FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Návrh a implementace modifikací algoritmu protisměrného vyhledávání ve stromech
Student:	Kamil Červený
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 algoritmus protisměrného vyhledávání ve stromech [1] a algoritmus protisměrného vyhledávání v řetězcích, ze kterého vychází.
2) Proveďte rešerši variant algoritmu protisměrného vyhledávání v řetězcích.
3) Po dohodě s vedoucím zvolte varianty nebo variantu algoritmu protisměrného vyhledávání v řetězcích pro adaptaci na vyhledávání ve stromech.
4) Nastudujte zvolené varianty algoritmu protisměrného vyhledávání v řetězcích a adaptujte je pro vyhledávání ve stromech.
5) Implementujte a otestujte adaptované algoritmy.

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.

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

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

V Praze dne 6. února 2018

Bakalářská práce

Návrh a implementace modifikací algoritmu protisměrného vyhledávání ve stromech

Kamil Červený

Katedra teoretické informatiky
Vedoucí práce: Ing. Jan Trávníček

14. května 2018

Poděkování

Chtěl bych touto cestou poděkovat vedoucímu práce Ing. Janovi Trávníčkovi za cenné rady, konzultace a vstřícné jednání při realizaci této bakalářské práce. Dále bych chtěl poděkovat rodině a všem lidským bytostem, které mi přály, abych tuto práci zdárně dovedl do konce.

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 14. května 2018 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Kamil Červený. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Červený, Kamil. *Návrh a implementace modifikací algoritmu protisměrného vyhledávání ve stromech.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

V této práci je navrhnutý nový algoritmus pro vyhledávání ve stromech. Jeho fungování je založeno na myšlence heuristiky *good-suffix-shift* algoritmu *Boyer-Moore* pro vyhledávání v řetězcích a poznatcích z existujícího adaptovaného algoritmu *Morris-Pratt* pro stromy. Algoritmus najde všechny výskyty vyhledávaného vzoru stromu v daném prohledávaném stromě, k tomu využívá dvě pomocné datové struktury. Při běhu algoritmu jsou vstupní stromy převedeny do linearizované podoby, konkrétně do postfixové, rankové notace. Implementovaný algoritmus je na závěr testován s nejlepšími existujícími algoritmy pro vyhledávání ve stromech a výsledky měření ukazují, že se řadí mezi nejrychlejší z nich.

Klíčová slova protisměrné vyhledávání ve stromech, protisměrné vyhledávání v řetězcích, good-suffix-shift, Boyer–Moore, Knuth–Morris–Pratt

Abstract

In this thesis is designed a new tree pattern matching algorithm. Its idea is based on a concept of *good-suffix-shift* heuristic of *Boyer-Moore* string searching algorithm and findings of an existing adaptation of *Morris-Pratt* algorithm for trees. The algorithm finds all occurences of a sought tree pattern in a given sought through tree using two auxiliary data structures. During the algorithm's run input trees are transformed into a linearised form, specifically postfix, ranked notation. Finally the implemented algorithm is tested with the best existing tree pattern matching algorithms and measurements outcomes show that it ranks among the fastest of them.

Keywords backward tree pattern matching, backward string searching, good-suffix-shift, Boyer-Moore, Knuth-Morris-Pratt

Obsah

Úvod **1**

1 Zavedení základních pojmů **3**
- 1.1 Řetězce 3
- 1.2 Stromy, vzory stromů 4

2 Vyhledávaní v textu **7**
- 2.1 Naivní algoritmus 7
- 2.2 Boyer-Moore pro řetězce 9
- 2.3 Knuth-Morris-Pratt pro řetězce 12

3 Vyhledávání ve stromech **15**
- 3.1 Boyer-Moore-Horspool pro stromy 15
- 3.2 Morris-Pratt pro stromy 17

4 Návrh vlastního řešení **25**
- 4.1 Uvedení 25
- 4.2 Relace matchesReversed 25
- 4.3 Tabulka posunů 26
- 4.4 Důkaz správnosti 28
- 4.5 Analýza složitosti 31

5 Implementace a testování **33**
- 5.1 Popis implementace 33
- 5.2 Testování implementovaného algoritmu 34

Závěr **37**

A Seznam použitých zkratek **39**

B Obsah přiloženého flash disku **41**

Literatura **43**

Seznam obrázků

1.1 Grafické znázornění stromu nad rankovou abecedou 5
1.2 Grafické znázornění vzorů stromů 6

2.1 Schéma sousměrného vyhledávání v textu 7
2.2 Schéma protisměrného vyhledávání v textu 7
2.3 Schéma *bad-character-shift* 9
2.4 Schéma *good-suffix-shift* a jeho vylepšené verze 10
2.5 Schéma *reduced-good-prefix-shift* 11
2.6 Schéma posunu pro algoritmus *Morris-Pratt* 13
2.7 Schéma posunu pro algoritmus *Knuth-Morris-Pratt* 13

5.1 Distribuce časů porovnávaných algoritmů pro prohledávané stromy o přibližně 150 vrcholech 35
5.2 Distribuce časů porovnávaných algoritmů pro prohledávané stromy o přibližně 500 vrcholech 36

Seznam tabulek

2.1 Ukázka tabulek BA a GSS 11
2.2 Ukázka tabulky $RGPS$ 11
2.3 Ukázka tabulky $kmpNext$ 12

3.1 Ukázka tabulky $LTBCS(p)$ pro příklad 3.1.1 16
3.2 Ukázka tabulky SJT 17
3.3 Ukázka běhu algoritmu 3.1.3 pro příklad 3.1.2 19
3.4 Ukázka tabulky $LTBA$ 20
3.5 Ukázka tabulky $LTBA(p)$ pro příklad 3.2.1 22
3.6 Ukázka běhu algoritmu 3.2.3 pro příklad 3.2.1 23

4.1 Ukázka tabulky SJT 27
4.2 Ukázka tabulky $LTGSS(p)$ pro příklad 4.3.2 28
4.3 Ukázka tabulky $SJT(p)$ pro příklad 4.3.2 30
4.4 Ukázka běhu algoritmu 4.3.4 pro příklad 4.3.2 30

Úvod

S vyhledáváním v řetězcích se setkává spousta lidí například na internetu při hledání nějakého textu na webové stránce nebo v textovém dokumentu. Postupů řešící tento problém existují spousty a ty nejstarší jsou známé desítky let. S rostoucí velikostí prohledávaného a vyhledávaného textu samozřejmě roste potřebný čas pro vyhledávání, proto je důležité se tímto problémem zabývat a hledat efektivní řešení. Vyhledávací algoritmy lze dělit na protisměrné a sousměrné, což udává směr, jakým se porovnává a posouvá hledaný řetězec vůči prohledávanému textu. V této práci budou představeni zástupci obou z nich a nakonec se budu zabývat návrhem algoritmu vyhledávání protisměrného.

Stromy jako datové struktury popřípadě grafy jsou neoddělitelnou součástí dnešních informatických odvětvích, pro své vlastnosti jsou využívány například pro ukládání dat na pevných discích počítačů a v databázích, jako pomocná struktura algoritmů či pro zobrazení adresářové struktury operačním systémem uživateli. Tak jako lze chápat vyhledávání v textu, lze chápat vyhledávání ve stromech, jako hledání zadaného stromu nebo vzoru stromu ve stromu jiném (neplést s hledáním jediného uzlu). Existující vyhledávací algoritmy pro stromy, které budou uvedeny v této práci, jsou inspirovány svými řetězcovými protějšky. Disciplína s názvem *Arbologie* zabývající se mimo jiné právě stromovými algoritmy má zakládající členy, kteří působí na Fakultě informačních technologií Českého vysokého učení technického v Praze, kde také probíhá její výzkumná činnost, což je jeden z důvodů, proč jsem si téma zvolil.

Práce se dělí na teoretickou a praktickou část. Cílem první z nich je seznámení čtenáře se základními pojmy z oblastí teorie grafů a formálních jazyků nezbytnými pro pochopení později uvedených postupů. Dále představení některých existujících algoritmů protisměrného a sousměrného vyhledávání v řetězcích a stromech, které budou sloužit jako základ pro praktickou část této práce. Cílem praktické části je adaptace algoritmu protisměrného vyhledávání v řetězcích pro stromy spolu s jeho analýzou. Poté jeho implemen-

tace v programovacím jazyce *Java* a testování, kde bude zkoumána jeho reálná efektivita.

Kapitola 1

Zavedení základních pojmů

V této kapitole budou uvedeny definice základních pojmů, které budou dále v práci využívany. Konkrétnější definice související s jednotlivými algoritmy budou uvedeny v příslušných dalších kapitolách.

Uvedené definice v této kapitole pochází z [1] a [2].

1.1 Řetězce

Definice 1.1.1. *Abecedou* rozumíme neprázdnou konečnou množinu symbolů, kterou značíme Σ.

Definice 1.1.2. *Rankovou abecedou* rozumíme abecedu, kde každý její symbol a má nezápornou celočíselnou aritu (rank). Aritu symbolu a značíme *arita(a)*. Množinu všech symbolů abecedy Σ mající aritu rovnou n značíme Σ_n. Symboly mající aritu 0, 1, 2,..., n nazýváme nulární (konstanty), unární, binární,...,n-nární. Předpokládáme, že ranková abeceda obsahuje alespoň jednu konstantu.

Definice 1.1.3. Nechť Σ je abeceda, *řetězcem* nad abecedou Σ rozumíme konečnou posloupnost symbolů této abecedy. *Prázdným řetězcem* rozumíme prázdnou posloupnost symbolů této abecedy a značíme ho ε.

Definice 1.1.4. Nechť $r =$ je řetězec. *Délka řetězce* r je rovna počtu jeho symbolů a značíme ji $|r|$. Délka prázdného řetězce je rovna 0.

Definice 1.1.5. Nechť $r = a_1a_2...a_n$ je řetězec, *prefixem* r rozumíme řetězec $r_p = a_1a_2...a_k$, kde $1 \leq k \leq n$, pokud $1 \leq k < n$ jedná se o *vlastní prefix*.

Definice 1.1.6. Nechť $r = a_1a_2...a_n$ je řetězec, *sufixem* r rozumíme řetězec $r_s = a_ka_{k+1}...a_{n-1}a_n$, kde $1 \leq k \leq n$, pokud $1 < k \leq n$ jedná se o *vlastní sufix*.

Definice 1.1.7. Nechť r je řetězec, *podřetězcem* r rozumíme řetězec r_{ss} takový, že r_{ss} je sufix nějakého prefixu r.

Definice 1.1.8. Nechť r je řetězec, *hranicí* r rozumíme takový vlastní prefix r, který je zároveň sufix r. Nejdelší hranici řetězce r značíme $Border(r)$.

Příklad 1.1.1. Nechť $\Sigma = \{a, b, c, d\}$ je abeceda, $r = cabca$ je řetězec nad abecedou Σ, množina vlastních prefixů prefixů r je $\{cabc, cab, ca, c, \varepsilon\}$, množina vlastních sufixů r je $\{abca, bca, ca, a, \varepsilon\}$. Množina hranic r je $\{ca, \varepsilon\}$ a $Border(r) = ca$.

Zde zavedu značení, že pokud $r = a_1a_2 \ldots a_n$ je řetězec délky n, tak $r[i..j] = a_ia_{i+1} \ldots a_{j-1}a_j$, pokud $1 \leq i \leq j \leq n$, a $r[i..j] = \varepsilon$, pokud $i > j$.

1.2 Stromy, vzory stromů

Definice 1.2.1. *Zakořeněný strom* $t = (V, E)$ je acyklický, souvislý, orientovaný graf se speciálním vrcholem $k \in V$ zvaným *kořen* takovým, že:
1. k má vstupní stupeň roven 0,
2. všechny ostatní vrcholy mají vstupní stupeň roven 1,
3. pro každý vrchol $v \in V, v \neq k$ existuje právě jedna cesta z k do v. Vrcholy s výstupním stupňem rovným 0 nazýváme *listy*. Řekneme, že vrchol h je *synem* vrcholu g pokud $(g, h) \in E$.

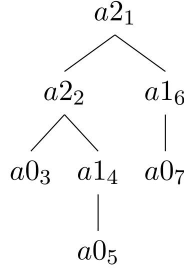
Definice 1.2.2. Nechť Σ je abeceda, *značený, zakořeněný strom* $t = (V, E)$ je zakořeněný strom takový, že každý $v \in V$ je označený nějakým symbolem $a \in \Sigma$.

Definice 1.2.3. Nechť Σ je ranková abeceda, *rankový, značený, zakořeněný strom* $t = (V, E)$ je značený, zakořeněný strom takový, že každý $v \in V$ je označený nějakým symbolem $a \in \Sigma$ a *arita(a)=„výstupní stupeň vrcholu v“*.

Definice 1.2.4. *Seřazený, rankový, značený, zakořeněný strom* $t = (V, E)$ je rankový, značený, zakořeněný strom takový, že synové každého vrcholu $v \in V$ jsou seřazeni.

Definice 1.2.5. Nechť t je seřazený, rankový, značený, zakořeněný strom, *prefixová, ranková notace* stromu t *prefRanked(t)*, je definována takto:
1. $prefRanked(t) = t0$, pokud t je list,
2. $prefRanked(t) = an\, prefRanked(s1)\, prefRanked(s2) \ldots$ $prefRanked(sn)$, kde a je kořen stromu t a *s1, s2,. . . , sn* jsou synové vrcholu a.

Definice 1.2.6. Nechť $r = a_1a_2 \ldots a_n$ je řetězec délky n nad rankovou abecedou Σ. Pak *ac(r)* (arity checksum řetězce r) je definována jako $ac(r) = arita(a_1) + arita(a_2) + \ldots + arita(a_n) - m + 1$. Řetězec r je prefixová, ranková notace nějakého stromu právě tehdy, když $ac(r) = 0$. A pokud r je vlastní prefix prefixové, rankové notace nějakého stromu, tak $ac(r) > 0$.



Obrázek 1.1: Grafické znázornění stromu nad rankovou abecedou

Definice 1.2.7. Nechť t je seřazený, rankový, značený, zakořeněný strom, *prefixová, ranková, barová notace* stromu t *prefRankedBar(t)*, kde $\uparrow n$ je takzvaný *barový symbol* arity n, je definována takto:
1. $prefRankedBar(t) = t0 \uparrow 0$, pokud t je list,
2. $prefRankedBar(t) = an\, prefRankedBar(s1)\, prefRankedBar(s2) \dots$
$prefRankedBar(sn) \uparrow n$, kde a je kořen stromu t a *s1, s2,. . . , sn* jsou synové vrcholu a.

Definice 1.2.8. Nechť t je seřazený, rankový, značený, zakořeněný strom, *postfixová, ranková notace* stromu t *postRanked(t)* je definována takto:
1. $postRanked(t) = t0$, pokud t je list,
2. $postRanked(t) = postRanked(s1)\, postRanked(s2) \dots postRanked(sn)\, an$, kde a je kořen stromu t a *s1, s2,. . . , sn* jsou synové vrcholu a.

Příklad 1.2.1. Nechť $\Sigma = \{a2, a1, a0\}$ je ranková abeceda, $t = (\{a2_1, a2_2, a0_3, a1_4, a0_5, a1_6, a0_7\}, \{(a2_1, a2_2), (a2_1, a1_6), (a2_2, a0_3), (a2_2, a1_4), (a1_4, a0_5), (a1_6, a0_7)\})$ je seřazený, rankový, značený, zakořeněný strom. Pak:
$prefRanked(t) = a2_1\, a2_2\, a0_3\, a1_4\, a0_5\, a1_6\, a0_7$,
$prefRankedBar(t) = a2_1\, a2_2\, a0_3 \uparrow 0_3\, a1_4\, a0_5 \uparrow 0_5 \uparrow 1_4 \uparrow 2_2\, a1_6\, a0_7 \uparrow 0_7 \uparrow 1_6 \uparrow 2_1$,
$postRanked(t) = a0_3\, a0_5\, a1_4\, a2_2\, a0_7\, a1_6\, a2_1$.
Grafické znázornění stromu t je na obrázku 1.1.

Definice 1.2.9. Nechť Σ je abeceda, symbol $S \notin \Sigma$ a $arita(S) = 0$. Symbol S slouží jako zástupný symbol pro jakýkoli strom o alespoň jednom vrcholu, pro jehož všechny vrcholy v platí $v \in \Sigma$. *Vzor stromu* je definován jako seřazený, rankový, značený, zakořeněný strom s vrcholy z množiny $\Sigma \cup \{S\}$, který obsahuje alespoň jeden vrchol $a \in \Sigma$. Vzor stromu obsahující alespoň jeden vrchol S nazýváme *šablonou stromu.*

Definice 1.2.10. Nechť t je strom a p je vzor stromu, řekneme, že p s k výskyty symbolu S se shoduje s t ve vrcholu v, pokud existují $t_1, t_2, \dots, t_k$ podstromy stromu *t(v)* takové, že strom p' vzniklý z p dosazením t_i za i-tý výskyt symbolu S je stejný jako strom t.

(a) Vzor stromu bez symbolu S (b) Vzor stromu se symbolem S

Obrázek 1.2: Grafické znázornění vzorů stromů

Příklad 1.2.2. Nechť t je strom z příkladu 1.2.1, $p_1 = (\{a2_1, a0_2, a1_3, a0_4\}, \{(a2_1, a0_2), (a2_1, a1_3), (a1_3, a0_4)\})$ je vzor stromu bez symbolu S (viz obrázek 1.2a) a $p_2 = (\{a2_1, S_2, a1_3, S_4\}, \{(a2_1, S_2), (a2_1, a1_3), (a1_3, S_4)\})$ je vzor stromu se symbolem S (viz Obrázek 1.2b). Vzor p_1 se vyskytuje jednou ve stromě t (shoduje se s t ve vrcholu $a2_2$) a vzor p_2 se vyskytuje dvakrát ve stromě t (shoduje se s t ve vrcholech $a2_2$ a $a2_1$).

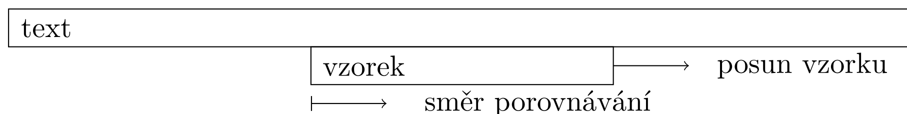
Kapitola 2

Vyhledávaní v textu

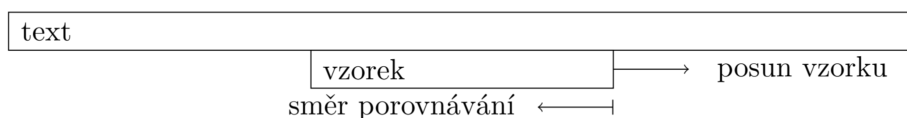
V této kapitole budou představeny některé existující algoritmy pro vyhledávíní v textu. Ty lze dělit na sousměrné a protisměrné. Sousměrný funguje tak, že vzorek, který je vyhledáván, je porovnáván s textem a posouván stejným směrem viz obrázek 2.1. Protisměrný se od něj liší tak, že vzorek je porovnáván s textem jedním směrem a posouván směrem druhým viz obrázek 2.2. U vyhledávacích algoritmů je jedním z určujících znaků efektivity posun, o který se posouvá vzorek vůči textu při neúspěšném porovnání a jeho nalezení. V následujících sekcích této kapitoly budou kromě naivního řešení představeny dva přístupy, jak takový posun zvolit, jejichž myšlenky budou dále v práci využity k vyhledávání ve stromech.

2.1 Naivní algoritmus

Naivní sousměrný algoritmus funguje tím způsobem, že se vyhledávaný vzorek přiloží na začátek textu a porovnává se znak po znaku od jeho začátku, pokud dojde k neúspěšnému porovnání znaků nebo nalezení celého vzorku, posouvá se vzorek vůči textu o jeden znak vpravo a začne se porovnávat od



Obrázek 2.1: Schéma sousměrného vyhledávání v textu



Obrázek 2.2: Schéma protisměrného vyhledávání v textu

svého začátku. Takové řešení neopotřebuje žádné předzpracování a jeho časová složitost je $O((n - m + 1) * m)$, jeho fungování je popsáno v algoritmu 2.1.1. Protisměrný naivní algoritmus se od sousměrného liší pouze tím, že znaky porovnává odzadu viz algoritmus 2.1.2. Lišící se řádky jsou 3 až 7. [5]

název : naivní sousměrné vyhledávání v textu
input : hledaný vzorek $r[0..m-1]$ délky m, prohledávaný text $T[0..n-1]$ délky n
output: seznam indexů s nalezeným vzorkem

```
1  i = 0;
2  while i ≤ n − m do
3  |  j = 0;
4  |  while j < m and r[j] == T[i + j] do
5  |  |  j = j + 1;
6  |  end
7  |  if j == m then
8  |  |  output(i);
9  |  end
10 |  i = i + 1;
11 end
```

Algoritmus 2.1.1: Naivní sousměrný algoritmus pro vyhledávání v textu

název : naivní protisměrné vyhledávání v textu
input : hledaný vzorek $r[0..m-1]$ délky m, prohledávaný text $T[0..n-1]$ délky n
output: seznam indexů s nalezeným vzorkem

```
1  i = 0;
2  while i ≤ n − m do
3  |  j = m − 1;
4  |  while j ≥ 0 and r[j] == T[i + j] do
5  |  |  j = j − 1;
6  |  end
7  |  if j == −1 then
8  |  |  output(i);
9  |  end
10 |  i = i + 1;
11 end
```

Algoritmus 2.1.2: Naivní protisměrný algoritmus pro vyhledávání v textu

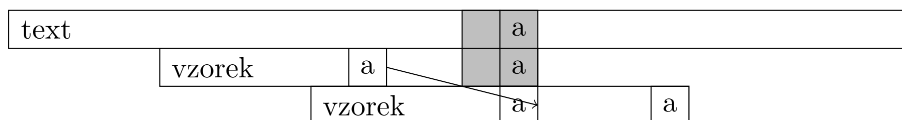
2.2 Boyer-Moore pro řetězce

V této sekci bude uveden algoritmus *Boyer-Moore*, který byl původně popsán v článku z roku 1977 viz [6], dále budou využity poznatky z [7] a [8].

Algoritmus je protisměrný. Je v něm využíváno tří heuristik *bad-character-shift* (BCS), *good-suffix-shift* (GSS) a *reduced-good-prefix-shift* (RGPS), na jejichž základě lze vylepšit naivní protisměrný algoritmus posunem vzorku o více než jeden znak.

2.2.1 Bad-character-shift

Heuristika *bad-character* spočívá v tom, že když během porovnávání dojde k neshodě znaků nebo nalezení vzorku, může být vzorek posouván do té doby, než dojde ke shodě mezi nejpravějším porovnávaným znakem textu a znakem vlastního prefixu vzorku (viz obrázek 2.3), pokud vzorek daný znak neobsahuje, je posunut o celou svou délku za poslední porovnávaný znak. K určení posunu se používá tabulka *BCS* (viz definice 2.2.1), která má velikost rovnou velikosti abecedy znaků, nad kterou je prohledávaný řetězec vytvořen. Tento posun je vždy takzvaně bezpečný a lze ho používat samostatně a nezávisle (nemůže při jeho použití dojít k posunu, který by vedl k nenalezení existujícího vzorku v textu).



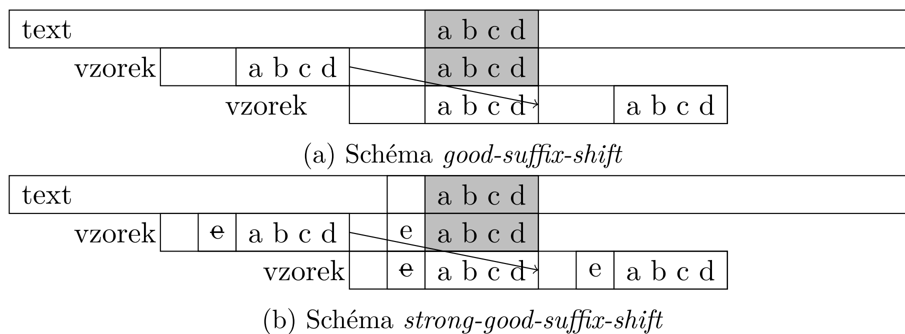
Obrázek 2.3: Schéma *bad-character-shift*

Definice 2.2.1. Nechť Σ je abeceda a $r[0..m-1]$ je vzorek o délce m nad abecedou Σ, tabulka *BCS* pro vzorek r je definována jako tabulka o velikosti $|\Sigma|$, kde pro každé $a \in \Sigma$ platí:
$BCS(r)[a] = min(\{m\} \cup \{j : r[m-1-j] = a \bigwedge j > 0\})$. [3]

Příklad 2.2.1. Nechť $\Sigma = \{a, b, c\}$ je abeceda a $r = a_0b_1b_2a_3b_4a_5b_6$ je vzorek nad abecedou Σ délky 7, pak $BCS(r)[a] = 1, BCS(r)[b] = 2, BCS(r)[c] = 7$.

2.2.2 Good-suffix-shift

Myšlenka *good-suffix-shift* heuristiky je ta, že když pro nějaké přiložení vzorku p k textu T ($p[0..m-1]$ k $T[i..i+m-1]$) během porovnávání dojde k neshodě znaků nebo nalezení vzorku a došlo ke shodě alespoň jednoho znaku ($t = p[j..m-1] = T[i+j..i+m-1] \neq \varepsilon$), je možné vzorek posouvat, dokud se všechny znaky t textu T budou znovu shodovat se vzorkem, vzorek tedy obsahuje znaky t na více než jednom místě (viz obrázek 2.4a). Pokud vzorek



(a) Schéma *good-suffix-shift*

(b) Schéma *strong-good-suffix-shift*

Obrázek 2.4: Schéma *good-suffix-shift* a jeho vylepšené verze

znaky t už dále neobsahuje, je posunut o svou délku. A pokud nedošlo ke shodě žádného znaku ($t = \varepsilon$), posouvá se vzorek o jedna.

Tento případ lze ještě vylepšit (viz obrázek 2.4b). Nechť $q = r[j..m-1]$ se shoduje s $t = T[k..k+m-1-j]$ a $c = r[j-1]$ je znak, kde došlo k neshodě. Ve vzorku se teď bude hledat výskyt q, kterému nepředchází znak c, takový výskyt se přiloží ke shodným znakům t textu T. Tomuto pravidlu se říká *strong-good-suffix-shift*. K tomuto posunu se využívá tabulka GSS (viz definice 2.2.3).

Takto zavedený GSS sám o sobě nezaručí bezpečné posunutí (mohlo by dojít k nenalezení existujícího výskytu v textu kvůli posunům o celou délku vzorku), proto se v kombinaci s ním využívá další heuritstika *reduced-good-prefix-shift*, která se někdy nazývá jako *druhý případ* GSS [9]. Tento GSS lze ovšem jednoduše upravit, aby prováděl pouze bezpečná posunutí a to tím způsobem, že posunutí o délku vzorku by se nahradilo posunutím o rozdíl délky vzorku a délky jeho nejdelší hranice, taková úprava by ale znemožnila provádět posunutí popsané v další sekci.

Definice 2.2.2. Nechť $r[0..m-1]$ je řetězec o délce m, tabulka BA (border array) pro vzorek r je definována jako tabulka o velikosti m indexovaná od 0 taková, že: $BA(r)[j] = max(i : i \leq m \bigwedge Border(r[j..m-1]) = r[i..m-1])$.

Definice 2.2.3. Nechť $r[0..m-1]$ je řetězec délky m. Tabulka GSS pro řetězec r je definována jako tabulka o velikosti $m+1$ indexovaná od 0 taková, že: $GSS(r)[j] =$
1. $j - max(i : BA(r)[i] = j)$, pokud existuje alespoň jedno i splňující $BA(r)[i] = j$
2. m, pokud neexistuje žádné i splňující $BA(r)[i] = j$.

2.2.3 Reduced-good-prefix-shift

Myšlenku *reduced-good-prefix-shift* lze použít, pokud pro nějaké přiložení ($p[j..m-1]$ k $T[i+j..i+m-1]$) vzorku p k textu T během porovnávání došlo

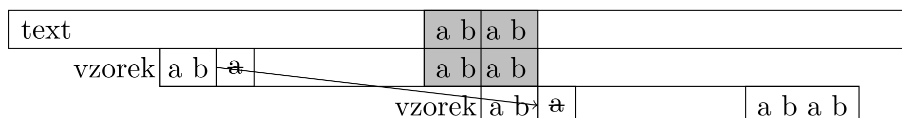
index	0	1	2	3	4	5	6	7
r	a	b	b	a	b	a	b	
$BA(r)$	5	6	4	5	6	7	7	
$GSS(r)$	7	7	7	7	2	2	2	1

Tabulka 2.1: Ukázka tabulek BA a GSS

index	0	1	2	3	4	5	6	7
r	a	b	b	a	b	a	b	
$RGPS(r)$	5	5	5	5	5	7	7	1

Tabulka 2.2: Ukázka tabulky $RGPS$

k neshodě znaků nebo nalezení vzorku a před tím došlo ke shodě alespoň jednoho znaku ($t = p[j..m-1] = T[i+j..i+m-1] \neq \varepsilon$) a znaky t se už dále nevyskytují ve vzorku p (GSS by tedy posouvalo vzorek o svou délku). V tomto případě lze vzorek posouvat, dokud se vlastní sufix t nebude shodovat s prefixem p (viz obrázek 2.5). K tomu se využívá tabulka $RGPS$ definovaná v definici 2.2.4. Posun zavedený tímto způsobem není sám o sobě bezpečný a může při něm dojít k příliš dlouhému posunu a nenalezení existujícího vzorku v textu. Z toho důvodu musí být vždy používán spolu s GSS uvedeným výše tak, že se jako finální posun vybere minimum z $RGPS$ a GSS.



Obrázek 2.5: Schéma *reduced-good-prefix-shift*

Definice 2.2.4. Nechť $r[0..m-1]$ je řetězec délky m. Tabulka $RGPS$ pro řetězec r je definována jako tabulka o velikosti $m+1$ indexovaná od 0 taková, že: $RGPS(r)[j] =$
1. 1, pokud $j = m$,
2. $min(i : j < i \leq m-1 \bigwedge r[i..m-1]\ je\ hranice\ r)$, pokud alespoň jedno takové i existuje,
3. m, pokud žádné takové i neexistuje.

S těmito třemi zadevenými heuristikami lze sestrojit algoritmus *Boyer-Moore*, tak že se řádek 10 Algoritmu 2.1.2 nahradí „$i = i + max(BCS[T[i+m-1]], min(GSS[j+1], RGPS[j+1]))$“, neboli se vybírá delší z obou posunů. Minimum z GSS a $RPGS$ je zde proto, aby nemohlo dojít k posunu, který by přeskočil výskyt vzorku v textu, jak bylo naznačeno dříve. Jeho časová složitost včetně předzpracování je $O(m*n+|\Sigma|)$, kde m je délka vzorku, n délka textu a $|\Sigma|$ velikost abecedy, ze které je prohledávaný text vytvořen.

index	0	1	2	3	4	5	6	7	8
r	G	C	A	G	A	G	A	G	
$kmpNext(r)$	-1	0	0	-1	1	-1	1	-1	1

Tabulka 2.3: Ukázka tabulky *kmpNext*

Pokud by se řádek 10 nahradil pouze „$i = i + BCS[T[i+m-1]]$“, jednalo by se o jednodušší algoritmus *Boyer-Moore-Horspool*.

2.3 Knuth-Morris-Pratt pro řetězce

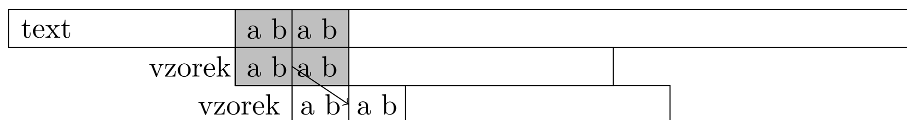
V této sekci bude představen algoritmus *Knuth-Morris-Pratt*, který byl původně popsán v článku z roku 1977 viz [10], dále budou využity poznatky z [11].

Knuth-Morris-Pratt (*KMP*) je sousměrný algoritmus, jeho základní myšlenka je ta, že když během porovnávání dojde k neshodě znaků na pozici $j > 0$ ve vzorku (indexovaném od 0), jsou již známy některé znaky textu. Vzorek lze pak posouvat do té doby, než se vlastní prefix shodných znaků vzorku $r[0..j-1]$ nebude shodovat se sufixem shodných znaků textu (viz obrázek 2.6). V té chvíli již není třeba znaky tohoto vlastního prefixu vzorku znovu porovnávat s textem, protože je jisté, že se budou shodovat. Tento přístup vede na jednodušší algoritmus *Morris-Pratt*.

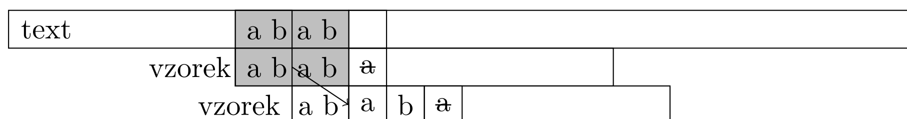
Uvedený případ lze ještě vylepšit (viz obrázek 2.7) a to tak, že daný vlastní prefix vzorku shodující se se sufixem v textu, není následovaný znakem $r[j]$, aby se předešlo neshodě hned za daným prefixem. Toto vylepšení vede na algoritmus *Knuth-Morris-Pratt*.

Algoritmus žádný znak textu, který se už jednou shodnul se vzorkem nikdy znovu neporovnává, což je jeho výhoda proti naivnímu řešení. Je jistá podobnost mezi přístupem *KMP* a dříve uvedeným *RGPS*, oba se koukají na prefix vzorku, ale *RGPS* ho hledá na konci vzorku za první neshodou znaků a *KMP* na konci vzorku před první neshodou znaků. *KMP* k určení posunu používá tabulku *kmpNext* (viz definice 2.3.1) konstruovanou algoritmem 2.3.1 a jeho fungování je popsáno v algoritmu 2.3.2. Toto řešení vede k časové složitosti vyhledávání včetně předzpracování $O(n+m)$, kde m je délka vzorku a n délka textu.

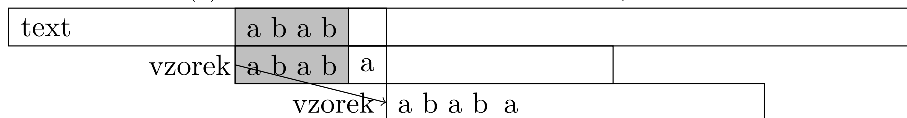
Definice 2.3.1. Nechť $r[0..m-1]$ je vzorek o délce m, tabulka $kmpNext$ pro vzorek r je definována jako tabulka o velikosti $m+1$, která má na indexu 0 hodnotu -1 a na indexech $j \in \{1..m\}$ délku nejdelší hranice řetězce $r[0..j-1]$ následovanou znakem c takovým, že $c \neq r[j]$, a -1, pokud taková hranice neexistuje.



Obrázek 2.6: Schéma posunu pro algoritmus *Morris-Pratt*



(a) Posun při existenci hranice shodných znaků



(b) Posun při neexistenci hranice shodných znaků

Obrázek 2.7: Schéma posunu pro algoritmus *Knuth-Morris-Pratt*

název : konstrukce tabulky *kmpNext*
input : vzorek $r[0..m-1]$ délky m, prázdná tabulka *kmpNext* velikosti $m+1$
output: vyplněná tabulka *kmpNext*

```
1  i = 0
2  j = −1
3  kmpNext[0] = −1
4  while i < m do
5      while j > −1 and r[i] ≠ r[j] do
6          j = kmpNext[j]
7      end
8      i = i + 1
9      j = j + 1
10     if r[i] == r[j] then
11         kmpNext[i] = kmpNext[j]
12     end
13     else
14         kmpNext[i] = j
15     end
16 end
```

Algoritmus 2.3.1: Konstrukce tabulky *kmpNext*

název : *Knuth-Morris-Pratt* pro řetězce
input : vzorek $r[0..m-1]$ délky m, tabulka $kmpNext(r)[0..m]$ velikosti $m+1$, prohledávaný text $T[0..n-1]$ délky n
output: seznam indexů s nalezeným vzorkem

```
1  i = 0
2  j = 0
3  while i < n do
4  |   while j > −1 and r[j] ≠ T[i] do
5  |   |   j = kmpNext[j]
6  |   end
7  |   i = i + 1
8  |   j = j + 1
9  |   if j ≥ m then
10 |   |   output(i − j)
11 |   |   j = kmpNext[j]
12 |   end
13 end
```

Algoritmus 2.3.2: *Knuth-Morris-Pratt* pro řetězce

Kapitola **3**

Vyhledávání ve stromech

Vyhledávání ve stromě je velmi podobné vyhledávání v textu. Pokud je zadaný vyhledávaný a prohledávaný strom, jedná se o nalezení vrcholu prohledávaného stromu takového, že je spolu s jeho potomky shodný se stromem vyhledávaným. Ve vyhledávaném stromě se může vyskytovat speciální vrchol S, který slouží jako zástupný symbol pro jakýkoli strom, vyžadující speciální přístup při vyhledávání. Na tomto místě upozorním, že zástupný symbol S se nemůže vyskytovat v prohledávaném stromě ale pouze ve vyhledávaném vzoru. Nejdříve je nutné určit, jakým způsobem budou stromy při vyhledávání reprezentovány, v této práci uvedených algoritmech je použita lineární reprezentace, se kterou lze pracovat jako s řetězcem (viz definice 1.2.8 a 1.2.7). V následujících sekcích budou představeny dva algoritmy pro vyhledávání ve stromech, jeden sousměrný a jeden protisměrný.

3.1 Boyer-Moore-Horspool pro stromy

V této sekci budou využívány poznatky, příklady, definice a algoritmy z [3].

Modifikace řetězcového *Boyer-Moore-Horspool* pro stromy je protisměrný algoritmus, využívá prefixové, rankové, barové notace pro prohledávaný strom a vyhledávaný vzor stromu. Pokud by vyhledávaný vzor neobsahoval žádný zástupný symbol S, vyhledávání by probíhalo úplně stejně jako u řetězců, což je výhoda této notace. Proto je u této modifikace nutné vyřešit pouze, jak na porovnávání a určení posunu při výskytu symbolu S ($\uparrow S$).

Algoritmus toto řeší pomocí dvou tabulek. Tabulka $LTBCS$ (linearised tree bad-charatcer-shift), jejíž fungování je velmi podobné té z řetězcového algoritmu *BMH*, tedy udává posun, aby se nejpravější porovnávaný znak stromu shodoval se znakem vlastního prefixu vzoru stromu. Ovšem výskyt symbolu S a $\uparrow S$ se může shodovat s jakýmkoli nebarovým a barovým znakem, na což je v tabulce dáván pozor (viz definice 3.1.2 a příklad 3.1.1). Dále tabulka SJT (viz definice 3.1.3 a tabulka 3.2) pro prohledávaný strom, potřebná pro přeskakování podstromů způsobeného symbolem $\uparrow S$. Ta se využívá k tomu,

Σ	$a3$	$a2$	$a1$	$a0$	$\uparrow 3$	$\uparrow 2$	$\uparrow 1$	$\uparrow 0$
$LTBCS(p)$	10	9	4	3	6	6	1	2

Tabulka 3.1: Ukázka tabulky $LTBCS(p)$ pro příklad 3.1.1

že když se má porovnávat symbol $\uparrow S$ ve vzoru se znakem c prohledávaného stromu, tak hodnota v tabulce pro pozici znaku c určuje, kde se má v prohledávaném stromě dál porovnávat. Neboli $S \uparrow S$ se shodne s podstromem, jehož nejpravější znak je c a porovnávání pokračuje nalevo od něj.

Algoritmus 3.1.2 konstruuje tabulku $LTBCS$, algoritmus 3.1.1 konstruuje tabulku SJT a samotné vyhledávání je popsáno v algoritmu 3.1.3 (ukázka běhu algoritmu je v příkladu 3.1.2). Předzpracování běží v čase $O(m + |\Sigma|)$ a vyhledávání v čase $O(n * m)$, kde m je délka vzoru stromu v prefixové, rankové, barové notaci, Σ je abeceda, ze které je vytvořen prohledávaný strom, a n je délka prohledávaného stromu v prefixové, rankové, barové notaci. Reálnou efektivitu tohoto řešení značně ovlivňuje pozice symbolu S v dané notaci, čím je pozice jeho nejpravějšího výskytu blíž pravému konci, tím kratší posunutí lze provádět (a naopak), to je způsobeno tím, že S (případně $\uparrow S$) se může shodovat s jakýmkoli znakem stromu.

Definice 3.1.1. Nechť Σ je ranková abeceda, symbolem $\uparrow_n$ budeme značit množinu všech barových symbolů $b \in \Sigma$.

Definice 3.1.2. Nechť Σ je ranková abeceda o velikosti n, $p[1..m]$ je vzor nějakého stromu nad abecedou Σ v prefixové, rankové, barové notaci o délce m. Tabulka $LTBCS(p)$ je definována jako tabulka velikosti n, kde pro každé $a \in \Sigma$ platí, že: $LTBCS(p)[a] = min(\{m\} \cup$
$\{j : p[m-j] = a \bigwedge m > j > 0\} \cup$
$\{j + arita(a) * 2 : p[m-j] = S \bigwedge m > j > 0 \bigwedge a \notin \uparrow_n\} \cup$
$\{j - 1 : p[m-j] = S \bigwedge m > j > 1 \bigwedge a \in \uparrow_n\})$

Příklad 3.1.1. Nechť $p = a2\, a1\, S \uparrow S \uparrow 1\, a1\, a0 \uparrow 0 \uparrow 1 \uparrow 2$ je vzor stromu v prefixové, rankové, barové notaci nad abecedou $\Sigma = \{a3, a2, a1, a0, S, \uparrow 3, \uparrow 2, \uparrow 1, \uparrow S\}$. $LTBCS(p)$ je zobrazena v Tabulce 3.1.

Příklad 3.1.2. Nechť $p = a2\, S \uparrow S\, S \uparrow S \uparrow 2$ je vzor stromu v prefixové, rankové, barové notaci a $subject = a2\ a2\ a0 \uparrow 0\ a0 \uparrow 0 \uparrow 2\ a2\ a0 \uparrow 0\ a0 \uparrow 0 \uparrow 2 \uparrow 2$ je strom v prefixové, rankové, barové notaci. Běh algoritmu 3.1.3 pro vyhledávaný vzor stromu p a prohledávaný strom $subject$ je zobrazen v tabulce 3.3.

Definice 3.1.3. Nechť t je vzor stromu a $p[1..m]$ je vzor stromu t v prefixové, rankové, barové notaci délky m. Tabulka $SJT(p)$ (subtree jump table) je definována jako tabulka velikosti m. Pro tu platí, že pokud $p[i..j]$ je prefixová, ranková, barová notace podstromu vzoru stromu t, tak $SJT(p)[i] = j + 1$ a $SJT(p)[j] = i - 1$, kde $1 \leq i < j \leq m$.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
p	$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$
$SJT(p)$	15	8	5	2	7	4	1	14	11	8	13	10	7	0

Tabulka 3.2: Ukázka tabulky SJT

název : ConstructSJT
input : vzor stromu t v prefixové, rankové, barové notaci $prefRankedBar(t) = p[1..n]$ délky n, index aktuálního vrcholu $rootIndex$ (defaultně 1), reference na $SJT(p)$ k vyplnění
output: vyplněná $SJT(p)$, index $exitIndex$

```
1 index = rootIndex + 1
2 for i in 1..arita(p[rootIndex]) do
3 |  index = ConstructSJT(p, index, SJT(p))
4 end
5 index = index + 1
6 SJT(p)[rootIndex] = index
7 SJT(p)[index] = rootIndex − 1
8 return index
```

Algoritmus 3.1.1: Algoritmus pro vyplnění tabulky SJT

3.2 Morris-Pratt pro stromy

V této sekci budou využívány poznatky, příklady, definice a algoritmy z [4].

Sejně jako *Morris-Pratt* pro řetězce je jeho modifikace pro stromy sousměrný algoritmus. Využívá prefixové, rankové, barové notace pro prohledávaný strom a vyhledávaný vzor stromu. Stejně jako bylo uvedeno v předcházející sekci platí i zde, že pokud by vyhledávaný vzor stromu neobsahoval symbol S, vyhledávání by pro tuto notaci fungovalo stejně jako *Morris-Pratt* pro řetězce.

Myšlenka modifikace *Morris-Pratt* pro stromy je velmi podobná té z jeho řetězcového protějšku. Tedy pokud během porovnávání dojde k neshodě znaků nebo nalezení výskytu vzoru stromu a došlo ke shodě alespoň jednoho znaku, vzor se posouvá do té doby, než dojde ke shodě vlastního prefixu shodných znaků vzoru a vlastního sufixu shodných znaků stromu. Ovšem v tomto případě, na rozdíl od řetězců, je určení, kdy se daný prefix a sufix shodují, složitější kvůli zástupnému symbolu S, který se může vyskytovat ve vzoru a který si vyžaduje speciální péči.

K určení shodnosti daného sufixu stromu a prefixu vzoru stromu je využita relace *matches* (viz definice 3.2.1 a algoritmus 3.2.1 rozhodující zda relace platí). Její základní myšlenka je ta, že pro prefix dané prefixové, rankové, barové notace vzoru stromu p se její sufix a prefix shodují, pokud se postupně shodují jejich znaky, kde při symbolu S se přeskakuje náležící podstrom, do

název : ConstructLTBCS
input : vzor stromu t v prefixové, rankové, barové notaci $prefRankedBar(t) = p[1..m]$ délky m nad abecedou Σ velikosti n
output: vyplněná $LTBCS(p)$

```
1  s = m
2  for i in 1..m do
3  |  if p[i] == S then
4  |  |  s = m − i
5  |  end
6  end
7  foreach x ∈ Σ do
8  |  LTBCS[x] = m
9  end
10 foreach x ∈ Σ do
11 |  if x ∉↑n then
12 |  |  shift = s + arita(x) ∗ 2
13 |  end
14 |  else if s ≥ 2 then
15 |  |  shift = s − 1
16 |  end
17 |  else
18 |  |  shift = s
19 |  end
20 |  if LTBCS[x] > shift then
21 |  |  LTBCS[x] = shift
22 |  end
23 end
24 for i in 1..m − 1 do
25 |  if p[i] ∉ {S, ↑S} and LTBCS[p[i]] > m − 1 then
26 |  |  LTBCS[p[i]] = m − 1
27 |  end
28 end
```

Algoritmus 3.1.2: Algoritmus pro vyplnění tabulky *LTBCS*

název : BackwardLTPM
input : prohledávaný strom v prefixové, rankové, barové notaci $subject[1..n]$ délky n, vyhledávaný vzor stromu v prefixové, rankové, barové notaci $pattern[1..m]$ délky m, $SJT(subject)$, $LTBCS(pattern)$
output: seznam indexů s nalezeným vzorem

1 $i = 0$
2 **while** $i \leq n - m$ **do**
3     $j = m$
4     $position = i + j$
5   **while** $j > 0$ **and** $position > 0$ **do**
6   **if** $subject[position] == pattern[j]$ **then**
7             $position = position - 1$
8   **end**
9   **else if** $pattern[j] == \uparrow S$ **and** $subject[position] \in \uparrow_n$ **then**
10             $position = SJT(subject)[position]$
11             $j = j - 1$
12   **end**
13   **else**
14   break
15   **end**
16         $j = j - 1$
17   **end**
18   **if** $j == 0$ **then**
19         $output(position + 1)$
20   **end**
21     $i = i + LTBCS(pattern)[subject[i + m]]$
22 **end**

Algoritmus 3.1.3: Protisměrný algoritmus pro vyhledávání ve stromech

1	2	3	4	5	6	7	8	9	10	11	12	13	14	index
$a2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$a2$	$a0$	$\uparrow 0$	$a0$	$\uparrow 0$	$\uparrow 2$	$\uparrow 2$	$subject$
14	6	2	2	2	2	6	6	2	2	2	2	6	14	velikosti podstromů
					$\uparrow 2$									$\uparrow 0 \neq \uparrow 2$, $shift = 1$
	$a2$	$S_\rightarrow$	$_\leftarrow S$	$S_\rightarrow$	$_\leftarrow S$	$\uparrow 2$								match, $shift = 1$
							$\uparrow 2$							$a2 \neq \uparrow 2$, $shift = 5$
							$a2$	$S_\rightarrow$	$_\leftarrow S$	$S_\rightarrow$	$_\leftarrow S$	$\uparrow 2$		match, $shift = 1$
$a2$	$S_\rightarrow$					$_\leftarrow S$	$S_\rightarrow$					$_\leftarrow S$	$\uparrow 2$	match, $shift = 1$

Tabulka 3.3: Ukázka běhu algoritmu 3.1.3 pro příklad 3.1.2

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p		$a2$	$a2$	$a0$	$\uparrow 0$	$a2$	S	$\uparrow S$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$a0$	$\uparrow 0$	$\uparrow 2$
$LTBA(p)$	-1	0	1	0	0	1	2	3	3	4	5	6	7	8	9	10	11

Tabulka 3.4: Ukázka tabulky $LTBA$

doby, než se u prefixu nebo sufixu dojde za jeho konec.

Pomocí této relace lze sestrojit tabulku $LTBA$ (viz definice 3.2.2) pro určení posunů, kterou konstruuje algoritmus 3.2.2. Její základní myšlenka je ta, že pro danou prefixovou, rankovou, barovou notaci p vzoru stromu má na každém svém indexu i délku nejdelší hranice prefixu $p[1..i]$. Pomocí této tabulky, lze určit posunutí vzoru stromu na základě počtu dosud shodných znaků.

Samotné vyhledávání modifikované na stromy je popsáno v algoritmu 3.2.3, jeho časová složitost je $O(m^3 + m * n)$ včetně předzpracování, kde n je délka vstupu *subject* a m je délka vstupu *pattern*.

Pro připomenutí, řetězcový *Morris-Pratt* nikdy znovu neporovnává znak prohledávaného stromu, který se už jednou shodnul se vzorem, tuto vlastnost zde uvedená modifikace na stromy nemá.

Definice 3.2.1. Nechť S_1 je prefixová, ranková, barová notace nějakého stromu a S_2 je podřetězec S_1. Řekneme, že S_1 a S_2 jsou v relaci *matches*, pokud:
1. $S_1 = xS_1^{'}$, $S_2 = xS_2^{'}$
$\quad \bigwedge S_1^{'}\, matches\, S_2^{'}$
2. $S_1 = SS_1^{'}$, $S_2 = SS_2^{'}$
$\quad \bigwedge S_1^{'}\, matches\, S_2^{'}$
3. $S_1 = x_1 \dots x_m S_1^{'}$, $S_2 = SS_2^{'}$
$\quad \bigwedge S_1^{'}\, matches\, S_2^{'}$
$\quad \bigwedge ac(x_1 \dots x_m) = 0 \wedge \forall k, 0 < k < n, ac(x_1 \dots x_k) \ngeq 0$
4. $S_1 = SS_1^{'}$, $S_2 = x_1 \dots x_m S_2^{'}$
$\quad \bigwedge S_1^{'}\, matches\, S_2^{'}$
$\quad \bigwedge ac(x_1 \dots x_m) = 0$
$\quad \bigwedge \forall k, 0 < k < n, ac(x_1 \dots x_k) \ngeq 0$
5. $S_1 = SS_1^{'}$, $S_2 = x_1 \dots x_m$
$\quad \bigwedge ac(x_1 \dots x_m) > 0$
$\quad \bigwedge \forall k, 0 < k < n, ac(x_1 \dots x_k) \neq 0$
6. $S_1 = \varepsilon$ nebo $S_2 = \varepsilon$

Definice 3.2.2. Nechť $pattern[1..m]$ je prefixová, ranková, barová notace nějakého vzoru stromu délky m. Pak $LTBA$ (linearised tree border array) je definována jako tabulka o velikosti $m + 1$ taková, že:
$LTBA[i] = i - min(j : pattern[1..m]\, matches\, pattern[j..i + j - 1])$.

název : Matches
input : řetězec $p[1..m]$ délky m, $SJT(p)[1..m]$,*offset*, *stop*
output: *true* nebo *false*

```
1  i = 1
2  while offset ≤ stop and i ≤ m do
3  |   if p[i] == p[offset] then
4  |   |   i = i + 1
5  |   |   offset = offset + 1
6  |   end
7  |   else if p[i] == S or p[offset] == S then
8  |   |   i = SJT[i]
9  |   |   offset = SJT[offset]
10 |   end
11 |   else
12 |   |   return false
13 |   end
14 end
15 return true
```

Algoritmus 3.2.1: Algoritmus určující zda $p[offset..stop]$ a $p[1..m]$ jsou v relaci *matches*

název : ConstructLTBA
input : řetězec $p[1..m]$ délky m, $SJT(p)[1..m]$, tabulka $LTBA[0..m]$ inicializovaná nulami
output: vyplněná tabulka $LTBA(p)[0..m]$

```
1  LTBA[0] = −1
2  for i in 1..m do
3  |   min = i
4  |   for j in 2..i do
5  |   |   if matches(p, SJT(p), stop = i, offset = j) then
6  |   |   |   min = j − 1
7  |   |   |   break
8  |   |   end
9  |   end
10 |   LTBA[i] = i − min
11 end
```

Algoritmus 3.2.2: Algoritmus pro konstrukci tabulky $LTBA$

```
název  : ForwardLTPM
input  : prohledávaný strom v prefixové, rankové, barové notaci
         délky n subject[1..n], vyhledávaný vzor stromu v prefixové,
         rankové, barové notaci délky m pattern[1..m],
         SJT(subject)[1..n], LTBA(pattern)[0..m]
output : seznam indexů s nalezeným pattern v subject

1  i = 1
2  while i ≤ n − m + 1 do
3      offset = i
4      j = 1
5      while j ≤ m and offset ≤ n do
6          if pattern[j] == subject[offset] then
7              j = j + 1
8              offset = offset + 1
9          end
10         else if pattern[j] == S then
11             offset = SJT[offset]
12             j = j + 2
13         end
14         else
15             break
16         end
17     end
18     if j > m then
19         output(i)
20     end
21     i = i + j − LTBA[j − 1] − 1
22 end
```

Algoritmus 3.2.3: *Morris-Pratt* pro stromy

index	0	1	2	3	4	5	6	7	8	9	10	11	12
p		$a2$	$a0$	$\uparrow 0$	$a2$	S	$\uparrow S$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$
$LTBA(p)$	-1	0	0	0	1	2	3	3	4	5	6	7	8

Tabulka 3.5: Ukázka tabulky $LTBA(p)$ pro příklad 3.2.1

Příklad 3.2.1. Nechť $p = a2\ a0 \uparrow 0\ a2\ S \uparrow S\ a1\ a0 \uparrow 0 \uparrow 1 \uparrow 2 \uparrow 2$ je vzor stromu v prefixové, rankové, barové notaci, $s = a2a0{\uparrow}0a2a2a0{\uparrow}0a2a0{\uparrow}0a1a0{\uparrow}0{\uparrow}1{\uparrow}2{\uparrow}2$ $a1\ a0 \uparrow 0 \uparrow 1 \uparrow 2 \uparrow 2$ je strom v prefixové, rankové, barové notaci. $LTBA(p)$ je zobrazena v tabulce 3.5 a běh algoritmu 3.2.3 pro vyhledávaný vzor stromu p a prohledávaný strom s je zobrazen v tabulce 3.6.

i	**1**	2	3	4	**5**	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
s	**a2**	$a0$	$\uparrow 0$	$a2$	**a2**	$a0$	$\uparrow 0$	$a2$	$a0$	$\uparrow 0$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$
p	**a2**	$a0$	$\uparrow 0$	$a2$	S											$\uparrow S$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$
p					**a2**	$a0$	$\uparrow 0$	$a2$	S	$\uparrow S$	$a1$	$a0$	$\uparrow 0$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$						
p									$a2$													
p										$a2$												
p											$a2$											

Tabulka 3.6: Ukázka běhu algoritmu 3.2.3 pro příklad 3.2.1

Kapitola **4**

Návrh vlastního řešení

V této kapitole bude představena adaptace *good-suffix-shift* heuristiky algoritmu *Boyer-Moore* (2.2.2) na stromy. Bude se tedy jednat o určení délky posunu vzoru stromu při protisměrném vyhledávání ve stromech při použití lineární reprezentace vstupů.

4.1 Uvedení

Stejně jako dříve uvedené adaptace i tato využívá lineární reprezentace pro vyhledávaný vzor stromu a prohledávaný strom. Na rozdíl od nich vzhledem k protisměrnosti vyhledávání bude využita postfixová, ranková notace (definice 1.2.8). Barová by pro tuto situaci nepřinesla navíc žádné užitečné informace a pouze by vedla na větší počet znaků pro lineární reprezentace obou vstupů a tím na pomalejší vyhledávání.

Pro přeskakování podstromů je potřeba pro prohledávaný strom tabulka, pomocí které lze při porovnávaní symbolu S efektivně posouvat index do stromu, ta je zavedena v definici 4.3.1 a konstruována algoritmem 4.3.1.

Pro připomenutí, *good-suffix-shift* určuje posun tím způsobem, že posouvá vzorek do té doby, než se již porovnané znaky textu znovu neshodují se vzorkem (dané znaky se ve vzorku vyskytují na více než jednom místě). U vzoru stromu se však může jako vrchol vyskytovat symbol S, který rozhodnutí, zda se řetězce shodují komplikuje.

4.2 Relace matchesReversed

Z toho důvodu je v definici 4.3.2 zavedena relace *matchesReversed*, která dokáže určit, zda jsou podřetězce nějaké postfixové, rankové notace vzoru stromu v jistém smyslu shodné. Definice využívá funkci *ac* (zavedenou v definici 1.2.6) ale pro postfixovou notaci, která má ovšem stejné vlastnosti jako pro prefixovou jen invertované. Pro rozhodnutí platnosti relace je použit al-

goritmus 4.3.2. Její základní myšlenka je podobná té z předchozí kapitoly (definice 3.2.1). Pro zde navrhovaný algoritmus bude potřeba rozhodovat, zda jsou v této relaci sufix (značící shodné znaky) a prefix (značící další výskyt shodných znaků) vzoru strom, což indikuje další výskyt sufixu ve vzorku. Pomocí těchto myšlenek lze vytvořit tabulku posunů, jak je popsáno v následující sekci.

4.3 Tabulka posunů

Stejně jako u řetězcového algoritmu funguje tabulka posunů tím způsobem, že se do ní indexem dotazuje na délku posunu pro aktuálně shodné znaky vzoru stromu. Zavedena je v definici 4.3.1 a konstruována algoritmem 4.3.3. Její myšlenka kopíruje tu z řetězců. Na každém indexu i udává délku posunu, pokud došlo ke shodě znaků $p[i..m-1]$ a neshodě znaku $p[i-1]$ vzoru stromu. Délka posunu je určena pomocí největšího indexu j takového, že $p[0..j]$ je v relaci *matchesReversed* s $p[i+1..m-1]$. Neboli znaky $p[i+1..m-1]$ se v jistém smyslu vyskytují na pozicích $p[k..j]$, kde $0 \leq k$. Kratší posun by určitě vedl k neshodě nějakého znaku prohledávaného stromu shodného s $p[i..m-1]$.

Výpočet této tabulky algoritmem 4.3.3 je podobný tomu při výpočtu tabulky BA u adaptace *Morris-Pratt* pro stromy (sekce 3.2). Hlavní rozdíly jsou dva. Zaprvé BA na rozdíl od $LTGSS$ neudává přímo velikost posunu a ten se musí dopočítávat při samotném vyhledávání. Za druhé výpočet $LTGSS$ využívá optimalizaci, kdy se proměnná *offset* při jeho běhu $O(m)$-krát zmenšuje, narozdíl od odpovídající proměnné j algoritmu 3.2.2, která je inkrementována $O(m^2)$-krát. To vylepšuje potřebný čas z $O(m^3)$ na $O(m^2)$.

Na tomto místě připomenu, že řetězcový *GSS* algoritmu *Boyer-Moore* není sám o sobě bezpečný (nezaručuje nepřeskočení existujícího vzorku v textu), a proto musí být používán spolu s *RGPS*. Zde navrhnutý posun založený na tabulce *LTGSS* zaručuje sám o sobě bezpečná posunutí, to proto že tabulka bere v potaz i případ, kdy pouze sufix shodných znaků vzoru stromu (ne nutně všechny shodné znaky) jsou v relaci *matchesReversed* s prefixem vzoru stromu, což je obdoba řetězcového *RGPS*.

Stejně jako u adaptovaného algoritmu *Boyer-Moore-Horspool* na stromy, je délka posunu značně ovlivněna pozicí symbolu S ve vzoru stromu. Čím je jeho pozice víc vpravo, tím menší posuny lze provádět. Nejdelší možný posun vzoru je ten, když se symbol S posune o jeho vzdálenost od pravého konce vzoru stromu v postfixové, rankové notaci. To je způsobeno tím, že jakýkoli shodný sufix vzoru se v jistém smyslu vždy shoduje (je v relaci *matchesReversed*) se samotným symbolem S.

Definice 4.3.1. Nechť t je vzor stromu a $p[0..m-1]$ je vzor stromu t v postfixové, rankové notaci délky m. *SJT(p)* (subtree jump table) je definována jako tabulka velikosti m. Pro tu platí, že pokud $p[i..j]$ je postfixová, ranková notace podstromu t, tak $SJT(p)[j] = i-1$, kde $0 \leq i \leq j \leq m-1$.

index	0	1	2	3	4	5
p	$a0$	$a1$	S	$a2$	$a0$	$a2$
$SJT(p)$	-1	-1	1	-1	3	-1

Tabulka 4.1: Ukázka tabulky SJT

název : constructSJTPostRanked
input : strom v postfixové, rankové notaci $p[0..m-1]$ délky m,
index aktuálního vrcholu $rootIndex$ (defaultně $m-1$),
reference na $SJT[0..m-1]$ k vyplnění
output: vyplněná SJT, index $exitIndex$

```
1 index = rootIndex − 1
2 for i in 1..arita(p[rootIndex]) do
3 |   index = constructSJTPostRanked(p, index, SJT)
4 end
5 SJT[rootIndex] = index
6 return index
```

Algoritmus 4.3.1: Vyplnění tabulky SJT pro strom v postfixové, rankové notaci

Definice 4.3.2. Nechť P je vzor stromu v postfixové, rankové notaci, P_1 a P_2 jsou podřetězce P. Řekneme, že P_1 a P_2 jsou v relaci *matchesReversed*, pokud platí jedno z následujících:
1. $P_1 = P_1^{'}x$, $P_2 = P_2^{'}x$
$\bigwedge P_1^{'}\ matchesReversed\ P_2^{'}$
2. $P_1 = P_1^{'}S$, $P_2 = P_2^{'}S$
$\bigwedge P_1^{'}\ matchesReversed\ P_2^{'}$
3. $P_1 = P_1^{'}x_1 \dots x_m$, $P_2 = P_2^{'}S$ nebo $P_2 = P_2^{'}x_1 \dots x_m$, $P_1 = P_1^{'}S$
$\bigwedge P_1^{'}\ matchesReversed\ P_2^{'}$
$\bigwedge ac(x_1 \dots x_m) = 0$
$\bigwedge \forall k, 1 < k \leq m, ac(x_k \dots x_m) > 0$
4. $P_1 = P_1^{'}S$, $P_2 = x_1 \dots x_m$ nebo $P_2 = P_2^{'}S$, $P_1 = x_1 \dots x_m$
$\bigwedge \forall k, 1 \leq k \leq m,\ ac(x_k \dots x_m) > 0$
5. $P_1 = \varepsilon$ nebo $P_2 = \varepsilon$

Příklad 4.3.1. Nechť $p = a0_0\ S_1\ a0_2\ a1_3\ a2_4\ a2_5$ je vzor stromu v postfixové, rankové notaci, $p_1 = a2_4\, a2_5$, $p_2 = a1_3\, a2_4\, a2_5$, $p_3 = a0_0\, S_1$, $p_4 = a0_0\, S_1\, a0_2\, a1_3$ jsou podřetězce p. Pak (p_1, p_3) a (p_2, p_3) jsou v relaci *matchesReversed*, a (p_1, p_4) a (p_2, p_4) nejsou v relaci *matchesReversed*.

Definice 4.3.3. Nechť $p[0..m-1]$ je vzor stromu v postfixové, rankové notaci délky m. $LTGSS(p)$ je definována jako tabulka velikosti $m+1$ taková, že pro každé $i \in \{0 \dots m\}$ platí:

název : matchesReversed
input : strom v postfixové, rankové notaci $p[0..m-1]$ délky m, tabulka $SJT(p)$, celá čísla *offset* a *stop*
output: *true* nebo *false*

```
1  i = offset
2  j = m − 1
3  while 0 ≤ i and stop ≤ j do
4      if p[i] == p[j] then
5          i = i − 1
6          j = j − 1
7      end
8      else if p[i] == S or p[j] == S then
9          i = SJT[i]
10         j = SJT[j]
11     end
12     else
13         return false
14     end
15 end
16 return true
```

Algoritmus 4.3.2: Algoritmus rozhodující, zda $p[0..\mathit{offset}]$ a $p[\mathit{stop}..m-1]$ jsou v relaci *matchesReversed*

index	0	1	2	3	4	5	6
p	$a0$	$a1$	S	$a2$	$a0$	$a2$	
$LTGSS(p)$	3	3	3	3	2	2	1

Tabulka 4.2: Ukázka tabulky $LTGSS(p)$ pro příklad 4.3.2

$LTGSS(p)[i] = m - 1 - max(j : -1 \leq j < m - 1 \bigwedge p[0..j]\ matchesReversed\ p[i..m-1])$.

Příklad 4.3.2. Nechť $p = a0\ a1\ S\ a2\ a0\ a2$ je vyhledávaný vzor stromu v postfixové, rankové notaci, $s = a0\ a1\ a0\ a1\ a0\ a2\ a0\ a2\ a2\ a0\ a2\ a1\ a0\ a1\ a2$ je prohledávaný strom v postfixové, rankové notaci. $LTGSS(p)$ je zobrazena v tabulce 4.2, $SJT(s)$ je zobrazena v tabulce 4.3 a běh algoritmu 4.3.4 pro p a s je zobrazen v tabulce 4.4.

4.4 Důkaz správnosti

Nechť $p[0..m-1]$ je vyhledávaný vzor stromu v postfixové, rankové notaci délky m, $subject[0..n-1]$ je prohledávaný vzor stromu ve stejné notaci,

název : constructLTGSS
input : strom v postfixové, rankové notaci $p[0..m-1]$ délky m, tabulka $SJT(p)$, tabulka $LTGSS[0..m]$ velikosti $m+1$ k vyplnění
output: vyplněná $LTGSS(p)$

```
1  offset = m − 2
2  for stop in m..0 do
3      LTGSS[stop] = m
4      while offset ≥ 0 do
5          if matchesReversed(p, SJT(p), offset, stop) then
6              LTGSS[stop] = m − 1 − offset
7              break
8          end
9          offset = offset − 1
10     end
11 end
```

Algoritmus 4.3.3: Konstrukce tabulky *LTGSS*

$LTGSS(p)$ je tabulka zkonstruovaná algoritmem 4.3.3 a $SJT(subject)$ je tabulka zkonstruovaná algoritmem 4.3.1. Pak algoritmus 4.3.4 najde všechny výskyty p v $subject$.

Dk.: Pokud by vyhledávaný vzor stromu neobsahoval zástupný symbol S a na řádku 20 by se proměnná i zvyšovala o 1, algoritmus by pracoval stejně jako naivní protisměrný řetězcový algoritmus 2.1.2. Stačí tedy ukázat, že zpracování symbolu S a posun na řádku 20 fungují správně.

Pokud algoritmus dojde na řádek 9 a podmínka „$p[j] = S$“ platí, znamená to, že podstrom *subjectu* s kořenem $subject[offset]$ (tento podstrom označím t) se má shodovat se symbolem S. Další porovnání má být mezi znaky $p[j-1]$ a znakem $subject[offset_1]$ takovým, že $subject[offset_1] \notin t$ a $subject[offset_1+1] \in t$. Tedy pro $offset_1$ platí, že $subject[offset_1]$ je znak o jedna nalevo od nejlevějšího znaku t, což je z definice 4.3.1 přesně hodnota v tabulce $SJT(subject)[offset]$.

Teď stačí dokázat, že posuny založené na tabulce *LTGSS* nemohou přeskočit žádný výskyt vyhledávaného vzoru. Nechť u znaků $p[l+1..m-1]$ došlo ke shodě a u znaku $p[l]$ k neshodě, posun vzoru by v tomto případě byl $LTGSS(p)[l+1] = k_1$, takže dle definice 4.3.3 jsou $p[l+1..m-1]$ a $p[0..m-1-k_1]$ v relaci *matchesReversed* a k_1 je nejmenší možné. Pro spor nechť existuje posun k_2, $0 < k_2 < k_1$, při kterém dojde k nalezení vzorku. Aby po posunu o k_2 znaků mohlo dojít k nalezení vzorku, musí být $p[l+1..m-1]$ a $p[0..m-1-k_2]$ v relaci *matchesReversed*. Takové k_2 ale nemůže existovat, protože k_1 s touto vlastností bylo zvoleno jako nejmenší možné. □

název : protisměrné vyhledávání ve stromech pomocí *LTGSS*
input : prohledávaný strom v postfixové, rankové notaci $subject[0..n-1]$ délky n, vyhledávaný vzor stromu v postfixové, rankové notaci $p[0..m-1]$ délky m, tabulky $SJT(subject)$ a $LTGSS(p)$
output: seznam indexů s nalezeným vzorem stromu

```
1  i = m − 1
2  while i < n do
3      offset = i
4      j = m − 1
5      while j ≥ 0 do
6          if p[j] == subject[offset] then
7              offset = offset − 1
8          end
9          else if p[j] == S then
10             offset = SJT(subject)[offset]
11         end
12         else
13             break
14         end
15         j = j − 1
16     end
17     if j == −1 then
18         output(i)
19     end
20     i = i + LTGSS(p)[j + 1]
21 end
```

Algoritmus 4.3.4: Protisměrné vyhledávání ve stromech pomocí *LTGSS*

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
s	$a0$	$a1$	$a0$	$a1$	$a0$	$a2$	$a0$	$a2$	$a2$	$a0$	$a2$	$a1$	$a0$	$a1$	$a2$
$SJT(s)$	-1	-1	1	1	3	1	5	1	-1	8	-1	-1	11	11	-1

Tabulka 4.3: Ukázka tabulky $SJT(p)$ pro příklad 4.3.2

i	0	1	2	3	4	5	6	**7**	8	9	**10**	11	12	13	14
s	$a0$	$a1$	$a0$	$a1$	$a0$	$a2$	$a0$	**a2**	$a2$	$a0$	**a2**	$a1$	$a0$	$a1$	$a2$
p				X	$a0$	$a2$									
p			$a0$	$a1$	S	$a2$	$a0$	**a2**							
p	$a0$	$a1$						S	$a2$	$a0$	**a2**				
p														X	
p														X	$a2$

Tabulka 4.4: Ukázka běhu algoritmu 4.3.4 pro příklad 4.3.2

4.5 Analýza složitosti

4.5.1 Časová složitost

Pro běh samotného vyhledávacího algoritmu pro vyhledávaný vzor stromu v postfixové, rankové notaci p o délce m a prohledávaný strom ve stejné notaci $subject$ o délce n jsou nezbytné dvě tabulky ($SJT(subject)$ a $LTGSS(p)$) konstruované algoritmy 4.3.1 a 4.3.3. Druhý z nich využívá algoritmus 4.3.2, který na vstupu vyžaduje tabulku $SJT(p)$. Tedy před tím, než může vyhledávání začít, musejí být známy tabulky tři.

Tabulky $SJT(p)$ a $SJT(subject)$ jsou konstruované v čase $O(m)$ a $O(n)$. Při vyplňování $LTGSS(p)$ se musí nejvíce $(2*m-1)$-krát ověřit relace *matchesReversed* pro podřetězce p, která pracuje v čase $O(m)$. Takže vyplnění $LTGSS(p)$ je v čase $(2*m-1)*O(m) = O(m^2)$. Samotné vyhledávání (algoritmus 4.3.4) pracuje v čase $O(m*n)$. Tedy zde navhrnuté řešení pracuje pro zadané vstupy $p[0..m-1]$ a $subject[0..n-1]$ celkově včetně předzpracování v čase $O(n)+O(m)+O(m^2)+O(m*n) = O(m^2+m*n)$.

4.5.2 Prostorová složitost

Vyhledávací algoritmus si musí během své činnosti pamatovat vstupy $p[0..m-1]$, $subject[0..n-1]$, $SJT(subject)[0..n-1]$, $LTGSS(p)[0..m]$ a konstantní počet vlastních proměnných majících konstantní velikost. To je v součtu $\Theta(m)+\Theta(n)+\Theta(n)+\Theta(m)+\Theta(1) = \Theta(m+n)$ paměti.

Kapitola 5

Implementace a testování

V této kapitole budou popsány některé detaily implementace a uvedeny výsledky testování navrhnutého algoritmu.

5.1 Popis implementace

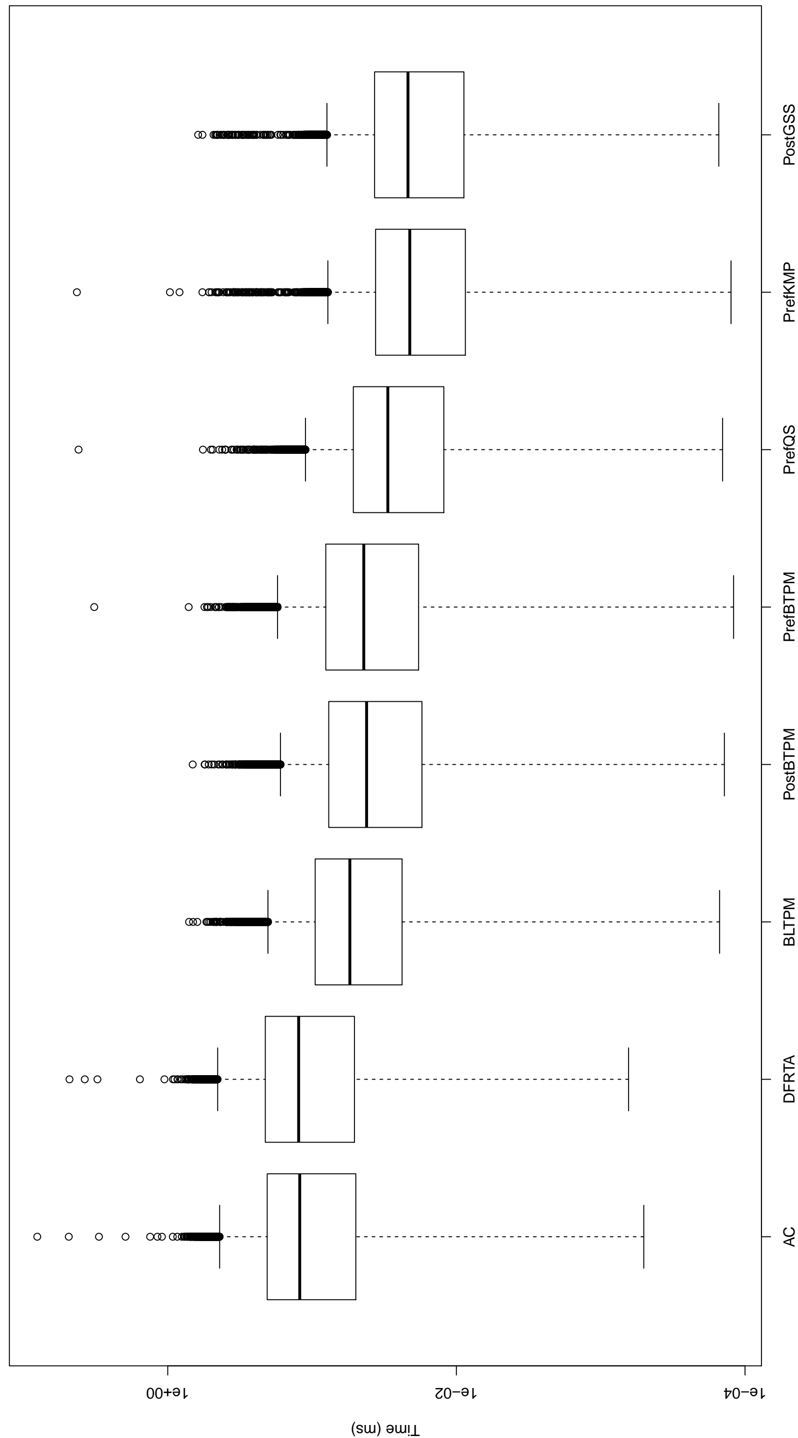
Jako jakzyk, ve kterém je zde navrhnuté řešení implementované, byla zvolena *Java* a to především z důvodu existence „Forest FIRE“ toolkitu, který je v tomto jazyce napsán, pro více informací o něm viz [12]. V něm jsou dostupné některé algoritmy týkající se automatů nebo vyhledávání ve stromech. Postupně byl rozšiřován o další včetně v této práci uvedených adaptací Boyer-Moore-Horspool a Morris-Pratt pro stromy a nyní je v něm obsažena i implementace algoritmu, který byl v této práci navrhnutý. Toolkit je doprovázen grafickým uživatelským prostředím „FIRE Wood“, které v kombinaci s dostupnými implementovanými algoritmy a testovacími nástroji a daty nabízí relativně dobrý nástroj pro testování na reálných datech, což bude využito v další sekci.

Samotná implementace zde navrhnutého řešení znamenala vytvoření třídy *GSSPostfixRankedMatcher*, která implementuje rozhraní *IMatcher* vyžadující implementaci metody *match*, která má jako argument prohledávaný strom. Vyhledávané vzory stromu jsou předávány instanci třídy v konstruktoru. Ve třídě byly vytvořeny tři privátní metody odpovídající algoritmům pro rozhodnutí platnosti relace *matchesReversed* ($matchesReversed()$), konstrukci tabulky $LTGSS$ ($computeLTGSS()$) a samotné vyhledávání ($matchUsingLTGSS()$). Toolkit má již implementované metody pro tvorbu tabulky SJT a vytvoření postfixové, rankové notace stromu, které metody této třídy také využívají. Takto napsaná třída je ihned připravená k testování.

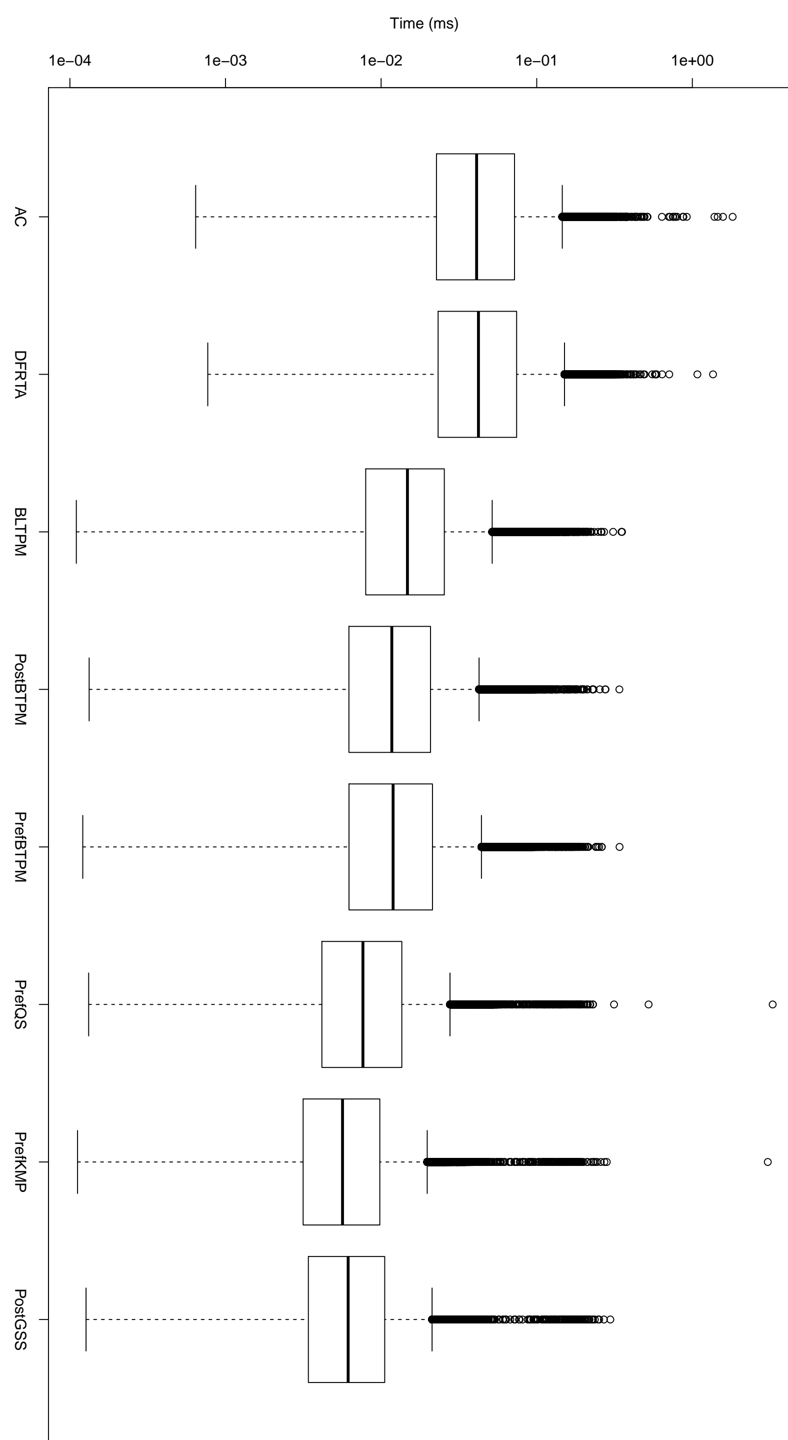
5.2 Testování implementovaného algoritmu

Testování navrhnutého implementovaného algoritmu probíhalo pomocí toolkitu FFW na stroji s 2 GHz Intel Core i7, 16 GB RAM na systému Opensuse Leap 42.3. s Javou SE 7.

Implementovaný algoritmus byl testován s dalšími sedmi algoritmy včetně v této práci uvedených adaptací *Boyer-Moore-Horspool* a *Morris-Pratt* pro stromy. Některé z testovaných algoritmů jsou pouze modifikace jednoho algoritmu pro barovou a nebarovou notaci vstupů, na těch lze vidět rozdíl výkonu ve prospěch nebarové (algoritmy *BLTPM* [barová] a *postBTPM* [nebarová]). Testovací data tvořily vyhledávané vzory stromů mající do 20 vrcholů a prohledávané stromy o přibližně 150 vrcholech (obrázek 5.1) a 500 vrcholech (obrázek 5.2). Zde navrhnutý algoritmus je v obrázcích uveden jako poslední s názvem *postGSS*. Výsledky měření ukazují, že navrhnuté řešení funguje ve srovnání s ostatními testovanými rychle a dosahuje druhého nejlepšího výsledku, překonává ho pouze adaptace algoritmu *Morris-Pratt* pro stromy (*PrefKMP*).



Obrázek 5.1: Distribuce časů porovnávaných algoritmů pro prohledávané stromy o přibližně 150 vrcholech



Obrázek 5.2: Distribuce časů porovnávaných algoritmů pro prohledávané stromy o přibližně 500 vrcholech

Závěr

Hlavním cílem bylo adaptovat protisměrný algoritmus pro vyhledávání v řetězcích na stromy. Po představení dvou řetězcových algoritmů a existujícími modifikacemi alespoň jejich částí pro stromy, byl zvolen přístup adaptace heuristiky *good-suffix-shift* algoritmu *Boyer-Moore*. Navrhnuté řešení se opírá o některé poznatky uvedené v sekci o adaptovaném algoritmu *Morris-Pratt* pro stromy. Jak ukázalo testování implementovaného navrhnutého algoritmu na reálných datech, jedná se o relativně výkonný algoritmus patřící mezi nejrychlejší dostupné. Do budoucna by stálo za to se zamyslet, jak by bylo možné zkombinovat zde navrhnuté řešení s adaptovaným algoritmem *Boyer-Moore-Horspool* na stromy, což by nemělo být složité, nebo jak zde navrhnuté řešení upravit, aby využívalo vylepšenou heuristiku *strong-good-suffix-shift*. Tím by vznikla adaptace celého algoritmu *Boyer-Moore* na stromy, která by mohla rychlostí zde navrhnuté řešení předčit.

Příloha A

Seznam použitých zkratek

BCS Bad-character-shift

GSS Good-suffix-shift

RGPS Reduced-good-suffix-shift

BA Border array

KMP Knuth-Morris-Pratt

LTBCS Linearised tree bad-character-shift

SJT Subtree jump table

LTBA Linearised tree border array

LTGSS Linearised tree good-suffix-shift

FFW Forest Fire Wood

BMH Boyer-Moore-Horspool

Příloha **B**

Obsah přiloženého flash disku

- `readme.txt` .... popis adresářové struktury tohoto flash disku
- `src` .... zdrojové kódy
  - `ForestFireWood.zip` .... FFW toolkit včetně implementace
  - `Cerveny_Kamil-thesis.zip` .... zdrojové kódy pro text této práce
- `thesis` .... texty práce
  - `BP_Cerveny_Kamil_2018.pdf` .... tato práce ve formátu PDF
  - `BP_Cerveny_Kamil_2018.ps` .... tato práce ve formátu PS

Literatura

[1] J. E. Hopcroft, R. Motwani, J. D. Ullman; *Introduction to automata theory, languages, and computation (3rd edition)*; Boston: Pearson/Addison Wesley, c2007; ISBN 0321455363

[2] G. Rozenberg, A. Salomaa; *Handbook of formal languages*; New York: Springer, c1997; ISBN 3540604200

[3] J. Trávníček, J. Janoušek, B. Melichar et al; *Backward Linearised Tree Pattern Matching*; LATA 2015, 2015

[4] R. Obůrka; *Dead zone tree pattern matching in trees*; Praha, 2016; Diplomová práce; Fakulta informačních technologií Českého vysokého učení technického v Praze

[5] *Searching for Patterns — Set 1 (Naive Pattern Searching)* [online]; [cit. 18.04.2018]; Dostupné z: `https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/`

[6] R. S. Boyer, J S. Moore; *A Fast String Searching Algorithm*; 1977; Communications of the ACM, 20, 10:762-772

[7] Ch. Charras, T. Lecroq; *Boyer-Moore algorithm* [online]; 14.01.1997 [cit. 26.04.2018]; Dostupné z: `http://www-igm.univ-mlv.fr/~lecroq/string/node14.html`

[8] B. Melichar; *Text searching algorithms: Volume II: Backward string matching* [online]; 2006 [cit. 18.04.2018]; Dostupné z: `http://www.stringology.org/athens/TextSearchingAlgorithms/tsa-lectures-2.pdf`

[9] *Boyer Moore Algorithm — Good Suffix heuristic* [online]; [cit. 18.04.2018]; Dostupné z: `https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic`

[10] D. E. Knuth, J. H. Morris, V. R. Pratt; *Fast Pattern Matching In Strings*; 1977; SIAM Journal on Computing 6(1):323-350

[11] Ch. Charras, T. Lecroq; *Knuth-Morris-Pratt algorithm* [online]; 14.01.1997 [cit. 26.04.2018]; Dostupné z: `http://www-igm.univ-mlv.fr/~lecroq/string/node8.html`

[12] L. Cleophas; *Forest FIRE and FIRE Wood: Tools for Tree Automata and Tree Algorithms*; 2008: pp. 191-198