VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ ALGORITMY PRO KONEČNÉ AUTOMATY

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ KANTOR**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ONDŘEJ LENGÁL, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kantor Tomáš**
Program: Informační technologie
Název: **Efektivní algoritmy pro konečné automaty**
Efficient Algorithms for Finite Automata
Kategorie: Teoretická informatika

Zadání:

1. Prostudujte současný stav poznání v oblasti efektivní práce s konečnými automaty.
2. Po konzultaci s vedoucím zvolte vhodný problém z oblasti konečných automatů na jehož optimalizaci se zaměříte. Uvažujte hlavně problémy komplementace, jazykové inkluze, univerzality či průniku.
3. Navrhněte efektivní algoritmus pro řešení zvoleného problému.
4. Algoritmus navržený v předchozím bodě implementujte a ověřte na vhodně zvolené sadě automatů.
5. Diskutujte dosažené výsledky.

Literatura:

- ABDULLA Parosh A., HOLÍK Lukáš, CHEN Yu-Fang, MAYR Richard a VOJNAR Tomáš. When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In: TACAS'10, LNCS 6015. ISBN 978-3-642-12001-5.
- van Glabbeek R., Ploeger B. (2008) Five Determinisation Algorithms. In: CIAA'08, LNCS 5148. https://doi.org/10.1007/978-3-540-70844-5_17

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

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

Vedoucí práce: **Lengál Ondřej, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 30. července 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce prezentuje nový algoritmus pro komplmentaci nedeterministických konečných automatů. Současné metody vyžadují převod na deterministický konečný automat, který může mít exponenciálně větší počet stavů. Navržený algoritmus pracuje iterativně po jednotlivých silně souvislých komponentách konečného automatu. Díky tomu umožňuje jednotlivé části efektivněji komplementovat, redukovat a následně propojit s ostatními částmi automatu. Tento algoritmus je tak efektivnější než současné metody pro určité typy konečných automatů.

Abstract

This thesis presents new algorithm for complement of nondeterministic finite automata. State-of-the-art methods require conversion to deterministic finite automata, which can have exponentially larger number of states. This new algorithm works separately on each strongly connected component of finite automata. This approach allows to create complement of each component, reduce it and combine with other parts. This algorithm was proven to create less states for specific types of finite automata than existing methods.

Klíčová slova

konečné automaty, nedeterministické konečné automaty, doplněk, antichain

Keywords

finite automata, nondeterministic finite automata, complement, antichain

Citace

KANTOR, Tomáš. *Efektivní algoritmy pro konečné automaty.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Lengál, Ph.D.

Efektivní algoritmy pro konečné automaty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Lengála, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Tomáš Kantor
26. července 2021

Poděkování

Rád bych poděkoval Ing. Ondřeji Lengálovi, Ph.D. za vedení v celém průběhu práce, za cenné praktické rady a za detailní zpětnou vazbu k textu práce. Také bych rád poděkoval Mgr. Lukáši Holíkovi, Phd. doc. RNDr. Janu Strejčkovi, Ph.D. a Mgr. Juraji Majorovi za pravidelné konzultace.

Obsah

Kapitola 1

Úvod

Konečné automaty jsou jednoduché výpočetní modely v teoretické informatice, které se skládají ze stavů a přechodů mezi nimi. Vstupem konečného automatu je posloupnost znaků. Jejich úkolem je rozhodnout, zda vstup je přijímán, nebo je odmítnut. Přestože se jedná o teoretické modely, mají široké praktické využití napříč mnoha současnými obory v oblasti informačních technologií. Mezi nejčastější případy jejich použití patří lexikální analýza v překladačích kompilovaných programovacích jazyků a v interpretech interpretovaných programovacích jazyků. Lexikální analýza ověří, zda textový soubor obsahuje korektně zapsaný programovací jazyk. Dále je lze využít při tzv. pattern matching, neboli proces rozhodování, zda daná posloupnost znaků odpovídá určitému vzoru, např. hledání slov v textu, ověření tvaru emailové adresy nebo telefonního čísla. Mezi další využití konečných automatů patří reprezentace systémů při formální verifikaci. Díky tomu je možno ověřit, zda chování systémů odpovídá specifikaci anebo mohou nastat nežádoucí stavy.

Komplement je jedna ze základních operací prováděných nad konečnými automaty. Jedná se o operaci reprezentující negaci v rozhodovacích procedurách pro logiky založené na automatech. Komplement automatu má opačné chování oproti původnímu automatu. Všechny posloupnosti znaků, které originální automat přijímá, jeho komplement odmítá a naopak. Jedna z aplikací této operace je testování inkluze dvou jazyků, které jsou vyjádřeny pomocí konečných automatů. Inkluze znamená, že jeden jazyk je podmnožinou druhého jazyka. Je využito faktu, že platí $L(A_1) \subseteq L(A_2) \Leftrightarrow L(A_1) \cap L(\overline{A_2}) = \emptyset$. Tento zápis znamená, že inkluze dvou jazyků je ekvivalentní prázdnosti průniku jednoho jazyka a komplementu druhého jazyka. Jedno z dalších využití komplementu je tzv. model checking, neboli testování modelu, kde komplement automatu popisující korektní chování daného modelu reprezentuje chybové stavy.

Současné algoritmy pro vytváření komplementu automatů vyžadují převod obecného konečného automatu na deterministický konečný automat. Tato operace má mnohdy za následek, že počet stavů automatu se exponenciálně zvětší. Tím se zvyšují časové i paměťové nároky programů, které tyto operace využívají. Cílem této práce je navrhnout, implementovat a otestovat algoritmus, který produkuje menší komplementy automatů než současné metody. Zbytek práce je strukturován následujícím způsobem. V kapitole 2 jsou definovány základní pojmy z oblasti teorie konečných automatů a teorie grafů. V kapitole 3 jsou popsány obě současné metody pro komplementaci konečných automatů, jejich vlastnosti, výhody i nevýhody. V kapitole 4 je uveden popis navrženého algoritmu. Hlavní myšlenkou algoritmu je rozdělit automat do několika části. Tyto části budou individuálně zpracovány za pomoci existujících metod pro komplementy konečných automatů. Tyto jednotlivé komponenty budou v poslední fázi skombinovány ve výsledný komplement. Jednou

ze silných stránek tohoto přístupu po částech je možnost tyto komponenty za běhu minimalizovat a zvolit výhodnější metodu. Kapitola 5 informuje o postupu implementace, využitých knihovnách, postupu testování a dosažených výsledcích.

Kapitola 2

Základní pojmy

Tato kapitola definuje a popisuje základní pojmy, které se vyskytují v rámci celé práce a které jsou pro její pochopení jsou klíčové. Jedná se o definice z oblastí teorie formálních jazyků, konečných automatů a teorie grafů. Definice byly převzaty z [12, 3].

2.1 Formální jazyky

Abeceda je definována jako konečná neprázdná množina symbolů. Zpravidla se značí znakem Σ.

Řetězec je definován následovně: ϵ je prázdný řetězec. Pokud $a \in \Sigma$ a zároveň R je řetězec, pak i aR je řetězec.

Reverzace řetězce x nad abecedou Σ, $reverse(x)$ je:

- $reverse(x) = \epsilon$ pro $x = \epsilon$.
- $reverse(x) = a_n...a_1$ pro $x = a_1...a_n$, kde $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1...n$.

Konkatenace neboli sřetězení dvou řetězců x a y je xy, např. konkatenace řetězců *aba* a *caa* je řetězec *abacaa*.

Σ^* je množina všech řetězců nad abecedou Σ včetně prázdného řetězce. Množina L je **jazykem** nad abecedou Σ pokud platí $L \subseteq \Sigma^*$.

Reverzace jazyka L je $L^R = \{x; reverse(x) \in L\}$. Jinými slovy, reverzace jazyka je jazyk obsahující všechny reverzace řetězců původního jazyka.

Pokud L_1 a L_2 jsou dva jazyky, pak jejich **konkatenace** je jazyk $L_1 \cdot L_2 = \{xy; x \in L_1 \wedge y \in L_2\}$

Jazyk L_2 je **komplementem jazyka** L_1 nad abecedou Σ pokud platí $L_2 = \Sigma^* \setminus L_1$. Tedy L_2 obsahuje všechny řetězce, které lze vytvořit nad touto abecedou, ale neobsahuje žádné řetězce z jazyka L_1. Komplement jazyka L_1 se značí $\overline{L_1}$.

Regulární jazyky jsou definovány takto:

- $\emptyset$ je regulární jazyk,
- $\{a\}$, pro $a \in \Sigma$ je regulární jazyk,
- $\{\epsilon\}$ je regulární jazyk,
- pokud L_1 a L_2 jsou regulární jazyky pak i L_1^*, $L_1\cdot, L_2$ a $L_1 \cup L_2$ jsou regulární jazyky,
- nic jiného regulární jazyk není.

Regulární výraz značící regulární jazyk na abecedou Σ je definován následovně:

- $\emptyset$ je regulární jazyk značící prázdný jazyk,
- ϵ je regulární jazyk značící jazyk $\{\epsilon\}$,
- a, kde $a \in \Sigma$ je regulární jazyk jazyk $\{a\}$,
- pokud r a s jsou regulární výrazy, které značí jazyky L_r a L_s, pak:
  - $(r.s)$ je regulární výraz, který značí jazyk $L = L_r \cdot L_s$,
  - $(r + s)$ je regulární výraz, který značí jazyk $L = L_r \cup L_s$,
  - (r^*) je regulární výraz, který značí jazyk $L = L_r^*$.

2.2 Teorie konečných automatů

Konečný automat, dále jen KA, je teoretický výpočetní model schopný rozhodnout, zda daný řetězec patří do regulárního jazyka. Pro každý regulární jazyk existuje odpovídající konečný automat. KA je pětice

$$M = (Q, \Sigma, R, S, F), \tag{2.1}$$

kde

- Q je konečná množina stavů,
- Σ je vstupní abeceda,
- $R \subseteq Q \times \Sigma \times Q$ je konečná množina přechodů tvaru $q_1 \xrightarrow{a} q_2$, kde $q_1 \in Q, a \in \Sigma, q_2 \in Q$,
- $S \subseteq Q$ je množina počátečních stavů,
- $F \subseteq Q$ je množina koncových stavů.

Konfigurace konečného automatu $M = (Q, \Sigma, R, S, F)$ je řetězec $x \in Q \times \Sigma^*$. Jedná se o kombinace stavu automatu a řetězce nad abecedou Σ.

Nechť pax a qx jsou dvě konfigurace konečného automatu M, kde $p, q \in Q, a \in \Sigma, x \in \Sigma^*$. Pokud existuje přechod $r = p \xrightarrow{a} q \in R$, pak automat M může provést **krok** z pax do qx. Zapsáno $pax \vdash qx[r]$ nebo zjednodušeně $pax \vdash qx$.

Nechť $X_0, X_1, ..., X_n$ je **sekvence kroků** konfigurací pro $n \geq 1$ a $X_{i-1} \vdash X_i[r_i]$, $r_i \in R$ pro všechna $i = 1, ..., n$, což znamená: $X_0 \vdash X_1[r_1] \vdash X_2[r_2] ... \vdash X_n[r_n]$. Pak M provede n přechodů z X_0 do X_n; zapisujeme: $X_0 \vdash^n X_n[r_1 ... r_n]$ nebo zjednodušeně $X_0 \vdash^n X_n$.

Jazyk přijímaný konečným automatem M, $L(M)$, je definován následujícím způsobem: $L(M) = \{w \in \Sigma^*; sw \vdash^* f, f \in F, s \in S\}$. Jinými slovy, jazyk přijímaný automatem M je množina řetězců, pro které existuje sekvence kroků od iniciálního stavu po stav koncový.

Deterministický konečný automat je takový konečný automat, pro který platí, že pro každé $a \in \Sigma$ a každé $q_1 \in Q$ existuje v R maximálně jeden přechod $q_1 \xrightarrow{a} q_2$. Jinými slovy, pro každý stav a každý možný symbol na vstupu existuje maximálně jeden stav, do kterého lze přejít. Každý nedeterministický konečný automat lze převést na deterministický automat přijímající stejný jazyk, viz sekce 3.4.

Úplný konečný automat je takový konečný automat, pro který platí, že pro každé $a \in \Sigma$ a každé $q_1 \in Q$ existuje v R alespoň jeden přechod $q_1 \xrightarrow{a} q_2$. V každém stavu a pro každý symbol na vstupu lze přejít do alespoň jednoho stavu.

Jazyk stavu je množina všech řetězců, pro které existuje posloupnost kroků z daného stavu do některého z koncových stavů. Pro stav q platí: $L(q) = \{w \in \Sigma^*; qw \vdash^* f, f \in F, s \in S\}$.

Simulace nad automatem $M = (Q, \Sigma, R, S, F)$ je relace $\preceq \subseteq Q \times Q$ taková, že platí $p \preceq r$, pouze pokud $p \in F \implies r \in F$ a pro každý přechod $p \xrightarrow{a} p' \in R$ existuje přechod $r \xrightarrow{a} r' \in R$ takový, že platí $p' \preceq r'$[2]. Stav p je simulován stavem r, pokud ze stavu r vedou přechody přes všechny symboly jako ze stavu p a tyto přechody vedou do stavů, které jsou také simulovány. Navíc, pokud je stav p koncový, pak i stav r, který ho simuluje, musí být koncový. Díky těmto vlastnostem platí $p \preceq r \implies L(p) \subseteq L(r)$.

(a) Stav p (b) Stav r

Obrázek 2.1: Příklad simulace, stav p je simulován pomocí stavu r

Na Obrázku 2.1 je příklad stavu p, který simuluje stav r. Přechod $r \xrightarrow{b} r_1$ je simulován pomocí přechodu $p \xrightarrow{b} p_1$ a také platí, že stav r_1 je simulován pomocí stavu p_1, jelikož stav r_1 nemá žádné přechody. Přechod $r \xrightarrow{a} r_2$ je simulován pomocí přechodu $p \xrightarrow{a} p_2$. Také platí, že stav r_2 je simulován pomocí stavu p_2. Protože je stav r_2 koncový, tak i stav p_2 musí být koncový, aby jej mohl simulovat. Přechody, které jsou navíc ($p \xrightarrow{a} p_3$ a $p_2 \xrightarrow{c} p_3$) nemají na simulaci stavu r žádný vliv.

2.3 Teorie grafů

Orientovaný graf D je uspořádaná dvojice $(V(D), E(D))$, kde $V := V(D)$ je množina vrcholů a $E := E(D)$ je množina hran různá od $V(D)$. Incidenční relace ψ_D přiřazuje každé hraně z $E(D)$ uspořádanou dvojici vrcholů (ne nutně různých) z $V(D)$.

Graf F je **podgraf** orientovaného grafu G, pokud $V(F) \subseteq V(G), E(F) \subseteq E(G)$ a ψ_F je omezením ψ_G do $E(F)$. Toto zapisuje $F \subseteq G$.

Cesta je graf s množinou vrcholů $V = \{x_1, x_2, \ldots, x_n\}$ a množinou hran $E = \{x_1x_2, x_2x_3, \ldots, x_{n-1}x_n\}$. Cesty značíme P_n Pokud je V jednoprvková množina , tak E neobsahuje žádnou hranu a cesta P_1 se nazývá *triviální* cesta.

Maximální silně souvislá komponenta je podgraf orientovaného grafu, který splňuje následující vlastnost. Z každého vrcholu tohoto podgrafu vede cesta do všech ostatních vrcholů a zároveň tento podgraf není součástí žádné jiné větší souvislé komponenty. Každý orientovaný graf lze rozložit na maximální silně souvislé komponenty pomocí např. Tarjanova algoritmu [15]. Orientovaný graf maximální silně souvislých komponent neobsahuje cykly.

Kapitola 3

Základní algoritmy nad konečnými automaty

Tato kapitola popisuje současné algoritmy pro práci s konečnými automaty. Obsahem jsou algoritmy pro sjednocení, průnik a konkatenaci dvou automatů, převod nedeterministického konečného automatu na deterministický, algoritmus pro komplement a algoritmus pro minimalizaci konečného automatu.

3.1 Sjednocení

Sjednocení dvou množin A a B, které se zapisuje $A \cup B$ je množina, obsahující prvky, které náleží množině A nebo B. $A \cup B = \{x; x \in A \vee x \in B\}$ [16].

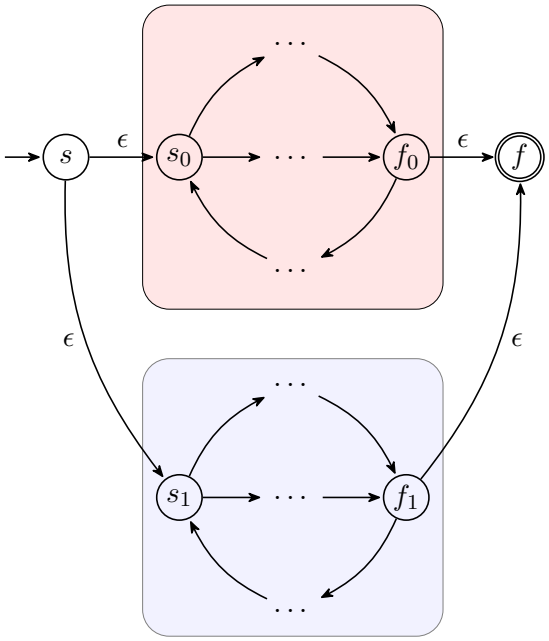
Následující algoritmus vytvoří ze dvou automatů M_0 a M_1 takový automat M, který splňuje $L(M) = L(M_0) \cup L(M_1)$. Je zapotřebí, aby automaty M_1 a M_2 neměly žádné společné stavy, formálně: $(Q_0 \cap Q_1 = \emptyset)$. Pokud tomu tak je, je zapotřebí tento společný stav v jednom z automatů přejmenovat [12].

Algorithm 1: Sjednocení dvou konečných automatů

Input: $M_0 = (Q_0, \Sigma_0, R_0, S_0, F_0), M_1 = (Q_1, \Sigma_1, R_1, S_1, F_1)$
Output: $M = (Q, \Sigma, R, S, F)$

1 $Q := Q_0 \cup Q_1$
2 $\Sigma := \Sigma_0 \cup \Sigma_1$
3 $R := R_0 \cup R_1$
4 $S := S_0 \cup S_1$
5 $F := F_0 \cup F_1$
6 **return** (Q, Σ, R, S, F)

Automat, který vznikne pomocí tohoto algoritmu má jeden iniciální stav s, ze kterého vedou epsilon přechody do iniciálních stavů původních automatů M_0 a M_1. Pro každý koncový stav z původních dvou automatů existuje přechod do nového koncového stavu. Tento automat přijímá slova z obou jazyků $L(M_0)$ a $L(M_1)$ a žádná jiná.



Obrázek 3.1: Vizualizace algoritmu pro sjednocení dvou automatů

3.2 Konkatenace

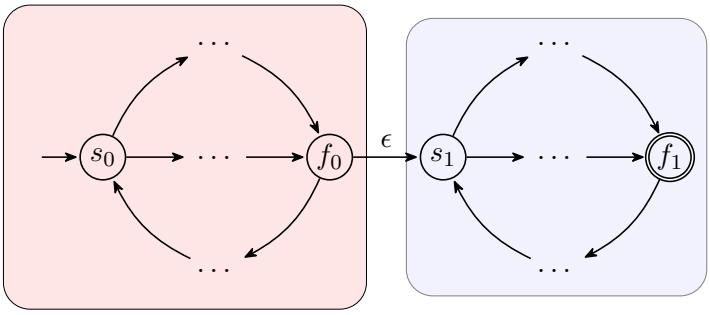
Každý řetězec v jazyku $L = L_1 \cdot L_2$ se skládá ze dvou částí, kde první část je z jazyka L_1 a druhá z jazyka L_2. Pokud máme dva automaty M_1 a M_2, pak můžeme vytvořit automat M, který splňuje $L(M) = L(M_1) \cdot L(M_2)$ pomocí následujícího algoritmu:

Algorithm 2: Konkatenace

Input: $M_0 = (Q_0, \Sigma_0, R_0, S_0, F_0), M_1 = (Q_1, \Sigma_1, R_1, S_1, F_1), Q_0 \cap Q_1 = \emptyset$
Output: $M = (Q, \Sigma, R, S, F)$

1 $Q := Q_0 \cup Q_1$
2 $\Sigma := \Sigma_0 \cup \Sigma_1$
3 $S := S_0$
4 $F := F_1$
5 $R := R_0 \cup R_1 \cup \{f \xrightarrow{\epsilon} s; f \in F_0, s \in S_1\}$
6 **return** (Q, Σ, R, S, F)

Automat M, který vznikne pomocí tohoto algoritmu, má stejné iniciální stavy jako automat M_0 a v případě, že se dostane do některého z koncových stavů v M_0, může přejít přes ϵ přechod do iniciálních stavů automatu M_1.



Obrázek 3.2: Vizualizace algoritmu pro konkatenaci dvou automatů

3.3 Průnik

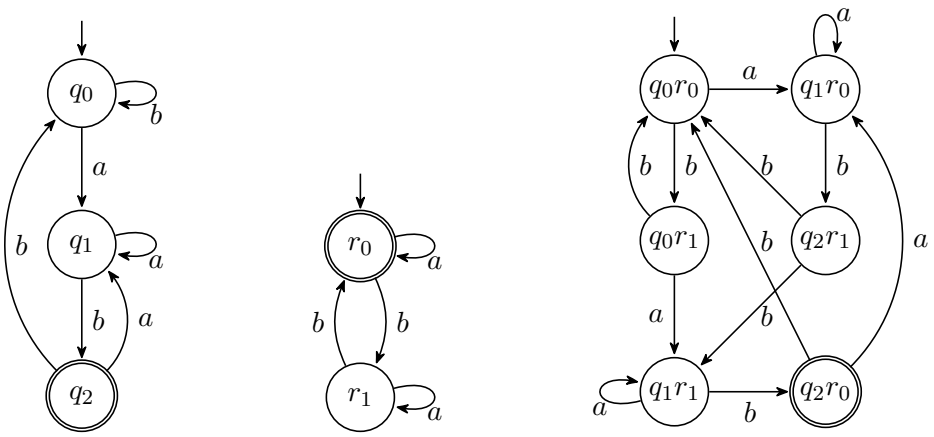
Průnik dvou jazyků je definován následovně: $L_1 \cap L_2 = \{x; x \in L_1 \wedge x \in L_2\}$. Nový jazyk $L = L_1 \cap L_2$ obsahuje všechny řetězce, které se nachází v obou jazycích a žádné jiné. Pokud máme dva deterministické automaty M_0 a M_1 je možno vytvořit automat M splňující $L(M) = L(M_0) \cap L(M_1)$ pomocí následujícího algoritmu:

Algorithm 3: Průnik dvou deterministických konečných automatů

Input: $M_0 = (Q_0, \Sigma_0, R_0, S_0, F_0), M_1 = (Q_1, \Sigma_1 R_1, S, F_1)$
Output: $M = (Q, \Sigma, R, S, F)$

```
1  S := {(s0, s1); s0 ∈ S0 ∧ s1 ∈ S1}
2  Σ := Σ0 ∩ Σ1
3  R := ∅
4  worklist := empty list
5  done := ∅
6  worklist.push((s0, s1))
7  while not worklist.empty() do
8      (q0, q1) := worklist.pop()
9      foreach a ∈ Σ do
10         if ∃q0' ∈ Q0 ∃q1' ∈ Q1 (q0, a, q0') ∈ R0 ∧ (q1, a, q1') ∈ R1 then
11             R := R ∪ {((q0, q1), a, (q0', q1'))}
12             done.insert((q0, q1))
13             if not (q0', q1') ∈ done then
14                 worklist.push((q0', q1'))
15             end
16         end
17     end
18 end
19 F := {(f0, f1); f0 ∈ F0 ∧ f1 ∈ F1}
20 return (done, Σ, R, S, F)
```

Každý stav v automatu M představuje dvojici stavů, kdy první je vždy stav z automatu M_0 a druhý z automatu M_1. Iniciální stav s je vždy dvojice iniciálních stavů (s_0, s_1). Pro každý symbol je pak nalezen přechod do další dvojice stavů. Pokud ze stavu pro daný symbol neexistuje přechod, pak není nový přechod vytvořen. Koncové stavy jsou ty dvojice stavů, kde jsou oba stavy koncové. Průnik dvou automatů lze provést i pro dva nedeterministické automaty.



(a) Automat M_0 (b) Automat M_1 (c) Automat M

Obrázek 3.3: Automat M vznikl průnikem automatů M_0 a M_1

Na Obrázku 3.3 je zobrazen automat M, pro který platí $L(M) = L(M_0) \cap L(M_1)$. Iniciální stav v tomto automatu je q_0r_0, což je dvojice iniciálních stavů v automatech M_0 a M_1. Přes symbol a vede ze stavu q_0 přechod do stavu q_1 a ze stavu r_0 vede přechod do q_0, proto z dvojice stavů q_0r_0 vede přes symbol a přechod do stavu q_1r_0. Podobně pro symbol b existuje přechod z q_0 přechod do stavu q_0 a ze stavu r_0 přechod do r_1, proto ze stavu q_0r_0 vede přes symbol b přechod do q_0r_1. Následně se se stejným postupem vytvoří přechody pro nové stavy q_1r_0 a q_0r_1. Stav q_2r_0 je tak koncovým stavem, protože oba stavy q_2 i r_0 jsou koncové.

3.4 Převod na deterministický automat

Nedeterministické konečné automaty mohou obecně mít přechod přes jeden symbol do více stavů. Z toho vyplývá, že takový automat může přejít pod jedním řetězcem do několika různých stavů. Pokud se automat nachází v množině stavů Q a přejde přes symbol a do množiny stavů Q', pak tento krok zapisujeme pomocí $Qa \vDash Q'$. Obdobně sekvence kroků z množiny stavů Q do množiny stavů Q', která čte slovo w, se zapisuje $Qw \vDash^* Q'$. Každý nedeterministický automat lze převést na ekvivalentní deterministický automat pomocí Algoritmu 4 [6].

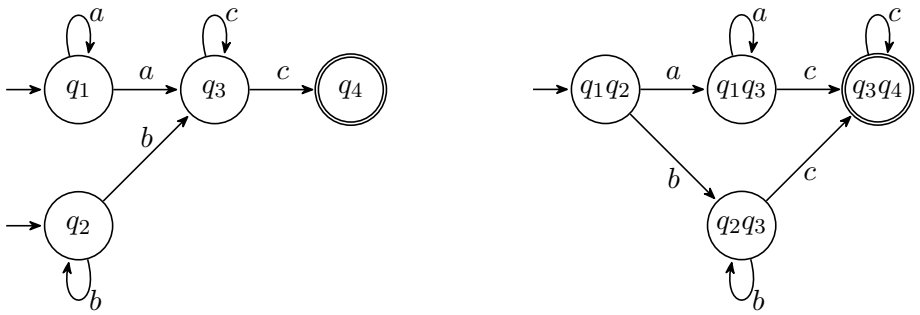
Algorithm 4: Převod na deterministický automat pomocí subset konstrukce

Input: Konečný automat $N = (Q_N, \Sigma_N, R_N, S_N, F_N)$
Output: Deterministický konečný automat $D = (Q_D, \Sigma_D, R_D, S_D, F_D)$, takový že $L(N) = L(D)$

1 $\Sigma_D := \Sigma$; $R_D := \emptyset$; $S_D := f(\{S_N\})$; $F_D := \emptyset$
2 $Q_D := \{S_D\}$ todo $:= \{S_D\}$ done $:= \emptyset$
3 **while** *todo* $\neq \emptyset$ **do**
4 $P :=$ todo.pop()
5 **foreach** $a \in \Sigma_N$ **do**
6 $P' := f(\{p' \in Q_N; \exists p \in P.p \xrightarrow{a} p'\})$
7 $Q_D := Q_D \cup \{P'\}$
8 $R_D := R_D \cup \{(P, a, P')\}$
9 todo $:=$ todo $\cup(\{P'\} \setminus$ done$)$
10 **end**
11 **if** $\exists p \in P.p \in F_N$ **then**
12 $F_D := F_D \cup \{P\}$
13 **end**
14 done $:=$ done $\cup\{P\}$
15 **end**
16 **return** $(Q_D, \Sigma_D, R_D, S_D, F_D)$

Automat D, který vznikne pomocí Algoritmu 4 má jeden iniciální stav, který reprezentuje všechny iniciální stavy v automatu N. Tento stav se umístí do zásobníku *todo*, který obsahuje všechny stavy ke zpracování. Postupně se odebírají ze zásobníku *todo* jednotlivé stavy, které se značí P, a které vždy reprezentují množinu stavů z automatu N. Pro každý symbol $a \in \Sigma_N$ je určena množina stavů $P' \subseteq Q_N$, do které lze přejít přes symbol a. Tato množina P' je následně přidána jako nový stav do automatu D. Pokud množina P obsahuje alespoň jeden koncový stav z automatu D, pak je stav P koncovým stavem v novém automatu. Algoritmus skončí ve chvíli, kdy na zásobníku *todo* už neexistují další stavy ke zpracování. Funkce f na řádcích 1 a 6 je v tomto algoritmu identita. Tato funkce vrací stejnou množinu jakou dostává jako argument. V sekci 3.5 bude funkce identity nahrazena jinou funkcí, která je schopná tento algoritmus optimalizovat.

V nejhorším případě může v automatu D vzniknout stav pro každou podmnožinu stavů v automatu N. Počet stavů v deterministickém automatu D může nanejvýš dosáhnout velikosti potenční množiny původních stavů: $|Q_D| \leq |\mathcal{P}(Q_N)| = 2^{|Q_N|}$.



(a) Automat N (b) Automat D

Obrázek 3.4: Deterministický konečný automat D vznikl z nedeterministického konečného automatu N pomocí Algoritmu 4

Automat N na Obrázku 3.4a je nedeterministický, protože ze stavu q_1 vedou dva přechody přes symbol a, ze stavu q_2 vedou dva přechody přes symbol b a ze stavu q_3 vedou dva přechody přes symbol c. Navíc má tento automat dva iniciální stavy. Tyto dva iniciální stavy tvoří nový iniciální stav q_1q_2 v deterministickém automatu D zobrazeném na Obrázku 3.4b. Ze stavů q_1 a q_2 vedou přechody přes symbol a do stavů q_1 a q_3, proto ze stavu q_1q_2 vede přechod přes symbol a do stavu q_1q_3. Stejný postup se provede i pro přechod $q_1q_2 \xrightarrow{a} q_2q_3$. Pro nové stavy q_1q_3 a q_2q_3 se provede stejná konstrukce. Stav q_3q_4 je koncový stav, jelikož obsahuje stav q_4, který je koncový. Algoritmus 4 by vytvořil i stav $\emptyset$, ze kterého přechody přes každý symbol vedou zpět do něj. A každý stav, který nemá přechod pro některý symbol, by měl přechod právě do stavu $\emptyset$. Pro větší přehlednost tento stav není na Obrázku 3.4a zobrazen.

3.5 Optimalizace převodu na deterministický automat

Subset konstrukci popsanou v sekci 3.4 lze optimalizovat za pomoci simulace. Cílem je přidat do jednotlivých stavů v deterministickém automatu D stavy z nedeterministického automatu N takovým způsobem, že jazyk stavu zůstane zachovaný. Díky tomu některé stavy, které by vytvořila klasická subset konstrukce nevzniknou, protože budou nahrazeny většími stavy. Funkce f v Algoritmu 4 může být nahrazena funkcí $close_{\sqsubseteq}(P)$, která je definována takto [6]:

$$close_{\sqsubseteq}(P) = \{p \in Q_N | p \sqsubseteq P\} \tag{3.1}$$

Jazyková inkluze $p \sqsubseteq P$ platí, pokud platí $L_N(p) \subseteq L_N(P)$. Množina stavů P je v algoritmu nahrazena množinou všech stavů, jejichž jazyk je podmnožinou jazyka $L_N(P)$. Z toho vyplývá, že $close_{\sqsubseteq}(P)$ obsahuje všechny stavy z P a případně některé navíc. Je dokázáno, že tato konstrukce produkuje minimální deterministický automat a tedy neexistuje automat, který má menší počet stavů a přijímá stejný jazyk.

Výpočet $close_{\sqsubseteq}(P)$ ovšem běží v exponenciálním čase, a je proto velmi nepraktický. Jazykové uspořádání $\sqsubseteq$ je možné nahradit pomocí simulace $\preceq$, protože platí $p \preceq r \implies p \sqsubseteq r$. Přístup za pomoci simulace neprodukuje vždy minimální deterministický automat, ale je výpočetně nenáročný, a proto v praxi lépe uplatnitelný.

3.6 Minimalizace

Pro daný regulární jazyk L existuje nekonečné množství různých automatů, které tento jazyk přijímají. Přitom platí, že pro konkrétní regulární jazyk existuje právě jeden deterministický automat (až na izomorfismus), který má nejnižší počet stavů. O takovém automatu říkáme, že je minimální. Jeden z algoritmů, který dokáže převést automat na minimální automat je Brzozowského minimalizace [5]:

1. Automat M je převeden na automat M^R, který přijímá reverzaci jazyka
2. Automat M^R je převeden na deterministický automat M_D^R, např. pomocí subset konstrukce z Algoritmu 4
3. Automat M_D^R je převeden na automat $(M_D^R)^R$ opět pomocí reverzace
4. Automat $(M_D^R)^R$ je převeden na deterministický automat M' pomocí subset konstrukce

Automat M' přijímá stejný jazyk jako M. První subset konstrukce po reverzaci má za následek, že budou zahozeny všechny stavy, ze kterých nelze provést sekvenci kroků, která by vedla do některého z koncových stavů. Jsou zpětně nedosažitelné z původních koncových stavů. Po druhé subset konstrukci budou zahozeny všechny stavy, které nejsou dosažitelné z iniciálních stavů. Tímto budou odstraněny všechny mrtvé stavy. Druhou podstatnou vlastností subset konstrukce je její schopnost spojit stavy, do kterých lze přejít přes stejné slovo. Pokud existuje dvojice stavů q a q', do kterých lze přejít po přečtení řetězce w, pak budou spojeny během subset konstrukce v jeden stav $\{q, q', \dots\}$. Tímto budou sjednoceny všechny nerozlišitelné stavy.

Kapitola 4

Současné algoritmy pro komplementaci konečných automatů

V současnosti existují dva hlavní přístupy pro komplementaci automatů. V této kapitole jsou oba přístupy popsány včetně jejich hlavních výhod i nevýhod.

4.1 Klasický algoritmus

Komplement konečného automatu lze získat pomocí následujících kroků. Mějme automat $N = (Q_N, \Sigma, R_N, S_N, F_N)$.

1. Automat je převeden na úplný deterministický automat $D = (Q_D, \Sigma, R_D, S_D, F_D)$, např. pomocí subset konstrukce.
2. Automat D je převeden na jeho komplement $C = (Q_D, \Sigma, R_D, S_D, Q_D \setminus F_D)$.

Hlavní nevýhodou tohoto algoritmu je nutnost převodu na deterministický automat, což může v některých případech způsobit exponenciální nárůst počtu stavů. Např. automaty pro regulární výrazy tvaru $(a+b)^*a(a+b)^n$ se vyznačují tím, že jejich minimální deterministický komplement má 2^{n+1} stavů . Tento regulární výraz je zápisem jazyka, který obsahuje pouze řetězce, které na pozici $n+1$ od konce mají symbol x. Je zřejmé, že je potřeba si uchovat informaci o posledních $n+1$ přečtených symbolech. Pro každý symbol ve vstupním slově existují dvě možnosti. Buď se jedná o symbol x, nebo ne. Celkem je tedy 2^{n+1} možností, které je nutné rozlišit. Každá z nich je v deterministickém automatu reprezentována jedním stavem.

4.2 Komplement pomocí reverzace

Tento algoritmus využívá skutečnosti, že platí $\overline{L} = \overline{L^R}^R$. Pokud je z jazyka L vytvořena jeho reverzace A^R, pak je možno tento nový jazyk komplementovat a získat jazyk $\overline{L^R}$, který přijímá komplement reverzace původního jazyka L. Pokud je jazyk $\overline{L^R}$ ještě jednou reverzován, pak vznikne jazyk $\overline{L^R}^R$, který je ekvivalentní jazyku $\overline{L}$. Jak je vysvětleno v podsekci 4.2, reverzací automatu nevznikají žádné nové stavy. A navíc existují případy, kdy

je mnohem výhodnější vytvořit komplement z reverzace automatu A než přímo z automatu A.

Reverzace automatu

Reverzaci automatu získáme následovně. Mějme automat $A = (Q_A, \Sigma_A, R_A, S_A, F_A)$. Automat přijímající reverzaci jazyka automatu A je $A^R = (Q_A, \Sigma_A, R_{A^R}, F_A, S_A)$, kde $R_{A^R} = \{q_2 \xrightarrow{a} q_1; q_1 \xrightarrow{a} q_2 \in R_A\}$.

Stavy, které byly původně koncové, jsou nyní počáteční a stavy, které byly počáteční, jsou nyní koncové. Pokud byl stav počáteční i koncový, zůstává počátečním i koncovým stavem. Každý přechod mezi dvěma stavy se otočí, ale zachovává se symbol, který se čte ze vstupu automatu.

Algoritmus pro komplement

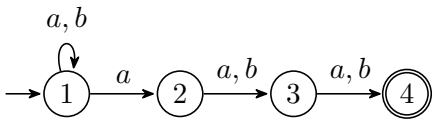
Komplement konečného automatu lze získat pomocí následujících kroků. Mějme automat $N = (Q_N, \Sigma, R_N, S_N, F_N)$ přijímající jazyk L.

1. Automat je převeden na zpětný automat $N^R = reverse(N)$.
2. Automat N^R je převeden na úplný deterministický automat $D = (Q_D, \Sigma, R_D, S_D, F_D)$.
3. Vytvoření automatu $C = (Q_C, \Sigma, R_C, S_C, F_C)$, kde $Q_C = Q_D$, $R_C = R_D$, $S_C = S_D$, $F_C = Q_D \setminus F_D$. Tento automat přijímá jazyk $\overline{L^R}$.
4. Opětovná aplikace operace reverzace.
5. Odstranění nedostupných stavů.

Na rozdíl od předchozího algoritmu má komplement automatu regulárního výrazu $(a + b)^*a(a+b)^n$ mnohem menší počet stavu. Jelikož je reversní automat N^R deterministický, k explozi počtu stavů nedojde.

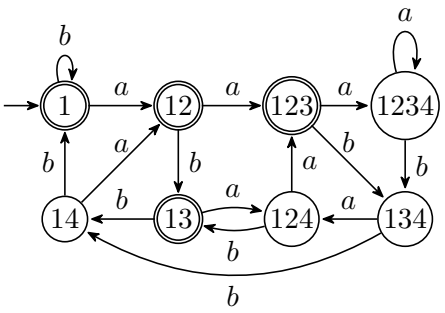
4.3 Srovnání

Rozdíly popsané v předchozích sekcích budou demonstrovány na příkladu automatu pro regulární výraz $(a+b)^*a(a+b)^n$, jehož přechodový diagram je na Obrázku 4.1. Obrázky v této kapitole byly převzaty z [13].



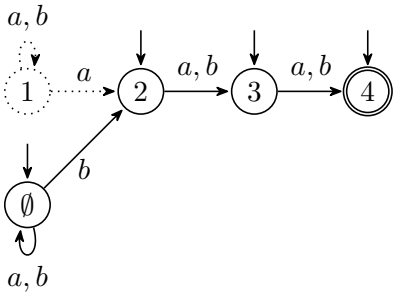
Obrázek 4.1: Konečný automat pro regulární výraz $(a+b)^*a(a+b)^n$

Odpovídající minimální deterministický automat, který přijímá komplement původního jazyka má osm stavů, viz Obrázek 4.2. Pro automaty stejného typu s větším počtem stavů bude tento rozdíl ještě větší.



Obrázek 4.2: Minimální deterministický automat pro komplement regulárního výrazu $(a + b)^*a(a + b)^n$

Komplement, který byl vytvořen pomocí reverzace, má pouze čtyři stavy, viz Obrázek 4.3. Za cenu nedeterminismu je tento algoritmus schopen v některých případech produkovat mnohem menší výsledné automaty.



Obrázek 4.3: Konečný automat pro komplement regulárního výrazu $(a + b)^*a(a + b)^n$ vytvořený pomocí reverzace

Z tohoto jednoho příkladu je možno nabýt mylného dojmu, že vytvoření komplementu pomocí reverzace je vždy výhodnější. Opak je pravdou. Např. pro komplement automatu pro regulární výrazy tvaru $(a+b)^*a(a+b)^n$ je situace přesně opačná. Komplement vytvořený klasickou metodou má menší počet stavů.

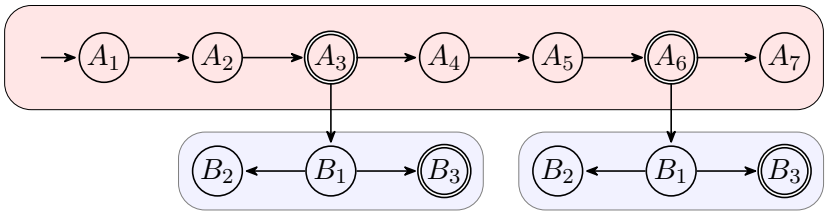
Kapitola 5

Komplement pomocí konkatenace

Tato kapitola se zabývá návrhem algoritmu pro vytvoření automatu, který přijímá komplement jazyka $A \cdot B$, kde A i B jsou regulární jazyky . Tento automat je vytvořen z úplného deterministického konečného automatu A_A, přijímajícího jazyk A a automatu $A_{\overline{B}}$ přijímajícího komplement jazyka B.

5.1 Hlavní myšlenka

Téměř všechny řetězce $w \in \overline{A.B}$ lze rozdělit na dva podřetězce u a v tak, že platí $w = u.v$, $u \in A$ a $v \in \overline{B}$. Jedinou výjimkou jsou takové řetězce w, pro které neexistuje žádný prefix $u \in A$. Pokud automat A_A čte vstupní řetězec a dostane se do koncového stavu, právě přečetl potenciální prefix u; v této chvíli se spustí simulace automatu $A_{\overline{B}}$. Toto nastává pokaždé, když je automat A_A v koncovém stavu. Pokud všechny simulace $A_{\overline{B}}$ po dočtení potenciálních sufixů v skončí v koncových stavech, pak řetězec $w = u \cdot v$ náleží do jazyka $\overline{A \cdot B}$. V každém kroku čtení řetězce se automat A_A nachází právě v jednom stavu. Naproti tomu automat $A_{\overline{B}}$ se nachází v mnoha stavech současně. Důvody jsou dva: automat $A_{\overline{B}}$ může být nedeterministický a může být spuštěno několik kopií současně.



Obrázek 5.1: Vizualizace myšlenky automatu, který přijímá jazyk $\overline{A.B}$

Tato konstrukce je zobrazena na Obrázku 5.1. Červená sekce znázorňuje stavy, ve kterých se nachází automat A_A. Stav A_1 je iniciální a stavy A_3 a A_6 jsou koncové. Pokud se automat nachází v koncovém stavu, jsou spuštěny kopie automatu $A_{\overline{B}}$. Kopie $A_{\overline{B}}$ jsou znázorněny modře. Pokud všechny instance automatu, které byly spuštěny, dojdou do koncového stavu B_3, řetězec náleží do jazyka. Pokud čtení řetězce probíhá takovým způsobem, že automat A_A do koncového stavu nepřejde ani jednou, je řetězec také přijat.

Mějme řetězec $X = x_1x_2x_3x_4$ a dva automaty přijímající jazyky A a $\overline{B}$. Aby se řetězec X nacházel v jazyce $\overline{A.B}$ musí platit toto: pokud se řetězec x_1 nachází v A, pak se řetězec $x_2x_3x_4$ musí nacházet v $\overline{B}$. Pokud se řetězec x_1x_2 nachází v A, pak se řetězec x_3x_4 musí

nacházet v $\overline{B}$. Pokud se řetězec $x_1x_2x_3$ nachází v A, pak se řetězec x_4 musí nacházet v $\overline{B}$. A pokud se řetězec $x_1x_2x_3x_4$ nachází v A, pak se prázdný řetězec ϵ musí nacházet v $\overline{B}$. Pokud se žádný prefix X nenachází v A, pak řetězec X patří do jazyka $\overline{A.B}$. d

5.2 Formální definice

Definice byla převzata z [13]. Pokud úplný deterministický konečný automat $A_A = (Q_A, \Sigma, R_A, S_A, F_A)$ přijímá jazyk A a konečný automat $A_B = (Q_B, \Sigma, R_B, S_B, F_B)$ přijímá komplement jazyka B, pak automat $A_C = (Q_C, \Sigma, R_C, S_C, F_C)$ přijímající jazyk $\overline{A.B}$ lze vytvořit následovně.

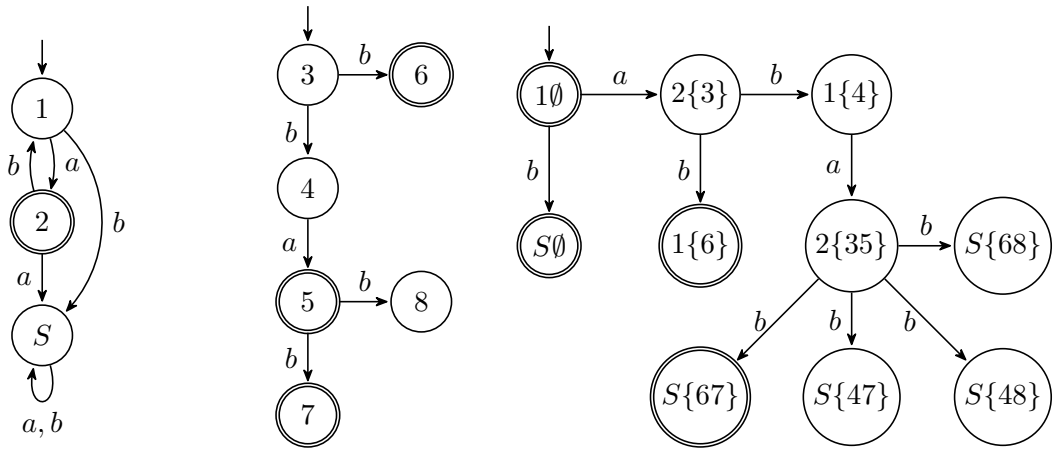
$$
\begin{aligned}
&Q_C = Q_A \times 2^{Q_B},\\
&F_C = Q_A \times 2^{F_B},\\
&S_C = \{(q, \emptyset) \mid q \in S_A \setminus F_A\} \cup \{(q, \{r\}) \mid q \in S_A \cap F_A, r \in S_B\},\\
&R_C((q, R), a) = \begin{cases} \{(R_A(q,a), S) \mid S \in \delta_B^{\times}(R, a)\} & \text{pokud } R_A(q,a) \notin F_A \\ \{(R_A(q,a), S \cup \{r\}) \mid S \in R_B^{\times}(R, a), r \in S_B\} & \text{pokud } R_A(q,a) \in F_A \end{cases}
\end{aligned}
$$

kde $\delta_B^{\times}(R, a)$ je tzv. *multi-product transition function*, která je definována tak, že pro $a \in \Sigma$ a $R \subseteq Q$, kde $R = \{r_1, \ldots, r_n\}$, $\delta_B^{\times}(R, a)$ obsahuje právě množiny $S = \{s_1, \ldots, s_n\}$, kde $s_i \in \delta(r_i, a)$ pro každé $1 \leq i \leq n$.

Je vhodné také podotknout, že může nastat situace $|S| \leq n$ z důvodu, že se některé stavy mohou opakovat. Tato konstrukce řeší i případy, kdy k žádnému přechodu do druhého ze dvou automatů nedojde. Tento algoritmus vyžaduje, aby automat A_A byl úplný a deterministický, pokud není, je možno ho na takovýto automat převést předem. Automat $A_{\overline{B}}$ může být nedeterministický a také libovolně minimalizovaný.

5.3 Příklad

Na Obrázku 5.2 se nachází příklad automatu, který byl vytvořen za pomocí algoritmu definovaného v sekci 5.2. Iniciálním stavem automatu $A_{\overline{A.B}}$ je stav $(1, \emptyset)$, který reprezentuje iniciální stav z prvního automatu a žádný stav z druhého automatu. Tento stav je zároveň i koncový, protože neobsahuje žádné stavy z automatu $A_{\overline{B}}$. Přechod přes symbol a vede do stavu $(2, \{3\})$. Stav 2 je koncový, a proto dochází ke spuštění simulace druhého automatu v jeho iniciálním stavu 3. Ze stavu $(2, \{3\})$ přes symbol b automat přechází do dvou stavů $(1, \{6\})$ a $(1, \{4\})$, protože stav 3 má přechod přes symbol b do dvou stavů 4 a 6. Stav $(1, \{6\})$ je koncový, protože všechny stavy druhého automatu (stav 6) jsou koncové. Přechod $(1, \{4\}) \xrightarrow{a} (2, \{3, 5\})$ vzniká takto: Stav 4 přechází do stavu 5. Stav 1 přechází do stavu 2, který je koncový a je tedy spuštěna druhá simulace automatu $A_{\overline{B}}$ z iniciálního stavu 3. Stav $(2, \{35\})$ není koncový, protože stav 5 není koncový. Ze stavu 3 lze přejít do množiny $M_1 = \{4, 6\}$ a ze stavu 5 lze přejít do množiny $M_1 = \{7, 8\}$. Ze stavu $(2, \{3, 5\})$ pak existují čtyři přechody, kdy každý obsahuje právě jeden prvek z množiny M_1 a jeden z množiny M_2. Toto chování formálně definuje funkce *multi-product transition function*. Z těchto čtyř stavů pouze $(S, \{6, 7\})$ je koncový, protože oba stavy 6 i 7 jsou koncové.

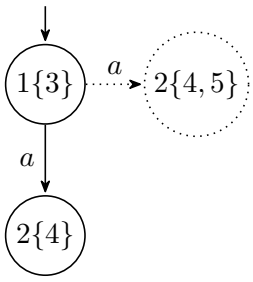


(a) Automat A_A (b) Automat $A_{\overline{B}}$ (c) Automat $A_{\overline{A.B}}$

Obrázek 5.2: Konečný automat přijímající jazyk $\overline{A.B}$ vytvořený z automatů přijímajících A a $\overline{B}$

5.4 Optimalizace little brothers

Algoritmus definovaný v sekci 5.2 je možno poměrně snadno optimalizovat. Pokud máme automaty A_A a $A_{\overline{B}}$ a vytváříme automat $A_{\overline{A.B}}$, pak můžeme využít faktu, že všechny běhy v automatu $A_{\overline{B}}$ musí být přijímající. V případě, že existují dva přechody $(x_0, X_0) \xrightarrow{a} (x_1, X_{1a})$ a $(x_0, X_0) \xrightarrow{a} (x_1, X_{1b})$ a zároveň platí $X_{1a} \subseteq X_{1b}$, pak platí i $L((x_1, X_{1b})) \subseteq L((x_1, X_{1a}))$. Z toho vyplývá, že přechod $(x_0, X_0) \xrightarrow{a} (x_1, X_{1b})$ je možno zahodit. Konkrétní případ je na Obrázku 5.3, pro přechody $(1, \{3\}) \xrightarrow{a} (2, \{4\})$ a $(1, \{3\}) \xrightarrow{a} (2, \{4, 5\})$. Pro přijímaný řetězec w musí platit $w \in L(4) \vee (w \in L(4) \wedge w \in L(5))$. Pokud provedeme substituce $a = L(4)$ a $b = L(5)$, pak je tato formule ekvivalentní formuli $a \vee (a \wedge b)$. Je možno využít zákonu absorpce z Booleovy algebry $a \vee (a \wedge b) = a$ [4]. Jelikož $a = L(4)$, pak celá formule $w \in L(4) \vee (w \in L(4) \wedge w \in L(5))$ je logicky ekvivalentní $w \in L(4)$. Proto je možné přechod $(1, \{3\}) \xrightarrow{a} (2, \{4, 5\})$ zahodit beze změny přijímaného jazyka.



Obrázek 5.3: Příklad optimalizace little brothers

Kapitola 6

Komplement pomocí portů

Komplement pomocí portů je zobecněním algoritmu založeném na konkatenaci v Kapitole 5. Na rozdíl od něj nemá žádné speciální podmínky pro dvě sousedící SCC (silně souvislé komponenty). Obě komponenty mohou obsahovat iniciální i koncové stavy. Zároveň mezi nimi může existovat více než jeden přechod z jedné do druhé SCC.

6.1 Formální definice

Definice byla převzata z [13].

Nechť $A = (Q_A, \delta_A, I_A, F_A)$ je konečný automat nad abecedou Σ. $\mathcal{D} \subseteq 2^{Q_A}$ je *dekompozicí* A pokud

1. $\mathcal{D} = \{C_1, \ldots, C_n\}$ je dekompozicí Q_A a
2. pro každé dvě *komponenty* $R, S \in \mathcal{D}$ takové, že $R \neq S$, platí, že pokud existují stavy $q_r \in R$ a $q_s \in S$ takové, že q_s je dosažitelný z q_r, pak neexistují žádné stavy $q'_r \in R$ a $q'_s \in S$ takové, že q'_r je dosažitelný z q'_s.

Například množina silně souvislých komponent stavového diagramu A je dekompozicí.

Nechť $\preceq \subseteq \mathcal{D} \times \mathcal{D}$ je taková relace, že platí $R \preceq S$ tehdy a pouze tehdy, pokud existují $q_r \in R$ a $q_s \in S$ takové, že q_s je dosažitelný z q_r (i triviálně). Relace $\preceq$ je tedy částečné uspořádání. Nechť $\leq$ je úplné uspořádání nad $\mathcal{D}$ takové, že je zesílením $\preceq$. Uspořádejme komponenty v $\mathcal{D}$ do posloupnosti $C_1 \leq C_2 \leq \cdots \leq C_n$ a pro každé $1 \leq i \leq n$ označme A_i konečný automat $A_i = (C_i, \delta_A|_{C_i}, \emptyset, F_A \cap C_i)$ a In_i označme množinu *vstupních portů* A_i definovaných jako $In_i = \{r \in C_i \mid \exists q \in Q_A \setminus C_i \colon q \xrightarrow{a} r \in \delta_A\}$. Předpokládejme, že všechny A_i kromě A_n jsou deterministické a úplné. Nechť δ_{trans} značí přechody mezi jednotlivými komponentami, například $\delta_{trans} = \delta_A \setminus (\bigcup_{1 \leq i \leq n} C_i \times \Sigma \times C_i)$. Nyní sestrojíme posloupnost dvojic $(B_1, InMap_1), \ldots, (B_n, InMap_n)$, kde B_i je konečný automat a $InMap_i \colon In_i \to 2^{Q_{B_i}}$ pro každé $1 \leq i \leq n$, následovně:

1. $B_n = (Q_n, \delta_n, I_n, F_n)$ je konečný automat (získaný jakýmkoliv způsobem) a $InMap_n \colon In_n \to 2^{Q_n}$ je mapování takové, že pro všechny $q \in In_n$ platí

$$\mathcal{L}_{A_n}(q) = \overline{\mathcal{L}_{B_n}(InMap_n(q))} = \bigcap_{r \in InMap_n(q)} \overline{\mathcal{L}_{B_n}(r)} \tag{6.1}$$

   a navíc $\mathcal{L}(B_n) = \overline{\mathcal{L}(A_n)}$. Např. pokud B_n je sestrojen pomocí subset konstrukce, např. $Q_n \subseteq 2^{C_n}$, pak $InMap_n(q) = \{\{q\}\}$ a $I_n = \{I_A \cap C_n\}$. Na druhou stranu, pokud B_n je

sestrojen pomocí zpětné subset konstrukce ($Q_n \subseteq 2^{C_n}$), pak $InMap_n(q) = \{R \subseteq C_n \mid q \notin R\}$ a $I_n = \{R \subseteq C_n \mid R \cap I_A = \emptyset\}$.

2. B_i, pro $1 \leq i < n$, je sestrojen z A_i a B_{i+1} jako $B_i = (Q_i, \delta_i, I_i, F_i)$, kde

   - $Q_i = (C_i \cup \{\bot\}) \times 2^{Q_{i+1}}$ (kde $\bot$ je dedikovaný *sink* stav),
   - $F_i = ((C_i \cup \{\bot\}) \setminus F_A) \times 2^{F_{i+1}}$ (např. běh v A_i nemůže přijmout a všechny běhy v B_{i+1} musí přijmout),
   - $\delta_i((q, R), a) = \{(p, S \cup T) \mid p = \delta_A(q, a), S \in \delta^{\times}_{i+1}(R, a), T \in \gamma^{\times}_{i+1}(q, a)\}$ kde
     (a) $\delta^{\times}_{i+1}$ je *multi-product transition function* z B_{i+1} definovaná tak, že $a \in \Sigma$ a $R \subseteq Q$, kde $R = \{r_1, \ldots, r_m\}$, $\delta^{\times}_{i+1}(R, a)$ obsahuje právě množiny $S = \{s_1, \ldots, s_m\}$, kde $s_j \in \delta_{i+1}(r_j, a)$ pro každé $1 \leq j \leq m$[1] (zdůrazňujeme, že pokud pro některé r_j platí $\delta_{i+1}(r_j, a) = \emptyset$, pak $\delta^{\times}_{i+1}(R, a) = \emptyset$),
     (b) podobně, $\gamma^{\times}_{i+1}$ je *intergalactic multi-product transition function* A_i a B_{i+1} takové, že pro $a \in \Sigma$, $q \in C_i$, a $\delta_{trans}(q, a) = \{q'_1, \ldots, q'_k\}$, definujeme $\gamma^{\times}_{i+1}(q, a)$ tak, aby obsahovala právě množiny $T = \{t_1, \ldots, t_k\}$, kde $t_j \in InMap_{i+1}(q'_j)$ pro každé $1 \leq j \leq k$.
   - I_i je definované jako:

$$I_i = \begin{cases} \{(q, \{s\}) \mid q \in I_A \cap C_i, s \in I_{i+1}\} & \text{pokud } I_A \cap C_i \neq \emptyset, \\ \{(\bot, \{s\}) \mid s \in I_{i+1}\} & \text{v opačném případě.} \end{cases}$$

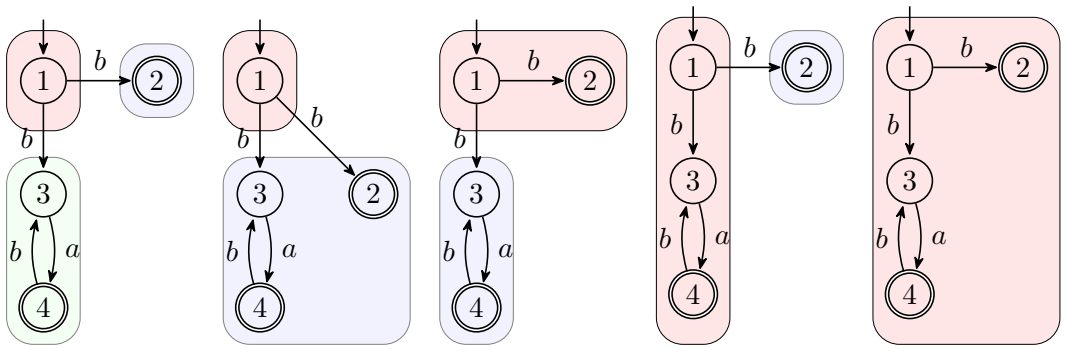
   $InMap_i$ je pak definována jako:

$$InMap_i(q) = \begin{cases} \{(\bot, \{s\}) \mid s \in InMap_{i+1}(q)\} & \text{pokud } q \in C_k \text{ for } k > i, \\ \{(q, \emptyset)\} & \text{pokud } q \in C_i. \end{cases}$$

6.2 Dekompozice

Každý konečný automat A může být převeden na orientovaný graf D, který jej reprezentuje následujícím způsobem. Pro každý stav $q \in Q_A$ vytvoříme vrchol q a pro každý přechod $q_1 \xrightarrow{a} q_2$ vytvoříme hranu (q_1, q_2). Nyní můžeme rozložit automat na jednotlivé SCC (silně

[1] $|S|$ může být menší, než m jelikož některé stavy s_j mohou být identické.



Obrázek 6.1: Všechny validní *dekompozice*

souvislé komponenty), např. pomocí Tarjanova algoritmu [15]. Komponenty, které vstupují do algoritmu, ovšem nemusí odpovídat právě SCC. Podstatné je, aby přechody mezi jednotlivými komponentami automatu byly pouze jedním směrem. Pokud existuje přechod z komponenty C_1 do C_2, pak již nesmí existovat přechod z C_2 do C_1. Toto platí pro každou dvojici komponent, které společně tvoří *dekompozici* automatu. Na Obrázku 6.1 jsou zobrazeny všechny validní dekompozice daného automatu. Příkladem chybné *dekompozice* by byla taková *dekompozice*, ve které některý ze stavů automatu chybí, např. $\{\{1\},\{3,4\}\}$ neobsahuje stav 2 a proto není validní. Další nevalidní dekompozicí by byla $\{\{1,3\},\{4\},\{2\}\}$, jelikož obsahuje přechod z komponenty $\{1,3\}$ do $\{4\}$ ($3 \xrightarrow{a} 4$), ale i přechod opačným směrem ($4 \xrightarrow{b} 3$). Nad *dekompozicí* D existuje relace $\preceq \subseteq \mathcal{D} \times \mathcal{D}$, která je částečným uspořádáním. V praxi nám udává pořadí jednotlivých komponent. V tomto pořadí jsou také postupně zpracovávány algoritmem. Obecně může nastat situace, kdy dvě komponenty nejsou porovnatelné a nelze jednoznačně určit, která bude zpracována dřív, viz Obrázek 6.1a. V tomto případě je možno si libovolně zvolit. Ideální by bylo nalézt heuristiku, která dokáže určit, které pořadí komponent vede na nejmenší výsledný automat. Nad komponentami, které jsou takto seřazeny, platí relace úplného uspořádání $\leq$, která je zesílením částečného uspořádání $\preceq$. Pro *dekompozici* na Obrázku 6.1a existují dvě možná úplná uspořádání: $\{1\} \leq \{2\} \leq \{3,4\}$ a $\{1\} \leq \{3,4\} \leq \{2\}$.

6.3 Vstupní a výstupní porty

Pro každou komponentu C_i sestrojíme automat A_i, který obsahuje všechny stavy z C_i a přechody mezi nimi. Je také potřeba si pamatovat, jaký je vztah tohoto nového automatu k ostatním A_i. K tomu slouží porty. Pro každý automat A_i v *dekompozici* existuje množina vstupních a výstupních portů. Vstupní porty jsou takové stavy, pro které platí, že do nich vede přechod z jiné komponenty. Iniciální stavy bychom mohli vnímat jako speciální případ vstupních portů, ale v algoritmu jsou zpracovávány zvlášť. Množinu vstupních portů značíme In_i, kde i je pořadí jejich komponenty. Podobně z výstupních portů vedou přechody do následujících komponent. Výstupní porty značíme Out_i, kde i je pořadí jejich komponenty. Poslední komponenta C_n v úplném uspořádání $\leq$ žádné výstupní porty nemá. Pokud uvažujeme uspořádání komponent $\{1\} \leq \{2,3,4\}$ automatu na obrázku 6.1b, pak $C_1 = \{1\}$ a $C_2 = C_n = \{2,3,4\}$. Komponenta C_1 má jeden výstupní port a žádný vstupní, $In_1 = \emptyset$ a $Out_1 = \{1\}$. Komponenta C_n má dva vstupní porty a žádné výstupní, $In_n = \{2,3\}$ a $Out_n = \emptyset$.

6.4 Komplement poslední komponenty

Z automatu A_n, který vznikl z komponenty C_n je potřeba na začátku algoritmu vytvořit jeho komplement, který je označen $B_n = (Q_n, \delta_n, I_n, F_n)$. Je to nutné provést způsobem, který zachovává informace o vstupních portech. K tomu slouží funkce $InMap_n$, která mapuje vstupní porty původního automatu A_n na porty v automatu B_n. Pro každý port $q \in In_n$ funkce vrací množinu portů z automatu B_n. Musí zároveň platit, že jazyk $\mathcal{L}_{B_n}(InMap_n(q))$ je komplementem jazyka $\mathcal{L}_{A_n}(q)$. V případě, že $InMap_n(q)$ je množina více stavů, pak musí platit, že jazyk stavu q, $\mathcal{L}(q)$ je komplementem průniku jazyků v $InMap_n(q)$. Formálně zapsáno:

$$\mathcal{L}_{A_n}(q) = \overline{\mathcal{L}_{B_n}(InMap_n(q))} = \bigcap_{r \in InMap_n(q)} \overline{\mathcal{L}_{B_n}(r)}$$

B_n a $InMap_n(q)$ můžou být získány libovolným způsobem. V zásadě existují dvě možnosti: Komplement pomocí dopředné subset konstrukce nebo zpětné subset konstrukce. Popsány jsou v Algoritmech 5 a 6.

Algorithm 5: Dopředná subset konstrukce komponenty

Input: $A_n = (Q_{A_n}, \delta_{A_n}, I_{A_n}, F_{A_n}), In_n$
Output: $B_n = (Q_{B_n}, \delta_{B_n}, I_{B_n}, F_{B_n}), InMap_n$ takové, že $L(B_n) = \overline{L(A_n)}$ a $\forall i \in In_n, L(i) = \overline{L(InMap_n(i))}$

```
1  worklist.push(I_{A_n}) ;
2  done := {I_{A_n}};
3  I_{B_n} := I_{A_n};
4  foreach q ∈ In_n do
5      worklist.push({q}) ;
6      InMap_n(q) := {q};
7      done := done ∪{{q}};
8  end
9  while not worklist.empty() do
10     Q := worklist.pop() ;
11     if Q ∩ F_{A_n} = ∅ then
12         F_{B_n} := F_{B_n} ∪ {Q} ;
13     end
14     foreach a ∈ Σ do
15         Q_new := ∅ ;
16         foreach q ∈ Q do
17             Q_new := Q_new ∪ {δ_{A_n}(q, a)};
18         end
19         δ_{B_n}(Q, a) := Q_new ;
20         if Q_new ∉ done then
21             done := done ∪{Q_new};
22             worklist.push(Q_new;
23         end
24     end
25 end
26 return B_n, InMap_n
```

Od klasické subset konstrukce se Algoritmus 5 liší tím, že je potřeba uvažovat nejen iniciální stavy, ale také i vstupní porty. Na řádcích 1 až 3 jsou vloženy do zásobníku *worklist* stavy, které reprezentují porty ve vytvářeném automatu B_n. Do těchto portů se přechází z

ostatních komponent v rámci automatu. Informace je následně uložena do $InMap_n$, která se využívá v hlavní části algoritmu pro konkatenaci pomocí SCC.

Algorithm 6: Zpětná subset konstrukce komponenty

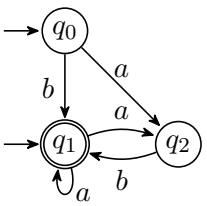
Input: $A_n = (Q_{A_n}, \delta_{A_n}, I_{A_n}, F_{A_n}), In_n$
Output: $B_n = (Q_{B_n}, \delta_{B_n}, I_{B_n}, F_{B_n}), InMap_n$ takové, že $L(B_n) = \overline{L(A_n)}$ a $\forall i \in In_n, L(i) = \overline{L(InMap_n(i))}$

```
1  oldInputPorts := I_{A_n};
2  foreach q ∈ In_n do
3      I_{A_n}.insert(q);
4  end
5  B_n = reverseSubsetComplement(A_n);
6  foreach state in B_n do
7      if state ∩ I_{A_n} = ∅ then
8          I_{B_n}.insert(state);
9      end
10     foreach portState in In_n do
11         if portState ∉ state then
12             InMap_n(portState).insert(state);
13         end
14     end
15 end
16 return B_n, InMap_n
```

Algoritmus 6 popisuje vytváření poslední komponenty pomocí algoritmu pro zpětnou subset konstrukci. Nejprve jsou k iniciálním stavům komponenty přidány vstupní porty. Pak je využito algoritmu pro komplement pomocí zpětné subset konstrukce. Následně jsou iniciální stavy nově vzniklého automatu rozlišeny na stavy, na skutečné iniciální stavy a stavy, které reprezentují vstupní porty. Informace o vstupních portech je uložena do $InMap_n$.

6.5 Iterativní část algoritmu

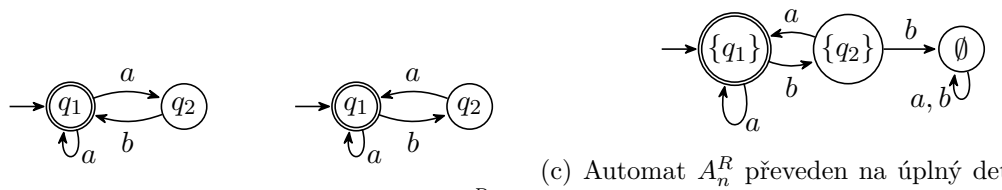
Algoritmus iterativně vytváří komplement automatu vždy pro dvě poslední komponenty v úplném uspořádání $\leq$ v *dekompozici* automatu. V principu funguje velmi podobně jako algoritmus pro komplementaci konkatenace dvou automatů. Hlavní rozdíl je tom, že mezi dvěma komponentami může vést více přechodů do různých stavů. Komponenty také mohou obsahovat iniciální stavy.



Obrázek 6.2: Automat ke komplementaci

Na Obrázku 6.2 se nachází automat ke komplementaci. Obsahuje dvě silně souvislé komponenty $\{q_0\}$ a $\{q_1, q_2\}$.

Z poslední komponenty $\{q_1, q_2\}$ je vytvořen automat A_n (Obrázek 6.3a), který obsahuje tyto dva stavy a přechody mezi nimi. Na tento automat je aplikována operace reverzace a je vytvořen automat A_n^R (6.3b). Tento automat je převeden na deterministický úplný automat (Obrázek 6.3c). Je přidán tzv. *sink* stav, který reprezentuje prázdnou množinu stavů. Do tohoto stavu vedou přechody ze všech stavů, které nemají přechod přes některý symbol. V tomto případě je přidán přechod $\{q_2\} \xrightarrow{b} \emptyset$.



(a) Automat A_n (b) Automat A_n^R (c) Automat A_n^R převeden na úplný deterministický automat

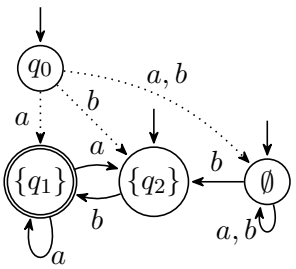
Obrázek 6.3: Komplementace poslední komponenty zpětnou subset konstrukcí (1. část)

Deterministický automat A_n^R je možné komplementovat výměnou koncových a nekoncových stavů. Tento nový automat je označen $\overline{A_n^R}$ na Obrázku 6.4a. Poslední úpravou je druhá reverzace a vzniká automat B_n na Obrázku 6.4b.

(a) Automat $\overline{A_n^R}$ (b) Automat $B_n = \overline{A_n^R}^R$

Obrázek 6.4: Komplementace poslední komponenty zpětnou subset konstrukcí (2. část)

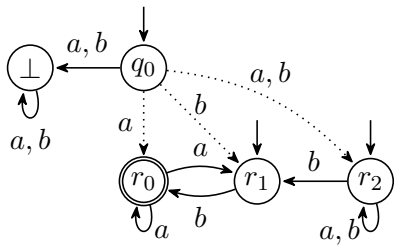
Když je nyní automat B_n hotov, je potřeba určit jakým způsobem se změnily porty této komponenty. Když je použita zpětná komplementace, tak jsou porty namapovány právě na ty stavy, které původní port neobsahují. Původní port q_1 je namapován na stavy $\{q_2\}$ a $\emptyset$ a původní port q_2 je namapován na stavy $\{q_1\}$ a $\emptyset$. Tato informace je uložena do $InMap_n$.



Obrázek 6.5: Dvě sousedící komponenty připravené k propojení a komplementaci

Na Obrázku 6.5 je zobrazená první komponenta A_1 s jediným stavem q_0. Druhá komponenta má tři stavy $\{q_1\}$, $\{q_2\}$ a $\emptyset$. Původní porty z druhé komponenty jsou nahrazeny novými porty. Tyto nové přechody jsou na Obrázku 6.5 zobrazeny pomocí tečkovaných přechodů. Pro lepší čitelnost v další části jsou stavy $\{q_1\}$, $\{q_1\}$ a $\emptyset$ po řadě přejmenovány na

stavy r_0, r_1 a r_2 (Obrázek 6.6). Do první komponenty byl také přidán stav $\perp$ který plní funkci *sink* stavu a díky tomu je první komponenta úplný deterministický automat.



Obrázek 6.6: Dvě sousedící komponenty připravené k propojení a komplementaci s přejmenovanými stavy

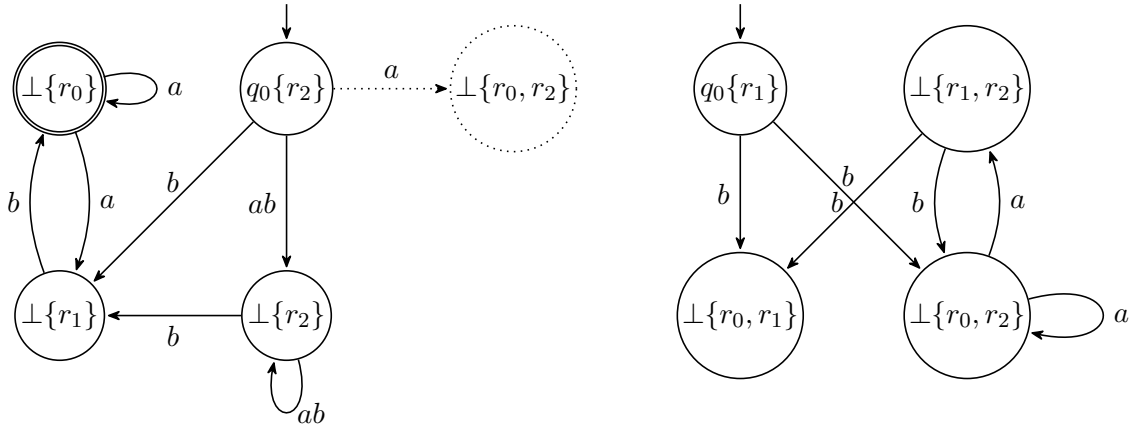
Dalším krokem bude tyto dvě komponenty propojit takovým způsobem, aby byl vytvořen komplement celého původního automatu. Začíná se od iniciálních stavů. Iniciální stavy jsou definovány jako:

$$I_i = \begin{cases} \{(q, \{s\}) \mid q \in I_A \cap C_i, s \in I_{i+1}\} & \text{pokud } I_A \cap C_i \neq \emptyset, \\ \{(\perp, \{s\}) | s \in I_{i+1}\} & \text{v opačném případě.} \end{cases}$$

V tomto případě je $C_i = \{q_0\}$ a $I_A = \{q_0, q_1\}$, tedy platí $I_A \cap C_i \neq \emptyset$. Vstupními stavy celkového automatu budou $(q_0, \{r_1\})$ a $(q_0, \{r_2\})$ viz Obrázek 6.7. Přechodová relace automatu je definována takto:

$$\delta_i((q, R), a) = \{(p, S \cup T) \mid p = \delta_A(q, a), S \in \delta^{\times}_{i+1}(R, a), T \in \gamma^{\times}_{i+1}(q, a)\}$$

Přechody ze stavu $q_0, \{r_1\}$ jsou vytvořeny takto: q je v tomto případě stav q_0, R je množina $\{r_1\}$ a a je symbol b. Ze stavu $(q_0, \{r_1\})$ se přes symbol b přechází do množiny stavů, které mají tvar $(p, \{S \cup T\})$. Stav p je takový stav, do kterého se přejde ze stavu q_0 v první komponentě. Pro symbol b se přechází do stavu $\perp$. Bude se tedy přecházet do množiny stavů tvaru $(\perp, \{S \cup T\})$. Funkce $\delta^{\times}_{i+1}(R, a)$ má následující chování: pro každý stav z $R = \{r_1, r_2, \ldots, r_n\}$ zjistí, do jaké množiny stavů stav r_i přechází přes symbol a. Z každé z těchto množin vybere jeden prvek a z nich vytvoří novou množinu. A nakonec vrací všechny možné množiny, které mohou vzniknout tímto způsobem. Pokud pro některé ze stavů v R neexistuje přechod, pak funkce $\delta^{\times}_{i+1}(R, a)$ vrací prázdnou množinu. V tomto případě $\delta^{\times}_{i+1}(\{r_1\}, b)$ vrací množinu $\{\{r_0\}\}$, protože stav r_1 má přes symbol b přechod do stavu r_0. Platí $S \in \{\{r_0\}\}$. Na prakticky stejném principu pracuje funkce $\gamma^{\times}_{i+1}(q, a)$, kde q je stav z první komponenty. Pro $\gamma^{\times}_{i+1}(q_0, a)$ funkce vrací $\{\{r_1\}, \{r_2\}\}$, protože ze stavu q_0 existují přechody do stavů r_1, r_2 v druhé komponentě. Platí $T \in \{\{r_1\}, \{r_2\}\}$ a $S \in \{\{r_0\}\}$. Stavy tvaru $(\perp, \{S \cup T\})$ pro tyto S a R jsou dva: $(\perp, \{\{r_0\} \cup \{r_1\}\})$ a $(\perp, \{\{r_0\} \cup \{r_2\}\})$.



Obrázek 6.7: Výsledný komplement celého automatu

Pokud stejný proces aplikuje i pro stejný stav $(q_0, \{r_1\})$ symbol a zjistíme, že stav r_1 nemá přes symbol a žádný přechod. Platí tedy $S \in \emptyset$ a žádný přechod do stavu tvaru $(p, \{S \cup T\})$ neexistuje.

Opět je v tomto algoritmu možné využít optimalizace *little brothers*, která byla detailněji popsána v sekci 5.4. Například ze stavu $(q_0\{r_2\})$ vedou přes symbol a přechody do stavů $(\bot\{r_2\})$ a $(\bot\{r_0, r_2\})$. Díky této optimalizaci je možno přechod do stavu $(\bot\{r_0, r_2\})$ zahodit beze změny přijímaného jazyka. Díky tomu tento stav ve výsledném automatu nikdy nevznikne. Na Obrázku 6.7 je proto zobrazen tečkovaně.

Jediný stav, který je koncový, je $(\bot, \{r_0\})$, protože všechny stavy z druhé komponenty, které obsahuje, jsou koncové (stav r_0). Například stav $(\bot, \{r_0, r_1\})$ koncový není, protože obsahuje i stav r_1, který koncový není. Jak je z Obrázku 6.7 patrné, ze stavu $(\bot, \{r_1\})$ nevede žádná cesta do koncového stavu a je proto možné jej zahodit. Stejně tak pro stavy $(\bot, \{r_0, r_1\})$, $(\bot, \{r_0, r_1\})$ a $(\bot, \{r_0, r_1\})$. Výsledný automat má pouze čtyři stavy: $(q_0, \{r_2\})$, $(\bot, \{r_0\})$, $(\bot, \{r_1\})$ a $(\bot, \{r_2\})$.

6.6 Metoda dvou komponent

V předchozí sekci byl popsán iterativní postup vytváření komplementu nedeterministického konečného automatu. Alternativní možností je rozložit automat pouze na dvě komponenty a to takovým způsobem, že druhá z komponent bude zpětně deterministická. Zpětně deterministická komponenta má tu vlastnost, že je po reverzaci deterministická.

Nalezení maximální zpětně deterministické komponenty popisuje Algoritmus 7. Automat $M = (Q_M, \delta_M, I_M, F_M)$ je automat, který je potřeba komplementovat. Dekompozice $\mathcal{D}_i$ obsahuje silně souvislé komponenty automatu M. Pro *dependency* platí:

$$dependency(d) = \{d_i \in \mathcal{D}_i | \exists (q, a, q') \in \delta_M . q \in d \wedge q' \in d_i\} \tag{6.2}$$

Jinými slovy *dependency*(d) vrací všechny komponenty z dekompozice $\mathcal{D}_i$, do kterých existuje přechod z komponenty d.

Algorithm 7: Nalezení maximální zpětně deterministické komponenty

Input: automat M,dekompozice $\mathcal{D}_i$
Output: dekompozice $\mathcal{D}_o$

```
1  done := ∅
2  lastComponent := ∅
3  change := true
4  todo := D_i
5  while change do
6      change := false
7      foreach d ∈ todo do
8          dependencyTodo := false
9          foreach d_d ∈ dependency(d) do
10             if d_d ∉ done then
11                 dependencyTodo := true
12                 break
13             end
14         end
15         if dependencyTodo then
16             continue
17         end
18         todo := todo \ {d}
19         newComponent := lastComponent ∪ d
20         M' := makeAutomata(newComponent, M)
21         M_R := reverse(M')
22         if isDeterministic(M_R) then
23             lastComponent := newComponent
24             change := true
25         end
26     end
27     firstComponent := ∅ foreach d ∈ D_i do
28         if d ∩ lastComponent = ∅ then
29             firstComponent :=firstComponent ∪ d
30         end
31     end
32     D_o := {firstComponent, lastComponent}
33 end
```

Algoritmus 7 se pokouší iterativně zvětšovat novou poslední komponentu a v každém kroku ověří, zda je zpětně deterministická. Pokud zpětně deterministická není, pokusí se připojit jinou komponentu. Algoritmus přidává jenom ty komponenty, které nemají žádné následující komponenty, anebo všechny následující komponenty se jíž nachází v poslední komponentě. Ve výsledku vzniká nová dekompozice $\mathcal{D}_i$, která obsahuje dvě komponenty: maximální zpětně deterministickou komponentu a komponentu, která obsahuje všechny ostatní stavy automatu M. S touto dekompozicí se provede jeden krok iterativního vytváření

komplementu s tím, že se na poslední komponentu využije zpětná subset konstrukce z Algoritmu 6.

Výhodou tohoto algoritmu je fakt, že se použije zpětná subset konstrukce na maximální zpětně deterministickou část automatu a v prvním kroku tak nedojde ke zvýšení počtu stavů. Zvýšení počtu stavů může nastat až ve chvíli determinizace první komponenty nebo vytváření celkového automatu $\overline{M}$ z těchto dvou komponent. Podobným způsobem by šlo algoritmus upravit tak, aby vytvořil maximální dopředně deterministickou první komponentu a do druhé komponenty umístil všechny zbylé stavy. Ideální algoritmus by byl schopný tyto dva přístupy kombinovat a dokázal by odhadnout nárůst počtu stavů.

6.7 Redukce komponent

Je velmi výhodné komponenty během algoritmu průběžně minimalizovat. V současnosti existuje mnoho algoritmů pro minimalizace automatů a to jak minimalizace pro deterministické i nedeterministické automaty. Minimalizace komponent vyžaduje, aby byly zachovány vstupní porty a výstupní porty. Toho jde dosáhnout následovně: vytvoříme nový unikátní iniciální stav a pro každý původní iniciální stav a vstupní port vytvoříme nový přechod z nového iniciální stavu přes unikátní symbol, který se nenachází v abecedě. Obdobně lze řešit koncové stavy a výstupní stavy vytvořením nového koncového stavu a přechody do něj. Tento postup je formálně popsán pomocí Algoritmu 8.

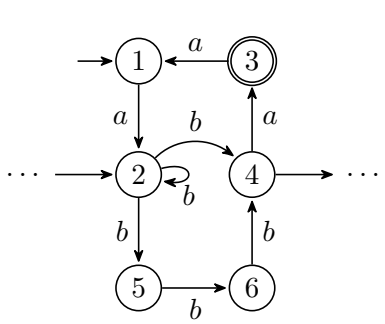
Algorithm 8: Příprava na minimalizace komponenty

Input: $A_n = (Q_{A_n}, \Sigma_{A_n}, \delta_{A_n}, I_{A_n}, F_{A_n})$, input ports In_n, output ports Out_n
Output: $B_n = (Q_{B_n}, \Sigma_{B_n}, \delta_{B_n}, I_{B_n}, F_{B_n})$, map_n takové, že
$\forall q_i \in In_n \cup I_{A_n}, (i_{B_n}, map_n(q_i), q') \in \delta_{B_n} \mid L(q') = L(q_i).i_{B_n} \in I_{B_n}$
$\forall q_o \in Out_n \cup F_{A_n}, (q', map_n(q_o), f_{B_n}) \in \delta_{B_n} \mid L(q_o) = L(q').f_{B_n} \in F_{B_n}$

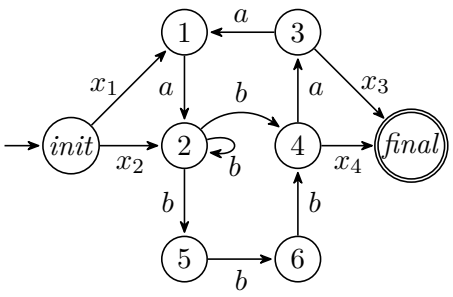
1 $I_{B_n} := \{i_{B_n} \mid i_{B_n} \notin Q_{A_n}\}$
2 $F_{B_n} := \{f_{B_n} \mid f_{B_n} \notin Q_{A_n}\}$
3 $\delta_{B_n} := \delta_{A_n}$; $\Sigma_{B_n} := \Sigma_{A_n}$
4 $Q_{B_n} := Q_{A_n} \cup \{i_{B_n}, f_{B_n}\}$
5 **foreach** $q_i \in (In_n \cup I_{A_n})$ **do**
6 $\quad \Sigma := \Sigma \cup \{a_{q_i} | a_{q_i} \notin \Sigma\}$
7 $\quad \delta_{B_n} := \delta_{B_n} \cup \{(i_{B_n}, a_{q_i}, q_i)\}$
8 $\quad map_n(a_{q_i}) := q_i$
9 **end**
10 **foreach** $q_o \in (Out_n \cup F_{A_n})$ **do**
11 $\quad \Sigma := \Sigma \cup \{a_{q_o} | a_{q_o} \notin \Sigma\}$
12 $\quad \delta_{B_n} := \delta_{B_n} \cup \{(q_o, a_{q_o}, f_{B_n})\}$
13 $\quad map_n(a_{q_o}) := q_o$
14 **end**
15 $B_n := (Q_{B_n}, \Sigma_{B_n}, \delta_{B_n}, I_{B_n}, F_{B_n})$
16 **return** B_n, map_n

Na obrázku 6.8a se nachází komponenta, kterou se pokoušíme minimalizovat. Stav 1 je vstupní stav. Stav 2 je vstupní port a stav 4 je výstupní port. Vytvoříme nový iniciální stav *init* a z něj vedeme přechod přes unikátní symbol x_1 do původního iniciálního stavu 1. Podobně vytvoříme přechod přes symbol x_2 do vstupního portu 2. Pro výstupní port 4 existuje přechod do nového koncového stavu *final* přes symbol x_4. A konečně poslední

přidaný přechod z původního koncového stavu 3 do *final* přes symbol x_3. Takto vzniklý automat (na obrázku 6.8b) je nyní možno minimalizovat za pomoci libovolné existující metody. Po aplikování minimalizace můžeme odstranit všechny přechody x_i. Díky jejich unikátnosti dokážeme určit, které stavy jsou novými porty komponenty. Je potřeba vzít na vědomí, že tato operace může odstranit všechny původní iniciální stavy. Pokud tato situace nastane, je potřeba přidat jeden iniciální stav, který je bude reprezentovat. Takový stav není koncový a nemá žádné přechody.



(a) Původní komponenta



(b) Komponenta připravena k minimalizaci

Obrázek 6.8: Minimalizace komponenty

Takto upravený automat je nyní možné minimalizovat pomocí existujících metod. Vstupní port q_i má jazyk $L(q_i)$, který je potřeba zachovat během minimalizace. Tím, že v automatu B_n vytvoříme přechod $i_{B_n} \xrightarrow{a_{q_i}} q_i$ bude platit $a_{q_i}.L(q_i).a_{q_o} \subseteq L(B_n)$. Díky tomu, že symbol a_{a_i} je unikátní v rámci automatu, jsme schopni po minimalizaci určit stav, který má stejný jazyk jako původní port q_i. Stejná myšlenka platí i pro výstupní porty komponenty.

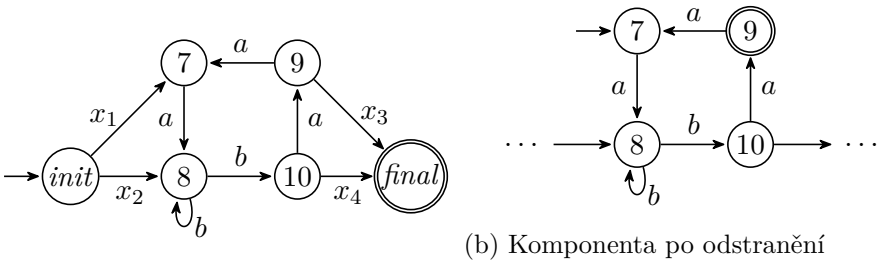
Po minimalizaci stačí odstranit přidané přechody a správně určit porty, iniciální stavy a koncové stavy. Tento postup je zapsán v Algoritmu 9.

Algorithm 9: Úprava po minimalizaci komponenty

Input: původní automat $A_n = (Q_{A_n}, \Sigma_{A_n}, \delta_{A_n}, I_{A_n}, F_{A_n})$,
minimalizovaný automat $C_n = (Q_{C_n}, \Sigma_{C_n}, \delta_{C_n}, I_{C_n}, F_{C_n})$, map_n
Output: $D_n = (Q_{D_n}, \Sigma_{D_n}, \delta_{D_n}, I_{D_n}, F_{D_n}), mapPorts$ takové, že $L(D_n) = L(A_n)$,
$mapPorts \subseteq In_n \times Q_{D_n}$ a $\forall i \in In_n, L(i) = L(mapPorts(i))$

$I_{D_n} := \emptyset$; $F_{D_n} := \emptyset$; $\Sigma_{D_n} := \Sigma_{A_n}$
$Q_{D_n} := Q_{C_n} \setminus (I_{C_n} \cup F_{C_n})$
$\delta_{D_n} := \{(q, a, q') \in \delta_{C_n} \mid q \notin I_{C_n} \wedge q' \notin F_{C_n}\}$
foreach $(q, a, q') \in \delta_{C_n}$ *such that* $q \in I_{C_n}$ **do**
    $p := map_n(a)$
    $mapPorts(p) := mapPorts(p) \cup \{q'\}$
    if $p \in I_{A_n}$ **then**
        $I_{D_n} := I_{D_n} \cup \{q'\}$
    end
end
foreach $(q, a, q') \in \delta_{C_n}$ *such that* $q' \in F_{C_n}$ **do**
    $p := map_n(a)$
    $mapPorts(p) := mapPorts(p) \cup \{q\}$
    if $p \in F_{A_n}$ **then**
        $F_{D_n} := F_{D_n} \cup \{q'\}$
    end
end

Komponenta připravená k minimalizaci na obrázku 6.8b má po minimalizaci o dva stavy méně (Obrázek 6.9a). Původní iniciální stav 1 se nyní namapuje na stav 7. Vstupní port 2 je namapován na stav 8. Koncový stav 3 je namapován na stav 9 a výstupní port 4 se namapuje na stav 10. Obecně může nastat situace, že se některý z portů namapuje na více než jeden stav a na jeden stav může být namapováno více portů.



(a) Komponenta po minimalizaci

(b) Komponenta po odstranění pomocných stavů a přechodů

Obrázek 6.9: Komponenta po minimalizaci

Mapování portů je nyní známé a poslední krokem je odstranění pomocných stavů *init* a *final* a všech přechodů, které je obsahují: $init \xrightarrow{x_1}, 7$ $init \xrightarrow{x_2} 8$, $9 \xrightarrow{x_3} final$ a $10 \xrightarrow{x_4} final$. Finální verze komponenty je zobrazena na Obrázku 6.9b.

Pokud je minimalizována poslední komponenta v řadě, nebude obsahovat žádné výstupní porty. V tomto případě není potřeba provést nahrazování výstupních portů a koncových stavů přechody přes unikátní symbol.

Kapitola 7

Implementace a testování

V této kapitole je popsána použitá knihovna pro konečné automaty, parametry pro zvolení metody pro komplementaci automatu a použití nástroje pro redukci komponent. V poslední sekci je popis generování automatů pro testování a způsob, jakým byla testována korektnost algoritmu.

7.1 Knihovna libvata2

Všechny verze navrženého algoritmu byly implementovány jako rozšíření knihovny *libvata2* [10] v jazyce C++. Tato knihovna poskytuje funkce pro základní operace nad automaty jako např. sjednocení, průnik, klasický komplement, determinizaci, otestování prázdnosti přijímaného jazyka, převod na úplný automat, reverzaci automatu, jazykovou inkluzi a další. Knihovna je schopná načíst soubor obsahující konečný automat ve formátu *vtf* [9].

7.2 Funkce pro komplementaci

Komplementace konečného automatu je v knihovně *libvata* implementována jako funkce `complement( Nfa* result, const Nfa& aut, const Alphabet& alphabet, const StringDict& params, SubsetMap* subset_map)`, která tvoří hlavní vstupní bod pro komplementaci automatů.

Pomocí argumentu `params` je možné zvolit metodu pro vytváření komplementu a případně její varianty. Možnosti hodnot pro klíč `"algo"` a odpovídající metody jsou zobrazeny v Tabulce 7.1.

Hodnota `"algo"`	Metoda
`"classical"`	subset konstrukce
`"classical_min"`	subset konstrukce s minimalizací
`"reverse"`	zpětná subset konstrukce
`"reverse_min"`	zpětná subset konstrukce s minimalizací
`"rabit"`	subset konstrukce a redukce pomocí nástroje RABIT
`"sccs"`	*Komplement pomocí portů*

Tabulka 7.1: Výběr metody komplementace přes klíč `"algo"` parametru `params`

Pokud je klíči `"algo"` přiřazena hodnota `"sccs"` je možno využít dalších parametrů, které umožňují výběr z několika variant této metody. Tabulka 7.2 zobrazuje čtyři nepovinné

parametry v `params`, které upravují chování metody. Pokud je použita metoda maximální poslední komponenty, předpokládá se i parametr `"reverse"`.

Klíč	Hodnota	Varianta metody
`"reverse"`	`"true"`	zpětná subset konstrukce poslední komponenty
`"min_first_part"`	`"true"`	minimalizace první ze dvou propojovaných komponent
`"min_second_part"`	`"true"`	minimalizace druhé ze dvou propojovaných komponent
`"join_sccs"`	`"true"`	*Metoda dvou komponent*

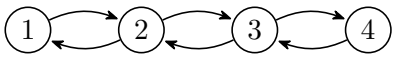
Tabulka 7.2: Výběr metody komplementace přes klíč `"algo"` parametru `params`

7.3 Redukce komponent

Pro minimalizaci komponent byl využit nástroj RABIT [1], který je určen k redukci Büchiho automatů i konečných automatů. Tento nástroj, implementovaný v jazyce *java*, načte konečný automat ze souboru ve formátu *ba* a v tomto formátu je i výstupní soubor. Při každém použití jsou vytvořeny dva dočasné soubory v adresáři /tmp. Jeden pro vstup a druhý pro výstup nástroje RABIT. Pro použití je potřeba nastavit proměnnou prostředí $RABITEXE, která obsahuje cestu k spustitelnému souboru Reduce.jar. k

7.4 Testování

Testování implementovaného algoritmu probíhalo na náhodných automatech. Postupně byl zvyšován počet silně souvislých komponent, ze kterých se automaty skládaly. Aby bylo možné zaručit, že automat obsahuje požadovaný počet silně souvislých komponent, byly stavy generovány způsobem zobrazeným na Obrázku 7.1. Tyto čtyři stavy jistě tvoří silně souvislou komponentu. Následně byly náhodně zvoleny iniciální a koncové stavy. Jako poslední krok byly přidány přechody mezi jednotlivými komponentami a to takovým způsobem, že mezi dvěma komponentami vedou přechody pouze jedním směrem. Knihovna *libvata2* obsahuje funkce pro test jazykové inkluze. Platí následující: $L(A_1) \sqsubseteq L(A_2) \wedge L(A_2) \sqsubseteq L(A_1) \implies L(A_1) = L(A_2)$, tedy pokud pokud jazyk automat A_2 obsahuje všechny řetězce jazyka automatu A_1 a opačně, pak oba automaty přijímají stejný jazyk. Díky tomu bylo možné testovat, zda automaty generované novým algoritmu jsou korektní. Pokud jazyková inkluze neplatí, funkce `is_incl` zjistí, pro který řetězec se přijímané jazyky liší. Díky tomu bylo opravování chyb usnadněné.



Obrázek 7.1: Příklad generované silně souvislé komponenty konečného automatu pro účely testování

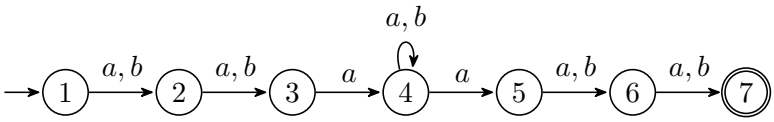
Kapitola 8

Experimenty

V této kapitole jsou popsány provedené experimenty pro implementovaný algoritmus a jeho varianty. Testování probíhalo na stroji s procesorem *Intel Xeon X5650 @ 2.67GHz* s operační pamětí *32GB*. Testy byly provedeny za pomoci nástroje *pycobench* [11], který umožňuje provádět jednotlivé části testů paralelně a následně vytvoří soubor ve formátu *csv* [7] obsahující souhrnné informace. Formát *csv* je v tomto případě velmi vhodný, jelikož umožňuje jednoduchou úpravu dat a následné statistické zpracování.

8.1 Motivační případ

Existují automaty, pro které oba současné algoritmy produkují exponenciálně větší konečný automat, který přijímá komplement původního jazyka. Příkladem je nedeterministický konečný automat pro jazyk definovaný regulárním výrazem $(a+b)^n a(a+b)^* a(a+b)^n$. Tento jazyk obsahuje všechny řetězce nad abecedou $\{a, b\}$, které mají symbol a na pozici $n+1$ od začátku řetězce a také symbol a na pozici $n+1$ od konce s minimální délkou $2n+2$. Automat pro $n = 2$ je zobrazen na Obrázku 8.1.



Obrázek 8.1: Konečný automat pro regulární výraz $(a+b)^n a(a+b)^* a(a+b)^n$, kde $n = 2$

Část automatu obsahující stavy $4, 5, 6, 7$ je nedeterministická. Pokud je automat převeden na jeho komplement, během převodu na deterministický automat dojde k explozi stavů. Pokud se zvolí metoda komplementace pomocí reverzace, dojde k nárůstu počtu stavů v části automatu se stavy $1, 2, 3, 4$. Velmi vhodné je v tomto případě využít *Metodu dvou komponent*, která umožňuje tento automat rozdělit na dvě komponenty $1, 2, 3$ a $4, 5, 6, 7$. První komponenta $1, 2, 3$ bude zpracována dopřednou subset konstrukcí a $4, 5, 6, 7$ bude zpracována zpětnou subset konstrukcí. Díky tomu budou obě části zpracovány pro ně vhodnou metodou a následně spojeny algoritmem komplementu *pomoci portů*.

Vstup		Classical		Reverse		Two components	
n	stavy	čas[s]	stavy	čas[s]	stavy	čas[s]	stavy
1	5	0.0	7	0.0	7	0.0	12
2	7	0.0	12	0.0	12	0.0	20
3	9	0.0	21	0.0	21	0.0	30
4	11	0.0	38	0.0	38	0.0	42
5	13	0.0	71	0.0	71	0.0	56
6	15	0.0	136	0.0	136	0.0	72
7	17	0.0	265	0.0	265	0.0	90
8	19	0.0	522	0.0	522	0.0	110
9	21	0.01	1035	0.01	1035	0.0	132
10	23	0.02	2060	0.03	2060	0.0	156
11	25	0.05	4109	0.05	4109	0.0	182
12	27	0.11	8206	0.11	8206	0.01	210
13	29	0.28	16399	0.28	16399	0.01	240
14	31	0.54	32784	0.58	32784	0.01	272
15	33	1.17	65553	1.2	65553	0.0	306
16	35	2.39	131090	2.5	131090	0.01	342
17	37	4.81	262163	5.04	262163	0.01	380
18	39	9.93	524308	10.3	524308	0.01	420
19	41	19.96	1048597	20.53	1048597	0.01	462
20	43	40.61	2097174	41.96	2097174	0.01	506

Tabulka 8.1: Tabulka zobrazuje srovnání klasické komplementace, komplementace za pomoci reverzace a komplementace za použití *Metody dvou komponent* na rodině automatů pro regulární výrazu tvaru $(a+b)^n a(a+b)^* a(a+b)^n$

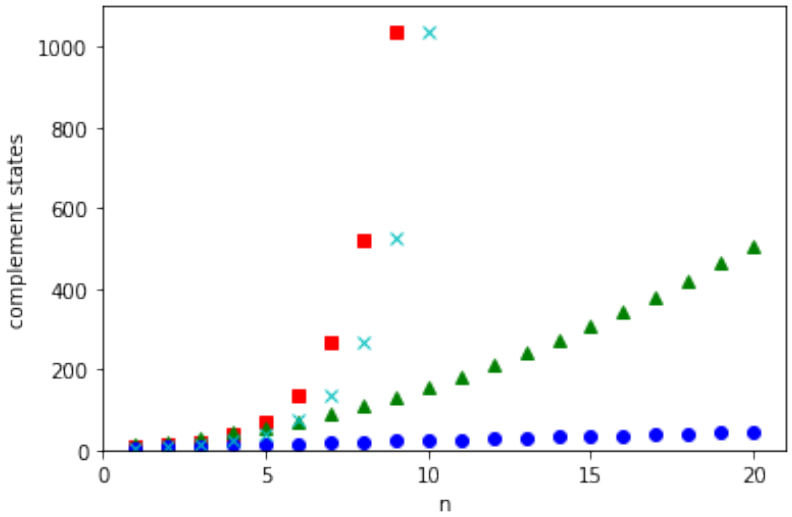
Automaty pro regulární výraz $(a+b)^n a(a+b)^* a(a+b)^n$ byly vygenerovány pro $n = 1$ až $n = 20$. Jelikož jsou automaty symetrické, metoda pomocí subset konstrukce i zpětné subset konstrukce generuje stejně velké automaty. Jak je z Tabulky 8.1 zřejmé, pro $n \geq 5$ je *Metoda dvou komponent* lepší. Klasický komplement pomoci subset konstrukce i komplement pomoci zpětné subset konstrukce mají exponenciální nárůst počtu stavů vzhledem k n a tedy i k počtu stavů na vstupu. Pro každé zvýšení n o 1 je výsledný počet stavů zhruba dvakrát větší. Tomu odpovídá i čas. Naproti tomu *Metoda dvou komponent* má kvadratický nárůst počtu stavů i času. Hlavním důvodem je skutečnost, že při převodu první komponenty na deterministickou se počet stavů nezmění, protože již deterministická je. Při převodu druhé komponenty na její komplement se také nezmění počet stavů, protože se použije zpětná subset konstrukce a tato komponenta je zpětně deterministická. Jediná část algoritmu, kdy přibudou stavy, je propojení těchto dvou komponent ve výsledný automat. Každý stav v novém automatu představuje jeden stav z první komponenty a množinu stavů z druhé komponenty. Potřebný čas k provedení tohoto algoritmu je přímo úměrný počtu stavů, které vzniknou.

Další experimenty byly provedeny na stejných automatech jako v předchozí tabulce. V tomto případě bylo využito pro obě existující metody i Brzozowského minimalizace [5]. Z výsledků vyplývá, že výsledné automaty v předchozí tabulce již byly minimální deterministické a snaha je minimalizovat měla za následek pouze nárůst spotřebovaného času. Časový limit byl nastaven na 60 s a pro $n = 18, 19, 20$ byl tento limit překročen u obou současných metod.

Tabulka 8.2 obsahuje stejné automaty, jako obě předchozí tabulky. V těchto experimentech bylo využito redukce automatů pomocí nástroje RABIT [1] s parametrem *lookahead* = 6. Automaty, které vznikly pomocí subset konstrukce a zpětné subset konstrukce byly redukovány jako poslední krok. V metodě *dvou komponent* byla redukována poslední komponenta před propojením obou komponent a následně i celkový automat v posledním kroku. I v tomto případě dosahuje nová metoda výrazně lepších výsledků než obě předchozí. Počet stavů a čas roste pouze lineárně.

Vstup		Classical		Reverse		Two components	
n	stavy	čas[s]	stavy	čas[s]	stavy	čas[s]	stavy
1	5	0.75	6	0.71	6	1.12	6
2	7	0.9	9	0.85	9	1.26	8
3	9	1.11	14	1.05	14	1.46	10
4	11	1.53	23	1.63	23	1.64	12
5	13	2.94	40	2.94	40	1.69	14
6	15	5.39	73	5.22	73	2.02	16
7	17	9.88	138	11.26	138	2.38	18
8	19	18.66	267	18.64	267	2.49	20
9	21	30.42	524	30.33	524	2.46	22
10	23	67.63	1037	66.69	1037	2.72	24
11	25	TO	TO	TO	TO	3.18	26
12	27	TO	TO	TO	TO	3.06	28
13	29	TO	TO	TO	TO	3.14	30
14	31	TO	TO	TO	TO	3.3	32
15	33	TO	TO	TO	TO	4.19	34
16	35	TO	TO	TO	TO	5.14	36
17	37	TO	TO	TO	TO	4.42	38
18	39	TO	TO	TO	TO	5.72	40
19	41	TO	TO	TO	TO	7.38	42
20	43	TO	TO	TO	TO	7.52	44

Tabulka 8.2: Tabulka zobrazuje srovnání klasické komplementace, komplementace pomoci reverzace a komplementace za použití *Metody dvou komponent* na rodině automatů pro regulární výrazu tvaru $(a + b)^n a(a + b)^* a(a + b)^n$. Všechny metody byly optimalizovány pomocí nástroje RABIT.



Obrázek 8.2: Srovnání metod na automatech pro regulární výraz $(a+b)^n a(a+b)^* a(a+b)^n$ v závislosti na n. Čtverec představuje subset konstrukci, křížek představuje subset konstrukcí s redukcí, trojúhelník představuje *Metodu dvou komponent* a kolečko představuje *Metodu dvou komponent* s redukcí. Redukce byly provedeny pomocí nástroje RABIT.

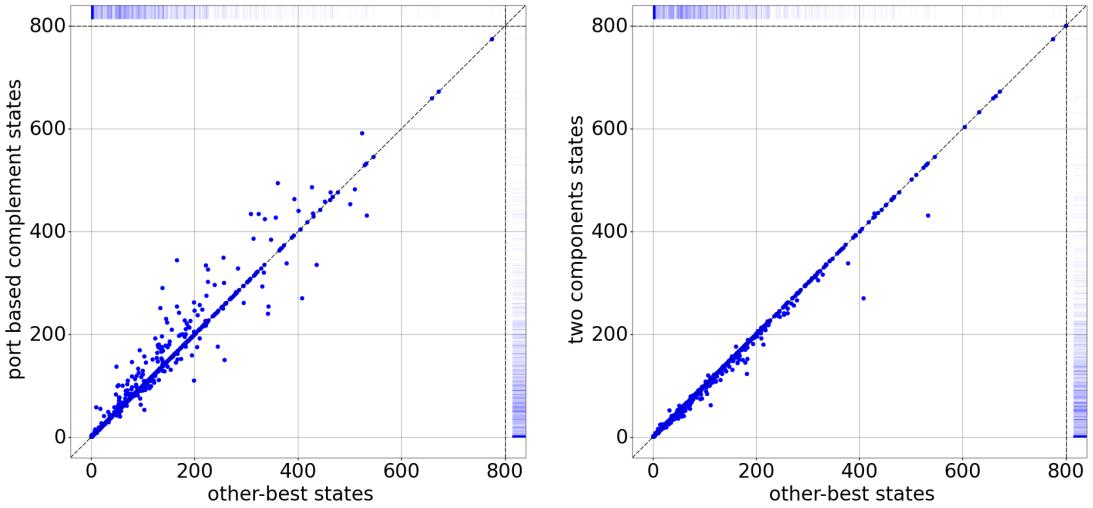
8.2 Náhodné automaty

Implementované algoritmy byly testovány na náhodně generovaných automatech. Tyto automaty byly generovány podle článku [14]. Článek uvádí několik parametrů při generování automatů. Každý automat $M = (Q, \Sigma, R, S, F)$ má pouze jeden iniciální stav ($|S| = 1$). Pro každý symbol $a \in \Sigma$ je vygenerován náhodný orientovaný graf, který má k hran, které odpovídají přechodům (q, a, q'). Poměr $r = \frac{k}{|Q|}$ se nazývá *hustota přechodů* pro symbol a. V tomto modelu jsou hustoty přechodů pro všechny symboly abecedy stejné. Dalším parametrem je $f = \frac{m}{|Q|}$, kde m je počet koncových stavů a f je označena *hustota koncových stavů*.

Tento model byl pro účely experimentů rozšířen o *hustotu iniciálních stavů* i, definovanou $i = \frac{n}{|Q|}$, kde n je počet iniciálních stavů. Minimální počet iniciálních stavů byl 1. Před vytvářením komplementu těchto automatů byly vždy odstraněny nedosažitelné stavy a stavy, ze kterých nelze přejít přes žádný řetězec do koncových stavů.

8.2.1 První test

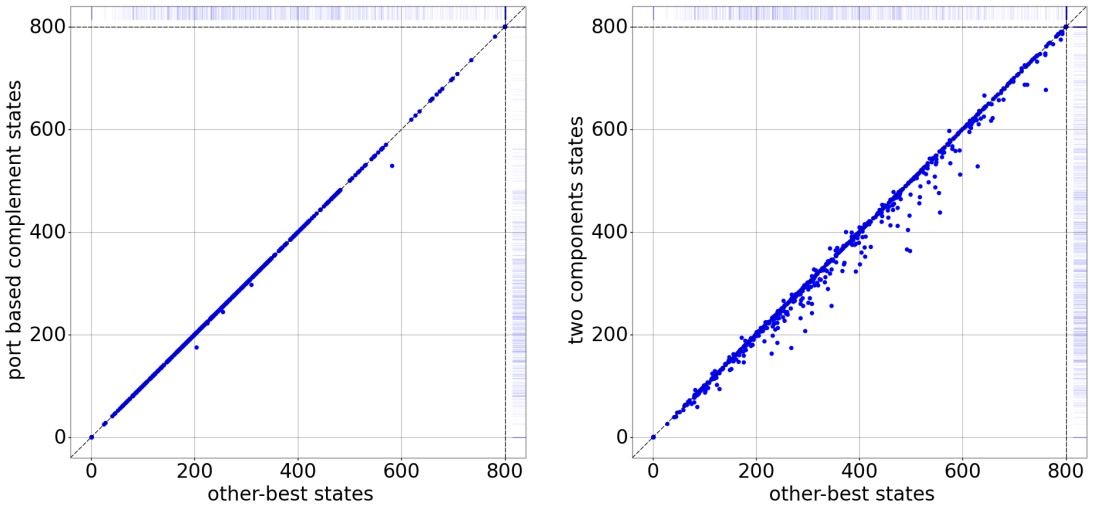
Obrázek 8.3 zobrazuje srovnání současných metod, *Komplement pomocí portů* a *Metodu dvou komponent*. U všech metod byly automaty redukovány pomocí nástroje RABIT. Bylo vygenerováno 1000 náhodných automatů s následujícími parametry: počet stavů $|Q| = 30$, hustota přechodů $r = 1$, hustota koncových stavů $f = 0.3$, počet iniciálních stavů $|S| = 1$ a velikost abecedy $|\Sigma| = 2$. Před samotným testováním byly odstraněny nedostupné stavy a stavy, ze kterých nejsou dostupné koncové stavy. To mělo z následek zpravidla nižší počet stavů než 30.



(a) *Komplement pomocí portů* (b) *Metoda dvou komponent*

Obrázek 8.3: Srovnání implementovaných metod na náhodně generovaných automatech. Na osách je zobrazen počet stavů. (1)

Obrázek 8.3a srovnává současné metody a *Komplement pomocí portů*. Všechny body pod diagonálou znázorňují automaty, pro které je nová metoda úspěšnější a produkuje menší automat. Jak z obrázku vyplývá, existuje mnoho případů, kdy je nová metoda lepší metoda. Obrázek 8.3b srovnává *Metodu dvou komponent* a z testů vyplývá, že téměř vždy je tato metoda lepší, rozdíly však nejsou tak velké.



(a) *Komplement pomocí portů* (b) *Metoda dvou komponent*

Obrázek 8.4: Srovnání implementovaných metod na náhodně generovaných automatech. Na osách je zobrazen počet stavů. (2)

8.2.2 Druhý test

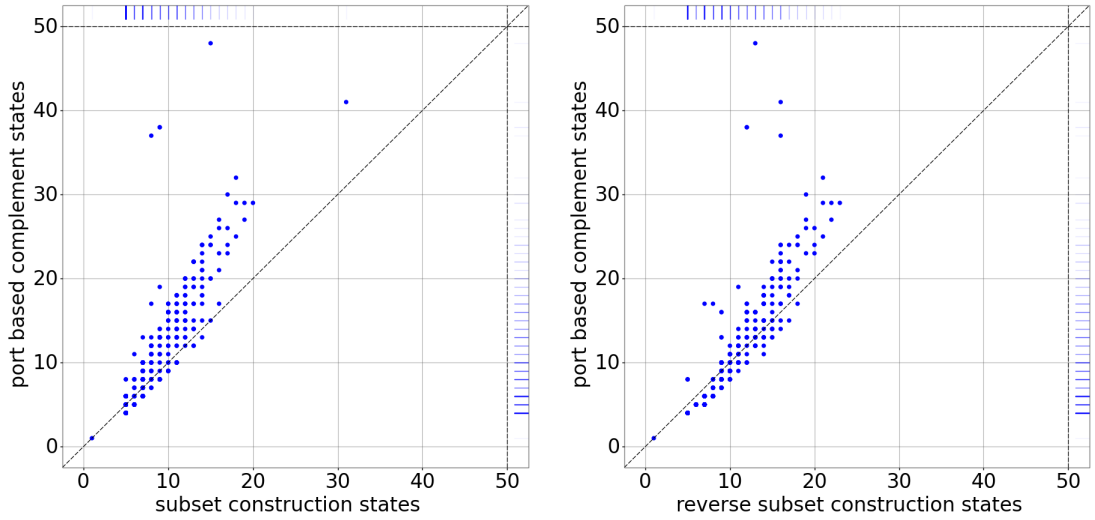
Na Obrázku 8.4 jsou zobrazeny výsledky testování automatů, které byly generovány s parametry: počet stavů $|Q| = 50$, hustota přechodů $r = 1$, hustota koncových stavů $f = 0.4$,

hustota iniciálních stavů $f = 0.3$ a velikost abecedy $|\Sigma| = 2$. Opět byly odstraněny nedosažitelné stavy a stavy, ze kterých nejde přejít do koncových stavů. Obrázek 8.4a zobrazuje výsledky *Komplement pomocí portů*. V tomto případě existuje pouze pár případů, kdy došlo ke zlepšení. Obrázek 8.4b zobrazuje výsledky *Metody dvou komponent*. Tato metoda produkuje v mnoha případech menší konečný automat než obě současné metody.

Hlavní nevýhodou *Komplement pomocí portů* je delší čas. V každé iteraci algoritmu je vytvořen zcela nový automat, která přijímá komplement jazyka zpracovaných komponent. V některých případech také vzniká mnoho stavů, které jsou mezi jednotlivými kroky odstraněny. Toto všechno přispívá k větší časové náročnosti, přestože je výsledný počet stavů menší než u současných metod.

8.3 Reálné automaty

Implementované algoritmy byly také otestována na reálných automatech, které vznikly jako výstup solveru MONA [8], který překládá formule logiky druhého řádu na konečné automaty. MONA tento automat analyzuje a dokáže rozhodnout zda je původní formule validní. Negace takové formule odpovídá komplementaci automatu.



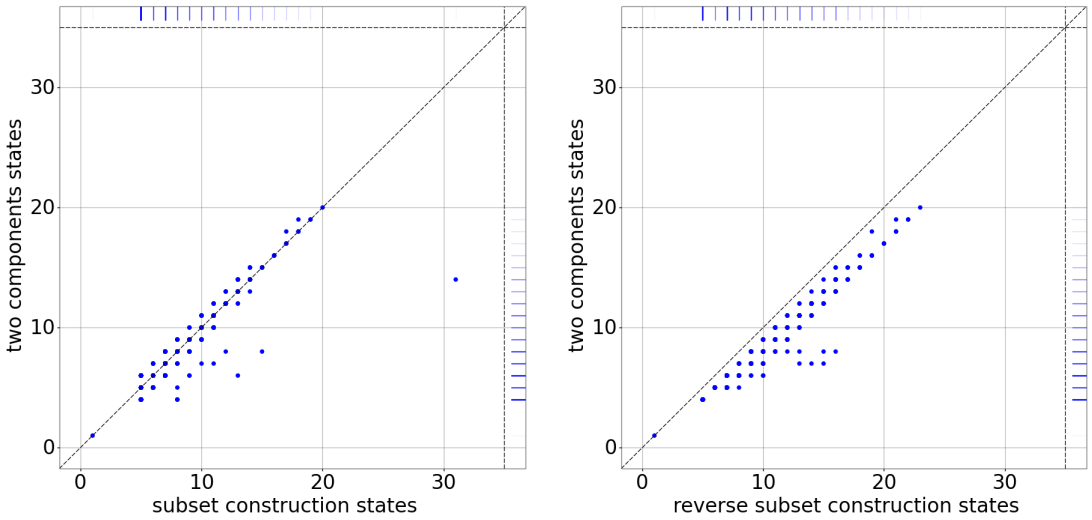
(a) *Komplement pomocí portů* a subset konstrukce

(b) *Komplement pomocí portů* a zpětná subset konstrukce

Obrázek 8.5: Srovnání *Komplement pomocí portů* se subset konstrukcí a zpětnou subset konstrukcí na automatech z nástroje MONA

Obrázek 8.5 zobrazuje srovnání počtu stavů dopředné a zpětné subset konstrukce s počtem stavů *Komplement pomocí portů*. *Komplementu pomocí portů* si vede ve většině případů hůř než současné metody. Hlavním důvodem je skutečnost, že nebylo využito žádné průběžně redukce. V každé iteraci algoritmu byly pouze odstraněny stavy, které nejsou koncové, a nelze dojít do žádného z koncových stavů. V každé iteraci algoritmu vznikají nové stavy při propojování jednotlivých komponent. Toto je nejvíc patrné u větších automatů, které se skládají z více komponent.

Na stejných automatech byla také otestována *Metoda dvou komponent*. Na Obrázku 8.6 je srovnána s oběma současnými metodami. Jak je z výsledku patrné tato metoda si ve většině případů stejně alespoň stejně dobře jako současné metody. V mnoha dalších případech si pak vede mnohem lépe.



(a) *Metoda dvou komponent* (b) *Metoda dvou komponent*

Obrázek 8.6: Srovnání *Metody dvou komponent* se subset konstrukcí a zpětnou subset konstrukcí na automatech z nástroje MONA

Kapitola 9

Závěr

V první části této práce byl představen algoritmus pro komplement konkatenace dvou jazyků. Jeho hlavní výhodou je, že druhý z jazyků může být komplementován jak dopřednou subset konstrukcí, tak i zpětnou subset konstrukcí a lze zvolit vhodnější ze dvou metod. Tento algoritmus nebyl z praktického hlediska příliš vhodný, protože ne každý konečný automat je možné rozdělit na více části tak, aby mezi těmito částmi platila konkatenace. Proto byl navržen algoritmus, který tuto myšlenku rozšiřuje a je založen na vstupních portech jednotlivých silně souvislých komponent nazvaný *Komplement pomocí portů*. Výhodou je, že pro tyto komponenty je možno využít různých metod komplementace a také je průběžně redukovat. Poslední představený algoritmus nazvaný *Metoda dvou komponent* rozdělí konečný automat na dvě komponenty, tak aby druhá z nich byla zpětně deterministická a největší možná. Tyto dvě komponenty jsou propojeny pomocí *Komplement pomocí portů*.

Poslední dva algoritmy byly implementovány a otestovány. Pro úzkou skupinu automatů přijímajících regulární jazyk tvaru $\Sigma^n a \Sigma^* a \Sigma^n$ je schopen algoritmus největší zpětně deterministické komponenty produkovat automaty, které mají pouze kvadratický nárůst počtu stavů. V případě použití redukcí komponent je nárůst počtu stavů jen lineární. Pro tyto jazyky je komplement vytvořený pomocí dopředné i zpětné subset konstrukce exponenciálně větší než původní vstupní automat.

Pro náhodně generované automaty dokážou být v některých případech představené algoritmy lepší než současné metody při použití metod pro redukci nedeterministických automatů. Nevýhodou je mnohdy vyšší čas. V případě testů na reálných automatech a srovnání bez použití redukcí si *Metoda dvou komponent* vedla obzvláště dobře. Prakticky ve všech případech byla stejně dobrá nebo i lepší.

Pokud by byla navržena varianta algoritmu, která pracuje po jednotlivých stavech a nikoli po silně souvislých komponentách, mohly by být představené metody výhodnější. Také by bylo možné zavést další optimalizace a odstraňovat neužitečné stavy průběžně.

Literatura

[1] *LanguageInclusion.org*. Dostupné z: http://www.languageinclusion.org/doku.php?id=tools.

[2] Abdulla, P. A., Chen, Y., Holík, L., Mayr, R. a Vojnar, T. When Simulation Meets Antichains. In: Esparza, J. a Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Springer, 2010, sv. 6015, s. 158–174. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-12002-2_14. Dostupné z: https://doi.org/10.1007/978-3-642-12002-2_14.

[3] Bondy, J. A. a Murty, U. S. R. *Graph Theory*. Springer, 2008. Graduate Texts in Mathematics. ISBN 978-1-84628-970-5. Dostupné z: https://doi.org/10.1007/978-1-84628-970-5.

[4] Brown, F. M. *Boolean reasoning - the logic of boolean equations*. Kluwer, 1990. ISBN 978-0-7923-9121-0.

[5] Brzozowski, J. Canonical regular expressions and minimal state graphs for definite events. In: *Mathematical Theory of Automata. MRI Symposia Series*. Polytechnic Press, 1962, s. 529–561.

[6] Glabbeek, R. J. van a Ploeger, B. Five Determinisation Algorithms. In: Ibarra, O. H. a Ravikumar, B., ed. *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings*. Springer, 2008, sv. 5148, s. 161–170. Lecture Notes in Computer Science. DOI: 10.1007/978-3-540-70844-5_17. Dostupné z: https://doi.org/10.1007/978-3-540-70844-5_17.

[7] Hoffman, C. *What Is a CSV File, and How Do I Open It?* How-To Geek, Apr 2018. Dostupné z: https://www.howtogeek.com/348960/what-is-a-csv-file-and-how-do-i-open-it/.

[8] Klarlund, N. a Møller, A. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`.

[9] Lengál, O. *Ondrik/automata-benchmarks*. Dostupné z: https://github.com/ondrik/automata-benchmarks/tree/master/vtf.

[10] Lengál, O. *Ondrik/libvata2*. Dostupné z: https://github.com/ondrik/libvata2.

[11] Lengál, O. *Ondrik/pycobench*. Dostupné z: https://github.com/ondrik/pycobench.

[12] Meduna, A. *Automata and languages - theory and applications*. Springer, 2000. ISBN 978-1-85233-074-3. Dostupné z: http://www.springer.com/computer/swe/book/978-1-85233-074-3.

[13] Strejcek, J., Lengál, O., Holík, L. a Major, J. *Nfa complement* [personal communication]. 2021. Dostupné z: https://github.com/VeriFIT/nfa-complement.

[14] Tabakov, D. a Vardi, M. Y. Experimental Evaluation of Classical Automata Constructions. In: Sutcliffe, G. a Voronkov, A., ed. *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*. Springer, 2005, sv. 3835, s. 396–411. Lecture Notes in Computer Science. DOI: 10.1007/11591191_28. Dostupné z: https://doi.org/10.1007/11591191_28.

[15] Tarjan, R. E. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1972, sv. 1, č. 2, s. 146–160. DOI: 10.1137/0201010. Dostupné z: https://doi.org/10.1137/0201010.

[16] Warner, S. *Pure mathematics for beginners: a rigorous introduction to logic, set theory, abstract algebra, number theory, real analysis, topology, complex analysis, and linear algebra*. 2018.