmy ve zmíněném souboru nástrojů.

Klíčová slova Vyhledávání vzorů ve stromech, Linearizace stromů, Quantum Leap algoritmus

Abstract

In this thesis a tree pattern matching is studied with a focus on linearised trees in prefix notation. The idea of the string matching algorithm called Quantum Leap is adapted to search tree patterns in linearised trees. Several variants of the algorithm are implemented as a part of the Forest FIRE toolkit and the empirically best implementations are compared with other existing algorithms in the toolkit.

Keywords Tree pattern matching, Tree linearisation, Quantum Leap algorithm

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vyhledávání v řetězcích je jedním z nejčastěji řešených problémů v informatice. Většinou jsou prohledávané texty a hledaná slova krátká a naivní algoritmus pro vyhledávání je naprosto dostačující, pokud se ale vyhledává výskyt nějakého textu v databázi nebo podřetězce v DNA, je třeba, aby byl algoritmus co nejefektivnější. Z tohoto důvodu je problém vyhledávání v řetězcích široce studován a existuje nespočet algoritmů pro jeho řešení.

Jeden pohled na vyhledávání ve stromech je, že se jedná o obecnější problém vyhledávání v řetězcích. Například napsaný program se často překládá do takzvaného abstraktního syntaktického stromu AST. Jednoduchým příkladem může být, pokud by bylo třeba vyhledat část kódu, jehož součástí je podmínka a blok kódu s ní související. Pokud obsah bloku není znám nebo není podstatný, může být nahrazen zástupným znakem, který reprezentuje celý blok libovolné délky. Tento blok kódu by v AST byl reprezentován jako nějaký úplný podstrom. Tento problém vyhledávání *bloku kódu* se tedy dá chápat jako problém vyhledávání stromového vzoru ve stromu a naopak vyhledávání stromových vzorů ve stromech se dá chápat jako vyhledávání v řetězcích, ve kterých se vyskytují znaky reprezentující text libovolné délky odpovídající nějakému syntaktickému celku.

Problém vyhledávání stromových vzorů v ohodnocených označených stromech je důležitým problémem s aplikací například v kompilátorech, interpretaci neprocedurálních jazyků nebo zpracování značkovacích jazyků. [2] Stromy mohou být reprezentovány řetězci za pomocí linearizace. Průchod stromem nějakým sekvenčním algoritmem odpovídá nějaké lineární reprezentaci, aniž by byla explicitně tvořena.

Cíl práce

Hlavním cílem práce je naimplementovat nový algoritmus pro vyhledávání stromových vzorů ve stromech. Je třeba naimplementovat několik verzí algo-

ritmu a vyhodnotit jejich efektivitu. Implementace a měření probíhá v rámci souborů nástrojů Forest FIRE, ve kterém již existuje nemalé množství různých algoritmů řešící problém vyhledávání ve stromech.

V rámci teoretické části je třeba nastudovat algoritmus Quantum Leap pro vyhledávání v řetězcích a několik algoritmů pro vyhledávání vzorů ve stromech. Upravit algoritmus Quantum Leap tak, aby byl schopný přijmout linearizovaný strom a stromový vzor a vrátit všechny výskyty vzoru ve stromu.

Struktura práce

Kapitola 1 shrnuje základní definice potřebné k pochopení této práce. Kapitola 2 popisuje potřebné algoritmy pro vyhledávání v řetězcích, které jsou s úpravami použity pro vyhledávání stromových vzorů ve stromech. Následně v kapitole 3 jsou popsány algoritmy pro vyhledávání ve stromech, vycházející s algoritmů popsaných v předchozí kapitole. V kapitole 4 se pojednává o souboru nástrojů Forest FIRE a samotné implementaci algoritmu Quantum Leap do zmíněného souboru nástrojů. V poslední kapitole 5 je popsáno, jakým způsobem probíhalo vyhodnocení implementace a samotné výsledky testování.

Kapitola 1

Základní definice

Kapitola shrnuje základní pojmy z teorie grafů a formálních jazyků, které jsou potřeba k pochopení této práce. Většina pojmů je převzata z [3], [4], [5] a [6]. Pokud je definice nebo její část převzata z jiného zdroje, je tak uvedeno přímo u ní.

1.1 Řetězce a vyhledávání v nich

V této části jsou definovány základní pojmy teorie formálních jazyků.

Definice 1 Abeceda *je konečná množina znaků a většinou je značená* Σ. Ohodnocená abeceda *je abeceda, kde má každý znak* $a \in \Sigma$ *přiřazené číslo* $k \in \mathbf{N}$*, které se nazývá arita.[2] Ohodnocený znak* a *s aritou* k *bude značen* a_k.

Definice 2 Řetězec *je konečná posloupnost znaků. Množina všech řetězců nad abecedou* Σ *je značena* Σ^**, přičemž prázdný řetězec je značen* ε*. Délka řetězce je značena* $|w|$ *a platí* $|\varepsilon| = 0$*. Znak na indexu* i *v řetězci* w *je značen* $w[i]$ *v této práci indexováno od 1.* Podřetězec w *je souvislá podposloupnost znaků a podřetězec od indexu* i *do indexu* j *bude značen* $w[i:j]$*. Pokud* $j < i$*, pak* $w[i:j] = \varepsilon$.

Definice 3 Zřetězení *je operace připojení jednoho řetězce za druhý. Nechť* $x, y \in \Sigma^*$ *jsou dva řetězce, jejich zřetězení je značené* xy*. Operace zřetězení je asociativní* $\forall x, y, z \in \Sigma^* : (xy)z = x(yz)$*, není komutativní* $\exists x, y \in \Sigma^* : xy \neq yx$ *a* ε *se pro ni chová jako neutrální prvek* $\forall x \in \Sigma^* : \varepsilon x = x\varepsilon = x$.

Přestože se práce zabývá vyhledáváním vzorů ve stromech, pracuje se s řetězcovou reprezentací stromů. Přesný popis této reprezentace je uveden v části 1.3. Je proto důležité říct, co je problém vyhledávání v řetězcích.

Definice 4 *Nechť $w, p \in \Sigma^*$ jsou dva řetězce délky $|w| = n$ a $|p| = m$.* Problém vyhledávání v řetězcích *je problém nalezení výskytů řetězce p v řetězci w. Tedy najít taková i, že pro podřetězce $w[i : i + m - 1]$ platí, že se rovnají řetězci p, nebo říct, že takové i neexistuje.*

1.2 Teorie grafů

Tato část definuje základní pojmy teorie grafů, které jsou třeba k pochopení této práce.

Definice 5 Graf *je uspořádaná dvojice (V, E), kde V je neprázdná množina vrcholů a E je množina hran. Hrana je dvojice vrcholů z V. Pokud je hrana neuspořádaná dvojice $\{u, v\} \in E, u, v \in V$, pak se graf nazývá* neorientovaný*. Pokud je uspořádaná $(u, v) \in E$, pak je graf* orientovaný *a vrchol u se nazve předchůdcem v a v následníkem u.*

Definice 6 Stupeň *vrcholu v v neorientovaném grafu G, značený $deg_G(v)$, je počet hran, ve kterých se nachází v. Pokud je zřejmé, o jaký graf se jedná, pak bude stupeň zapisován pouze jako $deg(v)$. V orientovaném grafu se definuje* výstupní stupeň *vrcholu v značený jako $deg^-_G(v)$ jako počet hran vycházejících z v.* Vstupní stupeň *značený $deg^+_G(v)$ je počet hran končících ve v. Vrchol se vstupním stupněm 0 se nazývá* zdroj *a vrchol s výstupním stupněm 0 je* stok*.*

Definice 7 Cesta *je neorientovaný graf $G(V, E)$, kde $V = \{1, \ldots, n\}$ a $E = \{\{i, i+1\} | i \in \{i, \ldots, n-1\}\}$. Neorientovaný graf $G = (V, E)$ se nazývá* souvislý *právě tehdy, když pro každé dva vrcholy $v, u \in V$ existuje cesta mezi v a u. Jinak je G* nesouvislý*. Orientovaný graf je souvislý, pokud je souvislý po odstranění orientace hran.*

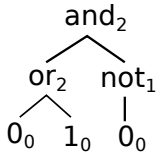
Definice 8 Kružnice *je neorientovaný graf $G = (V, E)$, kde $V = \{1, \ldots, n\}$ a $E = \{\{i, i + 1\} | i \in \{i, \ldots, n - 1\}\} \cup \{\{n, 1\}\}$.* Kružnice v grafu G *je právě tehdy, když nějaký jeho podgraf je izomorfní s kružnicí.*

Definice 9 Strom *je souvislý graf, ve kterém se nenachází kružnice. Orientovaný graf je strom, pokud je stromem po odstranění orientace hran. Vrcholy se stupněm 1 se nazývají* listy*.*

Zakořeněný strom *je strom, ve kterém je jeden vrchol $r \in V$, který se nazývá kořen. Pokud vrchol $u \in V$ leží na cestě z $v \in V$ do kořene, pak je u předek v. Vrchol v je potomek u. Pokud navíc platí $\{u, v\} \in E$, pak je u otec v a vrchol v je syn u. Zakořeněný orientovaný strom má orientované hrany. V této práci se vždy bude považovat orientace směrem od kořene. Kořen je potom jediný zdroj a všechny listy jsou stokem.*

Definice 10 Uspořádaný strom *je zakořeněný strom, ve kterém pro každý vrchol platí, že množina jeho synů je uspořádaná.* Ohodnocený označený strom *je uspořádaný strom, ve kterém je každý vrchol v označen znakem a_k z ohodnocené abecedy. Dále platí, že arita znaku a_k se rovná výstupnímu stupni vrcholu v. Tedy platí $k = deg^-(v)$. [4]*

Na obrázku 1.1 je příklad ohodnoceného označeného stromu nad abacedou $\Sigma = \{and_2, or_2, 0_0, 1_0\}$.



Obrázek 1.1: Příklad ohodnoceného označeného stromu

Práce se zabývá vyhledáváním v ohodnocených označených stromech, z tohoto důvodu, pokud nebude řečeno jinak, se pod pojmem *strom* myslí vždy ohodnocený označený strom.

Definice 11 Úplný podstrom *orientovaného stromu $T = (V, E)$ je strom $T' = (V', E')$, pro který platí:*

- $V' \subseteq V$,
- $E' = (V' \times V') \cap E$,
- *žádný vrchol $v \in V \setminus V'$ není potomkem vrcholu ve V'. [7]*

Neformálně řečeno pro úplný podstrom platí, že pokud je nějaký vrchol součástí podstromu, pak jsou i všichni jeho potomci součástí daného podstromu. Pokud by například na stromu uvedeném na obrázku 1.1 byl uvažován úplný podstrom, který obsahuje vrchol označený or_2, pak nutně musí být součástí daného podstromu i potomci tohoto vrcholu 0_0 a 1_0.

V této práci se vždy bude uvažovat úplný podstrom, a pokud nebude řečeno jinak, tak pod pojmem *podstrom* se myslí úplný podstrom.

1.3 Vyhledávání ve stromech a linearizace stromů

Pojmy jako linearizace stromů, stromový vzor a vyhledávání vzorů ve stromech jsou definovány v této části. Definice byly převzaty z [2], [8] a [9].

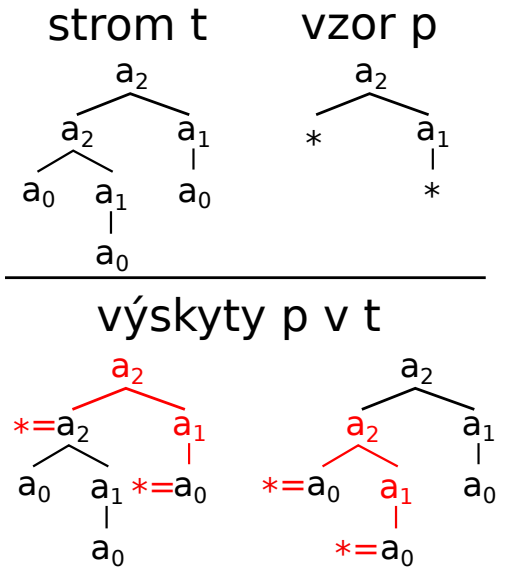
Definice 12 Stromový vzor *je uspořádaný strom, ve kterém se místo běžných vrcholů mohou vyskytovat výjimečné vrcholy značené $*$. Tyto vrcholy reprezentují libovolný úplný podstrom.*

Z definice plyne, že se vrcholy $*$ mohou vyskytovat pouze v listech stromu, protože zastupují *úplný* podstrom. Tento fakt je zde ale ještě jednou zdůrazněn, aby u čtenáře nedošlo ke zbytečnému zmatení. Z tohoto faktu je zřejmé, že v ohodnocených označených stromech mají vrcholy $*$ aritu 0 a měly by tedy být značeny $*_0$, jelikož je ale arita takového vrcholu zřejmá, bude vždy značen pouze $*$ bez uvedené arity.

Definice 13 Problém vyhledávání vzorů ve stromech *je problém nalezení všech výskytů stromového vzoru* $p = (V_p, E_p)$ *ve stromu* $t = (V, E)$. *Vzor* p *se vyskytuje ve vrcholu* $v \in V$ *pokud platí jedna z následujících možností:*

- $p = *$
- *Symbol v kořenu* p *se rovná symbolu ve vrcholu* v *a pro všechny podstromy* p *značené* $x_{p1}, x_{p2}, \ldots, x_{pn}$ *a podstromy vrcholu* v *značené* $x_{v1}, \ldots, x_{vn}$ *platí* $\forall i \in \{1, \ldots, n\} : x_{pi}$ *se vyskytuje ve stromu* x_{vi}, *kde* n *je arita kořene* p.

Na obrázku 1.2 je uveden příklad stromu t a stromového vzoru p nad abecedou $\Sigma = \{a_0, a_1, a_2\}$. Na obrázku jsou také znázorněny oba výskyty vzoru p v t.



Obrázek 1.2: Příklad výskytu vzoru ve stromu

Existují různé přístupy k řešení problému vyhledávání vzorů ve stromech, za zmínku stojí například algoritmy pracující s konečnými stromovými automaty. Tato práce se ale zabývá algoritmem založeným na linearizaci stromů.

Linearizace stromů je založená na tom, že stromy mohou být reprezentované řetězcem. Známým příkladem je závorková reprezentace, kterou lze

reprezentovat obyčejný zakořeněný strom. Například strom t z obrázku 1.2 by se dal reprezentovat jako (((()(()))(()))), samozřejmě za předpokladu, že je třeba reprezentovat pouze tvar stromu.

V případě nutnosti reprezentovat i označení stromů je třeba využít složitějšího zápisu. Takovýchto reprezentací opět existuje více a v této práci se pracuje s prefixovou ohodnocenou notací. [1] Za zmínku ale také stojí prefixová zarážková notace, ve které se zavádí speciální znak *zarážky*, který znázorňuje konec podstromu. [10]

Definice 14 Prefixová ohodnocená notace $pref(t)$ *pro strom* t *je definována následovně:*

- $pref(*) = *$*,*
- $pref(a) = a_0$*, pokud* a *je list,*
- $pref(t) = a_n\ pref(b_1)pref(b_2)\dots pref(b_n)$*, kde* a *je kořen stromu* t *a* n *je arita vrcholu* a *a* $b_1, \dots, b_n$ *jsou přímí potomci vrcholu* a*.*

Příkladem nechť jsou opět strom t a vzor p z obrázku 1.2, pro které platí $pref(t) = a_2a_2a_0a_1a_0a_1a_0$ a $pref(v) = a_2 * a_1*$.

Smysl této reprezentace spočívá v reprezentaci složité struktury, kterou je strom, jednoduchým řetězcovým zápisem. Vyhledávání v řetězcích je široce studované téma a na řešení tohoto problému existuje obrovské množství různě efektivních algoritmů. Prefixová notace dává možnost využít tyto algoritmy pro vyhledávání ve stromech. Jediným rozdílem oproti naprosto obyčejnému vyhledávání v řetězcích je v tomto případě proměnná $*$, která reprezentuje libovolně velký podstrom, a která se může vyskytovat ve vzoru. Problém vyhledávání vzoru ve stromu i po převedení do prefixové notace je tedy obecnější problém a do řetězcového algoritmu řešící tento problém je třeba přidat nějaké struktury, které dovolí s podstromy efektivně pracovat.

Kapitola 2

Algoritmy pro vyhledávání v řetězcích

Jak bylo řečeno na konci předchozí kapitoly, vyhledávání vzorů ve stromech je možné převést na vyhledávání v řetězcích jen za cenu několika úprav. Z tohoto důvodu se tato kapitola bude věnovat řetězcovým algoritmům, které byly v práci použity. V následující kapitole, kde jsou popsány algoritmy pro vyhledávání ve stromech, jsou popsány hlavně rozdíly oproti algoritmům pro vyhledávání v řetězcích.

2.1 Morris-Pratt

Algoritmus Morris-Pratt (dále jen MP) je nejspíše méně známý než jeho vylepšená verze Knuth-Morris-Pratt. Algoritmy se ale liší pouze ve funkci posunů, podle které se určuje velikost posunu hledaného vzoru po každém porovnání. Protože v této práci použitý algoritmus pro vyhledávání vzorů ve stromech vychází z algoritmu MP a ne KMP, je v této části popsána právě verze MP.

Algoritmus MP se řadí mezi dopředné algoritmy vyhledávání v řetězcích, což znamená, že porovnávání znaků textu se vzorem probíhá ve stejném směru jako posun vzoru. Algoritmus je velmi podobný naivnímu algoritmu, který vždy porovná znaky a při neshodě posune vzor o jeden znak doprava. Jediný rozdíl algoritmu MP oproti naivnímu vyhledávání je funkce posunů, díky které jsou možné posuny vzoru o více znaků.

Před uvedením algoritmu MP je nejprve třeba zadefinovat pojmy *border* a *border array*.

Definice 15 Border b *řetězce* w *je jakákoliv vlastní předpona* w*, která se rovná příponě* w*.* Border array *řetězce* w *je pole* β *velikosti* $|w|$*, pro které platí* $\beta[i] = |b^*(w[1..i])|$*, kde* $b^*(x)$ *udává velikost největšího borderu slova* x*.*

Neformálně řečeno má border array slova w na indexu i velikost největšího borderu předpony slova w, která končí na i-té pozici. Za zmínku také stojí, že prázdný řetězec neodporuje definici borderu a tedy každý řetězec má aspoň jeden border ε velikosti $|\varepsilon| = 0$.

Pro uvedení příkladu nechť $w = abrakadabra$. Slovo w má tři bordery a to konkrétně $\varepsilon, a, abra$. BA pro řetězec w je uveden v tabulce 2.1.

index	1	2	3	4	5	6	7	8	9	10	11
w	a	b	r	a	k	a	d	a	b	r	a
β	0	0	0	1	0	1	0	1	2	3	4

Tabulka 2.1: Příklad BA pro slovo $w = abrakadabra$.

Myšlenka posunu vzoru v algoritmu MP spočívá v tom, že pokud již porovnaná část vzoru p' má nějaký border b, tak je možné zarovnat p tak, aby se prvních $|b|$ znaků vzoru znovu neporovnávalo. Porovnání je možné přeskočit, protože přesně tento podřetězec b již byl porovnán na dané pozici v prohledávaném řetězci, jen se b vyskytovalo na konci p'.

V algoritmu 1 je pseudokód algoritmu MP.

Algoritmus 1 MP pro vyhledávání v řetězcích

Vstup: $w, p \in \Sigma^+ : |w| = n, |p| = m, \beta'$
Výstup: Výskyty p v řetězci w

```
1: i ← 1; j ← 1
2: while i ≤ n − m + j do
3:     (i, j) ← MATCH(i, j, m);
4:     if j = m + 1 then
5:         output i − m
6:     if j = 1 then
7:         i ← i + 1
8:     else
9:         j ← β′[j]
```

Pseudokód byl převzat z přednášek předmětu MI-EVY [11]. Funkce match porovnává znaky textu a vzoru zleva doprava a vrátí první i, j takové, že $w[i] \neq p[j]$. Funkce β' je definována jako $\beta'[i] = \beta[i-1] + 1$.

2.2 Boyer-Moore-Sunday

Algoritmus Boyer-Moore-Sunday [12] je opět nejspíše méně známý než Boyer-Moore-Horsepool. Algoritmy se v tomto případě liší ve znaku, kterým se indexuje do tabulky posunů. Zatímco verze Horsepool indexuje podle posledního zarovnaného znaku v řetězci w se vzorem p, verze Sunday využívá znak ležící

těsně za posledním zarovnaným znakem. Rozdíl je také v hodnotách samotné tabulky posunů.

Pro tabulku posunů se v algoritmu Boyer-Moore-Sunday používá název *bad character shift* se zkratkou BCS. Jak již bylo řečeno, do tabulky se indexuje znakem z abecedy a na začátku je inicializována na $m + 1$ pro každý znak z abecedy, přičemž platí $|p| = m$. Následně se pro každý znak ve vzoru p upraví hodnota v BCS tak, aby odpovídala pozici nejpravějšího výskytu znaku počítaného od konce p. Například pokud by byl znak a poslední znak v p, pak $BCS[a] = 1$. Pokud by znak b byl prvním znakem p a nevyskytoval se nikde jinde, pak $BCS[b] = |p| = m$.

Opět pro uvedení nějakého příkladu nechť $\Sigma = \{a, b, d, k, r\}$ a $p = kabra$. BCS pro řetězec p nad abecenou Σ je uveden v tabulce 2.2.

znak	a	b	d	k	r
BCS	1	3	6	5	2

Tabulka 2.2: Příklad BCS pro $p = kabra$ nad abecenou $\Sigma = \{a, b, r, k, d\}$.

Průběh algoritmu je možné popsat následovně. Algoritmus zarovná vzor a porovná znaky. Porovnání může být prováděno jak ve stejném směru, v jakém se provádí posun vzoru, tak i v opačném směru. Ve chvíli, kdy dojde k neshodě znaků, se z BCS vrátí hodnota pro znak $w[i + m]$, kde i je pozice ve w, se kterou je zarovnán p. Vzor p je následně posunut o vrácenou hodnotu a celý proces se opakuje.

Výpočet BCS pro vzor p je popsán pseudokódem v algoritmu 2 a samotný algoritmus Boyer-Moore-Sunday je pak popsán pseudokódem v algoritmu 3. Algoritmy byly s úpravami převzaty z [2].

Algoritmus 2 Výpočet BCS pro algoritmus Boyer-Moore-Sunday

Vstup: $p \in \Sigma^+ : |p| = m$
Výstup: $BCS(p)$
1: **foreach** $x \in \Sigma$ **do** $BCS[x] = m + 1$
2: **for** $i \leftarrow 1$ **to** m **do**
3: $\quad BCS[p[i]] = m - i + 1$

2.3 Quantum Leap

Algoritmus Quantum Leap, dále zkracováno na QL, byl popsán v článku Quantum Leap Pattern Matching [13], ve kterém vychází z algoritmu Boyer-Moore-Sunday. Základní myšlenku QL lze ale použít i u jiných algoritmů a například implementovaný algoritmus pro vyhledávání ve stromech vychází z kombinace algoritmů MP a Boyer-Moore-Sunday. Popis algoritmu v této části bude ale vycházet pouze z algoritmu Boyer-Moore-Sunday.

Algoritmus 3 Boyer-Moore-Sunday pro vyhledávání v řetězcích

Vstup: $w, p \in \Sigma^+ : |w| = n, |p| = m$, tabulka posunů BCS
Výstup: Výskyty p v řetězci w

```
1:  i ← 1
2:  while i ≤ n − m + 1 do
3:      j ← 1
4:      while j ≤ m do
5:          if p[j] ≠ w[i + j − 1] then
6:              break
7:          j ← j + 1
8:      if j > m then
9:          output i
10:     if i + m > n then
11:         break
12:     i ← i + BCS[w[i + m]]
```

QL využívá toho, že v řetězcích lze vyhledávat zleva doprava stejně jako zprava doleva. Konkrétně pro úpravu algoritmu Boyer-Moore-Sunday, aby prohledával řetězce zprava doleva, je třeba upravit BCS tak, aby v ní byla pozice znaku zleva místo zprava. Kromě této změny je ještě třeba upravit indexy tak, aby se hledaný vzor posouval od konce textu na začátek a do BCS se indexovalo znakem před tím, který je zarovnaný se vzorem a ne znakem za vzorem.

Konkrétně v algoritmu 2 by došlo pouze ke dvěma změnám a to na řádku 2, kde by se prohodilo m a 1, aby i běželo od m do jedné. Na řádku 3 se pak do BCS přiřadí pouze i místo $m - i + 1$.

V algoritmu 3 budou následující změny:

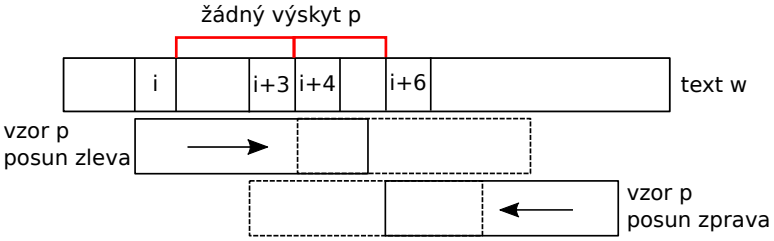
- řádek 1 se nahradí za $i \leftarrow n - m + 1$,
- na řádku 2 se podmínka změní na $i \geq 1$,
- podmínku na řádku 10 je třeba nahradit za $i \leq 1$,
- řádek 12 se nahradí za $i \leftarrow i - BCS[w[i - 1]]$.

Se vším, co bylo řečeno výše, je nyní možné vysvětlit myšlenku QL, která by se neformálně dala popsat následovně: Nechť $w, q \in \Sigma^+$ jsou text a hledaný vzor. K řešení problému je použit algoritmus Boyer-Moore-Sunday s tabulkou posunů BCS_1, který posouvá vzor zleva doprava, a který během svého běhu zarovná p s w na indexu i. Po porovnání znaků se algoritmus dotáže BCS_1 na velikost posunu, která pro jednoduchost příkladu vrátila číslo 4, což znamená, že p má být s w zarovnáno na indexu $i + 4$.

Nechť je ale v tuto chvíli běh algoritmu zastaven a nechť je spuštěn zcela nezávisle algoritmus Boyer-Moore-Sunday s tabulkou posunů BCS_2, který ale

porovnává vzor s textem od konce w a posouvá p doleva. Řekněme, že algoritmus zarovnal p s w na indexu $i + 6$. Po provedení porovnání vrátí BCS_2 hodnotu 3 a tedy další porovnání má být provedeno na indexu $i + 3$.

V tuto chvíli je ale bezpečné tvrdit, že mezi indexy i a $i + 6$ rozhodně není žádný výskyt p, protože první algoritmus tvrdí, že mezi i a $i + 4$ není žádný výskyt, ale druhý algoritmus tvrdí, že mezi $i + 3$ a $i + 6$ není žádný výskyt. Z předchozí věty plyne, že první algoritmus může udělat posun až na index $i + 6$. Vizualizace uvedeného příkladu je na obrázku 2.1.



Obrázek 2.1: Vizualizace příkladu myšlenky QL

Je zřejmé, že pouštět dva algoritmy s nadějí, že nastane výše zmíněná situace a ušetří se nějaké porovnání, není efektivní způsob vyhledávání. Zmíněnou myšlenku lze ale použít i tak, aby k takovémuto skoku došlo i vícekrát během běhu algoritmu. Algoritmus se může pokusit *odhadnout* číslo Z, pro které platí $i + Z - BCS_2[w[i + Z - 1]] < i + BCS_1[w[i + m]]$. Algoritmus poté může posunout vzor p o Z místo o hodnotu $BCS_1[w[i + m]]$. Je patrné, že čím větší je hodnota Z, tím větší bude část textu, která bude přeskočena, a tím je větší šance, že se ušetří nějaké porovnání. V algoritmu 4 je pseudokód popisující algoritmus QL.

Problém celého algoritmu je otázka, jakým způsobem odhadnout potřebné číslo Z tak, aby byla splněná zmíněná nerovnost a zároveň Z bylo co největší.

Nejlepší volbu Z lze zajistit tak, že algoritmus po každém porovnání projde zbytek textu a vybere tu pozici v textu, která vyhovuje oběma podmínkám. Je ale očividné, že tento přístup je neefektivní a hledání ideálního Z by celý algoritmus velmi zpomalilo. Je tedy třeba použít nějaký jiný přístup.

Dobrým začátkem je omezení rozsahu hodnoty Z. Už při lehčím zamyšlení by mělo být zřejmé, že záporné hodnoty Z jsou nesmyslné, protože by skok probíhal v opačném směru, než v jakém je třeba posouvat vzorek. Stejně tak je nesmyslná volba $Z = 0$, protože pak by se vzorek nikdy neposunul. Pro omezení hodnoty Z seshora je třeba se zamyslet nad fungováním algoritmu Boyer-Moore-Sunday a případem, kdy dochází k největšímu posunu vzoru.

K největšímu posunu dojde v případě, kdy se znak za vzorem v samotném vzoru nenachází. V takovém případě dojde k posunu o $m + 1$ a naprosto stejné tvrzení platí i pro posun zprava doleva. Z této úvahy je možné vyvodit, že nemá cenu volit Z větší než $2m + 1$, protože už při volbě $Z = 2m + 2$ nemůže

Algoritmus 4 QL pro vyhledávání v řetězcích

Vstup: $w, p \in \Sigma^+ : |w| = n, |p| = m$, tabulky posunů BCS_1 a BCS_2, celé číslo Z
Výstup: Výskyty p v řetězci w

```
1:  i ← 1
2:  while i ≤ n − m + 1 do
3:      j ← 1
4:      while j ≤ m do
5:          if p[j] ≠ w[i + j − 1] then
6:              break
7:          j ← j + 1
8:      if j > m then
9:          output i
10:     if i + m > n then
11:         break
12:     s ← ∞
13:     if i + Z ≤ n then
14:         s ← i + Z − BCS2[w[i + Z − 1]]
15:     if s < i + BCS1[w[i + m]] then
16:         i ← i + Z
17:     else
18:         i ← i + BCS1[w[i + m]]
```

nikdy platit podmínka $i + Z - BCS_2[w[i + Z - 1]] < i + BCS_1[w[i + m]]$, protože $i + (2m + 2) - (m + 1) = i + m + 1 \not< i + m + 1$. Volba Z je tedy omezená na interval $\langle 1, 2m + 1\rangle$.

Jak už bylo řečeno výše, čím větší Z se zvolí, tím delší budou skoky a ušetří se porovnání, nabízí se tedy otázka, proč nenastavit $Z = 2m + 1$. Problém je opět první podmínka. I když je snaha Z maximalizovat, u první podmínky s rostoucím Z klesá pravděpodobnost jejího splnění. Například ve zmíněném případě $Z = 2m + 1$ bude podmínka platit jen v případě, kdy se oba znaky $w[i + m]$ a $w[i + Z - 1]$ nebudou nacházet ve vzoru p. Je tedy vhodnější volit menší hodnoty Z, které častěji uspokojí první podmínku, a i když bude docházet pouze k menším skokům, bude k nim docházet častěji.

Při volbě příliš malého Z je pak pravděpodobnost skoku vysoká, skoky jsou ale krátké a výpočetní čas strávený režií kolem provádění skoku bude větší než čas ušetřený provedením skoku. Pro příliš malé Z je dokonce možné, že $i+Z < i+BCS_1[w[i+m]]$, což znamená, že skok zkrátí posun vzoru a provádění skoků naopak celý algoritmus zpomalí. Toto je možné ošetřit přidáním výrazu $i+Z > i+BCS_1[w[i+m]]$ do podmínky v algoritmu 4 na řádku 15. Výpočetní čas potřebný pro vyhodnocení této podmínky ale opět prodlouží dobu běhu.

Bohužel v současné době nejsou žádné vzorce nebo metody, které by pro

zadaný text a vzor vrátily ideální Z a volba hodnoty se dělá heuristicky. Hlavní náplní této práce je právě vyzkoušet několik heuristik pro volbu Z v QL pro vyhledávání vzorů ve stromech a experimentálně vyhodnotit jejich kvalitu. V případě zájmu čtenáře o téma hledání tohoto Z v řetězcích je možné si o tom přečíst v již zmíněném článku Quantum Leap Pattern Matching [13] a z něj vycházejících článků.

Kapitola 3

Algoritmy pro vyhledávání v linearizovaných stromech

Jak již bylo několikrát řečeno, tato práce se zabývá vyhledáváním vzorů ve stromech. Konkrétně se pracuje s prefixovou notací linearizovaných stromů, díky čemuž je možné s modifikacemi využít algoritmy pro vyhledávání v řetězcích. Na konci kapitoly 1 bylo řečeno, že kvůli proměnným $*$ ve stromových vzorech je problém vyhledávání ve stromech obecnějším problémem než obyčejné vyhledávání v řetězcích.

Jelikož symbol $*$ může reprezentovat libovolný podstrom, dá se $*$ v prefixové notaci chápat jako *mezera* libovolné velikosti splňující navíc podmínku, že tato mezera je lineárním zápisem úplného podstromu. Tento fakt komplikuje porovnávání znaků, protože při výskytu $*$ ve vzoru p je třeba přeskočit celý úplný podstrom v prohledávaném stromu w a velikost tohoto skoku může být vždy jiná.

Druhý problém, který způsobuje znak $*$, je při posunu vzoru v algoritmu Boyer-Moore-Sunday. V řetězcové doméně je možné vzhledem k fixní délce vzoru použít tabulky BCS využívané algoritmem Boyer-Moore-Sunday pro posun oběma směry za předpokladu, že pro oba směry je vlastní tabulka. Ve stromových vzorech způsobuje znak $*$ proměnnou délku stromu, se kterým se vzor překryje, čímž je znemožněno využít ekvivalentu BCS pro posun oběma směry najednou.

Pro prefixovou notaci využívanou v této práci není možné použít algoritmus Boyer-Moore-Sunday s posunem vzorů zleva doprava. Pro nějaké jiné notace by naopak nebylo možné použít algoritmus s posunem vzorů zprava doleva.

3.1 Podstromová skoková tabulka

Zmíněný problém s porovnáváním znaků stromu, které odpovídají zanku $*$, řeší tabulka anglicky zvaná *Subtree Jump Table* poprvé uvedená v [14]. Český název podstromová skoková tabulka je převzat z [2]. V této práci se pro tabulku bude používat zkratka *SJT*.

Definice 16 *Nechť t je strom a $pref(t) = a_1a_2 \dots a_n$ je jeho prefixová notace.* Podstromová skoková tabulka $sjt(pref(t))$ *je mapování z množiny $\{1 \dots n\}$ do množiny $\{2 \dots n+1\}$, pro které platí, že pokud $a_ia_{i+1} \dots a_{j-1}$ je prefixová notace podstromu stromu t, pak $sjt(pref(t))[i] = j, 1 \leq i < j \leq n+1$. [1]*

Neformálně SJT obsahuje záznam pro každý podstrom r stromu t. Záznam pro podstrom r je na pozici jeho kořene v $pref(t)$ a obsahuje index posledního znaku podstromu r v $pref(t)$ zvětšený o jedna. Příklad SJT je uveden v tabulce 3.1.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$pref(t)$	a_2	a_2	a_2	a_0	a_2	b_1	b_0	a_0	a_0	a_2	a_2	a_0	a_2	b_1	b_0	a_0	a_0
$sjt(pref(t))$	18	10	9	5	9	8	8	9	10	18	17	13	17	16	16	17	18

Tabulka 3.1: Příklad SJT [1]

Tabulka $sjt(t)$ má stejnou velikost jako $pref(t)$ a je možné ji zkonstruovat v čase $\mathcal{O}(n)$ [14]. SJT je možné zkonstruovat již během linearizace stromu [2]. V algoritmu 5 je pseudokód pro samostatnou konstrukci SJT.

Algoritmus 5 ConstructSJT - Konstrukce SJT pro prefixovou notaci stromu

Vstup: $pref(t)$, reference na prázdnou $sjt(pref(t))$, reference na proměnnou $index$ implicitně 1
Výstup: $sjt(pref(t))$

```
1: myIndex ← index
2: for i ← 1 to arita pref(t)[myIndex] do
3:     index ← index + 1
4:     ConstructSJT(pref(t), sjt(pref(t)), index)
5: sjt(pref(t))[myIndex] ← index + 1
```

3.2 Boyer-Moore-Sunday pro vyhledávání ve stromech

S definicí SJT je možné vysvětlil fungování algoritmu Boyer-Moore-Sunday pro vyhledávání vzorů ve stromech v prefixové notaci. Jak bylo řečeno na začátku této kapitoly, tak pro vyhledávání ve stromech není možné použít verzi

algoritmu Boyer-Moore-Sunday, který posunuje vzor zleva doprava. Proč tomu tak je je uvedeno na konci této části této kapitoly. Nejprve ale bude vysvětleno fungování algoritmu s posunem vzoru zprava doleva.

Neformálně lze říci, že algoritmus funguje stejně jako verze pro vyhledávání v řetězcích se dvěma rozdíly.

První rozdíl je při porovnávání znaků, během kterého se algoritmus při výskytu $*$ ve vzoru podívá do SJT, ze které získá index, od kterého má porovnávat další znaky vzoru.

Druhá změna je v tabulce posunů BCS. V částech 2.2 a 2.3 bylo zmíněno, že v BCS je pro každý znak z abecedy číslo, které je minimem z $|p| + 1$ a pozice daného znaku v p zleva. V algoritmu pro vyhledávání ve stromech se mezi položky, z nichž se bere minimum, přidá ještě třetí a to pozice nejlevějšího znaku $*$. Z této skutečnosti bohužel plyne, že pokud se $*$ vyskytuje někde na začátku vzoru, pak budou posuny vždy relativně krátké. Pokud například je hvězda první znak ve vzoru, bude v celé BCS pro každý znak hodnota jedna.

V algoritmu 7 je uveden pseudokód pro tuto verzi algoritmu.

Algoritmus 6 popisuje tvorbu BCS. Změny oproti procházení zleva doprava byly popsány v části 2.3 přibude ovšem zmíněná třetí podmínka pro výpočet BCS, tedy pozice nejlevějšího znaku $*$.

Algoritmus 6 Výpočet BCS pro algoritmus 7

Vstup: $pref(p) : |pref(p)| = m$, kde p je stromový vzor.
Výstup: $BCS(pref(p))$

```
1: index ← m + 1
2: if pref(p) obsahuje * then
3:     index ← pozice prvního znaku * zleva
4: foreach x ∈ Σ do BCS[x] = index
5: for i ← 1 to index − 1 do
6:     BCS[p[i]] = i
```

Nyní, když byl popsán algoritmus Boyer-Moore-Sunday s posunem vzoru zprava doleva, je vhodné vysvětlit, proč není možné použít algoritmus s průchodem zleva doprava. Důvodem je proměnlivá délka vzoru, kvůli které není možné zarovnat znak ve stromu, který se nachází za vzorem, s nejpravějším výskytem tohoto znaku ve vzoru.

Nabízí se možnost použít znak, který se nachází za vzorem tak, jak je momentálně zarovnaný. Tato možnost ale není správná, neboť po posunu může být vzor například mnohem kratší a k zarovnání požadovaných znaků nedojde. Protipříkladem k této myšlence je strom $a_2a_2a_1a_0a_0a_1b_0$ a vzor $a_2 * a_0$. Při prvním zarovnání dojde k nerovnosti znaků a_0 a a_1. Znak za vzorem je b_0 a dle BCS by tedy mělo dojít k posunu o velikosti 2. Je ale vidět, že v tu chvíli byl přeskočem výskyt na indexu 2.

Další možností je ignorovat proměnlivou délku vzoru a uvažovat znak na

Algoritmus 7 Boyer-Moore-Sunday pro vyhledávání ve stromech procházející $pref(t)$ zprava doleva

Vstup: $pref(t), pref(p) : |pref(t)| = n, |pref(p)| = m$, kde t je strom a p je stromový vzor, BCS, SJT pro pref(t)
Výstup: Výskyty p v t

```
1:  i ← n − m + 1
2:  while i ≥ 1 do
3:      j ← 1
4:      indexT ← i
5:      while j ≤ m do
6:          if pref(p)[j] ≠ pref(t)[indexT] then
7:              if pref(p) = ∗ then
8:                  indexT ← SJT[indexT] − 1
9:              else
10:                 break
11:         j ← j + 1; indexT ← indexT + 1
12:     if j > m then
13:         output i
14:     if i ≤ 1 then
15:         break
16:     i ← i − BCS[w[i − 1]]
```

indexu $i + m$, kde m je velikost vzoru. Jinými slovy brát znak $*$ tak, že má délku 1. Tento přístup je ale také nesprávný a protipříkladem může být strom $a_3a_2a_1b_0a_0a_0a_0$ a vzor $a_2 * a_0$. V tomto případě dojde k neshodě na prvním znaku. Znak na indexu $i + m = 1 + 3 = 4$ je b_0 a dle BCS by tedy mělo dojít k posunu o dva znaky. V tu chvíli je ale opět přeskočen výskyt vzoru na indexu 2.

3.3 BA pro linearizované stromy

Pro potřeby definice algoritmu QL je v práci uvedena definice algoritmu MP upraveného pro vyhledávání stromových vzorů. Sekce 3.4 se věnuje algoritmu MP pro vyhledávání ve stromech, nejprve je ale třeba zavést pojem border array pro stromové vzory, čemuž se věnuje tato část této kapitoly.

Tato a následující část této kapitoly jsou převzaty z článku Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array [1], ve kterém byly algoritmus a zmíněné border array uvedeny. Pokud není řečeno jinak, vše v těchto dvou částech je ze zmíněného článku.

Border array pro stromové vzory je složitější struktura než obyčejné border array pro řetězce a je nejprve třeba zavést několik pojmů, které se v definici objevují. Prvním pojmem je *kontrolní suma arity*.

Definice 17 *Nechť $w = a_1 a_2 \dots a_n, n \geq 1$ je řetězec nad ohodnocenou abecedou.* Kontrolní suma arity $ac(w) = \sum_{i=1}^{n} arita(a_i) - n + 1$.

Tvrzení 1 *Nechť t je strom a w je podřetězcem $pref(t)$. Platí, že w odpovídá prefixové notaci nějaké podstromu t právě když $ac(w) = 0$ a $\forall w_1 : ac(w_1) \geq 1$, kde w_1 je vlastní předponou w.[15].*

Tvrzení 1 dává formální způsob, jakým pro libovolný řetězec w nad ohodnocenou abecedou rozhodnout, zda $w = pref(t)$ pro nějaký strom t. Důkazem toho, že druhá část podmínky $\forall w_1 : ac(w_1) \geq 1$ je nutná, může být $w = a_0 a_1$, pro který platí $ac(w) = 0$, ale na první pohled je zřejmé, že se nejedná o strom.

Protože to bude využito v následující definici, je třeba poukázat na to, co pro nějaký řetězec w znamená, když $\forall i \in \{1, \dots, |w|\} : ac(w[1:i]) \geq 1$. Tedy pokud pro všechny předpony řetězce platí, že ac je větší rovno jedné. Pokud pro řetězec platí tato podmínka, pak se na daný řetězec dá dívat jako na *nedokončený* zápis nějakého stromu, respektive ho lze chápat jako zápis stromu, kterému *chybí* nějaké větve.

Před uvedením definice border array pro stromové vzory je nejprve třeba zavést relaci, která je v článku [1] pojmenovaná *matches*. V této práci se pro tuto relaci bude používat pojem *shoda*, a pokud dva řetězce budou v relaci, tak se o nich bude tvrdit, že se *shodují*.

Definice 18 *O dvou ohodnocených řetězcích w_1, w_2 se řekne, že se shodují, pokud:*

- $w_1 = x w_1', w_2 = x w_2'$ *a platí, že* w_1' *se shoduje s* w_2',
- $w_1 = *w_1', w_2 = *w_2'$ *a platí, že* w_1' *se shoduje s* w_2',
- $w_1 = x_1 \dots x_m w_1', w_2 = *w_2'$ *a platí, že* w_1' *se shoduje s* w_2' *a zároveň* $ac(x_1 \dots x_m) = 0$ *a zároveň* $\forall k, 1 \leq k < m : ac(x_1 \dots x_k) \geq 1$,
- $w_1 = *w_1', w_2 = x_1 \dots x_m w_2'$ *a platí, že* w_1' *se shoduje s* w_2' *a zároveň* $ac(x_1 \dots x_m) = 0$ *a zároveň* $\forall k, 1 \leq k < m : ac(x_1 \dots x_k) \geq 1$,
- $w_1 = *w_1', w_2 = x_1 \dots x_m$ *a platí, že* $\forall k, 1 \leq k \leq m : ac(x_1 \dots x_k) \geq 1$,
- $w_1 = \varepsilon$ *nebo* $w_2 = \varepsilon$.

Neformální vysvětlení jednotlivých bodů definice je následovné. První dva body říkají, že pokud se před řetězce, které se shodují, přidá stejný znak, tak se i poté oba řetězce stále shodují. Třetí a čtvrtý bod vyjadřuje situaci, kdy se před jeden z řetězců přidá úplný podstrom, který se v druhém řetězci *pokryje* znakem $*$. Poslední bod pak říká, že jakýkoliv řetězec se shoduje s prázdným řetězcem.

Nejsložitější na pochopení je nejspíše pátý bod. Tento bod tvrdí, že i pokud je druhý řetězec zápis stromu, kterému chybí nějaké větve, tak i přesto se

shoduje s řetězcem, který začíná na $*$. V druhém řetězci tedy nemusí být zápis posledního stromu kompletní.

Příklad 1 *Nechť $w_1 = a_2 * a_0, w_2 = a_2a_2a_1$ jsou řetězce nad ohodnocenou abecedou. Řetězec w_1 se shoduje s řetězcem w_2 a z definice lze k tomuto závěru dojít následovně.*

Za použití prvního bodu je možné z řetězců odstranit první a_2, aniž by se změnila jejich shoda a zbydou řetězce $w_1' = *a_0, w_2' = a_2a_1$. Dle pátého bodu je možné odstranit z prvního řetězce $*$ a z druhého nekompletní zápis stromu beze změny shody a zbydou $w_1'' = a_0, w_2'' = \varepsilon$. Z šestého bodu plyne, že řetězce w_1'', w_2'' se shodují, z čehož plyne, že i původní w_1, w_2 se shodují.

Nyní je možné uvést definici border array pro stromové vzory.

Definice 19 *Nechť p je stromový vzor. a $|pref(p)| = m$.* Border array pro stromový vzor *$ba(pref(p))$ je pole délky m reprezentující délky nejdelších borderů všech předpon $pref(p)$.*

$ba(pref(p))[i] = i - min(\{j : pref(p)$ se shoduje s $pref(p)[(j+1) : i] \wedge 1 \leq j \leq i \leq m\})$

Je vhodné poznamenat, že pro $i = j$ je $pref(p)[(j+1) : i] = \varepsilon$, které se shoduje s jakýmkoliv řetězcem. Minimum se tedy vždy vybírá alespoň z jednoprvkové množiny.

Příklad 2 *Nechť $pref(p) = a_2a_2 * a_2b_1 * a_0a_0$, kde p je stromový vzor. K výpočtu $ba(pref(p))[5]$ je dle definice 19 třeba vypočítat shody $pref(p)$ a $pref(p)[(j+1) : 5]$ pro $1 \leq j \leq 5$. Nejmenší j, pro které se $pref(p)$ shoduje s $pref(p)[(j+1) : 5]$ je 2 a tedy $ba(pref(p))[5] = 5 - 2 = 3$. Postup výpočtu je uveden v tabulce 3.2.*

index	1	2	3	4	5	6	7	8			
$pref(t)$	a_2	a_2	$*$	a_2	b_1	$*$	a_0	a_0			
$pref(t)[2:5]$	a_2	$*$	-	-	-	-	-	a_2			neshoda na pozici 8
$pref(t)[3:5]$	$*$	-	-	-	-	-	-	-	a_2	b_1	shoda
$pref(t)[4:5]$	a_2	b_1									neshoda na pozici 2
$pref(t)[5:5]$	b_1										neshoda na pozici 1
$pref(t)[6:5]$											shoda, protože $pref(t)[6:5] = \varepsilon$

Tabulka 3.2: Výpočet shody $pref(p)$ a $pref(p)[(j+1) : 5]$, pro $1 \leq j \leq 5$. [1]

Naivní výpočet BA dle definice vyžaduje $\mathcal{O}(m^3)$ času. V několikrát zmiňovaném článku [1] je uveden algoritmus řešící tento problém v kvadratickém čase. Na závěr této části kapitoly je tedy v algoritmu 8 popsán pseudokód pro výpočet BA pro stromové vzory.

Algoritmus 8 Výpočet BA pro stromový vzor

Vstup: $pref(p) : |pref(p)| = m$, kde p je stromový vzor, SJT pro pref(p)
Výstup: BA pro $pref(p)$

```
1: for i ← 1 to m do mins[i] ← i
2: for ofs ← 1 to m do
3:     i ← 1
4:     j ← ofs + 1
5:     while true do
6:         if i > m then
7:             while j ≤ m do
8:                 mins[j] ← min(mins[j], ofs)
9:                 j ← j + 1
10:            break
11:        else if j > m then
12:            break
13:        else if pref(p)[i] = pref(p)[j] then
14:            mins[j] ← min(mins[j], ofs)
15:            i ← i + 1
16:            j ← j + 1
17:        else if pref(p)[i] = ∗ ∨ pref(p)[j] = ∗ then
18:            for k ← j to SJT[j] − 1 do
19:                mins[k] ← min(mins[k], ofs)
20:            i ← SJT[i]
21:            j ← SJT[j]
22:        else
23:            break
24: BA[1] ← 0
25: for i ← 2 to m do BA[i] ← i − mins[i]
```

3.4 MP pro vyhledávání ve stromech

S definicí BA pro stromové vzory je možné přejít k algoritmu Morris-Pratt, který ke svému fungování BA vyžaduje. V algoritmu 9 je pseudokód MP pro vyhledávání stromových vzorů v linearizovaných stromech. Jak bylo řečeno v předchozí části této kapitoly, algoritmus je převzat z [1].

Časová složitost algoritmu samotného algoritmu je v obecném případě $\mathcal{O}(n \cdot m)$ a v případě, že se ve vzoru nevyskytuje žádný znak $*$ $\Theta(n)$. Je ale třeba zmínit, že výpočet BA je kvadratický, a pokud není k dispozici předpočítané BA, je jeho výpočet asymptoticky nejsložitější částí algoritmu. Typicky ale platí, že n je mnohonásobně větší než m a výpočet BA pro vzor tedy nebude mít znatelný vliv na celkovou dobu běhu algoritmu.

Algoritmus 9 MP pro vyhledávání vzorů v linearizovaných stromech

Vstup: $pref(t), pref(p) : |pref(t)| = n, |pref(p)| = m$, kde t je strom a p je stromový vzor, SJT pro $pref(t)$ BA pro $pref(p)$

Výstup: Výskyty p v t

```
1: Spos ← min({j : pref(p)[j] = ∗ ∧ 1 ≤ j ≤ m})
2: shift[1] ← 1
3: for i ← 2 to m + 1 do shift[i] ← i − BA[i − 1] − 1
4: i ← 0
5: j ← 1
6: while i ≤ n − m do
7:     offset ← i + j
8:     while j ≤ m and offset ≤ n do
9:         if pref(p)[j] = pref(t)[offset] then
10:            j ← j + 1
11:            offset ← offset + 1
12:        else if pref(p)[j] = ∗ then
13:            offset ← SJT[offset]
14:            j ← j + 1
15:        else
16:            break
17:    if j > m then output i + 1
18:    i ← i + shift[j]
19:    j ← max(1, min(Spos, j) − shift[j])
```

Příklad 3 *Nechť* $pref(t) = a_2a_2a_2a_0a_2b_1b_0a_0a_0a_2a_2a_0a_2b_1b_0a_0a_0$ *a* $pref(p) = a_2a_2 * a_2b_1 * a_0a_0$*, kde t je strom p je stromový vzor. V tabulce 3.3 je uvedeno BA pro* $pref(p)$ *a z něj odvozená tabulka posunů. V tabulce 3.4 je znázorněn běh algoritmu.*

index	1	2	3	4	5	6	7	8	9
$pref(p)$	a_2	a_2	$*$	a_2	b_1	$*$	a_0	a_0	
$ba(pref(p))$	0	1	2	2	3	4	5	6	
$tabulkaposun$	1	1	1	1	2	2	2	2	2

Tabulka 3.3: BA pro $pref(p)$ a z něj odvozená tabulka posunů [1]

3.5 QL pro vyhledávání ve stromech

Myšlenka QL pro vyhledávání stromových vzorů ve stromech je totožná s tím, co bylo uvedeno v sekci 2.3. Rozdíl oproti algoritmu uvedené ve zmíněné sekci je v tom, že pro vyhledávání zleva doprava je použitý jiný algoritmus, než

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$pref(t)$	a_2	a_2	a_2	a_0	a_2	b_1	b_0	a_0	a_0	a_2	a_2	a_0	a_2	b_1	b_0	a_0	a_0
$sjt(pref(t))$	18	10	9	5	9	8	8	9	10	18	17	13	17	16	16	17	18
1	a_2	a_2	$*$	-	-	-	-	-	a_2								
2 - výskyt p		a_2	a_2	$*$	a_2	b_1	$*$	a_0	a_0								
3				a_2													
4					a_2	a_2											
5						a_2											
6							a_2										
7								a_2									
8									a_2								
9 - výskyt p										a_2	a_2	$*$	a_2	b_1	$*$	a_0	a_0

Tabulka 3.4: Běh algoritmu na stromu a vzoru z příkladu 3 [1]

pro vyhledávání zprava doleva. Konkrétně pro vyhledávání zleva doprava je použit algoritmus vycházející z MP, který byl popsán v předchozích částech této kapitoly 3.3 a 3.4. Algoritmus, který se používá pro určení posunu zprava doleva po provedení skoku, je Boyer-Moore-Sunday popsaný ve druhé části této kapitoly 3.2.

Neformálně je možné o algoritmu říci, že funguje totožně jako MP až do chvíle, kdy se má provést posun vzoru. V tu chvíli se průběh MP zastaví a provede se *kvantový* skok velikosti Z. Algoritmus se dotáže BCS z algoritmu Boyer-Moore-Sunday na velikost posunu. Porovná se, jestli je navrácená hodnota nižší než posun, který byl získán z tabulky posunů algoritmu MP. Pokud je hodnota nižší, nastaví se v algoritmu 9 index i na řádce 18 na hodnotu $i+Z$ místo uvedeného $i + shift[j]$. Jelikož se tímto skokem ztratí informace o borderu, je třeba nastavit index j na řádce 19 vždy na 1. Pokud se skok neprovedl, algoritmus pokračuje naprosto stejně, jak je uvedeno v algoritmu 9.

I přes minimální množství změn je v algoritmu 10 uveden celý pseudokód algoritmu QL pro vyhledávání stromových vzorů v linearizovaných stromech.

V tabulce 3.5 je stejný příklad jako v případě tabulky 3.4 avšak pro algoritmus QL pro $Z = 3$. BCS je v tomto případě pro a_2 rovno jedné a pro zbytek znaků z abecedy třem. Jak je z příkladu vidět, tak se provedlo pouze šest zarovnání místo devíti, jak tomu bylo v algoritmu MP.

V prvním posunu se skok neprovedl, protože $BCS[pref(t)[i + Z - 1] = a_2] = 1$ a tabulka posunů z algoritmus MP tvrdí, že posun má být velikosti jedna.

Ve druhém kroku se skok provedl, protože $BCS[pref(t)[i+Z-1] = a_0] = 3$ a posun z MP měl být roven dvěma. Dle MP se tedy další porovnání mělo dělat na indexu $i = 4$, ale z BCS vyšlo, že na $i = 5$ je možné skočit až na pozici $i = 2$.

Ve třetím porovnání došlo k neshodě znaků na druhé pozici ve vzoru a posun dle MP má být velikost jedna. Dle BCS je ale možné provést posun z indexu $i = 8$ až na $i = 5$, skok se tedy provedl.

Po čtvrtém porovnání se skok neprovedl, protože $i + Z - 1 = 10$ a znak

Algoritmus 10 QL pro vyhledávání vzorů v linearizovaných stromech

Vstup: $pref(t), pref(p) : |pref(t)| = n, |pref(p)| = m$, kde t je strom a p je stromový vzor, SJT pro $pref(t)$ BA pro $pref(p)$, BCS z algoritmu 6, číslo Z

Výstup: Výskyty p v t

```
1:  Spos ← min({j : pref(p)[j] = * ∧ 1 ≤ j ≤ m})
2:  shift[1] ← 1
3:  for i ← 2 to m + 1 do shift[i] ← i − BA[i − 1] − 1
4:  i ← 0
5:  j ← 1
6:  while i ≤ n − m do
7:      offset ← i + j
8:      while j ≤ m and offset ≤ n do
9:          if pref(p)[j] = pref(t)[offset] then
10:             j ← j + 1
11:             offset ← offset + 1
12:         else if pref(p)[j] = * then
13:             offset ← SJT[offset]
14:             j ← j + 1
15:         else
16:             break
17:     if j > m then output i + 1
18:     if i + Z < n and i + Z − BCS[pref(t)[i + Z − 1]] < i + shift[j] then
19:         i ← i + Z
20:         j ← 1
21:     else
22:         i ← i + shift[j]
23:         j ← max(1, min(Spos, j) − shift[j])
```

na indexu 10 je a_2. Posuny doprava z MP a doleva dle BCS se tedy nepřekříží a skok není možné provést. Poslední skok se také neprovede, protože se do BCS opět indexuje znakem a_2, protože $i + Z - 1 = 11$.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$pref(t)$	a_2	a_2	a_2	a_0	a_2	b_1	b_0	a_0	a_0	a_2	a_2	a_0	a_2	b_1	b_0	a_0	a_0
$sjt(pref(t))$	18	10	9	5	9	8	8	9	10	18	17	13	17	16	16	17	18
1	a_2	a_2	$*$	-	-	-	-	-	a_2								
2 - výskyt p		a_2	a_2	$*$	a_2	b_1	$*$	a_0	a_0								
$3 - SKOK$					a_2	a_2											
$4 - SKOK$								a_2									
5									a_2								
6 - výskyt p										a_2	a_2	$*$	a_2	b_1	$*$	a_0	a_0

Tabulka 3.5: Běh algoritmu QL pro $Z = 3$ na stromu a vzoru z příkladu 3

Toto je základní verze QL s myšlenkou skoku, jak bylo řečeno v sekci 2.3. Jelikož cílem práce bylo najít co nejrychlejší verzi algoritmu, byly kromě různých hodnot Z na této základní verzi testovány i jiné přístupy k provádění skoků nebo práce s hodnotou Z.

3.5.1 QL s provedením skoku až po posunu

Jednou z myšlenek bylo nejprve provést posun podle algoritmu MP a až z této pozice provést skok o Z. V této verzi je třeba se opět zamyslet nad smysluplnými hodnotami Z.

V sekci 2.3 bylo řečeno, že pro Z nemá smysl volit hodnoty mimo interval $\langle 1, 2m + 1\rangle$. Pokud ale nejprve dojde k posunu a až následně dojde k pokusu o skok, pak je nejspíše patrné, že nemá smysl volit hodnoty Z vyšší než m, protože nejvyšší hodnotu, kterou BCS může vrátit je $m + 1$.

Na tuto verzi je možné také pohlížet z původního pohledu algoritmu tak, že před provedením skoku se algoritmus podívá, jak velký je posun zleva doprava a následně se hodnota Z zvolí dynamicky. Pokud je posun malý, tak je Z zvoleno menší, aby se zvedla pravděpodobnost skoku. Pokud je ale posun větší, pak je Z zvoleno větší, aby se případně provedl větší skok. Z výsledků, které jsou uvedeny v kapitole 5, pak vyplývá, že tato verze si v porovnání s ostatními vede velmi dobře.

K úpravě algoritmu 10 na výše zmíněnou verzi je třeba upravit řádky 19 až 23 tak, jak je popsáno v algoritmu 11.

Algoritmus 11 Úprava QL, aby ke skokům docházelo až po posunu vzoru

```
18: i ← i + shift[j]
19: j ← max(1, min(Spos, j) − shift[j])
20: if i + Z < n and i + Z − BCS[pref(t)[i + Z − 1]] < i then
21:     i ← i + Z
22:     j ← 1
```

3.5.2 QL s dvourozměrnou tabulkou posunů

K další verzi algoritmu vedla myšlenka převést vyhodnocení podmínky na řádku 18 do fáze předzpracování. Tohoto lze dosáhnout tak, že místo jednorozměrné tabulky posunů, v algoritmu 10 se jménem *shift*, bude tabulka dvourozměrná o velikost $m \cdot |\Sigma|$.

Neformálně řečeno by se pro každý znak abecedy vytvořila vlastní tabulka posunů a ve chvíli, kdy by mělo dojít k posunu, se nejprve vybere ta tabulka, která odpovídá znaku na pozici $pref(t)[i + Z - 1]$ a následně se s vybranou tabulkou pracuje jako v původní verzi algoritmu.

Hlavní rozdíl je v tom, že v této tabulce by do posunu byly započteny i případné skoky a nebylo by třeba podmínky na řádku 18. Cenou za odstranění

této podmínky je ale dvojí adresace a fakt, že z tabulky nelze jednoduše říci, jestli ke skoku došlo nebo nedošlo, kvůli čemuž musí být index j vždy nastaven na 1, což i nadále zpomalí celý algoritmus.

3.5.3 QL s dynamicky měnícím se Z

K této verzi vedl fakt, že volba Z závisí na stromu a vzoru a pro různé stromy a vzory jsou vyhovující různé volby Z. Myšlenka této verze algoritmu byla, že pokud je prohledávaný strom velký, pak by si algoritmus sám mohl najít vyhovující Z v průběhu prohledávání.

Určitě by bylo možné najít nějaké složitější metody s lepšími výsledky, ale v této práci byla použita verze algoritmu, ve které Z začíná na velikosti vzoru, a pokud se skok povede, pak je Z zvětšeno o jedna. Naopak pokud ke skoku nedojde, pak je od Z odečtena jednička. Změna v kódu je opět minimální, protože stačí mezi řádky 20 a 21 přidat inkrementaci Z a mezi řádky 21 a 22 dekrementaci.

3.5.4 QL se dvěma tabulkami posunů

Poslední verze algoritmu kombinovala dvě výše zmíněné myšlenky a to konkrétně z částí 3.5.1 a 3.5.2. Z první zmíněné části si bere myšlenku provádění posunu před samotným skokem a z druhé snahu odstranit podmínku z řádku 18.

V tabulce posunů v algoritmu 10 pojmenovanou $shift$ v této verzi nedojde k žádné změně, ale hlavní změna bude v tabulce BCS. Ve verzi algoritmu uvedené v části 3.5.1 bylo řečeno, že algoritmus nejprve posune vzor doprava, pak provede skok a zkontroluje, jestli $i+Z-BCS[pref(t)[i+Z-1]] < i$. Hodnotu i je možné odečíst z obou stran nerovnice a přičtením $BCS[pref(t)[i+Z-1]]$ na obě strany je výsledný vzoreček $Z < BCS[pref(t)[i+Z-1]]$. Jak je vidět, tak pokud má dojít ke skoku, pak musí být hodnota v BCS větší než Z. Z tohoto plyne, že stačí projít celé BCS a pokud je hodnota na dané pozici menší nebo rovna Z, je jisté, že se skok neprovede a hodnota může být nahrazena nulou. Pokud je hodnota větší, pak se skok provede, takže hodnota může být nahrazena hodnotou Z.

Ve výsledném algoritmu je pak možné odstranit podmínku na řádku 18 a pouze přičíst BCS k indexu i. Nevýhodou tohoto postupu je opět ztráta informace o borderu a je tedy nutné nastavovat index j vždy na 1. K tomuto je vhodné poznamenat, že informaci lze získat jednoduchou kontrolou BCS. Pokud je hodnota v BCS rovná Z, pak je třeba nastavit j rovno jedné, pokud je ale hodnota v BCS 0, pak je možné určit hodnotu j tak, jak tomu bylo v algoritmu MP 9 tedy $j \leftarrow max(1, min(Spos, j) - shift[j])$. Cena za tuto kontrolu je ale přidání podmínky a smyslem celé této verze je podmínku odstranit.

Pseudokód pro upravenou část algoritmu 10 je v algoritmu 12.

Algoritmus 12 Úprava QL s BCS obsahující jen hodnoty z množiny $\{0, Z\}$

18: $i \leftarrow i + shift[j]$
19: $j \leftarrow 1$
20: **if** $i + Z < n$ **then**
21: $\quad i \leftarrow i + BCS[pref(t)[i + Z - 1]]$

Kapitola 4

Implementace algoritmu Quantum Leap pro vyhledávání vzorů ve stromech

Pro implementaci algoritmů byl zvolen jazyk Java. Hlavním důvodem volby jazyka je existence souboru nástrojů Forest FIRE.

4.1 Soubor nástrojů Forest FIRE

Soubor nástrojů Forest FIRE byl poprvé uveden v [16]. V souboru nástrojů je implementováno velké množství algoritmů pro vyhledávání vzorů ve stromech a poskytuje nemalé množství stromů a stromových vzorů, na kterých je možné algoritmy testovat a měřit rychlost jejich běhu.

Soubor nástrojů je v průběhu měření schopný sledovat i další statistiky, jakým je například počet indexů, pro které se algoritmus pokusil ověřit existenci výskytu vzoru. Součástí této práce bylo přidání nové statistiky, která pro algoritmus Quantum Leap počítá počet provedených *kvantových* skoků. Forest FIRE tedy dává velmi dobrý způsob, jak experimentálně vyhodnotit kvalitu implementovaných algoritmů.

V souboru nástrojů jsou již implementovány některé algoritmy počítající pomocné struktury, jako například algoritmus pro výpočet SJT. Právě zmíněný již existující algoritmus pro výpočet SJT byl v implementaci využit.

Za zmínku stojí, že Forest FIRE je hlavní důvod, proč jsou v této práci řetězce a všechna pole indexována od jedné, neboť v tomto souboru nástrojů je vždy prvek na indexu 0 inicializován na nesmyslnou hodnotu nebo na prázdnou referenci a všechny algoritmy prvek ignorují. S tímto faktem je třeba počítat, protože skutečná velikost pole s m prvky je $m + 1$.

4.2 Implementace QL

Algoritmus je reprezentován třídou *PrefixQLMatcher*, která implementuje rozhraní *IMatcher*. Při tvoření instance *PrefixQLMatcher* se parametrem konstruktoru předá stromový vzor, pro který jsou vytvořeny všechny potřebné struktury.

Samotné vyhledávání, pak probíhá zavoláním metody *match()*, kterou předepisuje rozhraní *IMatcher*, a které se v parametru předá strom, ve kterém má být vzor vyhledán. Metoda vrací čas, jak dlouho probíhalo samotné vyhledávání vzoru. Pro vrácení samotných výskytů vzoru ve stromě se pak využívá takzvané *anotace*, kdy metoda strom získaný parametrem upraví a označí v něm výskyty vzoru.

Všechny potřebné struktury a metody pro vnitřní běh algoritmu jsou ve třídě soukromé. Výpočet BA a BCS zajišťují metody *computeBA()* a *computeBCS()*. Jelikož se do BA indexuje znakem, byla pro reprezentaci BA zvolena struktura *HashMap<>*. O zmíněnou anotaci stromu se stará metoda *annotateTree()*.

Všechny verze algoritmu byly implementovány v této třídě, a která z verzí se použije, se rozhoduje v přetíženém konstruktoru třídy. Základní verze algoritmu je implementována v metodě *QL()*. Ostatní verze mají vlastní metody, každá začínající písmeny QL. Tento způsob implementace sice značně zvyšuje duplicitu kódu, cílem ale bylo, aby všechny verze byly co nejrychlejší a podmínkami uvnitř algoritmu by se celý běh zpomalil.

Kapitola 5

Experimentální vyhodnocení

V této kapitole jsou popsány výsledky měření implementace algoritmu QL v souboru nástrojů Forest FIRE. Měřena byla doba běhu algoritmů bez času potřebného k předzpracování.

Každá verze algoritmu byla měřena pro různé hodnoty Z. Nejprve je tedy každá verze porovnána sama se sebou pro různé hodnoty Z. Z každé této verze je vybrána ta s nejlepšími výsledky, které jsou následně porovnány mezi sebou. Několik nejlepších verzí je nakonec porovnáno s již existujícími algoritmy v Forest FIRE.

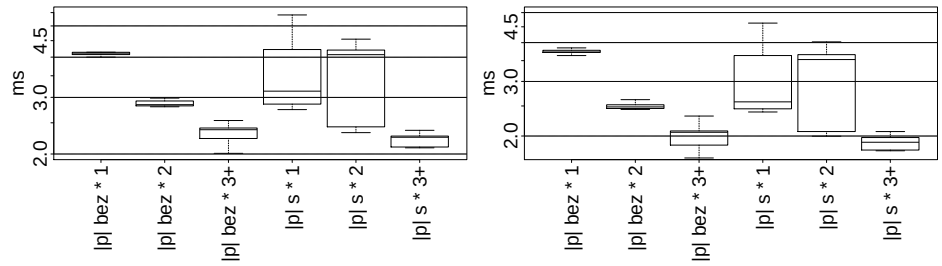
Protože v algoritmu závisí i na pozici znaku $*$, byla pro každou verzi algoritmu, pro každé měřené Z vyobrazena doba běhu v závislosti na prvním výskytu $*$. Konkrétně byly zvlášť měřeny výsledky na linearizovaných stromech, kde se $*$ vyskytuje zleva ve vzdálenosti jedna, dva a tři a více. Pokud se ve vzoru znak $*$ nevyskytoval, pak se měřila závislost doby běhu na velikosti vzoru. Všechny grafy jsou v přiloženém CD, ale v tomto textu budou tyto grafy uvedeny pouze pro dvě nejlepší hodnoty Z z každé verze.

Pro měření byla použita testovací data ze souboru nástrojů Forest FIRE. Tento soubor byl získán z gramatiky instrukčního setu Mono projektu X86, což je open source vývojová platforma založená na frameworku .NET. Sada má celkem 460 stromových vzorů různých velikostí.[2] Každý algoritmus byl spuštěn postupně s každým vzorem nad dvěma množinami stromů. První množina obsahuje 150 stromů zhruba velikosti 500 vrcholů. Druhá množina pak obsahuje 500 stromů, kde každý strom má zhruba 150 vrcholů. V této práci bude zachováno označení sad tak, jak je tomu v Forest FIRE a první zmíněná sada bude značena jako *sada 150* a druhá *sada 500*. Každý test je opakován dvacetkrát a výsledný čas běhu se bere jako průměr všech dvaceti běhů.

5.1 QL

V této části této kapitoly jsou uvedeny výsledky měření pro algoritmus QL jak byl popsán v algoritmu 10. Pro tuto verzi byla měřena Z od $\frac{1}{4}|p|$ do $2|p|$ s rozestupem $\frac{1}{4}|p|$.

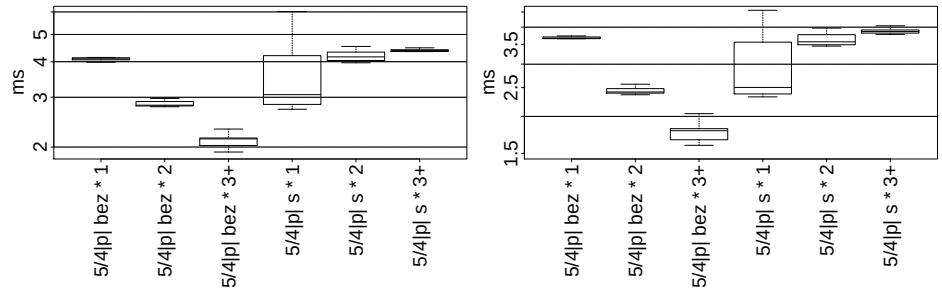
Na obrázku 5.1 je vyobrazena doba běhu v závislosti na prvním výskytu $*$. Poslední číslo ve jméně sloupce udává pozici $*$. Jak bylo řečeno, pokud se $*$ ve vzoru nevyskytuje, pak číslo odpovídá velikosti vzoru.



Obrázek 5.1: Závislost doby běhu algoritmu QL $Z = |p|$ na prvním výskytu znaku $*$ respektive velikosti vzoru. Vlevo sada 150 vpravo sada 500.

Z grafu je vidět, že v tomto případě s rostoucí pozicí zástupného znaku klesá doba běhu. Jelikož oba algoritmy MP i Boyer-Moore-Sunday mohou udělat nejvýše takový skok, jaká je pozice $*$ ve vzoru, je tento výsledek očekávaný.

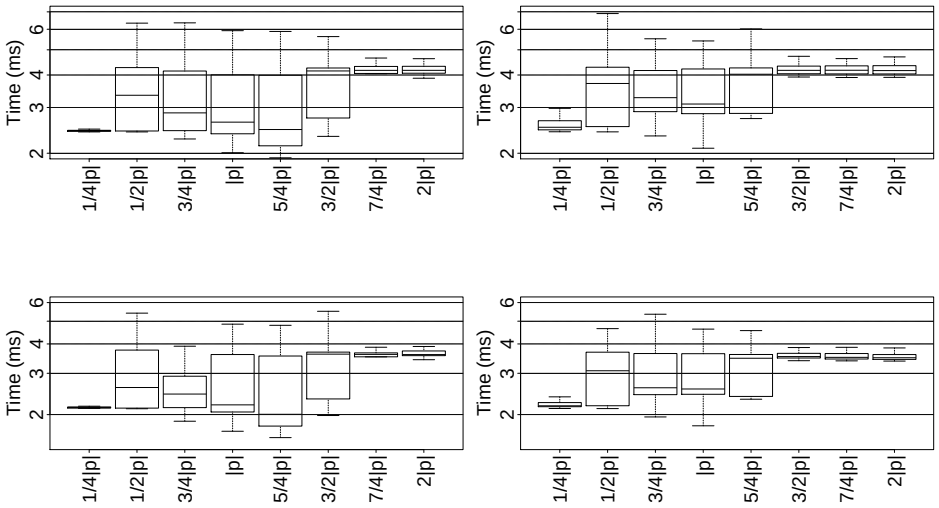
Stejný graf je pro hodnotu $Z = \frac{5}{4}|p|$ je uveden na obrázku 5.2. Pohled na výsledky se vzory s $*$ je ale překvapivý. Na grafu je vidět, že algoritmus se nejlépe chová v případě, kdy je hvězda na prvním místě a s rostoucí pozicí roste doba běhu. Tato změna oproti vzorům bez $*$ by mohlo být možné vysvětlit tím, že v datech, na kterých se algoritmus testoval, se pozice $*$ nezvyšuje ve stejném poměru jako velikost vzoru. Tedy pokud by se vzal poměr pozice prvního znaku $*$ a délky vzoru, pak by se tento poměr s rostoucí velikostí vzoru snižoval.



Obrázek 5.2: Závislost doby běhu algoritmu QL $Z = 5/4|p|$ na prvním výskytu znaku $*$ respektive velikosti vzoru. Vlevo sada 150 vpravo sada 500.

Na obrázku 5.3 jsou porovnány rychlosti algoritmu pro všechny měřené hodnoty Z. Z obrázku je vidět, že algoritmus běží nejlépe pro hodnoty menší než $\frac{5}{4}|p|$.

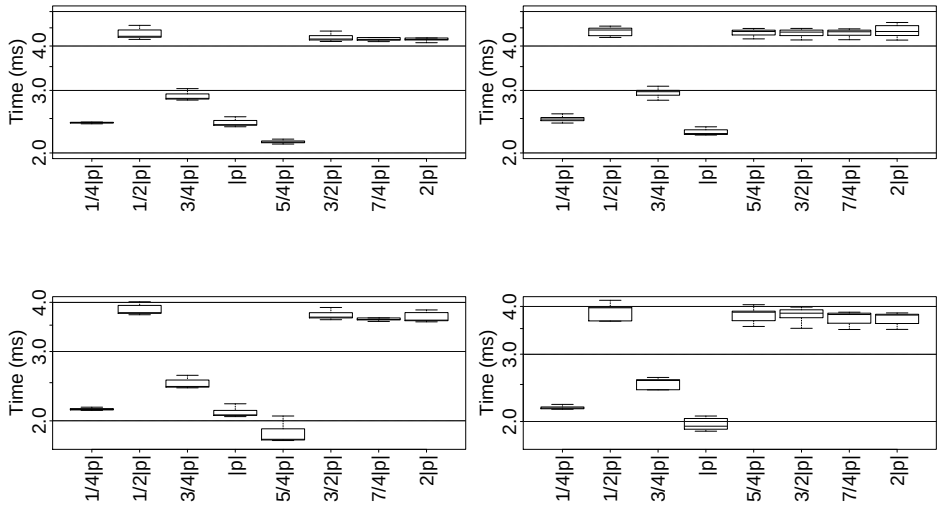
V datech je bohužel nezanedbatelné množství vzorů, které mají $*$ na první pozici, a jak bylo vidět z grafů na obrázcích 5.1 a 5.2, tento fakt může mít na dobu běhu značný vliv. Z tohoto důvodu byly algoritmy měřeny zvlášť na vzorech, ve kterých se znak $*$ vyskytuje na pozici tři a vyšší. Ve vzorech bez $*$ je pak jejich délka alespoň tři. Pro zkrácení zápisu budou tyto množiny vzorů značeny jako *3+*. Pokud tedy bude někde řečeno, že jsou uvažovány pouze vzory 3+, je tím myšlena takto zmenšená množina. Výsledek měření je na obrázku 5.4.



Obrázek 5.3: Srovnání dob běhů QL pro různé hodnoty Z na všech vzorech. Nahoře sada 150 dole sada 500 a vlevo vzory bez znaku $*$ vpravo s ním.

Na grafu na obrázku 5.4 je vidět značná změna oproti grafu na obrázku 5.3. Pro vzory bez $*$ dává nejlepší výsledky $Z = \frac{5}{4}|p|$ a druhou nejlepší volbou je $Z = |p|$. Je také vidět značný rozdíl u $Z = \frac{5}{4}|p|$ pro vzory bez znaku $*$ a s ním. Tento rozdíl byl ale už patrný z obrázku 5.2. Zajímavé ovšem je, že i přes omezení se na vzory 3+ je stále $Z = \frac{1}{4}|p|$ velmi dobrou volbou.

Důvodem k tomuto chování je optimalizace, která byla v implementaci přidána. Původní algoritmus provede skok vždy, když $i+Z-BCS[i+Z-1] < i+$ posun. Pokud je ale Z velmi malé, pak se může stát, že $i + Z$ bude menší než nový index získaný z posunu algoritmu MP. Zmíněná optimalizace spočívá v přeskočení výpočtu, jestli se má skok provést, pokud nastane zmíněná situace, že $i + Z$ je menší než posun získaný z tabulky posunů MP. Problémem algoritmu, který byl zjištěn až při měření, totiž je velký dopad nutnosti na-



Obrázek 5.4: Srovnání dob běhů QL pro různé hodnoty Z na vzorech 3+. Nahoře sada 150 dole sada 500 a vlevo vzory bez znaku $*$ vpravo s ním.

stavení indexu j na jedna v případě provedení skoku.

Jak bylo řečeno při popisu myšlenky algoritmu QL, pokud jsou skoky příliš malé, pak může být výpočetní čas strávený nad prováděním skoku vyšší, než čas ušetřený provedením skoku. V tomto případě je nutnost nastavení indexu j na jedna natolik nevýhodná, že je lepší kratší skok vůbec neprovádět.

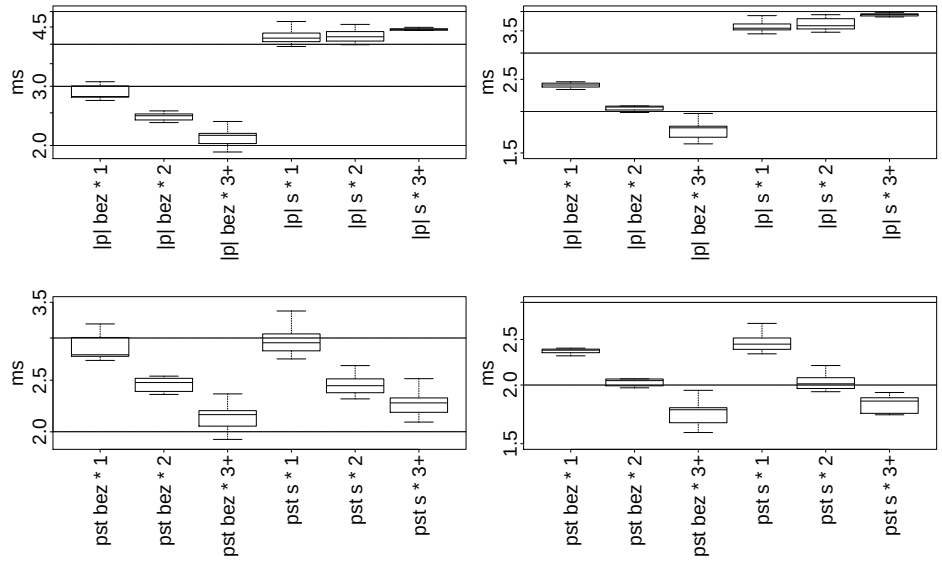
5.2 QL se skokem až po provedení posunu

V této části jsou výsledky měření verze QL popsané v sekci 3.5.1. Měřená Z byla z rozsahu $\frac{1}{4}|p|$ až $|p|$ opět s rozestupem $\frac{1}{4}|p|$. Navíc v této verzi bylo měřeno i Z vypočítané na základě pravděpodobnosti výskytu znaku. Formálně řečeno se Z vypočte následovně: $Z = \sum_{x \in \Sigma}(BCS[x] \cdot$ počet výskytů x ve $w)/|w|$. Protože Z je celočíselné, tak se hodnota zaokrouhlí.

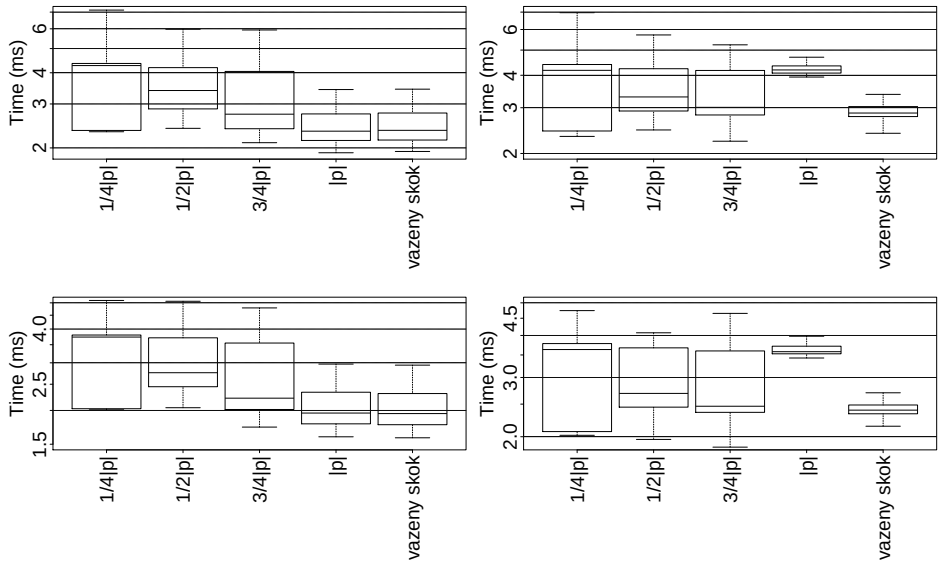
Na obrázku 5.5 jsou grafy závislosti doby běhu na pozici $*$ respektive velikosti vzoru. Horní dva obrázky jsou pro $Z = |p|$ a v dolních dvou bylo Z vypočítáno dle postupu uvedeného výše. Opět platí, že vlevo je sada 150 a vpravo 500.

Z obrázku je vidět, že pro $Z = |p|$ pro vzory se znakem $*$ nastává stejný problém, jako v obyčejném QL se $Z = \frac{5}{4}|p|$. Pro vzory bez $*$ si ale tato verze vede velmi dobře.

Srovnání všech zvolených hodnot Z na všech vzorech je na obrázku 5.6. Stejný graf na vzorech 3+ je pak na obrázku 5.7.

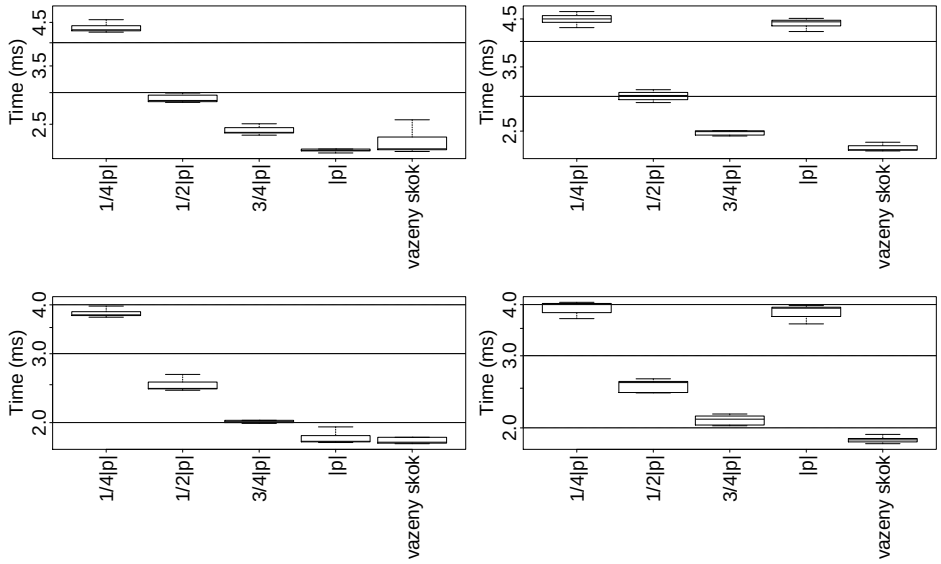


Obrázek 5.5: Závislost doby běhu QL se skokem až po posunu na pozici $*$ ve vzoru. Vlevo sada 150 vpravo 500. Nahoře $Z = |p|$ dole Z počítáno na základě pravděpodobnosti výskytu znaku.



Obrázek 5.6: Srovnání QL se skokem po posunu na všech vzorech. Nahoře sada 150 dole sada 500 a vlevo vzory bez znaku $*$ vpravo s ním.

Z grafů je vidět, že verze s váženým skokem si vede vždy velmi dobře. Zatímco stejně jako v případě obyčejného QL se $Z = \frac{5}{4}|p|$ si v tomto případě



Obrázek 5.7: Srovnání QL se skokem po posunu na vzorech 3+. Nahoře sada 150 dole sada 500 a vlevo vzory bez znaku $*$ vpravo s ním.

$Z = |p|$ vede dobře pouze na vzorech bez znaku $*$, ale na vzorech s výskytem tohoto znaku si vede pomalu nejhůře.

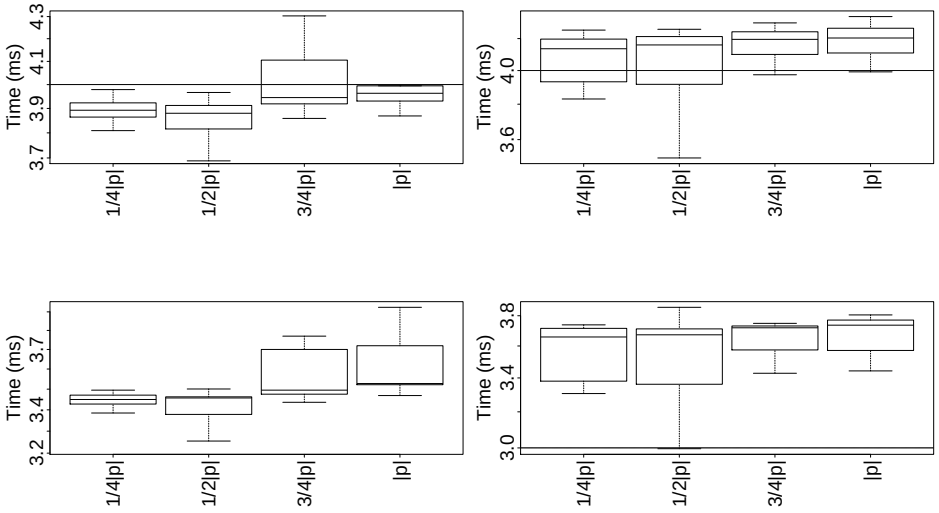
Výpočet rozdělení znaků v textu nebyl v této práci započítán do doby běhu a tento proces by dobu běhu značně zvýšil. Použití této verze tedy závisí na tom, jestli je dopředu známé rozdělení znaků. Pokud je rozdělení dopředu známé, pak je tato verze ve většině případů nejlepší ze všech implementovaných.

5.3 QL se dvěma tabulkami posunů

V této sekci jsou výsledky měření verze popsané v sekci 3.5.4. Z výsledků uvedených v poslední části této kapitoly 5.6 vyplývá, že tato verze je značně pomalejší než obyčejný algoritmus QL, QL se skokem až po posunu a QL s dynamicky měnícím se Z. Z tohoto důvodu bude v této kapitole vynechán obrázek s grafem závislosti doby běhu na pozici $*$ ve vzoru. V případě zájmu jsou ale všechny grafy dostupné na přiloženém CD.

Na obrázku 5.8 je graf doby běhu pro různé hodnoty Z pro verzi algoritmu se dvěma tabulkami posunů. Množina vzorů je 3+, jako vždy nahoře je sada stromů 150 a dole 500. Z obrázku je vidět, že nejlepší hodnota Z pro tuto verzi algoritmu je $\frac{1}{2}|p|$, a proto tato verze byla vybrána pro výsledné srovnání s ostatními verzemi.

Špatné výsledky této verze jsou pravděpodobně způsobeny tím, že indexu j je vždy třeba přiřadit hodnotu jedna. Tento problém byl již popsán v předchozí



Obrázek 5.8: Srovnání QL se dvěma tabulkami na vzorech 3+. Nahoře sada 150 dole sada 500 a vlevo vzory bez znaku $*$ vpravo s ním.

části 5.1. Protože určení, jestli ke skoku došlo nebo nedošlo, by znamenalo pouze nahrazení podmínky za jinou, je tato verze nevhodná v případě, kdy je jeden z použitých algoritmů vychází z algoritmu MP. Pokud by ale byl místo algoritmu MP zvolen nějaký jiný, pak by tato verze byla pravděpodobně nejlepší. Důvodem k této myšlence je fakt, že verze se skokem až po posunu je ve většině případů nejlepší a tato verze z ní vychází a ještě odebírá jednu podmínku.

5.4 QL s dvourozměrnou tabulkou posunů

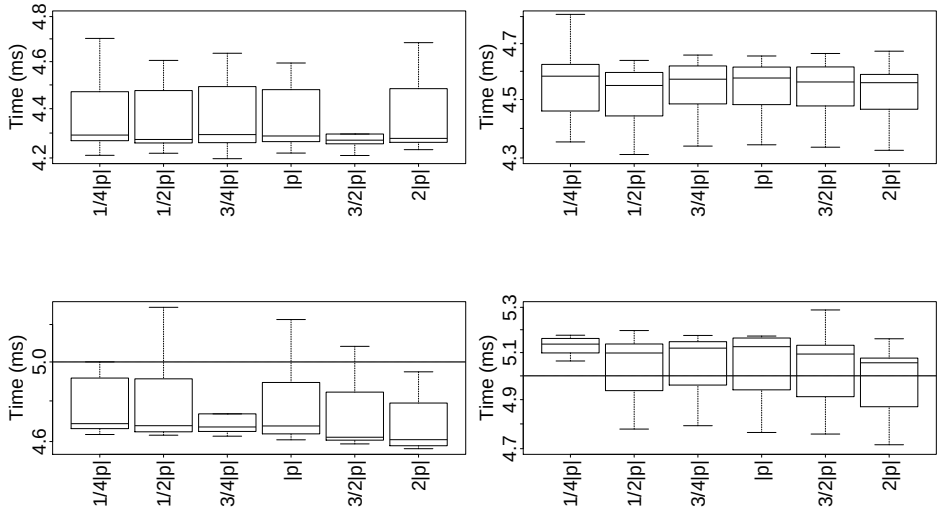
Měřená verze algoritmu v této sekci byla popsána v sekci 3.5.2.

Stejně jako verze s předchozí kapitoly byla i tato značně pomalejší než ty ostatní. Z tohoto důvodu, a z důvodu už tak velkého množství grafů v této práci, zde bude, stejně jako v předchozí sekci, uveden pouze graf srovnávající měřené hodnoty Z mezi sebou pro množinu vzorů 3+.

Výsledek měření je vidět na obrázku 5.9.

Tato verze má stejný problém jako ta uvedená v předchozí části této kapitoly, tedy nutnost nastavit index j vždy na jedna. Z tohoto důvodu nejsou horší výsledky této verze až tak překvapivé.

Vyšší paměťová náročnost kvůli dourozměrné tabulce je dalším záporem této verze, který stojí za zmínku.



Obrázek 5.9: Srovnání QL se dvourozměrnou tabulkou na vzorech 3+. Nahoře sada 150 dole sada 500 a vlevo vzory bez znaku $*$ vpravo s ním.

5.5 QL s dynamicky měnící se hodnotou Z

Tato část se zabývá verzí popsanou v sekci 3.5.3, tedy tou verzí, kde se při provedení skoku přičte k Z jednička a při neprovedení odečte jednička.

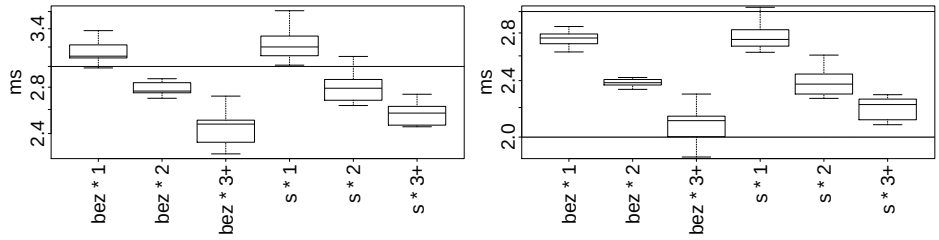
I když i pro tuto verzi algoritmu mohla být volena různá počáteční hodnota Z, nakonec byla měřena jen jedna počáteční hodnota $Z = |p|$ s tím, že tato verze má smysl pouze pro delší prohledávané stromy. Pokud je strom krátký, tak než stihne hodnota dokonvergovat ke vhodné hodnotě, prohledávání skončí. V takovémto případě je pak vcelku zbytečné používat tuto verzi a je lepší použít obyčejné QL. Pokud je prohledávaný strom dlouhý, tak na počáteční hodnotě Z tolik nezávisí, protože po nějaké době by Z teoreticky mělo konvergovat k nějaké optimální hodnotě nehledě na počáteční hodnotu.

Z tohoto důvodu není v této verzi s čím srovnávat a jediný graf, který je uveden na obrázku 5.10 je závislost doby běhu na pozici $*$ a velikosti vzoru.

5.6 Celkové srovnání algoritmů

V této části této kapitoly jsou srovnány implementované algoritmy s nejlepším výběrem Z mezi sebou a následně se všemi v souboru nástrojů Forest FIRE.

Na obrázku 5.11 jsou uvedeny všechny algoritmy již implementované v souboru nástrojů a posledních devět jsou různé verze algoritmu QL. Z každé verze byla vybrána ta hodnota Z, která dosahovala nejlepších výsledků na vzorech bez znaku $*$ a na vzorech s ním. Měřené výsledky jsou na sadě 150 se vzory



Obrázek 5.10: Závislost doby běhu algoritmu QL s dynamickým Z na pozici $*$ respektive velikosti vzoru. Vlevo sada 150 vpravo sada 500.

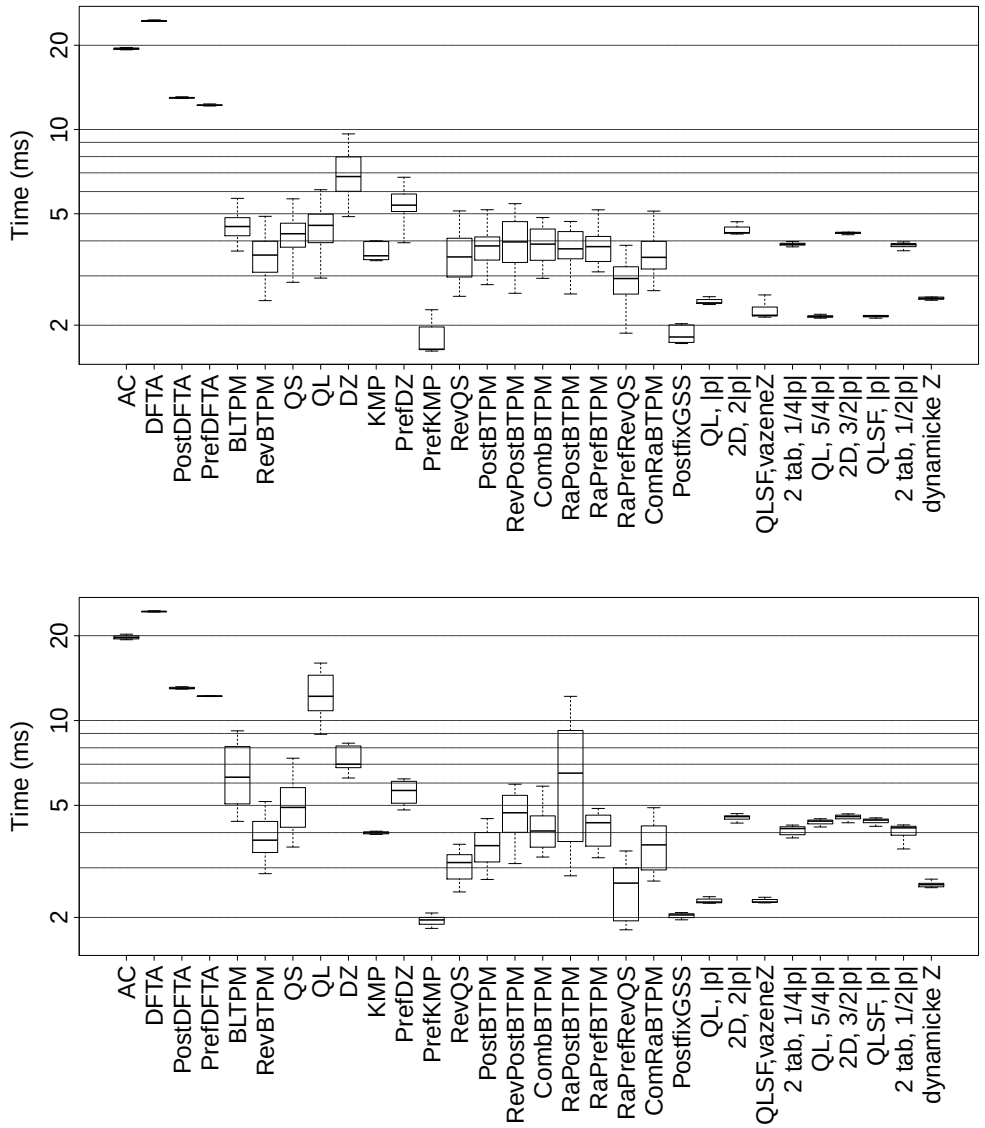
3+. Nahoře jsou vzory bez $*$ dole s tímto znakem. QL se skokem až po posunu je v grafu nazván *QLSF*. Grafy pro ostatní verze jsou na přiloženém CD.

Jak je z obrázku vidět, některé verze algoritmu si vedou opravdu velmi dobře. Protože je v souboru nástrojů velké množství algoritmů a většina z nich je pomalejších než nejlepší verze implementovaného QL, je navíc na obrázku 5.12 srovnání pouze pěti nejlepších verzí QL s algoritmem PrefixKMP a PostfixGSS.

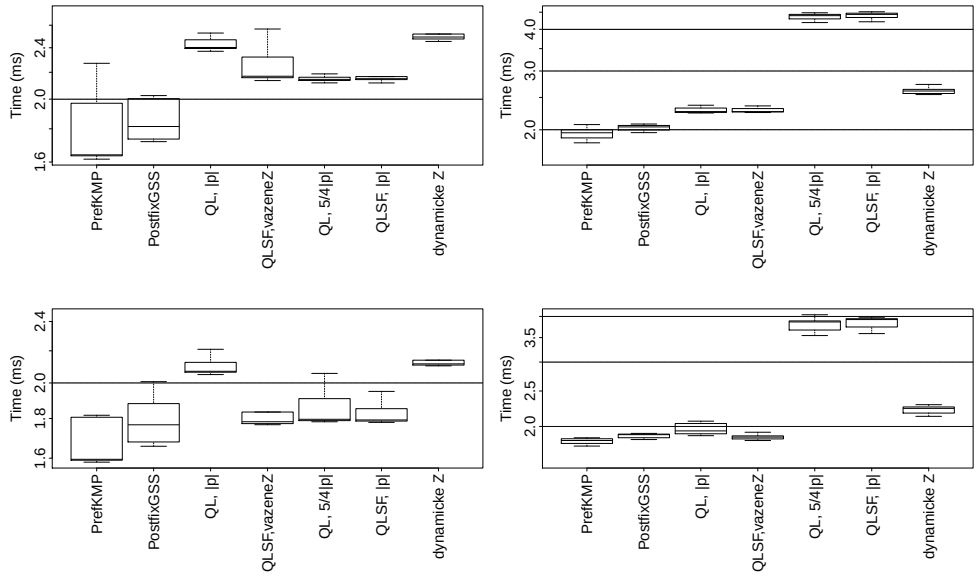
Z obrázku je vidět, že ve vzorech bez $*$ implementované QL značně zaostává, ale ve vzorech s tímto znakem jsou QL se $Z = |p|$ a QL se skokem až po posunu s váženým Z srovnatelné s nejlepšími již existujícími algoritmy v souboru nástrojů Forest FIRE.

Trochu smutné je, že i přes snahu najít nějakou lepší verzi, je nakonec obyčejné původní QL s $Z = |p|$ jednou z nejlepších verzí. I když velmi lehce zaostává za QL se skokem až po posunu s váženým Z, je rozdíl velmi zanedbatelný. Navíc pokud není dopředu známá distribuce znaků v prohledávaném textu, je nejspíše nevýhodné ztrácet výpočetní čas na jejím výpočtu.

Za zmínku také stojí, že i verze s dynamickým Z je v některých případech téměř srovnatelná s těmi nejlepšími. Je celkem překvapivé, že tato primitivní myšlenka dává takto dobré výsledky.



Obrázek 5.11: Srovnání nejlepších verzí implementovaného algoritmu se všemi již existujícími algoritmy. Sada stromů 150 množina vzorů 3+.



Obrázek 5.12: Srovnání nejlepších verzí QL s nejlepšími již existujícími algoritmy. Nahoře sada stromů 150 dole 500. Vlevo vzory beze znaku $*$ vpravo s ním. Množina vzorů 3+.

Závěr

Cílem práce bylo naimplementovat a otestovat algoritmus pro vyhledávání stromových vzorů ve stromech vycházející z algoritmu Quantum Leap pro vyhledávání v řetězcích.

Několik verzí algoritmu bylo naimplementováno v jazyce Java v rámci souborů nástrojů Forest FIRE. Implementace byly následně otestovány a srovnány jak mezi sebou, tak s ostatními existujícími algoritmy ve zmíněném souboru nástrojů. Z měření vyšlo, že některé vybrané verze algoritmu jsou relativně rychlé a problém řeší rychleji než většina implementovaných algoritmů v Forest FIRE.

Algoritmus by v budoucnu mohl být implementován i pro jiné notace linearizovaných stromů, případně by mohly být přidány další způsoby volby parametru Z. Další možností je využít myšlenku algoritmu Quantum Leap s jinými algoritmy než Morris-Pratt a Boyer-Moore-Sunday a třeba dosáhnout ještě lepších výsledků.

I když byly v této práci modifikace algoritmu QL aplikované na vyhledávání ve stromech, bylo by možné je aplikovat i na QL pro vyhledávání v řetězcích a provést podobné zhodnocení, jako tomu bylo v této práci.

Literatura

[1] Trávníček, J.; Oburka, R.; Pecka, T.; aj.: Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array. 2020.

[2] Cvach, M.: *Quick search vyhledávání ve stromech*. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2018.

[3] Mareš, M.; Valla, T.: *Pruvodce labyrintem algoritmu*. CZ. NIC, zspo, 2017.

[4] Plachỳ, Š.: *Automatová knihovna-Stromové automaty a algoritmy nad stromy*. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2015.

[5] Sedláček, J. E.: *Implementace vyhledávání v řetězcích pomocí kompaktního suffixového automatu*. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2018.

[6] Melichar, B.: *Jazyky a překlady*. Vydavatelství ČVUT, 2003, ISBN 978-80-01-02776-9.

[7] Oburka, R.: Vyhledávání ve stromech na principu mrtvých zón. 2016.

[8] Flouri, T.; Janoušek, J.; Melichar, B.: Subtree matching by pushdown automata. *Computer Science and Information Systems*, ročník 7, č. 2, 2010: s. 331–357.

[9] Janoušek, J.: String suffix automata and subtree pushdown automata. In *Proceedings of the Prague Stringology Conference*, ročník 2009, 2009, s. 160–172.

[10] Lahoda, J.; Žďárek, J.: Simple tree pattern matching for trees in the prefix bar notation. *Discrete Applied Mathematics*, ročník 163, 2014: s. 343–351.

[11] Holub, J.: Přednáška předmětu MI-EVY - 5. Exact string matching algorithms. 2019. Dostupné z: `https://courses.fit.cvut.cz/MI-EVY/lectures/mi-evy-05-exact_pattern_matching.pdf`

[12] Sunday, D. M.: A very fast substring search algorithm. *Communications of the ACM*, ročník 33, č. 8, 1990: s. 132–142.

[13] Watson, B. W.; Kourie, D. G.; Cleophas, L. G.: Quantum Leap Pattern Matching. In *Stringology*, Citeseer, 2015, s. 104–117.

[14] Janoušek, J.; Melichar, B.; Polách, R.; aj.: A full and linear index of a tree for tree patterns. In *International Workshop on Descriptional Complexity of Formal Systems*, Springer, 2014, s. 198–209.

[15] Melichar, B.; Janoušek, J.; Flouri, T.: Arbology: trees and pushdown automata. *Kybernetika*, ročník 48, č. 3, 2012: s. 402–428.

[16] Cleophas, L.: Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms. In *Proceedings of the 2009 conference on Finite-State Methods and Natural Language Processing: Post-proceedings of the 7th International Workshop FSMNLP 2008*, 2009, s. 191–198.

Příloha A

Seznam použitých zkratek

AST Abstraktní syntaktický strom

BCS Bad character shift - tabulka posunů v algoritmu Boyer-Moore-Sunday

KMP Algoritmus Knuth-Morris-Pratt

MP Algoritmus Morris-Pratt

QL Quantum Leap algoritmus

SJT Subtree Jump Table - podstromová skoková tabulka

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `ForestFireWood` ..... soubor nástrojů Forest FIRE s implementací
  - `ForestFIRE`
    - `src`
      - `forestfire`
        - `automata`
          - `matching`
            - `PrefixQLMatcher.java` ..... implementace
- `LaTeX` ..... adresář se zdrojovými kódy textu práce
  - `grafy` ..... všechny grafy s naměřenými výsledky
- `mereni` ..... adresář s výsledky měření
  - `test.run.150` ..... výsledky na množině stromů 150
  - `test.run.500` ..... výsledky na množině stromů 500
  - `csvFromResults` ..... skript převádějící výsledky do formátu CSV
- `DP_Sedlacek_JosefErik_2020.tex` ..... text práce ve formátu PDF