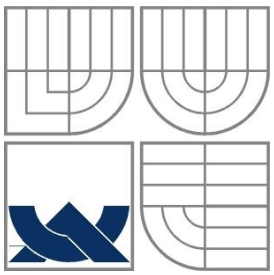
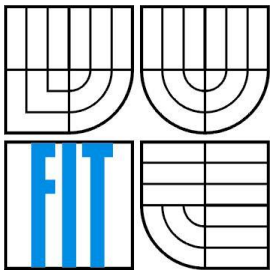


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

STROJOVÉ UČENÍ PRO ODVOZOVÁNÍ FORMÁLNÍCH JAZYKŮ

MACHINE LEARNING FOR FORMAL LANGUAGE MODEL INFERENCE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR BARDONEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBYNĚK KŘIVKA Ph. D.

BRNO 2015

Abstrakt

Tato bakalářská práce se zabývá odvozováním modelů formálních jazyků. Jedná se o vědní disciplínu na poli výzkumu umělé inteligence. Cílem je vytvořit aplikaci umožňující automatické vytvoření modelu, v podobě konečného automatu, pro neznámý formální jazyk na základě množiny řetězců neznámého formálního jazyka s využitím upravené metody strojového učení.

Abstract

This bachelor thesis deals with the formal language model inference, which is a science discipline on the research field of artificial intelligence. The aim is to create an appliaction that allows the automatic inference of model, such as the finite state machine, for an unknown formal language from the set of the strings of the unknown formal language using the modified machine learning method.

Klíčová slova

Konečné automaty, formální jazyk, abeceda, strojové učení, genetické algoritmy

Keywords

Finite automata, formal language, alphabet, machine learning, genetic algorithms

Citace

Petr Bardonek: Strojové učení pro odvozování modelů formálních jazyků, bakalářská práce, Brno, FIT VUT v Brně, 2015

Strojové učení pro odvozování modelů formálních jazyků

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky Ph. D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Petr Bardonek

20. května 2015

Poděkování

Na tomto místě chci poděkovat vedoucímu mé bakalářské práce, panu Ing. Zbyňkovi Křivkovi Ph. D. za cenné rady, čas a ochotu při konzultacích.

Obsah

1 Úvod

Výzkumníci se vždy snažili formalizovat přirozený jazyk, avšak ten se neustále mění a jeho formalizace je takřka nemožná. S příchodem počítačů přišla potřeba formalizovat jazyk, pomocí něhož se bude komunikovat s počítači jednotným způsobem na celém světě, to dopomohlo vzniknout formálním jazykům. Spolu se vznikem formálních jazyků zde vznikl také prostor pro problémy při jejich modelování. Existuje mnoho algoritmů pro odvozování modelů formálních jazyků z řetězců, jenž do daného jazyka patří. Rozhodl jsem se řešit tento problém v bakalářské práci netradičním způsobem, pomocí strojového učení. Součástí této práce je rovněž aplikace, která pomocí algoritmu metody strojového učení automaticky vytváří model formálního jazyka.

Využití této aplikace je například při zpracovávání přirozeného jazyka nebo kdekoliv, kde je třeba hledat vzorce v řetězcích, jako například v bioinformatice, při analýze sekvencí proteinů a nukleonových kyselin (DNA, RNA) nebo jiných dat typických pro analýzu v bioinformatice.

Práce se skládá hned z několika problémů. Nejdříve je třeba vybrat vhodnou metodu strojového učení a následně ji vhodně upravit. Po teoretické úpravě zvolené metody ji posléze naimplementuji. Pomocí experimentů budu naimplementovanou aplikaci nadále upravovat se záměrem dosáhnout zamýšlených výsledků.

Technická správa obsahuje 10 kapitol. V úvodních kapitolách uvedu základní teorii k formálním jazykům, jejich modelování, a strojovému učení. V dalších dvou kapitolách se budu věnovat podrobněji teorii zvoleného modelu formálních jazyků a zvolené metody strojového učení. V šesté kapitole probereme úpravy vybrané metody strojového učení a návrh samotné aplikace. Další kapitola bude popisovat implementaci programové části práce. V předposlední kapitole budou uvedeny prováděné testy, jejich výsledky a zhodnocení. V závěrečné kapitole shrneme celou práci.

2 Formální jazyky

V této kapitole se budeme zabývat základní teorií formálních jazyků a jejich modelování. Definice formálního jazyka se liší podle vědní disciplíny, ve které se právě pohybujeme. Nás zajímá definice formálního jazyka z pohledu informatiky.

2.1 Základy

Než si uvedeme definici formálního jazyka jako takového, musíme nejdříve projít základy, abychom pak pochopili definici jazyka samotného.

Definice 2. 1. 1 [7]

Abeceda je libovolná neprázdná konečná množina. Prvky abecedy nazýváme symboly.

Jako příklad abecedy si můžeme uvést jednoduchou množinu {a, b} o dvou prvcích.

Definice 2.1.2 [7]

Nechť Σ je abeceda. Řetězec nad touto abecedou Σ je pak konečná posloupnost symbolů z této abecedy. Prázdnou posloupnost nazýváme prázdný řetězec a označujeme jej ε.

Pokud budeme vycházet z výše uvedené definice abecedy a určíme si abecedu Σ={a, b}, pak jedno z možných slov nad touto abecedou je například aabb. Prázdný řetězec je řetězec nad libovolnou abecedou.

Definice 2. 1. 3 [7]

Nechť $w = a_1 \dots a_n$ je řetězec nad abecedou Σ, $a_i \in \Sigma$, $1 \leq i \leq n$, pro $n \geq 1$. Pak délku tohoto slova označujeme $|w| = n$.

Mějme řetězce ε, aa , aabb, aba nad abecedou Σ pak délky jednotlivých uvedených řetězců jsou $|\varepsilon| = 0$ (délka prázdného řetězce je vždy nulová), $|aa| = 2$, $|aabb| = 4$, $|aba| = 3$.

Definice 2. 1. 4 [7]

Zřetězením řetězců rozumíme jejich spojení. Nechť u a v, jsou dva různé řetězce nad abecedou Σ, pak zřetězením těchto dvou řetězců vznikne nový řetězec uv nad abecedou Σ.

Mějme řetězce ε, aa , aabb, aba nad abecedou Σ. Různým pořadím a kombinacemi zřetězování řetězců získáme různé nové řetězce nad abecedou Σ. Například εaa = aaε = aa, spojení řetězce aabb a aa pak dostaneme aabbaa atd., zde si můžeme všimnout, že spojením řetězce s řetězcem prázdným dostaneme řetězec spojovaný s prázdným řetězcem, toto platí pro spojení jakéhokoliv řetězce s řetězcem prázdným.

Další vlastnosti zřetězování řetězců je jejich asociativita, např. Mějme řetězce x, y, z nad abecedou Σ a tento příklad: (xy)z = x(yz) = xyz, závorky zde mají stejnou roli jako u jakéhokoliv matematického příkladu, tedy označují, které řetězce se spojí jako první.

Pokud se vrátíme k definici délky řetězce, je zřejmé, že po aplikaci spojení na dva řetězce různé délky vznikne řetězec délky součtu délek spojovaných řetězců. Mějme řetězce x, y, ε nad abecedou Σ pak

platí $|xy| = |x| + |y|$, pro prázdný řetězec samozřejmě platí $|x| + |\varepsilon| = |x|$, neboť jak jsme si uvedli výše, pro prázdný řetězec platí $|\varepsilon| = 0$, že jeho délka je nulová.

Definice 2. 1. 5 [7]

Nechť w je řetězec nad abecedou Σ. Pro $i \geq 0$, i-tá mocnina řetězce w, w^i, je definována:

1. $w^0 = \varepsilon$
2. pro $i \geq 1$: $w^i = ww^{i-1}$

Mějme řetězec x = ab nad abecedou Σ, pokud chceme například řetězec x^3 pak začneme od nejnižší mocniny, tedy od nuly, a jednoduchou iterací dosáhneme výsledného řetězce:

$$x^0 = \varepsilon$$

$$x^1 = ab\varepsilon$$

$$x^2 = abab\varepsilon$$

$$x^3 = ababab\varepsilon$$

Jak je patrné z výše uvedeného příkladu postupně připojujeme umocňovaný řetězec z levé strany, prázdný řetězec se ve výsledku samozřejmě nepíše, zde je proto, aby bylo patrné, že nikam nezmizel, a že je nultou mocninou jazyka.

Pokud chceme spojit stejné dva řetězce s různou mocninou, pak můžeme tento vztah vyjádřit jako jediný řetězec, jehož mocnina se bude rovnat součtu mocnin jednotlivých řetězců:

$$x^i x^j = x^j x^i = x^{i+j}$$

Definice 2. 1. 6 [7]

Nechť *w* je řetězec nad abecedou Σ. Reverzace řetězce *w, reversal(w),* je definována:

1. pokud $w = \varepsilon$ pak $reversal(\varepsilon) = \varepsilon$
2. pokud $w = a_1 \ldots a_n$ pak $reversal(a_1 \ldots a_n) = a_n \ldots a_1$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \ldots, n$

Mějme řetězec x = abcd nad abecedou Σ, pak pokud na tomto řetězci provedeme jeho reverzaci, dostaneme řetězec se symboly přesně v opačném pořadí:

$$x = abcd$$

$$reversal(x) = dcba$$

Definice 2. 1. 7 [7]

Nechť *x* a *y* jsou dva řetězce nad abecedou Σ; *x* je prefixem *y*, pokud existuje řetězec *z* nad abecedou Σ, přičemž platí $xz = y$

Mějme řetězec x = abcd nad abecedou Σ, podle výše zmíněné definice vyjádříme postupně všechny prefixy daného řetězce x:

$$\text{Prefixy řetězce } x = \{\varepsilon, a, ab, abc, abcd\}$$

Z výše uvedeného lze odvodit, že prefixem řetězce *x*, může být i celý řetězec, avšak prázdný řetězce a celý řetězec, označujeme jako prefixy nevlastní, tedy ty, jež nejsou ani prázdným ani celým řetězcem označujeme jako prefixy vlastní.

Definice 2. 1. 8 [7]

Nechť *x* a *y* jsou dva řetězce nad abecedou Σ; *x* je sufixem *y*, pokud existuje řetězec *z* nad abecedou Σ, přičemž platí *zx* = *y*

Mějme řetězec x = abcd nad abecedou Σ, podle výše zmíněné definice vyjádříme postupně všechny sufixy daného řetězce x:

Sufixy řetězce *x* = {ε, d, cd, bcd, abcd}

Z výše uvedeného lze odvodit, že sufixem řetězce *x*, může být i celý řetězec, avšak prázdný řetězce a celý řetězec, označujeme jako sufixy nevlastní, tedy ty, jež nejsou ani prázdným ani celým řetězcem označujeme jako sufixy vlastní.

Definice 2. 1. 9 [7]

Nechť *x* a *y* jsou dva řetězce nad abecedou Σ. x *je podřetězec y,* pokud existují řetězce *z* a *z'*, nad abecedou Σ přičemž platí *zxz' = y*.

Mějme řetězec x = 1010 nad abecedou Σ, podle výše zmíněné definice vyjádříme postupně všechny podřetězce daného řetězce x:

Podřetězce řetězce x = {ε, 1, 0, 10, 01, 101, 010, 1010}

Z výše uvedeného lze odvodit, že podřetězcem řetězce *x*, může být i celý řetězec, avšak prázdný řetězce a celý řetězec, označujeme jako podřetězce nevlastní, tedy ty, jež nejsou ani prázdným ani celým řetězcem označujeme jako podřetězce vlastní.

Nyní když jsme si vysvětlili co je to abeceda, řetězce a všechny jeho aspekty, můžeme přejít k definici a následnému vysvětlení jazyka.

2.2 Jazyk

Definice 2. 1. 10 [7]

Nechť Σ je abeceda a Σ^* značí množinu všech řetězců nad touto abecedou. Každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad abecedou Σ.

Ještě si uvedeme, že Σ^+ značí množinu všech řetězců, ale bez prázdného řetězce $\Sigma^* - \{\varepsilon\}$. Pro usnadnění pochopení definice jazyka, uvedeme jeden příklad. Mějme abecedu Σ = {a, b}, nad touto abecedou můžeme mít například tyto jazyky:

$$L_1 = \varnothing$$

$$L_2 = \{\varepsilon\}$$

$$L_3 = \{x: |x| = 1\}$$

$$L_4 = \{x: \text{ab je podřetězec x}\}$$

Jak můžeme vidět je mnoho způsobů jak vyjádřit jazyk nad danou abecedou. Musíme uvést jedno upozornění, a to že ∅ prázdná množina a {ε} množina s prázdným řetězcem, je jazykem nad každou abecedou, ale tyto jazyky se v žádném případě nerovnají, neboť jazyk obsahující prázdný řetězec,

obsahuje řetězec, i když prázdný, tedy by se mohlo zdát, že vyjadřuje totéž, co jazyk, jenž je vyjádřen prázdnou množinou, avšak ta neobsahuje ani ten prázdný řetězec.

Definice 2. 1. 11 [7]

Jazyk *L* je konečný, pokud *L* obsahuje konečný počet řetězců, jinak je nekonečný.

Pokud budeme vycházet z předchozího příkladu pro vysvětlení jazyků, pak můžeme uvést:

$L_1 = \varnothing$ je ***konečný jazyk***, protože počet řetězců v tomto jazyku je roven 0

$L_2 = \{\varepsilon\}$ je ***konečný jazyk***, protože počet řetězců v tomto jazyku je roven 1

$L_3 = \{x: |x| = 1\} = \{a, b\}$ je ***konečný jazyk***, protože počet řetězců v tomto jazyku je roven 2

L_4 = {x: ab je podřetězec x} = {ab, aab, abb, atd.} je ***nekonečný jazyk***

Jazyk L_4 je nekonečný neboť jeho definice neurčuje přesný počet podřetězců, jenž obsahuje.

Definice 2. 1. 12 [7]

Nechť L_1 a L_2 jsou dva jazyky nad abecedou Σ. Sjednocení jazyků L_1 a L_2, $L_1 \cup L_2$, je definováno: $L_1 \cup L_2 = \{x: x \in L_1 \text{ nebo } x \in L_2\}$

Mějme dva jazyky L_1 ={a, b, ab, ba} a L_2 = {aa, bb ,ab, ba}, z výše uvedené definice potom vyplývá, že nový jazyk vzniklý sjednocením bude obsahovat všechny řetězce z obou jazyků:

$$L_1 \cup L_2 = \{a, b, aa, bb, ab, ba\}.$$

Definice 2. 1. 13 [7]

Nechť L_1 a L_2 jsou dva jazyky nad abecedou Σ. Průnik jazyků L_1 a L_2, $L_1 \cap L_2$, je definováno: $L_1 \cap L_2 = \{x: x \in L_1 \text{ a } x \in L_2\}$

Mějme dva jazyky L_1 ={a, b, ab, ba} a L_2 = {aa, bb ,ab, ba}, z výše uvedené definice potom vyplývá, že nový jazyk vzniklý průnikem bude obsahovat pouze řetězce, které patří do obou jazyků:

$$L_1 \cap L_2 = \{ab, ba\}.$$

Definice 2. 1. 14 [7]

Nechť L_1 a L_2 jsou dva jazyky nad abecedou Σ, Rozdíl jazyků L_1 a L_2, $L_1 - L_2$, je definován:

$L_1 - L_2 = \{x: x \in L_1 \text{ a } x \notin L_2\}$

Mějme dva jazyky L_1 ={a, b, ab, ba} a L_2 = {aa, bb ,ab, ba}, z výše uvedené definice potom vyplývá, že nový jazyk vzniklý rozdílem bude obsahovat pouze řetězce, které patří pouze do jazyka L_1 :

$$L_1 - L_2 = \{a, b\}$$

Definice 2. 1. 15 [7]

Nechť *L* je jazyk nad abecedou Σ. Doplněk jazyka *L*,$\bar{L}$, je definován: $L = \Sigma^* - L$

Mějme abecedu Σ = {a, b} a jazyk nad touto abecedou L = {a, b, ab, ba}, pak pokud si vyjádříme všechny řetězce nad abecedou Σ^*, pak doplněk $\bar{L}$obsahuje všechny řetězce nad abecedou Σ, které nepatří do jazyka L.

Definice 2. 1. 15 [7]

Nechť L_1 a L_2 jsou dva jazyky nad abecedou Σ. Konkatenace jazyků L_1 a L_2, L_1L_2, je definována jako L_1L_2 = {xy: x ∈ L_1 a y ∈ L_2}.

Definice 2. 1. 16 [7]

Nechť L je jazyk nad abecedou Σ. Reverzace jazyka L, reverse (L), je definována: reverse (L) = {reverse (x): x ∈ L}

V podstatě dojde k reverzaci každého jednoho řetězce, jak jsme si již ukázali, v daném jazyce. Mějme jazyk L = {ab, abc}, reverzací tohoto jazyka dostaneme jazyk reverse (L) = {ba, cba}.

Definice 2. 1. 17 [7]

Nechť L je jazyk nad abecedou Σ. Pro i ≥ 0, i-tá mocnina jazyka L, L^i, je definována:

1. $L^0 = \{\varepsilon\}$
2. pro i ≥ 1: $L^i = LL^i$

Mějme jazyk L = {a, bc}, pokud budeme chtít například mocninu jazyka L^2 pak postupně od nulté mocniny začneme vytvářet výslednou mocninu jazyka následovně:

$$L^0 = \{\varepsilon\}$$

$$L^1 = \{a, bc\}$$

$$L^2 = \{aa, abc, bca, bcbc\}$$

Definice 2. 1. 17 [7]

Nechť L je jazyk nad abecedou Σ. Iterace jazyka L, L^* a pozitivní iterace jazyka L,L^+, jsou definovány $L^* = \coprod_{i=0}^{\infty} L^i$, $L^+ = \coprod_{i=1}^{\infty} L^i$

Mějme jazyk L = {a, ab} nad abecedou Σ = {a, b} pak dostaneme jazyky:

$$L^0 = \{\varepsilon\},\ L^1 = \{a, ab\},\ L^2 = \{aa, aab, aba, abab, ...\}$$

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots = \{\varepsilon, a, ab, aa, aab, aba, abab, \ldots.\}$$

$$L^+ = \quad L^1 \cup L^2 \cup \ldots = \{a, ab, aa, aab, aba, abab, \ldots.\}$$

Z výše uvedeného příkladu můžeme odvodit, že iterace jazyka není nic jiného než mocnina jazyka. Iterace jazyka a pozitivní iterace jazyka se odlišují pouze v tom, že pozitivní iterace neobsahuje prázdný řetězec.

2.3 Modely

Pro modelování jazyků lze například využít, týká se pouze konečného jazyka, vyjmenování všech možných řetězců daného jazyka. Tento způsob modelování je však zcela nevhodný pro nekonečné jazyky, což je případ této práce, neboť vyjmenování všech možných řetězců je nemožné. Využívá se proto například regulárních výrazů nebo konečných automatů, jenž patří k základním prostředkům pro vyjádření nekonečného jazyka. Pro tuto práci byl jako model formálního jazyka vybrán

nedeterministický konečný automat, který je probrán níže v kapitole 5 podrobně. Zde si uvedeme pouze stručně modely v podobě regulárních výrazů a gramatik.

2.3.1 Regulární výrazy

Jednoduše řečeno, jsou regulární výrazy takové řetězce, které definují množinu řetězců. Regulární řetězce se od normálních řetězců liší v tom, že obsahují znaky se speciálním významem. Uvedeme si zde pouze některé využívané základní operátory a jejich význam a poté uvedeme příklad, jak vypadá formální jazyk v podobě regulárního výrazu.

Prvním z operátorů, jenž si zde uvedeme je operátor „+“ nebo častěji používaný „|“ tento operátor nám říká, že výsledkem je jedna z alternativ. Dalším ze základních operátorů, který si zde uvedeme, je operátor „.“. Na místě tohoto operátoru se může vyskytnout jakýkoliv jiný znak. Následuje operátor „?“, který představuje volitelnost umístění znaku, za kterým se operátor nachází, do výsledného řetězce znaků. Posledními operátory, které zde uvedeme, jsou operátor „*“ a „+“, druhý uvedený operátor nemá s operátorem pro alternativu nic společného, proto se pro alternativu používá znak „|“. Poslední dva uvedené operátory mají stejný význam pouze s jedním rozdílem, zatímco operátor „*“ vyžaduje výskyt znaku, případně podvýrazu, který mu předchází, ve výsledném řetězci 0 až ∞ krát, operátor „+“, nám říká, že daný znak se bude ve výsledku vyskytovat nejméně jednou.

Nyní si ukážeme na příkladu, jak může vypadat jazyk formalizovaný regulárním výrazem.

Př. (lol) + (rofl)*.?[a-z] | [0-9]

Nyní si projdeme výše uvedený příklad. První část regulárního výrazu nám říká, že výsledný řetězec bude obsahovat minimálně jeden podřetězec „lol“, za ním bude následovat libovolně mnoho nebo žádný podřetězec „rofl“, následován libovolným znakem, který se ale nemusí ve výsledném řetězci znaků objevit a posledním znakem bude, buď malé písmeno abecedy, v hranatých závorkách je uveden rozsah možných znaků, nebo číslo. Uvedený regulární výraz reprezentuje například: lolxb, lollolroflye, lola, lolrofl9, … atd. Jak můžeme vidět, tak množství řetězců, jenž tento regulární výraz představuje je nekonečné, hlavní příčinou nekonečnosti představovaných řetězců jsou operátory „+“ a „*“, což nám umožňuje napsat regulární výraz pro nekonečný formální jazyk.

2.3.2 Formální gramatiky

Jednoduše řečeno se jedná o množinu pravidel, podle kterých se vytvářejí řetězce patřící do formálního jazyka, jehož je gramatika modelem.

Definice 2.3.2.1[7]

Gramatika je čtveřice G = (N, T, P, S), kde:

N je abeceda neterminálů

T je abeceda terminálů

$N \cap T = \varnothing$

$P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ je konečná množina pravidel

$S \in N$ je počáteční nonterminál

V podstatě se postupně aplikují pravidla na vytvářený řetězec formálního jazyka, začíná se neterminálem *S*, dokud jsou přítomny nějaké znaky z abecedy neterminálů a výsledný řetězec tedy obsahuje pouze znaky z abecedy terminálů. Na příkladu si ukážeme, jak vlastně taková gramatika generuje řetězce. Mějme gramatiku:

$N = \{S, A, B\}$

$T = \{a, b, c, d\}$

$P = \{S \rightarrow aAb, A \rightarrow aBcd, A \rightarrow aAa, B \rightarrow ccd\}$

Nyní, když jsme si definovali gramatiku, můžeme přejít ke generování řetězce.

$$\mathbf{S \Rightarrow aAb}$$
$$\mathbf{\Rightarrow aaAab}$$
$$\mathbf{\Rightarrow aaaAaab}$$
$$\mathbf{\Rightarrow aaaaBcdaab}$$
$$\mathbf{\Rightarrow aaaaccdcdaab}$$

V prvním kroku je aplikováno první pravidlo tedy nonterminální symbol *S* je nahrazen řetězcem *aAb*. Jak můžeme vidět, tak v množině pravidel jsou dvě pravidla aplikovatelná na nonterminál *A*, pokud si vybereme druhé pravidlo a nahradíme nonterminální symbol *A* řetězcem *aBcd* ,pak bude aplikovatelné už pouze jenom čtvrté pravidlo, kdy nonterminál *B* je nahrazen řetězcem *ccd*. Pokud ovšem v druhém kroku aplikujeme třetí pravidlo a nahradíme nonterminální symbol *A* řetězcem *aAa*, můžeme opět aplikovat pravidlo dvě a tři. Gramatika může být modelem nekonečného formálního jazyka díky pravidlu číslo tři, díky kterému je množina řetězců definovaných touto gramatikou nekonečná.

3 Strojové učení

V této kapitole si stručně řekneme, co to vlastně je strojové učení a k čemu slouží. Strojové učení vzrůstá na popularitě, neboť v dnešním světě neustále dochází k vysokému nárůstu dat, jenž je těžko zpracovatelný bez automatizovaného postupu. Strojové učení je nejvíce využíváno v problémech statistiky a pro dolování různých dat [1].

Strojové učení je vědní disciplína, která je podoblastí umělé inteligence. Strojové učení se zabývá metodami a algoritmy, které slouží pro zpracování velkého množství dat způsobem prospěšným pro náš problém, úlohu [1].

Strojové učení můžeme rozdělit podle způsobu učení nebo také podle způsobu zpracování. Jednotlivé metody a algoritmy strojového učení se vždy hodí nejvíce na jednu ze základních úloh strojového učení, avšak není vyloučena možnost úpravy algoritmu, metody strojového učení pro náš konkrétní problém.

Rozdělení podle způsobu učení:

- učení s učitelem – tento způsob učení spočívá v tom, že předem známe správný výstup pro vstupní data, to znamená, že existuje nějaká zpětná vazba, jež nám řekne, zda je daný výstup správný, využívá se trénovací sady, na níž se počítačový systém „naučí“ odvozovat správné výsledky obecně, aby byl schopen odvodit výsledek úspěšně na nových datech, toto učení se využívá u úloh klasifikace nebo úlohy regrese
- učení bez učitele – při tomto způsobu učení nevíme, jak má výstup pro daná data vypadat, to znamená, že neexistuje zpětná vazba, která by nám řekla, zda je daný výstup správný, to také odlišuje tento způsob učení od učení s učitelem a učení se zpětnou vazbou, využívá se například pro odhad hustoty ve statistice
- kombinace učení s učitelem a bez učitele – toto učení využívá malé množství vstupních dat se známým výsledkem a velké množství vstupních dat s výsledkem neznámým, používá se to, protože bylo zjištěno, že spojení velkého množství dat s neznámým výsledkem, s daty, pro něž je výsledek znám, způsobí značný narůst přesnosti v učení
- zpětnovazebné učení – tento způsob učení vychází ze zpětnovazebního učení v psychologii, jde o to vytvořit softwarové agenty v nějakém prostředí, většina prostředí strojového učení je tvořena Markovovým rozhodovacím procesem, agenti jsou umístění do tohoto prostředí na různá místa, sledují děje v procesu a učí se na nich, jak tento proces zefektivnit

Podle způsobu zpracování můžeme strojové učeni rozdělit na:

- dávkové – tento typ zpracování vyžaduje, aby hned na začátku zpracování vstupních dat byla všechna vstupní data známa
- inkrementální – tento typ zpracování na rozdíl od dávkového nepotřebuje znát všechna data předem a je schopno upravit model podle nově příchozích, na začátku neznámých dat

Základní druhy úloh:

- Klasifikace – je to problém ve strojovém určení týkající se statistiky, kdy musíme správně klasifikovat data do správných kategorií, nejdříve se využije trénovací sada, jenž má kategorie správně určeny, podle tohoto natrénovaného modelu se pak nadále klasifikují do kategorií nově příchozí data
- Regrese – využívá se k odhadu výsledku, kdy nejdříve se naučíme model mezi závislými a nezávislými proměnnými, kde poté pozorujeme, získáváme odhadnuté výsledky, pokud je některá z proměnných změněna
- Shlukování – tato metoda se využívá pro dolování dat a je to běžná technika pro statistickou analýzu dat, pracuje tak, že vstupní data rozdělí do skupin na základě jejich podobnosti, v podstatě se jedná o vícerozměrnou klasifikaci
- Pořadí – cílem této úlohy je seřadit vstupní data do určitého pořadí, výsledkem je částečné nebo úplné setřídění
- Učení strukturovaných dat – tato metoda slouží k budování výstupní struktury na základě vstupních dat, na rozdíl od předchozích úloh tato se nezabývá konkrétními hodnotami jako takovými, ale k vybudování určitého modelu, využívá se například pro učení syntaktických stromů, zarovnání několika sekvencí proteinů v bioinformatice atd.

Z toho, co jsme již uvedli v této práci, je zřejmé, že v našem případě se bude jednat o učení s učitelem, neboť víme, jak má vypadat požadovaný model, tedy spíše co má námi tvořený model být schopen akceptovat a také, že naše úloha je učení strukturovaných dat, neboť z mnoha vstupních dat budeme budovat jediný model, jenž bude představovat nám neznámý a námi hledaný jazyk v podobě konečného automatu.

4 Odvozování modelů formálních jazyků

Odvozování modelů formálních jazyků je problém, kdy se snažíme z konečné množiny příkladů daného formálního jazyka odvodit správný model např. gramatiku nebo konečný automat. Původně byla tato problematika studována pro možnost vytvořit model přirozeného jazyka, avšak kvůli jeho neustálým změnám je jeho možnost vymodelování téměř nemožná. Pojem gramatika zde byl zaveden v důsledku podobnosti s přirozeným jazykem a jeho gramatikou. Pro odvozování gramatik formálních jazyků jsou zavedeny různé modely učení. Všechny učící modely poskytují protokol učení a kritéria, kdy došlo k úspěšnému vymodelování gramatiky z příkladů.[6]
Mezi tři hlavní modely učení z příkladů patří: [6]

- Goldův model identifikace v limitu [10]
- Dotazovací model [12]
- PAC model učení [11]

Efektivita odvozovacího algoritmu je posuzována podle velikosti řetězce, pomocí kterého je model tvořen, a velikosti neznámého jazyka, kdy v případě konečných automatů je to počet stavů.

Nyní si něco povíme o odvození modelu formálního jazyka do podoby konečného automatu. Existuje mnoho přístupů k identifikaci neznámého formálního jazyka do podoby konečného automatu. Jedním z těchto přístupů je učení z příkladů řetězců neznámého modelovaného jazyka, kdy se používají pouze „pozitivní“ řetězce, řetězce které do modelovaného formálního jazyka patří nebo se využívá i řetězců „negativních“, tedy řetězců které do daného formálního jazyka nepatří. Dalším přístupem pro odvození modelu formálního jazyka je učení s učitelem, učitelem může být nějaký program nebo člověk, který zná cílový jazyk. Algoritmy postavené na tomto přístupu využívají dotazování na shodu a na členství. Dotazy členství fungují tak, že odvozovací algoritmus předloží řetězec učiteli a ten mu sdělí, zda tento řetězec patří do modelovaného jazyka. Dotazy na shodu fungují tak, že odvozovací algoritmus předloží učiteli model a ten mu sdělí, jestli odpovídá modelovanému jazyku. Všechny tyto přístupy se snaží o vymodelování co nejpřesnějšího modelu neznámého formálního jazyka za co nejkratší čas. Přesnost modelu můžeme odvodit od počtu řetězců, jenž do modelovaného formálního jazyka patří a jsou, v případě konečných automatů, přijaty a počtem řetězců, které do modelovaného formálního jazyka nepatří a jsou přijaty navzdory tomu, že do modelovaného formálního jazyka nepatří. Dalším problémem odvozování modelů formálních jazyků je vytvořit co nejmenší model, tedy konečný automat, který bude mít minimální počet stavů, ale bude přesným modelem neznámého formálního jazyka.

Další model jako výsledek odvozování neznámého jazyka jsou bezkontextové gramatiky. Bezkontextové gramatiky podle Chomského hierarchie jsou označovány jako typ 2. Bylo dokázáno, že bezkontextové gramatiky nemohou být pro neznámý formální jazyk identifikovány stejně rychle

jako konečné automaty pouze pomocí dotazů na shodu[6][9]. Bezkontextové gramatiky jsou těžce odvoditelné pouze z příkladových řetězců modelovaného formálního jazyka a nebudeme se jimi zde dále zabývat.

Další gramatikou, kterou se zabývá odvozování gramatik z formálních jazyků, je stochastická gramatika. Tato gramatika je důležitá pro praktické využití odvozování gramatik. Při odvozování stochastické gramatiky z neznámého formálního jazyka jde v podstatě o to, že každému kroku při generování řetězce předchází určitá pravděpodobnost, že se tento krok provede. Stochastické (pravděpodobnostní) automaty jsou pravděpodobnostním protějškem konečných automatů, které jsou známy jako skryté Markovovy modely hojně používané v rozeznávání řeči.[6] Jednoduše lze tedy říci, že skryté Markovovy modely jsou konečné automaty rozšířené o pravděpodobnost přechodů mezi jednotlivými konfiguracemi. Kromě stochastických konečných automatů, skryté Markovovy modely, je zde samozřejmě i stochastická varianta pro bezkontextové gramatiky. Analogicky k přidané pravděpodobnosti na přechody mezi jednotlivými konfiguracemi konečných automatů je pravděpodobnost bezkontextových gramatik aplikována rovněž na pravidla, pomocí kterých je generován řetězec pomocí bezkontextových gramatik.

Kromě konečných automatů a gramatik existuje mnoho dalších reprezentací jazyků, jako například regulární výrazy pro regulární jazyky. Regulární jazyky jsou vysvětleny v kapitole 2.3.1. Dalšími modely formálního jazyka, které nejsou postaveny na gramatice, jsou jednoduché rekurentní neuronové sítě v oblasti strojového učení. U rekurentních neuronových sítí je snaha o napodobení vlastností konečných automatů při rozpoznávání jazyka.[8]

5 Konečné automaty

Vlastnost vyjádřit pomocí nějakého modelu je potřeba u nekonečných jazyků, neboť na rozdíl od jazyků konečných, kde je možnost vyjádřit celý jazyk tím, že je jednoduše vypsán, u nekonečných jazyků tohoto dosáhnout nelze. Jazyk lze vyjádřit různými modely, a to například pomocí gramatik, regulárních výrazů, algebraicky a automatů. V případě této práce je využit jako model nedeterministický konečný automat.

Konečné automaty se obecně skládají z konečné množiny stavů, vstupní abecedy, konečné množiny pravidel, počátečního stavu a množiny koncových stavů, jež jsou podmnožinou množiny všech stavů daného automatu. Konečný automat v podobě modelu nějakého jazyka funguje tak, že se začíná v počátečním stavu a na vstupu tohoto automatu je dán řetězec nějakého jazyka. Vezme se první symbol (zleva) a vybere se pravidlo, ve kterém symbol odpovídá čtenému symbolu a kde výchozí stav se shoduje s počátečním posléze aktuálním stavem a následně se cílový stav pravidla stane stavem aktuálním a pokračuje se stejným způsobem pomocí dalšího znaku z řetězce. Po přečtení celého řetězce se musí aktuální stav rovnat jednomu ze stavů z konečné množiny koncových stavů, pokud přečtu celý řetězec, aniž bych skončil v koncovém stavu, daný řetězec do jazyka modelovaného tímto konečným automatem nepatří, to platí i v případě, kdy se vyskytne taková situace, že jsme nenačetli celý řetězec, avšak následující čtený symbol z čteného řetězce nám neumožní vybrat takové pravidlo, abychom se dostali z aktuálního stavu do stavu následujícího, čili neexistuje pravidlo, jehož symbol se rovná čtenému symbolu řetězce a zároveň se neshoduje aktuální stav s výchozím stavem pravidla. My si zde uvedeme dva druhy konečných automatů, a to nedeterministické konečné automaty a deterministické konečné automaty.

5.1 Nedeterministické konečné automaty

Definice 5.1.1 [7]

Nedeterministický konečný automat A je pětice $A = (Q, \Sigma, \delta, s, F)$, kde:

Q je abeceda stavů

Σ je abeceda vstupních symbolů

$\delta : Q \times \Sigma \rightarrow 2^Q$ je přechodová funkce (2^Q označuje množinu všech podmnožin množiny Q)

s je označením počátečního stavu, $s \in Q$

F je konečná množina koncových stavů, $F \subseteq Q$

Pro jednodušší pochopení nedeterministického konečného automatu si ukážeme následující příklad. Na následujícím obrázku máme vyobrazen nedeterministický konečný automat, jenž se skládá z:

$Q = \{1, 2, 3, 4, 5, 6, 7\}$

$\Sigma = \{a, b\}$

R = {1a → 1, 1b → 1, 1a → 2, 1b → 7, 2b → 3, 3b → 4, 4a → 5, 7a → 6. 6b → 5, 5a → 5, 5b → 5}
A = (Q, Σ, δ, 1, F)
F = {5}

Vstupní abecedou tohoto nedeterministického konečného automatu jsou symboly *a* a *b*, z čehož již na začátku můžeme usoudit, že tento nedeterministický konečný automat bude modelem pouze pro jazyky, jejichž abeceda Σ by obsahovala pouze tyto dva symboly, z nichž by se skládaly všechny jeho řetězce.

Tento nedeterministický konečný automat by mohl sloužit i jako model pro nekonečné jazyky, a to díky smyčkám ve stavech 1 a 5, které by mu umožnily, přijmout libovolný počet symbolů z dané abecedy, neboť obsahují pravidlo, jehož výchozí stav je shodný se stavem cílovým všemi symboly abecedy.

Nedeterminismus konečného automatu značí, že existuje jeden či více stavů, k nimž existují dvě a více pravidel umožňující přejit z aktuálního stavu do různých stavů, ale pomocí stejného symbolu. V nedeterministickém konečném automatu na obrázku to můžeme vidět na stavech 1 a 5. Problém s nedeterminismem spočívá v tom, že pokud přejdeme do jednoho ze stavů, nevíme, zda je to ten správný a daná cesta povede k úspěšnému přijetí řetězce nedeterministickým konečným automatem. Pro úplnost si zde uvedeme definice konfigurace, přechodu a jazyka přijímaného konečným automatem.

Definice 5.1.2[7]

Nechť A = (Q, Σ, δ, s, F) je konečný automat (KA). Konfigurace KA A je řetězec $\chi \in Q\Sigma^*$.

Jednoduše řečeno konfigurace je aktuální stav konečného automatu a zbytek řetězce, který ještě nebyl přijat tímto konečným automatem. Tedy počáteční konfigurace je počáteční stav a celý řetězec a koncová konfigurace je jeden z koncových stavů množiny a prázdný řetězec.

Definice 5.1.3[7]

Nechť *pax* a *qx* jsou dvě konfigurace KA A, kde p, q ∈ Q ∪ {ε} a x ∈ Σ^*. Nechť r = pa → q ∈ δ je pravidlo. Potom A může provést přechod z pax do qx za použití r, zapsáno pax |— qx [r] nebo zjednodušeně pax |— qx.

Jednoduše řečeno přechod je pojem označující celý proces jednoho kroku, tedy změna z jedné konfigurace do druhé za pomoci aplikace pravidla definovaného přechodovou funkcí konečného automatu, které odpovídá aktuální konfiguraci.

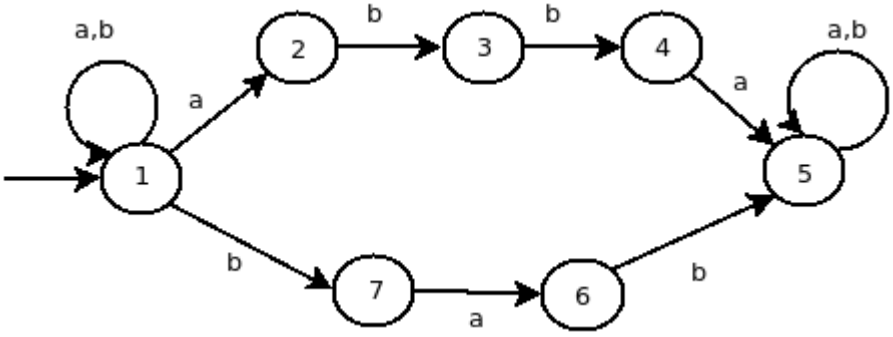
Definice 5.1.4[7]

Nechť A = (Q, Σ, δ, s, F) je KA. Jazyk přijímaný konečným automatem A, L(A), je definován:

L(A) = {w: w ∈ Σ^*,sw |—* *f*, *f* ∈ F}

Jednoduše řečeno přijímaný jazyk konečným automatem je takový jazyk, pro jehož řetězce existuje taková posloupnost pravidel, aby se z počáteční konfigurace po přečtení celého řetězce nacházel v jednom z koncových stavů.

Mějme nyní vstupní řetězec *x =aabababa*, aktuální stav, na začátku je to stav počáteční. Nyní budeme postupně odebírat symboly z řetězce *x* zleva. Hned na začátku se nám projeví problém nedeterministických konečných automatů, kdy z počátečního stavu se můžeme dostat pomocí prvního symbolu zleva do stavu 2 nebo zůstat ve stavu 1. Pro ukázku si ukážeme obě volby. Pokud přejdeme pomocí přečteného symbolu *a* do stavu 2 dojdeme k závěru, že daný řetězec, není přijat tímto nedeterministickým konečným automatem. Pokud se ale rozhodneme „přejít" do stavu 1, je možné nadále pokračovat. Nyní náš čtený řetězec vypadá následovně *x = abababa,* vidíme, že opět je čteným symbolem symbol *a*, což nám opět umožní přejít do stavu 1 nebo 2, avšak na rozdíl od předcházejícího kroku, pokud přejdeme do stavu 2, náš řetězec stále může pokračovat. Nyní si tedy ukážeme, kam povede dál tato cesta. Dalším čteným symbolem se stává symbol *b,* který nám umožní přejít do stavu 3, povšimněte si, že tato cesta je jediná možná, tuto část bychom mohli označit jako deterministickou. Tato cesta nám však příliš nepomohla neboť dalším čteným symbolem je *a*, což zapříčiní výsledek řetězec nepřijat nedeterministickým konečným automatem neboť jediný možný přechod ze stavu 3 je do stavu 4 pomocí symbolu *b.* Nyní se vraťme zpět k druhému čtenému symbolu *a*, ale tentokrát „přejdeme" do stavu 1. Další čtený symbol, *b*, nás dostane do stavu 7, opět je možné „přejít" do stavu 1, ale ukažme si nyní, kam se dostaneme, když půjdeme cestou přes stav 7. Vidíme, že následující dva symboly *a* a *b* přesně odpovídají pravidlům 7a → 6, 6b → 5, tím jsme se dostali do koncového stavu, a můžeme považovat řetězec za přijatý neboť ve stavu 5 je smyčka, jenž přijímá celou vstupní abecedu. Pokud bychom, zkoušeli nadále „horní" cestu automatu, nikdy bychom se nedostali do koncového stavu, protože si můžeme všimnout, že pro tuto cestu jsou třeba dva za sebou jdoucí symboly *b,* jenž náš vstupní řetězec neobsahuje.



Obrázek 5.1.1: Nedeterministický konečný automat

5.2 Deterministické konečné automaty

Definice deterministického konečného automatu se od definice nedeterministického konečného automatu, definice 5. 1. 1, liší pouze v definici přechodové funkce. U deterministického konečného automatu je přechodová funkce definována: $\delta : Q \times \Sigma \rightarrow Q$. Na rozdíl od nedeterministického konečného automatu, zde více pravidel, popsaných přechodovou funkcí, může mít stejný výchozí stav a přijímaný symbol, ale rozdílný cílový stav.

Na příkladu si ukážeme, že vyhodnocení zda řetězec projde daným deterministickým konečným automatem či nikoliv je v případě deterministického konečného automatu daleko jednodušší, neboť determinismus značí, že z každého stavu automatu, existuje pouze jediné pravidlo s daným symbolem. Pro snadnější pochopení rozdílu použijeme, konečný automat s odstraněným nedeterminismem, jenž byl způsoben přechody 1a → 1 a 1b → 1 u předchozího příkladu.

Q = {1, 2, 3, 4, 5, 6, 7}

Σ = {a, b}

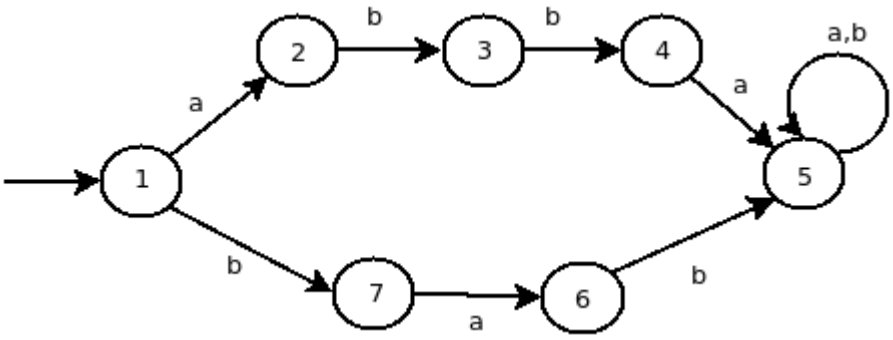
R = {1a → 2, 1b → 7, 2b → 3, 3b → 4, 4a → 5, 7a → 6. 6b → 5, 5a → 5, 5b → 5}

A = (Q, Σ, δ, 1, F)

F = {5}

Mějme nyní stejný řetězec jako v předcházejícím příkladu *x* =*aababababa*, již po prvním přečteném symbolu *a*, se dostaneme do stavu 2, odkud se nijak nemůžeme dostat, a řetězec nebyl přijat daným deterministickým konečným automatem. Nemusí se zde vyhodnocovat žádná další cesta, neboť determinismus nám jasně určuje, že z každého stavu vede jen jeden pravidlo s daným symbolem. Vezměme si tedy další řetězec *y* = *abbaababababa*, na tomto řetězci si můžeme ukázat další výhodu determinismu a to, že pokud se pozorně podíváme na daný deterministický konečný automat, tak je zjevné, že do koncového stavu se dostaneme, pokud náš řetězec obsahuje prefix *abba* nebo *bab*, další symboly nejsou potřeba kontrolovat, neboť koncový stav 5 obsahuje smyčku, která mu dovoluje přijmout všechny symboly vstupní abecedy. Díky tomuto faktu nám stačí zkontrolovat maximálně první čtyři symboly řetězce. Řetězec *y* obsahuje prefix *abba,* což nám umožní přejít do koncového stavu „horní“ cestou a tudíž je řetězec *y* přijat tímto deterministickým konečným automatem.

Pokud by stav 5 neobsahoval smyčku, tento deterministický konečný automat by přijímal pouze konečný jazyk s řetězci *abba* a *aba*, neboť proto, aby byl daný řetězec přijat konečným automatem, je třeba, aby se do po přečtení celého řetězce ocitl v koncovém stavu, to znamená, že řetězec nesmí být ani kratší ani delší. Z tohoto můžeme usoudit, že pokud máme konečný automat s konečnou množinou stavů a chceme s ním modelovat nekonečný jazyk, je třeba, aby obsahoval cyklus, to může vést v mnoha případech k nedeterminismu, proto byl pro tuto práci vybrán nedeterministický konečný automat jako model jazyka.



Obrázek 5.1.2: Deterministický konečný automat

6 Genetické algoritmy, GA

Pro tuto práci byla vybrána metoda strojového učení genetické algoritmy, jenž je součástí většího celku algoritmů nazvaných evoluční. Všechny tyto algoritmy patřící do evolučních algoritmů jsou inspirovány evolucí v přírodě (biologická evoluce). Tyto algoritmy zahrnují stejný základní princip jako je vytvoření počáteční populace. V této populaci dochází k přirozenému výběru, kdy nejsilnější jedinci mají větší šanci vstoupit do procesu reprodukce. Dalším faktorem je genetický drift, ten vyjadřuje náhodu při evoluci, což spočívá v náhodných úmrtích jedinců v populaci, ať už silných či slabých, mutace jedinců, kdy dochází k pozměnění náhodných částí genetického materiálu jedince, tyto náhodné procesy ovlivňují hlavně malé populace. Dalším aspektem evolučních algoritmů je reprodukční proces, kdy z aktuální populace je tvořena populace potomků a to pomocí křížení dvou jedinců z aktuální populace, které nadále označujeme jako rodiče, jenž vytvoří jednoho nového jedince v populaci potomků, y každého rodiče se vezme část jeho genetické informace.

Ještě než se pustíme do vysvětlování genetických algoritmů, ukážeme si etapy, které je třeba řešit při návrhu konkrétního genetického algoritmu [2][4]:

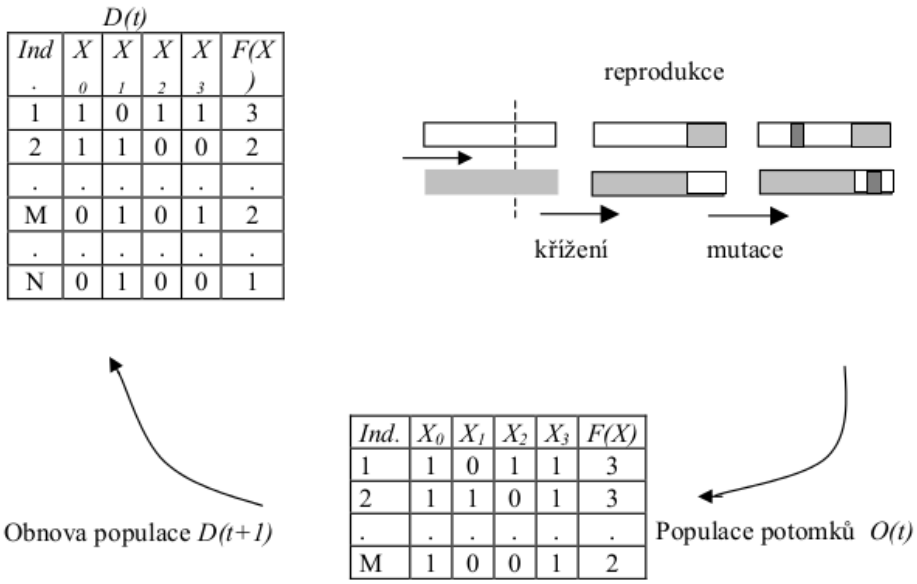
- Reprezentace problému
- Počáteční populace
- Evaluace individuí
- Operátory selekce
- Operátory rekombinace
- Operátory mutace
- Obnova populace
- Velikost populace
- Ukončení algoritmu

Nebudeme zde rozebírat genetický algoritmus podle těchto etap, ale následující text je všechny zahrnuje.

Pro představu jak vypadají genetické algoritmy, si zde uvedeme základní obecný algoritmus, jenž je stejný pro všechny genetické algoritmy a postupně si jej vysvětlíme.

Činnost standardního GA lze popsat následujícím pseudokódem[2][4]:

1. Nastav t=0, náhodně generuj počáteční populaci D(0) s mohutností N,
2. Proveď ohodnocení jedinců populace D(t) fitness funkcí F (X),
3. Generuj populaci potomků O (t) s mohutností $M \leq N$ použitím operátorů křížení a mutace,
4. Vytvoř novou populaci D(t+1) nahrazením části populace D(t) jedinci z O (t),
5. Nastav $t \leftarrow t+1$,
6. Pokud, není splněna podmínka pro ukončení algoritmu, jdi na 2.



Obrázek 6.1: Vývojový diagram genetického algoritmu[4]

Počáteční populace je generována většinou zcela náhodně, avšak může být generována na základě nějaké heuristiky. Je důležité vhodně zvolit zakódování počáteční populace, všechny následující populace jsou zakódovány stejně, pro úspěšné řešení problému, typické je pro genetické algoritmy zakódování jedinců pomocí binárních vektorů viz tab. 6.1.

Individuum č.	Chromozom
1	(1, 0, 1, 0, 1, 1, 0, 0)
2	(0, 1, 1, 1, 1, 0, 1, 1)
3	(0, 0, 0, 1, 0, 0, 0, 1)
4	(1, 1, 0, 0, 1, 1, 0, 0)

Tabulka 6.1: Jedinec jako binární vektor [2]

V dalším kroku dochází k ohodnocení, to se provádí pomocí tzv. Fitness funkce, zde genetické algoritmy nechávají zcela volnou ruku osobě, jenž daný genetický algoritmus realizuje.

Individuum č.	Chromozom	Ohodnocení
1	(1, 0, 1, 0, 1, 1, 0, 0)	4
2	(0, 1, 1, 1, 1, 0, 1, 1)	6
3	(0, 0, 0, 1, 0, 0, 0, 1)	2
4	(1, 1, 0, 0, 1, 1, 0, 0)	4

Tabulka 6.2: Ohodnocení jedinců [2]

Zbývající kroky obecného algoritmu budou vysvětleny níže již na konkrétních operátorech genetického algoritmu.

6.1 Selekce

Operátor selekce nám umožňuje vytvářet novou populaci. Vybírá nové jedince ze staré populace pro populaci novou. Selekční operátor musí upřednostňovat jedince s vyšším ohodnocením, avšak musí zde být brán zřetel, také na zachování různorodosti. Pokud bychom stále vybírali do reprodukčního procesu pouze jedince s nejvyšším ohodnocením, docházelo by k předčasné konvergenci, což znamená, že bychom nalezli pouze řešení s lokálním extrémem. V případě této práce by došlo k vytvoření nedeterministického konečného automatu, jenž by byl modelem pouze části jazyka. Rychlost konvergence lze pozorovat pomocí tzv. selekčního tlaku, čím je tento tlak vyšší, tím rychleji dochází ke konvergenci

Nejčastěji používané algoritmy, které poskytují použitelné výsledky, jsou následující[4]:

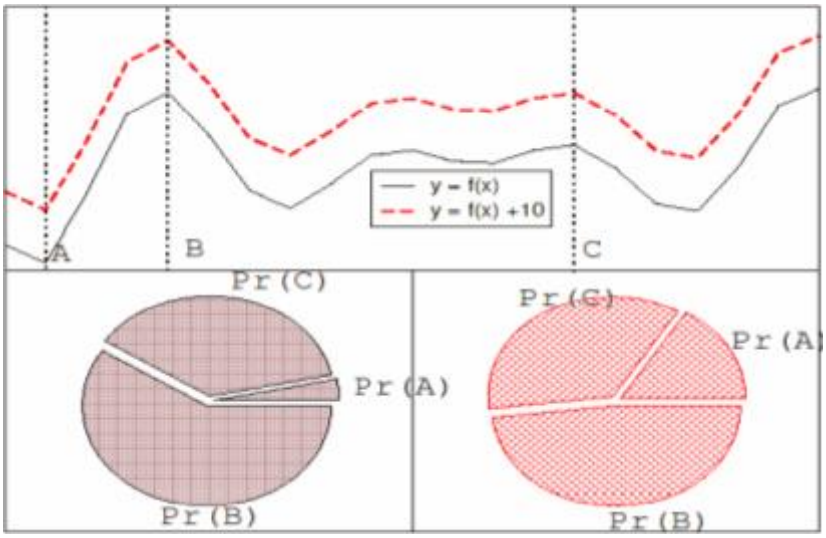
- Proporcionální selekce (roulette wheel selection),
- Selekce zbytková (truncation),
- Lineární uspořádání (ranking),
- Exponenciální uspořádání (ranking),
- Turnajová selekce

Selekce pomocí rulety je založena na náhodném výběru, ale každý jedinec dostane z koláče rulety kus velký přímo úměrný velikosti jeho ohodnocení, zde spočívá riziko v tom, že pokud nějaký jedinec bude mít hodnotu ohodnocení nesrovnatelně vyšší než ostatní, bude postupně celá populace nahrazena tímto jediným jedincem, proto se využívá různých technik, které toto riziko minimalizují.

Nejčastěji se používají dvě techniky[4]:

1. Komprimace fitness funkce (windowing):
   $f'(i) = f(i) + \beta^t$, kde β je nejhorší fitness v aktuální t-generaci
2. Sigma škálování:
   $f'(i) = \max(f(i) - (\langle f \rangle - c \bullet \sigma_f), 0.0)$, kde c je konstanta, obvykle 2.0 a $\langle f \rangle$ je střední hodnota fitness funkce.

Tyto techniky mají za úkol zmenšit rozdíl mezi nejlépe ohodnoceným a nejhůře ohodnoceným jedincem, aby byla zachována různorodost a předešlo se problému popsanému výše.



Obrázek 6.1.1: Princip škálování aditivní konstantou[4]

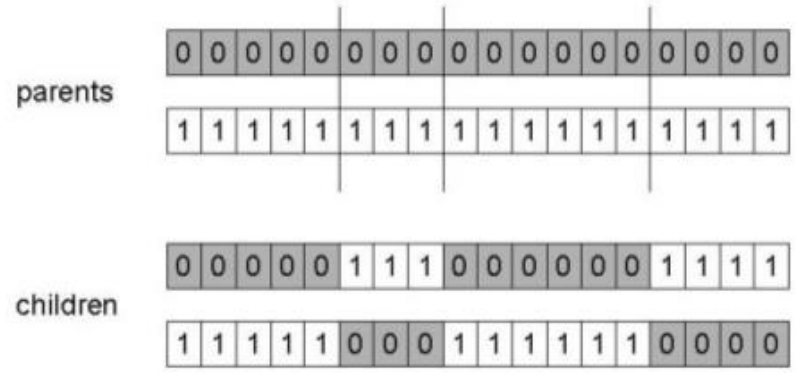
Lineární a exponenciální uspořádání spočívá v tom, že jsou jedinci seřazeni, podle lineární nebo exponenciální pravděpodobnosti výběru, kdy nejhůře ohodnocený jedinec dostává index 1 a nejlépe ohodnocený jedinec index N.

Turnajová selekce je z výše uvedených metod nejčastěji používána. Je založena na náhodě a také nevyžaduje, aby byla populace uspořádána jako u lineárního a exponenciálního uspořádání. Tato metoda selekce spočívá v tom, že se vybere náhodně určitý počet jedinců, ze kterých je následně vybrán ten nejlépe ohodnocený jedinec. Tento proces se opakuje, dokud nedosáhne nově vznikající populace požadované velikosti. Jak můžeme vidět tak z naznačeného postupu vyplývá, že mohou být vybráni i slabší jedinci pokud se do turnajového výběru dostane více slabých jedinců.

6.2 Křížení

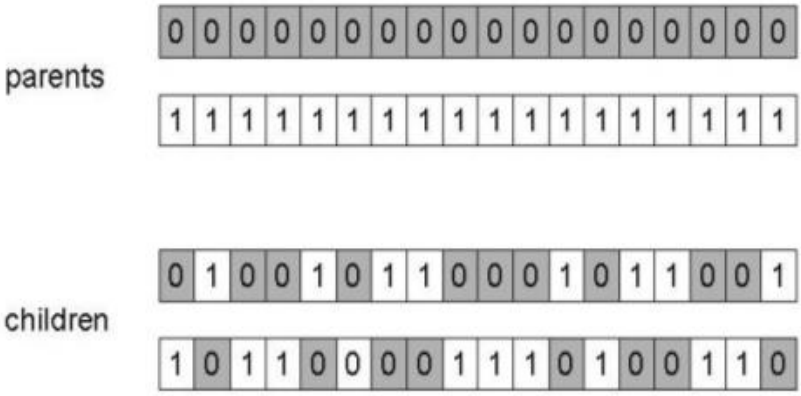
Základem operátoru křížení je náhodný výběr dvou jedinců, kteří se stanou rodiči jedince nového neboli potomka. Tato operace nemusí být využita na celou populaci, to znamená, že někteří jedinci se mohou dostat do nové populace nezměnění. Využívá se jednobodové a vícebodové křížení nebo křížení uniformní, tyto metody křížení jsou nejčastěji používané.

Jednobodové a vícebodové křížení funguje tak, že se označí jedno místo, v případě vícebodového křížení více míst, u obou jedinců, kteří byli vybráni jako rodiče. Označené místo/místa, tvoří zarážky, kdy nový jedinec je postupně budován tak, že se vybere část z jednoho rodiče až po zarážku, poté se bere část z rodiče druhého, pokud jde o vícebodové křížení, dochází k tomuto jevu vícekrát. Rodiče se při tvorbě nového jedince střídají, dokud jejich potomek nedosáhne dané velikosti.



Obrázek 6.2.1: Vícebodové křížení[4]

Uniformní křížení je více náhodné, neboť se postupně prochází chromozómy rodičovských jedinců a při tvorbě potomka dochází ke střídání rodičů v předávání genetické informace potomkovi na základě určité pravděpodobnosti.



Obrázek 6.2.2: Uniformní křížení[4]

6.3 Mutace

Operátor mutace není využíván v genetických algoritmech s velkou pravděpodobností, ale tento operátor přináší do genetického algoritmu nové informace, proto je důležité na tento operátor zcela nezanevřít a poskytnout mu dostatečnou pravděpodobnost. Pravděpodobnost nesmí být moc velká, neboť by docházelo k nestabilitě vývoje dané populace a naopak, kdyby byla pravděpodobnost mutace příliš malá, nebylo by získáno dostatečné množství nových informací. Máme různé metody mutace v genetických algoritmech jako například inverze jednotlivých bitů jedince.

6.4 Obnova populace

Obnova populace probíhá dvojím způsobem, buď dojde k úplnému nahrazení původní populace populací novou, nebo dojde k částečnému nahrazení populace, což spočívá v tom, že část staré

populace zůstává a část staré populace je nahrazena jedinci z nově vytvořené populace potomků. Pro částečnou obnovu populace se využívá různých technik[4]:

- Podle kvality
- Turnaj
- Elitismus
- Faktor přemnožení, při této technice se náhodně vybere podmnožina z původní populace a je nahrazena jediným potomkem, jenž má podobný genotyp

U všech těchto technik je vyžadováno zachování velikosti populace.

7 Návrh

V této práci je snaha o vytvoření aplikace, která bude využívat některou z metod strojového učení pro vytvoření modelu formálního jazyka na základě řetězců, jež do modelovaného formálního jazyka patří a pomocí řetězců, které do modelovaného formálního jazyka nepatří.

V této kapitole si projdeme návrh všech částí programové části této práce. Při návrhu bylo vycházeno z algoritmu metody strojového učení uvedené v kapitole 6. Při návrhu byla snaha o co nejvěrnější zachování klasických postupů používaných v genetických algoritmech. Postupně bude v této kapitole probrán celý algoritmus již z pohledu na programovou část této práce.

7.1 Způsob kódování

Jeden z nejdůležitějších aspektů pro úspěch genetického algoritmu je zvolení vhodného způsobu zakódování jedinců. Klasicky používaný způsob kódování jedinců při genetických algoritmech, binární vektor, je pro tuto práci zcela nepraktický ne-li úplně nevhodný. Pro modelování hledaného formálního jazyka byl vybrán nedeterministický konečný automat a podle tohoto výběru bude způsob zakódování jedinců realizován jako pole obsahující všechna pravidla přechodové funkce, představující jeden celý konkrétní nedeterministický konečný automat, jenž představuje jedince populace. Samotné pravidlo bude realizováno v podobě struktury obsahující výchozí stav, akceptovaný znak tímto pravidlem a stav cílový.

7.2 Generování počáteční populace

Při návrhu generování počáteční populace bylo vycházeno z možnosti využití heuristiky při její tvorbě. Konkrétně tato práce bude využívat trénovací sadu, která bude v podobě řetězců náležících do nám neznámého jazyka, který chceme modelovat pomocí této aplikace do podoby nedeterministického konečného automatu, pro vytvoření počáteční populace a to tak, že pro každý řetězec z trénovací sady bude vytvořen jeho vlastní nedeterministický konečný automat. To znamená, že každý nedeterministický konečný automat tvořící jedince počáteční populace bude modelem minimálně jednoho řetězce z trénovací sady. Tento postup se však později při testování ukázal jako nevhodný, neboť po průchodu jedince celým procesem implementovaného genetického algoritmu se jeho struktura zcela rozpadla. Proto bylo od tohoto způsobu generování počáteční populace upuštěno. Nedeterministické konečné automaty tvořící jedince počáteční populace se generují náhodně, dokud jejich ohodnocení není větší než nula, způsob ohodnocování viz níže, tzn., že zde zůstala původně zamýšlená heuristika avšak ve zcela odlehčené formě.

7.3 Fitness funkce

Fitness funkce byla původně navržena tak, že ohodnocení generované fitness funkcí bude rovno počtu řetězců trénovací sady, které budou ohodnocovaným nedeterministickým konečným automatem, jenž představuje jedince populace, přijímány. Během testování programové části této práce bylo zjištěno, že pokud by byli jedinci ohodnocováni pouze v případě přijetí celého řetězce z trénovací sady, aplikace by strávila dlouhou dobu na stejné populaci beze změn, proto byl způsob ohodnocování fitness funkcí jedinců populace pozměněn na hodnotu rovnu počtu znaků, které byly přijaty daným nedeterministickým konečným automatem představujícího jedince, i když daný řetězec z trénovací sady nedeterministickým konečným automatem nebyl přijat celý.

Výše zmíněné vlastnosti fitness funkce jsou prováděny samostatně pouze při generování počáteční populace, kde se využívá pouze trénovací sady řetězců, které patří do modelovaného neznámého jazyka. Fitness funkce dále využívá při procesu genetického algoritmu i trénovací sadu řetězců, která do modelovaného neznámého jazyka nepatří. Tyto „záporné“ řetězce na rozdíl od „kladných“, ovlivňují ohodnocení jedince populace pouze v případě, že jedinec v podobě nedeterministického konečného automatu tento „záporný“ řetězec přijal celý. Tedy pokud je „záporný“ řetězec přijat celý odečte se od ohodnocení jedince plná délka řetězce, který do daného modelovaného neznámého jazyka nepatří. Tyto řetězce byly zařazeny do fitness funkce pro potřebu vytvořit přesnější model neznámého jazyka.

7.4 Křížení

V této práci mohou být využity všechny metody křížení uvedené v kapitole 6. Jak bylo uvedeno dříve, jedinec bude reprezentován jako pole pravidel přechodové funkce představujících nedeterministický konečný automat.

Při takto zvolené reprezentaci mohou být využity jednotlivá pravidla v poli jako geny, jenž budou při operaci křížení vyměňovány. V programové části této práce bude využito uniformního křížení. Křížení nebude probíhat vždy ve stejných místech jedinců populace, ale bude zde zahrnuta pravděpodobnost, že ke křížení, výměně genu jedince, vůbec dojde, viz kapitola 6.2. Pro co nejvěrnější dodržení pravidel genetických algoritmů zde bude také zahrnuta pravděpodobnost provedení křížení dvou jedinců, avšak křížení jako základní kámen genetických algoritmů bude mít vysokou pravděpodobnost aktivace. Hodnota pravděpodobnosti křížení byla zvolena podle obecného genetického algoritmu na 95%.

Způsob výběru jedinců pro křížení, pro další generaci nedojde-li ke křížení, byla vybrána metoda turnajové selekce pro obecně dosahování nejlepších výsledků.

7.5 Mutace

Při mutaci se obecně mění jednotlivé geny, v klasickém genetickém algoritmu jde o inverzi bitů v bitovém vektoru představujícího jedince, v této práci je jedinec představován polem pravidel přechodové funkce, pokud tedy chceme aplikovat stejný postup, budeme celé jedno pravidlo přechodové funkce brát jako jeden gen jedince. Pravidlo se stává ze tří částí, jak bylo řečeno již dříve, tedy při mutaci bude docházet ke změně celého pravidla s určitou pravděpodobností.

Navzdory všeobecně nízkému využívání mutace v genetických algoritmech, její pravděpodobnost v klasickém genetickém algoritmu se pohybuje v deseti tisícinách procent, bude pravděpodobnost mutace v programové části této práce zvýšena o několik řádů. Tento rozdíl oproti klasickým genetickým algoritmům byl zaveden z důvodu stagnace aplikace, kdy ohodnocení jedinců populace se nemění a je třeba do oběhu dostat nové genetické informace, které by mohly daný genetický proces uvést zpět do pohybu.

Kromě znatelně vyšší hodnoty mutace oproti klasickým genetickým algoritmům bude také zavedeno zvyšování pravděpodobnosti mutace o konstantní hodnotu při stagnaci. Zvyšování pravděpodobnosti mutace zvyšuje šanci aplikace na opětovné rozběhnutí.

7.6 Obnova populace a ukončení cyklu

V implementovaném genetickém algoritmu bude docházet k obnově populace částečně a bude využita technika, podle kvality tzn., že se budou vybírat nejlépe ohodnocení jedinci z populace potomků i z populace rodičů a z nich vznikne nová populace, se kterou se bude nadále pracovat v dalším cyklu implementovaného genetického algoritmu.

Konec genetického algoritmu se odvíjí buď od předem stanoveného času, nebo od požadované hodnoty ohodnocení. Výsledná aplikace této práce využívá obě varianty. Uživatel si bude moci zvolit požadovanou hodnotu a zároveň si může stanovit čas, kdy dojde k ukončení aplikace, tedy tato aplikace bude provádět proces genetického algoritmu tak dlouho, dokud se v populaci nevyskytne nedeterministický konečný automat, jenž je považován, z hlediska trénovací sady řetězců patřících do modelovaného jazyka, za model neznámého jazyka nebo dokud nevyprší uživatelem stanovený čas.

Možnost stagnace vytvářené aplikace, může dojít i k tomu, že i pravděpodobnost mutace zvýšená na maximum nedokáže aplikaci uvést opět do pohybu, proto bude v aplikaci vložena „pojistka“, která při příliš dlouhé stagnaci, stagnace se bude odvíjet od počtu generací, při nichž nedošlo k žádným změnám, dojde k restartu genetického algoritmu. Dojde tedy k vytvoření nové počáteční populace a proces odvozování modelu pomocí genetického algoritmu začíná znova od nuly. Tato úprava byla navržena později na základě pozorování dosahovaných výsledků při testování aplikace při jejím vývoji. Z pohledu genetických algoritmů se jedná o genetický drift.

V závěru bude docházet k ohodnocení výsledného nedeterministického konečného automatu, k čemuž využijeme další sadu „pozitivních“ i „negativních“ řetězců. Na rozdíl od ohodnocení během procesu modelování genetickým algoritmem bude toto závěrečné ohodnocení vyjádřeno počtem přijatých řetězců, jak „pozitivních“ tak i „negativních“ nedeterministickým konečným automatem.

7.7 Generátor pro testování

Kromě samotné aplikace bude také nutné, kvůli vysoké spotřebě dat, na nichž mohou být prováděny testy a měření, vytvořit funkci pro generování hledaného neznámého jazyka. To bude provedeno vytvořením nedeterministického konečného automatu, tedy budeme mít model jazyka v podobě nedeterministického konečného automatu, pomocí kterého se vygenerují potřebné řetězce. Tento postup je nutný, neboť při náhodném generování řetězců bychom mohli dostat tak odlišné řetězce, že by aplikace mohla najít více vhodných modelů, ale nikdy by nenašla model neznámého formálního jazyka takový, aby došlo k úspěšnému ukončení aplikace.

8 Implementace

Pro implementaci programové části této práce byl zvolen programovací jazyk C. Tento programovací jazyk, byl zvolen, protože s ním má autor nejvíce zkušeností z programovacích jazyků, se kterými se doposud setkal.

V této kapitole budeme popisovat samotnou implementaci programové části této práce vycházející z návrhu uvedeného v předchozí kapitole. Aplikace je rozdělena do celkem 6 souborů, `main.c`, kde je řízení celé implementované aplikace, `pomocné_fce.c`, zde jsou umístěny pomocné funkce jako například `extrahovaniAbecedy`, `generator.c`, v tomto souboru jsou funkce související s generováním počáteční populace a dat určených pro testování aplikace, `procesy_gen_al.c`, zde jsou umístěny všechny funkce, které představují jednotlivé kroky implementovaného upraveného genetického algoritmu, `spol.h`, společná knihovna pro všechny zdrojové soubory a poslední soubor je soubor `konfig`, tento soubor slouží pro konfiguraci aplikace, avšak není povinný viz dále.

8.1 Main.c

V tomto zdrojovém souboru je pouze hlavní funkce `main`, která se stará o veškeré řízení běhu aplikace. Jako první je ve funkci `main` volána funkce `kontrolaArgumentu`, protože následně probíhá alokace paměti pro veškeré dynamické proměnné.

Po úspěšné alokaci začíná hlavní proces programu, který je vložen do `for` cyklu, který určuje, kolikáté měření probíhá podle zvoleného parametru počtu měření -m. Nejdříve se kontroluje, zda byly zadány všechny potřebné soubory s řetězci, pokud ne, je vygenerován neznámý formální jazyk v podobě nedeterministického konečného automatu pomocí funkce `genAutomat`, s nastaveným příznakem VYCHOZIMODEL, který funkci říká jakým způsobem má nedeterministický konečný automat generovat, následně jsou pomocí takto vygenerovaného nedeterministického konečného automatu vytvořeny testovací a modelovací řetězce pomocí funkci `gen_ret` a `genZret`, potřebné pro běh aplikace. Pokud jsou soubory zadány, jsou tyto sady řetězců postupně načteny ze souborů. Ve spojení s zadáváním vlastních souborů je spojena i možnost, že uživatel nemusí zadat počet znaků modelovaného jazyka nastavením parametru –z při spuštění aplikace na hodnotu -1. Aplikace následně provede extrakci všech znaků z řetězců bez kontroly se zadaným parametrem. K extrakci abecedy slouží funkce `extrahovaniAbecedy`. Tato funkce je volána i v případě známého počtu znaků kvůli následné kontrole výskytu požadovaného počtu znaků v řetězcích. Následně pokud nejsou soubory s řetězci zadány a došlo k jejich vygenerování vlastní aplikací, jsou tyto řetězce uloženy do souborů pomocí funkce `tiskRetS`, kdy jednotlivé názvy jsou odvozeny podle toho, zda

se jedná o „záporné“ nebo „kladné“ řetězce, podle čísla měření a počtu znaků. Následně je otevřen soubor určen pro výstup aplikace, pokud není jeho název zadán, použije se implicitní název „výstup“.

Následuje implementace samotného procesu genetického algoritmu. Tato část je složena ze dvou do sebe vnořených `while` cyklů, kdy první `while` cyklus slouží jako zarážka k resetování procesu modelování nedeterministického konečného automatu, takže se zde nastavují všechny proměnné do počátečního stavu, u kterých je to potřeba, a vygenerování nové počáteční populace pro nový start genetického algoritmu. Počáteční populace je generována pomocí funkcí `genAutomat`, který generuje jednotlivé jedince v podobě nedeterministického konečného automatu a funkce `ohodnoceniAutomatu`. Proces generování počáteční populace je uzavřen do `while` cyklu, který probíhá, dokud všichni jedinci populace nemají hodnocení větší než nula.

Po úspěšném vygenerování počáteční populace jsou všechny zbylé kroky implementovaného genetického algoritmu uzavřeny do `while` cyklu s podmínkou rovnosti ohodnocení některého jedince populace na 100%. V cyklu `while` je jako první konstrukce `if-else`, která měří a kontroluje míru stagnace aplikace, tedy při shodě staré maximální hodnoty, maximální hodnota z proměnné obsahující ohodnocení populace, je počítadlo stagnace inkrementováno a po dosažení určité hodnoty, udávaná konfiguračním souborem, parametry aplikace při spuštění nebo defaultní hodnotou, je zkontrolováno, zda již není dosaženo maximálního počtu povolených koncových stavů, pokud ano dochází k restartu procesu genetického algoritmu, jinak je přidán počet koncových stavů, jenž může být a je zvýšena pravděpodobnost mutace o konstantní hodnotu, která je rovněž nastavena jedním ze tří způsobů jako počet povoleného počtu generací při stagnaci.

Dalším krokem je selekce a následné křížení. Tento krok je uzavřen ve `for` cyklu, který je proveden tolikrát, kolik je počet poloviny populace. Vždy jsou totiž vybráni dva rodiče a z nich jsou vytvořeni dva potomci. Výběr rodičů probíhá pomocí funkce `turnajVyber`. Po výběru rodičovských jedinců je s pravděpodobností, vyjádřenou pomocí funkce `genCisla`, 95% provedeno křížení pomocí funkce `procesKrizeni`, pokud ke křížení nedochází, jsou vybraní rodičovští jedinci zkopírování do populace potomků. Po procesu křížení následuje proces mutace potomků pomocí funkce `procesMutace`.

V další části je prováděno přidávání či změna koncových stavů. Nejdříve se na základě statistiky vedené ve funkci `ohodnoceniAutomatu`, statistika zde funguje jako míra pravděpodobnosti, provede přidávání koncových stavů a jejich spočtení. Tento proces je uzavřen ve `for` cyklu. Následuje cyklus `while`, který má za úkol zredukovat počet koncových stavů na počet povolených koncových stavů, povolený počet koncových stavů je v části zabývající se stagnací aplikace. Cyklus vždy najde koncový stav s nejmenší hodnotou statistiky je odebrán z množiny koncových stavů. Tento cyklus běží, dokud se počet koncových stavů neshoduje s povoleným počtem koncových stavů.

Dalším krokem je ohodnocení populace potomků. Ohodnocení probíhá pomocí funkce `ohodnoceniAutomatu` a jak „zápornými“, tak „kladnými“ řetězci. Tento proces je uzavřen ve `for` cyklu, ohodnocení nedeterministického konečného automatu vždy probíhá pouze nad jedním nedeterministickým konečným automatem.

Následuje částečná obnova populace, která je uzavřena v jednom `for` cyklu, který se stará o správný počet jedinců v nové populaci. Nejdříve se pomocí `for` cyklu, uvnitř předchozího zmíněného, najde jedince nejvyšším ohodnocením, jak z původní populace, tak z populace potomků, a následně jej uloží do populace nové.

Posledním krokem je zjištění nejvyššího ohodnocení jedince, kdy pokud některý z jedinců dosáhl požadovaného ohodnocení je proces modelování ukončen.

Aplikace měří čas běhu. Měření začíná před prvním cyklem `while`, který slouží pro resetování procesu vytváření nedeterministického konečného automatu a mezičasy bere vždy po proběhnutí procesu jednou generací.

Pokud vše proběhlo v pořádku, je provedeno testovací ohodnocení pomocí funkce `ohodnoceniAutomatu`, které nám řekne kolik „kladných“ a kolik „záporných“ řetězců, prošlo výsledným nedeterministickým konečným automatem, modelem neznámého formálního jazyka. Následuje uvolňování alokované paměti a uzavření výstupního souboru.

8.2 Pomocné_fce.c

Funkce `kontrolaArgumentu` jako první kontroluje výskyt konfiguračního souboru `konfig`. Pokud není tento soubor nalezen nebo se jej nepodařilo otevřít, přechází se ke kontrole zadaných parametrů aplikace při spuštění. Aplikace využívá pro své nastavení celkem 16 parametrů, kdy pouze 2 z nich jsou povinné. Zadatelné parametry aplikace jsou, počet znaků abecedy, počet požadovaných měření, velikost populace, počet řetězců, délka nejdelšího řetězce, název výstupního souboru, názvy jednotlivých souborů s řetězci, počet pravidel automatu, nejvyšší číslo stavu automatu, stagnační konstanta, ta určuje kolik generací je tolerováno při stagnaci, než bude provedena změna v konfiguraci některých aspektu procesu modelování pomocí implementovaného genetického algoritmu, přírůstek mutace. Nezadané parametry jsou nahrazeny defaultními hodnotami. První z nastavitelných parametrů je `-h`, který slouží pro vypsání nápovědy. Tento parametr se kontroluje ještě před kontrolou výskytu konfiguračního souboru, pokud si tedy uživatel přeje vypsat pouze nápovědu, zadá tento parametr jako jediný při spuštění aplikace. Soubor `konfig` bude spolu se způsobem jeho kontroly popsán níže. Zadané parametry aplikace, při absenci konfiguračního souboru, jsou všechny kontrolovány postupně pomocí proměnné `poleParametru`, pole typu `*char`, kde jsou vypsány všechny parametry, proměnné `poleNezadaných`, pole typu `integer`,

které uchovává příznaky o zadaných nebo nezadaných parametrech aplikace při spuštění a proměnné `pouziteArg`, pole, kde jsou uchovány hodnoty parametrů, které jsou typu `integer`, pomocí 2 `for` cyklů. `For` cyklus postupně prochází všechny zadané parametry a druhý `for` cyklus postupně vždy porovná právě čtený parametr se všemi nastavitelnými parametry, které jsou nastaveny v proměnné `poleParametru`. Pro názvy souborů zadaných při spuštění aplikace, nebo uvedené v konfiguračním souboru, je vytvořena proměnná zvlášť pro každý soubor, kdy před uložením názvu je alokováno jisté množství paměti, pokud je tato paměť pro zadaný název souboru nedostačující, dochází k realokaci podle velikosti zadaného názvu souboru. Při nezadání souborů s řetězci aplikace automaticky sama vygeneruje hledaný formální jazyk v podobě nedeterministického konečného automatu a podle něj vygeneruje řetězce potřebné pro vytvoření nedeterministického konečného automatu jako modelu neznámého formálního jazyka. Kromě ukládání zadaných parametrů aplikace se kontroluje, i zda zadaný parametr vůbec pro tuto aplikaci existuje, tedy je zde zabráněno zadávání neznámých parametrů a probíhá kontrola zadání parametru ihned po jeho přepínači. Po kontrole parametrů jsou nahrazeny nezadané parametry defaultními hodnotami z proměnné `defaultniHodnoty`, pole typu `integer`, kde jsou hodnoty všech parametrů aplikace, které jsou typu `integer`. Všechny číselné parametry aplikace jsou nahrazeny kromě parametru určujícího počet znaků abecedy hledaného jazyka a počtu požadovaných měření. Počet znaků abecedy může být zadán i v podobě hodnoty -1, kdy parametry aplikace se soubory s řetězci určenými pro modelování a testování stanou povinnými, neboť následně je třeba provést extrakci abecedy a zjištění počtu znaků v abecedě pomocí funkce `extrahovaniAbecedy`.

Funkce `genCisla` slouží ke generování čísel typu `integer` od 0 po číslo zadáno jako parametr této funkce. Tato funkce je využita v aplikaci často při pravděpodobnostních rozhodováních, kterých se v implementovaném genetickém algoritmu nachází hodně.

Pro zjištění abecedy pro modelování ze zadaných souborů byla vytvořena funkce `extrahovaniAbecedy`, kdy z jedné ze sad řetězců, které používáme pro modelování v této aplikaci je dána do funkce jako argument. Funkce postupně prochází všechny řetězce pomocí cyklu `while` a kopíruje znaky do pole typu `char` pro abecedu, které se tam ještě nenachází. Pracuje se zde s předpokladem, že v použitých sadách řetězců určené pro modelování nedeterministického konečného automatu v této aplikaci se nacházejí všechny znaky abecedy neznámého modelovaného jazyka.

Funkce `tiskRetS` je implementována pro tištění řetězců, které jsou argumentem této funkce, do souboru, určeném argumentem funkce.

8.3 Generator.c

Funkce `genAutomat` je využita jak pro generování počáteční populace, tak pro generování neznámého formálního jazyka do podoby nedeterministického konečného automatu. Tato funkce se skládá ze tří `for` cyklů. První `for` cyklus slouží pro vytvoření základní struktury nedeterministického konečného automatu, tedy vytváří pravidla pro postupné spojení 0 stavu (prvního) s posledním stavem, který je zvolen při spouštění aplikace. Znak pro tato pravidla se vybírá náhodně pomocí funkce `genCislo`. Při spouštění aplikace se tedy nevolí počet stavů, protože počet stavů je vždy o jeden větší. Druhý `for` cyklus je použit pouze pro generování neznámého formálního jazyka, tedy jeho modelu v podobě nedeterministického konečného automatu. Stará se o to aby v nedeterministickém konečném automatu, tedy v jeho pravidlech, byly obsaženy všechny znaky abecedy. Zde není potřeba dodržovat náhodnost genetických algoritmů, proto je zde zajištěno, že vygenerovaný nedeterministický konečný automat bude obsahovat všechny znaky abecedy modelovaného formálního jazyka. Třetí a poslední `for` cyklus je zcela náhodný při výběru počátečního stavu, znaku a cílového stavu do pravidla a slouží pouze pro vygenerování zbytku nedeterministického automatu, kdy se pravidla generují, dokud není dosaženo požadovaného počtu pravidel, který byl zadán při spuštění aplikace.

Funkce `gen_ret` je tvořena jedním `for` a jedním `while` cyklem. `For` cyklus určuje, který řetězec se právě generuje. Cyklus `while` se provádí, dokud není aktuální stav roven cílovému stavu, v případě tohoto způsobu generování je to nejvyšší číslo stavu, a zároveň není dosaženo požadované délky řetězce. Nejdříve se vyberou všechna možná aplikovatelná pravidla, tedy pravidla, jejichž výchozím stavem je stav aktuální, poté se z těchto pravidel náhodně vybere jedno pomocí funkce `genCislo` a z vybraného pravidla je následně zkopírován jeho znak do vytvářeného řetězce a aktuální stav je nahrazen stavem cílovým vybraného pravidla, pokud není nalezeno žádné aplikovatelné pravidlo, je proces generování řetězce resetován, aktuální stav je nastaven na výchozí a číslo určující pozici v řetězci se nastaví na nulu. Také se zde kontroluje podmínka, shodná s podmínkou cyklu `while`. Je to zde pro resetování procesu generování řetězce. Je zde také pojistka proti zaseknutí, kdy po určitém čase vrací funkce neúspěch.

Funkce `genZret` byla implementována jako komplementární k funkce `gen_ret`, tato funkce generuje „záporné“ řetězce pro modelování a testování. Skládá se ze dvou `for` cyklů, první `for` cyklus určuje který řetězec je právě generován, druhý `for` cyklus probíhá, dokud není dosaženo požadované délky řetězce. Nejdříve je vybrán náhodně znak, ze stejné abecedy jakou má modelovaný neznámý formální jazyk, poté se najdou všechny možná pravidla, která odpovídají svým výchozím stavem aktuálnímu stavu a svým znakem právě vygenerovanému znaku řetězce. Poté se náhodně vybere jedno z aplikovatelných pravidel a cílový stav vybraného pravidla se stává stavem aktuálním. Pokud je dosaženo požadované délky řetězce a aktuální stav je stavem koncovým, je tento řetězec

označen jako neúspěšný a následně nahrazen novým řetězcem. Když se nenajdou žádná aplikovatelná pravidla je řetězec označen jako úspěšný a zbytek řetězce se již generuje náhodně bez kontroly procházení nedeterministického konečného automatu, který představuje neznámý modelovaný formální jazyk.

8.4 Procesy_gen_al.c

Funkce `ohodnoceniAutomatu` je název v programové části této práce pro fitness funkci potřebnou pro genetický algoritmus. Kromě ohodnocování tato funkce také vede statistiku o tom, ve kterém stavu bylo ukončeno přijímání řetězce, pokud byl přijat celý. Tato funkce je složena do dvou do sebe vnořených `for` cyklů, kdy první `for` cyklus určuje právě používaný řetězec pro ohodnocování jedince a druhý `for` cyklus slouží pro procházení samotného řetězce znak po znaku. Pro každý znak jsou nalezena všechna aplikovatelná pravidla z aktuální pozice v nedeterministickém konečném automatu, kdy je nutné, aby výchozí stav pravidla byl roven aktuálnímu stavu a znak v pravidlu byl shodný s aktuálně čteným znakem z řetězce, poté, pokud nějaké pravidlo bylo nalezeno, dochází k přiřazení cílového stavu pravidla do aktuálního stavu, pokud je aplikovatelných pravidel nalezeno více vybere se náhodně jedno z nich pomocí funkce `genCisla`. Po aplikování pravidla je vyhodnoceno, zda se nenacházíme na konci řetězce, pokud ano, kontroluje se, zda aktuální stav, je jeden z množiny koncových stavů a v případě že je, je k ohodnocení jedince přičtena délka celého řetězce, jinak je přičtena hodnota o jedničku menši, než je délka řetězce k ohodnocení jedince. Je zde také upravena statistika aktuálního stavu, kde byl řetězec ukončen je zvýšena statistická hodnota, která je použita pro pozdější pravděpodobnost, že se stav stane stavem koncovým. Pokud není nalezeno žádné aplikovatelné pravidlo na aktuální konfiguraci, je ohodnocování aktuálním řetězcem ukončeno a k ohodnocení jedince je přičten počet dosud přijatých znaků nedeterministickým konečným automatem. Toto platí pouze pro ohodnocování „kladnými“ řetězci v procesu modelování. Pro „záporné“ řetězce během modelování platí, že je odečtena délka řetězce od ohodnocení pouze v případě, že „záporný“ řetězec byl celý přijat nedeterministickým konečným automatem představujícího jedince populace. Po ukončení procesu je využita funkce `ohodnoceniAutomatu` ještě pro zhodnocení výsledného modelu kdy se počítá počet „kladných“ a „záporných“ řetězců přijatých výsledným nedeterministickým konečným automatem. Výběr způsobu ohodnocování, testování, modelování, „kladné“, „záporné“, je pomocí příznaku, jenž je parametrem funkce.

Funkce `turnajVyber` představuje v aplikaci selekci genetického algoritmu pro následné křížení. Turnajový výběr, je implementován dvěma `for` cykly, kdy první `for` cyklus pouze určuje, který rodič je právě vybírán, v druhém `for` cyklu je hlavní část turnajového výběru rodiče. Nejdříve jsou vybráni tzv. adepti na rodiče, k tomu slouží proměnná `rodiceAdepti`, pole typu `integer` o

velikosti ¼ populace, kdy je nejdříve náhodně vybrán jedinec pomocí funkce `genNum`, a následně se kontroluje, zda již není mezi adepty na rodiče. V dalším kroku je porovnána hodnota ohodnocení zvoleného jedince s aktuální nejvyšší hodnotou ohodnocení dosud vybraných jedinců jako adeptů na rodiče. Pokud je ohodnocení nově vybraného adepta na rodiče vyšší než dosavadně nejvyšší ohodnocení jsou všichni adepti smazáni a do proměnné `rodiceAdepti` je uložen nově vybraný adept. V proměnné `rodiceAdepti` jsou ve výsledku uloženi jedinci se stejným nejvyšším ohodnocením, které bylo při náhodném výběru jedinců nalezeno. Poté je náhodně z vybraných adeptů zvolen rodič. Tento proces se následně opakuje i pro druhého rodiče. Na konci je kontrola, zda oba rodiče nejsou tentýž jedinec, pokud ano probíhá nový výběr druhého rodiče.

Funkce `procesKrizeni` je implementován pomocí jednoho `for` cyklu, který postupně prochází všemi pravidly nedeterministického konečného automatu, který představuje jedince populace. Protože se jedná o uniformní křížení, dochází po každém kopírování jednoho pravidla k pravděpodobnostní výměně rodičů mezi vytvářenými jedinci. Tedy nedeterministické konečné automaty představující rodiče se prohazují v tom, ze kterého se bude kopírovat pravidlo do jednoho a do druhé právě vytvářeného nedeterministického konečného automatu představujícího potomka.

Funkce `procesMutace` je složena ze dvou do sebe zanořených `for` cyklů, kdy první `for` cyklus určuje právě mutovaný nedeterministický konečný automat. Druhý `for` cyklus postupně prochází všechna pravidla právě mutovaného nedeterministického konečného automatu. Samotná mutace je postavena na konstrukci `if-else`, protože je vyžadováno provedení vždy jen jednoho druhu mutace. Mutace má celkem 7 variant. První tři varianty provádí změnu (mutaci), pouze jedné části pravidla, pravidlo je složeno z výchozího stavu, znaku a cílového stavu, další tři varianty mění vždy dva prvky pravidla a poslední varianta je úplná změna pravidla, tedy změní se všechny části pravidla.

8.5 Spol.h a konfig

Soubor spol.h slouží jako společná knihovna pro všechny zdrojové kódy programové části této práce. Jsou zde deklarace všech funkcí vyskytujících se v této aplikaci. Nadále je zde definována struktura `pravidlo`, která nadále slouží jako datový typ pro všechny nedeterministické konečné automaty v aplikaci. Struktura pravidlo obsahuje tři proměnné. První proměnná `odkud` je typu `integer` a z pohledu pravidla obsahuje výchozí stav představovaného pravidla nedeterministického konečného automatu. Druhá proměnná `kam` je typu `integer` a z pohledu pravidla obsahuje cílový stav představovaného pravidla nedeterministického konečného automatu. Poslední proměnná `znak` je typu `char` a z pohledu pravidla obsahuje znak použit tímto pravidlem k přechodu z výchozího stavu

pravidla do cílového stavu pravidla. Dále jsou zde všechny knihovny využívané touto aplikací a definici maker sloužící pro snadnější orientaci v kódu.

Soubor `konfig` obsahuje všechny parametry, které lze zadat při spuštění aplikace, kromě parametru pro nápovědu -h. Tento soubor nemá nic společného s klasickými konfiguračními soubory, je zde pro ulehčení spouštění aplikace uživateli. Kontrola probíhá stejně, jako u ručně zadaných parametrů při spuštění aplikace tedy dva parametry jsou povinné, ostatní mohou být nahrazeny implicitními hodnotami a v případě parametru -z s hodnotou -1 se parametry s názvy souborů obsahujících řetězce stávají povinnými. Soubor konfig je určen pouze pro modifikaci za znakem „:“, změna jinde v konfiguračním souboru je kontrolována.

9 Testování

Testování bylo prováděno pomocí implementovaného generátoru (viz předchozí kapitola), který pro každé měření generoval nový formální jazyk v podobě nedeterministického konečného automatu, pomocí kterého byly následně generovány potřebné řetězce pro testování.

Následující testy byly prováděny s těmito parametry:

Počet stavů nedeterministického konečného automatu: 11

Počet pravidel nedeterministického konečného automatu: 35

Stagnační konstanta: 1000

Délka řetězců: 100

Počet řetězců v sadách: 100

Přírůstek mutace: 45

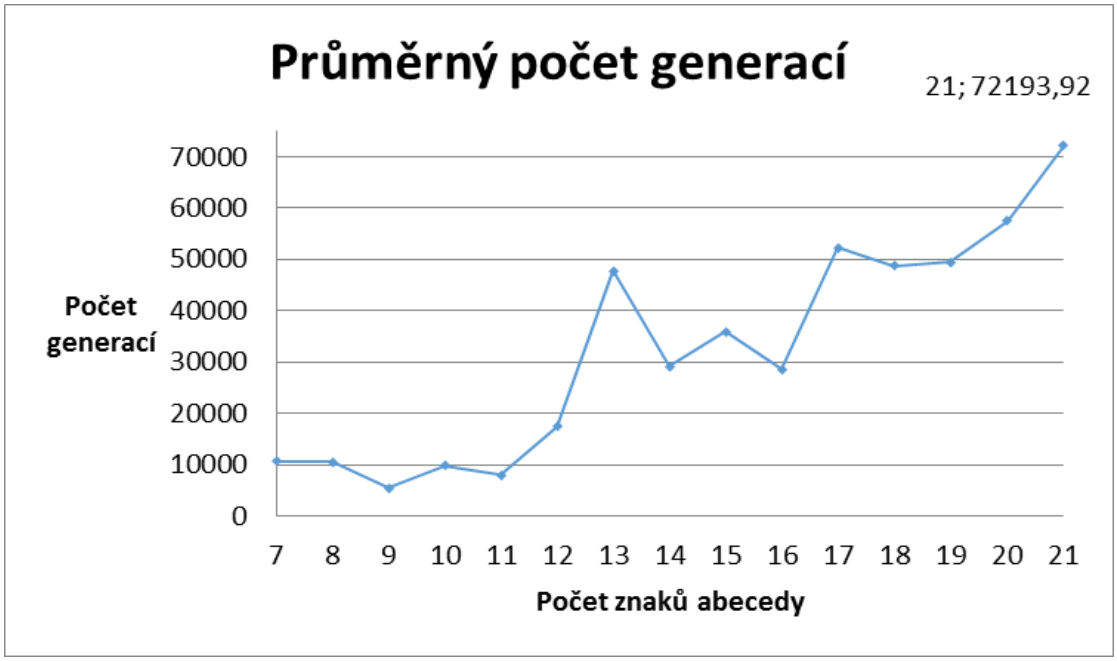
Počet měření: 100

Dále, testy byly prováděny pro 7 až 21 znaků abecedy, kdy bylo pozorována rychlost odvození nedeterministického konečného automatu, počet generací potřebných pro odvození nedeterministického konečného automatu a kvalita odvozeného nedeterministického konečného automatu v podobě počtu přijatých „kladných“ a „záporných“ řetězců viz tabulka 9.1.

	Generace	Čas	Pozitivní r.	Negativní r.
7	10668,55	511,0906104	96,95	34,44
8	10547,47	474,4989117	96,77	29,53
9	5524,15	187,641902	97,34	32,32
10	9885,08	278,9682684	99,69	28,46
11	8073,49	223,609899	98,33	28,63
12	17403,98	428,4624469	98,64	28,03
13	47721,02	825,767198	99,86	23,82
14	29103,69	641,709436	98,92	23,07
15	35849,66	725,397888	99,22	18,81
16	28601,78	514,076742	99,91	16,34
17	52343,82	813,113715	99,09	17,99
18	48764,14	839,9824657	99,98	14,47
19	49436,41	856,86657	99,98	11,88
20	57504,32	890,680219	99,99	7,22
21	72193,92	1087,499811	99,98	7,15

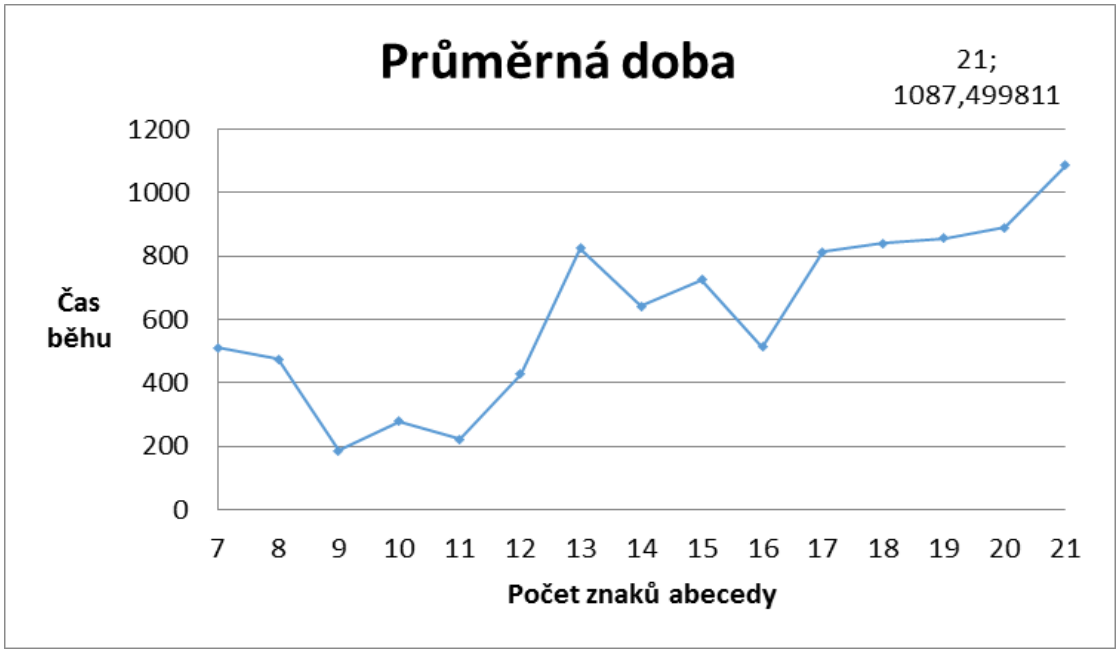
Tabulka 9.1: Průměrné výsledky

Všechny výsledky uvedené v tabulce byly vypočteny aritmetickým průměrem ze všech 100 měření. Grafy níže nám názorně ukáží, jak velké změny jsou v jednotlivých průměrných hodnotách při rozdílném počtu znaků v abecedě.



Graf 9.1: Průměrný počet generací

Na prvním grafu můžeme vidět, že první výraznější skok v počtu generací pro odvození nedeterministického konečného automatu je při přechodu z 12 znaků v abecedě formálního jazyka na 13 znaků abecedy. Dále můžeme vidět, že dochází k velkému snížení průměrného počtu generací pro počet znaků abecedy formálního jazyka 13 až 17, kdy hodnota opět skokově narůstá do předpokládaného průměrného počtu potřebných generací. Tento úkaz nás vede ke dvěma různým závěrům. Prvním závěrem může být, že průměrný počet potřebných generací pro odvození nedeterministického konečného automatu pro 13 znaků abecedy nemá mít takto skokovou hodnotu, přičemž by se jeho hodnoty měly pohybovat v rozmezí průměrných hodnot pro počet znaků abecedy 12 a 14. Druhým závěrem by mohlo být, že průměrný počet generací pro počet znaků abecedy 13 až 16 je pouhá náhoda, která by měla být odstraněna větším počtem měření. S ohledem na počet měření pro jednotlivé počty znaků abecedy formálního jazyka, tedy 100, se přikláním spíše k prvnímu závěru, kdy průměrný počet generací pro 13 znaků abecedy formálního jazyka je nedostatečně přesný a může být odstraněn větším počtem měření. Po počtu znak 17 abecedy již nevidíme žádné zvláštní odchylky od předpokladu, že s narůstajícím počtem znaků abecedy formálního jazyka narůstá také počet generací pro vytvoření nedeterministického konečného automatu.

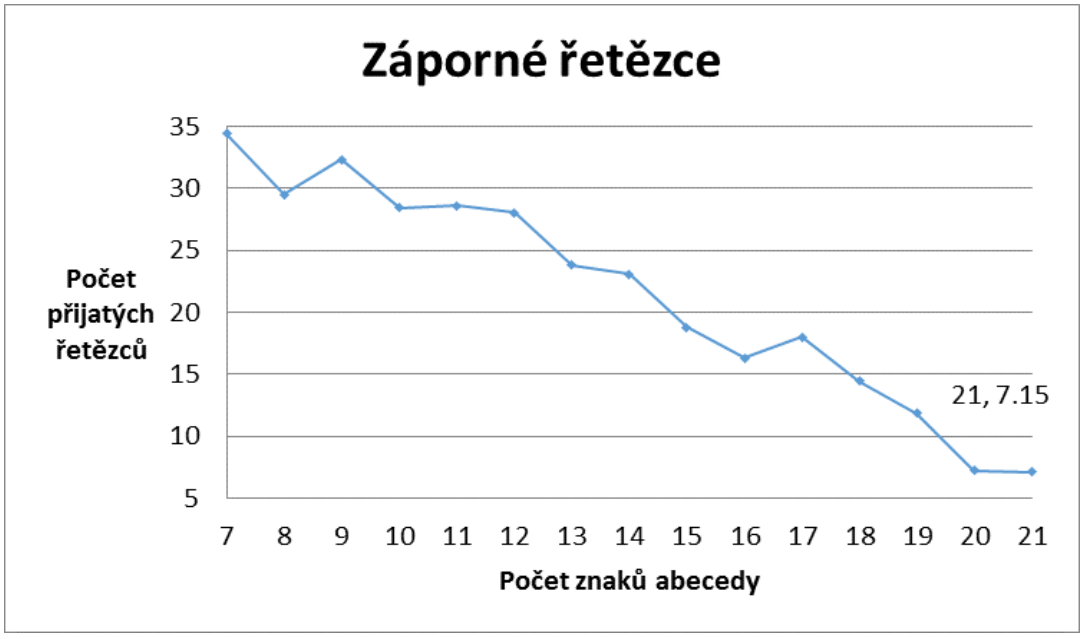


Graf 9.2: Průměrná doba odvozování

Na druhém grafu sledujeme průměrnou dobu potřebnou pro odvození nedeterministického konečného automatu pro formální jazyk. Na rozdíl od předchozího grafu je zde první skok vidět již při přechodu z 8 znaků abecedy formálního jazyka na 9. Abeceda o velikosti 7 a 8 má vyšší průměrnou dobu pro odvození nedeterministického konečného automatu. Tento jev je způsoben požadovanou velikostí výsledného nedeterministického konečného automatu, kdy zvolení vhodných parametrů na výsledný nedeterministický konečný automat, můžeme jeho odvození znatelně zrychlit. Pro možnost porovnání všech těchto měření byly zvoleny jednotné parametry, přičemž lze pozorovat, že pro ne všechny počty znaků abecedy byly tyto parametry vhodné. Mezi počty znaků abecedy formálního jazyka 13 až 17 sledujeme podobné kolísání průměrných hodnot jako u předchozího grafu. Zde vidíme souvislost mezi počtem generací potřebných pro odvození nedeterministického konečného automatu a potřebného času.

Graf 9.3: Počet přijatých „kladných“ řetězců

Nyní si zhodnotíme kvalitu výsledných odvozených nedeterministických konečných automatů. Na grafu 9.3, máme zobrazeny průměrné počty přijatých řetězců, patřících do modelovaného formálního jazyka. Jak můžeme vidět tak, nejhorší průměrná hodnota není menší jak 96,5 řetězce. Na grafu můžeme pozorovat, že od počtu znaků v abecedě 9 neklesá průměrný počet přijatých řetězců pod 98. Kolísání průměrného počtu přijatých řetězců, by mohlo být teoreticky odstraněno větším počtem měření. Počáteční výrazně nižší hodnoty pro počet znaků abecedy 7 až 9 si můžeme vysvětlovat jako důsledek počtu znaků abecedy.



Graf 9.4: Počet přijatých „záporných“ řetězů

Na grafu 9.4 zhodnotíme kvalitu odvozeného nedeterministického konečného automatu z hlediska počtu řetězců, které přijme navzdrory tomu, že do modelovaného formálního jazyka nepatří. Tento graf snižuje počet přijatých řetězců téměř linárně. Tento jev je způsoben narůstajícím počtem znaků v řetězci, který má za následek, že výsledný nedeterministický konečný automat je mnohem přesnější, neboť se snižuje pravděpodobnost, že bude více pravidel na stejných stavech s rozdílným symbolem a to zabraňuje řetězcům které do modelovaného jazyka nepatří, aby byly přijaty.

Další testy byly prováděny na předem připravených jazycích, kdy byla sledována schopnost vytvořit nedeterministický konečný automat pro jazyk, který je nám znám. Bylo provedeno deset měření s různými konfiguracemi výsledného nedeterministického konečného automatu a za pomoci testovacích a trénovacích sad obsahující 100 řětězců o délce 10 pro každý formální jazyk.

Testované jazyky:

1. a*b*
2. (a*)+(b*)
3. (ab)*

Generace	Čas	Kladné r.	Záporné r.	Poč. Kon. St.
1	0.007	49	100	1
1	0.104	100	100	1
3014	7.1027	49	65	1
3666	12.3838	95	21	1
1	0.0138	100	100	1
10897	33.5745	98	39	2
1	0.0152	0	30	1
1	0.0077	100	73	1
63	0.2889	99	11	1
1271	4.6561	24	42	2

Tabulka 9.2: Naměřené hodnoty pro první jazyk.

Jak můžeme vyčíst z tabulky, odvození, která byla úspěšně ukončena téměř okamžitě nedosahuje příliš dobrých výsledků. Mohlo by se zdát, že delší měření a vyšší počet potřebných generací pro odovzení nedeterministického konečného automatu dosahuje z hlediska přesnosti lepších výsledků, avšak tvrzení se nám nepotrvzuje kvůli výsledkům na řádku 3, kde je dosaženo průměrných výsledků a předposledním řádku kde je počet generací a celkový čas odovozování nízký s výsledky, které jsou nejlepší ze všech měření. Z tabulky tedy můžeme usoudit, že kvalita odvozeného nedeterministického konečného automatu je hodně závislá na jeho zvolené velikosti (počet stavů a pravidel).

Generace	Čas	Kladné r.	Záporné r.	Poč. Kon. St.
1	0.0079	44	60	1
1	0.0182	99	100	1
4007	13.4151	63	100	1
5955	15.9225	0	40	1
16	0.0694	1	26	1
1764	8.6677	100	9	1
8766	40.5941	0	46	2
3681	16.8145	1	46	3
4900	16.8964	34	46	5
1	0.0067	0	0	1

Tabulka 9.3 Naměřené hodnoty pro druhý jazyk.

Odvozování nedeterministického konečného automatu pro druhý jazyk nepřineslo příliš dobré výsledky. Jak můžeme vidět v tabulce, výsledné nedeterministické konečné automaty převážně přijímají více „záporných“ řetězců než „kladných“. Jedinou výjimkou je šesté měření, kdy výsledný nedeterministický konečný automat je téměř, pro testující sady řetězců, téměř perfektní.

Generace	Čas	Kladné r.	Záporné r.	Poč. Kon. St.
1	0.007	100	49	1
1	0.0077	100	100	1
1	0.0041	100	0	1
1	0.0116	100	100	1
4	0.01	100	0	1
1	0.0022	100	0	1
1	0.0126	100	100	1
3	0.0199	100	0	1
4	0.0154	100	0	1
11	0.0482	100	0	1

Tabulka 9.4: Naměřené hodnoty pro třetí jazyk.

Výsledky odvozování pro poslední jazyk jsou velice zajímavé. Všechny zvolené konfigurace velikosti výsledného nedeterministického konečného automatu přináší buď dokonalé výsledky, z pohledu testovacích sad řetězců, anebo naprosto nevyhovující. Co se týká množství času a počtu generací potřebných pro odvození výsledného nedeterministického konečného automatu je v nich zanedbatelný rozdíl.

10 Závěr

Cílem této práce bylo provést experiment v podobě možnosti využití upravené metody strojového učení pro problém odvozování modelů formálních jazyků. Řešení této práce bylo provedeno vhodnou úpravou metody genetické algoritmy a jako model neznámého formálního jazyka byl zvolen nedeterministický konečný automat.

Programová část práce se skládá ze dvou částí. První část slouží pro poskytování dostatečného množství dat pro ladění a následné testování výsledné aplikace. Řešením nedostatku dat se stalo generování vlastních formálních jazyků v podobě nedeterministických konečných automatů, které pak jsou dále využity pro generování trénovacích a testovacích sad řetězců potřebných pro správnou funkci aplikace. Generátor je přímo implementován jako součást aplikace, kdy je na uživateli zda jej využije nebo poskytne aplikaci své vlastní testovací a trénovací sady řetězců. Generování nedeterministického konečného automatu je řízeno pouze částečně a to pro zajištění všech požadovaných znaků abecedy formálního jazyka ve výsledném nedeterministickém automatu, na kterém se posléze generují řetězce.

V druhé části programové části práce, je řešena vhodná úprava metody strojového učení genetické algoritmy. Klasickým kódováním této metody jsou binární vektory, proto byl prvním problémem této práce změnit vhodně způsob zakódování našeho problému. Od způsobu kódování problému se pak odvíjí všechny provedené úpravy na genetickém algoritmu s cílem zachovat co nejvěrněji podobu upraveného genetického algoritmu vůči klasickému. V celé práci se proto pracuje s pravděpodobnostmi a částečně řízenou náhodou.

Výsledná aplikace byla testována na vlastních vytvořených datech. Výsledky některých měření byly překvapivé, například se ukázalo, že čas potřebný pro odvození vhodného nedeterministického konečného automatu může být v některých případech téměř stejný navzdory rozdílnému počtu znaků v abecedě neznámého modelovaného jazyka. Dosažené výsledky nám ukazují, že takto upravená metoda strojového učení pro problém odvozování modelů formálních jazyků je použitelná. Bohužel se nezdařilo nalézt algoritmus běžně používaný pro odvozování modelů formálních jazyků, který by byl implementován a následně testován stejnými daty, které by přinesly výsledky pro porovnání s výsledky námi navržené aplikace.

Aplikace má nyní konzolovou podobu. V budoucnu by bylo možné této aplikaci přidat grafické rozhraní pro ulehčení používání aplikace uživatelem. Aplikace je skoro plně konfigurovatelná uživatelem. Do budoucna by bylo vhodné navrhnout úpravy aplikace, aby byla schopna odvozování modelů formálních jazyků i se sadami řetězců, které by neobsahovaly pouze řetězce stejné délky. Další z možností úpravy do budoucna by mohlo být přepsání aplikace do jiného programovacího jazyka například C++, pokud by se tedy ukázalo, že je takto přepsaná aplikace efektivnější.

Literatura

[1] MURPHY, Kevin P. 2012. *Machine learning: a probabilistic perspective*. Cambridge: MIT Press, xxix, 1067 s. Adaptive computation and machine learning series. ISBN 978-0-262-01802-9.

[2] HYNEK, Josef. 2008. *Genetické algoritmy a genetické programování*. 1. vyd. Praha: Grada, 182 s. ISBN 978-80-247-2695-3.

[3] Yokomori, T.: Grammatical Inference and Learning. In: Formal Languages and Applications (Studies in Fuzziness and Soft Computing Volume 148), Springer, 2004, 507-528

[4] SCHWARZ, Josef. 2008. *Aplikované evoluční algoritmy: EVO*. Brno: Fakulta informačních technologií, 100 s.

[5] Zemek, P.: An alternative introduction to Genetic and Evolutionary Algorithms [online]. 12.6.2014. Dostupný z WWW: <http://www.codeproject.com/Articles/783225/An-alternative-introduction-to-Genetic-and-Evoluti>

[6] Sakakibara, Y.: Theoretical Computer Science 185 : Recent advances of grammatical inference. Department of Information Sciences, Tokyo Denki University, Hatoyama, Hiki-gun, Saitama 350-03, Japan, (1997) 15-45

[7] Meduna, A., Lukas, R. Formální jazyky a překladače: Studijní opora IFJ. [Online] Brno : Fakulta informačních technologií Vysokého učení technického v Brně, 2009-2012

[8] C.L. Giles, G.Z. Sun, H.H. Chen, Y.C. Lee, D. Chen, Higher order recurrent networks & grammatical inference, in: Advances in Neural Information Processing Systems, Vol. 2, Morgan Kaufmann, Los Altos, CA, 1990, pp. 380 - 387.

[9] D. Angluin, Queries and concept learning, Machine Learning 2 (1988) 319-342.

[10] E. M. Gold, Language identification in the limit, Inform. Control 10 (1967) 447-474.

[11] L. G. Valiant, A theory oft he learnable, Comm ACM 27 (1984) 1134-1142

[12] D. Angluin, Negative results for equivalence queries, Machine Learning 5 (1990) 121-150