Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Algoritmy pro po částech testovatelné jazyky

2021

Eliška Foltasová

Vedoucí práce: RNDr. Tomáš Masopust, PhD.

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Eliška Foltasová
Název práce:	Algoritmy pro po částech testovatelné jazyky
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	RNDr. Tomáš Masopust, PhD.
Počet stran:	54
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Eliška Foltasová
Title:	Algorithms for piecewise-testable languages
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Applied Computer Science, full-time form
Supervisor:	RNDr. Tomáš Masopust, PhD.
Page count:	54
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem této bakalářské práce je seznámit čtenáře s tématem jazyků testovatelných po částech a představit algoritmy ověřující tuto vlastnost regulárních jazyků. Praktická část práce se věnuje implementaci tří z popsaných algoritmů a jejich vzájemnému porovnání.

Synopsis

The aim of this bachelor's thesis is to introduce the topic of piecewise testable languages to the reader and to present algorithms verifying this property of regular languages. The practical part of the thesis focuses on the implementation of three of the algorithms described, as well as on their mutual comparison.

Klíčová slova: automat; regulární jazyk; po částech testovatelný jazyk; diskrétní událost; formální jazyk; testovatelnost po částech

Keywords: automaton; regular language; piecewise-testable language; discrete event; formal language; piecewise testability

Tímto děkuji RNDr. Tomáši Masopustovi, PhD. za jeho odborný a zároveň vstřícný přístup při vedení této práce. Dále moc děkuji svým přátelům a rodině.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam vět

Seznam zdrojových kódů

1 Úvod

Regulární jazyky patří k mocným nástrojům informatiky, v níž nacházejí rozličná uplatnění. V této obecné množině lze dále nalézt skupiny jazyků vykazujících určité vlastnosti. Jednou z těchto skupin jsou i jazyky testovatelné po částech.

Pojem *po částech testovatelné události* byl představen roku 1975 v článku Imre Simona *Piecewise testable events*. Simon definuje tento pojem několika ekvivalentními způsoby, které budou v této práci postupně představeny.

V kapitole 2 jsou definovány a ilustrovány elementární i specializované pojmy z oblastí teorie formálních jazyků a automatů, algebry a teorie grafů, které jsou využity jednak k definování jazyka testovatelného po částech, dále pak k popisu fungování představených algoritmů.

Kapitola 3 postupně představuje algoritmy rozpoznávající jazyky testovatelné po částech, a to v chronologickém pořadí dle doby jejich vzniku. Součástí každé podkapitoly je sekce věnující se hlavní myšlence algoritmu, popis algoritmu a dále analýza jeho asymptotické časové složitosti.

Kapitola 4 se věnuje popisu praktické části této práce, tedy implementaci tří algoritmů představených v kapitole předchozí. Představuje knihovnu, s jejíž pomocí program vznikl, popisuje strukturu implementovaného zásuvného modulu a některé jeho konkrétní součásti.

V poslední kapitole 5 jsou pak shrnuty výsledky porovnávání chování implementovaných algoritmů v praxi.

2 Základní pojmy

2.1 Regulární jazyk

Abecedou Σ rozumíme konečnou množinu, jejíž prvky se označují *písmena*. Σ^* je množina všech konečných posloupností znaků ze Σ, jež se značí *slova*. Pro slovo x představuje $|x|$ délku tohoto slova, tedy počet písmen, která obsahuje. *Prázdné slovo* neboli řetězec délky 0 označujeme ε. *Jazyk* L nad abecedou Σ je podmnožinou Σ^*.

Definice 1 (Zřetězení slov)
Je dána abeceda A a slova $a, b \in A^*$, kde
$a = a_1, a_2, \ldots a_n$ a $b = b_1, b_2, \ldots b_m$. Zřetězením slov a, b rozumíme posloupnost $a_1a_2...a_nb_1b_2...b_m$ a zapisujeme jej jako $a \cdot b$.

Zřetězení je tedy asociativní binární operace definovaná na množině A^* pro nějakou abecedu A.

Pomocí operace zřetězení definujeme i-tou mocninu jazyka L následovně:

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

$$L^1 = L.$$

$$L^{i+1} = L \cdot L^i.$$

Definice 2 (Kleeneho uzávěr)

$$L^* = \cup_{i=0}^{\infty} L^i.$$

Třídu **regulárních jazyků** nad abecedou A definují následující podmínky:

- Každý prázdný jazyk $\emptyset$ je regulární.
- Pro každé písmeno $a \in A$ je jazyk $\{a\}$ regulární.
- Je-li L regulární jazyk, pak i jazyk L^* je regulární.
- Jsou-li L_1, L_2 regulární jazyky, pak i jazyky $L_1 \cdot L_2$, $L_1 \cup L_2$ jsou regulární.
- Žádný jiný jazyk není regulární.

2.2 Deterministický konečný automat

Definice 3

Deterministický konečný automat (dále jen DKA) $\mathcal{A}$ je pětice $(\Sigma, Q, q_0, \delta, F)$, kde

- Σ je abeceda
- Q je konečná neprázdná množina stavů
- $q_0 \in Q$ je tzv. **počáteční stav**
- δ je **přechodová funkce** ve tvaru $\delta : Q \times \Sigma \to Q$
- $F \subseteq Q$ je množina **konečných stavů**.

Není-li řečeno jinak, ve zbytku této práce rozumíme pod pojmy *automat* či *DKA* právě tuto pětici.

DKA je jednoduchý výpočetní model, jehož fungování lze popsat následujícím způsobem: Na jeho vstupu je zadána posloupnost symbolů $a_1, a_2, a_3, \ldots a_n; a_i \in \Sigma$, jež se nazývá *vstupní slovo*. Na počátku je DKA ve stavu q_0. S každým přečteným symbolem vstupního slova přechází ze stavu q_i do stavu q_{i+1}. Tato posloupnost stavů $q_1, q_2, q_3, \ldots q_n; q_i \in Q$ se nazývá *výpočet* DKA. Stav q_n označujeme jako *výsledný stav* výpočtu.

Řekneme, že automat $\mathcal{A}$ **přijímá** slovo w, pokud jeho výpočet nad w skončí ve stavu $q_i \in F$. V opačném případě automat slovo nepřijímá. Jazyk L je **rozpoznávaný** automatem $\mathcal{A}$ právě tehdy, když automat přijímá všechna slova $w \in L$ a nepřijímá žádná slova, která do tohoto jazyka nepatří.

Třídu jazyků rozpoznávaných deterministickými konečnými automaty označujeme jako *regulární jazyky*. Pro každý regulární jazyk L tedy platí, že lze sestavit takový DKA, který akceptuje právě všechna slova z L (a žádná jiná).

Dva automaty $\mathcal{A}_1$ a $\mathcal{A}_2$ nazveme ekvivalentní právě tehdy, když rozpoznávají tentýž jazyk L.

Existuje-li přechod ze stavu q_i do stavu q_j, označuje se stav q_j jako **přímý následovník** stavu q_i a stav q_i jako **přímý předchůdce** stavu q_j.

Automat $\mathcal{A}$ rozpoznávající jazyk L označíme jako **minimální**, nelze-li nalézt automat ekvivalentní k $\mathcal{A}$, který obsahuje menší počet stavů než $\mathcal{A}$.

Automat $\mathcal{A}$ nazveme **úplný**, je-li definována přechodová funkce δ pro každý stav $q \in Q$ a každý symbol $a \in \Sigma$. Každý deterministický konečný automat lze tzv. zúplnit.

V některých případech je pro zvýšení přehlednosti vhodné zvolit alternativní zápis aplikace přechodové funkce δ na argumenty, a to za pomoci operátoru $\cdot$:

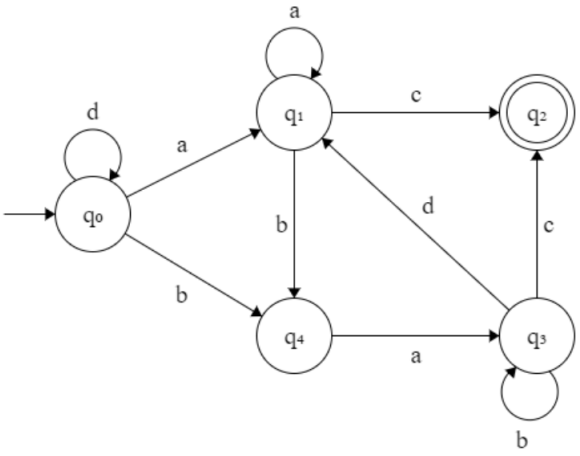
$$\begin{gathered}\delta(q,a) = q \cdot a = qa,\\ \delta(\delta(q,a),b) = q \cdot a \cdot b = q \cdot ab = qab,\\ \delta(\delta(q,a),a) = q \cdot a \cdot a = q \cdot a^2 = qa^2.\end{gathered}$$

2.3 Automat jako graf

Definice 4 (Orientovaný graf)
Orientovaný graf je dvojice $G = (V, E)$, kde V je neprázdná množina vrcholů a $E \subseteq V \times V$ je množina uspořádaných dvojic vrcholů neboli orientovaných hran.

Orientovaný graf obecně slouží k reprezentaci objektů a vztahů mezi nimi. Pro každý deterministický konečný automat $\mathcal{A}$ lze vytvořit jemu odpovídající orientovaný graf $G(\mathcal{A})$. Množina vrcholů V grafu představuje množinu stavů Q automatu a množina hran E je určena přechodovou funkcí δ. Pro stavy q_i, q_j chápeme hranu $\langle q_i, q_j \rangle$ jako přechod ze stavu q_i do stavu q_j. Existuje-li ze stavu q_i do q_j více než jeden přechod, pro větší přehlednost je v grafickém znázornění automatu možné všechny tyto přechody sjednotit jedinou společnou hranou. Každá hrana je pak označena odpovídajícími prvky abecedy Σ.

V náčrtu grafu příslušného DKA dle konvence do počátečního stavu míří šipka a každý konečný stav je zvýrazněn dvojitým kroužkem.



Obrázek 1: Příklad DKA $\mathcal{A}$ se stavy $Q = \{q_0, q_1, q_2, q_3, q_4\}$, počátečním stavem q_0, množinou konečných stavů $F = q_2$ a abecedou $\Sigma = \{a, b, c, d\}$.

Sledem v grafu $G = (V, E)$ rozumíme posloupnost vrcholů a hran $P = (v_0, e_0, v_1, \ldots, e_{n-1}, v_n)$, přičemž pro každou hranu platí $e_i = \langle v_{i-1}, v_i \rangle$.

Příkladem sledu v grafu 1 je posloupnost q_1, q_4, q_3, q_2. Vidíme, že každá dvojice po sobě jdoucích vrcholů je spojena hranou. Oproti tomu posloupnost q_0, q_2, q_4, q_0 není v tomto grafu sledem.

Cestou v grafu $G = (V, E)$ označujeme sled, v němž se žádný vrchol (a tedy ani hrana) neopakuje, tzn. pro každou dvojici vrcholů v_i, v_j platí: $i \neq j \Rightarrow v_i \neq v_j$.

Z hlediska automatu chápeme cestu i sled jako posloupnost stavů symbolizující výpočet nebo jeho část. Každý výpočet je tedy sledem, který vede z počátečního stavu q_0 do nějakého stavu q_n (tento stav může a nemusí patřit do množiny konečných stavů F podle toho, jestli automat dané slovo akceptuje).

Existuje-li v automatu $\mathcal{A}$ sled ze stavu p do stavu q, řekneme, že stav q je **dosažitelný** z p. Tento vztah značíme jako $p \succ q$.

Není-li stav q_i dosažitelný z počátečního stavu q_0, řekneme o něm, že je **nedosažitelný**.

Hloubkou d automatu je délka nejdelší cesty začínající v počátečním stavu q_0.

2.3.1 Acyklický graf

Definice 5 (Cyklus)

Jako **cyklus** neboli uzavřenou cestu v orientovaném grafu označíme posloupnost $P = (v_0, e_0, \ldots, e_{n-1}, v_n)$, kde $v_0 = v_n$ a pro všechny ostatní dvojice vrcholů platí, že pokud $i \neq j$, pak ani $v_i \neq v_j$.

Např. v grafu 1 tvoří cyklus posloupnost vrcholů q_1, q_4, q_3, q_1.

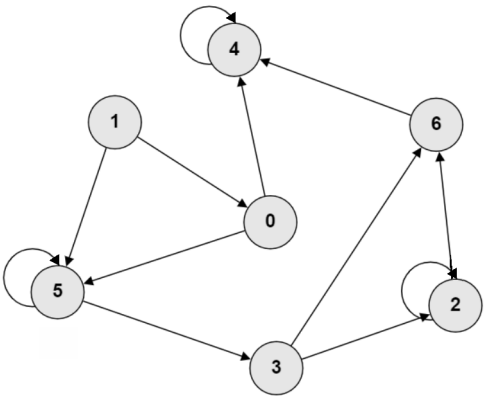
Definice 6 (Smyčka)

Mějme orientovaný graf $G = (V, E)$. **Smyčkou** nazveme takovou hranu $e \in$

$E, e = \langle v_i, v_j \rangle$, pro niž platí $v_i = v_j$.

Smyčky v grafu též označujeme jako tzv. *triviální cykly*.

Orientovaný graf označíme jako **acyklický**, neobsahuje-li žádný cyklus s výjimkou triviálních. Některá literatura ([6]) používá pro DKA v tomto případě označení **částečně uspořádaný automat**. Tyto pojmy jsou ekvivalentní: automat je částečně uspořádaný právě tehdy, když je jeho graf acyklický.



Obrázek 2: Příklad acyklického grafu se smyčkami ve vrcholech 2, 4, 5.

Hledání cyklu v grafu uskutečňuje algoritmus založený na prohledávání grafu do hloubky (zkráceně DFS z anglického *depth-first search*). Tento algoritmus začíná graf procházet v prvním vstupním vrcholu a poté se přesouvá vždy do prvního dosud nenavštíveného následovníka současného vrcholu v pořadí.. V případě, že algoritmus dorazí do vrcholu, který nemá žádné následovníky nebo již byl navštíven, vrací se zpět do předchůdce tohoto vrcholu a postup opakuje, dokud nejsou navštíveny všechny vrcholy.

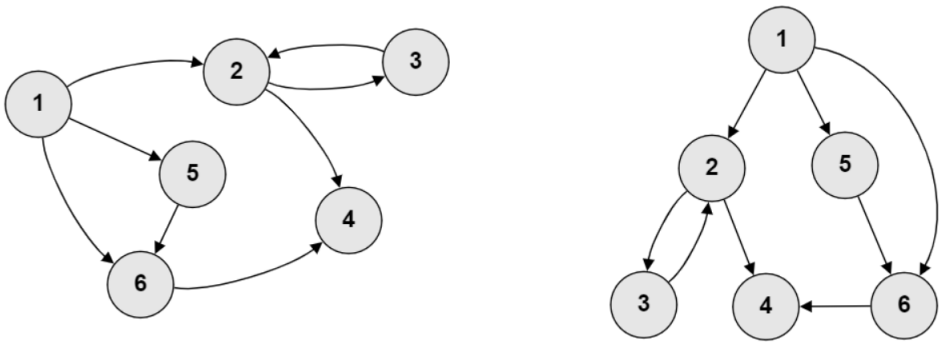
Orientovaný graf lze znázornit formou tzv. DFS-stromu. První navštívený vrchol, který v automatu obvykle odpovídá počátečnímu stavu q_0, je kořenem tohoto stromu.

Každý vrchol grafu se nachází v jednom ze tří možných stavů:

- FRESH: vrchol zatím nebyl navštíven (na počátku jsou v tomto stavu všechny vrcholy).
- OPEN: vrchol nebo některý z jeho následovníků je právě navštěvován.
- CLOSED: vrchol byl navštíven i se všemi následovníky. Algoritmus se vrací o úroveň výše, nenachází-li se v kořeni stromu (po skončení běhu algoritmu musí být v tomto stavu všechny vrcholy).

Cyklus v grafu existuje právě tehdy, když jemu příslušný DFS-strom obsahuje tzv. **zpětnou hranu**. Jedná se o hranu, která vede do vrcholu nacházejícího se ve stavu OPEN. Příkladem zpětné hrany je hrana $\langle 3; 2 \rangle$ na obrázku 3.

Algoritmus si tedy o každém vrcholu pamatuje dva údaje: zda byl navštíven v rámci celého grafu a zda byl navštíven v rámci právě procházeného sledu z kořene. Tyto dvě informace zajistí, že běh algoritmu skončí po prohledání všech vrcholů a že případná existence cyklu bude zjištěna.



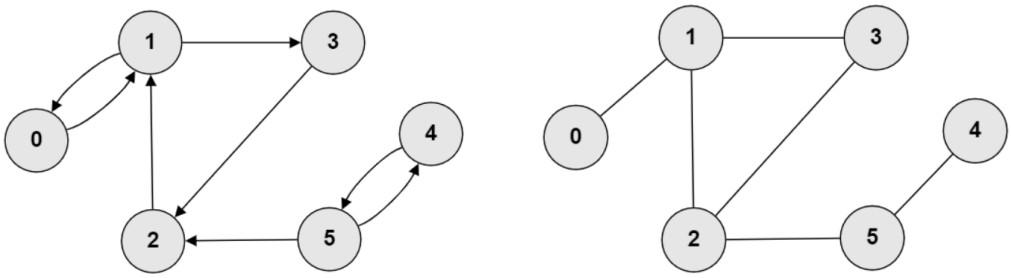
Obrázek 3: Orientovaný graf (vlevo) a tentýž graf v podobě DFS.stromu.

2.3.2 Neorientovaný graf

Neorientovaný graf $G = (V, E)$ definujeme analogicky jako orientovaný graf (definice 4) s tím rozdílem, že množina hran E je tvořena dvouprvkovými množinami vrcholů ve tvaru $\{v_i, v_j\}$ pro dvojice stavů v_i, v_j, kde $i \neq j$ (na rozdíl od orientovaného grafu tedy nezáleží na jejich pořadí). Tyto hrany se nazývají neorientované.

Neorientovanou kopii N orientovaného grafu $G = (V, E)$ vytvoříme nahrazením každé hrany $\langle u, v \rangle$ množiny E neorientovanou hranou $\{u, v\}$. V případě, že mezi vrcholy u a v vedou orientované hrany oběma směry, nahradíme dvojici hran $\langle u, v \rangle$, $\langle v, u \rangle$ pouze jednou neorientovanou hranou $\{u, v\}$. Smyčky přitom zanedbáváme.

Tato kopie se též nazývá **symetrizace** orientovaného grafu G.



Obrázek 4: Příklad orientovaného grafu (vlevo) a jeho symetrizace.

2.3.3 Komponenta grafu

Neorientovaný graf $G = (V, E)$ je **souvislý** právě tehdy, když pro každé dva vrcholy $u, v \in V$ existuje cesta mezi vrcholy u a v.

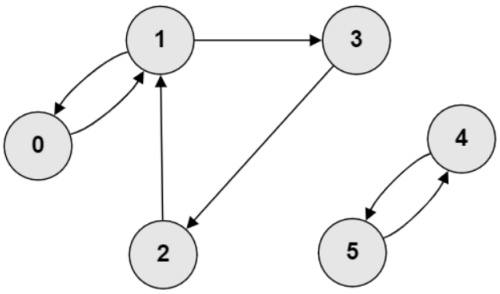
Graf $H = (V_H, E_H)$ se nazývá **podgrafem** grafu $G = (V, E)$, jsou-li splněny následující podmínky:

- $V_H \subseteq V$
- $E_H \subseteq E$
- Pro každou hranu (u, v) množiny E_H musí být oba vrcholy u, v obsaženy v množině V_H.

Komponentou $C = (V_C, E_C)$ neorientovaného grafu G je jeho největší souvislý podgraf.

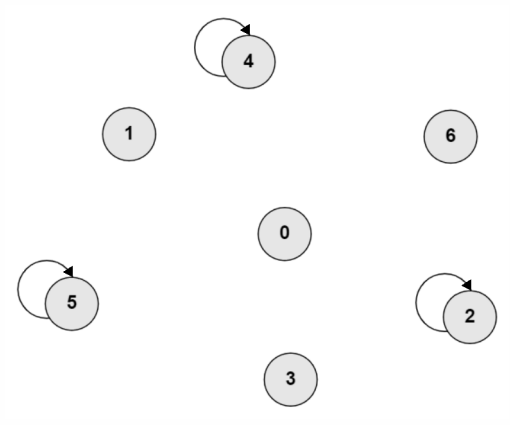
V případě orientovaného grafu rozlišujeme dva typy komponent:

- **Slabě souvislou komponentou** (zkráceně WCC z anglického *weakly connected component*) je takový indukovaný podgraf, v němž pro každou dvojici stavů platí buď $p \succ q$ nebo $q \succ p$.
- **Silně souvislá komponenta** neboli SCC (z anglického *strongly connected component*) je takový největší indukovaný podgraf grafu $G(\mathcal{A})$, v němž pro každou dvojici stavů p, q platí $p \succ q$ a zároveň $q \succ p$. Jedná se o zvláštní případ slabě souvislé komponenty.



Obrázek 5: Graf 4 obsahuje dvě silně souvislé komponenty.

Orientovaný graf je acyklický právě tehdy, když každá jeho silně souvislá komponenta je jednoprvková. Tato skutečnost přímo plyne z definic SCC a cyklu v grafu.



Obrázek 6: Acyklický graf se sedmi vrcholy z obrázku 2 obsahuje právě sedm jednoprvkových silně souvislých komponent.

Stejně jako cyklus, i oba typy komponent jsou v orientovaném grafu hledány modifikovaným algoritmem DFS. Pro nalezení komponenty grafu $G = (V, E)$ obsahující vrchol w je klíčová množina všech vrcholů dosažitelných z w a množina všech vrcholů, z nichž je naopak dosažitelný vrchol w. První jmenovaná z těchto množin je nalezena spuštěním DFS ve vrcholu w grafu G, druhá z nich stejným postupem v grafu G', který vytvoříme z G nahrazením každé hrany $\langle u, v\rangle \in E$ hranou $\langle v, u\rangle$.

Silně souvislá komponenta grafu obsahující v je průnikem těchto množin, slabě souvislá komponenta pak jejich sjednocením.

2.4 Charakterizace jazyka testovatelného po částech

Stěžejním vědeckým dílem v odvětví jazyků testovatelných po částech je článek I. Simona *Piecewise testable events* [6] vydaný roku 1975. Pojem *událost testovatelná po částech* (v tomto kontextu chápaná jako jazyk) autor představil již o tři roky dříve ve své disertační práci.

2.4.1 Charakterizace z hlediska syntaktického monoidu

Definice 7

Algebraická struktura s jednou binární operací je dvojice $(M, \cdot)$, kde

- M je množina
- $\cdot$ je binární operace definovaná na množině M,

přičemž M je vůči této operaci **uzavřená**, tzn. pro všechny prvky $a, b \in M$ platí: $a \cdot b \in M$.

Definice 8 (Neutrální prvek)

Je dána algebraická struktura $(M, \cdot)$. Neutrální prvek $e \in M$ je takový prvek, že pro každý prvek $a \in M$ platí: $a \cdot e = a$, zároveň $e \cdot a = a$.

Definice 9 (Asociativní operace)
Je dána algebraická struktura $(M, \cdot)$. Operace $\cdot$ je *asociativní* právě tehdy, když pro libovolné tři prvky $a, b, c \in M$ platí, že $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

Definice 10 (Monoid)
Monoid je algebraická struktura $(M, \cdot, e)$, kde

- M je množina
- $\cdot$ je asociativní operace na množině M
- e je neutrální prvek množiny M.

Definice 11
Nechť A je abeceda. Pak algebraická struktura $(A^*, \cdot, \varepsilon)$, kde $\cdot$ je operací zřetězení slov z A^* a $\varepsilon \in A^*$ je prázdné slovo, se nazývá **volný monoid generovaný množinou** A.

Definice 12
Nechť n je nějaké přirozené číslo. Pak **booleovskou kombinací** jazyků $L_1, L_2, \ldots, L_n$ rozumíme jazyk, který vznikl z jazyků $L_1, L_2, \ldots, L_n$ pomocí konečného počtu aplikací operací sjednocení, průniku a doplňku.

Věta 13
Po částech testovatelný jazyk *je konečnou booleovskou kombinací jazyků ve tvaru* $A^*a_1A^*a_2...A^*a_nA$, *kde* $n \geq 0$, $a_1, \ldots a_n$ *jsou písmena abecedy* A *a* A^* *je volným monoidem nad* A.

Definice 14 (Greenovy relace)
Mějme monoid $(M, \cdot)$. Definujeme na něm relace $\mathcal{R}$, $\mathcal{L}$, $\mathcal{J}$ následujícím způsobem:

$$\begin{aligned} a\mathcal{R}b &\iff a \cdot M^1 = b \cdot M^1 \\ a\mathcal{L}b &\iff M^1 \cdot a = M^1 \cdot b \\ a\mathcal{J}b &\iff M^1 \cdot a \cdot M^1 = M^1 \cdot b \cdot M^1 \end{aligned}$$

Definice 15
O pologrupě M řekneme, že je **$\mathcal{J}$-triviální**, právě když pro všechny prvky $a, b \in M$ platí, že pokud $a\mathcal{J}b$, pak $a = b$.

Následující věta je významným poznatkem dosaženým v Simonově článku [6]:

Věta 16
Regulární jazyk je po částech testovatelný právě tehdy, když je jeho syntaktický monoid konečný a $\mathcal{J}$*-triviální.*

Na základě této věty Simon vyvozuje, že testovatelnost jazyka po částech je rozhodnutelným problémem. Tento fakt je nezbytným základem pro budoucí vývoj algoritmů ověřující testovatelnost jazyka po částech. Nemá smysl vytvářet algoritmus pro problém, o němž není známo, zda je rozhodnutelný.

2.4.2 Charakterizace automatu přijímajícího jazyk

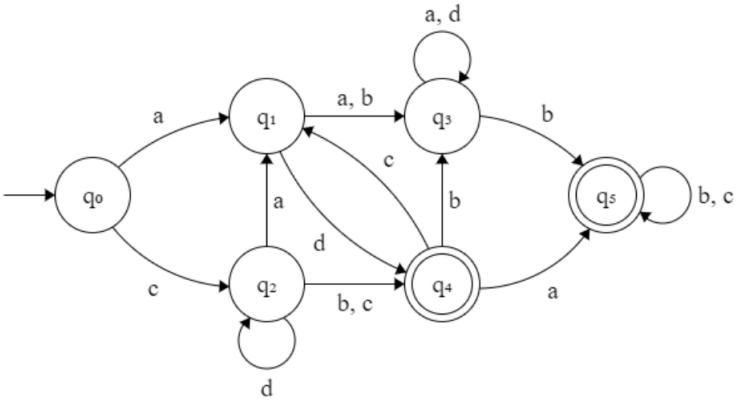
Přestože poznatek 16 má z algebraického hlediska velký význam, jeho formulace neposkytuje přímočarý návod pro hledání algoritmu, který v praxi ověří, jestli je regulární jazyk testovatelný po částech.

Simon v poslední části svého článku zveřejňuje alternativní definice jazyka testovatelného po částech s ohledem na automat rozpoznávající tento jazyk. Tuto ideu později další autoři využívají jako základ pro své algoritmy.

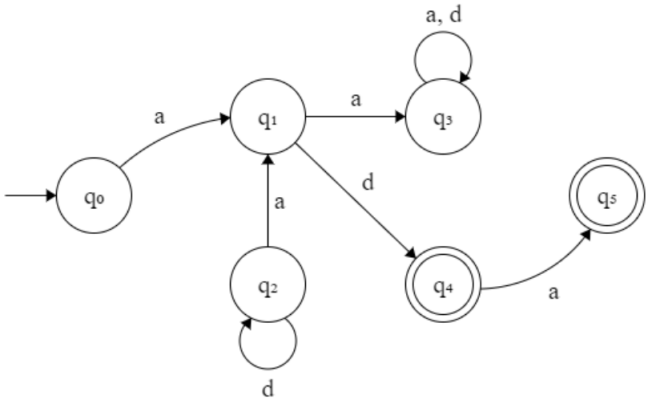
Pro plné pochopení této definice je nezbytné představit několik nových pojmů. První z nich Simon ve své práci označuje jako *dead state*, v další literatuře ([7], [8]) je nazýván *maximal state*:

Stav $q \in Q$ se v automatu $G(\mathcal{A})$ označuje jako **maximální**, platí-li pro každé písmeno $a \in \Sigma : \delta(q, a) = q$. Jedná se tedy o stav, z nějž není v grafu příslušném automatu $\mathcal{A}$ dosažitelný žádný jiný stav $p \in Q, p \neq q$.

Automatem $\mathcal{A}_B$ chápeme takový automat, jehož množina přechodů je omezena abecedou $B \subseteq \Sigma$. $\mathcal{A}_B$ vytvoříme z $\mathcal{A}$ odstraněním všech přechodů označených písmeny $a \in (\Sigma - B)$. I tento automat je v různých zdrojích označován odlišně, např. Stern a Trahtman používají značení $G(\mathcal{A}, \Gamma)$ (jedná se doslova o orientovaný graf reprezentující automat $\mathcal{A}$ s redukovanou abecedou $\Gamma \subseteq \Sigma$).



(a)



(b)

Obrázek 7: Automat $\mathcal{A}$ s abecedou $\Sigma = \{a, b, c, d\}$ (nahoře) a jeho restrikce $\mathcal{A}_B$, kde $B = \{a, d\}$.

Věta 17 (Simonova alternativní charakterizace jazyka testovatelného po částech)

Je dán regulární jazyk $L \subseteq \Sigma^*$, automat $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ rozpoznávající tento jazyk a syntaktický monoid M generovaný jazykem L. Následující tvrzení jsou ekvivalentní:

a) Jazyk L je po částech testovatelný.

b) Automat $\mathcal{A}$ je částečně uspořádaný a pro každý stav $q \in Q$ a každou dvojici slov $x, y \in \Sigma^*$ platí: jestliže $qx = q(xx) = q(xy)$ a $qy = q(yy) = q(yx)$, pak $qx = qy$.

c) Automat $\mathcal{A}$ je částečně uspořádaný a pro každou neprázdnou abecedu $B \subseteq \Sigma$ obsahuje každá komponenta automatu $\mathcal{A}_B$ právě jeden maximální stav.

d) M je $\mathcal{J}$-triviální.

3 Algoritmy rozpoznávající po částech testovatelné jazyky

Tato kapitola představuje čtyři různé algoritmy vytvořené od publikace Simonova článku, na nějž každý z nich svým způsobem odkazuje.

Na konci každé podkapitoly je určena asymptotická časová složitost daného algoritmu v nejhorším případě, přesněji O-notace. Jedná se o řádovou hodnotu, která poskytuje představu o chování algoritmu v závislosti na velikosti vstupních hodnot. Při jejím výpočtu jsou ignorovány konstanty (např. výraz $5n + 12$ lze zjednodušit jako $O(n)$) a největší důraz je kladen na nejrychleji rostoucí funkci (tedy na tu část algoritmu, která se nejvíce opakuje). Toto zjednodušení může mít za následek zkreslení výsledné hodnoty.

Asymptotická časová složitost je pro každý algoritmus orientačním ukazatelem, který na rozdíl od reálné doby běhu programu není závislý na konkrétní podobě implementace a použitých technologiích.

3.1 Sternův algoritmus

První polynomiální algoritmus pro rozpoznávání testovatelnosti jazyka po částech popsal J.Stern [7]. Stern modifikoval původní Simonovu definici minimálních automatů rozpoznávajících jazyky testovatelné po částech.

Přestože algoritmus byl od doby své publikace překonán řádově rychlejšími následovníky, jeho vznik je důležitým milníkem v oblasti výzkumu testovatelnosti jazyků po částech a zaslouží si proto své místo mezi ostatními algoritmy.

3.1.1 Idea algoritmu

Stern používá výraz $G(\mathcal{A}, \Gamma)$, jehož definice je prakticky totožná s

S využitím tohoto pojmu formuluje Stern na základě Simonových poznatků následující podmínky testovatelnosti jazyka po částech:

Věta 18
Nechť L je regulární jazyk a $\mathcal{A}$ je minimální automat rozpoznávající jazyk L. Jazyk L je po částech testovatelný právě tehdy, když jsou splněny následující předpoklady:

1. *$G(\mathcal{A})$ je orientovaný acyklický graf*
2. *Pro každou podmnožinu Γ abecedy Σ má každá komponenta grafu $G(\mathcal{A}, \Gamma)$ právě jeden maximální stav.*

Jelikož podmnožin abecedy o velikosti n existuje právě 2^n, algoritmus ověřující výše zmíněné podmínky pro každou podmnožinu $\Gamma \subseteq \Sigma$ by pracoval s exponenciální časovou složitostí. Taková složitost je ovšem pro praktické využití nevyhovující.

Stern definuje pro stav q množinu $\Sigma(q)$, s níž dále pracuje i Trahtman a označuje ji pojmem **stabilizátor** (v originále *stabilizer*):

$$\Sigma(q) = \{a \in \Sigma : \delta(q, a) = q\}.$$

Stabilizátorem stavu q je tedy podmnožina abecedy Σ, jejíž všechny prvky označují smyčky vedoucí z q do q.

Pojmy *maximální stav grafu* a *stabilizátor stavu* spolu úzce souvisejí. Tento vztah Stern upřesňuje následovně:

Lemma 19
Stav q je maximálním stavem komponenty C acyklického grafu $G(\mathcal{A}, \Gamma)$ právě tehdy, když platí $q \in C$ a zároveň $\Gamma \subseteq \Sigma(q)$.

Můžeme si povšimnout, že např. v grafu 7 *b*) je množina B stabilizátorem stavu q_3, který je zároveň maximálním stavem grafu $\mathcal{A}_B$.

Stern dochází k závěru, že jsou-li dva vzájemně různé stavy q, q' oba maximálními stavy nějaké komponenty C grafu $G(\mathcal{A})$, pak jsou také nutně oba maximálními stavy nějaké komponenty grafu $G(\mathcal{A}, \Sigma(q) \cap \Sigma(q'))$.

Tato skutečnost přímo vyplývá z lemmatu 19: Jsou-li q a q' maximálními stavy komponenty C, abeceda Γ_i označující všechny hrany C je nutně podmnožinou stabilizátorů $\Sigma(q)$ i $\Sigma(q')$, a tedy i podmnožinou jejich průniku.

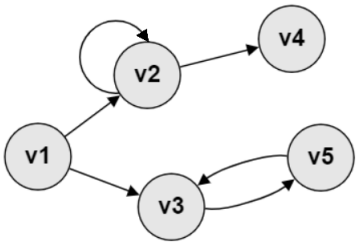
Na základě tohoto tvrzení je možné se při ověřování podmínek 18 omezit pouze na ty podmnožiny abecedy Σ, které jsou ve tvaru $\Sigma(q) \cap \Sigma(q')$ pro dvojice vzájemně různých stavů q, q'. Právě toto výrazné omezení počtu podmnožin výchozí abecedy Σ umožňuje sestrojit první algoritmus ověřující testovatelnost jazyka po částech, který pracuje s polynomiální časovou složitostí.

3.1.2 Tranzitivní uzávěr grafu

Tranzitivní uzávěr $\overline{G} = (\overline{V}, \overline{E})$ orientovaného grafu $G = (V, E)$ je graf, který vznikne z grafu G následovně:

- Množina vrcholů je zachována beze změny, tzn. platí $\overline{V} = V$.
- Pro každou dvojici vrcholů $u, v \in V$, pro niž platí $u \succ v$, je množina $\overline{E}$ doplněna o hranu $\langle u, v \rangle$.

Hledání tranzitivního uzávěru orientovaného grafu uskutečňuje Stern pomocí Warshallova algoritmu [1]. Graf coby vstupní parametr je zde reprezentován maticí o velikostí $n \times n$, kde $n = |V|$. Na i, j-té pozici v matici se nachází hodnota 1 právě tehdy, když existuje hrana $\langle i, j \rangle$. Pokud tato hrana neexistuje, na pozici se nachází hodnota 0.

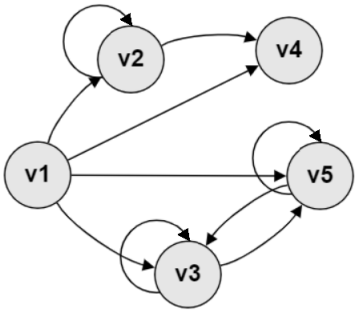


	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	0	0
v_2	0	1	0	1	0
v_3	0	0	0	0	1
v_4	0	0	0	0	0
v_5	0	0	1	0	0

Obrázek 8: Příklad orientovaného grafu (vlevo) a jeho reprezentace maticí o rozměrech 5×5.

Algoritmus pro každou trojici vrcholů $u, v, w \in V$ ověří, zda existuje hrana $\langle u, v \rangle$. V záporném případě je na u, v-tou pozici v matici dosazeno maximum z hodnot na u, w-té a w, v-té pozici.

Jelikož Warshallův algoritmus prochází matici třemi vnořenými cykly, jeho časová složitost je $O(n^3)$ pro $n = |V| = |Q|$.



	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	0	1	0	1	0
v_3	0	0	1	0	1
v_4	0	0	0	0	0
v_5	0	0	1	0	1

Obrázek 9: Tranzitivní uzávěr grafu z obrázku 8.

3.1.3 Průběh algoritmu

Je dán regulární jazyk L a automat $\mathcal{A}$, který jej rozpoznává. Testovatelnost jazyka L po částech zjistíme následující posloupností kroků:

1. Určíme graf $G(\mathcal{A})$.
2. Tomuto grafu vytvoříme tranzitivní uzávěr $\overline{G(\mathcal{A})}$.
3. Ověříme, zda graf $G(\mathcal{A})$ je acyklický. V záporném případě jazyk L nemůže být po částech testovatelný a algoritmus vrací hodnotu $false$.
4. Pro každý stav $q \in Q$ určíme jeho stabilizátor $\Sigma(q)$.
5. Pro každou dvojici vzájemně různých stavů $(q, q') \in Q \times Q$ určíme:
   a) graf $G(\mathcal{A}, \Sigma(q) \cap \Sigma(q'))$
   b) tranzitivní uzávěr $\overline{G(\mathcal{A}, \Sigma(q) \cap \Sigma(q'))} = \overline{G(q, q')}$

c) pro každý stav $p \in Q, p \neq q, p \neq q'$ ověříme, že $\langle p, q\rangle$ a $\langle p, q'\rangle$ nejsou obě současně hranami grafu $\overline{G(q, q')}$.

3.1.4 Analýza časové složitosti

Časová složitost algoritmu je vázána na dvě proměnné - počet stavů $|Q| = n$ a velikost abecedy $|\Sigma| = m$. Zajímá nás horní odhad časové složitosti algoritmu v nejhorším případě, konstanty přitom zanedbáváme. Zanalyzujeme každý krok algoritmu zvlášť a následně určíme hodnotu nejrychleji rostoucí funkce, která je pro odhad asymptotické časové složitosti klíčová.

1. Předpokládáme, že je zvolena vhodná reprezentace automatu tak, abychom na něm mohli provádět grafové operace, využívat grafové algoritmy apod. Časovou složitost tohoto kroku tedy můžeme zanedbat.

2. Tranzitivní uzávěr $\overline{G(M)}$ hledáme Warshallovým algoritmem, jehož časová složitost je $O(n^3)$.

3. Hledání cyklu uskutečňujeme vhodně upraveným algoritmem DFS, jehož časová složitost je $O(mn + n)$.

4. Stabilizátor jednoho stavu q nalezneme procházením hran vycházejících z tohoto stavu, jejichž počet je nejvýše m. Pro n stavů má tedy tento krok celkovou časovou složitost $O(mn)$.

5. Následující kroky provádíme pro každou dvojici vzájemně různých stavů q, q', tedy celkem $\frac{n \cdot n}{2}$-krát (pro zjednodušení $O(n^2)$-krát):

   (a) Graf $G(\mathcal{A}, \Sigma(q) \cap \Sigma(q'))$ vytváříme z grafu $G(\mathcal{A})$ odstraněním všech přechodů označených písmeny, která nepatří ani do množiny $\Sigma(q')$, ani do $a \in \Sigma(q')$. Velikost množiny $\Sigma(q) \cap \Sigma(q')$ je nejméně 0, nejvýše m. Odstraňujeme proto nejvýše $O(mn)$ hran.

   (b) Nalezení tranzitivního uzávěru grafu $G(q, q')$ opět zabere $O(n^3)$ kroků, jelikož i graf $G(q, q')$ obsahuje n vrcholů.

   (c) Stavů v grafu je kromě q, q' právě $n - 2$, ověření existence hran $\langle p, q\rangle$, $\langle p, q\rangle$ zabere při vhodně zvolené reprezentaci $O(1)$ kroků. Časová složitost tohoto kroku je tedy $O(n)$.

Vidíme, že krok 5*b*) algoritmu zabere řádově $n^2 \cdot n^3 = n^5$ kroků, což je nejvyšší nalezená hodnota. Ostatní dílčí časové složitosti tedy můžeme zanedbat.

Asymptotická časová složitost Sternova algoritmu v nejhorším případě je $O(n^5)$.

3.2 Trahtmanův algoritmus

Trahtmanovi se podařilo modifikovat Sternem definované podmínky testovatelnosti jazyka po částech a představit tak algoritmus s nižší časovou složitostí.

Zásadním Trahtmanovým vylepšením Sternova algoritmu je upuštění od nutnosti hledat tranzitivní uzávěr grafu $G(\mathcal{A})$ pro každou dvojici stavů q, q'. Jak bylo uvedeno v sekci 3.1.4, právě nalezení tranzitivního uzávěru je časově nejnáročnější částí programu.

3.2.1 Idea algoritmu

Řekneme, že stavy p, q automatu jsou **propojené** právě tehdy, když platí buď $p \succ q$ nebo $q \succ p$.

Trahtman představuje ve svém článku následující lemma, pomocí nějž dále definuje alternativní verzi podmínek testovatelnosti jazyka po částech:

Lemma 20

Předpokládejme, že graf $G(\mathcal{A})$ je acyklický a pro nějakou podmnožinu Σ_i abecedy Σ má graf $G(\mathcal{A}, \Sigma_i)$ dva vzájemně různé, propojené maximální stavy. Potom pro nějaký stav $p \in Q$ má graf $G(\mathcal{A}, \Sigma(p))$ také dva vzájemně různé, propojené maximální stavy, přičemž stav p je jedním z těchto stavů.

Důkaz

Předpokládejme, že p a q jsou dva různé, vzájemně propojené maximální stavy nějaké silně souvislé komponenty grafu $G(\mathcal{A}, \Sigma_i)$. Jelikož jsou všechny SCC acyklického grafu jednoprvkové, pak nutně musí platit buď $p \not\succ q$, nebo $q \not\succ p$.

Předpokládejme, že platí např. $q \not\succ p$, a vezměme si graf $G(\mathcal{A}, \Sigma(p))$. Pak určitě platí $G(\mathcal{A}, \Sigma_i) \subset G(\mathcal{A}, \Sigma(p))$ a stav p je tedy maximálním stavem grafu $G(\mathcal{A}, \Sigma(p))$. Stavy p a q jsou zcela určitě propojeny i v tomto grafu. Stav q nebo nějaký jeho následovník musí být také maximálním stavem v též komponentě $G(\mathcal{A}, \Sigma(p))$ jako p. Tímto následovníkem ovšem určitě nemůže být stav p, jelikož dle předpokladu platí $q \not\succ p$. □

Věta 21

Nechť L je regulární jazyk nad abecedou Σ a $\mathcal{A}$ je minimální automat rozpoznávající L. Pak jazyk L je po částech testovatelný právě tehdy, když jsou splněny následující podmínky:

- *$G(\mathcal{A})$ je acyklický orientovaný graf.*
- *Pro každý stav p má maximální silně souvislá komponenta SCC grafu $G(\mathcal{A}, \Sigma(p))$ obsahující p právě jeden maximální stav.*

3.2.2 Průběh algoritmu

Mějme regulární jazyk L a minimální automat $\mathcal{A}$, který tento jazyk rozpoznává a je reprezentován grafem $G(\mathcal{A})$. Pro zjištění, zda je L po částech testovatelný,

postupujeme následovně:

1. Ověříme, zda je graf $G(\mathcal{A})$ acyklický. V záporném případě není L po částech testovatelný.

2. Pro každý stav p určíme následující:

   - Stabilizátor $\Sigma(p)$.
   - Graf $G(\mathcal{A}, \Sigma(p))$. Tento graf obsahuje všechny vrcholy grafu $G(\mathcal{A})$ a pouze ty hrany, které jsou označeny písmeny ze stabilizátoru $\Sigma(p)$.
   - Neorientovanou kopii N orientovaného grafu $G(\mathcal{A}, \Sigma(p))$.
   - Takovou komponentu C grafu N, která obsahuje p.
   - Pro každý stav $r \in C, r \neq p$ zjistíme, jestli z něj v grafu $G(\mathcal{A}, \Sigma(p))$ vedou hrany, které nejsou smyčky. Pokud z r nevedou žádné hrany nebo všechny hrany z něj vycházející jsou smyčky, pak jazyk L nemůže být po částech testovatelný.

3.2.3 Analýza časové složitosti

Pro určení složitosti algoritmu jsou stěžejní dva parametry automatu: počet stavů množiny Q a velikost abecedy Σ. Označíme si $|Q| = n$, $|\Sigma| = m$. Využijeme také údaj maximálního počtu hran grafu $G(\mathcal{A})$. Ta je rovna násobku počtu stavů a velikosti abecedy, tzn. nm.

1. Hledání cyklu je uskutečněno prohledáváním grafu $G(\mathcal{A})$ do hloubky. Pro prohledání celého grafu do hloubky je nutné uskutečnit $O(n + mn)$ kroků.

2. Následující kroky jsou provedeny v nejhorším případě pro každý stav $p \in Q$, tzn. maximálně n-krát:

   - Velikost stabilizátoru $\Sigma(p)$ stavu p je shora omezena velikostí abecedy Σ, což je také maximální počet hran vycházející z vrcholu p. Pro nalezení stabilizátoru je nutné všechny tyto hrany projít, časová složitost tedy dosáhne $O(m)$ kroků pro jeden stav.
   - Při vytváření grafu $G(\mathcal{A}, \Sigma(p))$ z původního grafu $G(\mathcal{A})$ procházíme všechny hrany grafu $G(\mathcal{A})$ a odstraňujeme ty, které nejsou obsaženy v množině $\Sigma(p)$. Pro jeden takový graf dosahuje časová složitost $O(mn)$ kroků, pro n grafů $O(mn^2)$ kroků.
   - Časová složitost vytváření neorientované kopie N orientovaného grafu je vázána na počet hran $G(\mathcal{A})$, které nahrazujeme neorientovanými hranami. Shora ji omezuje hodnota $O(mn)$.
   - Komponentu grafu N obsahující stav p opět získáme pomocí modifikace algoritmu DFS, který pracuje s časovou složitostí $O(n + mn)$.

- V komponentě C se kromě stavu p nachází maximálně $n-1$ stavů. U každého z nich ověřujeme, zda z něj vychází alespoň jedna hrana, která není smyčka. Největší možný počet smyček vycházející z nějakého stavu $q \in C$ je roven velikosti stabilizátoru $\Sigma(p)$. Celkem tedy hovoříme o $O(nm)$ krocích.

Časová složitost Trahtmanova algoritmu v nejhorším případě je $O(mn^2)$.

3.3 Algoritmus Klímy a Poláka

Stejně jako tvůrci předchozích dvou popsaných algoritmů, i O. Klíma a L. Polák ve svém díle ([4]) vycházejí ze Simonových poznatků představených v sekci 2.4.2, konkrétně pak z podmínky b). Výsledkem jejich práce je zjednodušení této podmínky a vytvoření algoritmu, jehož přímočarost umožňuje i jeho využití při výpočtech prováděných člověkem.

Za zmínku také stojí fakt, že pro dokázání korektnosti svého algoritmu autoři volí zcela nový postup, který se neopírá o původní důkazy prezentované Simonem ani jinými autory.

3.3.1 Idea algoritmu

V sekci 2.4.1 byly uvedeny podmínky, za kterých je syntaktický monoid generovaný množinou $\mathcal{J}$-triviální. Autoři ve svém článku představují alternativní podmínky této vlastnosti, a sice:

Konečný monoid M je J-triviální tehdy a pouze tehdy, když existuje přirozené číslo $m \geq 1$ takové, že pro všechny prvky $a, b \in M$ platí:

$$(ba)^m = (ab)^m = b(ab)^m.$$

Z tohoto vztahu vyplývají také rovnosti $a(ab)^m = (ab)^m$, $(ab)^m a = (ab^m)$, $(ab)^m b = (ab)^m$, $a^{m+1} = a^m$.

Můžeme si povšimnout souvislosti tohoto tvrzení s bodem b) Simonovy alternativní charakterizace jazyka testovatelného po částech (17).

Věta 22

Nechť L je regulární jazyk rozpoznávaný automatem hloubky k, který je minimální a acyklický. Pak je L testovatelný po k částech.

Stav $q \in Q$ se v automatu nazývá **absorbující**, jestliže pro každé písmeno $a \in \Sigma$ platí $\delta(q, a) = q$. Tento pojem je přitom ekvivalentní k pojmu *maximální stav* představenému v části 2.4.2. Pro abecedu $B \subseteq \Sigma$ nazveme stav $q \in Q$ **B-absorbující** právě tehdy, když je absorbující v automatu $\mathcal{A}_B$.

Autoři pro abecedu $B \subseteq \Sigma$ definují relaci $E_B \subseteq Q \times Q$ následovně:

$$E_B = \{(q, q \cdot b) \mid q \in Q, b \in B\} \cup \{(q \cdot b, q) \mid q \in Q, b \in B\}.$$

Reflexivně-tranzitivní uzávěr E_B je relací ekvivalence na Q a značí se $\approx_B$. Třídami rozkladu příslušného této ekvivalenci na množině Q jsou slabě souvislé komponenty grafu $G(\mathcal{A})$.

Věta 23

Nechť L je regulární jazyk a $\mathcal{A}$ minimální automat rozpoznávající tento jazyk. Pak je L po částech testovatelný tehdy a právě tehdy, když jsou splněny následující podmínky:

a) $G(\mathbb{A})$ je acyklický graf

b) pro každou podmnožinou $B \subseteq \Sigma$ platí: jestliže $p \approx_B$ jsou B-absorbující stavy, pak $p = q$.

Definice 24

Nechť $\mathcal{A}$ je automat a $B \subseteq \Sigma$ je podmnožinou abecedy toho automatu. Řekneme, že $\mathcal{A}$ je **B-souběžný**, jestliže pro každý stav $q \in Q$ a pro každou dvojici slov $u, v \in B^*$ existuje takové slovo w, že platí: $(q \cdot u) \cdot w = (q \cdot v) \cdot w$.

Automat se nazývá **souběžný** právě tehdy, když je B-souběžný pro každou podmnožinu B abecedy Σ.

Acyklický automat se nazývá **lokálně souběžný** právě tehdy, když pro každý stav $q \in Q$ a každou dvojici písmen $a, b \in \Sigma$ existuje takové slovo $w \in \{a, b\}^*$, pro které platí: $(q \cdot a) \cdot w = (q \cdot b) \cdot w$.

Autoři docházejí k následujícímu poznatku:

Důsledek 1. *Je dán regulární jazyk L. Tvrzení a) – d) jsou vzájemně ekvivalentní:*

a) L je po částech testovatelný.

b) Syntaktický monoid L je $\mathcal{J}$-triviální.

c) Minimální automat rozpoznávající L je acyklický a lokálně souběžný.

d) Minimální automat rozpoznávající L je acyklický a souběžný.

Lemma 25

Nechť $\mathcal{A}$ je minimální automat s m stavy, který rozpoznává jazyk $L \subseteq \Sigma^*$, a graf $G(\mathcal{A})$ reprezentující tento automat je acyklický. Pak následující podmínky jsou vzájemně ekvivalentní:

a) Pro každou podmnožinu $B \subseteq \Sigma$ a každou dvojici stavů $p, q \in Q$ platí: Jestliže $p \approx_B q$ a p, q jsou oba B-absorbující stavy, pak $p = q$.

b) Pro každý stav $q \in Q$ a každou dvojici písmen $a, b \in \Sigma$ platí $q \cdot a(ab)^m = q \cdot b(ab)^m$.

c) Automat $\mathcal{A}$ je lokálně souběžný.

d) Automat $\mathcal{A}$ je souběžný.

Důkaz

"$a) \Longrightarrow b)$"

Zvolme libovolný stav q a abecedu $B = \{a, b\}$. Pro každé $i = 0, \ldots, m$ si položme $q_i = q \cdot a(ab)^i$. Jelikož automat $\mathcal{A}$ obsahuje právě m stavů, v posloupnosti $q_0, q_1, \ldots, q_m$ se musí některý stav objevit dvakrát. Jelikož je $G(\mathcal{A})$ dle předpokladu acyklický, pro nějaké $i \in \{0, 1, \ldots, m\}$ a pro stav q_i musí platit $q_i \cdot ab = q_i$. Stav q_i je tedy B-absorbující. Jelikož máme $q \cdot a(ab)^m = q_m$, stav q_m je B-absorbující. Analogickým postupem bychom ověřili, že i stav $q \cdot b(ab)^i$ je B-absorbující. Platí $q \cdot a(ab)^m \approx_B q \approx_B q \cdot b(ab)^m$, a využitím podmínky $a)$ nutně i $q \cdot a(ab)^m = q \cdot b(ab)^m$.

"$b) \Longrightarrow c)$"

Stačí dosadit do definice lokálně souběžného automatu $w = (ab)^m$.

"$c) \Longrightarrow d)$"

Vyplývá přímo z Důsledku 1.

"$d) \Longrightarrow a)$"

Mějme B-absorbující stavy p, q. Z ekvivalence $p \approx_B q$ můžeme vyvodit, že existuje posloupnost stavů $p = r_0, r_1, \ldots, r_n = q$ taková, že pro každé $i = 1, \ldots, n$ platí $(r_{i-1}, r_i) \in E_B$.

Chceme dokázat, že pro každé $i = 0, \ldots, n$ existuje slovo w_i takové, že platí $r_i \cdot w_i = p$. Důkaz provedeme indukcí ve vztahu k hodnotě i:

Pro $i = 0$ si zvolme libovolné slovo $w_0 \in B^*$ a předpokládejme, že platí $r_{i-1} \cdot w_{i-1} = p$. Jelikož $(r_{i-1}, r_i) \in E_B$, přicházejí v potaz dvě možnosti:

Jestliže existuje $b \in B$ takové, že platí $r_i \cdot b = r_{i-1}$, pak můžeme položit $w_i = bw_{i-1}$ a získáváme $r_i \cdot bw_{i-1} = r_{i-1} \cdot w_{i-1} = p$.

Jestli naopak existuje $b \in B$ takové, že platí $r_{i-1} \cdot b = r_i$, pak z B-souběžnosti $\mathcal{A}$ plyne existence slova $w_i \in B^*$ takového, že platí $p \cdot w_i = r_i \cdot w_i$. Jelikož stav p je dle předpokladu B-absorbující, získáme vztah $r_i \cdot w_i = p \cdot w_i = p$, čímž je požadovaný důkaz dokončen.

Pro $i = n$ předpokládáme, že existuje slovo $w_n \in B^*$ takové, aby platilo $q \cdot w_n = p$. Jelikož q je B-absorbující, z této rovnosti plyne $p = q$. □

Právě podmínka $b)$ tvrzení je základem Klímova a Polákova algoritmu ověřující testovatelnost jazyka po částech.

Věta 26

Regulární jazyk je po částech testovatelný právě tehdy, když je jeho minimální automat acyklický a lokálně souběžný.

3.3.2 Průběh algoritmu

Nechť L je regulární jazyk rozpoznávaný deterministickým automatem $\mathcal{A}$, který je úplný a minimální. Pak testovatelnost jazyka L po částech ověříme následujícím způsobem:

1. Ověříme, zda je graf $G(\mathcal{A})$ acyklický. V záporném případě nemůže být jazyk L po částech testovatelný.

2. Pro každý stav $q \in Q$ a každou dvojici vzájemně různých písmen $a, b \in \Sigma$ postupně nalezneme posloupnosti $q, q \cdot a, q \cdot a^2, q \cdot aab, \cdots, q \cdot a(ab)^n$ a $q, q \cdot b, q \cdot ba, q \cdot bab, \cdots, q \cdot b(ab)^n$, kde $n = |Q|$.

3. Zjistíme, zda platí rovnost $q \cdot a(ab)^n = q \cdot b(ab)^n$. Pokud se výrazy nerovnají, jazyk L není po částech testovatelný.

3.3.3 Analýza časové složitosti

Časová složitost tohoto algoritmu
Opět pro automat $\mathcal{A}$ označíme parametry: $|Q| = n$, $|\Sigma| = m$.

1. Hledání cyklu pomocí DFS zabere stejně jako v Trahtmanově algoritmu $O(n + mn)$ kroků.

2. Provádíme výpočet pro každý stav $q \in Q$ a každou dvojici písmen $a, b \in \Sigma, a \neq b$, výpočtů je tedy celkem $n \cdot m^2$. Nalezení výsledných stavů $q \cdot a(ab)^n$, $q \cdot b(ab)^n$ zabere $2n + 2$ kroků pro každou trojici (q, a, b). Dohromady je časová složitost této části algoritmu rovna $O(m^2n^2)$

3. V souladu s předchozím bodem porovnáváme $n \cdot m \cdot m$ dvojic výrazů. Celkem má tento krok časovou složitost $O(m^2n)$.

Časová složitost původního algoritmu Klímy a Poláka je $O(m^2n^2)$.

3.4 Algoritmus Klímy, Kunce a Poláka

Autoři v článku [3] navazují na některé své dříve dosažené poznatky [4] a vynalézají postup, který se podobá Trahtmanově algoritmu. Oproti Trahtmanovi, který vytváří dílčí grafy v závislosti na stabilizátorech jednotlivých stavů, autoři tohoto algoritmu se zaměřují na dvouprvkové abecedy.

3.4.1 Idea algoritmu

Lemma 27

Pro každý acyklický automat $\mathcal{A}$ a libovolnou abecedu $B \subseteq \Sigma$ jsou následující podmínky ekvivalentní:

a) Pro každý stav $q \in Q$ a každou dvojici slov $u, v \in B^*$ existují slova $w, w' \in B^*$ taková, že platí $(q \cdot u) \cdot w = (q \cdot v) \cdot w'$.

b) Pro každý stav $q \in Q$ a každou dvojici slov $u, v \in B^*$ existuje slovo $w \in B^*$ takové, že platí $(q \cdot u) \cdot w = (q \cdot v) \cdot w$.

c) Automat $\mathcal{A}_B$ obsahuje v každé slabě souvislé komponentě právě jeden B-absorbující stav.

Důkaz

"$a) \Longrightarrow c)$"

Dva B-absorbující stavy $p, q \in Q$ patří do stejné slabě souvislé komponenty automatu $\mathcal{A}_B$ právě tehdy, když existují stavy $q_1, \ldots, q_m$ a slova $u_1, \ldots, u_m, v_1, \ldots, v_m \in B^*$ taková, že platí $q_1 \cdot u_1 = p$, $q_m \cdot v_m = q$ a $q_i \cdot v_i = q_{i+1} \cdot u_{i+1}$ pro každé $i \in \{1, \ldots, m-1\}$.

Absence netriviálních cyklů v automatu zaručuje, že pro každý stav $r \in Q$ existuje slovo $w \in B^*$ takové, že stav $r \cdot w$ je B-absorbující. Můžeme předpokládat, že všechny stavy $q_i \cdot v_i$ jsou B-absorbující. Dle podmínky $a)$ pro každé $i \in \{1, \ldots, m\}$ existují slova $w_i, w_i' \in B^*$ splňující podmínku $(q_i \cdot u_i) \cdot w_i = (q_i \cdot v_i) \cdot w_i'$. Jestliže jsou ale $q_i \cdot u_i$ a $q_i \cdot v_i$ B-absorbující stavy, pak musí platit $q_i \cdot u_i = q_i \cdot v_i$, a tudíž i $q = p$.

"$c) \Longrightarrow b)$"

Vezměme si libovolný stav $q \in Q$ a slova $u, v \in B^*$. Mějme slovo x vytvořené zřetězením všech znaků z abecedy B, a položme $w = x^d$, kde d je hloubka automatu $\mathcal{A}$. Čteme-li slovo x z libovolného stavu, který není B-absorbující, nutně se posouváme do dalšího stavu. Začneme-li tedy číst slovo w v libovolném stavu, po jeho přečtení se nutně ocitneme v B-absorbujícím stavu. Jelikož stavy $(q \cdot u) \cdot w$, $(q \cdot v) \cdot w$ patří do stejné slabě souvislé komponenty automatu $\mathcal{A}_B$, dle předpokladu se oba rovnají tomuto stavu.

"$b) \Longrightarrow a)$"

Platí triviálně položením $w = w'$. □

3.4.2 Průběh algoritmu

1. Ověříme, že je graf $G(\mathcal{A})$ acyklický. V záporném případě jazyk L nemůže být po částech testovatelný.

2. Pro každou dvouprvkovou abecedu $B \subseteq \Sigma$:

   - vytvoříme graf $G(\mathcal{A}, B)$
   - nalezneme všechny slabě souvislé komponenty C grafu $G(\mathcal{A}, B)$
   - ověříme, že každá z těchto komponent obsahuje právě jeden B-absorbující stav. Obsahuje-li více takových stavů, jazyk L není po částech testovatelný.

3.4.3 Analýza časové složitosti

1. Stejně jako u již popsaných dvou algoritmů ověřujeme acykličnost grafu pomocí DFS s časovou složitostí $O(n + mn)$.

2. Dvouprvkových podmnožin abecedy Σ existuje právě $\binom{m}{2}$. Tolikrát dohromady provedeme následující kroky v nejhorším případě:

   - Graf $G(\mathcal{A}, B)$ obsahuje n vrcholů a $2n$ přechodů, jeho vytvoření tedy zabere $3n$ kroků (ve skutečnosti není nutné tento nový graf vytvářet, stačí nám v původním grafu ignorovat všechny přechody neoznačené písmeny z B).
   - Slabě souvislé komponenty grafu hledáme s využitím DFS. Graf $G(\mathcal{A}, B)$ obsahuje n vrcholů a $2n$ hran. K jeho prohledání je tedy potřeba $n + 2n = 3n$, zjednodušeně $O(n)$ kroků.
   - Zjistit, zda stav je B-absorbující, zabere pro dvouprvkovou abecedu B nejvýše 2 kroky. Součet vrcholů ve všech komponentách grafu $G(\mathcal{A}, B)$ je roven n. Ověřit, každá komponenta má právě jeden B-absorbující stav, lze proto v čase $2n$.

Časová složitost hledání cyklu v grafu stejně jako u předchozích dvou algoritmů nehraje zásadní roli, jelikož řádově nedosahuje složitosti následující části algoritmu.

Hodnotu $\binom{m}{2}$ lze rozepsat jako $\frac{m!}{(m-2)!\cdot 2!} = \frac{m\cdot(m-1)}{2}$. Tento výraz můžeme shora omezit funkcí m^2.

Celková časová složitost kroků prováděných pro každou dvouprvkovou abecedu je $3n + 3n + 2n = 8n$, což zobecníme jako $O(n)$.

Časová složitost novější verze algoritmu Klímy a Poláka je proto $O(m^2 n)$.

4 Implementace algoritmů

Praktická část práce si klade za cíl implementovat poslední tři představené algoritmy a následně běh těchto programů vzájemně porovnat. Tato kapitola se věnuje popisu této implementace a představení použitých technologií.

4.1 Knihovna LibFAUDES

Knihovna libFaudes [5] byla naprogramována v jazyce C++ s pomocí *Standard Template Library*. Jejím primárním účelem je usnadnění práce se systémy diskrétních událostí, kam se řadí i konečné automaty a regulární jazyky.

Název libFAUDES je složen ze slova *library* a ze zkratek FAU (*Friedrich-Alexander-Universität*), kde je knihovna primárně vyvíjena, a DES (*discrete event system*).

Byla vytvořena primárně pro platformu Linux, ale existuje i v rozšířeních spustitelných na platformách MS Windows a Mac OS X.

Open-Source licence (tzv. LGPL - Lesser General Public License) umožňuje volné šíření zdrojového kódu knihovny i její využití pro komerční účely, ovšem bez záruky spolehlivosti.

Knihovnu lze rozdělit na dvě části. Jádro *CoreFaudes* poskytuje definice základních datových typů a funkcí. Ty z nich, které byly využity při vytváření praktické části této práce, jsou představeny ve zbytku této podkapitoly.

Druhou pomyslnou částí je sada zásuvných modulů, jejichž úkolem je rozšíření funkcionality knihovny o nástroje usnadňující její používání a také umožňující práci se specializovanými odvětvími teorie formálních jazyků a automatů. Technologie zásuvných modulů představuje praktický způsob, jak obohatit knihovnu o nové nástroje bez ovlivnění funkcionality jádra. Struktura implementovaného pluginu *Piecewise Testability* je čtenáři přiblížena v podkapitole 4.2.

Třída **faudes::Generator** reprezentuje konečný automat, který s výjimkou nezaručení deterministočnosti (podrobněji popsáno v části 4.3.1) koresponduje s uvedenou definicí 3. V kontextu knihovny libFAUDES jsou tedy pojmy *automat*, *DKA* a *generátor* vzájemně zaměnitelné, není-li řečeno jinak.

Jedním z nejdůležitějších objektů potřebných pro plnohodnotnou reprezentaci konečného automatu je množina. Tu objekt typu **Generator** využívá hned třemi způsoby, a to jako abecedu Σ, množinu stavů Q a množinu přechodů určených funkcí δ.

Třída **faudes::TBaseSet** k implementaci základní množiny využívá šablonu knihovny STL obohacenou o copy-on-write práci s pamětí. Z této knihovny přebírá řadu metod, které dále dětí všichni potomci třídy:

- metody vracející iterátor na první, popř. za poslední prvek množiny: **Begin()**, **End()**
- metoda zjišťující počet prvků množiny: **Size()**
- predikát ověřující existenci prvku a metoda vracející iterátor na prvek: **Exist()**, **Find()**
- metody pro přidání a odebrání prvku: **Insert()**, **Erase()**
- predikát zjišťující, zda je množina prázdná: **Empty()**

Třída dále disponuje operátory provádějícími základní množinové operace: $+$ (sjednocení), $*$ (průnik), $-$ (rozdíl), $<=$ (podmnožinovost), $>=$ (nadmnožinovost).

Z této základní třídy je odvozena třída **faudes::IndexSet**, která implementuje množinu indexů. Nejnižší možnou hodnotou indexu je 1 (hodnota 0 představuje neplatný index). Datový typ **Idx** je odvozen z typu **unsigned int** jazyka C++.

Při přidávání prvků do množiny lze porušit následnost indexů, tzn. **IndexSet** o 5 prvcích nemusí být množina ve tvaru $\{1, 2, 3, 4, 5\}$, ale může sestávat z libovolných kladných hodnot, např. $\{2, 7, 25, 100, 9999\}$. Metoda třídy **MaxIndex()** vrací hodnotu maximálního indexu (0 v případě prázdné množiny).

Indexy jsou automaticky bez ohledu na pořadí jednotlivých provedení operace **Insert()** seřazeny vzestupně a přidání duplicitních indexů je zabráněno: příslušnému prvku je přiřazen jiný, dosud nepoužitý index s nejmenší možnou hodnotou.

faudes::NameSet je odvozena z třídy **IndexSet**. Jedná se o reprezentaci množiny pojmenovaných indexů. Objekt tohoto typu obsahuje odkaz na tabulku symbolů (objekt typu **faudes::SymbolTable**), která uchovává pro každý index jeho jméno.

Třída **faudes::StateSet** je přímým potomkem třídy **IndexSet** a reprezentuje množinu stavů. Pojmenování jednotlivých stavů je možné, nikoliv však povinné.

Třída **faudes::EventSet** představuje množinu událostí neboli abecedu. Jelikož je tato třída přímým potomkem třídy **faudes::NameSet**, každé události je při jejím přidávání do množiny nutné přiřadit název.

Zatímco události jsou globální subjekty, stavy jsou definovány lokálně pro každý generátor. Např. obsahují-li dva různé generátory událost s názvem *alfa*, tato událost má pro oba z nich stejný index a je zapsána v globální tabulce symbolů. Obsahují-li ale dva generátory oba stav s názvem *idle*, jedná se o dva různé stavy, z nichž každý má svůj vlastní index a každý je zapsán v lokální tabulce symbolů příslušného generátoru.

Třída **faudes::TransSet** reprezentuje množinu přechodů určených přechodovou funkcí Σ. Každý přechod je určen trojicí předchůdce (značen jako **X1**), událost (**Ev**) a následovník (**X2**). Výchozí seřazení těchto prvků má podobu X1-Ev-X2, dle potřeby však lze využít i jiného řazení (např. Ev-X1-X2). Přestože objekt typu **TransSet** lze inicializovat samostatně, význam získává až přiřazením ke konkrétnímu generátoru, jinak se jedná pouze o množinu trojic indexů.

Důležitým nástrojem pro časově efektivní procházení množiny je iterátor. Třída **faudes::Iterator** je rovněž přímo odvozena z šablony knihovny *STL* a přebírá od ní základní metody

Stavy, události a přechody mohou být tedy v generátoru adresovány třemi způsoby:

- názvem, je-li definován (časově nejméně efektivní způsob)
- indexem (efektivní způsob fungující na bázi prohledávání setříděného pole)
- iterátorem (časově nejefektivnější způsob využívající dereferenci ukazatele)

Při vytváření programu byly dále použity následující metody třídy **Generator**:

- **StateSet SuccessorStates(Idx st1)** - metoda vrací množinu následovníků stavu st1 v daném generátoru, tzn. pro všechny existující přechody ve tvaru st1-ev-st2 vrací množinu stavů vyskytujících se na pozici st2.
- **void RestrictAlphabet(Eventset ev)** - metoda odstraní z abecedy daného generátoru všechny události, které nejsou obsaženy v množině ev.

- **bool ExistsTransition(Idx st1, Idx ev)** - metoda vrací hodnotu *true* v případě, že generátor obsahuje alespoň jeden přechod ve tvaru st1-ev-st2, kde na pozici st2 je libovolný stav generátoru.
- **bool ExistsTransition(Idx st1, Idx ev, Idx st2)** - tato metoda vrací *true* v případě, že generátoru obsahuje přechod ve tvaru st1-ev-st2 pro konkrétní vstupní hodnoty st1, ev, st2.
- **TransSet ActiveTransSet(Idx st)** - metoda vrací množinu přechodů generátoru, které obsahují stav st na pozici X1.

4.2 Struktura zásuvného modulu

Vývoj nového zásuvného modulu knihovny LibFAUDES lze shrnout do několika kroků:

- implementace nových algoritmů, popř. nových datových typů
- zavedení dokumentace pomocí run-time rozhraní knihovny
- vytvoření odpovídajících vazeb luabindings pro nástroj luafaudes
- integrace modulu se zbytkem knihovny pomocí jejího build-systému

Soubory tvořící zásuvné moduly knihovny libFAUDES jsou pro docílení maximální přehlednosti organizovány jednotným způsobem (s výjimkou možnosti vynechání nepovinných prvků). Užitečným pomocníkem při vytváření nového modulu je ukázkový plugin **example**, který autoři knihovny zahrnuli do jejího standardního instalačního balíčku. Adresářová struktura a formát zdrojových souborů tohoto pluginu sloužily jako šablona i pro vývoj zásuvného modulu Piecewise testability.

Název hlavní složky **pwtestability** odpovídá oficiálnímu názvu pluginu. Jako pro každý plugin libFAUDES je kromě jeho názvu definován i jedinečný prefix (zde **PWT**) zabraňující zmatení se soubory ostatních pluginů v hlavním makefile souboru knihovny, kam je importován jejím build-systémem.

4.2.1 Makefile soubory

Soubor **Makefile.plugin** zajišťuje integraci pluginu se zbytkem knihovny.

Jsou v něm uvedeny následující informace:

- oficiální název pluginu *pwtestability* (proměnná PWT_NAME) odpovídající názvu jeho hlavní složky
- cestu k hlavní složce pluginu v adresáři knihovny (PWT_BASE)
- umístění zdrojových souborů v adresáři (PWT_SRCDIR)

- zdrojové (PWT_CPPFILES, PWT_SOURCES), hlavičkové (PWT_HEADERS), objektové (PWT_OBJECTS) soubory pluginu
- hlavní include soubor pluginu (PWT_INCLUDE)
- soubor s definicemi (PWT_RTIDEFS) a dokumentací (PWT_RTIFREF) pro run-time rozhraní

```
1  PWT_NAME = pwtestability
2  PWT_BASE = ./plugins/$(PWT_NAME)
3  PWT_SRCDIR = $(PWT_BASE)/src
4
5  PWT_CPPFILES = pwt_adjlist.cpp pwt_adjlistud.cpp pwt_adjmatrix.cpp
       pwt_isacyc.cpp pwt_klimakuncpolak.cpp pwt_klimapolak.cpp
       pwt_trahtman.cpp
6  PWT_INCLUDE = pwt_include.h
7  PWT_RTIDEFS = pwt_definitions.rti
8  PWT_RTIFREF = pwt_index.fref
9
10 PWT_HEADERS = $(PWT_CPPFILES:.cpp=.h) $(PWT_INCLUDE)
11 PWT_SOURCES = $(PWT_CPPFILES:%=$(PWT_SRCDIR)/%)
12 PWT_OBJECTS = $(PWT_CPPFILES:%.cpp=$(OBJDIR)/%$(DOT_O))
13 PWT_RTIDEFS := $(PWT_RTIDEFS:%=$(PWT_SRCDIR)/registry/%)
14 PWT_RTIFREF := $(PWT_RTIFREF:%=$(PWT_SRCDIR)/registry/%)
```

Zdrojový kód 1: Ukázka části souboru Makefile.plugin.

Dále je zajištěno připojení hlavního include souboru pluginu k hlavičkovému souboru knihovny libfaudes.h, zařazení všech výše uvedených součásti pluginu mezi ostatní soubory knihovny, integrace pluginu s run-time rozhraním a s nástrojem luafaudes. Tyto úkoly má na starosti build systém knihovny.

Každý zásuvný modul obsahuje jednoduchý tutoriál, který demonstruje uživateli způsob jeho používání. Zdrojové soubory tutoriálu jsou umístěny ve složce **pwtestability/tutorial**, soubory obsahující vstupní generátory pak ve složce **pwtestability/tutorial/data**.

Soubor **Makefile.tutorial** obsahuje názvy zdrojových a spustitelných souborů tutoriálu spolu s cestou k jejich adresáři. Také jsou v něm definována implicitní pravidla pro kompilaci souborů.

4.2.2 Dokumentace

Autoři knihovny sestavili přehledný systém vytváření uživatelské příručky, který umožňuje automatické zahrnutí dokumentace nového zásuvného modulu při dodržení několika zásad.

Pro vygenerování dokumentace přímo ze zdrojového kódu programu využívá knihovna nástroj **doxygen**. Build systém knihovny automaticky spustí doxygen na všech souborech formátu .h a .cpp.

```
1
2  PWT_TUTORIAL_DIR = ./plugins/pwtestability/tutorial
3
4  PWT_TUTORIAL_CPPFILES = \
5    pwt_tutorial_klimapolak.cpp pwt_tutorial_trahtman.cpp
         pwt_tutorial_klimakuncpolak.cpp
6
7  PWT_TUTORIAL_EXECUTABLES = $(PWT_TUTORIAL_CPPFILES:%.cpp=$(
       PWT_TUTORIAL_DIR)/%$(DOT_EXE))
8
9  $(PWT_TUTORIAL_DIR)/%$(DOT_EXE): %.cpp $(LIBFAUDES)
10   $(call FNCT_COMP_APP,$<,$(OBJDIR)/$*$(DOT_O))
11   $(call FNCT_LINK_APP,$(OBJDIR)/$*$(DOT_O),$@)
12   $(call FNCT_POST_APP,$@)
13
14 VPATH += $(PWT_TUTORIAL_DIR)
15 TUTORIAL_EXECUTABLES += $(PWT_TUTORIAL_EXECUTABLES)
16 CLEANFILES += $(PWT_TUTORIAL_EXECUTABLES)
17 SOURCES += $(PWT_TUTORIAL_CPPFILES:%=$(PWT_TUTORIAL_DIR)/%)
```

Zdrojový kód 2: Soubor Makefile.tutorial.

◆ IsBAbsorbing()

bool faudes::IsBAbsorbing (Generator g,
Idx st,
Idx ev1,
Idx ev2
)

This function verifies if the state represented by index st is B-absorbing for the two-element alphabet B which consists of events ev1 and ev2.

For EventSet B, a state q is B-absorbing if for every letter x from B, the transition delta(q, x) is a loop.

Parameters

- **g** input generator
- **st** index representing some state of generator g
- **ev1** index representing a letter from B (different from ev2)
- **ev2** index representing a letter from B (different from ev1)

Returns

'true' if both transitions delta(st, ev1) and delta(st, ev2) are loops, otherwise 'false'

Definition at line 8 of file pwt_klimakuncpolak.cpp.

Obrázek 10: Dokumentace z kódu 3 vygenerovaná pomocí nástroje doxygen.

```
1  $
2  /**
3   * This function verifies if the state represented by index st is B-
        absorbing for the two-element alphabet B which consists of
        events ev1 and ev2.
4   *
5   * For EventSet B, a state q is B-absorbing if for every letter x
        from B, the transition delta(q, x) is a loop.
6   *
7   * @param g
8   * input generator
9   *
10  * @param st
11  * index representing some state of generator g
12  *
13  * @param ev1
14  * index representing a letter from B (different from ev2)
15  *
16  * @param ev2
17  * index representing a letter from B (different from ev1)
18  *
19  * @return
20  * 'true' if both transitions delta(st, ev1) and delta(st, ev2) are
21  * loops, otherwise 'false'
22  *
23  * @ingroup PwtestabilityPlugin
24  **/
25
26 bool IsBAbsorbing(Generator g, Idx st, Idx ev1, Idx ev2);
27 $
```

Zdrojový kód 3: Ukázka dokumentace funkce ze souboru pwt_klimakuncpolak.h.

4.2.3 Soubory run-time rozhraní

Run-time rozhraní (zkráceně RTI) je pokročilou součástí knihovny. Jeho hlavním úkolem je usnadnění vývoje aplikací a používání rozšiřujících nástrojů jako např. překladače luafaudes.

Rozhraní obsahuje registry typů **TypeRegistry** a funkcí **FunctionRegistry**.

Třídy, s nimiž RTI pracuje, se nazývají faudes-typy a jsou všechny potomky třídy **faudes::Type**. Registr těchto typů zahrnuje některé základní třídy knihovny CoreFAUDES (např. Generator, Alphabet), ale také třídy definované v rámci zásuvných modulů. Každý faudes-typ lze načíst ze souboru nebo naopak do něj zapsat. Jelikož zásuvný model pwtestability pracuje primárně s již existujícím typem Generator, nepotřebuje definovat žádné nové faudes-typy. Není proto nutné se faudes-typům podrobněji věnovat.

Funkce definované v rámci RTI se nazývají faudes-funkce a jsou organizovány

obdobným způsobem jako faudes-typy. Abstraktní třída **faudes::Function** poskytuje rozhraní, které přidělí hodnoty argumentům faudes-funkce a danou funkci spustí na těchto argumentech.

Každé faudes-funkci je přidělen objekt typu **faudes::FunctionDefinition**. Tento typ sestává ze stručné dokumentace faudes-funkce, její signatury neboli seznamu povolených vstupních a výstupních parametrů a také z instance odpovídajícího typu (tzv. prototypu), jejímž úkolem je vytvoření instance této funkce zavoláním metody NewFunction().

Všechny vstupně-výstupní parametry faudes-funkce musí být faudes-typy s výjimkou základních typů jazyka C++ bool, string a integer, které jsou automaticky převedeny překladačem.

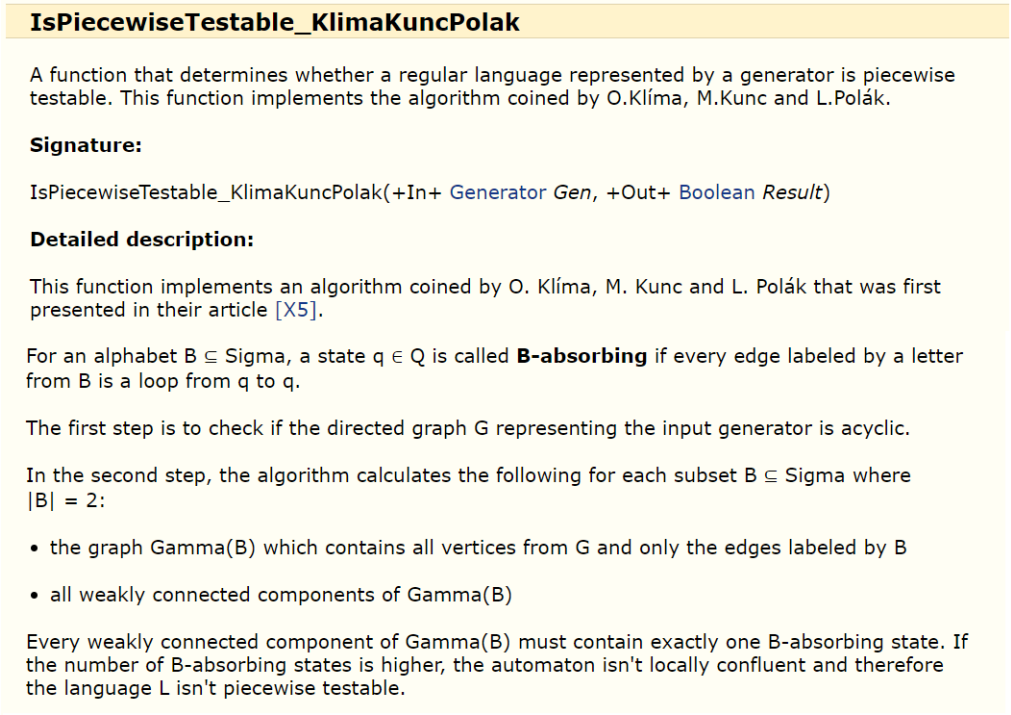
Soubor **pwt_definitions.rti** obsahuje definice celkem tří faudes-funkcí zásuvného modulu. Každá z nich je implementací jednoho ze tří představených algoritmů ověřujících testovatelnost jazyka po částech.

Každá faudes-funkce musí splňovat podmínky popsané v sekci ...

```
1  <FunctionDefinition name="PWTestability::
       IsPiecewiseTestable_Trahtman" ctype="faudes::
       IsPiecewiseTestable_Trahtman">
2
3  <Documentation ref="pwtestability_index.html">
4  A function that determines whether a regular language represented by
        a generator is piecewise testable. This function implements the
        algorithm coined by A.Trahtman.
5  </Documentation>
6  <Keywords>
7  Piecewise testability testable Trahtman
8  </Keywords>
9
10 <VariantSignatures>
11 <Signature name="IsPiecewiseTestable_Trahtman(Gen)">
12 <Parameter name="Gen" ftype="Generator" access="In"/>
13 <Parameter name="Result" ftype="Boolean" access="Out" creturn="true
       "/>
14 </Signature>
15 </VariantSignatures>
16
17 </FunctionDefinition>
```

Zdrojový kód 4: Definice funkce IsPiecewiseTestble_Trahtman() v souboru pwt_definitions.rti. Obdobným způsobem jsou definovány i zbylé dvě faudes-funkce.

Detailnější dokumentace faudes-funkcí je obsažena v souboru **pwtestability-/registry/pwt_index.fref**. Pomocí nástroje **ref2html** je při instalaci knihovny vygenerován soubor ve formátu HTML obsahující popis funkcí, případně další údaje jako ilustrační obrázky či seznam použité literatury).



IsPiecewiseTestable_KlimaKuncPolak

A function that determines whether a regular language represented by a generator is piecewise testable. This function implements the algorithm coined by O.Klíma, M.Kunc and L.Polák.

Signature:

IsPiecewiseTestable_KlimaKuncPolak(+In+ Generator *Gen*, +Out+ Boolean *Result*)

Detailed description:

This function implements an algorithm coined by O. Klíma, M. Kunc and L. Polák that was first presented in their article [X5].

For an alphabet B ⊆ Sigma, a state q ∈ Q is called **B-absorbing** if every edge labeled by a letter from B is a loop from q to q.

The first step is to check if the directed graph G representing the input generator is acyclic.

In the second step, the algorithm calculates the following for each subset B ⊆ Sigma where |B| = 2:

- the graph Gamma(B) which contains all vertices from G and only the edges labeled by B
- all weakly connected components of Gamma(B)

Every weakly connected component of Gamma(B) must contain exactly one B-absorbing state. If the number of B-absorbing states is higher, the automaton isn't locally confluent and therefore the language L isn't piecewise testable.

Obrázek 11: Ukázka části RTI dokumentace vygenerované ze souboru pwt_index.fref nástrojem ref2html.

4.2.4 Soubor rozhraní SWIG

Používání knihovny libFAUDES není omezeno pouze na jazyk C++. **Lua** je vysokoúrovňový skriptovací jazyk podporující procedurální, objektově orientované i funkcionální programování. Přístup jazyku Lua k funkcím a datovým typům knihovny libFAUDES umožňuje prostředí **luafaudes**, které je implementováno pomocí zásuvného modulu **luabindings**.

Nástroj **SWIG** (Simplified Wrapper and Interface Generator) umožňuje snadné propojení vysokoúrovňových skriptovacích jazyků (Javascript, PHP, Perl) s programy napsanými v jazyce C/C++ vytvořením tzv. wrapper-kódu. Tímto způsobem integruje také jazyk Lua s knihovnou libFAUDES.

Tuto integraci zařizuje SWIG rozhraní pluginu definované v souboru **pwt_interface.i**, který se nachází ve složce pwtestability/src/registry/.

4.2.5 Tutoriál

Součástí každého zásuvného modulu je jednoduchý návod seznamující uživatele s možným způsobem jeho používání. Plugin Piecewise Testability poskytuje jeden tutoriál zvlášť pro každý ze tří implementovaných algoritmů.

Zdrojové soubory tutoriálů jsou umístěny ve složce **pwtestability/tutorial**, vstupní data pak ve složce **pwtestability/tutorial/data**. Složka **pwtestability/src/doxygen/faudes_images** obsahuje navíc obrázky ilustrující podobu

```
1  /**
2
3  @file pwt_interface.i
4
5  SWIG interface for piecewise testability plugin.
6
7  See the luabindings README for more documentation.
8
9  **/
10
11 %module pwtestability
12
13 #ifndef SwigModule
14 #define SwigModule "SwigPWTestability"
15 #endif
16
17 %include "faudesmodule.i"
18
19 %luacode {
20 for k,v in pairs(pwtestability) do faudes[k]=v end
21 }
22
23 SwigHelpTopic("PWTestability", "Piecewise Testability Plug-In");
24
25 #if SwigModule=="SwigPWTestability"
26 %include "../../../include/rtiautoload.i"
27 #endif
```

Zdrojový kód 5: Soubor pwt_interface.i.

každého z generátorů, které se v tutoriálech objeví. Knihovna libFAUDES poskytuje užitečnou možnost vygenerování obrázkového výstupu typu .png či .svg ze souboru typu .gen pomocí nástroje **dot** balíčku **Graphviz**.

4.3 Potřebné vlastnosti vstupního automatu

Pro správné fungování algoritmů ověřujících testovatelnost jazyka po částech je nutné, aby automat na vstupu splňoval několik vlastností - musí být deterministický, minimální a v případě obou algoritmů Klímy a Poláka, potažmo Klímy, Kunce a Poláka, i úplný.

Jelikož libovolnému automatu lze vytvořit jeho deterministickou, minimální a úplnou verzi, programu nic nebrání na vstupu přijmout i generátor, který tyto vlastnosti nesplňuje, a dodatečně je pro něj zajistit.

V souboru **pwt_checkinput.cpp** je definována funkce **CheckInputProps()**, která danému vstupnímu generátoru vytvoří ekvivalentní deterministický, minimální a úplný generátor.

Ověření těchto tří vlastnosti jsou prováděna v pořadí, v jakém jsou uvedena

v této sekci. Zvolení jiného pořadí zbytečně zvyšuje celkovou časovou složitost programu, metoda minimalizující generátor navíc nepřijímá jako argument nedeterministický generátor.

4.3.1 Determinističnost

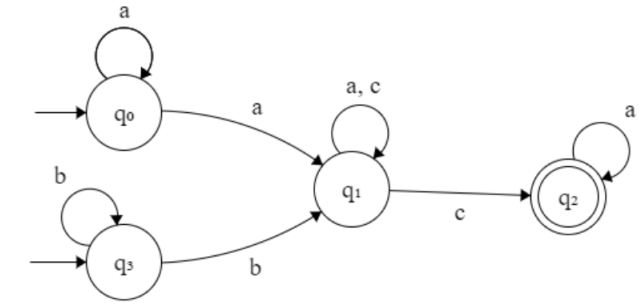
Definice 28

Konečný automat je **nedeterministický**, splňuje-li alespoň jednu z následujících podmínek:

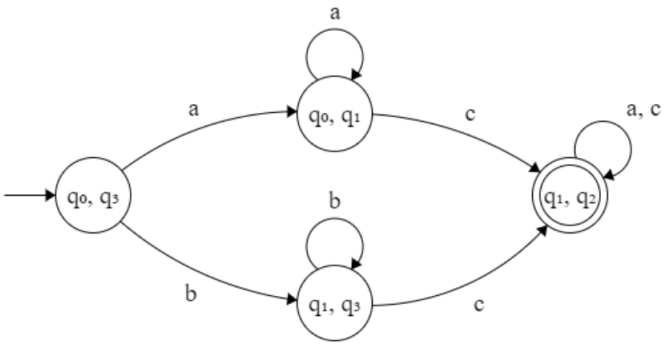
- Existuje více než jeden počáteční stav.
- Pro nějaký stav $q_i \in Q$ a nějaké písmeno $a \in \Sigma$ existuje více než jeden přechod ve tvaru $\delta(q_i, a)$.

Přestože všechny tři zde uvedené algoritmy ověřující testovatelnost jazyka po částech byly navrženy pro deterministické konečné automaty, objekt typu Generator v knihovně libFAUDES není touto podmínkou omezen. Je proto nutné zajistit, aby generátor na vstupu implementovaných funkcí byl deterministický. Každému nedeterministickému konečnému automatu lze sestavit ekvivalentní deterministický konečný automat.

Otázku, zda je instance typu Generator deterministická, ověřuje funkce **IsDeterministic**. V záporném případě pak funkce **Deterministic** vytvoří vstupnímu automatu ekvivalentní deterministický automat (tzn. jazyk L rozpoznávaný vstupním generátorem funkce je roven jazyku rozpoznávanému výstupním, determinizovaným generátorem). Výsledná množina stavů Q' deterministického automatu $\mathcal{A}'$ se rovná nějaké podmnožině potenční množiny původního automatu $\mathcal{A}$. Časová složitost algoritmu implementovaného funkcí Deterministic roste exponenciálně vzhledem k počtu stavů původního automatu.



(a)

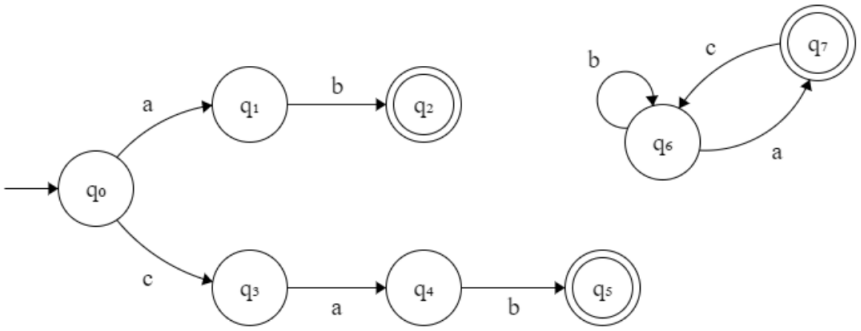


(b)

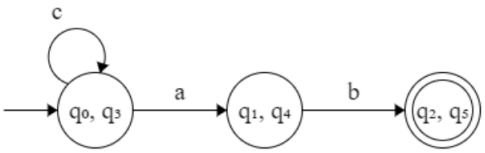
Obrázek 12: Příklad nedeterministického automatu (a) a jeho determinizované verze (b).

4.3.2 Minimálnost

Minimalizace automatu $\mathcal{A}$ je realizována funkcí **StateMin** využívající Hopcroftův algoritmus [2], který odstraňuje všechny nedosažitelné stavy a následně provede rozklad množiny Q na třídy vzájemně nerozlišitelných stavů. Každá z těchto tříd pak tvoří v minimalizovaném automatu $\mathcal{A}'$ jeden stav. Časová složitost tohoto algoritmu je v nejhorším případě $O(mn \cdot \log m)$ pro $|Q| = m; |\Sigma| = n$.



(a)



(b)

Obrázek 13: Příklad automatu (a), který není minimální, a jeho minimalizované verze (b) vzniklé odstraněním nedosažitelných stavů q_6, q_7 a sjednocením vzájemně nerozlišitelných stavů.

Vzhledem ke skutečnosti, že ověřování minimálnosti automatu není triviální proces a jeho složitost se blíží složitosti samotné minimalizace, je žádoucí každý automat rovnou minimalizovat i za cenu toho, že to je "zbytečné."

4.3.3 Úplnost

DKA se nazývá **úplný** právě tehdy, když pro každý stav $q \in Q$ a každé písmeno $a \in \mathcal{A}$ definuje funkce δ přechod $\delta(q, a)$.

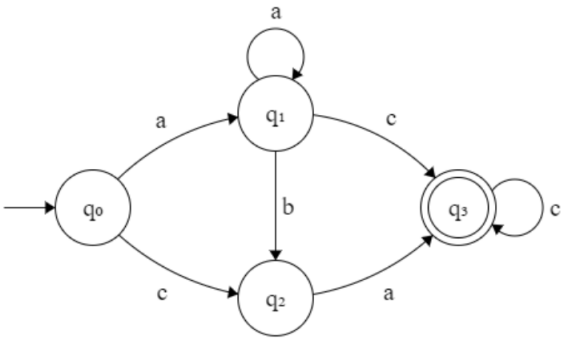
Každý deterministický konečný automat lze přitom tzv. zúplnit. Stačí dodefinovat stav q_{dump}, $q_{dump} \in (Q - F)$ a pro každou dvojici (q_i, a), pro niž funkce δ nedefinuje přechod, doplnit přechod ve tvaru $\delta(q_i, a) = q_{dump}$. Tento postup zabere v nejhorším případě $O(mn)$ kroků, kde $|Q| = m$, $|\mathcal{A}| = n$.

Původní verze knihovny libFAUDES nedisponuje funkcí ověřující úplnost generátoru (metody IsComplete() a Complete() třídy Generator existují, ale váží se ke zcela odlišné vlastnosti).

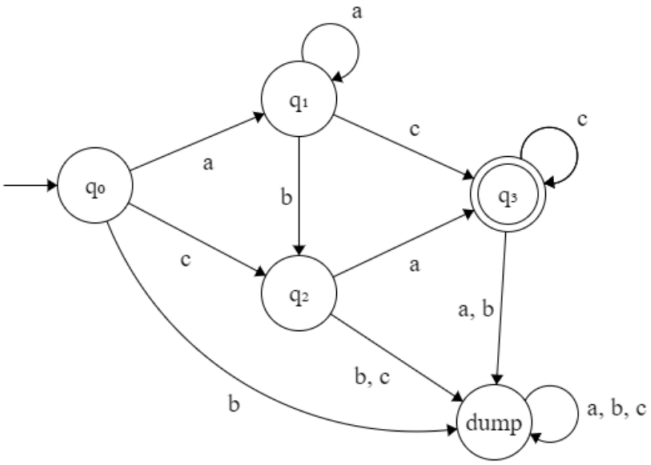
V souboru pwt_checkinput.cpp je definována funkce **IsAutomaton()** ověřující, zda je vstupní generátor automatem. U deterministického generátoru stačí ověřit, je-li počet jeho přechodů roven maximálnímu možnému počtu (tzn. součinu velikosti abecedy a stavové množiny).

Zúplnění generátoru provádí metoda **Automaton**. Název funkce odkazuje na alternativní definici konečného automatu oproti 3, podle níž musí být přechodová funkce δ úplná neboli definovaná pro každou dvojici stavu a písmene. Není-li vstupní generátor úplný, funkce vytvoří speciální stav q_x (tzv. *dump-state*) a

následně pro každou dvojici stavu $q \in Q$ a události $a \in \Sigma$, pro niž neexistuje přechod $\delta(q, a)$, vytvoří přechod ve tvaru $\delta(q, a) = q_x$.



(a)



(b)

Obrázek 14: Příklad automatu (a), který není úplný, a jeho zúplněné verze (b).

4.4 Pomocné datové struktury

Orientovaný graf lze v programovacím jazyce C++ reprezentovat několika různými způsoby, z nichž nejhojněji využívanými jsou matice sousednosti a seznam sousednosti. Přestože s objektem typu Generator je možné v knihovně libFAUDES do určité míry pracovat jako s grafem, pomocné datové struktury nabízejí zjednodušený pohled na automat coby na graf a usnadňují tak implementaci grafových algoritmů. Jejich využití za cenu mírného zhoršení časové složitosti programu zvyšuje celkovou přehlednost kódu.

4.4.1 Matice sousednosti

Matice sousednosti pro automat $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ je tabulka o rozměrech $m \times n$, kde m je velikost abecedy Σ a n je počet stavů automatu neboli velikost

množiny Q. Z hlediska implementace se jedná o dvojrozměrné pole typu unsigned int.

Na i-tém řádku v j-tém sloupci matice se nachází index stavu, který je výsledkem aplikace přechodové funkce δ na stav s indexem i a písmeno s indexem j.

Pokud pro i-tý stav a j-té písmeno přechod neexistuje, nachází se na této pozici nedefinovaná hodnota, jejíž konkrétní podoba závisí na zvolené implementaci. Při práci s knihovnou libFAUDES se nabízí použít hodnotu 0 reprezentující neplatný index, jelikož množiny implementované potomky třídy IndexSet jsou indexovány od 1.

	a	b	c	d
q_0	q_1	q_4		q_0
q_1	q_1	q_4	q_2	
q_2				
q_3		q_3	q_2	q_1
q_4	q_3			

	1	2	3	4
1	2	5	0	1
2	2	5	3	0
3	0	0	0	0
4	0	4	3	2
5	4	0	0	0

Obrázek 15: Grafické znázornění matice sousednosti reprezentující graf 1 (vlevo) a podoba této matice implementované třídou AdjacencyMatrix, která využívá pro reprezentaci stavů a písmen číselné indexy.

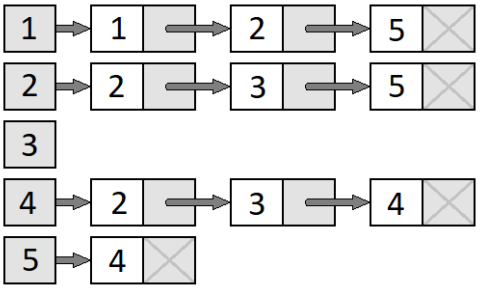
Třída **AdjacencyMatrix** a její metody jsou deklarovány v hlavičkovém souboru **adjmatrix.h** a definovány v souboru **adjmatrix.cpp**.

4.4.2 Seznam sousednosti v orientovaném grafu

Jedná se o reprezentaci orientovaného grafu příslušného automatu $\mathcal{A}$ polem seznamů. Velikost n tohoto pole je rovna počtu stavů automatu $\mathcal{A}$. Na i-té pozici pole se nachází seznam následovníků stavu s indexem i. Délka každého z těchto seznamů se pohybuje mezi 0 a n (každý stav má nejvýše n následovníků a žádný následovník se v seznamu neopakuje).

Z orientovaného seznamu sousednosti oproti struktuře 4.4.1 není možné vyčíst, jak vypadají jednotlivé přechody určené funkcí δ. Bere v potaz pouze informaci, zda nějaký stav q_j je následovníkem stavu q_i či nikoliv. Takové zjednodušení si můžeme dovolit. Dokonce nám usnadňuje práci, jelikož při hledání komponent ani cyklů v grafu nepotřebujeme znát konkrétní podobu všech jeho hran.

Třída **AdjacencyList** a její metody jsou deklarovány v hlavičkovém souboru **adjlist.h** a definovány v souboru **adjlist.cpp**.



Obrázek 16: Graf 1 reprezentovaný seznamem sousednosti.

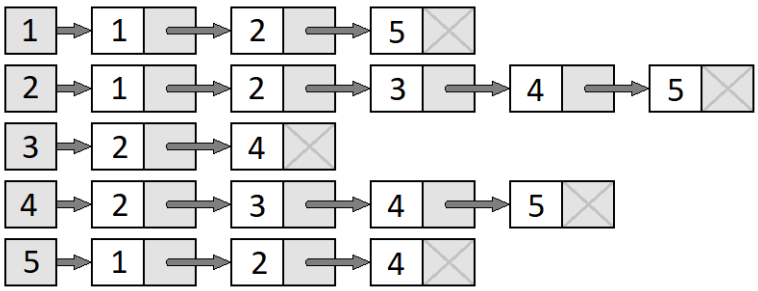
4.4.3 Seznam sousednosti v neorientovaném grafu

Tato datová struktura funguje na podobném principu jako 4.4.2 s tím rozdílem, že namísto orientovaného grafu příslušného automatu $\mathcal{A}$ reprezentuje jeho neorientovanou kopii. Opět se jedná o pole seznamů, ovšem na i-té pozici pole se nachází seznam všech *sousedů* stavu s indexem i, tedy předchůdců i následovníků.

Implementace této struktury využívá skutečnosti, že libovolnému neorientovanému grafu lze vytvořit ekvivalentní orientovaný graf. Každou hranu $\{v_i, v_j\}$ neorientovaného grafu nahradíme dvojicí orientovaných hran $\langle v_i, v_j \rangle$, $\langle v_j, v_i \rangle$. Tento postup nám umožňuje simulovat neorientovaný graf orientovaným bez nutnosti vytvářet novou třídu.

Tato datová struktura je využita v algoritmech 3.2 a 3.4 a jejím účelem je usnadnění hledání slabě souvislé komponenty grafu.

Třída **AdjacencyListUndirected** reprezentující tuto datovou strukturu je spolu se svými metodami deklarována v hlavičkovém souboru **adjlistud.h** a definována v souboru **adjlistud.cpp**.



Obrázek 17: Symetrizace grafu 1 reprezentovaná seznamem sousednosti.

5 Srovnání algoritmů

5.1 Porovnání asymptotických časových složitostí

V kapitole 3 byl určen horní odhad časové složitosti v nejhorším případě pro každý ze tří implementovaných algoritmů. Tato hodnota nám poskytuje řádovou

představu o chování algoritmu v praxi, která se sice může od reality v závislosti na konkrétní implementaci a na různých typech vstupů lišit, přesto je velice užitečná.

Výpočet časové složitosti je vázán na dva z parametrů generátoru: velikost n stavové množiny Q a velikost m abecedy Σ. S těmito hodnotami souvisí i velikost přechodové množiny, která se pohybuje v intervalu $\langle 0; mn \rangle$ (v úplném automatu je pak vždy rovna $m \cdot n$).

Z hlediska tohoto odhadu je zdánlivě nejpomalejší algoritmus Klímy a Poláka, který pracuje v kvadratickém čase vzhledem k oběma parametrům. Oproti němu Trahtmanův algoritmus a algoritmus Klímy, Kunce a Poláka běží v kvadratickém čase pouze vzhledem k jednomu parametru - v prvním jmenovaném algoritmu je tímto parametrem počet stavů n, v druhém pak velikost abecedy m. Lze tedy předpokládat, že výběr optimálního algoritmu závisí na poměru velikosti abecedy a stavové množiny algoritmu: je-li velikost abecedy výrazně větší než velikost stavové množiny, nabízí se využít Trahtmanův algoritmus, v opačném případě pak algoritmus Klímy, Kunce a Poláka.

5.2 Způsoby hledání cyklu

Ve všech třech představených algoritmech je prvním krokem hledání netriviálního cyklu v grafu reprezentujícím vstupní automat. Není-li tento graf acyklický, jazyk rozpoznávaný automatem nemůže být po částech testovatelný a program ihned vrací hodnotu *false*.

Hledání cyklu je možné uskutečnit několika způsoby, které fungují na stejném principu prohledávání grafu do hloubky. V zásuvném modulu je pro každý algoritmus na ukázku implementován jiný z těchto způsobů. Hlavní rozdíl mezi nimi spočívá v odlišném využití pomocných datových struktur. Jedná se o:

- přímé hledání cyklu v generátoru (funkce **IsAcyclic()** definovaná v souboru **isacyc.cpp**)
- vytvoření seznamu sousednosti z matice sousednosti
- vytvoření seznamu sousednosti z generátoru

Při vzájemném porovnávání chování algoritmů by ovšem rozdíly v implementaci této části mohly zkreslit celkový výsledek pozorování. Za tímto účelem byl ve všech třech programech jednotně použitý první jmenovaný způsob, který hledá cyklus přímo v generátoru.

Je-li zajištěno, že první krok je u každého ze tří algoritmů totožný, jeho průběh nemá na celkový výsledek srovnávání vliv. Není pro nás tedy přínosné zkoumat chování algoritmů pro automat, jehož graf obsahuje cykly a program se pro něj zastaví dříve, než dokážeme vypozorovat vzájemné odlišnosti mezi algoritmy. Z tohoto důvodu je test omezený pouze na acyklické automaty.

5.3 Praktické porovnání algoritmů

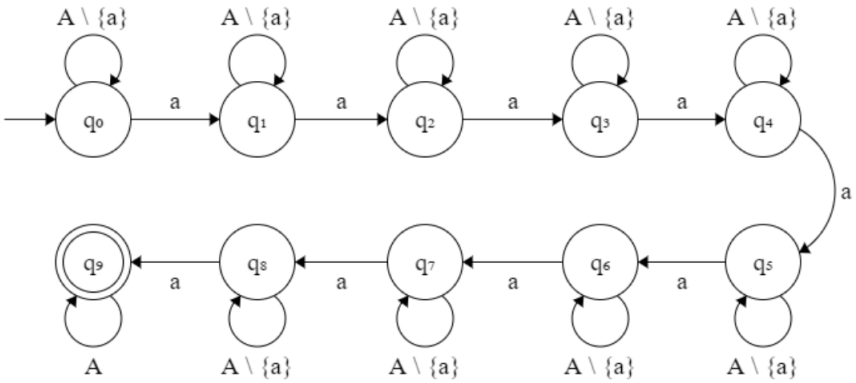
Běh programů byl měřen pomocí nástrojů knihovny **std::chrono** jazyka C++, konkrétně třídy **high_resolution_clock**, která měří čas s přesností řádově na stovky nanosekund (přesná hodnota závisí na konkrétním zařízení a jeho operačím systému).

5.3.1 Závislost časové složitosti na velikosti abecedy

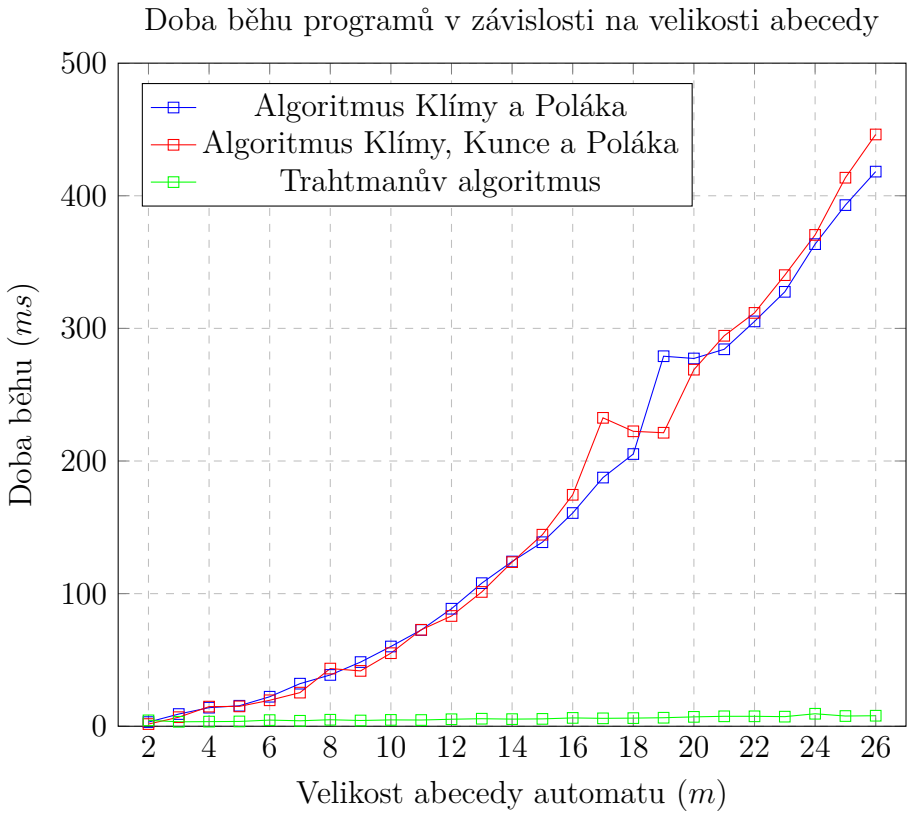
Pro každý ze tří programů implementujících zkoumané algoritmy byla měřena doba běhu na 25 vstupech. Vstupními argumenty jsou objekty typu Generator s konstantním počtem stavů $n = 10$ a velikostí abecedy m pohybující se v intervalu $\langle 2; 26 \rangle$. Obecná podoba generátoru je znázorněna na obrázku 18. Každý z těchto generátorů je deterministický, minimální a úplný.

Nejnižší průměrná doba běhu programu byla naměřena u Trahtmanova algoritmu, a to $5,738$ ms. Z tvaru odpovídající křivky v grafu 5.3.1 lze vyčíst, že tato doba se s rostoucí velikostí abecedy výrazně nezvyšuje (rozdíl maximální a minimální naměřené hodnoty je pouhých $6,063$ mikrosekund). Tento příznivý výsledek souvisí s horním odhadem asymptotické časové složitosti algoritmu, která je vůči hodnotě m lineární, ale je také ovlivněn konkrétní podobou vstupních generátorů.

Pro další dva algoritmy jsou dílčí i průměrné výsledky měření podobné, a to konkrétně 161.802 ms pro algoritmus Klímy, Kunce a Poláka a 159.083 ms pro algoritmus Klímy a Poláka. Tento výsledek také odpovídá vypočítané asymptotické časové složitosti obou algoritmů, která roste kvadraticky vzhledem k parametru m.

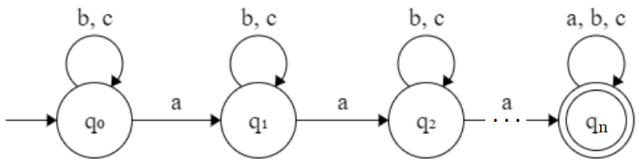


Obrázek 18: Vizualizace obecné podoby vstupního generátoru s abecedou A o proměnlivé velikosti.



5.3.2 Závislost časové složitosti na počtu stavů

Test zkoumající závislost doby běhu programů na velikosti stavové množiny byl uskutečněn stejným způsobem jako test popsaný v předchozí sekci 5.3.1. Byl proveden celkem pro 22 vstupů. Velikost m abecedy $A = \{a, b, c\}$ je pro každý vstupní generátor rovna 3, počet stavů n se pohybuje v intervalu $\langle 3; 24 \rangle$. Obecná podoba generátoru je znázorněna na obrázku 19. Každý jazyk přijímaný tímto generátorem je po částech testovatelný bez ohledu na hodnotu parametru n.

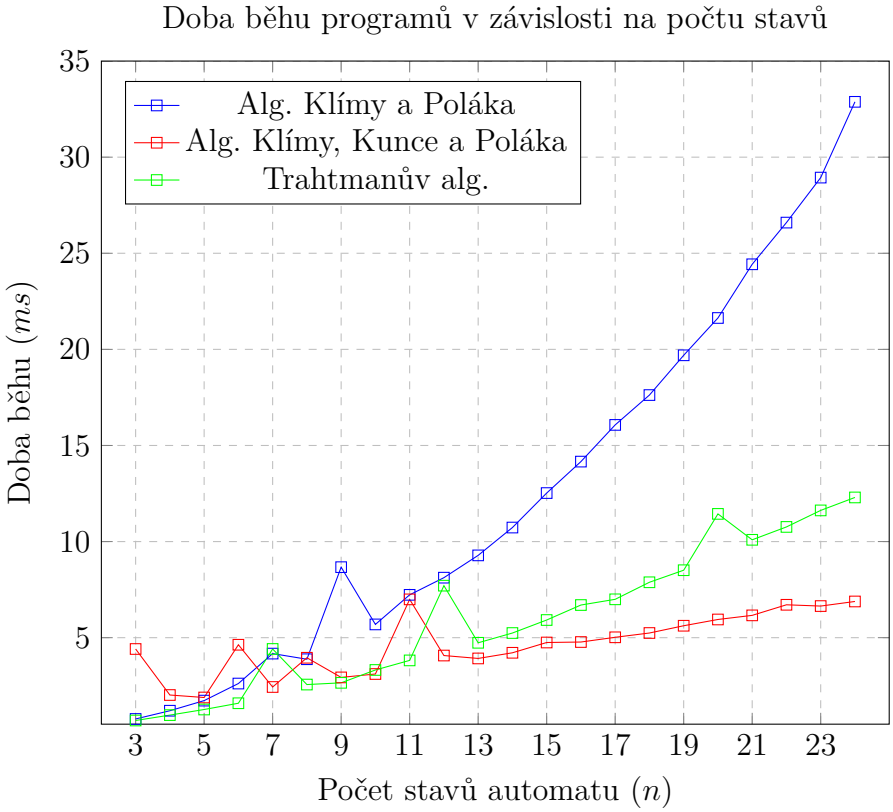


Obrázek 19: Vizualizace obecné podoby vstupního generátoru s n stavy.

Oproti předchozímu testu jsou rozdíly mezi naměřenými časy méně výrazné, a to zejména pro menší počet stavů ($n < 12$).

Průměrně nejnižší doba běhu programu byla naměřena pro algoritmus Klímy, Kunce a Poláka, a to $4,651\ ms$. Tento výsledek odpovídá předpokladu vyplývajícímu z horního odhadu asymptotické časové složitosti, která je u tohoto algoritmu lineární vůči hodnotě n.

Pro Trahtmanův algoritmus je tato hodnota mírně vyšší, konkrétně $5,963$ ms. Pro algoritmus Klímy a Poláka pak byla průměrně naměřena doba trvání $12,665$ ms. Vidíme tedy, že pro rostoucí počet stavů jsou rozdíly mezi algoritmy oproti rostoucí velikosti abecedy generátoru méně výrazné.



Závěr

Hlavními cíli stanovenými na počátku vzniku této práce bylo jednak shrnutí tématu algoritmů ověřujících testovatelnost regulárního jazyka po částech, dále implementace těchto algoritmů a jejich porovnání na teoretické i praktické rovině.

S pomocí příslušných literárních zdrojů byl popsán koncept testovatelnosti jazyka po částech a byl objasněn princip fungování každého z představených algoritmů. Také byl upřesněn vzájemný vztah některých pojmů využívaný různými autory.

Tři z těchto algoritmů byly implementovány v jazyce C++ s využitím knihovny libFAUDES a tato implementace byla následně využita jako základ nového zásuvného modulu této knihovny.

Při praktickém otestování programů s různými vstupy bylo zjištěno, že doba jejich trvání víceméně odpovídá předpokladům stanoveným z vypočítaných asymptotických časových složitostí. Bylo dosaženo závěru, že při výběru optimálního algoritmu je zásadní poměr velikostí stavové množiny a abecedy vstupního automatu.

Z tohoto hlediska lze cíl práce považovat za splněný.

Conclusions

There were two main goals set at the beginning of the creation of this thesis: firstly, it was to summarize the topic of piecewise testable languages and the algorithms recognizing this property. The second goal was to implement three of these algorithms and to compare them both from the theoretical and practical perspective.

Using the relevant literary sources, the concept of piecewise testability was described and the principle of each introduced algorithm was explained. The relationship between certain terms used by different authors was clarified.

Three of the algorithms presented were implemented in C++ using the libFAUDES library. This implementation was then used as the basis of a new plug-in of the library in question.

In the process of running the application with different input data, it was discovered that the duration of each program is more or less consistent with the estimations based on the computed time complexities of the algorithms. The main conclusion reached from this experiment is the fact that the choice of the optimal algorithm depends mostly on the ratio of of the number of states to the size of the alphabet of the input DFA.

In this regard, the purpose of this thesis can be considered as fulfilled.

A Obsah přiloženého CD

`text/`
: Složka obsahující text této práce ve formátu PDF a její zdrojový kód.

`libfaudes/`
: Složka obsahující soubory nejnovější dostupné verze knihovny libFAUDES.

`pwtestability/`
: Složka obsahující všechny zdrojové soubory zásuvného modulu Piecewise testability.

`readme.txt`
: Soubor obsahující instrukce pro integraci pluginu s knihovnou libFAUDES a její instalaci.

Literatura

[1] Alfred Aho, John E. Hopcroft and Jeffrey D. Ullman. *Data Structures and Algorithms*. První vyd. Pearson, 1983. 448 pp. ISBN: 0-201-00023-7.

[2] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Třetí vyd. Pearson Education, 2006. 560 pp. ISBN: 0-321-45536-3.

[3] Ondřej Klíma, Michal Kunc and Libor Polák. „Deciding k-piecewise testability“. In: (2014).

[4] Ondřej Klíma and Libor Polák. „Alternative automata characterization of piecewise testable languages“. In: *Lecture Notes in Computer Science* 7907 (2013), s. 289–300.

[5] Bernd Opitz aj. *libFAUDES*. 2006. URL: https://fgdes.tf.fau.de/faudes/index.html (cit. 08/03/2021).

[6] Imre Simon. „Piecewise testable events“. In: *Lecture Notes in Computer Science* 33 (1975), s. 214–222.

[7] Jacques Stern. „Complexity of Some Problems from the Theory of Automata“. In: *Information and Control* 66.3 (1985), s. 163–176.

[8] A. N. Trahtman. „Piecewise and local testability of DFA“. In: *Lecture Notes in Computer Science* 2138 (2001), s. 347–358.