VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

INFERENCE SKÁKAJÍCÍCH FORMÁLNÍCH MODELŮ

JUMPING FORMAL MODELS INFERENCE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TINA HEINDLOVÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání diplomové práce

Studentka: **Heindlová Tina, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Inteligentní zařízení
Název: **Inference skákajících formálních modelů**
Jumping Formal Models Inference
Kategorie: Teoretická informatika
Zadání:

1. Seznamte se s různými skákajícími formálními modely pro formální jazyky (např. skákající automaty a gramatiky).
2. Nastudujte různé metody strojového učení (se zaměřením na ty biologií inspirované) vhodné pro odvozování formálního modelu ze zadaného (konečného) podjazyka.
3. Dle konzultací s vedoucím navrhněte experimenty pro vyhodnocení odvozování modelů a v souvislosti s tím vyberte alespoň dva různé modely a dvě různé metody odvozování, které případně upravte, aby byly vhodné pro odvozování formálních modelů.
4. Implementujte vybrané metody a další potřebné nástroje pro realizaci experimentů (např. generátor konečného podjazyka ze zadaného formálního skákajícího modelu).
5. Proveďte experimentální vyhodnocení vybraných metod pro každý z modelů, především z hlediska vhodnosti pro odvozování.

Literatura:

- K. P. Murphy: *Machine Learning* (*A Probabilistic Perspective*), The MIT Press, 2012.
- J. Hynek: *Genetické algoritmy a genetické programování,* Grada, 2008.
- T. Yokomori: Grammatical Inference and Learning. In: Formal Languages and Applications (Studies in Fuzziness and Soft Computing Volume 148), Springer, 2004, 507-528.
- A. Meduna, P. Zemek: Jumping Finite Automata. *International Journal of Foundations of Computer Science*. 2012, roč. 23, č. 7, s. 1555-1578. ISSN 0129-0541.
- Z. Křivka, A. Meduna: Jumping grammars. International Journal of Foundations of Computer Science, 26(6):709-731, 2015.

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a částečně bod 3 a k tomu odpovídající prototyp.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 26. října 2021

Abstrakt

Práce se věnuje gramatické inferenci z hlediska evolučních algoritmů pro skákající formální modely. Nejdříve vysvětluje skákající formální modely, které se dělí na skákající gramatiky a skákající automaty. Poté popisuje gramatickou inferenci, evoluční algoritmy a všechny jejich důležité části, jako je generování řetězců a zjišťování členství řetězce do jazyka definovaného automatem. Daný algoritmus se pak aplikuje na vybrané druhy skákajících konečných automatů. Těmi jsou skákající konečné automaty, zobecněné skákající konečné automaty a doprava jednosměrně skákající konečné automaty. Testovány byly čtyři typy skákajících automatů a celkem bylo provedeno šestnáct experimentů. Z výsledků vyplývá, že inference fungovala nejlépe pro automaty bez větvení, a pro ty obsahující malé množství stavů a malou vstupní abecedu.

Abstract

This thesis is focused on grammatical inference in the way of evolutionary algorithms for jumping finite models. The first part explains jumping finite models as itself. More specifically, it describes jumping grammars and jumping automata. The next part deals with grammatical inference, evolutionary algorithms, and their important parts. According to the developed jumping models, said parts include strings generation and membership testing. These two algorithms are applied to chosen types of jumping finite automata—jumping finite automata, general jumping finite automata, and right one-way jumping finite automata. These four types of automata were tested, and in total, sixteen experiments were run. Results show that the inference works much better for automata without branching and with a small number of states and a small alphabet.

Klíčová slova

Skákající modely, skákající automaty, evoluční algoritmy, gramatická inference

Keywords

Jumping models, jumping automata, evolution algorithms, grammatical inference

Citace

HEINDLOVÁ, Tina. *Inference skákajících formálních modelů.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Inference skákajících formálních modelů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

. . . .
Tina Heindlová
16. května 2022

Poděkování

Chtěla bych poděkovat svému vedoucímu Ing. Zbyňku Křivkovi, Ph.D. za jeho čas a rady při psaní této práce.

Obsah

1 Úvod **2**

2 Klasické formální modely **4**
- 2.1 Gramatiky 4
- 2.2 Automaty 5

3 Skákající formální modely **6**
- 3.1 Skákající gramatiky 6
- 3.2 Skákající automaty 7
  - 3.2.1 Zobecněné skákající konečné automaty 9
  - 3.2.2 Jednosměrně skákající automaty 10

4 Evoluční algoritmy **13**
- 4.1 Genetické algoritmy 13
  - 4.1.1 Selekce 13
  - 4.1.2 Křížení a mutace 14
- 4.2 Gramatická inference 15

5 Návrh **16**
- 5.1 Genetický algoritmus 16
- 5.2 Testování členství řetězců do jazyka 18
- 5.3 Generování trénovací a testovací sady 20

6 Implementace **24**
- 6.1 Generování řetězců 24
- 6.2 Testování členství řetězců do jazyka 27
- 6.3 Genetický algoritmus 28

7 Experimenty **32**
- 7.1 JFA 33
- 7.2 Zobecněné skákající konečné automaty 41
- 7.3 Doprava jednosměrně skákající automat 43
- 7.4 Doprava jednosměrně zobecněný skákající automat 46

8 Závěr **48**

Literatura **49**

Kapitola 1

Úvod

Cílem této práce je otestovat funkcionalitu a použitelnost evolučních algoritmů pro inferenci skákajících formálních modelů. tím je míněno otestovat, zda evoluční algoritmy jsou schopny najít hledaný skákající model. Tato práce je inspirována pracemi *Grammar Induction and Genetic Algorithms* a *A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata*, které se zaměřují na inferenci klasických (neskákajících) konečných a zásobníkových modelů.

V první kapitole jsou krátce popsány klasické modely (myšleno neskákající), kterými jsou automaty a gramatiky. Je v ní vysvětleno, jak se definují tyto automaty a gramatiky, jaké typy existují a jak fungují.

Ve druhé kapitole jsou rozebrány skákající formální modely, kterými jsou skákající automaty a skákající gramatiky, a jejich rozdíly oproti dříve popsaným klasickým modelům. Protože se práce zabývá primárně skákajícími automaty jsou zde rozebrány více a důkladněji, než skákající gramatiky. Ale i ty jsou stručně popsány v této kapitole, hlavně jejich funkcionalita a rozdíly oproti klasickým gramatikám. Z hlediska skákajících automatů se v této kapitole vysvětlují jednotlivé typy, jejich funkčnost a jaké mezi sebou mají rozdíly. Byly vybrány jen některé typy skákajících automatů, těmi jsou skákající konečné automaty (JFA), zobecněné skákající konečné automaty (GJFA) a jednosměrně skákající konečné automaty.

Třetí kapitola se věnuje evolučním algoritmům a gramatické inferenci. Je zde popsáno, jak se evoluční algoritmy dělí a důkladněji se popisuje jedna jeho část, kterou je genetický algoritmus. Dále jsou popsány všechny dílčí metody, kterými je křížení, mutace, selekce a reprodukce. Po tomto algoritmu se rozepisuje gramatická inference, co je k ní potřeba a o co se snaží.

Ve čtvrté kapitole se práce věnuje návrhu jednotlivých složek pro implementaci. Jak se budou řešit a co vše je k nim zapotřebí. Popisují se veškeré podpůrné funkce, jako je generování řetězců podle jednotlivých skákajících automatů a testování členství řetězců do jazyka popsané těmito automaty. Tyto funkce jsou odlišné pro různý typ automatu a jsou zde rozebrány pro všechny vybrané druhy. Naposled se v této kapitole popisuje genetický algoritmus, hlavně návrh chromozomu a fitness funkce.

Po návrhu se věnuje implementačním detailům práce. Podrobně popisuje veškeré části genetického algoritmu a potřebných metod pro gramatickou inferenci z hlediska implementace.

Poslední kapitola se zaměřuje na experimenty pro všechny typy vybraných skákajících automatů a pro různá nastavení těchto skákajících automatů. Ukazuje odlišnosti vybraných

fitness funkcí a vybraných automatů. Hodnotí úspěšnost genetického algoritmu pro všechny typy vybraných automatů.

Kapitola 2

Klasické formální modely

Mezi klasické (neskákající) formální modely patří *konečné automaty* (*KA*) a *gramatiky*. V této sekci si zavedeme definice těchto formálních modelů a krátce si rozepíšeme, jak fungují.

2.1 Gramatiky

Gramatika je čtveřice $G = (N, \Sigma, P, S)$ [13], kde

- N - konečná neprázdná množina neterminálů (značeny velkými písmeny)
- Σ - konečná neprázdná množina terminálů (značeny malými písmeny)
- P - konečná množina přepisovacích pravidel, přičemž každé pravidlo je tvaru $(\Sigma \cup N)^*N(\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
- S - počáteční symbol, kde $S \in N$

Gramatika G generuje jazyk L podle přepisovacích pravidel. Vezme levou část pravidla a nahradí ji pravou částí. Například pravidlo $S \rightarrow ab$ nahradí neterminál S terminály ab.

Podle tvarů přepisovacích pravidel se gramatiky dělí na typ 0, typ 1, typ 2 a typ 3. Toto rozdělení zavedl pan Noam Chomsky [12] a říká se mu Chomského klasifikace gramatik nebo Chomského hierarchie jazyků [4].

Gramatiky typu 0, nazývané také neomezenými gramatikami, obsahují pravidla ve tvaru:

$$\alpha \rightarrow \beta,\ \alpha \in (N \cup \Sigma)^*N(N \cup \Sigma)^*,\ \beta \in (N \cup \Sigma)^*.$$

Gramatiky typu 1, též nazývané kontextové, obsahují pravidla ve tvaru:

$$\alpha A\beta \rightarrow \alpha\gamma\beta,\ A \in N,\ \alpha, \beta \in (N \cup \Sigma)^*,\ \gamma \in (N \cup \Sigma)^+,$$

nebo

$$S \in \varepsilon, \text{ pokud se } S \text{ neobjevuje na pravé straně žádného pravidla.}$$

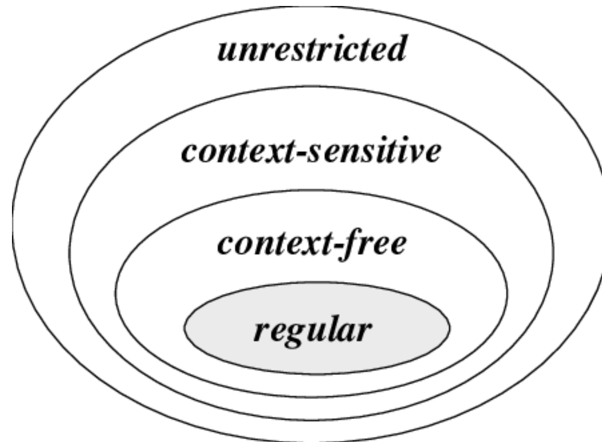
Gramatiky typu 2, též nazývané bezkontextové, obsahují pravidla ve tvaru:

$$A \rightarrow \gamma,\ A \in N,\ \gamma \in (N \cup \Sigma)^*.$$

Gramatiky typu 3, kterým se také někdy říká regulární a její pravidla jsou ve tvaru:

$$A \to xB \text{ nebo } A \to x;\ A, B \in N,\ x \in \Sigma^*.$$

Vztahy mezi jednotlivými typy gramatik lze vidět na obrázku 2.1, kde unrestricted = neomezené, context-sensitive = kontextové, context-free = bezkontextové a regular = regulární gramatiky.



Obrázek 2.1: Chomského hierarchie jazyků [17]

2.2 Automaty

Kromě gramatik patří mezi klasické formální modely i konečné automaty (KA). Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$ [13], kde:

- Q - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (abeceda)
- R - konečná množina pravidel tvaru $pa \to q \in R$, kde $R \subseteq Q \times (\Sigma^* \cup \varepsilon) \times Q$
- s - počáteční stav, kde $s \in Q$
- F - množina koncových stavů $F \subseteq Q$

Automat začne v počátečním stavu a čte vstupní symboly řetězce zleva doprava. Při přečtení symbolu přejde automat z daného stavu do jiného stavu, který je dán přechodovým pravidlem pro aktuální stav a daný symbol. Pokud automat přečte celý řetězec a skončí v koncovém stavu, pak automat řetězec přijal a řetězec patří do jazyka přijímané tímto automatem. V opačném případě automat řetězec nepřijal. Typy jazyků, které jsou přijímané konečnými automaty, jsou jazyky typu 3 (tedy regulární).

Pokud pro každé $p \in Q$ a pro každé $a \in \Sigma$ platí, že existuje maximálně jedno pravidlo $pa \to q \in R$, pak se jedná o *deterministický konečný automat* (DFA). V opačném případě se jedná o tzv. *nedeterministický konečný automat.*

Kapitola 3

Skákající formální modely

Skákající formální modely jsou rozšířením klasických modelů, jako jsou automaty a gramatiky. Tyto modely pracují sekvenčně zleva doprava. Vezmou vstupní řetězec a symbol po symbolu ho zpracují. Skákající modely nemusí pracovat sekvenčně. U těchto modelů může skákat jejich čtecí hlava při zpracování vstupního řetězce libovolným směrem a zpracovat znaky v libovolném nebo velmi volném pořadí. Takovými modely jsou skákající gramatiky a skákající automaty.

3.1 Skákající gramatiky

Skákající gramatiky jsou jedním ze skákajících formálních modelů. Klasická gramatika vezme část již vygenerovaného řetězce, podle pravidla ho přepíše na jiný a vloží zpátky na místo, odkud jej vzala. Například: Mějme již vygenerovaný řetězec aSa a gramatiku obsahující pravidlo $aS \to b$. Při aplikaci pravidla, se vezme část řetězce aS, nahradí se symbolem b a vloží se namísto aS. Výsledný řetězec pak bude ba. Skákající gramatika může vložit symbol b na libovolné místo řetězce (není vázaná na místo, odkud jej vzala). Může tedy například vygenerovat i řetězec ab.

Z formálního hlediska je skákající gramatika čtveřice $G = (N, \Sigma, P, S)$ [13], kde

- N - konečná neprázdná množina neterminálů
- Σ - konečná neprázdná množina terminálů
- P - konečná množina přepisovacích pravidel, přičemž každé pravidlo je tvaru $(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$
- S - počáteční symbol, kde $S \in N$

Tato gramatika G obsahuje 4 typy přepisovacích pravidel nad abecedou Σ^*. Derivace těchto pravidel se zapisují jako ${}_s\!\Rightarrow$, ${}_{lj}\!\Rightarrow$, ${}_{rj}\!\Rightarrow$ a ${}_j\!\Rightarrow$. [13]

Jsou dány u, $v \in \Sigma^*$. Pravidla gramatiky jsou definována následovně:

i) $u \, {}_s\!\Rightarrow v$ právě tehdy, když existuje $x \to y \in R$ a $w, z \in \Sigma^*$, pro které platí, že $u = wxz$ a $v = wyz$.

ii) $u \, {}_{lj}\!\Rightarrow v$ právě tehdy, když existuje $x \to y \in R$ a $w, t, z \in \Sigma^*$, pro které platí, že $u = wtxz$ a $v = wytz$.

iii) $u\,{}_{rj}\!\Rightarrow v$ právě tehdy, když existuje $x \to y \in R$ a $w, t, z \in \Sigma^*$, pro které platí, že $u = wxtz$ a $v = wtyz$.

iv) $u\,{}_{j}\!\Rightarrow v$ právě tehdy, když platí $u\,{}_{lj}\!\Rightarrow v$ nebo $u\,{}_{rj}\!\Rightarrow v$. [13]

Mezi základní druhy patří skákající bezkontextové gramatiky (CFGs), skákající regulární gramatiky (RGs), skákající pravé lineární gramatiky (RLGs) a skákající lineární gramatiky (LGs) [13].

3.2 Skákající automaty

Jak již bylo zmíněno na začátku této kapitoly, *skákající automaty* nezpracovávají vstupní řetězec sekvenčně zleva doprava, ale při zpracování vstupního řetězce může jejich čtecí hlava skákat libovolným směrem. Vezměme si například vstupní řetězec *abc*. Klasický automat (mluvíme o těch základních, ne o speciálních variantách) ho musí číst symbol po symbolu, nejdříve přečte a, pak b a nakonec c. JFA může například nejdřív přečíst symbol c, pak a a jako poslední b.

JFA je pětice $M = (Q, \Sigma, R, s, F)$ [13], kde:

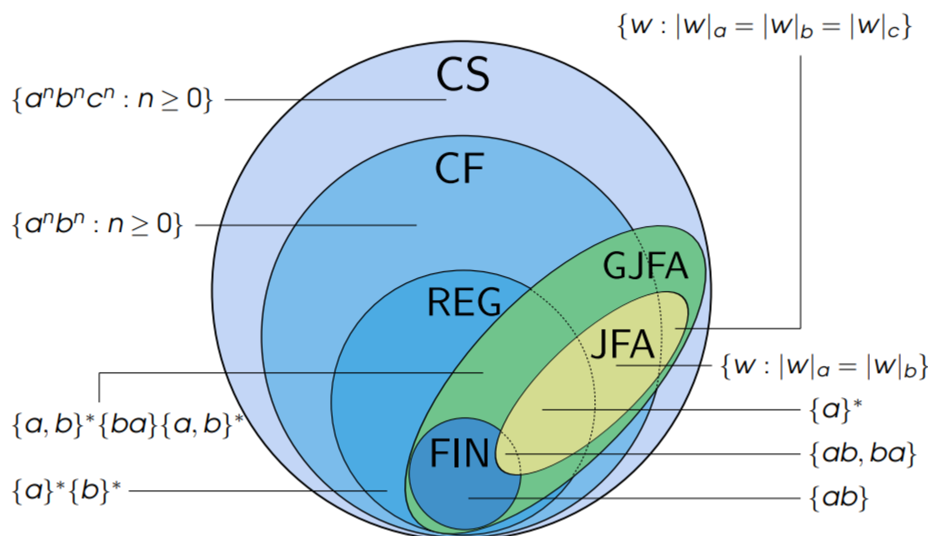
- Q - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (abeceda)
- R - konečná množina pravidel, kde $R \subseteq Q \times \Sigma^* \times Q$, přičemž pravidlo se zapisuje ve tvaru $py \to q \in R$ a všechny pravidla splňují podmínku $|y| \leq 1$.
- s - počáteční stav, kde $s \in Q$
- F - množina koncových stavů $F \subseteq Q$

Binární skoková relace, která se značí $\curvearrowright$, se definuje následovně:

Mějme zobecněný skákající automat M a $x, z, x', z' \in \Sigma^*$, přičemž platí $xz = x'z'$ a $py \to q \in R$, pak automat M skočí z konfigurace $xpyz$ do konfigurace $x'qz'$, psáno jako $xpyz \curvearrowright x'qz'$. [13]

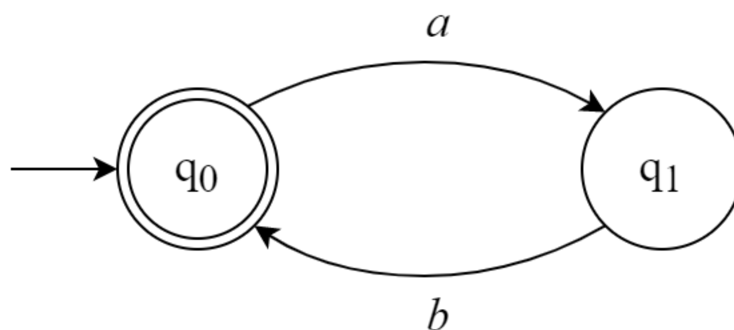
Existuje mnoho variant JFA. Samotný JFA může skákat libovolným směrem a čte vždy jen jeden symbol v rámci jednoho přechodu. Variantě JFA, které umožníme číst více symbolů během jednoho přechodu, se říká *zobecněný skákající konečný automat*, zkráceně GJFA. Další variantou jsou *jednosměrně skákající konečné automaty*, které mohou skákat vždy jenom jedním směrem. Těm se pak říká doprava či doleva jednosměrně skákající automaty podle směru, kterým mohou skákat. Podrobnější popis těchto automatů bude vysvětleno později v této sekci.

Vztahy mezi JFA, GJFA a ostatními typy formálních modelů lze vidět na obrázku 3.1, kde FIN, REG, CF a CS označují třídy **konečných, regulárních, bezkontextových** a **kontextových jazyků**.



Obrázek 3.1: Formální modely a vztahy mezi nimi i s příklady jazyků [14]

Příklad:
Mějme automat $M_1 = (\{q_0, q_1\}, \{a, b\}, \{q_0 a \rightarrow q_1, q_1 b \rightarrow q_0\}, q_0, \{q_0\})$.



Obrázek 3.2: Grafické znázornění automatu M_1

M_1 čte symboly a a b, přitom všechny tyto symboly se mohou nacházet kdekoliv ve vstupním řetězci. Vezměme si například vstupní řetězec *baba*. M_1 nemůže přečíst první symbol, protože pro něj nemá přechod v daném stavu. Může ale přečíst libovolné a ve vstupním řetězci. Přečte například první výskyt znaku a a ze vstupního řetězce zbude *bba*. Automat M_1 pak posune svou čtecí hlavu na libovolné místo, kde se nachází znak b. Vezměme například první výskyt tohohle znaku, ten se přečte a zůstane *ba*. M_1 opět posune svou čtecí hlavu k místu, kde se vyskytuje znak a, přečte ho a jako poslední přečte zbývající znak b. Tímto způsobem M_1 přijal vstupní řetězec *baba*.

Z ukázky lze vidět, že M_1 přijímá jazyk:

$$L_1 = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\},$$

který je **bezkontextový** a **není regulární**.

3.2.1 Zobecněné skákající konečné automaty

Zobecněné skákající konečné automaty (*General jumping finite automaton*), zkráceně GJFA mohou stejně jako JFA skákat libovolným směrem, ale oproti JFA mohou číst v rámci jednoho kroku více symbolů (řetězec). Například může obsahovat pravidlo $pab \rightarrow q$, které v jednom kroku přečte řetězec ab.

GJFA je pětice $M = (Q, \Sigma, R, s, F)$ [13], kde:

- Q - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (abeceda)
- R - konečná množina pravidel, kde $R \subseteq Q \times \Sigma^* \times Q$, přičemž pravidlo se zapisuje ve tvaru $py \rightarrow q \in R$.
- s - počáteční stav, kde $s \in Q$
- F - množina koncových stavů $F \subseteq Q$

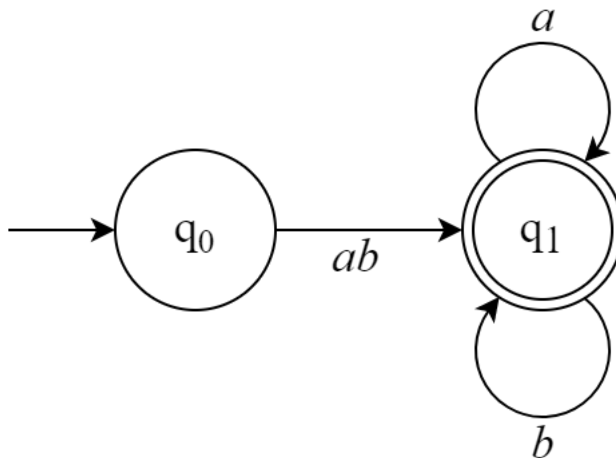
Od JFA se liší konečnou množinou pravidel, která obsahovala podmínku, že všechna pravidla musí splňovat $|y| \leq 1$. To u GJFA neplatí. Ten tuto podmínku neobsahuje.

Jazyk, který je přijímán automatem M je definován jako:

$$L(M) = \{uv \mid u, v \in \Sigma^*, usv \curvearrowright^* f, f \in F\},$$

kde $\curvearrowright^*$, neboli sekvence skoků, je reflexivní a tranzitivní uzávěr relace $\curvearrowright$ [14].

Příklad:
Je dán automat $M_2 = (\{q_0, q_1\}, \{a, b\}, \{q_0ab \rightarrow q_1, q_1a \rightarrow q_1, q_1b \rightarrow q_1\}, q_0, \{q_1\})$.



Obrázek 3.3: Grafické znázornění automatu M_2

Automat nejdřív přijme ab, které se může vyskytovat kdekoliv ve vstupním řetězci, pak následně čte symboly a a b. Jazyk, který tento automat přijímá je:

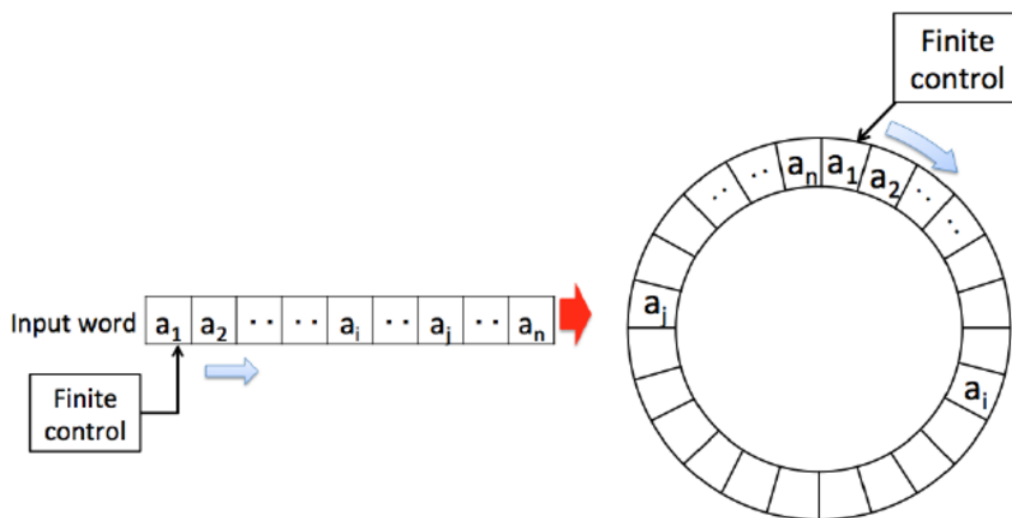
$$L_2(M_2, \curvearrowright) = \{a, b\}^*\{ab\}\{a, b\}^*.$$

3.2.2 Jednosměrně skákající automaty

Jednosměrně skákající konečné automaty (*one-way jumping finite automaton*) [13] jsou automaty založené na principu JFA a GJFA, akorát vstupní řetězec čtou pouze jedním směrem. Ty se dělí na doprava a doleva jednosměrně skákající automaty (*right/left one-way jumping finite automaton*). Fungování těchto automatů bude podrobněji vysvětleno v této sekci. Kromě těchto dvou dále ještě existují další typy skákajících automatů, jako jsou například *skákající zásobníkové automaty* nebo *obousměrně skákající automaty* (*two-way jumping automata* [8]).

Doprava jednosměrně skákající automaty

Doprava jednosměrně skákající automat M je založen na konečném automatu, který zpracovává vstupní řetězec zleva doprava jedním směrem, ale nečte vstupní řetězec znak po znaku. Automat M začne číst na začátku řetězce a pohne se doprava. Pokud na vstupu bude znak, pro který nemá M přechod, posune svou čtecí hlavu doprava a přesune se na následující znak. Tímto způsobem může automat M vynechat znaky, ke kterým se vrátí později. Pokud automat dojde na konec řetězce, posune se opět na začátek řetězce a pokračuje ve čtení znaků, které ještě nečetl. Řetězec lze tedy chápat jako cyklický *buffer*. Grafické zobrazení lze vidět na obrázku 3.4.



Obrázek 3.4: Grafické zobrazení čtení řetězce [3]

Formálně je to pětice $M = (Q, \Sigma, R, s, F)$, kde Q, Σ, R, s a F jsou definovány stejně jako u deterministického konečného automatu (KA) [3]. Tedy:

- Q - konečná neprázdná množina stavů
- Σ - konečná neprázdná množina vstupních symbolů (abeceda)
- R - množina přechodů, která se zapisuje ve tvaru $pa \rightarrow q \in R$ a pro kterou platí $R \subseteq Q \times \Sigma \times Q$ a zároveň pro každé $p \in Q$ a pro každé $a \in \Sigma$ platí, že existuje maximálně jedno pravidlo $pa \rightarrow q \in R$.
- s - počáteční stav, kde $s \in Q$

- F - množina koncových stavů $F \subseteq Q$

Relace skoku doprava jednosměrně skákajícího automatu se symbolizuje $\circlearrowright$ a je daná následovně:

Je dáno $x, y \in \Sigma^*$, $a \in \Sigma$ a stavy $p, q \in Q$, přičemž platí, že $pa \to q \in R$, pak automat M skočí z konfigurace $pxay$ do konfigurace qyx, psáno $pxay \circlearrowright qyx$, pokud $x \in \{\Sigma \setminus \Sigma_p\}^*$, kde $\Sigma_p = \{b \in \Sigma \mid pb \to q \in R \text{ pro některé } q \in Q\}$ [3].

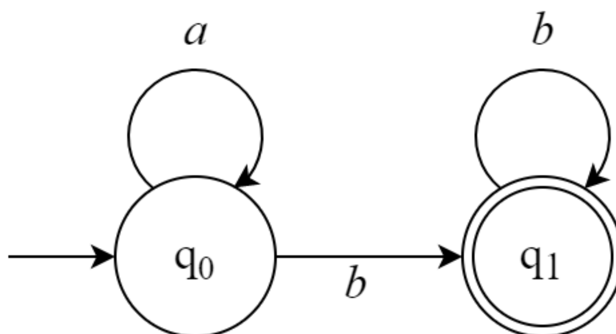
Jazyk přijímaný automatem M je definován jako:

$$L(M) = \{w \in \Sigma^* \mid sw \circlearrowright^* f, f \in F\},$$

kde $\circlearrowright^*$ je reflexivní a tranzitivní uzávěr relace $\circlearrowright$ [3].

Příklad:

Mějme doprava skákající automat $M_3 = (\{q_0, q_1\}, \{a, b\}, \{q_0 a \to q_0, q_0 b \to q_1, q_1 b \to q_1\}, q_0, \{q_1\})$.



Obrázek 3.5: Grafické znázornění automatu M_3

Automat M_3 čte, dokud má možnost číst. Tím pádem čte nejdříve všechny výskyty znaku a za sebou a následně znaky b. Pokud na začátku narazí na znak b, ten přečte a už nikdy se nemůže vrátit ke čtení znaků a.

Takový jazyk je pak definován jako:

$$L(M_3) = \{w \mid w = \{a\}^*\{b\}^+\}$$

a JFA ho nedokáže přijmout.

Doleva jednosměrně skákající automaty

Doleva jednosměrně skákající automat N pracuje podobně jako předchozí případ, ale začíná číst od konce řetězce, čte řetězec a provádí skoky směrem doleva. Dostane-li se na začátek řetězce, posune se na konec a pokračuje ve čtení zbylého řetězce.

Automat N je pětice $N = (Q, \Sigma, R, s, F)$, kde Q, Σ, s a F jsou definovány stejně jako u doprava jednosměrně skákajících automatů a R je množina přechodů $q \leftarrow ap \in R$ [3].

Relace skoku automatu N se značí $\circlearrowleft$ a je definována následovně:

Je dáno $x, y \in \Sigma^*$, $a \in \Sigma$, $p, q \in Q$ a $q \leftarrow ap \in R$, pak N skočí z konfigurace $yaxp$ do konfigurace xyq, psáno $yaxp \circlearrowleft xyq$, pokud $x \in \{\Sigma \setminus \Sigma_p\}^*$, kde $\Sigma_p = \{b \in \Sigma \mid q \leftarrow bp \in R$ pro některé $q \in Q\}$ [3].

Jazyk přijímaný automatem N je definován jako:

$$L(N) = \{w \in \Sigma^* \mid ws \circlearrowleft^* f, f \in F\},$$

kde $\circlearrowleft^*$ je reflexivní a tranzitivní uzávěr relace $\circlearrowleft$ [3].

Kapitola 4

Evoluční algoritmy

Evoluční algoritmy vznikly inspirací v přírodě podle Darwinovy teorie, kdy se jedinci žijící v populaci časem vyvíjí a mění, aby se přizpůsobili různým vnějším vlivům. Cílem evolučních algoritmů je najít jedince, který splňuje námi dané kritérium a využívají se například pro prohledávající a optimalizační úlohy.

Mezi tyto algoritmy se řadí genetické algoritmy, evoluční strategie, evoluční a genetické programování [15]. Všechny tyto metody využívají křížení, mutaci, selekci (výběr jedince z populace) a reprodukci. Tato práce se dál věnuje genetickým algoritmům a jejich použití pro gramatickou inferenci.

4.1 Genetické algoritmy

Genetické algoritmy (GA) poprvé představil John Holland v roce 1975. Ty vycházejí z neodarwinovské teorie přirozeného výběru jedinců, kde jenom ti nejlepší přežívají a reprodukují se. Jedinci se kříží, reprodukují, mutují a předávají svým potomkům část své genetické informace. [11]

Jedinci se zde říká chromozom a ten se skládá z posloupnosti genů. Jeden gen lze chápat jako jeden bit a chromozom jako řetězec bitů. Množině chromozomů se říká populace. Chromozomy v populaci se časem mění a každý z nich představuje jedno řešení daného problému.

Nejdříve se vytvoří náhodně populace chromozomů, každý se ohodnotí pomocí fitness funkce. Z nich se vybere určitý počet jedinců (selekce), kteří se budou reprodukovat (křížení). Z nich se vytvoří noví jedinci, kteří budou tvořit novou populaci. Může se k nim přidat část jedinců z předešlé generace (náhodně či ti s nejvyšší fitness funkcí). Pokud se do nové generace dostane n nejlepších jedinců z předešlé generace, mluvíme o takzvaném *elitismu*. Po vytvoření nových jedinců může u některých nastat mutace. Tohle celé se děje po dobu x generací, nebo do nalezení jedince s konkrétní fitness hodnotou.

4.1.1 Selekce

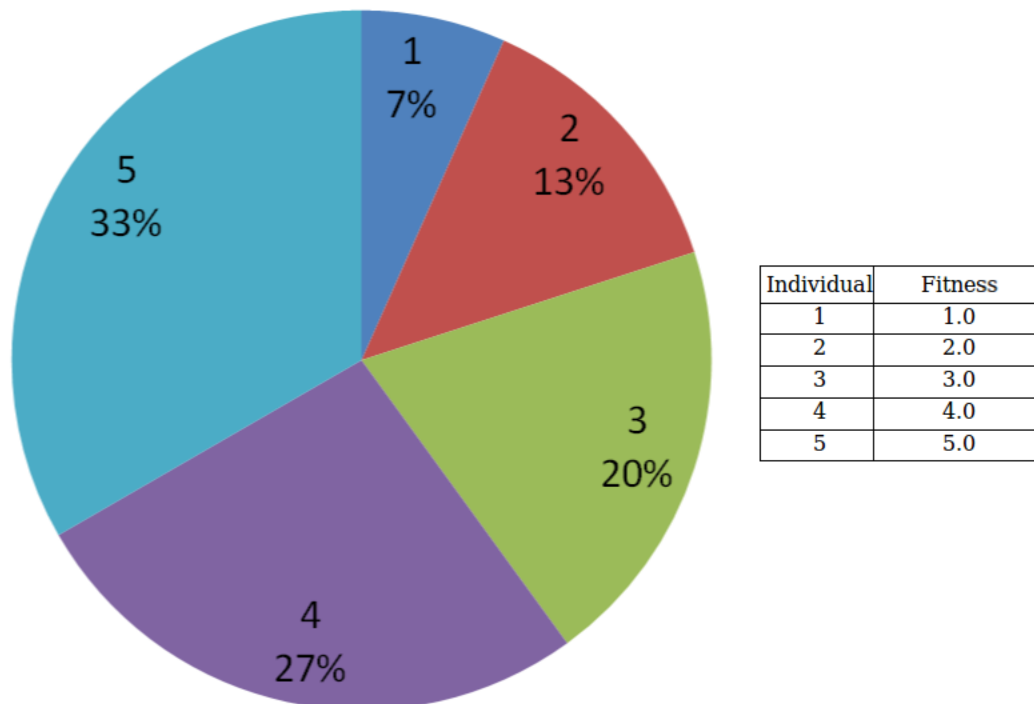
Selekce neboli výběr chromozomů je důležitou součástí genetických algoritmů. Aby mohl vzniknout nový chromozom, je potřeba nejdřív vybrat dva chromozomy z populace a nový následně vznikne křížením těchto dvou vybraných chromozomů. [5]

Volit chromozomy lze mnoha způsoby. Jedním z nich je pomocí rulety, která je znázorněna na obrázku 4.1. Každý jedinec je ohodnocen pomocí fitness funkce. Ta udává, jak moc

je jedinec lepší či horší vůči ostatním jedincům. Pravděpodobnost výběru jedince je pak daná vzorcem:

$$p_i = \frac{f_i}{\sum_{j=1}^{N_{pop}} f_j},$$

kde f_i je fitness hodnota jedince a N_{pop} je velikost populace [1].



Individual	Fitness
1	1.0
2	2.0
3	3.0
4	4.0
5	5.0

Obrázek 4.1: Výběr chromozomů pomocí rulety [1]

Dalším způsobem je turnaj. V turnaji se vybere k náhodných jedinců z populace a z nich se pak vybere ten nejlepší (ten s největší fitness hodnotou). Tímto způsobem se zaručí, že vybraný jedinec nemusí být tím nejlepším z celé populace, neboť nemusí být vybrán do turnaje. [7]

Kromě těchto dvou metod existují i jiné. Můžeme vybírat podle hodnoty (anglicky rank), kdy se nebere v potaz fitness jedince, ale jedinci je přidělena hodnota podle jejího fitness a výběr je pak určen touhle hodnotou. Nebo můžeme použít dva náhodné chromozomy z populace.

4.1.2 Křížení a mutace

Křížení je genetický operátor, kdy chromozomy předají část svého genetického kódu jinému. Vyberou se dva chromozomy, ty se zkříží a vytvoří pak nového jedince, který je jejich kombinací. Křížení může probíhat na jednom či více místech v chromozomu.

Jednobodové křížení funguje tak, že se vybere bod v řetězci a veškerá část kódu za tímto bodem se mezi oběma chromozomy vymění [16]. Příklad lze vidět níže. Noví jedinci pak obsahují X částí z jednoho chromozomu a dalších Y částí z druhého.

Rodič 1 = 111|0010

Rodič 2 = 000|1101
Potomek 1 = 111|1101
Potomek 2 = 000|0010

Vícebodové křížení je podobné jednobodovému, akorát místo jednoho bodu křížení je jich více [16]. Příklad pro vícebodové se dvěma body (konkrétně dvoubodové) lze vidět níže.

Rodič 1 = 111|0000|1010
Rodič 2 = 000|1111|1001
Potomek 1 = 111|1111|1010
Potomek 2 = 000|0000|1001

V uniformním křížení se nevybírají žádné body jako v předchozích dvou příkladech, ale každý gen potomka je vybrán náhodně ze dvou rodičů [16]. Příklad lze vidět níže.

Rodič 1 = 1111 1111
Rodič 2 = 0000 0000
Potomek 1 = 1011 1001
Potomek 2 = 0100 0110

Mutace je dalším operátorem genetických algoritmů. Některé geny jedinců mohou s pravděpodobností P náhodně mutovat. Mutací změní svou hodnotu genu, například z 0 na 1 a opačně. Vezmeme-li Potomka 1 z příkladu uniformního křížení a nastane u něho mutace na druhém a posledním genu, výsledný chromozom pak bude 1111 1000.

4.2 Gramatická inference

Gramatická inference (GI) nebo taky gramatická indukce je proces odvození gramatiky či automatu z trénovacích dat. Jsou dány konečné množiny S_+ a S_-. S_+ obsahuje řetězce patřící do jazyka L a S_- obsahuje řetězce, které do jazyka L nepatří, tzv. protipříklady. Jazyk L existuje, pokud pro něj existuje gramatika G (či automat M), pro který platí $S_+ \subseteq L(G)$ a $S_- \cap L = \emptyset$. Pro dostatečně velké množiny S_+ a S_- pak platí, že $L = L(G)$.

Je důležité říct, že chromozomy kódující formální model se nezvládnou naučit jazyk L bez trénovací sady protipříkladů. Bez protipříkladů by každá jiná množina S, kde $S_+ \subset S$, též patřila do jazyka L. Proto je v GI důležité mít trénovací sadu S_-.

Z hlediska evolučních algoritmů se jedná o proces odvození gramatiky, která je daná nějakou reprezentací a kdy se tato reprezentace mění a vyvíjí pomocí evolučních procesů. Zakódovaná gramatika (chromozom) se kříží, mutuje a snaží se splnit ukončující podmínku (mít námi požadovanou fitness hodnotu).

Roku 1978 pan Angluin dokázal, že pokud budeme hledat nejmenší deterministický konečný automat M z předem dané trénovací sady, který splňuje podmínky $S_+ \subseteq L(M)$ a $S_- \cap L(M) = \emptyset$, bude se jednat o NP-těžký problém [2].

Problém rozhodnutelnosti, zda existuje deterministický konečný automat M pro aspoň t stavů, kde t je kladné celé číslo, a který splňuje tu jistou podmínku ($S_+ \subseteq L(M)$ a $S_- \cap L(M) = \emptyset$), je NP-úplný. Tvrzení bylo dokázáno v roce 1967 panem Goldem. [10]

Celá tato sekce vychází z článku *A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata* [11].

Kapitola 5

Návrh

Tato kapitola se věnuje návrhu evolučního algoritmu pro skákající konečné automaty. Je zde popsáno zakódování automatu do chromozomu, samotný genetického algoritmu, jeho fitness hodnota a všechny důležité podsložky pro GA jako je testování členství řetězců do jazyka popsané skákajícím automatem a generování trénovací sady S_+ a S_-.

5.1 Genetický algoritmus

Obecný princip genetických algoritmů byl popsán v kapitole 4. Na začátku genetického algoritmu se náhodně vytvoří populace chromozomů. Z hlediska inference automatů je chromozom zakódováním automatu. Přesněji řečeno jsou v něm zakódovány přechody automatu.

Genetický algoritmus tvoří populace chromozomů a jednou z jeho nejdůležitější částí pro inferenci automatů je zakódování jejich přechodů do chromozomu. Protože se může jednat o nedeterministické varianty automatů, bude chromozom obsahovat každý přechod pro každý symbol automatu do každého stavu. Předem určíme maximální počet stavů Q, abecedu Σ, celkový počet jejich symbolů a množinu koncových stavů F. Pak lze vytvořit řetězec bitů tak, že každý bit bude představovat jeden přechod automatu.

Nevýhodou tohohle způsobu je, že automat $M = (Q, \Sigma, R, s, F)$ bude vyžadovat chromozom délky $l = |Q| \cdot |\Sigma| \cdot |Q|$, ale z hlediska funkcionality GA je tento způsob výhodný, protože jsou všechny chromozomy stejné délky a každý bit má stejnou interpretaci. Tudíž lze použít veškeré genetické operátory, jako jsou například křížení a mutace, které byly zmíněny v kapitole o genetických algoritmech 4. Pokud bychom měli chromozomy odlišných délek, nebylo by možné je bez předchozích úprav křížit. Mohl by vzniknout chromozom, jehož geny by bylo složitější interpretovat a vyžadovaly by dřívější či pozdější úpravy. Například by vznikl chromozom obsahující víckrát tytéž přechody, ale s rozdílnou hodnotou genu atd.

Další důležitou součástí genetického algoritmu je fitness funkce, která udává, jak moc se chromozom přibližuje či oddaluje od námi požadovaného řešení. Hledáme takový chromozom, který generuje všechny řetězce ze sady S_+ a negeneruje jediný ze sady S_-. Jednou z variant (ta nejjednodušší) je zvolit fitness funkci:

$$f = \frac{x_+ + x_-}{n+m}, \text{ kde}$$

- x_+ je počet správně přijatých řetězců ze sady S_+
- x_- je počet správně odmítnutých řetězců ze sady S_-

- n je počet řetězců v S_+
- m je počet řetězců v S_-.

Hodnoty fitness funkce se nachází v intervalu $f \in \langle 0; 1 \rangle$ a cílem je najít takový chromozom, který má fitness hodnotu 1. Z této fitness funkce jde vidět, že čím víc řetězců dokáže správně rozřadit, tím vyšší tato fitness hodnota bude.

Tato metoda výpočtu fitness hodnoty ale obsahuje jednu nevýhodu. Mějme počet řetězců v $|S_+| = |S_-|$. Chceme, aby se genetický algoritmus, co nejvíc zlepšoval. Ale, pokud budeme mít automat M, který přijímá úplně každý řetězec, bude jeho fitness $f_M = 0,5$. M přijme všechny v S_+ i v S_-. Stejně tak, budeme-li mít automat M_2, který nepřijímá jediný řetězec. Fitness tohohle automatu bude též $f_{M_2} = 0,5$. Přitom ale v těchto dvou případech chceme, aby měl, co nejmenší fitness funkci, protože ani jeden nepatří mezi ty, které by se blížily námi hledaného řešení. Na tento problém narazili autoři článku o genetické inferenci zásobníkových automatů [11]. Jejich úprava, aby docílili většího zlepšování genetického algoritmu, tkvěla v tom, že vynásobí výsledky správně přijatých a odmítnutých řetězců. Tedy:

$$f_1 = \frac{x_+}{n} \cdot \frac{x_-}{m},$$

kde x_+, x_-, n a m znamenají to stejné jako v případě výše.

Tím eliminují, že automaty přijímající a odmítající všechno budou mít fitness hodnotu $f = 0,5$ a vrací hodnoty v rozmezí $f_1 \in \langle 0; 1 \rangle$. Danou rovnici ještě upravili, aby se fitness hodnota vylepšovala podle délky správně přijatých prefixů všech řetězců. [11]

Pokud bychom chtěli tento způsob použít, potřebujeme ho upravit, aby fungoval i pro skákající automaty. Nemůžeme se dívat na délku prefixů jako to udělali pro zásobníkové automaty, ale musíme se dívat na počet správně přijatých symbolů řetězce. Tato fitness funkce pak bude vypadat následovně:

Nadefinujme si nejdříve funkci pro sumu přijatých symbolů:

$$accepted(M, w) = \frac{symbols}{|w|},$$

kde *symbols* je počet přijatých symbolů řetězce automatem M a $|w|$ je délka řetězce.

Funkce pro všechny řetězce S_+ je pak ve tvaru:

$$accepted(M, S_+) = \frac{1}{|S_+|} \sum_{w \in S_+} accepted(M, w)$$

Tato funkce vrací hodnotu od 0 do 1. Pokud přijme všechny řetězce, pak vrátí 1. Pokud nepřijme žádnou část žádného řetězce, pak vrátí hodnotu 0.

Fitness hodnota je pak:

$$f_2(M, S_+) = (accepted(M, S_+) \cdot \frac{1}{2} + \frac{x_+}{n} \cdot \frac{1}{2}) \cdot \frac{x_-}{m} \text{ [11]}$$

a vrací hodnotu $f_2(M, S_+) \in \langle 0; 1 \rangle$.

Pokud už máme představu o tom, jak bude vypadat chromozom a fitness funkce, pak už nám jen stačí vysvětlit, jak pracuje genetický algoritmus. Ten funguje podle následujícího pseudokódu:

Algorithm 1 Genetický algoritmus

vytvoření a inicializace populace
nastav počítadlo generací na 0
ohodnoť každého jedince pomocí fitness funkce
while *nenašli jsme jedince s požadovanou fitness nebo nedosaženo X generací* **do**
  selekce jedinců z populace
  vytvoř novou populaci pomocí křížení a mutace
  ohodnoť jedince pomocí fitness funkce
  zvyš počítadlo generací o 1
end
vrať nejlepšího nalezeného jedince (toho s nejlepší fitness hodnotou)

5.2 Testování členství řetězců do jazyka

Na začátku této kapitoly byl zmíněn genetický algoritmus a jeho důležitá část, kterou je fitness funkce. V ní bylo rozhodnuto, že se bude tvořit ze sady pozitivních příkladů S_+ a negativních příkladů S_-. Cílem je najít automat, který generuje všechny řetězce ze sady S_+ a negeneruje žádný ze sady S_-. K tomu je nutná funkce, která zjistí členství řetězce do jazyka, který je dán nějakým automatem.

Snažíme se zjistit, zda pro zadaný řetězec $w \in \Sigma^*$ a automat M platí, že $w \in L(M)$ [9]. Tedy, zda automat přijímá daný řetězec či nikoliv. Dle článku Characterization and complexity results on jumping finite automata [9] z roku 2017 dostupném v časopise Theoretical Computer Science je tento problém pro libovolné $w \in \Sigma^*$ NP-úplný problém, ale máme-li fixní slovo a JFA, lze tento problém řešit v polynomiálním čase. V našem případě testujeme jen některé řetězce $w \in \Sigma^*$ a tím pádem, je daný problém pro JFA řešitelný v polynomiálním čase. V nejhorším případě musíme vyzkoušet všechny možnosti, aby se dalo říct, zda řetězec patří do daného jazyka. Pro testování, zda řetězec nepatří do daného jazyka musíme vyzkoušet všechny tyto možnosti.

JFA

Způsob popsaný v článku [9] využívá toho, že JFA pracuje po jednom znaku a posouvá svou čtecí hlavu na libovolné místo řetězce. Tím pádem se lze na řetězec w dívat jako na počet výskytů jednotlivých symbolů abecedy Σ a nemusíme řešit pořadí jednotlivých znaků. Tomuhle způsobu se říká *Parikhovo mapování* [9].

Definice:
Mějme abecedu $\Sigma = a_1, a_2, ..., a_k$, pak *Parikhovo mapování* je funkce $p(w) = (|w_{a_1}|, |w_{a_2}|, ..., |w_{a_k}|)$, kde $|w_{a_1}|$ je výskyt všech znaků a_1 v řetězci w [6].

Příklad:
Mějme abecedu $\Sigma = a, b$ a slovo $w = abbabb$, pak dle *Parikhova mapování* je $p(w) = (|a|, |b|)$, neboli $p(w) = (2, 4)$.

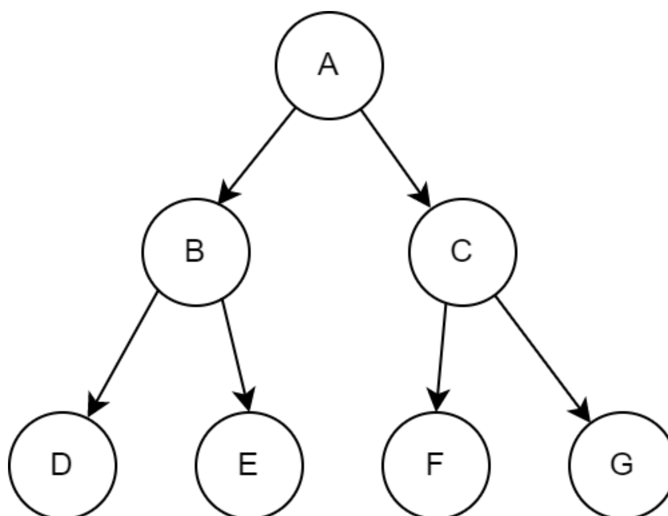
Vytvoříme vektor výskytů znaků abecedy podle *Parikhova mapování* a následně simulujeme průběh automatu M. Pokud existuje přechod pro dané $x \in \Sigma$, pak se sníží hodnota v adekvátní složce vektoru. Pokud se došlo do koncového stavu a všechny složky vektoru jsou nulové, pak $w \in L(M)$. [9] Použití daného způsobu sníží časovou složitost pro JFA z exponenciálního na polynomiální pro fixní řetězce.

Zobecněné skákající automaty

Pro GJFA je dle stejného článku [9] tento problém jak pro libovolná $w \in \Sigma^*$, tak pro fixní řetězce, NP-problém a dříve uvedený způsob není možné použít. Je to způsobeno tím, že GJFA nečte řetězce po jednom znaku, ale po několika a může tedy do sebe symboly vkládat. Konkrétní příklad bude zmíněn později v této sekci v podsekci o generování trénovací sady pro GJFA.

Testovat členství řetězců u GJFA lze řešit různými metodami, které prohledávají stavový prostor. Jednou z nich je například *prohledávání se zpětným navrácením*, anglicky *backtracking*. Ten patří mezi metody, které prohledávají do hloubky a má exponenciální složitost. Systematicky prochází všechny možnosti, aby se dostal ke hledanému řešení a skončí úspěšně, nalezne-li toto řešení, nebo skončí neúspěšně, prohledá-li všechna možnosti a nenarazí na hledané řešení.

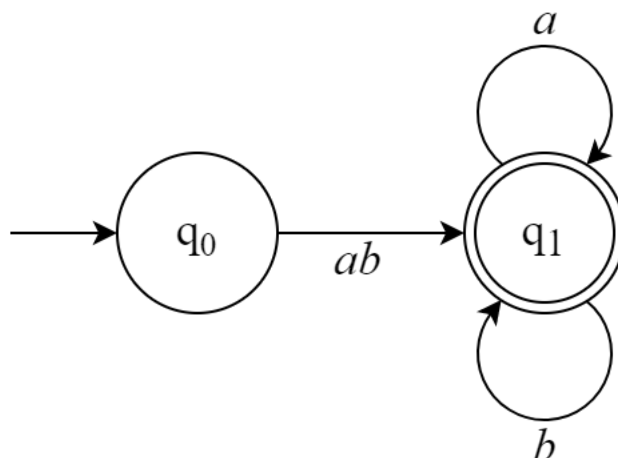
Například vezměme si následující případ, který je graficky znázorněn na obrázku níže. Kolečka označena A-G značí možnosti řešení. Metoda prochází možnosti v pořadí ABDECFG a narazí-li na řešení, vrátí úspěch a prohledávání končí.



Obrázek 5.1: Grafické znázornění metody backtracking

Z hlediska testování členství funguje tato metoda následovně. Vezme se řetězec a simuluje se průchod zobecněným skákajícím konečným automatem. Začne se v počátečním stavu a hledá se přechod, který čte symbol (případně množinu symbolů), který se nachází v řetězci. Postupně se zkouší všechny symboly řetězce, dokud se nenajde existující přechod. Ten se pak z řetězce odstraní, posune se aktuální stav a v něm se pak hledá další existující přechod. Nenajde-li jediný použitelný přechod, vrátí se simulace o krok zpět a zkouší se další přechody. Simulace pak pokračuje z tohohle stavu a odstraní-li se všechny symboly z řetězce a poslední stav byl koncový, automat tento řetězec přijímá. V opačném případě automat řetězec nepřijímá.

Příklad: Je dán zobecněný skákající automat $M_1 = (\{q_0, q_1\}, \{a, b\}, \{q_0ab \to q_1, q_1a \to q_1, q_1b \to q_1\}, q_0, \{q_1\})$, který přijímá jazyk $L_1 = w \in \{a, b\}^*\{ab\}^*\{a, b\}^*$, a řetězec *aabb*.



Obrázek 5.2: Grafické znázornění automatu M_1

Nejdříve se vezme první symbol (a) a hledá se existující přechod. Ten se nenajde, tak se přejde na další symbol řetězce. Pro druhé a též neexistuje přechod, ale existuje přechod pro ab. Tyto symboly se přijmou, smažou z řetězce (zbude z něj pak ab), posune se aktuální stav a znova se hledá přechod pro první symbol a. Teď už pro něj existuje přechod, tak se přijme a smaže z řetězce. Následně se aktualizuje aktuální stav a zbude poslední symbol b. Ten se též přijme, a protože se skončilo v koncovém stavu, řetězec patří do jazyka L_1 přijímané automatem M_1.

Doprava jednosměrně skákající automaty

Testování doprava jednosměrně skákajících automatů funguje obdobným způsobem dle funkcionality těchto automatů. Automat vezme řetězec, prochází ho zleva doprava a hledá existující přechod. Pokud existuje, symbol se z řetězce smaže a pokračuje se dalším symbolem. Pokud přechod neexistuje, tento symbol se přeskočí a vstupní hlava se posune o jednu pozici doprava. Pokud se narazí na konec řetězce, vstupní hlava se posune zpátky na začátek a pokračuje se ve čtení. Toto se opakuje, dokud nebyl celý řetězec přijat (smazán), nebo dokud se neprošel celý řetězec a nebyl z něho odstraněn jediný symbol. Tedy, nebyl aplikován jediný přechod a automat postupně přesunul svou čtecí hlavu zpátky na začátek řetězce.

Výhodou doprava jednosměrně skákajících automatů je, že nemají exponenciální složitost, díky jejich determinismu a díky tomu, že pokud můžou číst symbol, tak ho musí přečíst a nemají možnost ho přeskočit a přečíst jiný.

5.3 Generování trénovací a testovací sady

Jak již bylo zmíněno v kapitole 4 o gramatické inferenci, je potřeba mít trénovací sadu S_+ a S_-. Kromě trénovacích sad je potřeba mít i testovací sady S_{test+} a S_{test-}. Tvorba trénovací a testovací sady se provádí stejným způsobem, ale je odlišná v rámci typu automatu, kterými jsou JFA, GJFA a jednosměrně skákající konečné automaty. Tyto způsoby budou popsány samostatně.

JFA

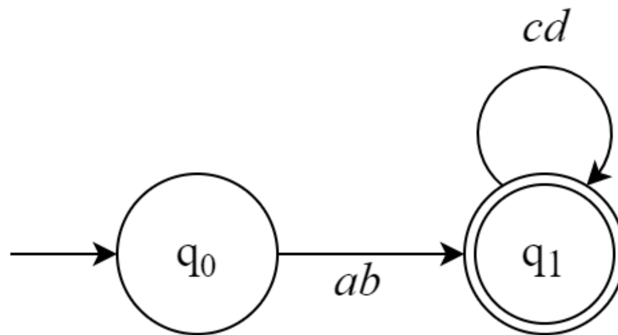
Nejdříve si popíšeme tvorbu sady S_+ pro JFA. Sada vzniká podle zadaného automatu M, který obsahuje počáteční stav, množinu koncových stavů, počet stavů, přechodová pravidla a abecedu. Protože JFA čte po jednom znaku, neexistuje zde možnost zanoření řetězců jako je to možné u GJFA. Tím pádem můžeme generovat řetězce simulováním funkcionality automatu. Začne se v počátečním stavu a náhodně se vybírají přechody z daného stavu. S každým přechodem se zapíše adekvátní znak abecedy do výsledného řetězce na náhodné místo tohohle řetězce (JFA může posouvat svou čtecí hlavu na náhodné místo řetězce). Jakmile se dojde do jednoho z koncových stavů, generování skončí. Takto vzniklý řetězec je jedním ze sady trénovacích řetězců S_+. To se zopakuje několikrát a vzniklé řetězce pak vytvoří sadu S_+.

Sada S_- se generuje obdobně. Též se generuje podle automatu M, ale přidá se pár dříve neexistujících přechodů a všechny stavy se označí jako koncové. Takto vzniklý automat se bude lišit od zadaného automatu M a bude přijímat jiný jazyk, jiné řetězce a tím pádem i tvořit jiné řetězce. Tato úprava ale nezaručí, že takto upravený automat nebude taky přijímat a vytvářet řetězce, které vytváří a přijímá automat M. Každý vytvořený řetězec je nutné otestovat, jestli nepatří do jazyka přijímané automatem M a trénovací sady S_+. Do sady S_- se pak přidají jen ty řetězce, které do tohohle jazyka nepatří.

Protože se řetězce vytvářejí náhodným způsobem z automatů, mohou vzniknout duplicity a v takovém případě se tyto řetězce do sad nepřidají. Zpomalilo by to průběh genetického algoritmu.

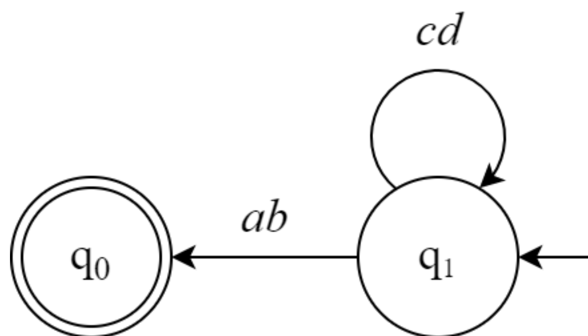
Zobecněné skákající automaty

Tvorba sad pro GJFA nemůže probíhat jako u JFA, protože může vzniknout zanoření. Například vezměme si automat $M_1 = (\{q_0, q_1\}, \{a, b, c, d\}, \{q_0ab \to q_1, q_1cd \to q_1\}, q_0, \{q_1\})$.



Obrázek 5.3: Grafické znázornění automatu M_1

Pokud bychom vytvářeli řetězce stejným způsobem jako u JFA, M_1 nejdříve vytvoří část řetězce *ab* a pak bude někam vkládat znaky *cd*. Tímto způsobem by mohl například vzniknout řetězec *acdb*. Ale automat M_1 tento řetězec nepřijímá. M_1 chce nejdříve číst znaky *ab* a následně *cd*. Tím pádem není možné použít stejný způsob jako u JFA. Aby GJFA generovalo správně řetězce, musí se vytvářet opačným způsobem. Transformujeme automat M_1 na automat $M_2 = (\{q_0, q_1\}, \{a, b, c, d\}, \{q_1ab \to q_0, q_1cd \to q_1\}, q_1, \{q_0\})$.



Obrázek 5.4: Grafické znázornění automatu M_2

M_2 vznikl prohozením stavů v přechodech (přechod ze stavu q_0 do stavu q_1 se změnil na přechod ze stavu q_1 do stavu q_1, atd.) a prohozením počátečního a koncového stavu. Pokud by automat měl více koncových stavů, bude se pro každý řetězec vybírat náhodně jeden z nich. Řetězec se pak vytvoří obdobným způsobem jako se tvoří řetězce pro JFA.

Jednosměrně skákající automaty

Dále si rozepíšeme, jak generovat sady pro doprava jednosměrně skákající automaty a doprava jednosměrně zobecněné automaty. Sady těchto automatů nelze generovat stejným způsobem, jako se generovaly řetězce pro JFA nebo GJFA, protože skáčou jen jedním směrem a pokud můžou aplikovat přechod, tak ho aplikují. Například již dříve zmíněný automat M_3 se chová stejně jako klasický neskákající automat. Čte symboly a a pokud narazí na symbol b, přečte ho a už nikdy nemůže přijmout a. JFA tento jazyk nedokáže přijmout a tím pádem generování řetězců stejným způsobem není možné. Toto generování musíme doplnit o testování členství řetězce do jazyka, aby ho bylo možné použít. Řetězce doprava jednosměrně skákajících automatů jsou generovány podobným způsobem jako jsou generovány řetězce JFA/GJFA, ale jsou doplněny o tento test a generování se provádí jiným ale ekvivalentním způsobem.

Ten je odvozen od funkcionality klasických neskákajících modelů, protože všechny doprava jednosměrně skákající automaty přijímají řetězce, které generují i tyto modely. Tím pádem stačí simulovat průchod automatem a uložit za sebe všechny přečtené symboly. Pro inferenci by pak mohlo stačilo použít jen tyto řetězce, ale protože se tím omezí testovací sada, byla tato metoda upravena, aby se v ní nacházely i jiné řetězce, které generuje doprava jednosměrně skákající automat. Místo, aby se symboly za sebe ukládaly při simulaci, se všechny symboly nejdříve uloží do struktury, ze které se pak budou náhodně vybírat a vkládat za sebe. Tím se vytvoří řetězec s náhodně seřazenými symboly, ale u takto vytvořených řetězců není jistota, že je bude automat přijímat. Proto se všechny takto vytvořené řetězce musí otestovat, zda je automat přijímá či ne. Ačkoli tento způsob obsahuje testování členství řetězců, vygeneruje větší škálu řetězců, než dříve zmíněný způsob, který toto testování nepotřeboval.

Vezměme si například doprava jednosměrně skákající automat M_1, který byl zmíněn v sekci 3. Tento automat generuje stejný jazyk jako generoval JFA, tedy jazyk generující stejný počet symbolů a a symbolů b. Při simulaci se budou ukládat všechny přečtené symboly a a b. Po skončení simulace pak budeme mít například uložené symboly a, b, a, b v tomto pořadí. Ty se poté vyberou v náhodném pořadí a vytvoří řetězec. Takovým způsobem může vzniknout jeden z následujících řetězců *aabb*, *abab*, *bbaa*, *baba*, *abba* a *baab*. Výsledný řetě-

zec se pak otestuje, zda ho automat přijímá. V našem případě M_1 přijímá všechny takto vygenerované řetězce.

Tímto způsobem se generují jak řetězce sady S_+, tak řetězce sady S_-. S tím rozdílem, že řetězce sady S_- se generují podle upraveného automatu, jehož úprava je stejná jako u JFA.

Testovací řetězce S_{test+} a S_{test-} se generují stejným způsobem jako trénovací. V sadě S_{test+} se nacházejí řetězce patřící do jazyka popsané automatem M a S_{test-} obsahuje řetězce, které nepatří do jazyka popsané automatem M. Dále platí, že že $S_{test+} \cap S_+ = \emptyset$ a $S_{test-} \cap S_- = \emptyset$.

Kapitola 6

Implementace

Programová část byla napsána v jazyce C/C++ a je rozdělena na dvě samostatné složky - *generate* a *evoinference*. První z nich se věnuje generování řetězců podle zadaného automatu do trénovacích sad S_+ a S_-. Druhá se skládá z genetického algoritmu a samotné inferenci.

6.1 Generování řetězců

Generování řetězců probíhá pomocí programu *generate*. Jak již bylo řečeno dříve, trénovací sada se skládá ze dvou částí. První z nich je množina řetězců S_+ generované automatem M a druhou z nich je sada protipříkladů S_-, tedy těch, které M negeneruje. Nejdříve si popíšeme, jak probíhá generování řetězců $w \in L(M)$ a ve druhé části pak generování řetězců $w \notin L(M)$.

Sada S_+ se vytváří podle zadaného automatu, který se skládá z množiny stavů, abecedy, počátečního stavu, množiny koncových stavů a množiny přechodů. Po nastavení všech těchto složek automatu a zadání typu automatu (JFA, GJFA, doprava jednosměrně skákající) se následně začnou vytvářet řetězce.

Typ automatu se nastavuje pomocí parametru, se kterým může být program spuštěn. Těmito parametry jsou:

- bez argumentu - JFA
- 1 - GJFA
- 2 - doprava jednosměrný JFA

Nastavení automatu se určuje v souboru *generateparams.h*. Veškeré složky jsou brány jako makra, které se nebudou v průběhu programu měnit. Příklad nastavení pro GJFA lze vidět níže.

```
#define START 0
#define NUMBER_OF_STATES 3
#define FINISH {1}
#define ALPHABET {"a", "b", "ab"}
#define NUMBER_OF_ALPHABET 3
#define RULES {{0, 0, 0, 0, 0, 1, 0, 0, 0}, \
               {0, 0, 0, 0, 0, 0, 1, 0, 0}, \
               {0, 0, 0, 0, 1, 0, 0, 0, 0}}
```

Množina stavů je brána číselně a číslují se od nuly po n. Například stav q_0 je zde chápán jako 0, stav q_1 jako 1, atd. Zadává se počet stavů automatu uložené v proměnné $NUMBER_OF_STATES$. START udává počáteční stav a FINISH je množina koncových stavů. ALPHABET udává množinu pravidel, které jsou uloženy v datovém typu *string*. Tento typ byl vybrán, kvůli GJFA, který může číst část řetězce a ne jenom jeden symbol. Ačkoli abeceda formálního modelu GJFA neobsahuje řetězce, ale jsou dány až přechodovými pravidly, toto nastavení tyto řetězce vyžaduje. Ve dříve zmíněném příkladu lze vidět nastavení abecedy, kde automat může číst symboly a, b a řetězec ab. Poslední je množina pravidel. Protože se může jednat o nedeterministický automat, je množina pravidel definována jako dvourozměrné pole, ve kterém 0 značí neexistující přechod a 1 existující. Prvním parametrem je stav, ve kterém se momentálně nacházíme, a druhým je přechod pro daný stav a přijímaný symbol (řetězec).

Například nastavení pravidel pro JFA přijímající jazyk $L_1 = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$, který byl zmíněn v sekci 3 o skákajících automatech, lze reprezentovat pomocí následující tabulky:

Stav	$(stav, a) \to q_0$	$(stav, b) \to q_0$	$(stav, a) \to q_1$	$(stav, b) \to q_1$
q_0	0	0	1	0
q_1	0	1	0	0

Tabulka 6.1: Příklad přechodové tabulky

Řetězce se ukládají do datového typu *string*, díky jednoduché práci s tímto datovým typem. Postupně se začnou tvořit řetězce pomocí simulace automatu M. Začne se v počátečním stavu a v poli pravidel se náhodně vybere existující přechod (ten s hodnotou 1) z tohohle stavu. Výběr náhodně existujícího přechodu je děláno pomocí pole, které obsahuje indexy všech možných přechodů z aktuálního stavu. Hodnoty tohohle pole se náhodně zamíchají pomocí funkce *shuffle* a pak se systematicky procházejí všechny tyto náhodně seřazené indexy v tomto poli, dokud se nenarazí na index, jehož hodnota v poli pravidel je 1. Neboli pro stav q_0 se vezme odpovídající řádek v množině přechodů, ten se náhodně zamíchá a pak se systematicky zleva doprava vybírá existující přechod. Dle tohohle indexu a adekvátní hodnoty pole pravidel se na začátek řetězce *word* typu *string* uloží znak. U GJFA může být těchto znaků v jednom přechodu více. Pak se podle pole pravidel přepíše aktuální stav a stejným způsobem se přidávají další znaky do řetězce. Ty se ale nepřidávají na začátek řetězce, ale na libovolné místo tohohle řetězce. Takhle se generují znaky řetězce, dokud aktuální stav není koncovým stavem.

Automaty přijímají řetězce od počátečního po koncový stav, ale nemusí skončit, jakmile dorazí do koncového stavu. Mohou dále přijímat a skončit až do něj dorazí například podruhé, potřetí, po sté atd. Nám stačí mít jen omezenou sadu, proto skončíme generování hned, co dorazíme do koncového stavu. Takovým způsobem vygenerujeme pár řetězců. V dalším kroku generujeme další řetězce, ale místo, abychom v koncovým stavu skončili poprvé, co do něho automat vstoupí, tak bude pokračovat simulace a bude tím vytvářet nové a delší řetězce. Například pro jazyk L_1, který byl zmíněn výše, se tímhle způsobem vygenerují navíc řetězce jako *abab, abba, bbaa* a ne jenom *ab, ba.* Tímto způsobem se zvýší počet různě vygenerovaných řetězců. Počet takto vygenerovaných řetězců je dán hodnotou makra *WORDS_GENERATE*, která se nachází v souboru *generateparams.h.* V našem případě je tato hodnota nastavena na hodnotu 15.

Generování řetězců pro GJFA nemůže fungovat dle stejných pravidel jako u JFA. Nejdříve musíme otočit přechodová pravidla automatu a prohodit počáteční a koncové stavy. Jeden z koncových stavů se stane počátečním stavem a počáteční stav se stane koncovým. Poté je potřeba otočit pole přechodů, kde výchozí stav se stane cílovým a cílový výchozím stavem. Například pokud byl zadán přechod $(q_0, a) \to q_1$, v otočeném případě se tento přechod stane $(q_1, a) \to q_0$. U dříve zmíněného příkladu generování řetězců pro JFA, bude celková tabulka vypadat následovně:

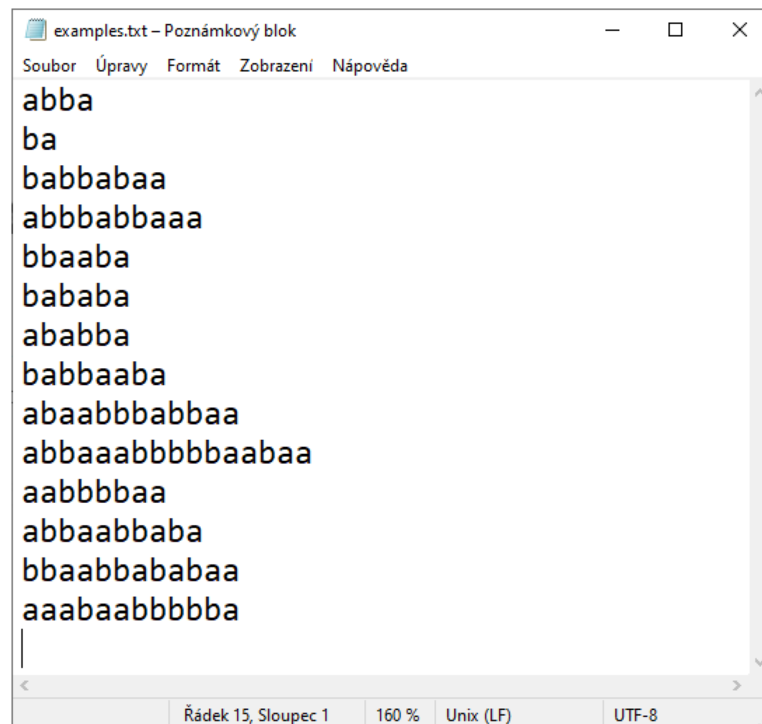
Stav	$(stav, a) \to q_0$	$(stav, b) \to q_0$	$(stav, a) \to q_1$	$(stav, b) \to q_1$
q_0	0	0	0	1
q_1	1	0	0	0

Tabulka 6.2: Příklad přechodové tabulky

Samotné generování pak funguje stejným způsobem jako se generují řetězce pro JFA. Každý takhle vygenerovaný řetězec se uloží do složky *examples* do souboru *examples.txt*. Kvůli vybírání náhodných přechodů, existuje možnost, že se nějaký řetězec vygeneruje vícekrát. Takový řetězec se pak do souboru neukládá.

Protipříklady se generují podobným způsobem a ty se pak ukládají do složky *examples* do souboru *contra_examples.txt* za předpokladu, že se již v souboru nenachází. Ukládání řetězců do souboru bylo zvoleno z důvodu ručního přidávání a úprav řetězců.

Všechny řetězce v souborech *examples.txt* i *contra_examples.txt* jsou od sebe odděleny novým řádkem. Příklad struktury ukládání obsahu souboru *examples.txt* pro JFA přijímající jazyk L_1 lze vidět na obrázku 6.1.



Obrázek 6.1: Ukázka souboru *examples.txt*

Kromě JFA a GJFA lze generovat i řetězce jejich doprava jednosměrných variant. Těm se nehýbe jejich čtecí hlava náhodným směrem o náhodný počet znaků, ale mohou se hýbat jenom doprava, pokud nemají možnost přijmout znak či řetězec u GJFA. Nejdříve se simuluje průchod automatem a do datové struktury *vector* se vkládají veškeré symboly a řetězce. Po simulaci se tyto hodnoty zamíchají algoritmem *shuffle*, který se nachází v knihovně *algorithm*, a výsledek se uloží do datové struktury *string*. Řetězec je pak otestován na členství a v případě, že automat řetězec přijímá, uloží se do souboru *examples.txt*.

Protipříklady jsou generovány obdobným způsobem a ukládají se do souboru *contra_examples.txt*.

Kromě trénovacích sad se generují i testovací sady. Ty se generují stejným způsobem a ukládají se do souborů *test_examples.txt* a *test_contra_examples.txt*. Řetězce se generují ve stejnou dobu jako řetězce trénovací sady a obsahují stejný počet jako tato sada.

6.2 Testování členství řetězců do jazyka

Fungování testování členství řetězců bylo popsáno v kapitole 5, kde se píše, že se tato problematika řeší odlišně pro JFA, zobecněné skákající automaty i doprava skákající automaty. Proto si nejdříve popíšeme JFA, pak GJFA a nakonec doprava skákající automaty.

JFA

U JFA je potřeba si nejdříve spočítat četnost jednotlivých znaků a ty si následně uložit do pole, případně vektoru. Toto pole se nazývá *occur*. Pokud si optimálním algoritmem seřadíme posloupnost znaků v řetězci, lze pak na jeden průchod spočítat jednotlivé četnosti. Pro seřazení zde byla použita knihovní funkce *sort*, která se nachází v knihovně *algorithm*. Seřazenou posloupnost pak projdeme zleva doprava a přičítáme jedničku za každý výskyt daného znaku. Pro usnadnění je zde ještě proměnná *len*, obsahující délku řetězce. Je-li tato proměnná na konci testování nulová a nacházíme se v koncovém stavu, řetězec patří do jazyka L popsané automatem M.

Testování je řešeno rekurzí, protože jinak bychom si museli ukládat (nejlépe na zásobník) všechny předchozí stavy a předchozí hodnoty pole *occur*. Neboli, jak vypadal vektor četností a automat před aplikací přechodu. Funkce obsahuje parametry udávající aktuální stav, aktuální délku řetězce *len* a aktuální hodnoty pole *occur*. V ní se hledá postupně existující přechod a pokud se daný znak ještě nachází v řetězci, pak se počítadlo tohohle znaku v poli *occur* sníží, sníží se hodnota *len*, posune aktuální stav a znova se zavolá tato funkce s těmito parametry. Při vynoření se obnoví stav pole *occur*, hodnoty *len* a aktuálního stavu. Funkce končí, pokud se *len* bude rovnat nule a nacházíme se v koncovém stavu. V takovém případě vrátí funkce nulu, což znamená, že automat řetězec přijímá. Pokud se *len* rovná nule, ale nenacházíme se v koncovém stavu, funkce vrátí -1, vynoří se a hledají se jiné možnosti. Pokud se projdou všechny přechody v aktuálním stavu, ale nenašli jsme jediný validní přechod, funkce vrátí -1.

Zobecněné skákající automaty

Bohužel, jak již bylo zmíněno v kapitole 5 o testování řetězců GJFA, tento způsob nelze použít pro GJFA, kvůli tomu, že dokáže číst více symbolů naráz v jednom přechodu. Musíme proto projít všechny možné způsoby přijmutí řetězce GJFA. To budeme dělat systematicky zleva doprava pomocí způsobu *backtracking*. Vždy vezmeme jeden symbol (případně mno-

žinu symbolů) a pokud bude existovat přechod, tak ho z řetězce odstraníme. Kvůli tomu je vhodné si řetězce uložit do datového typu, ze kterého lze snadno odstraňovat symboly. Nejvýhodnějším datovým typem s touto vlastností je datový typ *string*, díky jeho implementovaným funkcím a jednoduché práci s řetězci. Další výhodou je hledání podřetězců v řetězci, kvůli existenci přechodů o více znacích naráz.

Iteruje se řetězcem zleva doprava a hledají se existující přechody pro daný symbol či symboly (podřetězce) tohohle řetězce. Kvůli porovnávaní podřetězců byla použita funkce *compare*, která se nachází v knihovně pro práci s datovým typem *string*. Pokud daný přechod existuje, symbol či symboly se z řetězce odstraní a funkce se volá znova, dokud nevrátí prázdný řetězec či nebude moct dál pokračovat (nebude existovat žádný přechod pro zbylé symboly řetězce). Pokud nebude moct dál pokračovat, vynoří se z funkce, obnoví aktuální stav, stav řetězce před smazáním symbolu či symbolů a zkouší jiné existující přechody. Nenarazí-li na jiný existující přechod, posune se na další symbol a pokračuje v testování. Funkce vrací 0 v případě členství řetězce do jazyka a -1 pokud řetězec do jazyka nepatří.

Doprava jednosměrně skákající automaty

Posledním typem automatů jsou doprava jednosměrně skákajících automaty. U nich se členství zjišťuje nejjednodušeji. Testování probíhá podobným způsobem jako u klasických neskákajících automatů. Řetězec je též uložen v datovém typu *string* jako u předchozích variant. Během testování se iteruje zleva doprava přes celý řetězec a aplikují se pravidla automatu, pokud je možné nějaký z nich použít. V takovém případě se znak z řetězce smaže a pokračuje se dalším. Pokud neexistuje přechod, symbol se přeskočí a čtecí hlava automatu se posune doprava na další symbol. Pokud automat došel na konec řetězce a řetězec nebyl celý smazán, posune svou čtecí hlavu na začátek řetězce a pokračuje ve čtení od tohohle místa. Pokud byl smazán celý řetězec a automat se nachází v jednom z koncových stavů, automat řetězec přijal a funkce vrací hodnotu 0. Pokud nebyl uplatněn jediný přechod pro všechny symboly řetězce (či zbylého řetězce), automat řetězec nepřijal a vrací hodnotu -1.

6.3 Genetický algoritmus

Genetický algoritmus se nachází v programu *evoinference* a jeho základní struktura byla převzata ze stránek www.geeksforgeeks.org[1], která je psána v jazyce C++. Tento příklad se snaží vygenerovat řetězec *TARGET*, který byl zadán jako konstanta v datovém typu *string*, pomocí genetických algoritmů. Tím pádem se chromozom skládá z množiny genů, které mohou nabývat libovolného znaku z konstanty *GENE*, která udává, jakých hodnot může 1 gen nabývat. Zde jsou to všechny znaky a symboly anglické abecedy. Samotný jedinec (chromozom) populace je reprezentován jako třída *Individual*. Proměnné a metody, která tato třída obsahuje, lze vidět níže.

```
class Individual
{
    public:
        string chromosome;
        int fitness;
        Individual(string chromosome);
        Individual mate(Individual parent2);
```

[1] https://www.geeksforgeeks.org/genetic-algorithms/

```
        int cal_fitness();
};
```

Proměnná *chromosome* obsahuje hodnotu chromozomu a proměnná *fitness* ohodnocení tohohle jedince pomocí fitness funkce. Funkce *mate()* kříží tohohle jedince s jedincem *parent2* a vrací nového jedince. Poslední funkcí je funkce *cal_fitness()*, která provádí ohodnocení jedince fitness funkcí a vrací toto ohodnocení. Zde je to počet symbolů, které se shodují v chromozomu a hledaném řetězci.

Na začátku se vytvoří tolik takovýchto jedinců, kolik je nastaveno v makru *POPULATION_SIZE* udávající počet jedinců v populaci. Všichni jedinci budou vytvořeni náhodně a budou obsahovat náhodné hodnoty z konstanty *GENE*. K výběru náhodné hodnoty je zde použita funkce mutace, která vybírá náhodnou hodnotu z konstanty *GENE*. Poté se spočítá fitness hodnota každého jedince a zjistí se, zda není nějaký jedinec ten, kterého hledáme. Pokud ano, algoritmus skončí a vrátí daného jedince. Pokud ne, pokračuje dle ostatních kroků genetického algoritmu a neskončí dokud se nenalezne hledaný řetězec *TARGET*. V tomhle konkrétním algoritmu je použit 10% elitismus. To znamená, že 10 % nejlepších jedinců z populace se dostane do nové. Nových 90 % vznikne křížením poloviny nejlepších jedinců, přičemž se vždy vybírají oba rodiče náhodně. V tomhle příkladě není aplikována ruleta ani turnaj a je zde použito jednobodové křížení, které říká, že s pravděpodobností 45 % bude mít nový jedinec buď gen z rodiče 1 či z rodiče 2. Zbylých 10 % říká, že tento gen nevznikne ani z jednoho rodiče, ale bude mutovat. Mutování bylo zmíněno dříve v této kapitole a nastavuje gen náhodně podle hodnot konstanty *GENE*.

Takto vypadá základní převzatá struktura genetického algoritmu z dříve zmíněných stránek. Aby ho bylo možné použít pro inferenci skákajících modelů, musela být upravena a doplněna o další funkce. Kvůli tomu byl změněn chromozom, aby odpovídal našemu problému. Chromozom je brán jako řetězec 0 a 1, kde každý bit znamená buď existující nebo neexistující přechod $py \to q$. 1 znamená, že daný přechod existuje a 0, že ne. Délka chromozomu je dána nastavením hodnot *NUMBER_OF_STATES* a *NUMBER_OF_ALPHABETS*. *NUMBER_OF_STATES* udává počet stavů, které má mít hledaný automat, a *NUMBER_OF_ALPHABETS* udává počet znaků abecedy, který tento automat má obsahovat. Pokud hledáme zobecněný skákající automat (GJFA), veškeré podřetězce, které se mají přečíst v jednom přechodu, jsou též brány jako znak abecedy. Například, pokud pomocí genetických algoritmů hledáme automat, který může číst a, b i ab (tedy oba znaky v jednom kroku), pak hodnota v *NUMBER_OF_ALPHABETS* bude 3. Jak *NUMBER_OF_STATES*, tak *NUMBER_OF_ALPHABETS* jsou brány jako makra a nachází se v souboru *evoinference.h*. Délka chromozomu je pak

$$|ch| = states \cdot alphabets \cdot states,$$

kde *states* odpovídá hodnotě *NUMBER_OF_STATES* a *alphabets* odpovídá *NUMBER_OF_ALPHABETS*.

Další změna byla provedena u fitness funkce. V sekci 5 byly probrány 2 způsoby řešení této funkce a v práci byly vyzkoušeny oba tyto způsoby. Oba způsoby se volají a počítají pro každý chromozom po vytvoření (náhodně na začátku a po křížení a mutaci). Tato funkce se jmenuje *cal_fitness()* a výsledek se ukládá do proměnné *fitness*.

První způsob počítá počet řetězců, které patří a mají patřit do jazyka popsaného automatem a současně i ty, které nepatří a nemají patřit do tohohle jazyka popsaného tímto automatem. K tomu je využita funkce pro zjišťování členství řetězců do jazyka podle zadaného automatu. Typ automatu se zadává pomocí argumentů při spuštění programu *evoinference*

stejně jako u programu *generate*. Program lze spustit bez argumentu nebo s argumenty 1 a 2. Význam těchto argumentů je stejný jako u programu *generate*. To je: bez - JFA, 1 - GJFA a 2 - doprava jednosměrný JFA. Podle zadaného argumentu se vybere jiná metoda testování členství řetězců do jazyka při zjišťování fitness funkce.

Druhá způsob tuto fitness upravuje, aby zvýhodňovala ty jedince, kteří přijímají větší část řetězce a znevýhodňuje ty, které obsahují všechny či žádný přechod. Nejjednodušší by bylo volat funkce, které byly zmíněny v sekci 5 a vypočítat ji podle nich. Ale místo volání těchto funkcí, se fitness hodnota zjišťuje ve funkce, která testuje členství řetězce do jazyka, která vrací, zda řetězec do jazyka patří či nepatří. Ta mimo jiné zjistí i délku nejdelšího přečteného řetězce a tuto hodnotu vrátí v proměnné *numb_of_acc_letters*. Poté se fitness hodnota spočítá podle vzorců $accepted(M, S_+)$ a $f_2(M, S_+)$, které byly zmíněny v sekci 5. Pro vybrání této fitness funkce lze program spustit s parametrem $-f$.

Na začátku GA náhodně vznikne několik chromozomů (počet je dán konstantou *POPULATION_SIZE*, která má stejný význam jako u příkladu pro vygenerování konkrétního řetězce), tedy velikost populace. Geny chromozomu nabývají hodnot 0 nebo 1 a na začátku GA se náhodně vybere, zda ten gen bude 0 či 1. Průběh GA zůstal nezměněn, jen mutace a výběr jedinců byly změněny. Pokud má daný gen mutovat, tak se náhodně změní na 0 či 1, přičemž i při mutaci genu se může stát, že daný gen zůstane nezměněn. Výběr jedinců pro křížení byl změněn z náhodného výběru na turnajový, protože z předchozích zkušeností se turnajový výběr více osvědčil a ukázal se jako víc spolehlivější. Tento výběr byl inspirován příkladem genetického algoritmu v předmětu Aplikované evoluční algoritmy[2] (EVO), kde byl odzkoušen a kde se ukázal jako velice praktický a spolehlivý způsob výběru jedinců. Ten vybere čtyři náhodné jedince, porovná jejích fitness hodnoty a pro křížení vybere ty nejlepší z nich.

Výstup programu je psán na standardní výstup a vypisuje průběžně nejlepšího jedince v každé generaci ve formátu:

Generation: xxx   Chromozom: xxxxxx   Fitness: x

Například:

Generation: 1   Chromozom: 110110   Fitness: 0.5
Generation: 2   Chromozom: 110100   Fitness: 0.6
Generation: 3   Chromozom: 110100   Fitness: 0.6

Nejlepší jedinec úplně poslední generace je pak vrácen ve formátu:

Generation: xxx   Fitness: x
Chromozom:
xxx xxx
xxx xxx

Například:

Generation: 10   Fitness: 1.0
Chromozom:
00 10
01 00

[2] https://www.fit.vut.cz/study/course/EVO/

Tento přepis odpovídá již dříve zmíněné tabulce, kterou lze chromozom vyjádřit. Řádky a sloupce oddělené mezerou udávají stavy od 0 po n a samotná čísla ve sloupcích přechod pro daný symbol abecedy podle nastavení abecedy $ALPHABET$.

Kapitola 7

Experimenty

Experimenty byly provedeny pro všechny typy dříve zmíněných automatů, přičemž pro každý tento typ bylo otestováno několik druhů automatů přijímající odlišné jazyky. Pro každý tento automat byly vyzkoušeny obě dříve zmíněné fitness funkce. Cílem bylo zjistit, zda bude genetický algoritmus dávat adekvátní výsledky pro některou ze zmíněných fitness funkcí a případně, která z nich vede genetický algoritmus dříve k hledanému výsledku. Testy byly rozděleny podle druhu automatu a pro každý automat bylo provedeno 50 běhů programu.

V následující tabulce jsou ukázány konkrétní automaty, pro které se dělaly testy.

Typ automat	automat
JFA	$M_1 = (\{q_0, q_1\}, \{a, b\}, \{q_0a \to q_1, q_1b \to q_0\}, q_0, \{q_0\})$
JFA	$M_2 = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{q_0a \to q_0, q_0b \to q_1, q_0c \to q_2, q_1b \to q_1, q_2c \to q_2\}, q_0, \{q_1, q_2\})$
JFA	$M_3 = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{q_0a \to q_0, q_0b \to q_1, q_1b \to q_1, q_1c \to q_2, q_2c \to q_2\}, q_0, \{q_2\})$
JFA	$M_4 = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{q_0a \to q_1, q_1b \to q_2, q_2c \to q_0\}, q_0, \{q_0\})$
JFA	$M_5 = (\{q_0, q_1, q_2, q_3\}, \{a, b, c, d\}, \{q_0a \to q_1, q_1b \to q_2, q_2c \to q_3, q_3d \to q_0\}, q_0, \{q_0\})$
GJFA	$M_6 = (\{q_0, q_1\}, \{a, b\}, \{q_0ab \to q_1, q_1a \to q_1, q_1b \to q_1\}, q_0, \{q_1\})$
GJFA	$M_7 = (\{q_0, q_1, q_2\}, \{a, b, c, d, e, f\}, \{q_0ab \to q_1, q_0cd \to q_2, q_1e \to q_1, q_2f \to q_2\}, q_0, \{q_1, q_2\})$
GJFA	$M_8 = (\{q_0, q_1, q_2\}, \{a, b, c, d, e\}, \{q_0ab \to q_1, q_1e \to q_1, q_1cd \to q_2\}, q_0, \{q_2\})$
one-way JFA	$M_9 = M_1$
one-way JFA	$M_{10} = M_2$
one-way JFA	$M_{11} = M_3$
one-way JFA	$M_{12} = M_4$
one-way JFA	$M_{13} = M_5$
one-way GJFA	$M_{14} = (\{q_0, q_1\}, \{a, b, c, d\}, \{q_0ab \to q_1, q_1cd \to q_1\}, q_0, \{q_1\})$
one-way GJFA	$M_{15} = M_7$
one-way GJFA	$M_{16} = (\{q_0, q_1, q_2\}, \{a, b, c, d, e\}, \{q_0ab \to q_1, q_1e \to q_1, q_1cd \to q_2, q_2e \to q_2\}, q_0, \{q_2\})$

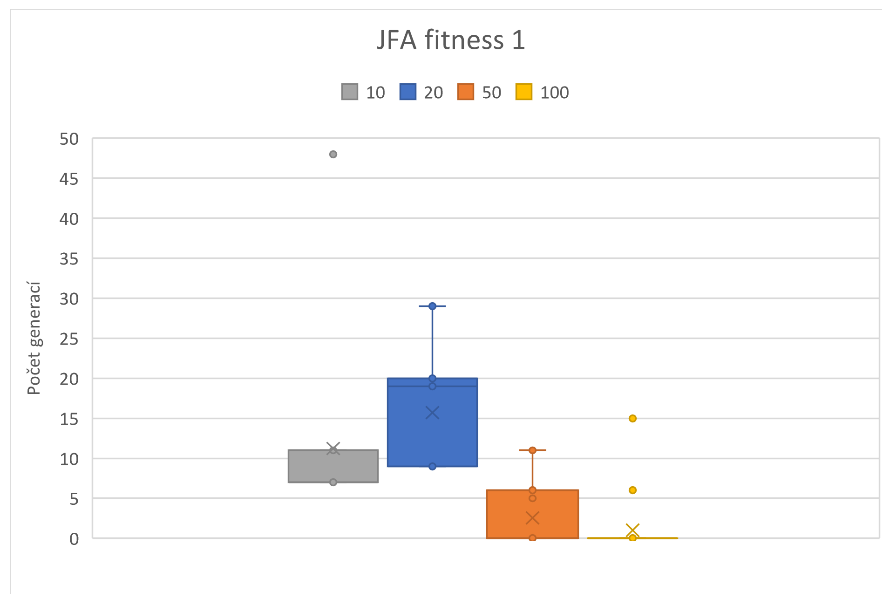
Tabulka 7.1: Přehled testovaných automatů

Maximální počet generací byl zvolen 200, pravděpodobnost mutace byla nastavena na hodnotu 0,1. První experiment testuje vliv velikosti populace na nalezení řešení. Ta byla zvolena 10, 20, 50 a 100. Veškeré ostatní experimenty nastavují velikost populace na hodnotu 100. Dále byl vyzkoušen vliv počtu stavů na hledané řešení a na algoritmus, protože existuje nekonečně mnoho automatů přijímající stejný jazyk s rozdílným počtem stavů. Například JFA přijímající jazyk $L_1 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$, který byl zmíněn v sekci 3 obsahoval dva stavy. Kromě hledání automatu o dvou stavech přijímající stejný jazyk, byl vyzkoušen i genetický algoritmus hledající automat o více stavech, který též přijímá jazyk L_1. Pro tento automat se testoval počet stavů 2, 3 a 4. Testovaná abeceda zůstala podle definice námi hledaného automatu.

7.1 JFA

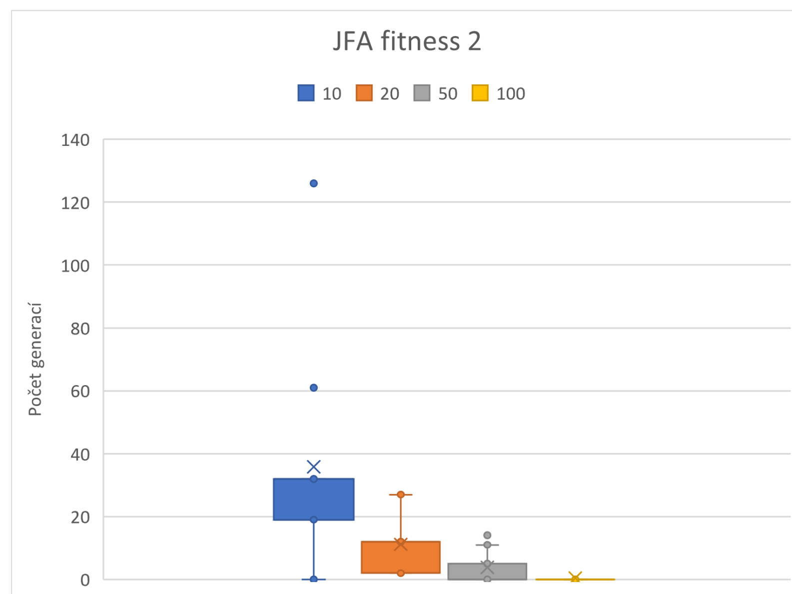
První experiment byl proveden pro JFA přijímající jazyk $L_1 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$. Tento automat byl zmíněn v sekci 3 jako příklad skákajícího konečného automatu. Nastavení bylo stejné jako bylo na začátku této kapitoly a byl hledán automat o dvou stavech. Pro tento automat a toto nastavení bylo ve 100 % případů nalezeno řešení, neboli byl nalezen jedinec s fitness hodnotou 1. Všichni jedinci navíc dokázaly správně rozlišit řetězce i z testovací sady a i tato úspěšnost byla ve všech případech 100 %.

Graf 7.1 zobrazuje počet generací potřebných pro nalezení řešení. Pro zobrazení byl použit tzv. krabicový graf. X značí střední hodnotu. Krabice (barevný obdélník) zobrazuje hodnoty v rozmezí prvního a třetího kvartilu. Hodnoty mimo krabici jsou hodnoty mimo tyto kvartily. Z grafu lze vidět, že až na pár výjimek, byl požadovaný počet generací pro nalezení řešení za použití fitness funkce 1 pod hodnotou 30. Pro nastavení počtu populace 20 dával algoritmus nejhorší výsledky. To je způsobeno náhodnými prvky při tvorbě počáteční populace. Můžeme ale vidět, že při zvedajícímu se počtu jedinců v populaci, algoritmus vyžaduje méně generací pro nalezení námi hledaného řešení. Pro populaci 100 je tento jedinec nalezen ve většině případů během první generace.



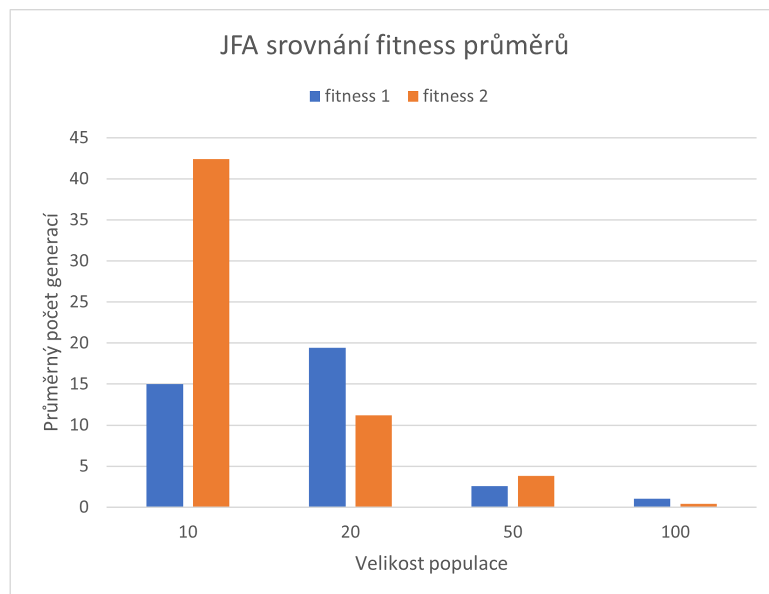
Obrázek 7.1: Výsledky dle počtu generací pro automat M_1 za použití první fitness funkce

Druhá fitness funkce při stejném nastavení dávala obdobné výsledky jako fitness funkce 1. Toho si můžeme všimnout na obrázku 7.2. Se zvedajícím se počtem populace se řešení nacházejí rychleji. Pro velikost populace o deseti jedincích konvergovala tato fitness funkce o dost pomaleji, než fitness funkce 1.



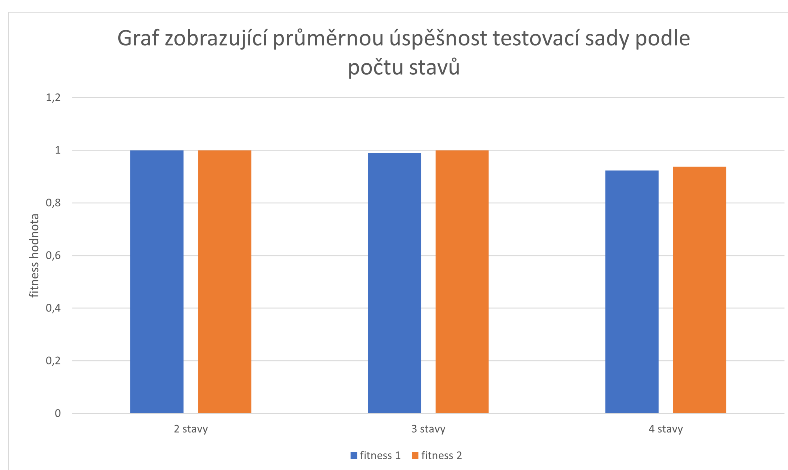
Obrázek 7.2: Výsledky dle počtu generací pro automat M_1 za použití druhé fitness funkce

Dále srovnáme, kolik generací potřebovaly obě fitness funkce k nalezení řešení. Počet generací je brán jako průměr generací pro všechny běhy programu s daným nastavením. Graf srovnání těchto fitness funkcí lze vidět na grafu 7.3. Můžeme si všimnout, že ve většině případů pro dané nastavení parametrů fungovala fitness funkce 1 lépe než fitness funkce 2. Obě ale dokázaly najít hledané řešení, které dokázalo správně přijmout a odmítnout řetězce z trénovacích i testovacích sad, než byl dosažen maximální počet generací, a to ve skoro 100 % případů.



Obrázek 7.3: Porovnání obou fitness funkcí pro M_1

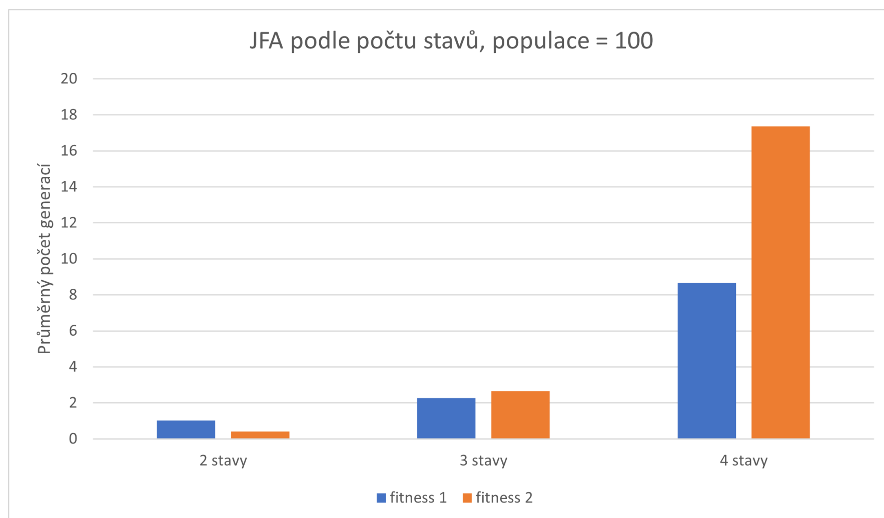
Poslední srovnání pro automat M_1, který přijímá dříve zmíněný jazyk L_1, bude podle počtu stavů automatu. Velikost populace byla nastavena na hodnotu 100 a počet stavů, který se testoval, byl nastaven na hodnoty 2, 3 a 4. Všechny běhy programu pro veškerá nastavení vrátila chromozom s fitness hodnotou 1, ale ne každý z nich dokázal správně rozlišit řetězce ze sad S_{test+} a S_{test-}. Graf zobrazující úspěšnost testovací sady lze vidět níže.



Obrázek 7.4: Porovnání úspěšnosti testovací sady pro M_1 podle počtu stavů

Z grafu 7.4 můžeme vyčíst, že pro automat o dvou či třech stavech dokázal výsledný automat správně rozlišit řetězce testovacích sad skoro ve 100 % případů. Naopak úspěšnost pro automat o 4 stavech klesá. Fitness funkce 2 vracela lepší výsledky při větším množství stavů, než fitness funkce 1, ale i tak pro obě fitness funkce úspěšnost s narůstajícím počtem stavů klesá.

Porovnáme-li počet potřebných generací, tak na grafu 7.5 můžeme vidět, že s narůstajícím počtem stavů se zvýší počet potřebných generací pro nalezení řešení pro obě fitness funkce. Fitness funkce 2 se neukázala tak praktická jako fitness funkce 1 pro automat o 4 stavech.

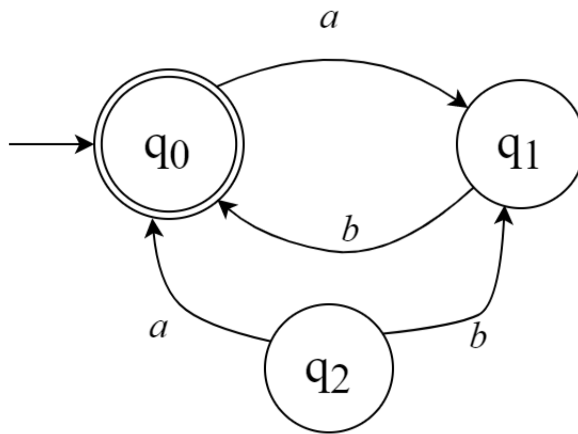


Obrázek 7.5: Porovnání obou fitness funkcí pro M_1 podle počtu stavů pro populaci 100

Z testů pro tento druh automatu pro jazyk L_1 vyplývá, že při větším počtu stavů se zvedá počet možností, které algoritmus musí (či zkusí) projít, a tím pádem se zvyšuje počet potřebných generací pro nalezení řešení. Je tedy možné, že pokud nastavíme maximální počet generací malý, při větším počtu stavů nemusí algoritmus najít požadované řešení.

Během těchto testů vznikaly často automaty, které byly ekvivalentní k námi zadanému automatu. Tedy automaty, které přijímaly stejný pod/jazyk. To je způsobeno tím, že neexistuje jenom jedna reprezentace automatu, který přijímá jeden jazyk. Například pro jazyk L_1 existuje, jak automat $M_1 = (\{q_0, q_1\}, \{a, b\}, \{q_0a \rightarrow q_1, q_1b \rightarrow q_0\}, q_0, \{q_0\})$, tak automat $M_{17} = (\{q_0, q_1\}, \{a, b\}, \{q_0b \rightarrow q_1, q_1a \rightarrow q_0\}, q_0, \{q_0\})$.

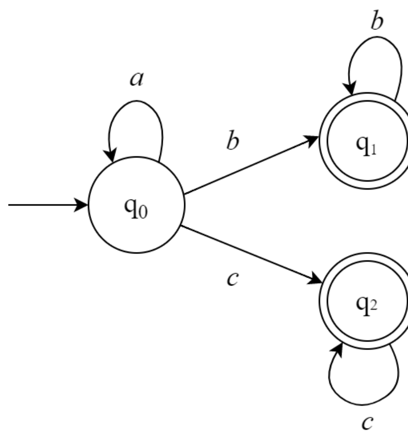
Dalšími ekvivalentními automaty, které během běhu algoritmu vznikly, jsou automaty obsahující nedosažitelné stavy. Tyto stavy nijak neovlivňují hledané řešení, protože během simulace se k nim automat nikdy nedostane. Jedním takovým automatem byl například automat $M_{18} = (\{q_0, q_1, q_2\}, \{a, b\}, \{q_0a \rightarrow q_1, q_1b \rightarrow q_0, q_2a \rightarrow q_0, q_2b \rightarrow q_1\}, q_0, \{q_0\})$. Grafické zobrazení lze vidět na obrázku 7.6. Tento automat byl výsledkem během jednoho běhu genetického algoritmu.



Obrázek 7.6: Ukázka automatu s nedosažitelným stavem

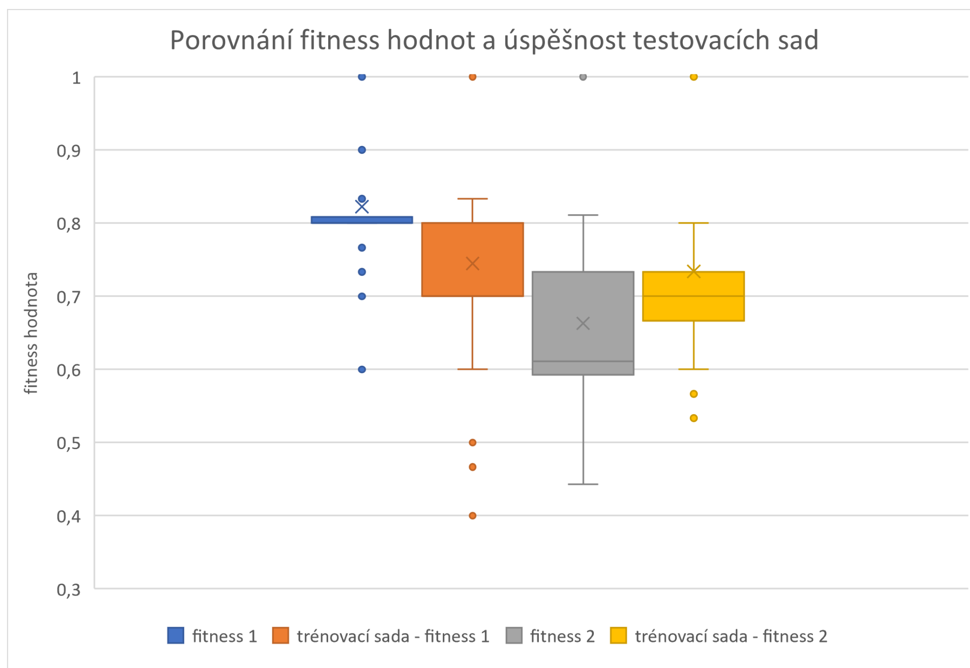
Kromě nedosažitelných stavů vznikaly i stavy, které nebyly koncové a nebylo možné se z nich dostat do koncového stavu. Takové stavy též neovlivňují výsledné řešení, ale jenom zpomalují testování členství řetězců do jazyka.

Druhým testovaným automatem typu JFA byl automat M_2, jehož grafické zobrazení lze vidět na obrázku 7.7. Nastavení parametrů bylo stejné jako u předchozího příkladu. Pravděpodobnost mutace byla nastavena na 0,1, maximální počet generací byl zvolen 200 a díky předchozím testům pro první typ byla testována jenom velikost populace 100, protože při nižší velikosti populace se hledaný chromozom nacházel později.



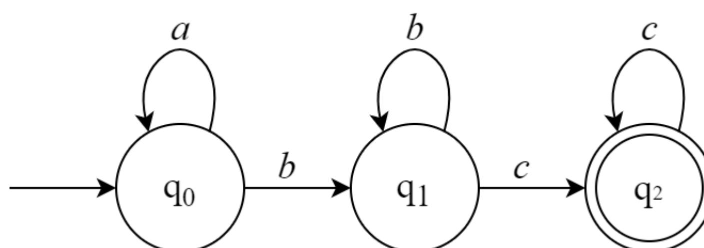
Obrázek 7.7: Grafické zobrazení automatu M_2

Na grafu 7.8 lze vidět výsledné fitness hodnoty 50 běhů programu a úspěšnost rozlišení řetězců v testovacích sadách. Oproti prvnímu experimentu, který skončil skoro ve 100 % případů úspěšným nalezením hledaného chromozomu, končil většinou tento experiment neúspěchem a to i při zvolené populaci 100, která je nejvyšší, kterou jsme testovali u automatu M_1. Pokud porovnáme obě fitness funkce, tak obě byly schopny aspoň jednou během 50 běhů nalézt požadované řešení a dokázalo nalézt i řešení, které správně přijalo/odmítlo řetězce z testovacích sad. I přes tento fakt skončila majorita testů neúspěchem.



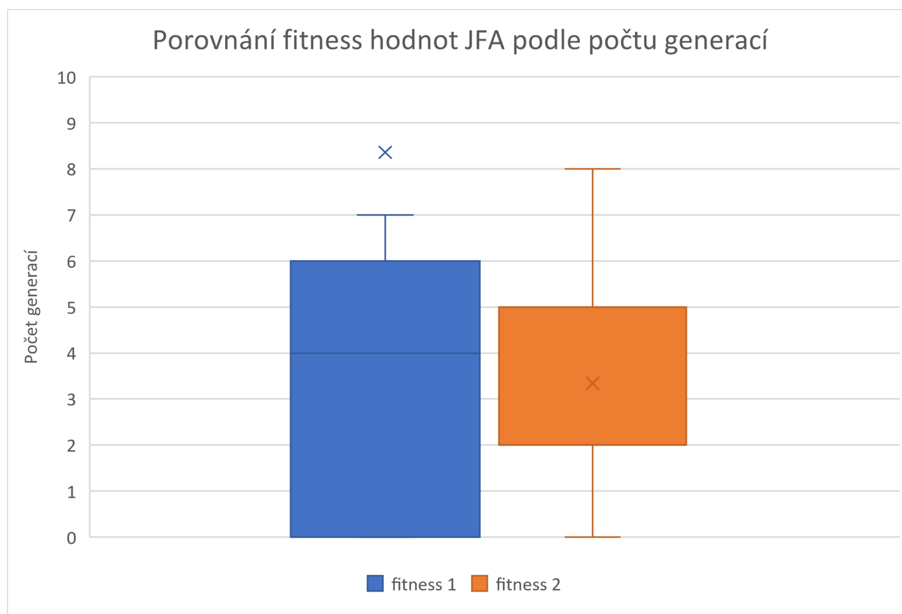
Obrázek 7.8: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_2

Třetím testovaným automatem typu JFA je automat M_3, jehož grafické zobrazení lze vidět na obrázku 7.9. M_3 přijímá jazyk obsahující libovolný počet symbolů a, aspoň jeden symbol b a aspoň jeden symbol c.



Obrázek 7.9: Grafické zobrazení automatu M_3

Testování proběhlo se stejnými parametry jako u automatu M_2 a výsledky běhů lze vidět na grafu 7.10. Tento graf zobrazuje počet populací, které byly potřeba, aby se našlo požadované řešení. Lze vidět, že obě fitness funkce byly schopny najít požadované řešení v rámci malého množství generací. Všech 50 běhů bylo schopno najít řešení během 9 generací a i úspěšnost pro testovací sady pro automat M_3 byla 100%.

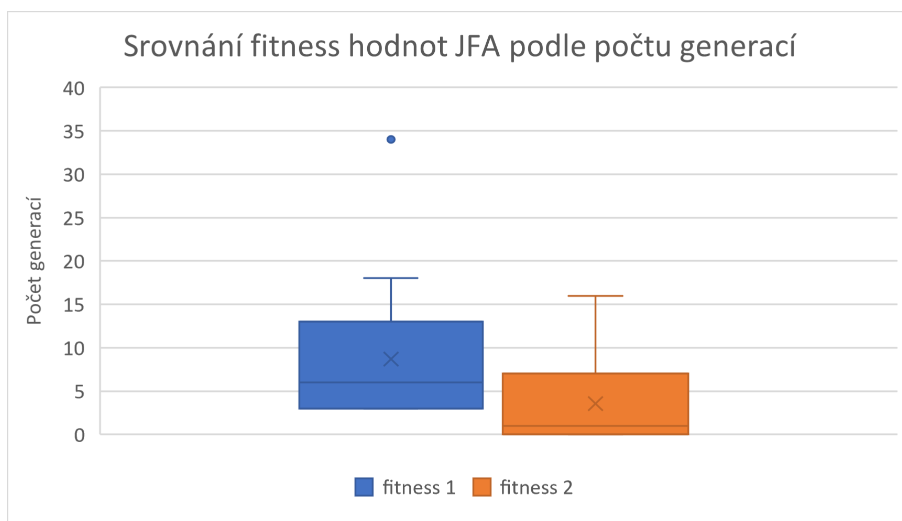


Obrázek 7.10: Graf zobrazující výsledné fitness hodnoty pro automat M_3

Čtvrtým testovaným automatem byl automat M_4, který přijímá jazyk:

$$L_2 = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}.$$

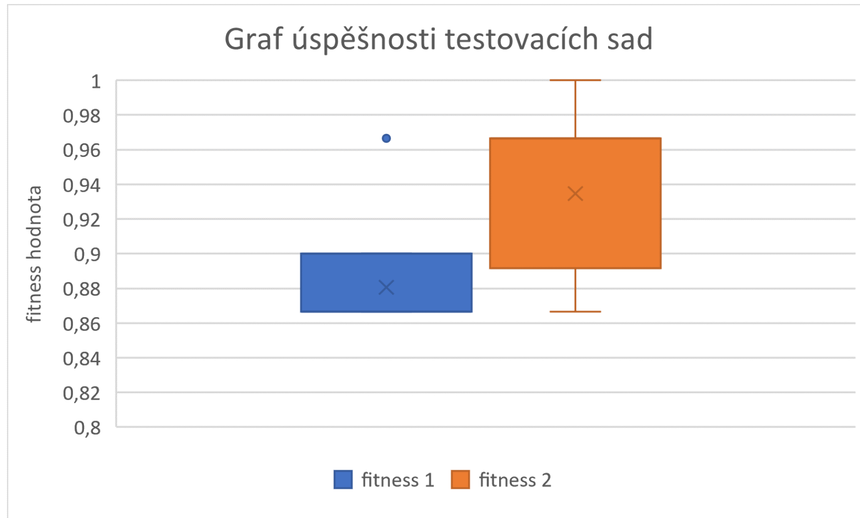
M_4 je podobný automatu M_1 z prvního experimentu, ale je rozšířen o další stav a symbol. Testy byly provedeny pro populaci velikosti 100 ze stejného důvodu jako u předchozího experimentu. Stejně jako u automatu M_1 byla úspěšnost z hlediska nalezení řešení 100%. Graf zobrazující počet potřebných generací je možno vidět níže.



Obrázek 7.11: Graf zobrazující výsledné fitness hodnoty pro automat M_4

Z hlediska testovacích sad nebyly výsledky stejné jako u automatu M_1. Toho si jde všimnout na grafu 7.12, který zobrazuje úspěšnost automatu správně určit řetězce testovacích

sad. Lze vidět, že tato úspěšnost byla nulová pro fitness funkce 1 a skoro nulová pro fitness funkci 2.

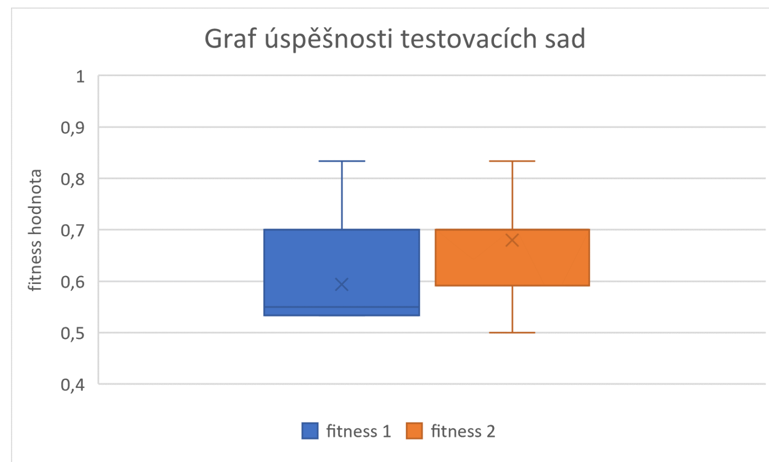


Obrázek 7.12: Graf úspěšnosti trénovacích sad pro automat M_4

Poslední experiment souvisí s experimenty M_1 a M_4, protože chce ukázat, zda záleží na počtu symbolů a počtu stavů automatu. Tento automat rozšiřuje automat M_4 o další symbol (d) a další stav. Poslední testovaný automat je tedy M_5, který přijímá jazyk:

$$L_3 = \{w \in \{a, b, c, d\}^* \mid |w|_a = |w|_b = |w|_c = |w|_d\}.$$

Tento experiment skončil vždy úspěšným nalezením hledaného jedince, ale stejně jako tomu bylo u automatu M_4, s trénovací sadou měl problémy. Graf 7.13 zobrazuje úspěšnost rozlišení testovacích sad. Ve srovnání s předchozím experimentem byla tato úspěšnost o dost nižší.



Obrázek 7.13: Graf zobrazující výsledné fitness hodnoty pro automat M_5

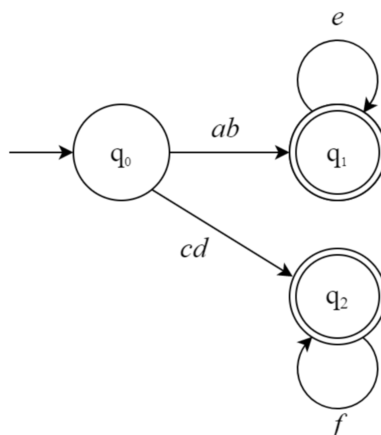
Bylo provedeno 5 různých experimentů pro automat typu JFA. Z nich vyplývá, že s narůstajícím počtem stavů se snižuje úspěšnost nalezení řešení pro maximální počet generací

(200) a též se snižuje schopnost odlišit jiné řetězce patřící do stejného jazyka. Z výsledků testů můžeme vidět, že největším problémem pro genetický algoritmus pro tento typ automatů bylo větvení, kdy automat má přejít buď do jednoho či druhého stavu. Tento typ v mnoha případech nenacházel ani pro jednu zvolenou fitness správné řešení.

7.2 Zobecněné skákající konečné automaty

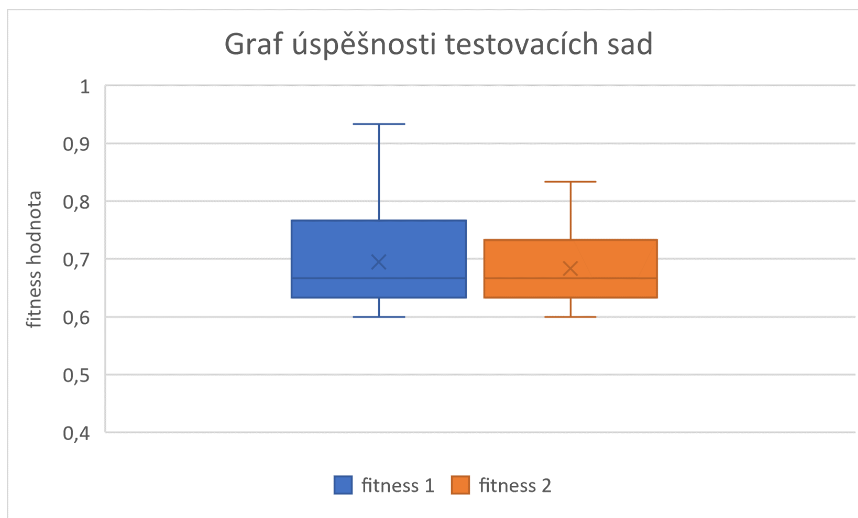
Další experimenty se týkají automatů GJFA. První byl proveden pro $M_6 = (\{q_0, q_1\}, \{a, b\}, \{q_0ab \rightarrow q_1, q_1a \rightarrow q_1, q_1b \rightarrow q_1\}, q_0, \{q_1\})$. M_6 čte řetězec ab a poté symboly a a b a nečte řetězce, které neobsahují aspoň jeden řetězec ab. Testy byly provedeny s nastavením parametrů, které byly zmíněny na začátku této kapitoly. Velikost populace byla zvolena 100 a byly testovány různé počty stavů. Všech 50 běhů dokázalo najít požadované řešení okamžitě v první generaci pro obě fitness funkce pro počet stavů 2 a 3. Čtyři stavy vyžadovaly o několik generací víc. Všechny testy vrátily jedince s fitness hodnotou 1 a všichni tito jedinci dokázali správně přijmout a odmítnout všechny řetězce testovacích sad. Úspěšnost pro tento automat byla tedy 100% v obou případech pro obě fitness funkce.

Druhým testovaným automatem byl automat M_9, jehož grafické zobrazení můžeme vidět na obrázku 7.14. Cílem bylo otestovat větvení, které tento automat obsahuje.



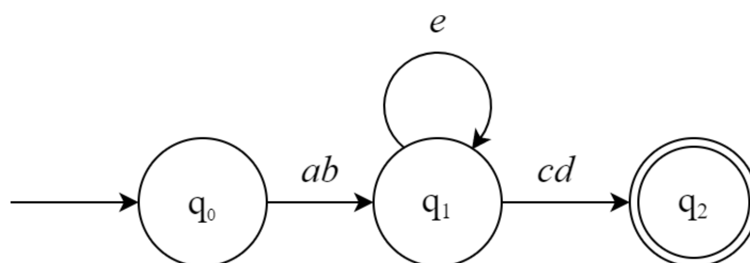
Obrázek 7.14: Grafické zobrazení automatu M_7

I tento automat skončil ve 100 % případů úspěšným nalezením jedince s fitness hodnotou 1 a to pro obě fitness funkce. Bohužel úspěšnost po testování na testovací sadě nebyla stejně úspěšně vysoká jako byla například pro automat M_6. Výsledky lze vidět na grafu 7.15, který zobrazuje úspěšnost automatu rozlišit řetězce testovacích sad. Lze si povšimnout, že obě fitness funkce vracely podobné automaty.



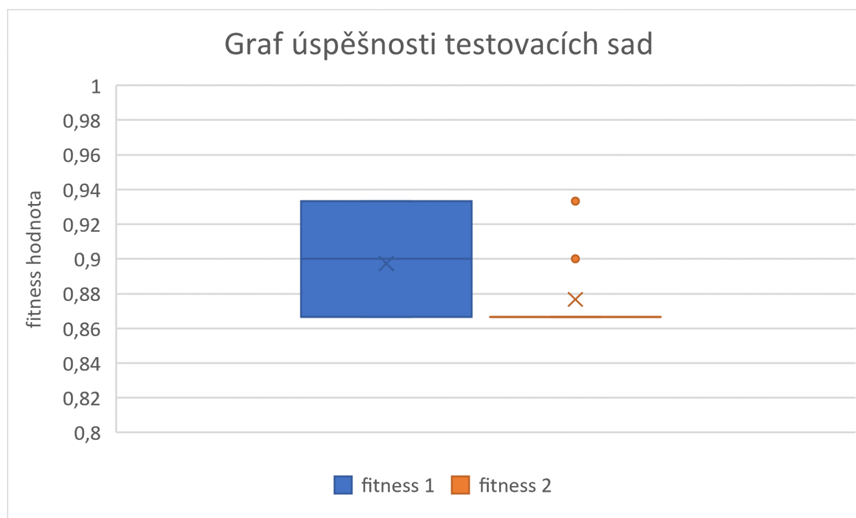
Obrázek 7.15: Graf zobrazující úspěšnost testovací sady pro M_7

Poslední automat, který se testoval, neobsahoval větvení, ale obsahoval 3 stavy. Jeho grafické zobrazení se nachází níže.



Obrázek 7.16: Grafické zobrazení automatu M_8

Stejně jako u předchozích dvou testů i tento test skončil v rámci fitness hodnot úspěšně pro všechny běhy programu. Algoritmus byl schopen najít jedince, jehož fitness hodnota byla 1 ve všech bězích programu. Následný test, zda tento jedinec dokáže správně přijmout a odmítnout řetězce testovacích sad, tak úspěšný nebyl. Toho si můžeme všimnout na grafu 7.17. Ani jeden výsledný jedinec nebyl schopen správně přijmout a odmítnout všechny řetězce testovacích sad, ale můžeme vidět, že fitness funkce 1 dokázala vytvořit úspěšnější jedince.



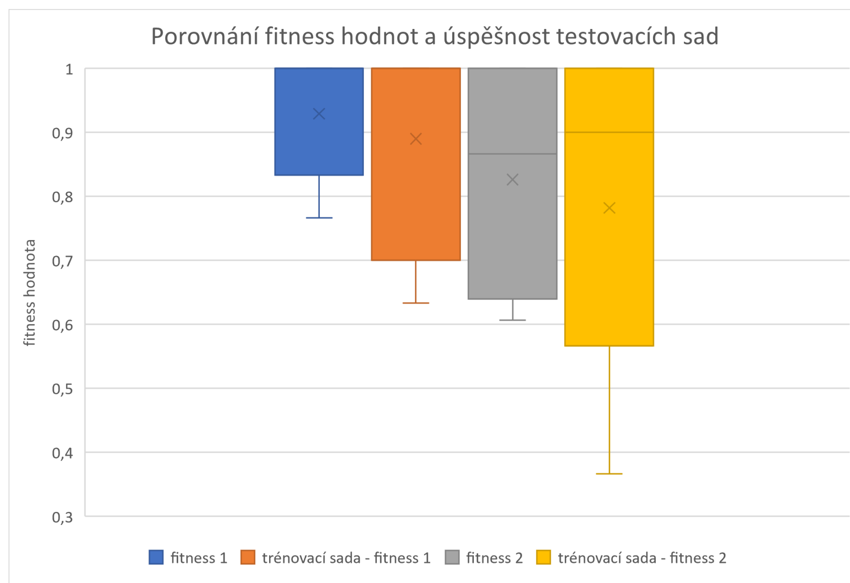
Obrázek 7.17: Graf zobrazující úspěšnost testovací sady pro M_8

Všechny tři testované automaty skončily úspěšným nalezením řešení, ale kromě automatu M_6 žádný z nich nedokázal správně přijmout a odmítnout i další řetězce námi hledaného jazyka.

7.3 Doprava jednosměrně skákající automat

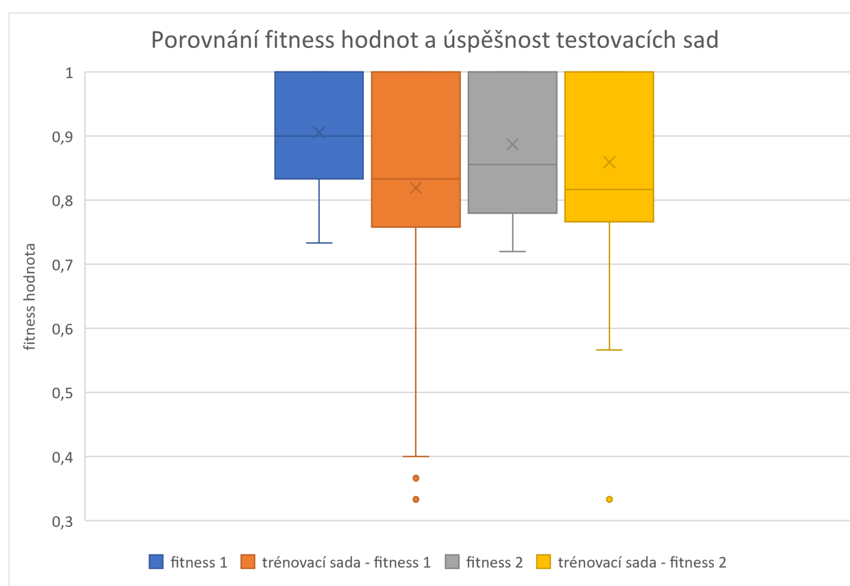
Kromě klasických automatů typu JFA se testovala i jejich doprava jednosměrně skákající varianta. Bylo provedeno 5 testů pro stejné automaty jako u JFA, ale s rozdílem, že se jednalo o tuto variantu. Přijímaný jazyk byl tedy jiný, než u automatů typu JFA a GJFA. První test byl proveden pro doprava jednosměrně skákající variantu automatu M_1 a stejně jako u JFA bylo testováno nastavení pro různý počet stavů automatu. Úspěšnost algoritmu a výsledného automatu po testování na testovacích sadách klesala se zvedajícím se počtem stavů. Pro dva stavy byla tato úspěšnost 100%. U třech stavů se řešení nenalezla ve třech případech pro fitness funkci 1. U fitness funkce 2 byla úspěšnost 100%. Naopak pro 4 stavy byla úspěšnost menší pro fitness funkce 2, kde 9 výsledných automatů nebylo schopno rozlišit všechny řetězce trénovacích ani testovacích sad. U fitness 1 byly tyto automaty jenom tři. Všechny automaty, které správně přijaly a odmítly všechny řetězce trénovacích sad, dokázaly i správně přijmout a odmítnout všechny řetězce testovacích sad. Úspěšnost těchto automatů byla 100%.

Druhým testovaným automatem byl automat M_{10}, který je doprava jednosměrně skákající variantou automatu M_2 a byla testována úspěšnost obou fitness. Tento automat obsahoval větvení. Z grafu níže lze vyčíst, že během 50 běhů bylo nalezeno několikrát řešení s fitness hodnotou 1. Jedinci s fitness hodnotou 1 navíc dokázali správně rozlišit řetězce testovacích sad. Pokud porovnáme obě fitness funkce, tak fitness 1 měla větší úspěšnost.



Obrázek 7.18: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_{10}

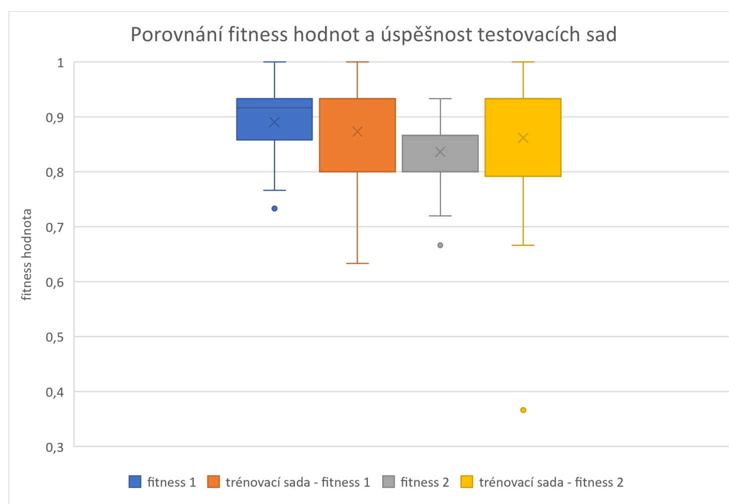
Třetí a čtvrtý test byl proveden pro automat se třemi stavy, stejně jako tomu bylo u JFA testů. Pokud se podíváme na grafy 7.19 a 7.20, můžeme vidět, že třetí test vykazoval větší úspěšnost jak pro trénovací sadu, tak pro testovací a to pro obě fitness funkce. Experimenty, které skončily s fitness funkcí 1 vracely i 100% úspěšnost pro řetězce testovací sady.



Obrázek 7.19: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_{11}

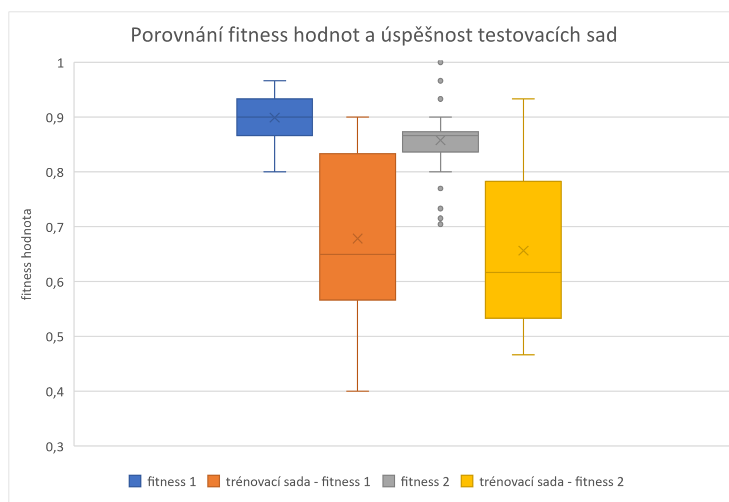
I když třetí i čtvrtý test obsahoval tři stavy, tak úspěšnost čtvrtého testu nebyla stejná jako u třetího. Výjimečně se našlo požadované řešení a ani tato řešení nedokázala správně rozlišit některé řetězce z testovací sady. Naopak některá řešení, která neměla fitness funkci

1, dokázala správně rozlišit všechny tyto řetězce. Toho si jde všimnout u fitness funkce 2, která nikdy nevrátila jedince s fitness hodnotou 1, ale i tak dokázal tento jedinec správně rozlišit další řetězce.



Obrázek 7.20: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_{12}

Poslední experiment byl proveden pro automat M_{13} obsahující 4 stavy. Porovnáme-li předchozí grafy s grafem výsledků tohohle automatu (graf 7.21), lze vidět, že s rostoucím množství stavů, končí algoritmus častěji neúspěchem. Během experimentů se výjimečně našel hledaný jedinec a ani ten nedokázal správně rozlišit jiné řetězce, které se nacházejí v testovacích sadách.

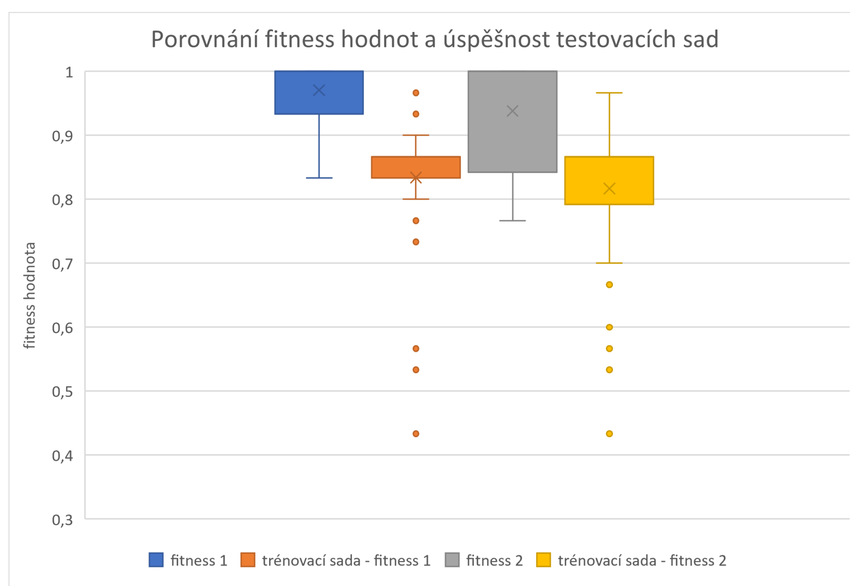


Obrázek 7.21: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_{13}

7.4 Doprava jednosměrně zobecněný skákající automat

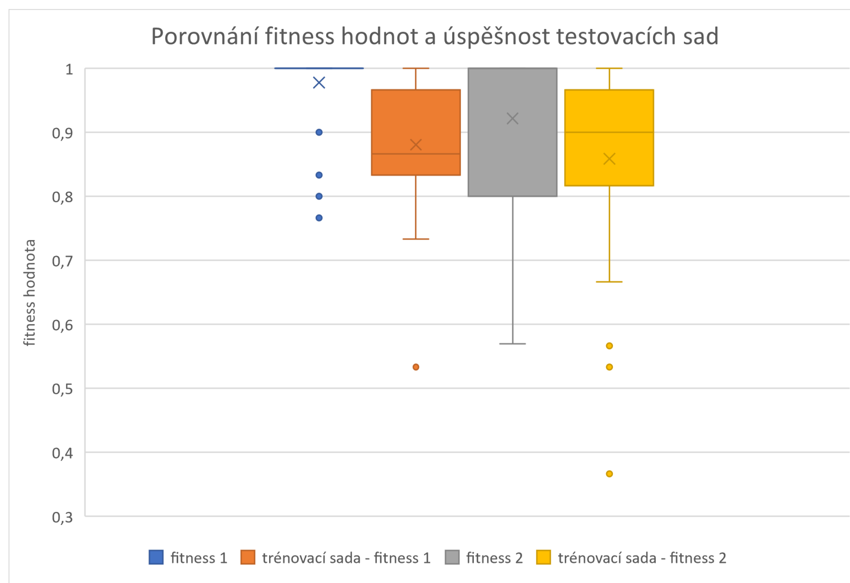
Dalším a posledním typem automatů byly doprava jednosměrně zobecněné skákající automaty. Nastavení parametrů bylo zvoleno stejné jako u předchozích variant. Maximální počet generací byl zvolen 200, pravděpodobnost mutace byla zvolena 0,1 a velikost populace byla zvolena 10, 20, 50 a 100. Test byl proveden pro automat M_{14}, který čte řetězec ab a následně řetězce cd. Úspěšnost pro tento typ automatu byla 100% a všechny běhy programu dokázaly najít řešení a správně rozlišit i všechny řetězce testovacích sad.

Druhým automatem byl vybrán M_7 (obrázek 7.14). Podobný automat byl testován u GJFA experimentů. Automat obsahoval větvení a GA končil úspěšným nalezením řešení, ale měl problémy s testovací sadou. Výsledné fitness hodnoty a úspěšnost správně přijmout/odmítnout řetězce z testovacích sady 50 běhů lze vidět na grafu 7.22. Obě fitness funkce vedly algoritmus k nalezení hledaného jedince, ale ani jedna nenašla jediného jedince, který by byl schopen přijmout/odmítnout řetězce testovacích sad. Stejně tomu bylo u testování automatu M_7 typu GJFA.



Obrázek 7.22: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_{15}

Poslední test byl proveden pro doprava jednosměrně skákající automat M_8 (viz obrázek 7.16). Na grafu 7.23 můžeme vidět, že až na pár výjimek našel algoritmus s fitness funkcí 1 skoro vždy chromozom s fitness hodnotou 1. Fitness funkce 2 vracela častěji jedince, kteří nedokázali správně přijmout a odmítnout všechny řetězce z trénovacích sad. Pokud srovnáme úspěšnost pro řetězce z testovacích sad, lze vidět, že obě fitness funkce si vedly stejně a dokázaly přijmout a odmítnout stejný počet řetězců.



Obrázek 7.23: Graf zobrazující výsledné fitness hodnoty a úspěšnost testovacích sad pro automat M_{16}

Z experimentů plyne, že při špatně vygenerované trénovací sadě nebude genetický algoritmus fungovat podle našich přestav, ačkoli algoritmus skončí s požadovanou fitness hodnotou. Je vhodné volit dostačující počet trénovacích příkladů a zařídit, aby byly dostatečně různorodé. Velkým problémem bylo testování členství řetězce do jazyka popsaného zobecněným skákajících automatem kvůli jeho časové složitosti. Je nutné proto nevolit trénovací sady příliš velké. Z tohohle důvodu není genetický algoritmus vhodný pro odvození GJFA, pokud by byl automat příliš velký (měl příliš rozsáhlou abecedu a počet stavů).

I přesto, že se ve většině případů našel hledaný jedinec, skončila většina testů neúspěchem pro jiné řetězce, než ty, které se nacházely v trénovací sadě. Zjistilo se, že se zvětšujícím se počtem požadovaných stavů se snižuje úspěšnost nalezení jedince, případně úspěšnost přijmout a odmítnout jiné řetězce, než z trénovací sady.

Dále bylo zjištěno, že při větším počtu stavů se testovalo příliš mnoho možností, které neměly žádný dopad na výsledný automat, jako jsou například nedosažitelné stavy nebo stavy, které nekončí v koncových stavech. To začalo mít negativní vliv na dobu běhu programu.

Dalším poznatkem bylo, že obě fitness vracely obdobné výsledky, i když pro většinu testů vykazovala fitness funkce 1 lepší výsledky, než fitness funkce 2.

Kapitola 8

Závěr

Cílem práce bylo zjistit, zda se genetický algoritmus dá použít pro gramatickou inferenci skákajících formálních modelů. Bylo vybráno několik typů těchto modelů a práce hodnotí, které z nich lze odvozovat pomocí genetického algoritmu.

Protože se práce věnovala skákajícím formálním modelům, bylo potřeba nejdříve vysvětlit pojem formální model a jejich typy. Proto se první kapitola věnovala těmto modelům a byly zde krátce definovány.

Po klasických modelech se podrobněji popisovaly skákající formální modely, které z nich vycházejí. Mezi tyto modely patří skákající gramatiky a skákající automaty. V této kapitole byly zmíněny typy a definice těchto gramatik a automatů. Protože se práce více zabývá skákajícími automaty, byly tyto automaty popsány podrobněji.

Následně se práce věnovala inferenci z hlediska evolučních algoritmů. Bylo zde vysvětleno, co to jsou evoluční algoritmy, jak se dělí, jaké složky obsahuje a význam pojmu gramatická inference, neboli odvození gramatiky nebo automatu.

V návrhu se popisovaly veškeré složky potřebné pro inferenci, kterými jsou trénovací sady a zjištění členství řetězce do jazyka popsané automatem. Ty jsou rozdílné pro různé typy automatů. Nakonec se zde popisoval samotný genetický algoritmus, návrh fitness funkce a zakódování chromozomu.

Předposlední část popisovala podrobnou implementaci všech dílčích složek, které byly zmíněny v návrhu. Dále byl popsán zvolený jazyk, nastavení, vstupy a výstupy programů.

Poslední částí byly experimenty pro zvolené typy automatů. Byly vyzkoušeny dvě různé fitness funkce a tyto fitness funkce byly srovnány pro čtyři typy skákajících automatů. Byly jimi JFA, GJFA, doprava jednosměrně skákající JFA a doprava jednosměrně skákající GJFA. Celkem bylo testováno šestnáct různých automatů. Pět pro JFA, tři pro GJFA, pět pro doprava jednosměrně skákající JFA a tři pro doprava jednosměrně skákající GJFA. Pro daná nastavení bylo zjištěno, že pro většinu automatů fungovala fitness funkce 1 lépe než fitness funkce 2, ale samotný rozdíl mezi těmito dvěma funkcemi nebyl příliš velký. Dále bylo zjištěno, že většina výsledných automatů přijímala jiný jazyk, než který jsme hledali. To bylo zjištěno po otestování řetězců z testovacích sad.

V rámci budoucího vývoje se plánuje otestovat i jiné typy skákajících automatů a jejich deterministické varianty. Dalším plánem do budoucna by mohlo být testování jazyků, které nepřijímá tento typ skákajících automatů, místo těch, které tyto automaty přijímaly. Další oblastí budoucího výzkumu je vyvinout optimálnější metody pro zjišťování členství řetězců do jazyka, která je důležitou součástí gramatické inference a která se pro automaty typu GJFA ukázala jako problematická.

Literatura

[1] Amorim, E., Xavier, C., Campos, R. a Santos, R. Comparison between Genetic Algorithms and Differential Evolution for Solving the History Matching Problem. In:. červen 2012, sv. 7333, s. 635–648. DOI: 10.1007/978-3-642-31125-3_48.

[2] Angluin, D. On the Complexity of Minimum Inference of Regular Sets. *Inform. Control*. 1978, sv. 39, č. 3, s. 337–350.

[3] Chigahara, H., Fazekas, S. a Yamamura, A. One-Way Jumping Finite Automata. *RIMS Kôkyûroku*. Listopad 2015, sv. 1964, s. 3–14. DOI: 10.1142/S0129054116400165.

[4] Chomsky, N. On certain formal properties of grammars. *Information and Control*. 1959, sv. 2, č. 2, s. 137–167. DOI: https://doi.org/10.1016/S0019-9958(59)90362-6. ISSN 0019-9958. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0019995859903626.

[5] Choubey, N. a Kharat, M. GRAMMAR INDUCTION AND GENETIC ALGORITHMS- AN OVERVIEW. *Pacific journal of science and technology*. Prosinec 2009, sv. 10, s. 884–889.

[6] Esparza, J., Ganty, P., Kiefer, S. a Luttenberger, M. Parikhs theorem: A simple and direct automaton construction. *Information Processing Letters*. 2011, sv. 111, č. 12, s. 614–619. DOI: https://doi.org/10.1016/j.ipl.2011.03.019. ISSN 0020-0190. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0020019011000822.

[7] Fang, Y. a li, J. A Review of Tournament Selection in Genetic Programming. In:. říjen 2010, s. 181–192. DOI: 10.1007/978-3-642-16493-4_19. ISBN 978-3-642-16492-7.

[8] Fazekas, S., Hoshi, K. a Yamamura, A. Two-Way Jumping Automata. In:. Září 2020, s. 108–120. DOI: 10.1007/978-3-030-59901-0_10. ISBN 978-3-030-59900-3.

[9] Fernau, H., Paramasivan, M., Schmid, M. L. a Vorel, V. Characterization and complexity results on jumping finite automata. *Theoretical Computer Science*. 2017, sv. 679, s. 31–52. DOI: https://doi.org/10.1016/j.tcs.2016.07.006. ISSN 0304-3975. Implementation and Application of Automata. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0304397516303206.

[10] Gold, E. M. Language identification in the limit. *Information and Control*. 1967, sv. 10, č. 5, s. 447–474. DOI: https://doi.org/10.1016/S0019-9958(67)91165-5. ISSN 0019-9958. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0019995867911655.

[11] Lankhorst, M. A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata. Červen 1999.

[12] Lent, J. a Amazeen, M. Noam Chomsky. In:. Leden 2015, s. 1–12. DOI: 10.1057/9781137463418_1. ISBN 978-1-349-56468-2.

[13] Meduna, A., Horáček, P. a Tomko, M. *Handbook of Mathematical Models for Languages and Computation*. The Institution of Engineering and Technology, Stevenage, UK, prosinec 2019. ISBN 978-1-78561-659-4.

[14] Meduna, A. a Zemek, P. Jumping Finite Automata. *International Journal of Foundations of Computer Science*. 2012, sv. 23, č. 7, s. 1555–1578. DOI: 10.1142/S0129054112500244. ISSN 0129-0541. Dostupné z: https://www.fit.vut.cz/research/publication/9795.

[15] Slowik, A. a Kwasnicka, H. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*. 2020, sv. 32, s. 12363–12379. DOI: 10.1007/s00521-020-04832-8. Dostupné z: https://link.springer.com/article/10.1007/s00521-020-04832-8.

[16] Umbarkar, D. A. a Sheth, P. CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW. *ICTACT Journal on Soft Computing (Volume: 6 , Issue: 1)*. Říjen 2015, sv. 6. DOI: 10.21917/ijsc.2015.0150.

[17] Yoon, B.-J. a Vaidyanathan, P. HMM with auxiliary memory: a new tool for modeling RNA secondary structures. *Proceedings of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers: 2004*. Leden 2004, sv. 2.