Sem vložte zadání Vaší práce.

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

Bakalářská práce

Automatová knihovna - Stromové automaty a algoritmy nad stromy

Štěpán Plachý

Vedoucí práce: Ing. Jan Trávníček

12. května 2015

Poděkování

Děkuji Ing. Janu Trávníčkovi za pomoc s realizací této práce. Dále děkuji své rodině za velkou podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2015 .....

Odkaz na tuto práci

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

Abstrakt

Práce se zabývá konečnými stromovými automaty, což je výpočetní model pro zpracování regulárních stromových jazyků, a jejich implementací v projektu Automatová knihovna. V práci jsou navrženy a implementovány datové struktury stromů a stromových automatů, kterými je nutné knihovnu rozšířit. Z algoritmů je v práci navrženo a implementováno generování náhodných ohodnocených a neohodnocených označených uspořádaných stromů, determinizace bottom-up konečného stromového automatu, přijetí ohodnoceného stromu deterministickým a nedeterministickým bottom-up stromovým automatem a hledání výskytů jazyka deterministického bottom-up konečného stromového automatu v ohodnoceném stromu.

Klíčová slova konečný stromový automat, ohodnocený označený uspořádaný strom, regulární stromový jazyk, generátor náhodných stromů, determinizace

Abstract

The topic of this thesis is finite tree automata, which is a computational model for processing regular tree languages, and their implementation in project

Automata library. In the thesis are proposed and implemented data structures of trees and tree automata, with which is necessary to extend the library. Algorithms proposed and implemented in this thesis are random ranked and unranked labeled ordered tree generation, bottom-up finite tree automaton determinization, acceptance of ranked tree by deterministic and nondeterministic bottom-up finite tree automaton and finding occurrences of language of deterministic bottom-up finite tree automaton in ranked tree.

Keywords finite tree automaton, ranked labeled ordered tree, regular tree language, random tree generator, determinization

Obsah

Seznam obrázků

Úvod

Výpočetní modely stavových strojů jsou důležitou součástí teoretické informatiky pro svou funkci automatizace zpracování znakových řetězců, případně libovolných lineárních datových struktur, které lze na řetězec abstrahovat. Mimo lineárních struktur se velmi často používají stromové struktury, které lze zpracovávat těmito stavovými stroji, jelikož každý strom lze linearizovat, ale to nemusí být vždy žádaná operace.

Pro přímé zpracování stromů tedy existují podobné výpočetní modely. Jedním z nich je konečný stromový automat, který přijímá regulární stromové jazyky. V těch se mimo jiné počítá, že stromy jsou ohodnocené, tedy že počet uzlů potomků je vázán na hodnotu uloženou v uzlu. Podobně, jako u zásobníkového automatu, existují dvě verze podle směru výpočtu, a to top-down (shoda dolů) a bottom-up (zdola nahoru).

Pro práci s formalizmy z teorie automatů je pod vedením Jana Trávníčka vytvářena automatová knihovna. Tato práce má za úkol rozšířit tuto knihovnu o implementaci bottom-up stromového automatu a některé s ním související algoritmy. K tomu je nejprve nutné rozšířit knihovnu o chybějící datové struktury.

Jelikož jsem o tomto tématu nenašel žádné materiály v češtině, u některých pojmů mi není znám český ekvivalent, proto jejich překlad v této práci zavádím. Termíny *ohodnocený* a *označený*, jsou překlady anglických pojmů *ranked* a *labeled*.

Cíle práce

Prvním cílem je nastudování teorie konečných stromových automatů a algoritmů nad nimi.

Dalším cílem je naimplementovat datové struktury pro reprezentaci ohodnoceného a neohodnoceného stromu, a deterministického a nedeterministického bottom-up stromového automatu.

Posledním cílem je nad těmito strukturami naimplementovat náhodný generátor stromů, algoritmus determinizace stromového automatu, simulaci průchodu stromu stromovým automatem a hledání výskytů jazyka ve stromu stromovým automatem.

Kapitola 1

Teorie

1.1 Strom

Strom je nelineární datová struktura neobsahující cykly.

Definice 1.1. *Strom* t je definován rekurzivním vztahem

$$t = (v, c)$$
$$c = \{t_1 \dots t_n\}$$

kde v je hodnota určitého datového typu a $n \in \mathbb{N}$.

Jednotku t tvořící strukturu nazýváme *uzel* stromu. Hodnota uzlu bude značena $value(t)$.

Definice 1.2. *Potomek* uzlu t je každý uzel $t_i \in c$

Definice 1.3. *Rodič* uzlu t je uzel t_r, kde $t \in c_r$

Definice 1.4. *Sourozenec* uzlu je každý uzel, který má stejného rodiče.

Definice 1.5. *Kořen* je uzel, který nemá rodiče.

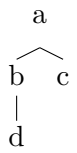
Definice 1.6. *Stupeň* uzlu t je $deg(t) = |c|$ [1]

Definice 1.7. *List* je každý uzel t, kde $deg(t) = 0$

Definice 1.8. *Vnitřní uzel* je každý uzel t, kde $deg(t) \neq 0$

Definice 1.9. *Hloubka* uzlu je vzdálenost uzlu[2] od kořenu.

Definice 1.10. *Výška* stromu je největší hloubka listů.



Obrázek 1.1: Příklad stromu

Příklad 1.1. Mějme strom na obrázku 1.1.

Uzel a je kořenem, uzly a a b jsou vnitřní uzly a uzly c a d jsou listy stromu.

Uzly b a c jsou potomky uzlu a a uzel a je rodičem uzlů b a c. Podobně uzel d je potomkem uzlu b a uzel b je rodičem uzlu d. Uzly b a c jsou sourozenci.

Stupeň uzlu a je 2, uzlu b 1, a uzlů c a d 0.

Hloubka uzlu a je 0, uzlů b a c 1 a uzlu d 2. Výška stromu je 2.

Definice 1.11. Strom nazýváme *uspořádaný* pokud množina potomků c v každém uzlu je uspořádaná. V opačném případě nazýváme strom *neuspořádaný*.

Definice 1.12. *i-tý potomek* $t[i]$ uzlu t v uspořádaném stromu je $c[i]$, kde c je množina potomků stromu t, $i \in \mathbb{N}$.[3]

Definice 1.13. Strom nazýváme *ohodnocený*, pokud existuje relace $[value(t), deg(t)]$ pro každý uzel t.

1.2 Ohodnocený stromový jazyk

1.2.1 Ohodnocený symbol

Definice 1.14. *Ohodnocený symbol* je dvojice (s, r), kde s je symbol a $r \in N$ nazýváme aritou symbolu.

Definice 1.15. *Konstantní symbol* nebo *konstanta* je ohodnocený symbol s aritou 0.

Arita v následujících kapitolách bude označovat stupeň uzlu ve stromu.

Na ohodnocený symbol lze také pohlížet jako na funkci, např. $f(x)$ pro symbol $(f, 1)$. Arita má v tomto kontextu význam počtu argumentů funkce.

Ohodnocené symboly budou v této práci značeny ve tvaru s_r a arita symbolu $arity(s)$.

1.2.2 Abeceda

Definice 1.16. *Abeceda* je konečná neprázdná množina symbolů.

[1] $|c|$ ozačuje velikost množiny c

[2] Vzdáleností je myšleno počet uzlů, které je třeba projít nejkratší cestou od jednoho uzlu k druhému.

[3] $c[i]$ značí i-tý prvek množiny c

1.2.3 Ohodnocená abeceda

Definice 1.17. *Ohodnocená abeceda* je konečná neprázdná množina ohodnocených symbolů.

Ohodnocená abeceda se nejčastěji se označuje velkými písmeny.

Příklad 1.2. $F = \{and_2, or_2, not_1, 1_0, 0_0\}$
Ohodnocená abeceda F je abecedou konstantních logických výrazů.

1.2.4 Označený strom

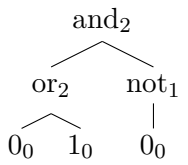
Definice 1.18. *Označený strom* t nad abecedou Σ je uspořádaný strom, kde pro každý uzel t platí $value(t) \in \Sigma$.

1.2.5 Označený ohodnocený strom

Definice 1.19. *Označený ohodnocený strom* t nad ohodnocenou abecedou F je ohodnocený uspořádaný strom, kde pro každý uzel t platí $value(t) \in F, deg(t) = arity(value(t))$.

Tato práce se již bude zabývat pouze stromy, které budou vždy označené a uspořádané, proto se jejich označení bude o tyto pojmy zkracovat. Rozlišujeme tedy pouze ohodnocené a neohodnocené stromy.

Příklad 1.3. $F = \{and_2, or_2, not_1, 1_0, 0_0\}, t :$



Obrázek 1.2: Příklad ohodnoceného stromu

Ohodnocený strom můžeme zapsat také lineárním zápisem:

$$t = and(or(0, 1), not(0))$$

F^* značí množinu všech ohodnocených stromů nad abecedou F.

1.2.6 Formální ohodnocený stromový jazyk

Definice 1.20. *Formální ohodnocený stromový jazyk* je libovolná množina ohodnocených stromů L nad abecedou F, kde $L \subseteq F^*$

Příklad 1.4. $F = \{and_2, or_2, not_1, 1_0, 0_0\}$

$$L = \{and(1, 1), or(1, 1), or(1, 0), or(0, 1), not(0)\}$$

Jazyk L je jazykem logických výrazů s hloubkou stromu 1, které jsou pravdivé.

1.3 Stromový automat

Automat je abstraktní výpočetní model pro rozhodnutí, zda řetězec patří do určitého jazyku. Stromový automat namísto toho rozhoduje, zda strom patří do stromového jazyka. K vyhodnocení používá konečnou množinu stavů, kterými přechodovou funkcí rekurzivně šíří informaci o struktuře stromu přes uzly.

Výpočetní silou se při linearizaci stromu nachází mezi konečným a zásobníkovým automatem a s konečným automatem sdílí některé vlastnosti.

Rozlišujeme dva druhy stromových automatů podle směru průchodu stromu, a to buď od listů zdola nahoru (bottom-up) nebo od kořene shora dolů (top-down). Oba mají stejnou výpočetní sílu, ale, narozdíl od bottom-up automatu, k nedeterministickému top-down automatu nemusí existovat evivalentní deterministický automat, proto je deterministický top-down automat výpočetně slabší[1].

1.3.1 Bottom-up stromový automat

Definice 1.21. *BU NFTA* je čtveřice $A = (Q, F, Q_f, \Delta)$, kde Q je množina stavů, F je množina vstupních ohodnocených symbolů, $Q_f \subseteq Q$ je množina finálních stavů a Δ je zobrazení $F \times Q_u \to Q^*$, kde Q_u je množina všech uspořádaných množin prvků z Q. [4]

Definice 1.22. Buď $A = (Q, F, Q_f, \Delta)$ BU NFTA, t ohodnocený strom nad abecedou F a $\Delta_t(t)$ funkce definovaná vztahem:

$$\Delta_t(t) = \Delta\left(value(t), \{\Delta_t(t[1]) \dots \Delta_t(t[n])\}\right)[i]$$

pro libovolné $i \in \mathbb{N}$, kde $n = arity(value(t))$.

Říkáme, že BU NFTA A *přijímá* ohodnocený strom t, pokud $\exists \Delta_t(t) \in Q_f$, v opačném případě říkáme, že automat ohodnocený strom *nepřijímá*.

Přechodová funkce bude zapisována výčtem mapovacích pravidel pro uzly ve tvaru

$$f(q_1 \dots q_n) \to q$$

pro symbol uzlu $f \in F$, stav i-tého potomka $q_i \in Q$, $i \in \mathbb{N}$, $n = arity(f)$, a výsledný stav $q \in Q$.

Příklad 1.5. Mějme jazyk všech binárních stromů s vnitřními uzly a a listy b a c, kde b je vždy levým potomkem a c vždy pravým. Vytvořme BU NFTA přijímající tento jazyk.

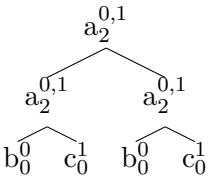
$A = (Q, F, Q_f, \Delta)$
$F = \{a_2, b_0, c_0\}$
$Q = \{0, 1\}$

[4] Q^* značí potenční množinu množiny Q

$Q_f = \{0, 1\}$
Δ :

$$b \to 0$$
$$c \to 1$$
$$a(0,1) \to 0$$
$$a(0,1) \to 1$$

Stav 0 označuje, že podstrom je levým potomkem, 1 pravým. Uzly b a c mohou být pouze levým resp. pravým potomkem, kdežto uzel a může být obojí. Kvůli tomu má pravidlo $a(0,1)$ více pravých stran. To způsobuje nedeterminismus při vyhodnocování. Ukažme si výsledek průchodu na následujícím stromu:



Obrázek 1.3: Výsledek příkladu 1.5

V horním indexu symbolu jsou všechny stavy, ve kterých se může uzel nacházet. Jelikož alespoň jeden ze stavů kořene je z množiny finálních stavů, automat strom přijímá.

1.3.1.1 Deterministický bottom-up stromový automat

BU DFTA je speciálním případem BU NFTA, kde se přechodovou funkcí může uzlu přiřadit pouze jeden stav. Průchod automatem je proto zcela jednoznačný a je jednodušší pro výpočet. Platí, že ke každému BU NFTA existuje ekvivalentní BU DFTA přijímající stejný jazyk[2].

Definice 1.23. *BU DFTA* je čtveřice $A = (Q, F, Q_f, \Delta)$, kde Q je množina stavů, F je množina vstupních ohodnocených symbolů, $Q_f \subseteq Q$ je množina finálních stavů a Δ je zobrazení $F \times Q_u \to Q$, kde Q_u je množina všech uspořádaných množin prvků z Q.

Pro přechodová pravidla tedy platí, že pro každou levou stranu může existovat pouze jedna pravá strana. Pokud pro nějakou konfiguraci uzlu není pravidlo definováno, bude výsledkem stav, který bude standardně značen q_{fail}, pro který platí $q_{fail} \notin Q_f$.

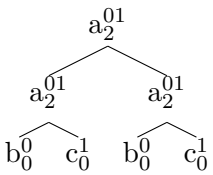
Příklad 1.6. Vytvořme deterministickou variantu automatu z příkladu 1.5.
$A = (Q, F, Q_f, \Delta)$
$F = \{a_2, b_0, c_0\}$

$Q = \{0, 1, 01\}$
$Q_f = \{0, 1, 01\}$
Δ :

$$\begin{aligned} b &\to 0 \\ c &\to 1 \\ a(0,1) &\to 01 \\ a(01,1) &\to 01 \\ a(0,01) &\to 01 \\ a(01,01) &\to 01 \end{aligned}$$

Jelikož pravidlo pro symbol a způsobovalo nedeterminismus, jelikož uzel se symbolem a může být na levé i pravé straně, přidáme třetí stav 01 označující, že uzel může být vlevo i vpravo. Tím musíme přidat další pravidla pro každou konfiguraci uzlu, pro kterou se mohou někteří potomci v tomto stavu nacházet.

Výsledek průchodu je v tomto případě následující:



Obrázek 1.4: Výsledek příkladu 1.6

Jelikož je automat deterministický, každý uzel stromu se již nachází pouze v jednom stavu. Protože stav kořene je z množiny finálních stavů, automat strom, stejně jako v předchozím příkladě, přijímá.

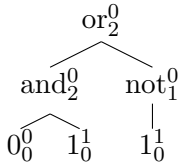
Příklad 1.7. Vytvořme automat přijímající všechny pravdivé konstantní logické výrazy.

$A = (Q, F, Q_f, \Delta)$
$F = \{and_2, or_2, not_1, 1_0, 0_0\}$
$Q = \{0, 1\}$
$Q_f = \{1\}$
Δ :

$$\begin{aligned} 0 &\to 0 & and(0,0) &\to 0 & or(0,0) &\to 0 \\ 1 &\to 1 & and(0,1) &\to 0 & or(0,1) &\to 1 \\ not(0) &\to 1 & and(1,0) &\to 0 & or(1,0) &\to 1 \\ not(1) &\to 0 & and(1,1) &\to 1 & or(1,1) &\to 1 \end{aligned}$$

Stavy označují pravdivostní ohodnocení uzlu, pravidla proto vycházejí z tabulek pravdivostních ohodnocení logických operací. Výsledek průchodu si ukažme na stromu na obrázku 1.5.

Pravdivostní ohodnocení kořene je 0, což není konečný stav, proto automat tento strom nepřijímá.



Obrázek 1.5: Výsledek příkladu 1.7

1.3.2 Top-down stromový automat

Definice 1.24. *TD NFTA* je čtveřice $A = (Q, F, Q_s, \Delta)$, kde Q je množina stavů, F je množina vstupních ohodnocených symbolů, $Q_s \subseteq Q$ je množina počátečních stavů a Δ je zobrazení $Q \times F \to Q_u^*$, kde Q_u je množina všech uspořádaných množin prvků z Q.

Definice 1.25. Buď $A = (Q, F, Q_s, \Delta)$ TD NFTA, t ohodnocený strom nad abecedou F a $\Delta_t(q, t)$ funkce definovaná vztahem:

$$\Delta_t(q,t) = (arity(value(t)) = 0) \vee (\Delta_{ti}(q,t,1) \wedge \cdots \wedge \Delta_{ti}(q,t,n))$$

$$\Delta_{ti}(q,t,i) = \Delta_t(\Delta(q, value(t))[i], t[i])$$

kde $q \in Q$ a $n = arity(value(t))$.

Říkáme, že TD NFTA A *přijímá* ohodnocený strom t, pokud $\exists \Delta_t(q_s, t)$ pro libovolné $q_s \in Q_s$, v opačném případě říkáme, že automat ohodnocený strom *nepřijímá*.

Mapovací pravidla pro TD automat budou mít tvar

$$q(f) \to (q_1 \dots q_n)$$

pro stav uzlu $q \in Q$, symbol uzlu $f \in F$ a výsledný stav i-tého potomka $q_i \in Q$, $i \in \mathbb{N}$, $n = arity(f)$.

BU a TD NFTA lze mezi sebou převézt transformací mapovacích pravidel:

$$f(q_1 \dots q_n) \to q \Leftrightarrow f(q) \to (q_1 \dots q_n)$$
$$Q_f \Leftrightarrow Q_s$$

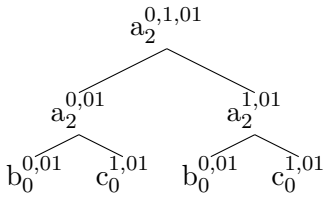
Pokud je automat deterministický, zobrazení Δ prosté a $|Q_f| = 1$ u BU automatu, bude i ekvivalentní automat deterministický.

Příklad 1.8. Vytvořme TD verzi automatu z příkladu 1.6.
$A = (Q, F, Q_s, \Delta)$
$F = \{a_2, b_0, c_0\}$
$Q = \{0, 1, 01\}$
$Q_s = \{0, 1, 01\}$

Δ :

$$\begin{aligned} 0(b) &\to () \\ 1(c) &\to () \\ 01(a) &\to (0,1) \\ 01(a) &\to (01,1) \\ 01(a) &\to (0,01) \\ 01(a) &\to (01,01) \end{aligned}$$

Ač je původní automat deterministický, v přechodové funkci existovali mapovací pravidla se stejnou pravou stranou, proto není prostá. Navíc množina finálních stavů obsahovala více než jeden prvek. Z těchto důvodů není výsledný automat deterministický. Výsledek průchodu v tomto případě vypadá následovně:



Obrázek 1.6: Výsledek příkladu 1.8

V horním indexu se nacházejí stavy, které mohli být uzlu přiřazeny z rodiče nebo z množiny počátečních stavů pro kořen. Jelikož každému listu byl přiřazen alespoň jeden stav, pro který existuje mapovací pravidlo, automat strom přijímá.

1.3.2.1 Deterministický top-down stromový automat

Definice 1.26. *TD DFTA* je čtveřice $A = (Q, F, q_s, \Delta)$, kde Q je množina stavů, F je množina vstupních ohodnocených symbolů, $q_s \in Q$ je počáteční stavů a Δ je zobrazení $Q \times F \to Q_u$, kde Q_u je množina všech uspořádaných množin prvků z Q.

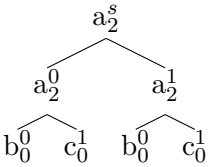
Pro přechodová pravidla platí, stejně jako u BU DFTA, že pro každou levou stranu může existovat pouze jedna pravá strana.

Příklad 1.9. Vytvořme deterministickou variantu automatu z příkladu 1.8.
$A = (Q, F, q_s, \Delta)$
$F = \{a_2, b_0, c_0\}$
$Q = \{0, 1, s\}$
$Q_s = s$

Δ :

$$
\begin{aligned}
s(b) &\to () \\
0(b) &\to () \\
s(c) &\to () \\
1(c) &\to () \\
s(a) &\to (0,1) \\
0(a) &\to (0,1) \\
1(a) &\to (0,1)
\end{aligned}
$$

Tento automat vychází z nedeterministické verze BU automatu, jelikož si vystačíme s informací, zda je uzel levým či pravým potomkem, protože to můžeme z rodiče jednoznačně určit. Jelikož ale měl BU automat více finálních stavů, přidáme další stav s (a k němu odpovídající pravidla), který bude počáteční a bude značit, že uzel je kořen. Výsledek průchodu v tomto případě vypadá následovně:



Obrázek 1.7: Výsledek příkladu 1.9

Jelikož je automat deterministický, stejně jako v případě BU DFTA je každému uzlu přiřazen pouze jeden stav a protože každý list má přiřazený stav, pro který existuje mapovací pravidlo, automat strom přijímá.

K TD NFTA nemusí existovat ekvivalentní TD DFTA přijímající stejný jazyk. Takový jazyk si ukážeme v následujícím příkladě.

Příklad 1.10. Mějme jazyk všech ohodnocených stromů se symboly a_2, b_1 a c_0, ve kterých se vyskytuje pouze jeden uzel se symbolem b_1. Vytvořme BU a TD DFTA přijímající tento jazyk.

$A = (Q, F, Q_f, \Delta)$
$F = \{a_2, b_1, c_0\}$
$Q = \{0, 1\}$
$Q_f = \{1\}$

Δ :

$$\begin{aligned} c &\to 0 \\ b(0) &\to 1 \\ a(0,0) &\to 0 \\ a(0,1) &\to 1 \\ a(1,0) &\to 1 \end{aligned}$$

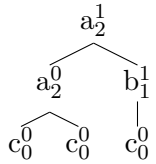
V tomto BU DFTA stav uzlu značí, zda podstrom obsahuje symbol b_1. Jelikož vyhodnocujeme od listů a máme tím tuto informaci od každého potomka, můžeme v každém uzlu jednoznačně říct, zda symbol obsahuje nebo ne. Pokud bychom vyhodnocovali od kořene, při průchodu uzlem nevíme, v kterého potomka podstromu se symbol nachází, proto průchod TD automatem nemůže být deterministický. Jelikož si mezi podstromy nelze předávat informace, může TD DFTA testovat podmínky, které platí pro všechny podstromy.

V tomto případě tedy TD NFTA může vypadat následovně:
$A = (Q, F, Q_s, \Delta)$
$F = \{a_2, b_1, c_0\}$
$Q = \{0, 1\}$
$Q_s = \{1\}$
Δ :

$$\begin{aligned} 0(a) &\to (0,0) \\ 1(a) &\to (0,1) \\ 1(a) &\to (1,0) \\ 1(b) &\to (0) \\ 0(c) &\to () \end{aligned}$$

V každém uzlu předpokládáme, že známe výsledek. Pokud předpokládáme, že se v podstromu symbol nachází, zkoušíme postupně všechny podstromy, zda se právě v něm symbol nachází.

Výsledek průchodu obou automatů s výslednými korektními stavy lze ukázat na následujícím stromu:



Obrázek 1.8: Výsledek příkladu 1.10

Kapitola 2

Automatová knihovna

Automatová knihovna je rozsáhlý projekt, vedený *Janem Trávníčkem*, zaměřený na práci s formalizmy z teorie automatů. Základ knihovny vytvořil ve své bakalářské práci *Martin Žák*[3]. Projekt dále rozšířili ve svých závěrečných pracích *Jan Veselý*[4] a *Tomáš Pecka*[5]. Projekt je psaný v jazyce `C++` (konkrétně `C++11`).

Základem projektu jsou dvě dynamické knihovny. `libalib2data.so` sdružuje datové struktury a `libalib2algo.so` algoritmy nad těmito strukturami. Tyto knihovny nabízení rozhraní pro tvorbu programů.

Nad dynamickými knihovnami pak je tvořena sada malých jednoúčelových programů, které lze spojovat unixovými rourami. Vstupy a výstupy programů jsou datové struktury ve formátu XML, dle návrhu [3]. Pro práci s XML formátem je použita knihovna `SAX`.

Programy, které jsou ovlivněny touto prací, jsou `arand2`, `adeterminize2` a `arun2`.

2.1 Program arand2

Program `arand2` slouží ke generování náhodných struktur. V tuto chvíli umí generovat konečné automaty a regulární výrazy, a po této práci i ohodnocené a neohodnocené stromy.

Povinným přepínačem `-t` se určí typ struktury, a to hodnotou `FSM` pro konečný automat, `RE` pro regulární výraz, `UT` pro neohodnocený strom a `RT` pro ohodnocený strom.

Další nepovinné přepínače jsou `-a` pro určení velikosti abecedy, `-d` pro určení hustoty přechodových pravidel automatu, `-n` pro určení počtu uzlů automatu nebo stromu, `-l` pro určení počtu listů regulárního výrazu, `-e` pro určení hloubky regulárního výrazu nebo stromu a `-r` pro určení maximálního počtu potomků uzlu stromu.

Přepínače ovlivňující generování stromu jsou `-a`, `-n`, `-e` a `-r`.

Výstupem programu je vygenerovaná struktura ve formátu XML.

2.2 Program adeterminize2

Program `adeterminize2` slouží k determinizaci libovolného automatu na vstupu. Algoritmus determinizace je zvolen podle struktury na vstupu. Pokud již je automat deterministický, je vrácen nezměněný.

Nedeterministický automat načítá ve formátu XML ze standardního vstupu, případně ze souboru při použití přepínače `-i`. Výstupem je deterministický automat ve formátu XML, případně chyba, pokud determinizace dané struktury není podporována.

2.3 Program arun2

Program `arun2` slouží k simulaci průchodu automatu lineárním řetězcem nebo stromem. Pracuje ve dvou režimech, kde buď rozhoduje, za řetězec nebo strom patří do jazyku automatu, nebo v něm hledá výskyty jazyka.

Typ průchodu se nastavuje přepínačem `-t` s hodnotou `accept` pro přijetí nebo `occurrences` pro hledání výskytů. Standardně je použit typ `accept`.

Přepínačem `-a` se určí soubor s XML reprezentací automatu a přepínačem `-i` soubor s XML reprezentací struktury k průchodu. Pokud není přepínač použit, načítá se ze standardního vstupu, kde se ale může nacházet jenom jedna struktura.

Výstupem při režimu přijetí je booleovská hodnota zapsaná v XML formátu. Při hledání výskytů je výstupem v XML formátu zapsaná množina indexů, kde výskyt začíná. V případě stromu je indexem pořadí uzlu při prefixovém zápisu.

Kapitola 3

Návrh datových struktur

V této kapitole jsou popsány datové struktury, o které se touto prací knihovna rozšířila.

3.1 Ohodnocený symbol

Ohodnocený symbol je dle definice 1.2.1 dvojice symbolu a kladné celočíselné arity. Je reprezentován třídou `RankedSymbol` nacházející se v souboru `RankedSymbol.h`.

Atributy jsou existující třídy `Label` a `Integer`. Label může být vytvořen číslem, znakem nebo řetězcem. Integer je jedním z wrapperů primitivních datových typů pro snadnější snadnější převody mezi XML reprezentací.

Převody mezi XML reprezentací zajišťují třídy `SymbolFromXMLParser`, nacházející se v souboru `SymbolFromXMLParser.h`, a `SymbolToXMLComposer` v souboru `SymbolToXMLComposer.h`.

3.2 Ohodnocený strom

Ohodnocený strom je dle definice 1.2.5 tvořen množinou ohodnocených symbolů a stromovou strukturou, kde uzel je ohodnocený symbol a uspořádaná množina uzlů o velikosti arity symbolu.

Ohodnocený strom reprezentují třídy `RankedTree` v souboru `RankedTree.h` a `RankedNode` v souboru `RankedNode.h`. Aby si třídy mohli manipulovat s privátními atributy, jsou vzájemně friend třídy.

Převody mezi XML reprezentací zajišťují třídy `TreeFromXMLParser`, nacházející se v souboru `TreeFromXMLParser.h`, a `TreeToXMLComposer` v souboru `TreeToXMLComposer.h`.

3.2.1 Třída RankedTree

Atributy třídy jsou ohodnocená abeceda `set<RankedSymbol>` a ukazatel na kořen stromu `RankedNode*`.

Třída umožňuje následující operace:

- *addSymbol* – přidá symbol do ohodnocené abecedy
- *removeSymbol* – odebere symbol z ohodnocené abecedy. Vyhodí výjimku, pokud se symbol ve stromu vyskytuje

3.2.2 Třída RankedNode

Atributy třídy jsou symbol `RankedSymbol`, ukazatel na rodiče `RankedNode*`, ukazatele na potomky `vector<RankedNode*>` a ukazatel na ohodnocený strom držící abecedu `RankedTree*`, do kterého uzel patří.

Pro uchování potomků byl zvolen vektor, protože potomci jsou uspořádaní, a jelikož počet potomků je pevně daný při vytváření uzlu aritou symbolu a dále je neměnný, není třeba používat žádné spojové struktury s rychlým přidáváním prvku, které jsou paměťově náročnější a nedovolují náhodný přístup.

Uzel je obousměrný, jelikož je zamýšlen na obecné použití.

Pro zachování konzistence nejsou povoleny některé operace manipulující se strukturou stromu a provádí se validace symbolu, které v případě neúspěchu vyhodí výjimku.

Objekt se vytváří se symbolem a vektorem potomků. Pokud počet potomků nesedí s aritou symbolu, je vyhozena výjimka. Ukazatele na rodiče a na strom jsou `NULL`. Všem potomkům je nastaven nový uzel jako rodič. Dokud není přiřazen uzel ke stromu, neprovádí se validace symbolu.

Konstrukcí objektu třídy `RankedTree` je stromová struktura přiřazena k abecedě, a proto se od kořene rekurzivně přiřadí strom každému uzlu a zkontroluje se, zda symbol patří do abecedy. Pokud ne, je vyhozena výjimka. Pokud již kořen byl přiřazen do stromu, je též vyhozena výjimka.

Třída umožňuje následující operace:

- *setSymbol* – nastaví uzlu symbol. Pokud má nový symbol rozdílnou aritu nebo nepatří do abecedy, je vyhozena výjimka
- *switchSubtree* – prohodí dva uzly. U obou uzlů se vyhledá jejich pozice ve vektoru potomků rodiče a prohodí se ukazatele. Pokud uzel nemá rodiče, je tedy kořen, vymění se ukazatel kořene stromu. Pokud jsou uzly z rozdílných stromů, rekurzivně se nastaví nový strom a zkontroluje symbol.

Prohození podstromů je validní operace, jelikož se u žádného uzlu nemění počet potomků ani symbol.

3.3 Neohodnocený strom

Neohodnocený strom je dle definice 1.2.4 tvořen množinou symbolů a stromovou strukturou, kde uzel je symbol a uspořádaná množina uzlů.

Neohodnocený strom je reprezentován, podobně jako ohodnocený strom, třídami `UnrankedTree` v souboru `UnrankedTree.h` a `UnrankedNode` v souboru `UnrankedNode.h`, které jsou vzájemně friend třídy.

Stejně jako u ohodnoceného stromu, převody mezi XML reprezentací jsou zajištěny třídami `TreeFromXMLParser` v souboru `TreeFromXMLParser.h` a `TreeToXMLComposer` v souboru `TreeToXMLComposer.h`.

3.3.1 Třída UnrankedTree

Atributy třídy jsou abeceda `set<LabeledSymbol>` a ukazatel na kořen stromu `UnrankedNode*`.

`LabeledSymbol` je existující třída reprezentující symbol.

Třída umožňuje následující operace:

- *addSymbol* – přidá symbol do abecedy
- *removeSymbol* – odebere symbol z abecedy. Vyhodí výjimku, pokud se symbol ve stromu vyskytuje

3.3.2 Třída UnrankedNode

Atributy třídy jsou symbol `LabeledSymbol`, ukazatel na rodiče `UnrankedNode*`, ukazatele na potomky `list<UnrankedNode*>` a ukazatel na neohodnocený strom držící abecedu `UnrankedTree*`, do kterého uzel patří.

Pro uchování potomků byl zvolen list, jelikož neohodnocený strom umožňuje manipulovat se strukturou a list umožňuje rychle vkládat prvky.

Problém s konzistencí narozdíl od ohodnoceného stromu nenastává při změně stromové struktury, stále se ale musí validovat symbol.

Objekt se vytváří symbolem a listem potomků. Podobně jako u ohodnoceného stromu jsou ukazatele na rodiče a na strom `NULL`. Všem potomkům je nastaven nový uzel jako rodič. Dokud není přiřazen uzel ke stromu, neprovádí se validace symbolu.

Při konstrukci `UnrankedTree` se rekurzivně všem uzlům přiřadí strom a zvaliduje se symbol a při neúspěchu, nebo pokud již byl kořen přiřazen ke stromu, se vyhodí výjimka.

Třída umožňuje následující operace:

- *setSymbol* – nastaví uzlu symbol. Pokud symbol nepatří do abecedy, je vyhozena výjimka
- *extract* – vyjme podstrom z neohodnoceného stromu. Odebere uzel z listu potomků rodiče a rekurzivně nastaví ukazatel na strom na `NULL`

- *pushBackChild* – přidá potomka na konec listu. Zkontroluje, zda je uzel kořen, zda nemá přiřazen strom a zda symboly podstromu patří do abecedy, případně vyhodí výjimku
- *insertSibling* – přidá sourozence za svojí pozici. Vyhledá svojí pozici v listu potomků rodiče a přidá uzel za svojí pozici. Zkontroluje, zda je uzel kořen, zda nemá přiřazen strom a zda symboly podstromu patří do abecedy, případně vyhodí výjimku
- *switchSubtree* – prohodí dva uzly. Postup je stejný, jako u ohodnoceného uzlu

3.4 Bottom-up konečný stromový automat

Dle definic 1.21 a 1.23 je stromový automat množina stavů, množina vstupních ohodnocených symbolů, množina finálních stavů a přechodová funkce realizovaná výčtem přechodových pravidel. Deterministická a nedeterministická verze se liší pouze v přechodové funkci.

Deterministický automat je reprezentován třídou `DFTA` v souboru `DFTA.h` a nedeterministický třídou `NFTA` v souboru `NFTA.h`.

Převod z XML reprezentace je zajištěn ve třídě `AutomatonFromXMLParser` v souboru `AutomatonFromXMLParser.h` a převod do XML reprezentace ve třídě `AutomatonToXMLComposer` v souboru `AutomatonToXMLComposer.h`.

Stav je reprezentován existující třídou `State` a množina stavů a finálních stavů se nachází v existující třídě `States`, ze které oba automaty dědí. Vstupní abecedu v obou případech reprezentuje `set<RankedSymbol>`.

V nedeterminististickém automatu je na levé straně přechodového pravidla ohodnocený symbol a množina stavů potomků, a na pravé straně výsledný stav. U deterministického automatu je na pravé straně místo množiny stavů jediný výsledný stav.

Pravidla jsou ukládána v `map`, kde klíčem je levá strana a hodnotou pravá strana. Levá strana je typu `pair<RankedSymbol, vector<State>>` a pravá strana `set<State>` u nedeterministického automatu a `State` u deterministického.

Třída umožňuje následující operace:

- *addState* – přidá stav
- *setStates* – nastaví množinu stavů. Pokud některé pravidlo neobsahuje stav z nové množiny, je vyhozena výjimka
- *removeState* – odebere stav. Pokud se stav vyskytuje v některém pravidlu, je vyhozena výjimka
- *addFinalState* – přidá finální stav. Pokud stav není v množině stavů automatu, je vyhozena výjimka

- *removeFinalState* – odebere finální stav
- *addInputSymbol* – přidá symbol do vstupní abecedy
- *removeInputSymbol* – odebere symbol ze vstupní abecedy. Vyhodí výjimku, pokud se symbol vyskytuje v některém přechodovém pravidlu
- *addTransition* – přidá přechodové pravidlo. Pokud symbol nepatří do vstupní abecedy nebo některý ze stavů nepatří do množiny stavů, je vyhozena výjimka. U nedeterministického automatu, pokud existuje levá strana, je stav přidán do množiny stavů na pravé straně. U deterministického automatu, pokud existuje pravidlo s rozdílnou pravou stranou, je vyhozena výjimka
- *removeTransition* – odebere přechodové pravidlo. U deterministického automatu, pokud existuje pravidlo s rozdílnou pravou stranou, je vyhozena výjimka

Kapitola 4

Generování náhodných označených stromů

Při generování označených stromů je třeba vygenerovat stromovou strukturu a abecedu, a z ní poté vygenerovat hodnoty uzlů. Parametry navrženého algoritmu jsou velikost abecedy, počet uzlů, hloubka stromu a maximální počet potomků.

Při generování neohodnoceného stromu lze všechny parametry buď splnit nebo konstatovat nemožnost konstrukce. U ohodnoceného stromu je problém s velikostí vstupní abecedy, jelikož ta závisí na vygenerované stromové struktuře a na počtu různých stupňů uzlů, který není předem znát. V navrženém algoritmu tedy není velikost abecedy ohodnoceného stromu zaručena.

4.1 Návrh generování stromové struktury

Principem algoritmu je postupně generovat uzly a držet si je v poli a to tak, že na levé straně pole budou uzly, ke kterým můžeme přidávat potomky, a na pravé straně budou uzly, které mají buď maximální hloubku nebo maximální počet potomků.

Každý nový uzel je přidán jako potomek náhodnému uzlu z levé strany pole, a je uložen na levou nebo pravou stranu, podle toho, zda dosáhl maximální hloubky. Pokud rodič dosáhl maximálního počtu potomků, je přesunut na pravou stranu.

Aby se splnila zadaná hloubka stromu, na začátku se nejprve vygeneruje cesta s délkou hloubky a uzly se uloží na začátek pole a list na konec.

Pokud budeme potomky uzlům přidávat postupně zleva, bude levá větev stromu mít vždy maximální délku, proto je potřeba po vygenerování stromu náhodně zamíchat levou větev. Toho dosáhneme tak, že u všech uzlů v levé větvi prohodíme levého potomka s náhodným dalším potomkem (nebo zůstane

na místě), případně při implementaci potomků kruhovým bufferem náhodně buffer otočíme.

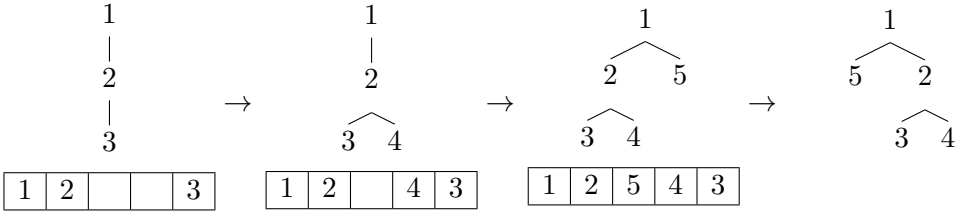
Algoritmus má tedy následující tři fáze pro počet uzlů n, hloubku stromu d a maximální počet potomků r:

1. Vygeneruj cestu délky $d+1$ a vnitřní uzly ulož na začátek pole a list na konec.

2. Dokud není v poli n uzlů, vyber náhodně uzel v levé části pole a přidej mu potomka. Pokud je hloubka potomka d, ulož ho na začátek pravé části pole, jinak na konec levé části pole. Pokud počet potomků náhodného uzlu je r, přesuň ho na začátek pravé části pole a na jeho pozici přesuň poslední uzel z levé části pole.

3. Náhodně zamíchej levou větev stromu.

Příklad 4.1. $n = 5, d = 2, r = 3$



Obrázek 4.1: Příklad generování náhodné stromové struktury

Algoritmus 1 Generování stromové struktury

Vstup: $d \in \mathbb{N}, n \in \mathbb{N}, r \in \mathbb{N}, d < n <= r^{d+1} - 1$
Výstup: Tree

```
nodes[1...n]
nodes[1] ← Node()
for i ← 2 to d + 1 do
    nodes[i] ← Node()
    nodes[i − 1].addChild(nodes[i])
end for
nodes[n] ← nodes[d + 1]

i_a ← d + 1
i_f ← n − 1
while i_f ≥ i_a do
    i ← rand(1...i_a)
    t ← Node()
    nodes[i].addChild(t)
    if t.depth < d then
        nodes[i_a] ← t
        i_a ← i_a + 1
    else
        nodes[i_f] ← t
        i_f ← i_f − 1
    end if
    if nodes[i].rank = r then
        nodes[i_f] ← nodes[i]
        nodes[i] ← nodes[i_a − 1]
        i_a ← i_a − 1
        i_f ← i_f − 1
    end if
end while
shuffleLeftBranch(nodes[0])
return nodes[0]
```

4.2 Návrh generování neohodnoceného stromu

Při generování neohodnoceného stromu je potřeba vygenerovat stromovou strukturu a abecedu. Pro potřeby automatové knihovny postačují jako symboly znaky anglické abecedy. Abeceda je tedy náhodná podmnožina anglické abecedy o velikosti zadané parametrem.

Po vygenerování abecedy a stromové struktury je každému uzlu přiřazen náhodný symbol z abecedy.

4.3 Návrh generování ohodnoceného stromu

Při generování ohodnoceného stromu je stejně jako u neohodnoceného stromu potřeba vygenerovat stromovou strukturu a abecedu. Rozdílem je, že abeceda je vázána na vygenerovanou strukturu.

Každému symbolu musí být přiřazena arita a to nejlépe taková, která je stupněm některého uzlu stromu, aby ke každému uzlu mohl být přiřazen symbol.

Myšlenkou tedy je nejprve vygenerovat stromovou strukturu a z ní získat množinu stupňů uzlů, které se v ní vyskytují. Následně vygenerovat neohodnocenou abecedu a rozdělit prvky této množiny mezi symboly.

Náhodného rozdělení můžeme dosáhnout tak, že oddělíme skupiny symbolů, kterým pak přiřadíme jednu aritu. Oddělení skupin provedeme vygenerováním množiny indexů, které budou sloužit jako oddělovače.

Tímto způsobem je každému symbolu přiřazena právě jedna arita. Pokud je zadaná velikost abecedy menší než počet různých stupňů uzlů, je nutné velikost abecedy zvětšit, aby každému uzlu mohl být přiřazen ohodnocený symbol, proto tento parametr není zaručen. Velikost abecedy lze zaručit pouze v případě, že že zadaný maximální stupeň uzlů je menší, než velikost abecedy.

Pokud by symboly mohli mít více arit, strom by stále byl validní, ale to by mohlo situaci zhoršit, jelikož se jedná o různé symboly a velikost abecedy by tedy nemusela být zaručena ani v případě omezení stupňů uzlů, případně by se musel navrhnout jiný a pravděpodobně složitější způsob generování.

Po vygenerování stromové struktury a ohodnocené abecedy je každému uzlu přiřazen náhodný ohodnocený symbol v závislosti na jeho stupni.

Algoritmus 2 Generování ohodnocené abecedy struktury

Vstup: $a \in N, ranks \subset \mathbb{N}$
Výstup: F

```
a ← max(a, |ranks|)
Σ ← generateUnrankedAlphabet(a)
S ← {1, a + 1}
while |S| ≠ |ranks| + 2 do
    S ← S ∪ rand(2 ... a)
end while
for i ← 1 to |S| − 1 do
    for j ← S_i to S_{i+1} − 1 do
        F ← F ∪ (Σ_j, ranks_i)
    end for
end for
return F
```

4.4 Implementace

Implementace se nachází v souboru `RandomTreeFactory.cpp`.

Aby se předešlo limitacím navrhnutých tříd stromů, je zde vytvořena pomocná struktura pro reprezentaci stromů, na které se generování provádí.

Pro náhodnost symbolů přiřazených k aritě není neohodnocená abeceda generována jako množina, ale jako vektor. To je dosaženo vytvořením vektoru anglických písmen, jeho náhodné promíchání a oříznutí na zadaný počet symbolů.

Ohodnocená abeceda je pro potřeby generování reprezentována mapou, kde klíč je arita a hodnota je vektor symbolů. Místo množiny stupňů uzlů je předána funkci tato mapa, do které se předvyplní stupně uzlů jako klíče a hodnoty jako prázdné vektory.

Kapitola 5

Determinizace bottom-up konečného stromového automatu

Při průchodu stromu nedeterministickým automatem se může uzel nacházet ve více stavech najednou. Principem determinizace proto je vytvořit deterministický automat, který bude kopírovat průběh nedeterministického. K tomu, aby byl automat deterministický je třeba, aby se každý uzel nacházel pouze právě v jednom stavu.

Pokud se tedy uzel nachází ve více stavech, vytvoří se nový stav, který označuje, že se uzel nachází ve všech původních stavech. Přidáním nového stavu je potřeba přidat také další přechodová pravidla. Všechna pravidla, která obsahují na levé straně některý z původních stavů, mohou obsahovat i stav nový. Jelikož v BU NFTA se na levé straně nachází uspořádaná množina stavů, musí se vytvořit pravidla pro všechny variace nových stavů s původními.

5.1 Návrh algoritmu

V této kapitole popíši naivní algoritmus z [2] a návrh efektivnějšího algoritmu.

5.1.1 Naivní algoritmus

Principem algoritmu je vygenerovat všechny levé strany přechodových pravidel, tedy pro každý symbol vygenerovat všechny variace s opakováním ze všech stavů vytvořených kombinací stavů automatu, a k těmto pravidlům najít odpovídající pravé strany.

Na pravé straně se bude nacházet stav, který tvoří všechny stavy, které se nacházejí na pravých stranách pravidel, kde každý stav na levé straně je obsažen ve stavu nového pravidla.

Koncové stavy deterministického automatu jsou všechny stavy, které obsahují alespoň jeden koncový stav nedeterministického automatu.

Algoritmus 3 Naivní determinizace BU NFTA

Vstup: NFTA $A = (Q, F, Q_f, \Delta)$
Výstup: DFTA
$Q_d, \Delta_d, Q_{df} \leftarrow \emptyset$
// pro všechny levé strany přechodového pravidla
for all $t_d = (f(q_1 \dots q_n) \rightarrow q), f \in F, q_i \in Q^*, q = \emptyset, n = arity(f)$ **do**
  // najdi všechny stavy na pravé straně
  for all $t \in \Delta$ **do** // $t = (f(q_1 \dots q_n) \rightarrow Q_t), Q_t \subseteq Q$
    if $(t.f = t_d.f)$ **and** $(t.q_1 \in t_d.q_1)$ **and** $\dots$ **and** $(t.q_n \in t_d.q_n)$ **then**
      $t_d.q \leftarrow t_d.q \cup t.Q_t$
    end if
  end for
  $\Delta_d \leftarrow \Delta_d \cup t_d$
  $Q_d \leftarrow Q_d \cup q$
end for

// najdi finální stavy
for all $q_d \in Q_d, q_d \cap Q_f \neq \emptyset$ **do**
  $Q_{df} \leftarrow Q_{df} \cup q_d$
end for
return $DFTA(Q_d, F, Q_{df}, \Delta_d)$

Algoritmus předpokládá, že se stavem lze nakládat jako s množinou stavů a opačně.

5.1.2 Efektivní algoritmus

Problémem naivního algoritmu je, že není citlivý na vstupní data. Vytváří nedosažitelné stavy a tím i zbytečná pravidla. K některým novým pravidlům ani nemusí být nalezený žádný stav na pravé straně.

Navrhovaný algoritmus proto nejprve překopíruje pravidla z nedeterministického automatu a pokud narazí na pravidlo s více stavy na pravé straně vytvoří z nich jeden deterministický stav a uloží ho ke zpracování.

Z každým novým stavem poté prochází levé strany pravidel deterministického automatu a pokud narazí na stav nedeterministického automatu, který je obsažen v novém stavu, přidá do deterministického automatu nové pravidlo, kde je tento stav nahrazen novým.

Tímto postupem by se mohla některá pravidla vytvářet zbytečně vícekrát, protože pokud bychom například zpracovávali stav 01 měli následující levou stranu:

$$f(\dots 1 \dots 1 \dots)$$

Z toho bychom vytvořili pravidla s levými stranami:

$$f(\ldots 01 \ldots 1 \ldots)$$
$$f(\ldots 1 \ldots 01 \ldots)$$

Při průchodu obou pravidel bychom pak vytvářeli pravidlo s levou stranou:

$$f(\ldots 01 \ldots 01 \ldots)$$

Proto u pravidel vytvořených aktuálně zpracovávaným stavem se nemá smysl zabývat stavy na pozicích do pozice tohoto stavu. Proto si při zpracování každého nového stavu budeme u každého pravidla držet pozici, před kterou jsme provedli změnu stavu a od té začneme zpracovávat pravidlo, až k němu dojdeme.

Jediná situace, kdy můžeme vytvořit duplicitní pravidlo, je, pokud bychom měli například levé strany:

$$f(\ldots 0 \ldots)$$
$$f(\ldots 1 \ldots)$$

U obou pravidel bychom dosadili stav 01.

Algoritmus 4 Efektivní determinizace BU NFTA

Vstup: NFTA $A = (Q, F, Q_f, \Delta)$
Výstup: DFTA

```
Q_d ← Q, Q_df ← Q_f
Δ_d, Q_new ← ∅
// překopíruj pravidla a najdi nové deterministické stavy
for all t ∈ Δ do // t = (f(q_1 ... q_n) → Q_t), Q_t ⊆ Q
  q_d = t.Q_t
  Δ_d = Δ_d ∪ (t.f(t.q_1 ... t.q_n) → q_d)
  if |t.Q_t| < 1 and q_d ∉ Q_d then
    Q_d ← Q_d ∪ q_d
    Q_new ← Q_new ∪ q_d
  end if
end for
for all q_new ∈ Q_new do
  Δ_new = Δ_d // Δ_new = {t, t.i ← 0}
  for all t ∈ Δ_new do
    for i ← t.i to arity(t.f) do
      if |t.q_i| = 1 and t.q_i ∈ q_new then
        t_d ← t
        t_d.q_i ← q_new
        // najdi všechny stavy na pravé straně
        for all t_n ∈ Δ do
          if (t_n.f = t_d.f) and (t_n.q_1 ∈ t_d.q_1) and ... and (t_n.q_n ∈ t_d.q_n)
          then
            t_d.q ← t_d.q ∪ t_n.Q_t
          end if
        end for
        if t_d.q ∉ Q_d then
          Q_d ← Q_d ∪ t_d.q
          Q_new ← Q_new ∪ t_d.q
        end if
        if i ≠ arity(t_d.f)and t_d ∉ Δ_d then
          Δ_new ← Δ_new ∪ (t_d, t_d.i ← i + 1)
        end if
        Δ_d = Δ_d ∪ t_d
      end if
    end for
  end for
  if q_new ∩ Q_f ≠ ∅ then
    Q_df ← Q_df ∪ q_new
  end if
end for
return DFTA(Q_d, F, Q_df, Δ_d)
```

5.2 Implementace

Implementace se nachází v souboru `DeterminizeNFTAPart.cxx`.

V algoritmu se iteruje přes kontejner, do kterého jsou za běhu cyklu přidávány nové prvky. S tím má většina standardních kontejnerů problém, protože přidáním nebo odebráním prvku se mohou zneplatnit iterátory. S tím nemá problém fronta `deque`.

Nové stavy čekající na zpracování jsou tedy ukládány do `deque`. Přechodová pravidla jsou uložená v mapě, ale jsou také za běhu přidávána, proto se před průchodem cyklu musí do `deque` zkopírovat.

Implementace nepovažuje za ekvivalentní stav a množinu stavů. Místo toho se používají existující převodní funkce `createDFAState` a `recreateNFAStates`.

Kapitola 6

Přijetí stromu deterministickým bottom-up konečným stromovým automatem

Přijetí datové struktury je hlavním účelem stavových strojů. V této kapitole je popsán algoritmus pro přijetí stromu deterministickou verzí stromového automatu.

6.1 Návrh algoritmu

Algoritmus vychází z definice 1.22. Rekurzivní funkcí se určí stav kořene a ohodnocený strom je přijatý, pokud tento stav patří do množiny finálních stavů. Jelikož je automat deterministický, výsledkem této funkce je vždy právě jeden stav.

Algoritmus 5 DFTA stav uzlu ohodnoceného stromu (calculateState)

Vstup: DFTA $A = (Q, F, Q_f, \Delta)$, RankedNode n
Výstup: $q \in Q$
return $\Delta(n.f, calculateState(A, n[1]), \ldots, calculateState(A, n[arity(n.f)])$

Algoritmus 6 Přijetí ohodnoceného stromu DFTA automatem

Vstup: DFTA $A = (Q, F, Q_f, \Delta)$, RankedTree t
Výstup: Boolean
return $calculateState(A, t.root) \in Q_f$

6.2 Implementace

Implementace se nachází v soboru `Accept.h` v metodách `calculateState` a `accept` pro DFTA. Pokud v metodě `calculateState` není nalezeno přechodové pravidlo pro konkrétní situaci, vrací metoda stav `FAILSTATE`, což je konstanta deklarovaná ve třídě `State`. Pokud některý z potomků vrátí `FAILSTATE`, je metoda přerušena a vrací také `FAILSTATE`, jelikož je už poté jasné, že strom nebude přijat.

Kapitola 7

Přijetí stromu nedeterministickým bottom-up konečným stromovým automatem

Přijetí netederministickým automatem je složitější kvůli nejednoznačnosti průchodu, je proto nutné simulovat průchod všemi možnosti. V této kapitole je popsán algoritmus pro takový průchod.

7.1 Návrh algoritmu

Uzel stromu se, oproti deterministickému automatu, může nacházet ve více stavech zároveň, proto musíme počítat se všemi těmito stavy. Rekurzivní funkce tedy bude vracet množinu stavů, ve kterých se uzel nachází.

Pro zjištění stavů uzlu tedy zjistíme stavy potomků a poté vyhledáme přechodová pravidla, která pro každého potomka mají na levé straně stav obsažený ve stavech, ve kterých se potomek nachází. Uzel se pak nachází ve všech stavech na pravých stranách těchto pravidel.

Strom je automatem přijatý, pokud alespoň jeden ze stavů kořene je finální stav.

Algoritmus 7 NFTA stavy uzlu ohodnoceného stromu (calculateStates)

Vstup: NFTA $A = (Q, F, Q_f, \Delta)$, RankedNode n
Výstup: $Q_r \subseteq Q$

```
Q_Qr ← ∅
ar ← arity(n.f)
for i ← 1 to ar do
  Q_Qr[i] ← calculateStates(A, n[i])
end for

for all tr ∈ Δ do
  if tr.f = n.f and tr.q_1 ∈ Q_Qr[1] and ... and tr.q_ar ∈ Q_Qr[ar] then
    Q_r ← Q_r ∪ tr.Q_t
  end if
end for
return Q_r
```

Algoritmus 8 Přijetí ohodnoceného stromu DFTA automatem

Vstup: NFTA $A = (Q, F, Q_f, \Delta)$, RankedTree t
Výstup: Boolean

```
return calculateStates(A, t.root) ∩ Q_f ≠ ∅
```

7.2 Implementace

Implementace se nachází v souboru `Accept.h` v metodách `calculateStates` a `accept` pro NFTA.

U nedeterministického vyhodnocení není třeba používat `FAILSTATE`, jelikož metoda v případě neúspěchu vrací prázdnou množinu stavů.

Pokud některý potomek vrátí prázdnou množinu, je metoda ukončena a také vrací prázdnou množinu.

Kapitola 8

Vyhledávání výskytů vzoru deterministickým bottom-up konečným stromovým automatem

Při vyhledávání výskytů v ohodnoceném stromu hledáme všechny podstromy, které patří do jazyka přijímaného automatem. Každý takový podstrom bude končit v listech původního stromu, což vzhledem k tomu, že vyhodnocujeme od listů, zjednodušuje algoritmus.

8.1 Návrh algoritmu

Výskytem rozumíme každý uzel, jehož stav je v množině finálních stavů. Algoritmus je proto upravenou verzí algoritmu z kapitoly 6.1. Množina výskytů je reprezentována množinou indexů uzlů při prefixovém průchodu.

Algoritmus 9 DFTA stav uzlu ohodnoceného stromu s výskyty (calculateState)

Vstup: DFTA $A = (Q, F, Q_f, \Delta)$, RankedNode n, $i \in \mathbb{N}$, $O \subset \mathbb{N}$
Výstup: $q \subseteq Q$, i, O

```
idx ← i
ar ← arity(n.f)
q_r ← Δ(n.f, calculateState(A, n[1], i, O), ..., calculateState(A, n[ar], i, O)
if q_r ∈ Q_f then
    O ← O ∪ idx
end if
return q
```

Algoritmus 10 Hledání výskytů jazyka DFTA automatem

Vstup: DFTA $A = (Q, F, Q_f, \Delta)$, RankedTree t
Výstup: $O \in \mathbb{N}$

```
O ← ∅
i ← 0
calculateState(A, t.root, i, O)
return O
```

8.2 Implementace

Implementace se nachází v souboru `Occurrences.h` v metodách `occurences` pro DFTA.

Výsledná množina výskytů a čítač uzlu jsou v rekurzivní funkci předávány referencí jako vstupně výstupní parametry.

Pokud některý potomek vrátí `FAILSTATE`, nelze v tomto případě hned ukončit vyhodnocení, jelikož se výskyt může nacházet ještě v jiném podstromu. Místo toho je vrácen `FAILSTATE` až po vyhodnocení potomků.

Kapitola 9

Testování

Testování je prováděno s knihovnou `cppunit`. V souborech `AutomatonTest.h` a `TreeTest.h` se nachází sady testovacích funkcí.

Každá funkce provádí řadu operací a za pomoci knihovních maker testuje, zda je jejich výsledek dle očekávání.

Testování je pak spuštěno při kompilaci.

Závěr

Cílem této práce bylo naimplementovat v Automatové knihovně potřebné datové struktury pro konečné stromové automaty a algoritmy nad těmito strukturami.

Z datových struktur byli naimplementovány reprezentace ohodnoceného a neohodnoceného označeného uspořádaného stromu (dále jen ohodnocený a neohodnocený strom), a deterministického a nedeterministického bottom-up konečného stromového automatu (dále jen deterministický a nedeterministický stromový automat).

Z algoritmů byl navržen a implementován algoritmus pro generování ohodnoceného a neohodnoceného stromu, popsán naivní algoritmus determinizace stromového automatu a navržena a implementována efektivnější verze, navržen a implementován algoritmus pro simulaci průchodu deterministickým a nedeterministickým stromovým automatem a vyhledávání výskytů jazyka deterministického stromového automatu v ohodnoceném stromě.

Na tuto práci může navázat ještě mnoho dalších rozšíření. Z algoritmů nad stromovým automatem lze naimplementovat například ještě redukci a minimalizaci. Mimo deterministický a nedeterministický stromový automat ještě existuje například nedeterministický stromový automat s ϵ-přechody. Dále je možné naimplementovat i top-down verzi automatu, stromové gramatiky a stromové regulární výrazy. Možné jsou též i převody automatů, jelikož konečný stromový automat lze převést například na zásobníkový automat.

Literatura

[1] Wikipedia: Tree automaton — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Tree%20automaton&oldid=625685926`, 2015, [Online; accessed 05-May-2015].

[2] Comon, H.; Dauchet, M.; Gilleron, R.; aj.: Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007, release October, 12th 2007.

[3] Žák, M.: *Automatová knihovna – vnitřní a komunikační formát*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[4] Veselý, J.: *Automatová knihovna – determinizace konečných a zásobníkových automatů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[5] Pecka, T.: *Automatová knihovna – převody mezi regulárními výrazy, regulárními gramatikami a konečnými automaty*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

Příloha **A**

Seznam použitých zkratek

NFTA Nedeterministický konečný stromový automat (Nondeterministic Finite Tree Automaton)

DFTA Deterministický konečný stromový automat (Deterministic Finite Tree Automaton)

BU Zdola nahoru (Bottom-Up, typ stromového automatu)

TD Shora dolů (Top-Down, typ stromového automatu)

XML eXtensive Markup Language

Příloha B

Uživatelská příručka

B.1 Požadavky

Pro úspěšnou kompilaci je vyžadováno:

- program `make` ve verzi alespoň 3.9
- program `g++` ve verzi alespoň 4.8
- knihovna `binutils-dev`
- knihovna `libiberty-dev`
- knihovna `libcppunit-dev`
- knihovna `libtclap-dev`

B.2 Instalace

Zdrojové kódy se zkompilují příkazem

```
$ make release
```

Jednotlivé aplikace se pak nacházejí v adresáři `bin-release/`. Pro spouštění programů je potřeba mít tuto složku v proměnné `PATH`, případně aplikace spouštět z této složky.

Uvedených příkladech jsou cesty souborů uváděné relativně vůči adresáři `bin-release/`.

B.3 Generování stromu programem arand2

Přepínače ovlivňující generování stromu jsou:

- `-t <UT|RT>, --type <UT|RT>` – povinný argument typu generované struktury. `UT` generuje neohodnocený strom a `RT` ohodnocený strom
- `-a <integer>, --alphabetSize <integer>` – specifikuje velikost abecedy stromu
- `-n <integer>, --nodes <integer>` – specifikuje počet uzlů stromu
- `-e <integer>, --depth <integer>` – specifikuje hloubku stromu
- `-r <integer>, --maxRank <integer>` – specifikuje maximální stupeň uzlů

Pro vizualizaci vygenerovaného stromu lze na výstup programu použít pomocný program `tniceprint`

Příklad B.1. `$ arand2 -t RT -a 3 -n 5 -e 3 -r 2 | tniceprint`

B.4 Determinizace bottom-up konečného stromového automatu programem adeterminize2

Program očekává XML reprezentaci nedeterministického automatu buď na vstupu případně, s přepínačem, v souboru.

Přepínač je:

- `-i <file>, --input <file>` – specifikuje soubor s nedeterministickým automatem

Příklad B.2. `$ adeterminize2 -i ../examples2/automaton/NFTA.xml`

B.5 Přijetí stromu a hledání výskytů ve stromu programem arun2

Program očekává XML reprezentaci automatu a (v tomto případě) stromu buď na vstupu nebo, s přepínačem, v souboru. Na vstupu může být jen jedna struktura.

Při přijetí je na výstupu v XML reprezentaci booleovská hodnota, při hledání výskytů list indexů uzlů při prefixovém číslování.

Přepínače jsou:

- `-t <occurrences|accept>, --type <occurrences|accept>` – zvolí typ průchodu, standardně `accept`
- `-a <file>, --automaton <file>` – specifikuje soubor s automatem
- `-i <file>, --input <file>` – specifikuje soubor s ohodnoceným stromem

Příklad B.3.
```
$ arun2 -a ../examples2/automaton/NFTA.xml -i
../examples2/tree/RankedTree.xml
```

Příklad B.4.
```
$ arun2 -a ../examples2/automaton/DFTA.xml -i
../examples2/tree/RankedTree.xml
```

Příklad B.5.
```
$ ./arun2 -a ../examples2/automaton/DFTA.xml -i
../examples2/tree/RankedTree.xml -t occurrences
```

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu CD
- `automata-library` ..... zdrojové kódy Automatové knihovny
- `text`
  - `BP_Plachy_Stepan_2015.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojové soubory práce ve formátu LaTeX