VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SROVNÁNÍ EFEKTIVITY RŮZNÝCH PROGRAMOVACÍCH JAZYKŮ PŘI PRÁCI S AUTOMATY

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ONDŘEJ POLANSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Mgr. LUKÁŠ HOLÍK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Polanský Ondřej**
Program: Informační technologie
Název: **Srovnání efektivity různých programovacích jazyků při práci s automaty**
Efficient Algorithms for Finite Automata
Kategorie: Algoritmy a datové struktury

Zadání:

1. Nastudujte základní automatové algoritmy (test prázdnosti, minimalizace, Booleovské operace) a alespoň dva pokročilé algoritmy, jako je výpočet relace simulace a test inkluze jazyků pomocí protiřetězců.
2. Implementujte vybrané algoritmy v alespoň třech, nejlépe čtyřech a více, programovacích jazycích.
3. Porovnejte a diskutujte výkonnost jednotlivých implementací a programátorskou pracnost v závislosti na jazyce.

Literatura:

1. Lucian Ilie, Gonzalo Navarro, Sheng Yu:On NFA Reductions. Theory Is Forever 2004: 112-124
2. Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, Tomáš Vojnar: When Simulation Meets Antichains. TACAS 2010: 158-174

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

Vedoucí práce: **Holík Lukáš, Mgr., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

V této práci jsou srovnány jazyky C++, C#, OCaml a Python na základě rychlosti, paměťové náročnosti a programátorské přívětivosti. Práce si klade otázku, jak moc se liší programy pracující s konečnými automaty, pokud jsou zapsané v různých jazycích. V každém jazyce je implementována stejná sada základních a pokročilých automatových algoritmů a následně je měřena jejich efektivita na vzorku 200 konečných automatů na unixovém operačním systému. Závěrem jsou prezentovány výsledky a je diskutována vhodnost jednotlivých jazyků pro práci s automaty. Tato práce může posloužit například při výběru jazyka pro tvorbu knihoven pro práci s automaty nebo při návrhu programů a prototypů algoritmů pracujících s automaty.

Abstract

This thesis compares languages C++, C#, OCaml and Python based on speed, memory requirements and programming comfort. The goal of this thesis is to find out how much does the choice of a certain programming language impact the performance of programs working with finite automata. The same set of basic and advanced automata algorithms was implemented in each language and their efficiency was measured on a sample of 200 finite automata using a unix based operating system. Finally, the author presents results and discusses suitability of each language for work with finite automata. This thesis can help with selecting an appropriate programming language for multitude of purposes, including development of automata algorithm libraries or the process of designing programs and prototypes that work with finite automata.

Klíčová slova

programovací jazyky, C++, C#, OCaml, Python, efektivita, měření rychlosti, konečné automaty, automatové algoritmy

Keywords

programming languages, C++, C#, OCaml, Python, efficiency, speed measurement, finite automata, automata algorithms

Citace

POLANSKÝ, Ondřej. *Srovnání efektivity různých programovacích jazyků při práci s automaty.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Mgr. Lukáš Holík, Ph.D.

Srovnání efektivity různých programovacích jazyků při práci s automaty

Prohlášení

Prohlašuji, že jsem bakalářskou práci *Srovnání efektivity různých programovacích jazyků při práci s automaty* vypracoval samostatně pod vedením Mgr. Lukáše Holíka, Ph.D. a v seznamu literatury jsem uvedl všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Ondřej Polanský
20. května 2020

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu Mgr. Lukáši Holíkovi, Ph.D. za odborné vedení a cenné rady při tvorbě této práce.

Obsah

1 Úvod **2**

2 Konečné automaty **5**
2.1 Definování pojmů 5
2.2 Datové struktury pro automaty a jejich implementace 6

3 Algoritmy pro konečné automaty a implementační jazyky **10**
3.1 Algoritmy pro konečné automaty 10
3.1.1 Algoritmus 1: Test prázdnosti 11
3.1.2 Algoritmus 2: Odstranění zbytečných stavů 12
3.1.3 Algoritmus 3: Průnik (Produkt) 13
3.1.4 Algoritmus 4: Determinizace KA 15
3.1.5 Algoritmus 5: Minimalizace DKA 16
3.1.6 Algoritmus 6: Výpočet relace simulačního předuspořádání 19
3.1.7 Algoritmus 7: Test univerzality 21
3.1.8 Algoritmus 8: Test inkluze 23
3.2 Implementační jazyky 24

4 Měření efektivity programovacích jazyků a výsledky **27**
4.1 Popis měření 27
4.2 Výsledky 28
4.2.1 Test prázdnosti 28
4.2.2 Odstranění zbytečných stavů 31
4.2.3 Produkt 33
4.2.4 Determinizace 35
4.2.5 Minimalizace 37
4.2.6 Výpočet relace simulace 39
4.2.7 Test univerzality 41
4.2.8 Test inkluze 43
4.2.9 Posloupnost operací 47
4.3 Zhodnocení jazyků 49

5 Závěr **54**

Literatura **55**

Kapitola 1

Úvod

Měření doby běhu programů a jejich paměťových nároků je jednou ze základních metod posuzování efektivity algoritmů, jazyků a téměř všeho ostatního v informatice. Každý program lze totiž vnímat jako krabičku, která nějakou posloupností kroků převádí vstupní hodnoty na hodnoty výstupní. Pokud krabička funguje správně a výstupní hodnoty nejsou chybné, jde již jen o to, jak rychle je toho schopná a kolik k tomu potřebuje zdrojů (paměti). Tato práce si klade otázku, jak moc se liší takové programy, pokud jsou napsané v různých programovacích jazycích a vstupními hodnotami jsou konečné automaty.

Konečný automat je jednou z významných kapitol informatiky s obrovským využitím napříč informatickými odvětvími, ale i různými vědními disciplínami. Využití má třeba ve zpracování textu a regulárních výrazů, návrhu systémů, které v reakci na uživatelský vstup mění svůj stav, lexikální a syntaktické analýze překladačů nebo lingvistice či elektrotechnice (teoreticky je počítač vlastně velký konečný automat). Konečný automat je matematický výpočetní model, skládající se z konečného počtu stavů, z nichž právě jeden je aktivní. Aktivní stav se mění čtením znaků ze vstupu. Konečných automatů existuje velké množství typů, které jsou podrobněji definovány níže v kapitole 2.1. Některé typy automatů jsou pro určité úlohy vhodnější než jiné, z tohoto důvodu existují automatové algoritmy pro jejich konverzi nebo určení.

Jedním ze základních problémů při práci s automaty je jejich velikost, tzn. počet stavů a přechodových pravidel, která může být značná. Některé algoritmy mohou navíc velikost ještě výrazně zvýšit. Při implementaci algoritmů pracujících s automaty je tedy nutné dbát jednak na jejich efektivitu, aby výpočet netrval neúnosně dlouhou dobu, na druhou stranu může výrazně pomoci i výběr vhodného implementačního jazyka, na což je tento text zaměřen. V současné době existují práce věnující se srovnání efektivity jazyků v různých oborech a kontextech, například [3] nebo [7], bohužel ale neexistují žádná srovnání ve specifickém kontextu konečných automatů. Tato práce si dává za cíl tento stav napravit. Pro tento účel byly vybrány čtyři populární běžně používané programovací jazyky, a to C++, Python, C# a OCaml, jejichž efektivita při práci s automaty byla testována na sadě základních automatových algoritmů.

Výsledkem této práce je srovnání vybraných programovacích jazyků, které může posloužit například pro výběr jazyka při návrhu knihoven pro práci s automaty, případně při vývoji nebo úpravě samotných automatových algoritmů. Tehdy bude srovnání užitečné pro návrh a tvorbu prototypu řešení, pomocí kterého bude možné experimentálně odhalovat chyby, porovnávat efektivitu jednotlivých verzí algoritmu nebo vizualizovat průběh výpočtu pro lepší porozumění problému. V případě návrhu knihoven nebo výběru jazyka pro náročné automatové výpočty bude jistě hlavním parametrem rychlost a spotřeba paměti, při tvorbě

testovacího prototypu pak bude nejspíš záležet i na přímočarosti a rychlosti implementace, rychlému ladění, jednoduchosti modifikací v závislosti na vývoji algoritmu a obecné programátorské přívětivosti. V této práci jsou tedy jazyky hodnoceny na základě všech těchto kritérií, které jsou závěrem přehledně shrnuty a je posouzena a diskutována jejich vhodnost pro výše zmíněné účely.

Bylo zjištěno, že z pohledu rychlosti a spotřeby paměti je nejlepší C++, a to s poměrně výrazným náskokem na druhý OCaml (2,25 krát pomalejší) a třetí C#, u kterého je srovnání tohoto typu obtížnější a je tedy popsáno v hlavní části práce. Odhadem je 3-4 krát pomalejší než C++. Python v testech příliš neuspěl (8,72 krát pomalejší než C++). Z hlediska paměti bylo s mírným náskokem nejšetrnější C++, avšak OCaml byl jen o trochu horší. Dále byly zaznamenány velké komfortní rozdíly mezi C++ a ostatními třemi jazyky hlavně díky moderním prvkům, které citelně zjednodušují a urychlují práci.

Související práce

Je bohužel jen málo aktuálních odborných prací, věnujících se srovnání efektivity jazyků. Jednou z nich je [14], která také naznačuje rychlostní dominanci C++, jazyk C# uvádí jako mírně rychlejší než je výše uvedený odhad a jazyk Python má být údajně 46 krát pomalejší než C++, což je o hodně horší než v této práci. Problém je, že zde vůbec není uveden postup měření, implementace, ani testovací sada algoritmů. Tyto výsledky tedy nejsou příliš významné. Další práce věnující se srovnání jazyků je [7], která srovnává mimo jiné i jazyky C++, C# a Python z pohledu rychlosti. Pořadí jazyků je C++, C# a Python poslední, co je ovšem zajímavé je zjištění, že C# měl na unixovém operačním systému poměrně nestabilní výkonnost, což naznačují i výsledky této práce. Co také částečně odpovídá je spotřeba paměti C#, která v citovaném srovnání byla údajně přibližně 20 krát větší než C++. I v této práci totiž dosahoval C# nejhorších výsledků. [7] také naznačuje, že semi-kompilované jazyky jsou téměř stejně rychlé jako kompilované jazyky, což zde zástupci těchto dvou skupin (C++ a C#) příliš nepotvrdili, neboť rozdíly mezi těmito dvěma jazyky byly značné. Na druhou stranu se citovaná práce shoduje s touto v tom, že C# je kompromisem mezi jednoduchostí Pythonu a rychlostí C++, což je i jeden ze zdejších závěrů. Autor citované práce dokonce srovnával jazyky podle počtu řádků. Dle jeho závěrů jich mělo C++ největší počet a C# s Pythonem stejný, o něco nižší počet. V této práci byly počty řádků C++ a C# přibližně stejné (přibližně 2100) a zdrojový kód Pythonu byl o mnoho kratší (přibližně 1300 řádků). Tento údaj sice není příliš spolehlivý, protože záleží na mnoha různých faktorech, například na programovacím stylu nebo míře nezkrotnosti potřeby komentovat, ale i tak je velmi překvapivé, že citovaný autor dokázal vytvořit stejně dlouhé zdrojové kódy v jazyce se závorkováním a bez závorkování.

Organizace členění práce

Práce je organizována následovně. Po úvodu následuje kapitola 2, kde jsou uvedeny základní definice týkající se konečných automatů. Vzápětí jsou uvedeny datové struktury, které automaty v této práci reprezentují, a jejich implementace v jednotlivých jazycích. Následuje kapitola 3, která v sekci 3.1 obsahuje popis použitých algoritmů a specifika jejich implementace a v sekci 3.2 se nachází základní informace o implementačních jazycích. Autor zastává názor, že v práci tohoto typu je lepší, aby detaily implementace daného algoritmu následovaly ihned za jeho teoretickým vysvětlením. Jednotlivé algoritmy jsou totiž na

sobě nezávislé a vystupují samostatně. Umožní to jejich přirozenější pochopení v kontextu specifické reprezentace automatů v této práci. Ze stejného důvodu je popis implementace konečného automatu zařazen hned za jeho teoretické definování. Výsledky měření pro jednotlivé algoritmy jsou prezentovány v sekci 4.2 a v sekci 4.3 je uvedeno celkové zhodnocení jazyků. Následuje závěr, ve kterém je práce shrnuta.

Kapitola 2

Konečné automaty

V této kapitole je nejprve definován konečný automat společně se souvisejícími pojmy a následně je popsán jeho návrh a implementace v této práci.

2.1 Definování pojmů

Zde jsou uvedeny základní definice a pojmenování symbolů, které jsou používány v této práci k popisu automatů a algoritmů.

Nechť Σ je *abeceda* symbolů a Σ^* množina všech *řetězců (slov)* nad abecedou Σ včetně *prázdného řetězce*, který je značen ε. Symbol Σ^+ značí množinu $\Sigma^* - \{\varepsilon\}$. *Jazyk* L nad abecedou Σ je definován jako jakákoli podmnožina $L \subseteq \Sigma^*$.

Konečný automat bez ε-přechodů (zkráceně KA) je pětice $M = \{Q, \Sigma, \delta, S, F\}$, kde Q je konečná množina *stavů*, Σ je *vstupní abeceda* automatu, S značí množinu *počátečních stavů* a F značí množinu *koncových stavů* automatu. δ je konečná množina *přechodů* $pa \to q$, kde $p,\ q\ \in\ Q$, $a\ \in \Sigma$, což lze matematicky zapsat jako relaci $\delta\ \subseteq\ Q \times \Sigma \times Q$, tedy množinu trojic (p, a, q). Pokud $a \in \Sigma$, je definována funkce $\delta(p, a)$ jako množina všech stavů q, pro které existuje přechod $pa \to q$. Pokud $w\ \in\ \Sigma^*$ a $w_0 \dots w_n$ jsou jednotlivá písmena řetězce, je definována funkce $\delta(p, w)$ jako množina všech stavů q, pro které existuje sekvence přechodů $pw_0 \to p_1, p_1w_1 \to p_2, \dots, p_{n-1}w_{n-1} \to q$. Jelikož všechny algoritmy prezentované v této práci pracují s konečnými automaty bez ε-přechodů, zkratka KA, stejně jako označení konečný automat, je v dalších částech práce využívána k označení konečných automatů bez ε-přechodů.

Nechť M je KA, p a $q \in Q$ a platí, že $\forall p : \forall a \in \Sigma$ existuje nejvýše jedno q takové, že $pa \to q\ \in \delta$, a množina S počátečních stavů obsahuje nejvýše jeden stav, pak M nazýváme *deterministický* konečný automat (zkráceně DKA). Jestliže pro všechny stavy $p \in Q$ automatu M platí, že pro každé $a\ \in \Sigma$ existuje stav $q \in Q$ takový, že přechod $pa \to q \in \delta$, pak takový KA nazýváme *úplný*. Je-li M deterministický KA a zároveň platí, že neexistuje žádný jiný ekvivalentní deterministický KA N s menším počtem stavů a $L(M) = L(N)$, je M *minimální* DKA.

Nechť M je KA. Stav $p \in Q$ je *dostupný*, pokud $p \in S$ nebo existuje sekvence přechodů $x_0a_0 \to x_1, x_1a_1 \to x_2, \dots, x_{n-1}a_{n-1} \to p \in \delta$, $n \in \mathbb{N}$ a $x_0 \in S$. Jinak je p *nedostupný.* Stav $p \in Q$ je *ukončující*, pokud $p \in F$ nebo existuje sekvence přechodů $pa_0 \to x_1, x_1a_1 \to x_2, \dots, x_{n-1}a_{n-1} \to x_n \in \delta$, $n \in \mathbb{N}$ a $x_n \in F$. Jinak je p *neukončující.*

Nechť M je KA a w je řetězec nad abecedou Σ. *Jazyk* přijímaný automatem M je definovaný jako $L(M) = \{w | w \in \Sigma^*, f = \delta(s, w), f \in F, s \in S\}$. Konečný automat M

je univerzální, pokud $L(M) = \Sigma^*$. Dále je pro stav $q \in Q$ definován $L(M)(q) = \{w|w \in \Sigma^*, f = \delta(q, w), f \in F\}$ (jazyk stavu q v M) a $L_{pref}(M)(q) = \{w|w \in \Sigma^*, q = \delta(s, w), s \in S\}$ (všechny prefixy jazyka stavu q v M). Místo $L(M)(q)$ může být dále v textu používáno značení $L_M(q)$ znamenající totéž nebo $L(q)$, pokud je z kontextu zřejmé, který automat má autor na mysli.

Reverzní automat M^r automatu M je definován $M^r = \{Q, \Sigma, \delta^r, F, S\}$, kde došlo k výměně množin koncových a počátečních stavů a $\delta^r = \{qa \to p|pa \to q \in \delta\}$. Nechť $A = \{Q_A, \Sigma, \delta_A, S_A, F_A\}$ a $B = \{Q_B, \Sigma, \delta_B, S_B, F_B\}$ jsou KA. Jejich sjednocení $A \cup B = \{Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, S_A \cup S_B, F_A \cup F_B\}$ a průnik $A \cap B = \{Q, \Sigma, \delta, S, F\}$, kdy $L(A \cap B) = L(A) \cap L(B)$ a stavy průniku jsou podmnožinou $Q_A \times Q_B$ takové, že $(p, p')a \to (q, q') \iff (pa \to q) \wedge (p'a \to q')$ pro $p, q \in Q_A$, $p', q' \in Q_B$ a $a \in \Sigma$.

2.2 Datové struktury pro automaty a jejich implementace

Pro všechny algoritmy byly použity stejné datové struktury reprezentující konečné automaty, ne všechny algoritmy však využívají všechny jejich proměnné.

Pro uchování konečných automatů byly použity dvě základní struktury, moduly nebo třídy (podle jazyka). První z nich, nazvaná *FA*, reprezentuje samotný konečný automat a obsahuje proměnné:

- **name** – Řetězec určující jméno KA.
- **states** – Neseřazená množina stavů KA (struktur *State*).
- **alphabet** – Seřazená množina určující abecedu KA. Za písmeno abecedy se považuje jakýkoli řetězec. Jelikož žádný z implementovaných algoritmů nemění abecedu KA, je množina seřazena pouze jednou při načítání KA a není tedy třeba vynakládat žádné další prostředky pro udržení seřazenosti množiny v průběhu provádění algoritmů. Seřazení této množiny umožňuje rychlé ověření rovnosti abeced dvou automatů, což je využíváno algoritmy pracujícími se dvěma KA. Jedinou výjimku představují algoritmy, které tvoří nový automat, a tedy i vytváří novou množinu abecedy. To je ovšem zajištěno pomocí mělké nebo hodnotové (C++) kopie celé množiny.
- **start_states** – Neseřazená množina počátečních stavů. Žádný z implementovaných algoritmů v této množině nevyhledává, pouze jí iteruje, není tedy třeba množinu udržovat seřazenou. Ke zjištění, zda je stav počáteční, slouží speciální proměnná ve stuktuře State.
- **final_states** – Neseřazená množina koncových stavů. Stejná funkce jako množina *start_states*.

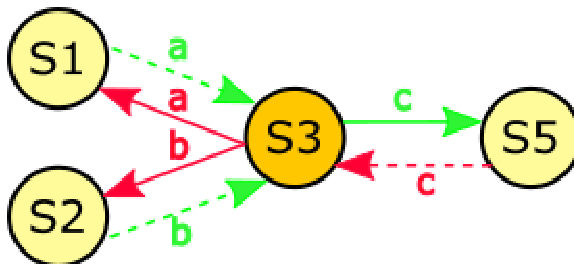
Druhá z nich, nazvaná *State*, reprezentuje stav z množiny Q v KA a obsahuje proměnné:

- **name** – Řetězec určující jméno stavu. Jméno stavu je jeho jedinečný identifikátor v rámci daného KA.
- **transit_states** – Množina přechodů, tedy dvojic klíč-hodnota (písmeno abecedy – odkaz na stav) v C++ nebo množina dvojic klíč-seznam hodnot (písmeno abecedy – množina odkazů na stavy). Obsahuje odkazy na následnické stavy dosažitelné z daného

stavu pomocí jednotlivých písmen abecedy. Musí být optimalizovaná pro velké množství vyhledávání, ovšem nepředpokládá se, že bude příliš velká, neboť obsahuje pouze přechody z jednoho stavu.

- **reversed_transit_states** – Obdoba množiny *transit_states* pro reverzní přechody.
- **start_st** – Booleovská proměnná indikující, že je stav počáteční.
- **final_st** – Booleovská proměnná indikující, že je stav koncový.
- **flag, card** – Pomocné proměnné, využívané některými algoritmy.

Struktura FA neobsahuje množinu přechodů mezi stavy δ přímo, přechody jsou kódovány pomocí množiny transit_states. Důvodem je zjištění, že implementované algoritmy po zpracování aktuálního stavu vyhledávají jeho následníky, tedy stavy, do kterých z aktuálního stavu vede přechod. Není tedy třeba procházet všechny přechody automatu, stačí projít jen přechody vycházející z daného stavu. Je oprávněné se domnívat, že takováto úprava výrazně zvýší efektivitu algoritmů při práci s KA s velkým množstvím přechodů a stavů, neboť urychlí dvě základní operace prováděné algoritmy – nalezení všech následníků daného stavu a nalezení následníků daného stavu dostupných daným písmenem abecedy. Množina přechodů δ tedy byla distribuována do jednotlivých stavů, kdy každý stav obsahuje množinu odkazů na následnické stavy a množinu odkazů na předchozí stavy (stejný princip pro množinu reverzních přechodů δ^r). Vznikla tak obousměrně propojená síť stavů (znázorněno v obrázku 2.1). Tato modifikace sice mírně zvyšuje náročnost úprav KA (přidávání a odebírání stavů) pro některé algoritmy, konkrétně algoritmy odebírání zbytečných stavů a minimalizace KA, panuje však přesvědčení, že zisk převyšuje ztráty. Jako možné rozšíření této práce se nabízí ověření, zdali je tomu opravdu tak.



Obrázek 2.1: **Síť stavů.** Zeleně jsou vyznačeny přechody, červeně reverzní přechody. Plná čára – přechody stavu S3, přerušovaná čára – přechody okolních stavů.

Celková myšlenka této reprezentace KA spočívá v tom, že jednotlivé stavy se všemi svými proměnnými budou uloženy pohromadě na jednom místě v množině states a všechny ostatní proměnné struktur FA a State se budou do této množiny odkazovat. V závislosti na implementačním jazyce byly pro odkazy použity ukazatele (C++) nebo princip referenčního typu spojený s automatickou správou paměti (ostatní jazyky). Tento přístup nejenže šetří paměť a odstraňuje redundanci jednotlivých stavů, ale hlavně řeší problém se zacyklením v případě přechodů typu $pa \to p$. Z tohoto důvodu musí být množina states neseřazená a pořadí prvků se nesmí nekontrolovaně měnit. Tento požadavek nemá žádný negativní dopad na efektivitu, protože v množině stavů žádný zadaný algoritmus nevyhledává.

Množiny start_states a final_states ve struktuře FA neslouží k ověřování, zdali je stav počáteční nebo koncový, k tomuto účelu jsou určené booleovské proměnné start_st a final_st, které obsahuje každý stav zvlášť. Tento přístup sice přidává menší redundanci,

avšak zajišťuje jak rychlé ověření, jestli je stav počáteční nebo koncový, bez nutnosti prohledávání obou množin, tak také rychlou iteraci přes počáteční a koncové stavy bez nutnosti procházení všech stavů. Obě tyto operace jsou algoritmy hojně využívány.

Datové struktury použité v jednotlivých jazycích

Zde jsou popsány implementace výše uvedených datových struktur v jednotlivých jazycích. Jelikož většina algoritmů využívá množinu, do které se ukládají prvky ke zpracování, a která bývá nejčastěji implementována pomocí zásobníku nebo fronty, jsou zde popsány i časové složitosti těchto struktur. Údaje o časových složitostech byly získány z dokumentací jednotlivých jazyků, dostupných na internetu [1], [15], [13] a [10].

V jazyce C++ byl pro neseřazené množiny použit vektor. Jedinou výjimkou je množina stavů states, která v implementaci s ukazateli musí být lineárně zřetězeným seznamem kvůli nutnosti odebírat a přidávat stavy bez porušení platnosti odkazů ostatních stavů. Vektor má konstantní časové složitosti pro vyhledání pomocí indexu a vkládání na konec (O(1)), odstranění prvku má lineární složitost (O(n)), avšak odstranění z konce, které se používá převážně, má složitost O(1). Lineárně zřetězený seznam má konstantní časové složitosti pro vkládání a odstraňování prvků. Jelikož se v množině stavů nikdy nevyhledává, nevadí ani lineární složitost vyhledávání. Množina abecedy je reprezentována seřazeným vektorem. Vektor je řazen pouze jednou při načítání automatu ze souboru mimo měřené úseky. Množiny přechodů jsou implementovány pomocí Multimapy, jelikož hashovací tabulka v C++ nezachovává pořadí vložených prvků a je složitější vyhledat konkrétní stav pro konkrétní písmeno abecedy. Nabízela se i možnost vytvořit hashovací tabulku množin odkazů na stavy, ale nakonec byla vybrána jednodušší a specializovaná třída Multimap. Vyhledání, vkládání a odstranění prvků má tedy logaritmickou časovou složitost, při odstraňování se ovšem často používá odstranění na základě iterátoru, které má složitost konstantní. Předpokládá se, že množství přechodů každého stavu v drtivé většině případů nepřesáhne 50–100, rozdíl mezi konstantní složitostí hashovací tabulky, která ovšem musí vypočítat hash pro každý prvek, a logaritmickou složitostí multimapy tedy bude zanedbatelný. Algoritmy použité v této práci pro měření výsledků měly průměrně méně než 10 přechodů na jeden stav. Dále jsou použity třídy Queue a Stack, které mají konstantní složitosti vkládání a odebírání prvků. Množina card je uložena ve třídě Map, standardně jsou prováděny operace vkládání a vyhledání se složitostí O(log_n). Množina konečných stavů je uložena ve třídě Set – O(log_n) pro vkládání, vyhledání a odstranění prvku.

V jazyce Python 3 je využíván převážně List (množina stavů, abecedy, ...). Je snaha používat jen operace se složitostí O(1), tedy vložení na konec, odebrání z konce nebo vyhledání prvku pomocí indexu. Odebírání prvků z množiny stavů tedy probíhá tak, že je nejdříve odstraňovaný prvek vyměněn s posledním a poté je smazán z konce Listu. Bohužel bylo nutné občas použít i neefektivní Listové operace jako průnik (O(n^2)) a rozdíl množin (O(n)). List je efektivně použit i pro implementaci zásobníku. Fronta je implementována pomocí třídy Queue, která nabízí konstantní O(1) složitosti pro vkládání a odebírání prvků. Množiny konečných a počátečních stavů jsou implementovány pomocí třídy Set, která má operace vkládání a odebírání také konstantní. Složitost iterace je O(n). Množiny přechodů jsou uloženy ve Slovníku s formátem písmeno abecedy (klíč) – list stavů dostupných daným písmenem (hodnota). Vkládání a vyhledání množiny stavů je tedy průměrně konstantní, ovšem vyhledání konkrétního stavu v dané množině už je nejhůře lineární, tedy O(n) kdy n je velikost množiny. To by opět nemělo příliš vadit, jelikož se jedná o přechody pouze jednoho stavu a ne celého automatu. Množina card je také implementována pomocí Slovníku.

V C# byly struktury FA a State implementovány velmi podobně jako v Pythonu. Často je využíván List (množiny stavů, počátečních stavů a koncových stavů), který má časovou složitost O(1) pro vkládání na konec, odstraňování z konce a vyhledávání pomocí indexu. Iterace má lineární časovou složitost. Dále jsou používány třídy Queue a Stack, opět se složitostí O(1) pro vkládání a odebírání prvků. Přechody pro jednotlivé stavy jsou uloženy ve třídě Dictionary (stejným způsobem, jako v Pythonu), která má průměrnou složitost O(1) pro vkládání, odebírání a vyhledávání prvků. Množina card byla také implementována pomocí třídy Dictionary. Abeceda byla uložena ve třídě SortedSet, ovšem vždy je vytvořena při načítání automatů a v měřených úsecích již není nijak upravována. Její časová složitost iterace je O(n). Všechny časové složitosti pro vkládání jsou amortizované, při naplnění kontejneru uvedených tříd je třeba ho zvětšit s lineární časovou složitostí.

V jazyce OCaml byl pro množiny stavů, počátečních stavů a koncových stavů použit modul List se stejnými vlastnostmi jako obecný lineárně zřetězený seznam. Vložení a odstranění prvku má konstantní časovou složitost, vyhledání a iterace má složitost O(n). Fronta a zásobník byly implementovány pomocí rekurze v rámci implementací jednotlivých algoritmů. Pro uložení abecedy byl použit Set řetězců, prvky se pouze iteruje (O(n)). Přechody pro každý stav byly uloženy v modulu Hashtbl (opět stejným způsobem, jako v Pythonu), který má stejné časové náročnosti jako Pythonovský Slovník. Jedinou změnou oproti ostatním jazykům bylo to, že z důvodu snahy o funkcionální řešení algoritmů nebyly umístěny pomocné proměnné card a flag v jednotlivých stavech, ale místo toho byly obě proměnné uloženy do jedné hashovací tabulky pro všechny stavy.

Dále jsou v jednotlivých algoritmech využívány specifické pomocné struktury, které budou popsány pro každý algoritmus zvlášť.

Kapitola 3

Algoritmy pro konečné automaty a implementační jazyky

V této kapitole je čtenář nejdříve seznámen s automatovými algoritmy použitými v této práci a jejich implementací. Ve druhé části jsou uvedeny základní informace o vybraných implementačních jazycích, které byly testovány.

3.1 Algoritmy pro konečné automaty

Bylo vybráno celkem osm testovacích algoritmů, z toho pět základních – Test prázdnosti, Odstranění zbytečných stavů, Průnik (produkt), Determinizace KA, Minimalizace DKA a tři pokročilé – Výpočet relace simulace, Test univerzality, Test inkluze. Tento soubor obsahuje běžné automatové algoritmy od nejjednodušších až po velmi komplikované. Čtyři algoritmy pouze prochází stavy automatu a čtyři za běhu upravují starý nebo vytváří nový automat. Tato práce a algoritmy v ní používají pouze KA bez epsilon přechodů. Dále platí, že algoritmy, které mění nebo konstruují KA, jsou implementovány tak, aby nový automat opravdu vytvořily a ne jen vypsaly na výstupní terminál. Pro popis algoritmů je využit pseudokód založený na programovacím jazyce C. Názvy použitých datových struktur v jednotlivých jazycích a informace o nich byly převzaty z dokumentací [1], [15], [10] a [13].

3.1.1 Algoritmus 1: Test prázdnosti

Vstup: KA M = $\{Q, \Sigma, \delta, S, F\}$
Výstup: *True* pokud neexistuje cesta z počátečního stavu do koncového, jinak *False*.

1. W = NEWSTACK()
2. Visited = NEWHASHTABLE()
3. **for** $s \in S$ **do**
4. vlož s do Visited
5. vlož s do W
6. **while** $W \neq \varnothing$ **do**
7. odstraň stav s z W
8. **if** $s \in F$ **then return** False
9. **else**
10. **for all** $a \in \Sigma$, $s' \in \delta(s, a)$ **do**
11. **if** s' **not in** Visited **then**
12. vlož s' do Visited
13. vlož s' do W
14. **return** True

Algoritmus pro test prázdnosti ověřuje, zda jazyk KA M, $L(M) = \varnothing$. V takovém případě vrací hodnotu True, jinak vrací False. Jazyk KA je prázdný, pokud neexistuje žádná posloupnost přechodů z počátečního stavu do stavu koncového. Tento algoritmus byl navržen autorem práce.

Test prázdnosti je jeden z jednodušších algoritmů, jedná se v podstatě o algoritmus prohledávání do hloubky (Depth-first search), začínající z počátečních stavů. Počáteční stavy z množiny S jsou na začátku naskládány do zásobníku (řádky 2–4). Algoritmus následně pomocí přechodů hledá koncový stav a v případě, že ho najde, prohlásí jazyk KA za neprázdný. Žádný stav není navštíven dvakrát díky vyhledávání v tabulce *Visited*, do které je každý již navštívený stav uložen. Nehrozí tedy zacyklení. Jelikož KA má konečný počet stavů a žádný stav není možno zpracovat vícekrát než jednou, algoritmus musí po konečném počtu kroků skončit. Jazyk KA je prohlášen za prázdný, pokud je množina stavů ke zpracování (W) prázdná a nebyl nalezen koncový stav.

Za zmínku stojí druhá varianta algoritmu, a to prohledávání do šířky (Breadth-first search). V případě, že $L(M) = \varnothing$, není mezi těmito variantami žádný rozdíl. V obou případech je třeba zpracovat všechny dostupné stavy. Pokud je $L(M)$ neprázdný a neobsahuje žádné neukončující stavy, pak je ve většině případů lepší prohledávání do hloubky, neboť jakákoli cesta vede do konečného stavu. Jedinou výjimkou je KA s extrémními rozdíly v délkách cest z počátečních stavů do stavů koncových. Nespornou výhodu má ovšem prohledávání do šířky v případě, kdy $L(M)$ je neprázdný a KA obsahuje velké množství slepých (neukončujících) větví. I přes to se prohledávání do šířky jeví jako horší varianta, protože vítězí jen v extrémních a nepravděpodobných případech. Jako možné překlenutí rozdílů mezi těmito přístupy se nabízí přesunout řádek 8 za řádek 10 a zaměnit s za s'. Prohledávalo by se tak stále do hloubky, ale test konečnosti stavů by byl prováděn do šířky při každém kroku algoritmu.

Pro implementaci Visited byl použit Unordered_set (C++), Dictionary (Python), HashSet (C#), Hashtbl (OCaml). Pořadí prohledávání stavů se liší mezi jazyky, například v OCamlu se prohledávají stavy v opačném pořadí než v ostatních jazycích, protože je algoritmus implementován pomocí rekurze.

3.1.2 Algoritmus 2: Odstranění zbytečných stavů

Vstup: KA M = $\{Q, \Sigma, \delta, S, F\}$
Výstup: KA M = $\{Q, \Sigma, \delta, S, F\}$ bez zbytečných stavů.

1. W = NEWQUEUE()
2. Visited1 = NEWHASHTABLE()
3. Visited2 = NEWHASHTABLE()
4. **for** $s \in S$ **do**
5. vlož s do Visited1
6. vlož s do W
7. **while** $W \neq \varnothing$ **do**
8. odstraň stav s z W
9. **for all** $a \in \Sigma$, $s' \in \delta(s, a)$ **do**
10. **if** s' **not in** Visited1 **then**
11. vlož s' do Visited1
12. vlož s' do W
13. opakuj kroky 4–12 pro reverzní automat, místo do Visited1 vkládej do Visited2
14. Restore_FA()
15. **return** M

Druhý algoritmus odstraňuje zbytečné stavy. Za zbytečné stavy jsou považovány stavy nedostupné nebo neukončující, jelikož jejich přítomnost nemá vliv na jazyk KA. Tento algoritmus nevytváří nový KA, ale upravuje KA obdržený na vstupu. Algoritmus byl navržen autorem práce.

Tento algoritmus je rozdělen do tří částí, dvě z nich značí stavy pro odstranění a ve třetí části probíhá jejich mazání. V první části (řádky 1–10) algoritmus prochází všechny stavy dostupné z počátečních stavů a každý navštívený stav poznačí uložením do tabulky Visited1. Nepoznačené stavy jsou stavy nedostupné. Každý stav je opět zpracován nejvýše jednou, nehrozí tedy zacyklení. Druhá část je obdobná s tím rozdílem, že je použit reverzní KA M^r a stavy jsou ukládány do tabulky Visited2. V tomto průchodu nepoznačené stavy představují stavy neukončující. V případě obousměrně propojeného modelu KA, který byl použit v této práci, není třeba vytvářet reverzní automat, což může být náročná operace, ale stačí místo počátečních stavů použít koncové stavy a místo normálních přechodů reverzní přechody. Na závěr, v poslední části algoritmu, jsou pomocí funkce Restore_FA() odstraněny všechny stavy, které se nenachází v obou tabulkách Visited1 a Visited2. V tomto algoritmu je použit princip prohledávání do šířky, ale stejně tak lze bez vlivu na efektivitu použít i prohledávání do hloubky, jelikož se stejně musí projít všechny dostupné a ukončující stavy.

Funkce Restore_FA() má dva úkoly, odstranit všechny přechody do a z mazaného stavu a následně odebrat samotný stav z množiny Q. Při velkém množství odebíraných stavů je tato část algoritmu nejnáročnější. Díky obousměrné provázanosti modelu KA je ze stavu $q \in Q$ třeba odebrat reverzní přechod $r' = qa \to p$ pro každý přechod $r = pa \to q$ z mazaného stavu p, zároveň je třeba z q odebrat přechod $r = qa \to p$ pro každý reverzní přechod $r' = pa \to q$ z téhož stavu p, $a \in \Sigma$. Jelikož má každý stav své přechody uloženy zvlášť ve své struktuře, není třeba procházet všechny stavy a přechody automatu, ale jen stavy s mazaným stavem propojené, což by mělo při nižším procentu odstraňovaných stavů zvýšit efektivitu zpracování KA s velkým počtem stavů. Samotné odstranění stavu z množiny Q je závislé na jazyku. V C++ je množina stavů Q uložena v lineárně zřetězeném seznamu, je tedy možné stav přímo smazat bez narušení validity ukazatelů a vytváření děr. V jazyce Python 3 je množina Q uložena v seznamu (List), pro zvýšení efektivity je tedy před

mazáním seznam setřepán metodou, kdy je v Q zleva hledán stav pro smazání a zprava je hledán první platný stav. Je-li nalezena dvojice, proběhne výměna. Díky funkcionalitě proměnných v Pythonu není touto výměnou narušena validita odkazů. Výsledkem je množina se stavy ke smazání naskládanými na konci, lze je tedy jednoduše odstranit. Tento postup je použit i v OCamlu a C#. Visited1 a Visited2 jsou implementovány stejně jako v testu prázdnosti.

3.1.3 Algoritmus 3: Průnik (Produkt)

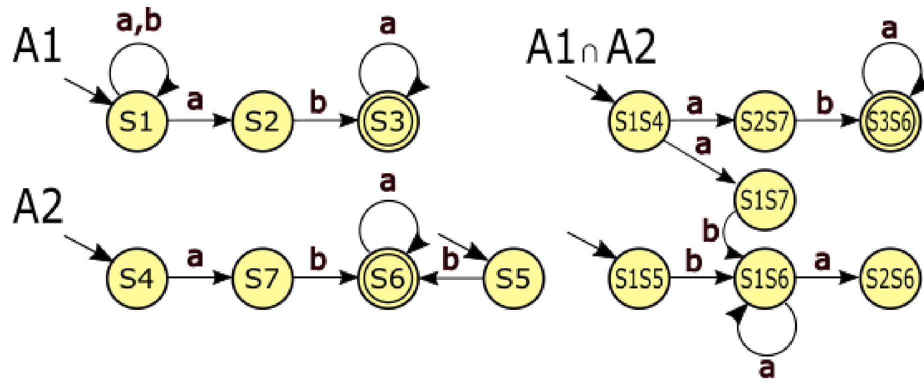
Vstup: KA $A_1 = \{Q_1, \Sigma, \delta_1, S_1, F_1\}$, $A_2 = \{Q_2, \Sigma, \delta_2, S_2, F_2\}$
Výstup: KA $A_1 \cap A_2 = \{Q, \Sigma, \delta, S, F\}$, $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

1. Q, δ, $F \leftarrow \varnothing$; $S \leftarrow S_1 \times S_2$
2. $W \leftarrow S$
3. **for** $[q_1, q_2] \in S$ **do** // vloženo
4. vlož $[q_1, q_2]$ do Q // vloženo
5. **if** $q_1 \in F_1$ **and** $q_2 \in F_2$ **then** vlož $[q_1, q_2]$ do F // vloženo
6. **while** $W \neq \varnothing$ **do**
7. vyjmi $[q_1, q_2]$ z W // po 7. ř. 2 řádky odebrány
8. **for all** $a \in \Sigma$ **do**
9. **for all** $q_1' \in \delta_1(q_1, a)$, $q_2' \in \delta_2(q_2, a)$ **do**
10. **if** $[q_1', q_2'] \notin Q$ **then**
11. vlož $[q_1', q_2']$ do Q // vloženo
12. **if** $q_1' \in F_1$ **and** $q_2' \in F_2$ **then** vlož $[q_1', q_2']$ do F // vloženo
13. vlož $[q_1', q_2']$ do W
14. vlož ($[q_1, q_2]$,a,$[q_1', q_2']$) do δ

Výše uvedený algoritmus a teorie pro tuto kapitolu byly převzaty z [6], strana 84 – 86.

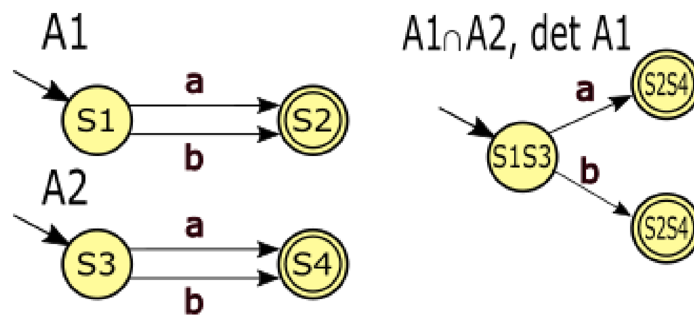
Třetí algoritmus vytváří průnik dvou KA, tedy KA, pro jehož jazyk platí $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$. Automaty A_1 a A_2 musí mít shodné abecedy. Principem algoritmu je vyrábění dvojic stavů, první z dvojice náleží prvnímu automatu, druhý z dvojice náleží druhému automatu. Výsledkem algoritmu je nový KA, vstupní automaty zůstávají zachované. Na začátku se fronta dvojic stavů určených ke zpracování (W) naplní kartézským součinem počátečních stavů obou automatů, následně jsou dvojice z W postupně vybírány. Pro každé písmeno abecedy jsou poté nalezeny přechody z obou stavů dvojice do stavů následnických. Nad oběma množinami následnických stavů je potom opět proveden kartézský součin. Vzniklé dvojice se přidávají do Q jen tehdy, pokud tam ještě nejsou, v takovém případě se zároveň dvojice vloží i do W. Dvojice stavů $[q_1, q_2]$ reprezentuje nový stav ve výsledném automatu. Platí, že nový stav je koncový právě tehdy, když oba stavy dané dvojice jsou koncové v rámci KA, do kterých náleží [6]. Činnost tohoto algoritmu je znázorněna na obrázku 3.1.

Oproti původnímu algoritmu proběhla jedna úprava, a to odstranění dvou řádků mezi 7–8 a jejich vložení na řádky 4–5 a 11–12 (vyznačeno poznámkami). V původním algoritmu se dvojice vkládala do Q na jediném místě ihned po vytažení z W. Test na řádku 10 ovšem hledá dvojici jen v Q a ne ve W, mohlo by se tedy stát, že jeden stav bude do W vložen vícekrát, protože ještě neproběhlo jeho vytažení z W a vložení do Q (obrázek 3.2). V původním algoritmu vložení dvojice do Q nebylo ničím podmíněno. Problém je vyřešen přesunem vložení dvojice do Q na stejné místo, na jakém se vkládá do W (řádky 11 a 12). Tento přesun, kromě opravy chybného chování, kterým vícenásobné vložení stejného stavu do množiny Q



Obrázek 3.1: **Ukázka činnosti algoritmu průniku.** Nejdříve jsou vytvořeny dva počáteční stavy S1S2 a S1S5, z nich jsou následně vytvářeny další dvojice kartézským součinem následníků obou stavů. $A_1 \cap A_2$ pěkně ilustruje postupný vývoj automatu od počátečních stavů.

zjevně je, nijak nemění výsledný KA, jelikož dle původního algoritmu musela být každá dvojice ve W dříve či později vložena do Q. Změna pouze zajišťuje vložení okamžité, po jejím provedení již v kombinaci s podmínkou na řádku 10 nemůže dojít k redundantnímu vložení stavu do Q. Řádky 3–5 zajišťují vložení všech dvojic počátečních stavů do Q, které by jinak po úpravě neproběhlo.



Obrázek 3.2: **Příklad chyby v původním algoritmu.** Výsledný algoritmus napravo má zbytečně mnoho stavů, protože při zpracovávání stavu S1S3 byl stav S2S4 vložen do W dvakrát. Přeskládáním řádků byla chyba opravena.

Na řádku 14 se nachází vložení přechodu ze zdrojového stavu do cílového stavu. Jelikož jsou přechody uchovávány uvnitř každého stavu, je nutné znát zdrojový stav $[q_1, q_2] \in Q$, jako i zdrojovou dvojici stavů $q_1 \in Q_1$ a $q_2 \in Q_2$ pro nalezení nových stavů $[q'_1, q'_2]$. Proto je v implementaci tohoto algoritmu do fronty W ukládána trojice $[q_1, q_2, [q_1, q_2]]$. Dále je pro urychlení testu $q \in Q$ použita hashovací tabulka, do které jsou ukládány názvy stavů v Q.

3.1.4 Algoritmus 4: Determinizace KA

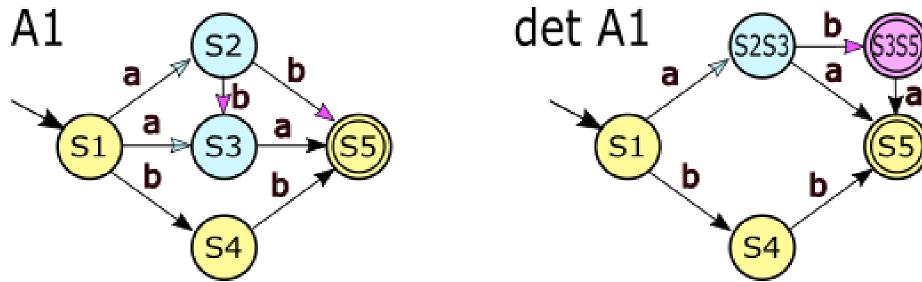
Vstup: KA $A = \{Q_A, \Sigma, \delta_A, S, F_A\}$
Výstup: DKA $B = \{Q_B, \Sigma, \delta_B, S, F_B\}$, $L(B) = L(A)$

1. Q_B, δ_B, $F_B \leftarrow \varnothing$; $s \leftarrow S$
2. $W \leftarrow \{S\}$
3. vlož s do Q_B // vloženo
4. **if** $S \cap F_A \neq \varnothing$ **then** vlož s do F // vloženo
5. **while** $W \neq \varnothing$ **do**
6. vyjmi Q' z W // po 6. ř. 2 řádky odebrány
7. **for all** $a \in \Sigma$ **do**
8. $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, a)$
9. **if** $Q'' \notin Q_B$ **then**
10. vlož Q'' do Q_B // vloženo
11. **if** $Q'' \cap F_A \neq \varnothing$ **then** vlož Q'' do F // vloženo
12. vlož Q'' do W
13. vlož (Q', a, Q'') do δ_B

Výše uvedený algoritmus a teorie pro tuto kapitolu byly převzaty z [6], strana 27 – 30.

Algoritmus pro determinizaci KA očekává na vstupu nedeterministický KA A, ze kterého poté vytváří DKA B. Automat A zůstává beze změny. Je-li KA deterministický, pak musí pro každý stav automatu a jakýkoli symbol abecedy existovat nejvýše jeden přechod. Jinými slovy vždy pro dané slovo, zpracovávané automatem, existuje nejvýše jedna cesta (průchod) a v každé situaci musí být jasné, kudy se půjde dále. Nedeterministický KA takovouto vlastnost nemá a připouští nejednoznačnost. Tato vlastnost je velmi nepříjemná, neboť narušuje bezkontextovost KA – v nejhorším případě automat při zpracovávání slova nebude až do přečtení celého slova vědět, kterou cestou se dát. Naštěstí existuje způsob převodu KA na DKA. Princip spočívá ve sloučení všech různých průchodů automatem pro stejné slovo do jednoho. Kdykoli se vyskytne více možností, kudy z daného stavu pokračovat, algoritmus si prostě vybere všechny a zapamatuje si stavy, do kterých se dostal, jejich sloučením a vytvořením nového stavu. Následně hledá přechody ze všech stavů, ze kterých se tento stav skládá a tímto způsobem tvoří další stavy DKA. Pro výše uvedený algoritmus z toho plyne jednoduché pravidlo. Automat je procházen z počátečního stavu, vzniknuvšího sloučením všech počátečních stavů KA a umístěného do fronty W. Vždy, když je stav Q' vytažen z W, jsou pro každé $a \in \Sigma$ zvlášť nalezeny všechny přechody $\delta(p, a)$, $p \in Q'$. Z nově získaných stavů je vytvořen stav Q'', který je vložen do W k dalšímu zpracování. Na obrázku 3.3 je ukázána činnost tohoto algoritmu.

Jelikož stav Q' obsahuje všechny stavy KA, do kterých se dá dostat přečtením stejné posloupnosti znaků, musí platit, že pokud alespoň jeden stav $p \in Q'$ je koncový, pak je koncový i celý stav Q'. Stejně jako v algoritmu 3 byly provedeny změny oproti původní verzi algoritmu. Byly odebrány dva řádky za řádkem 6, které byly následně vloženy na místa 3–4 a 10–11 (vyznačeno poznámkami). Důvod byl opět stejný, řádek 9 ověřuje přítomnost Q'' pouze v množině Q_B, ale vkládal Q'' do W. Každý stav z W byl sice vložen do Q_B, ale až po jeho vyjmutí a zpracování. Hrozilo tedy, že bude stejný stav vložen do W a následně i do Q_B vícekrát (například v případě, že by se z jednoho stavu dalo přejít dvěma různými znaky do stejných nových stavů), což nejspíše není zamýšlené chování, protože mimo jiné bezdůvodně zvyšuje počet stavů. Ukázka chyby v původní verzi algoritmu je na obrázku 3.2.



Obrázek 3.3: **Ukázka činnosti algoritmu determinizace.** Barevně jsou vyznačeny slučované stavy a přechody. V automatu A1 je vidět nedeterminizmus ve stavu S1, kdy z něj vedou dva přechody pro znak a. Proto jsou přechody sloučeny v jeden a je vytvořen nový stav S2S3 zastupující obě cesty.

Úprava algoritmu způsobila nutnost v implementaci do fronty W ukládat dvojici [množina stavů $p \in Q'$, Q'], aby bylo možno propojit již dříve vytvořený stav Q' s nově vytvořeným stavem Q''. Optimalizováno bylo také vyhledávání v množině stavů Q_B, a to vytvořením hashovací tabulky názvů stavů, které jsou v Q_B, což výrazně urychlí test $Q'' \notin Q_B$. Pro rychlejší kontrolu této podmínky a zamezení výskytu stejných stavů s různým pořadím podstavů např. S1S2S3, S2S3S1, S3S2S1 atd. bylo jako součást řádku 8 přidáno seřazení stavů množiny Q'' podle jmen. Implementace dále umí přepínat mezi verzemi „DKA“ a „úplný DKA“ s přidáním stavu fail a podmínkou, že každý stav musí mít právě jeden přechod pro každý znak abecedy.

3.1.5 Algoritmus 5: Minimalizace DKA

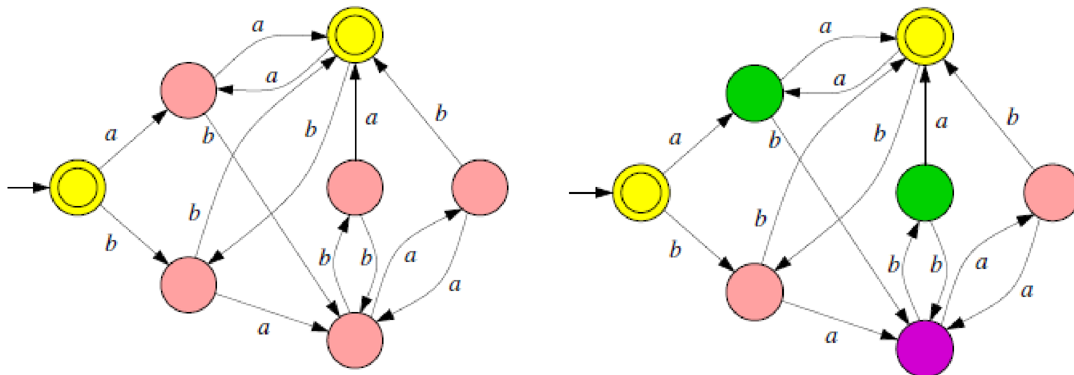
Hopcroftův algoritmus:
Vstup: DKA $A = \{Q, \Sigma, \delta, s, F\}$
Výstup: Jazykový rozklad P.

1. **if** $F = \varnothing$ **or** $Q \setminus F = \varnothing$ **then return** {Q}
2. **else** $P \leftarrow \{F, Q \setminus F\}$
3. W $\leftarrow \{(a, min\{F, Q \setminus F\}) | a \in \Sigma\}$
4. **while** $W \neq \varnothing$ **do**
5.   vyjmi (a, B') z W
6.   **for all** $B \in P$ rozdělený splitterem (a, B') **do**
7.   nahraď B pomocí B_0 a B_1 v P
8.   **for all** $b \in \Sigma$ **do**
9.   **if** $(b, B) \in W$ **then** nahraď (b, B) pomocí (b, B_0) a (b, B_1) ve W
10.   **else** vlož $(b, min\{B_0, B_1\})$ do W
11. **return** P

Tato verze Hopcroftova algoritmu a informace v této kapitole byly převzaty z [6], strana 51-62.

Vstupem algoritmu pro minimalizaci je DKA, výstupem je jeho minimální forma. Původní DKA zůstává zachován. Minimalizace se skládá ze dvou částí, a to rozdělení množiny Q na podmnožiny podle určitých pravidel (což provádí výše uvedený Hopcroftův algoritmus) a následného vytvoření stavů minimálního DKA z podmnožin Q. Každá podmnožina odpovídá jednomu stavu. Princip činnosti tohoto algoritmu je znázorněn na obrázku 3.4.

Smyslem minimalizace je mnohdy výrazné snížení počtu stavů a přechodů DKA, a tedy i paměťové a výpočetní náročnosti. Další využití má minimalizace při zjišťování, zda se dva regulární jazyky rovnají, kdy se vytvoří jejich minimální DKA a ověří se, jestli jsou izomorfní [6]. *KA je minimální, pokud neexistuje žádný jiný KA, který rozeznává stejný jazyk a má méně stavů (s výjimkou izomorfních KA). Každý regulární jazyk má unikátní minimální DKA* (obě předchozí tvrzení převzaty z [6]).



Obrázek 3.4: **Ilustrace činnosti Hopcroftova algoritmu.** Barevně jsou vyznačeny stavy, které náleží do stejného bloku. Vlevo je počáteční stav, vpravo je výsledný rozklad. (Převzato z [6], strana 57)

Pro pochopení činnosti algoritmu je nutné si nadefinovat w-reziduál jazyka L, $L \in \Sigma^*$, který se značí $L^W = \{u \in \Sigma^* | wu \in L\}$. Jinými slovy L^W je množina sufixů řetězců jazyka L s prefixem w. S tím úzce souvisí dříve definovaný pojem jazyk stavu automatu $L_A(q)$ pro $A = \{Q, \Sigma, \delta, s, F\}$ a $q \in Q$. Platí totiž [6], že:

1. Pro každé $w \in \Sigma^*$ existuje $q \in Q$ takové, že $L_A(q) = L^W$.
2. Pro každý stav $q \in Q$ existuje $w \in \Sigma^*$ takové, že $L_A(q) = L^W$.

Konečná množina $P = \{B_1, \ldots, B_n\}$, kde $B_1, \ldots, B_n$ jsou vzájemně disjunktní bloky takové, že jejich sjednocením dostaneme množinu stavů Q, se nazývá *rozklad množiny Q*. Rozklad P je *hrubší* než jiný rozklad P', pokud pro každý blok z P' platí, že je obsažen v nějakém bloku v P. Principem minimalizace je tedy rozdělení množiny stavů Q na vzájemně disjunktní bloky tak, že jazykem všech stavů jednoho bloku je stejný reziduál, a tedy i jazyky všech těchto stavů jsou stejné. Takové rozdělení se nazývá *jazykový rozklad*. Následně dojde k vytvoření stavů minimálního DKA sloučením stavů každého bloku zvlášť.

Zároveň platí, že stavy v různých blocích musí rozeznávat různé jazyky. Počáteční rozdělení Q v Hopcroftově algoritmu je tedy $P_0 = \{F, Q \setminus F\}$ pokud F a $Q \setminus F$ jsou neprázdné, $P_0 = \{F\}$ pokud $Q \setminus F$ je prázdná nebo $P_0 = \{Q\}$ pokud F je prázdná. Je to z toho důvodu, že ukončující stavy vždy přijímají prázdné slovo, ale neukončující stavy ne. Rozeznávají tedy různé jazyky. Toto počáteční rozdělení se poté v průběhu algoritmu mění dělením jednotlivých bloků na dvě části. Platí $L(\delta(p, a)) \neq L(\delta(q, a)) \Rightarrow L(p) \neq L(q)$ pro $p, q \in Q$ a každé $a \in \Sigma$ zvlášť. Blok je tedy možno rozdělit v případě, že stavy p a q do něj oba patří, ale stavy $\delta(p, a)$ a $\delta(q, a)$ pro dané $a \in \Sigma$ do jednoho bloku nepatří. Takovým rozdělením totiž nebude narušena podmínka ze začátku tohoto odstavce.

Nechť P je jazykový rozklad, $B, B' \in P$, $a \in \Sigma$, pak pár (a, B') dělí blok B, pokud existují $p, q \in B$ takové, že $\delta(p, a) \in B'$ a $\delta(q, a) \notin B'$. Výsledný rozklad $P = \{(P \setminus \{B\}) \cup$

$\{B_0, B_1\}\}$, kde $B_0 = \{q \in B | \delta(q, a) \notin B'\}$ a $B_1 = \{q \in B | \delta(q, a) \in B'\}$. Jazykový rozklad P je *nestabilní*, pokud obsahuje bloky, které mohou být děleny nějakým párem (a, B'), jinak je *stabilní*. Hopcroftův algoritmus hledá nejhrubší stabilní rozklad množiny Q.

Nechť $V = \{Q_v, \Sigma, \delta_v, s_v, F_v\}$ je výsledný minimální DKA. Jak již bylo řečeno, bloky v jazykovém rozkladu P představují stavy v Q_v. Stav bude počáteční, pokud korespondující blok obsahuje alespoň jeden počáteční stav. Stejně tak bude stav koncový, pokud odpovídající blok obsahuje alespoň jeden koncový stav. Mezi bloky B a B' bude přechod $(B, a, B') \in \delta_v$, $a \in \Sigma$, pokud $(p, a, q) \in \delta$ pro nějaký $p \in B$, $q \in B'$. Jinými slovy, pokud existuje přechod mezi stavy dvou rozdílných bloků, bude existovat stejný přechod i mezi bloky samotnými.

Jelikož všechny bloky obsahují stavy, které rozeznávají stejné jazyky, a tedy mají pro všechny znaky přechody do stejných bloků, stačí prozkoumat jen první stav každého bloku a není třeba iterovat přes všechny. Pokud stav nemá pro nějaké písmeno přechod, nemůže být ve stejném bloku jako stavy, které přechod pro dané písmeno mají. Pokud tedy prvnímu stavu přechod chybí, musí chybět i všem ostatním stavům stejného bloku.

Posledním tématem je výběr párů (a, B') nazývaných *splittery*. Hopcroftův algoritmus optimalizuje výběr splitterů následovně. Na začátku algoritmu obsahuje fronta W všechny splittery (a, B') pro $a \in \Sigma$ a $B' = min(F, Q \setminus F)$. Následně algoritmus po jednom vybírá splittery z W a snaží se jimi postupně rozdělit každý blok aktuálního jazykového rozkladu P. Pokud je blok B rozdělen splitterem (a, B'), pak je nutná následující úprava W. Pro každé $b \in \Sigma$, pokud splitter (b, B) je ve W, bude smazán a nahrazen dvěma splittery (b, B_0) a (b, B_1). Pokud splitter (b, B) není ve W, je třeba do W přidat pouze jeden splitter $(b, min(B_0, B_1))$.

Z důvodu nutnosti odstraňovat a přidávat prvky množiny splitterů W a množiny jazykového rozkladu P z/na jakoukoli pozici jsou tyto množiny implementovány pomocí lineárně zřetězeného seznamu. Pro C++ je to třída List, v C# se jedná o LinkedList a v OCamlu je to modul List. Python bohužel takovou datovou strukturu nenabízí, a tak musela být použita méně efektivní třída List, která je v Pythonu implementována jako klasické dynamické pole. Pro reprezentaci bloků byl v C++ použit Vektor, v Pythonu a C# třída List a v OCamlu stejnojmenný modul. Řádek 6 z algoritmu byl implementován následovně. Bylo nutné projít všechny bloky v P, pro každý projít přes všechny jeho stavy a zjistit, jestli přechod z daného stavu vede do bloku ve splitteru. Pokud ano, byl stav uložen do pomocného bloku B_0, pokud ne, byl uložen do pomocného bloku B_1. Ve chvíli, kdy byly takto prozkoumány všechny stavy bloku, algoritmus porovnal velikosti obou nových bloků. Pokud byla velikost jednoho z nich nulová, znamenalo to, že se původní blok dělit nemá. Pokud byly obě velikosti nenulové, byl původní blok rozdělen vyjmutím z P a následným vložením obou pomocných bloků B_0 a B_1. Zbytek implementace se nelišil od postupu uvedeném v algoritmu.

Na začátku kapitoly je znázorněna pouze první část, druhá část, tedy sestavování DKA z jazykového rozkladu, již není součástí převzatého algoritmu a je tedy výtvorem autora tohoto textu.

Vstup: Jazykový rozklad P.
Výstup: minimální DKA $A = \{Q, \Sigma, \delta, s, F\}$

1. Q, δ, F, $W \leftarrow \varnothing$ // Σ je stejná, jako v původním DKA
2. **for all** $B \in P$ **do**
3. spoj stavy v B a vlož jej do Q
4. **for all** $\delta(p, a)$, $a \in \Sigma$ **do** // p je nějaký jeden stav z B
5. vlož $(\delta(p, a), a, B)$ do W
6. **if** nějaké $q \in B$ je *final* **then** vlož B do F
7. **if** nějaké $q \in B$ je *starting* **then** $s \leftarrow B$
8. **for all** $B \in P$ **do**
9. **for all** $p \in B$ **do**
10. **for all** (p, a, B') ve W **do**
11. vlož (B', a, p) do δ

Algoritmus vytvoří stav pro každý blok rozkladu, následně vybere jeden stav z onoho bloku a najde všechny jeho přechody. Není třeba procházet více stavů, neboť přechody všech stavů z jednoho bloku vedou do nejvýše jednoho jiného bloku. Pro každý přechod poté vytvoří "požadavek na spojení" se stavem nějakého potenciálně ještě nevytvořeného bloku, který vloží do W. W je dobré implementovat jako hashovací tabulku pro rychlé vyhledávání podle jména stavu. Následně se zkontrolují podmínky pro koncový nebo počáteční stav. Na závěr je nutné propojit bloky. Každý stav z každého bloku zkontroluje, jestli se ve W nevyskytuje jeho jméno, pokud ano, dojde k propojení bloku uloženého ve W s blokem aktuálního stavu.

V C++ bylo W implementováno jako třída Multimap, v Pythonu byl použit Slovník, v C# třída Dictionary a v OCamlu modul Hashtbl.

3.1.6 Algoritmus 6: Výpočet relace simulačního předuspořádání

Vstup: KA $A = \{Q, \Sigma, \delta, S, F\}$
Výstup: relace simulace R

1. **for** $q \in Q$, $a \in \Sigma$ **do** // 1-4 příprava
2. // $\delta^r(q, a)$ již vypočítáno, vysvětlení v kapitole 2.2
3. vypočítej $card(\delta(q, a))$
4. inicializuj $N(a)$ pomocí nul
5. $R \leftarrow \varnothing$; $W \leftarrow NEWQUEUE()$ // 5-18 výpočet
6. $R \leftarrow (F \times (Q \setminus F)) \cup \{(q, r) | \exists a \in \Sigma : \delta(q, a) \neq \varnothing \wedge \delta(r, a) = \varnothing\}$ // úprava
7. $W \leftarrow R$
8. **while** $W \neq \varnothing$ **do**
9. vyjmi (i, j) z W
10. **for** $a \in \Sigma$ **do**
11. **for** $k \in \delta^r(j, a)$ **do**
12. $N(a)_{ik} \leftarrow N(a)_{ik} + 1$
13. **if** $N(a)_{ik} = card(\delta(k, a))$ **then**
14. **for** $j \in \delta^r(i, a)$ **do**
15. **if** $(j, k) \notin R$ **then**
16. $R \leftarrow R \cup \{(j, k)\}$
17. vlož (j, k) do W
18. **return** $(Q \times Q) \setminus R$ // úprava

Výše uvedený algoritmus a informace v této kapitole byly převzaty z [9]. Úprava algoritmu na řádku 6 je převzatá z [8], strana 7.

Algoritmus pro výpočet relace simulace (definováno níže), který je prvním z pokročilých algoritmů v této práci, očekává na vstupu KA a vrací relaci simulace nad jeho stavy. Relace simulace má mnoho využití, například při zpracovávání regulárních výrazů a s tím související redukce KA nebo při testu univerzality a inkluze nedeterministických KA (viz algoritmy 3.1.7 a 3.1.8). V následujícím textu se často používá spojení "slovo prochází přes stav". Pokud slovo w ve tvaru $a_0a_1, \ldots, a_{n-1}$ prochází přes stav p, pak to znamená, že existuje posloupnost přechodů $x_0a_0 \rightarrow x_1, \ldots, x_{n-1}a_{n-1} \rightarrow x_n$ taková, že x_0 je počáteční stav, x_n je koncový stav a $p = x_i$ pro $0 \leq i \leq n$.

Relace simulace R je reflexivní a tranzitivní binární relace. Nechť p, q, r jsou stavy v Q, pak pro R platí:

1. $(p,p) \in R$
2. $(p,q) \in R \wedge (q,r) \in R \Rightarrow (p,r) \in R$

Maximální relaci simulace se říká *simulační předuspořádání*. V tomto textu se nadále pod pojmy *relace simulace* nebo jen *simulace* myslí relace simulačního předuspořádání. Pod pojmem *reverzní simulace* je myšlena relace simulace reverzního automatu.

R je největší relace simulace na množině Q, která splňuje:

1. $R \cap (F \times (Q \setminus F)) = \varnothing$
2. pro všechna $p, q \in Q$, $a \in \Sigma$, $(pRq \Rightarrow \forall p' \in \delta(p,a), \exists q' \in \delta(q,a), p'Rq')$

Jinými slovy, ukončující a neukončující stavy nikdy nesmí být v páru a do relace přidáváme jen ty dvojice stavů, u kterých všechny přechody z pokrývaného stavu mají do dvojice přechod z pokrývajícího stavu takový, že oba následnické stavy jsou také v relaci.

Platí, že když $(p,q) \in R$, pak $L(p) \subseteq L(q)$. Dvojice (p,q) tedy vlastně říká, že stav p je (ve směru od počátečních do koncových stavů) plně pokryt stavem q. Pokud je zároveň dvojice (q,p) také v R, znamená to, že $L(p) = L(q)$, přechody z obou stavů jsou "stejné", a tudíž je možné stavy p a q beze změny jazyka KA spojit do jednoho. Na takovém principu pracují algoritmy redukující KA pomocí relace ekvivalence. Pánové Champarnaud a Coulon [4] ovšem zjistili, že lepší redukce je možné dosáhnout použitím relace simulace. Pokud totiž stav q pokrývá p v obou směrech (tedy dvojice (p,q) se nachází v simulaci i reverzní simulaci), lze tyto stavy také spojit, neboť v takovém případě existují pro určitá slova dva ekvivalentní průchody automatem, jeden přes p a druhý přes q. Pro všechna slova konkrétně platí, že když prochází přes stav p, pak prochází také přes stav q.

Pro výpočet relace simulace se používá negace výše zmíněných podmínek. R je tedy nejmenší relace předuspořádání na množině Q, která splňuje:

1. $(F \times (Q \setminus F)) \subseteq R$
2. pro všechna $p, q \in Q$, $a \in \Sigma$, $(\exists p' \in \delta(p,a), \forall q' \in \delta(q,a), p'Rq' \Rightarrow pRq)$

Přidáváme tedy do relace R dvojice (p,q) tehdy, když existuje $p' \in \delta(p,a)$, pro které počet $q' \in \delta(q,a)$ takových, že $p'Rq'$, je $card(\delta(q,a))$, tedy všechny. Card je zkratka pro kardinalitu množiny. K uchování čítačů q přechodů slouží matice $N(a)_{pq} = card(\{l \in \delta(q,a) | pRl\})$. Všechny čítače jsou na počátku vynulované.

Původní verze algoritmu vracela doplněk relace simulace, proto byla autorem tohoto textu doplněna úprava na řádku 18 a nová verze již vrací požadovanou relaci simulace. Dále byl oproti původnímu algoritmu změněn řádek 6. Algoritmus by bez této změny byl

nefunkční, protože by přidával jen dvojice stavů, které mají přechody se stejnými znaky abecedy. Pokud by dva stavy měly přechody s rozdílnými znaky, nebyly by přidány a po provedení $(Q \times Q) \setminus R$ na posledním řádku by výsledná relace dané stavy obsahovala. Jelikož relace simulace obsahuje dvojice stavů, které mohou být spojeny, nastal by problém, protože stavy s přechody s rozdílnými znaky v daném směru spojeny být v žádném případě nemohou. Byla tedy přidána část $\cup\{(q,r)|\exists a \in \Sigma : \delta(q,a) \neq \varnothing \wedge \delta(r,a) = \varnothing\}$, převzatá z [8], která do inicializace relace předuspořádání na začátku algoritmu přidá všechny dvojice stavů s rozdílnými znaky přechodů pro nějaký vstupní znak.

Pro reprezentaci množiny simulačního předuspořádání byla v C++ použita třída Unordered_set, což je hashovaná množina klíčů. Klíčem byla dvojice (název stavu1, název stavu2). Tato množina bývá obrovská a musí být perfektně optimalizovaná pro vyhledávání, které má konstantní průměrnou časovou složitost, stejně jako vkládání. V Pythonu se jednalo o Slovník, kde klíčem byla stejná dvojice a hodnota byla nastavena na None. Stejně to bylo řešeno v C# (třída Dictionary) a OCamlu (modul Hashtbl). V algoritmu uvedená struktura N byla ve všech jazycích implementována jako statické trojrozměrné pole o velikostech N[velikost abecedy][počet stavů][počet stavů]. Řádek 6 byl implementován ve třech vnořených cyklech. První dva vytvářely kartézský součin $Q \times Q$ a každá dvojice (p,q) tohoto součinu, jejíž p byl koncový stav a q nebyl koncový stav, byla vložena do R. Pro každou dvojici, která toto nesplnila, byl spuštěn cyklus, procházející písmena abecedy. Pokud se počet přechodů z p nerovnal nule a počet přechodů z q rovnal nule pro dané písmeno abecedy, byla tato dvojice taktéž vložena do R.

3.1.7 Algoritmus 7: Test univerzality

Vstup: KA $A = \{Q, \Sigma, \delta, S, F\}$ a relace simulace SIM.
Výstup: *True* pokud A je univerzální, jinak *false*.

1. **if** S nepřijímá **then return** false
2. Processed ← ∅
3. Next ← {Minimize(S)}
4. **while** Next ≠ ∅ **do**
5.   vyjmi macro-state R z Next a přesuň jej do Processed
6.   **foreach** $P \in \{Minimize(R')|R' \in \delta(R)\}$ **do**
7.   **if** P nepřijímá **then return** false
8.   **else if** neexistuje I ∈ Processed ∪ Next takový, že I $\leq^{\forall\exists}$ P **then**
9.   odstraň všechny I z Processed ∪ Next takové, že P $\leq^{\forall\exists}$ I
10.   vlož P do Next
11. **return** true

Výše uvedený algoritmus a informace v této kapitole jsou převzaty z [2].

Algoritmus pro test univerzality je druhý z pokročilých algoritmů. Zkoumá, jestli platí $L(A) = \Sigma^*$, tedy jestli jazyk automatu A obsahuje všechny řetězce složitelné ze znaků v Σ. V takovém případě vrací true.

Pro pochopení zápisu algoritmu je třeba vysvětlit některé části. Makro-stav je podmnožina stavů v Q. Říkáme, že makro-stav je koncový (přijímá), jestliže obsahuje alespoň jeden koncový stav. V opačném případě říkáme, že není koncový (nepřijímá). Jazyk makro-stavu P je definován jako $L(A)(P) = \bigcup_{p \in P} L(A)(p)$. Makro-stav P je univerzální, pokud platí $L(A)(P) = \Sigma^*$. Pro makro-stavy P, R znamená zkratka $P \leq^{\forall\exists} R$ následující: $\forall p \in P :$

$\exists r \in R : pSIMr$. Jestliže tato podmínka platí, pak $L(A)(P) \subseteq L(A)(R)$ [2]. Pro makro-stav P je definována množina $\delta(P) = \{P' | a \in \Sigma : P' = \bigcup_{\forall p \in P} \delta(p, a)\}$. Množina $\delta(P)$ tedy obsahuje množiny následnických stavů ze stavů P, každá množina odpovídá přechodu pomocí jiného znaku. Funkce *Minimize(S)* implementuje druhou optimalizaci, na řádcích 8–10 je první optimalizace. Obě optimalizace jsou popsány níže.

Výše uvedený algoritmus spojuje dva přístupy: metodu simulace pomocí relace simulace a metodu konstrukce podmnožin (subset construction) pomocí protiřetězců (antichain). Pokud je místo relace simulace použita relace identity, algoritmus funguje pouze na principu protiřetězců. Klasický algoritmus pro test univerzality fungoval tak, že nejprve determinizoval automat A pomocí konstrukce podmnožin a následně se přesvědčil, že každý makro-stav je koncový. Algoritmus lze ukončit okamžitě, když je nalezen nekoncový makro-stav. Nevýhoda tohoto přístupu spočívá ve velmi rychlém nárůstu počtu stavů. Na řadu tedy přišla optimalizace pomocí protiřetězců, která je popsána v [5]. Ta spočívala v tom, že po zkonstruování nového makro-stavu bylo vždy před jeho uložením do množiny next zkontrolováno, zdali se již v next $\cup$ processed nenachází makro-stav, jehož množina stavů by byla podmnožinou množiny stavů nového makro stavu. Pokud byl takový makro-stav nalezen, nový makro-stav se neuložil a algoritmus pokračoval. Například pokud již byl zpracován makro-stav M = [p, q, r] (uložený v množině processed), nově vytvořený makro-stav N = [p, q, r, s] nebude uložen do množiny next, protože je jasné, že $L(M) \subseteq L(N)$ a tudíž pokud je M koncový, musí být koncový i N. Stačí tedy dále zpracovávat pouze M. Dále pokud je v next $\cup$ processed nalezen makro-stav s množinou stavů, která je nadmnožinou množiny stavů nového makro-stavu, je tento stav z next $\cup$ processed odstraněn a do next je místo něj uložen nový makro-stav. Důvod je totožný.

Tato optimalizace se však ukázala být stále nedostatečná, z toho důvodu autoři výše uvedeného algoritmu ([2]) navrhli optimalizace odstraňující další zbytečné makro-stavy, které pouhé vylepšení protiřetězci eliminovat nedokázalo. Tyto optimalizace jsou založeny na relaci simulace, která je popsána výše v kapitole 3.1.6, a jsou principově velmi podobné těm původním. První z nich je opět založena na myšlence, že prohledávání z makro-stavu M nemusí pokračovat, pokud již byl nalezen makro-stav N, který je "menší". Jestliže totiž slovo není přijato makro-stavem M, pak není přijato ani makro-stavem N, jehož jazyk je podmnožinou jazyka L(A)(M). Použitím dříve definovaného operátoru N $\leq^{\forall\exists}$ M zjistíme, že pokud pro každý stav p z N platí, že existuje stav q z M takový, že $(p, q) \in SIM$, prohledávání ze stavu M již nemusí pokračovat, neboť $L(A)(P) \subseteq L(A)(R)$. Druhá optimalizace předpokládá, že každý stav je reprezentován pouze podmnožinou svých maximálních stavů. Pokud se tedy v makro-stavu P nacházejí dva stavy p a q, přičemž $(p, q) \in SIM$, je možné stav p z P odstranit, neboť $L(p) \subseteq L(q)$, a je tedy evidentní, že $L(A)(P) = L(A)(P \setminus \{p\})$ [2]. Pokud P přijímá slovo se stavem p, bude ho přijímat i beze stavu p. To stejné platí i pro makro-stav, který je nekoncový.

Celý algoritmus funguje na principu algoritmu 3.1.4 – determinizace KA s jediným rozdílem, který spočívá v aplikaci obou optimalizací. Algoritmus přesouvá makro-stavy z množiny Next do množiny Processed, dokud Next není prázdná. Pro každý makro-stav jsou nalezeny všechny následnické makro-stavy R', které jsou výsledkem $\delta(R)$. Pokud je některý makro-stav nekoncový, je vrácena hodnota false. Každý R' je minimalizován a vložen do Next, pokud neexistuje žádný menší makro-stav v Next nebo Processed. Pokud je makro-stav přidán do Next, algoritmus zároveň odstraní všechny větší makro-stavy z Next a Processed.

Next a Processed jsou implementované jako neseřazené množiny (např. vector, list ...), jelikož se v nich nevyhledává, řádky 8 a 9 vyžadují iteraci přes všechny prvky a kontrolu

speciální podmínky. Jednotlivé následnické makro-stavy jsou tvořeny postupně, nejprve je zpracován jeden, poté je vytvořen druhý. Aby se předešlo situaci, kdy makro-stav obsahuje vícero stejných stavů (např. [s, s, x, y, s]), což se může stát ve chvíli, kdy dva a více stavů určitého makro-stavu mají přechod stejným písmenem do stejného stavu, byla použita hashovací tabulka jmen stavů a stavy jsou vkládány do makro-stavu jen tehdy, pokud ještě nejsou v tabulce. Funkce Minimize(S) je implementovaná pomocí dvou vnořených cyklů, procházejících množinu stavů daného makro-stavu. Pokud se dvojice (i, j) nachází v relaci simulace, je stav i z makro-stavu odstraněn a vnější cyklus pokračuje v procházení od stavu i+1. Ověřování funkce is_rejecting je provedeno pomocí procházení makro-stavu v cyklu a ověřování, jestli je alespoň jeden stav koncový (což odpovídá porovnání hodnoty v proměnné). Ověřování P $\leq^{\forall\exists}$ R je opět implementováno pomocí dvou cyklů, jeden prochází P a druhý R, přičemž pro každý p z P musí být nalezen r z R takový, že (p, r) je v relaci simulace. Řádky 8 a 9 jsou prováděny zároveň jedním průchodem přes Next a Processed, jelikož se navzájem vylučují. Pokud nebude splněna podmínka na řádku 8, nemůže být odstraněn žádný makro-stav řádkem 9. V Next a Processed jsou uchovávány struktury macro_st se dvěma položkami – neseřazené množiny stavů daného makro stavu a booleovskou proměnnou rejecting, indikující koncovost makro-stavu. Množina stavů je neseřazená, protože se v ní nevyhledává, ale iteruje.

3.1.8 Algoritmus 8: Test inkluze

Vstup: KA $A = \{Q_A, \Sigma, \delta_A, S_A, F_A\}$, $B = \{Q_B, \Sigma, \delta_B, S_B, F_B\}$ a relace simulace SIM nad $A \cup B$.
Výstup: *True* pokud $L(A) \subseteq L(B)$, jinak *false*.

1. **if** existuje product-state, který přijímá, v $\{(s, S_B) | s \in S_A\}$ **then return** false
2. Processed $\leftarrow \emptyset$
3. Next $\leftarrow$ Initialize($\{(s, Minimize(S_B)) | s \in S_A\}$)
4. **while** Next $\neq \emptyset$ **do**
5.   vyjmi product-state (r, R) z Next a přesuň jej do Processed
6.   **foreach** $(p, P) \in \{(r', Minimize(R')) | (r', R') \in \delta((r, R))\}$ **do**
7.   **if** (p, P) je product-state, který přijímá **then return** false
8.   **else if** neexistuje $p' \in P$ takové, že $pSIMp'$ **then**
9.   **if** neexistuje $(i, I) \in$ Processed $\cup$ Next takový, že $pSIMi \wedge I \leq^{\forall\exists} P$ **then**
10.   odstraň všechny (i, I) z Processed $\cup$ Next takové, že $iSIMp \wedge P \leq^{\forall\exists} I$
11.   vlož (p, P) do Next
12. **return** true

Výše uvedený algoritmus a informace v této kapitole jsou převzaty z [2].

Algoritmus pro test inkluze je třetí z pokročilých. Je principiálně velmi podobný algoritmu pro test univerzality. Jedná se o ověření, že $L(A) \subseteq L(B)$. Opět platí, že je-li místo relace simulace použita relace identity, funguje algoritmus pouze na principu protiřetězců ([5]).

Definujeme doplněk automatu A jako automat $\bar{A}$, přičemž platí, že $L(\bar{A}) = \{w | w \in \Sigma^*, w \notin L(A)\}$. V praxi se takový automat vytvoří výměnou koncových a nekoncových stavů. Běžný algoritmus pro test inkluze průběžně vytváří průnik automatů $A \times \bar{B}$, kdy $\bar{B}$ značí doplněk automatu B. Stavem takového automatu je dvojice (p, P), kde p představuje stav z A a P je makro-stav z B. Takové dvojici se říká product-state. Product-state (p, P) přijímá, pokud p je koncový stav v A a P není koncový. V případě, že algoritmus nalezne

takový stav, vrací false. Myšlenka je jednoduchá, pokud totiž $L(A) \subseteq L(B)$, musí platit $L(A) \cap (\Sigma^* \setminus L(B)) = \emptyset$, a tak nesmí nastat situace, kdy p slovo přijímá a P slovo nepřijímá.

Stejně jako u algoritmu 3.1.7 existují dvě optimalizace, které jsou upraveny pro potřeby tohoto algoritmu. První optimalizace říká, že je možné ukončit procházení ze stavu (p, P), pokud a) existuje navštívený product-state (r, R) takový, že $pSIMr$ a zároveň $R \leq^{\forall\exists} P$ nebo b) $\exists p' \in P : pSIMp'$. Princip a) je úplně stejný jako u testu univerzality. Optimalizace 1 b) je odůvodněna tím, že pokud platí podmínka, znamená to, že jazyk $L(p) \subseteq L(P)$, a tedy není třeba z tohoto product-state pokračovat, protože vždy, když bude následník stavu p koncový, logicky musí být koncový i následnický makro-stav, a tak product-state nebude nikdy přijímat. Procházením následníků stavu (p, P) tedy nebude nikdy splněna podmínka ukončení algoritmu s návratem false. Optimalizace 2 lze použít také, neboť pokud $L(B)(P) = L(B)(P \setminus \{p\})$, pak evidentně $L(A, B)(p, P) = L(A, B)(p, P \setminus \{p\})$.

Pseudokód algoritmu je téměř stejný, jako zápis testu univerzality. Operace Minimalize(S_B) je implementována úplně stejně, taktéž $P \leq^{\forall\exists} I$ a operace is_rejecting pro makro-stavy. Test na is_accepting pro product-state je prováděn jednoduchým porovnáním dvou proměnných makro-stavu P a stavu p. Přidána byla jedna operace Initialize(), která aplikuje obě části optimalizace 1 na vzniklou množinu product-state. Oproti testu univerzality je také nutné definovat operaci $\delta((p, P)) = \{(p', P') | a \in \Sigma : p' = \delta(p, a), P' = \{p'' | p'' = \bigcup_{\forall p \in P} \delta(p, a)\}\}$. Pro odstranění redundance stavů v jednom makro-stavu je opět použita hashovací tabulka, uchovávající názvy stavů, které makro-stav obsahuje. V Next a Processed jsou uchovávány struktury product_st se třemi položkami – odkazu na stav z A, neseřazené množiny stavů makro stavu z B a booleovskou proměnnou rejecting. Množina stavů je neseřazená, protože se v ní nevyhledává, ale iteruje.

3.2 Implementační jazyky

Pro účely této práce byly využívány pouze základní verze jazyka se standardními knihovnami bez jakýchkoli externích prvků. Pro C++ je to standard C++17, Python verze 3.7.6, C# verze C#8.0 (.NET Core 3.1) a OCaml verze 4.08.1.

C++

C++ je populární programovací jazyk navržený dánským informatikem Bjarne Stroustrupem. Vznikl přidáním tříd do jazyka C, po kterém je pojmenovaný a od kterého zdědil efektivitu, nízkoúrovňovou sémantiku a flexibilitu. Další inspirací byly jazyky Simula, Algol68 a BCPL. První verze C++ byla vytvořena roku 1985 s mnoha novinkami, například virtuální funkce, přetěžování operátorů, reference, konstanty a manuální správa paměti. Následovalo 32 let vývoje do současné podoby (C++17), která podporuje polymorfizmus, dědičnost, templates, výjimky, třídy jmen a další. C++ v průběhu své evoluce ovlivnilo mnoho moderních jazyků – Java, PHP, C#, Python a jiné. Tento univerzální jazyk nabyl za dobu své existence obrovské celosvětové obliby a pravidelně okupuje vrchní příčky v žebříčcích popularity programovacích jazyků.

Sám Bjarne Stroustrup označil C++ za lepší C, se kterým si dodnes uchovává zpětnou kompatibilitu. Dále se dá stručně charakterizovat jako staticky typovaný kompilovaný general-purpose jazyk podporující vícero programovacích stylů – imperativní, objektově-orientovaný, funkcionální a generický. V této práci představuje ve srovnání s ostatními testovanými jazyky jazyk starší, nízkoúrovňový s komplikovanou syntaxí a údajně velmi rychlý a efektivní. Informace v těchto dvou odstavcích byly čerpány z [16].

Pro překlad tohoto jazyka byl použit program g++ s úrovní optimalizací -O3, tedy nejvyšší možnou. Na základě zběžného porovnání je odhadováno, že neoptimalizovaná verze je přibližně 4-4,7krát pomalejší než verze optimalizovaná.

Python

Dalším z vybraných jazyků je Python, vytvořený roku 1990 nizozemským informatikem jménem Guido van Rossum. Hlavní inspirace pochází z jazyka ABC, na kterém van Rossum předtím pracoval. V průběhu svého vývoje až do roku 2020 si prošel třemi verzemi. První vydaná roku 1990, druhá roku 2000 a třetí roku 2008. Jelikož jazyk prošel mezi druhou a třetí verzí výraznou proměnou, jsou tyto verze nekompatibilní a dodnes jsou vyvíjeny souběžně. Python se tedy, i přes problémy s tím spojené, vydal jinou cestou než C++ a jeho vztah s C. I díky tomu je Python 3 poměrně moderní programovací jazyk s mnoha prvky usnadňující programátorům práci, jako například generátorová notace seznamů (anglicky list-comprehensions) a intuitivní práce s datovými strukturami nebo extrémně příjemná vlastnost, kdy programátor pracuje pouze s referencemi na objekt a ne s objektem samotným.

V této práci zastupuje skriptovací dynamicky typované interpretované jazyky. Podporuje imperativní, objektově-orientované a částečně také funkcionální programování. Myšlenka Pythonu od začátku spočívá v jednoduchosti a přehlednosti kódu. Hlavním cílem je čitelnost a estetický dojem, programy psané v Pythonu jsou často napsány na menším počtu řádků a vývoj trvá kratší dobu. Jedná se tedy o poměrně vysokoúrovňový jazyk, který ovšem dokáže být ve své základní podobě bez externích knihoven (a mnohdy i s nimi) poměrně limitující. Další z jeho nevýhod je jeho údajně nízká rychlost ve srovnání s ostatními jazyky z tohoto výběru. Informace o Pythonu byly získány z [15].

V této práci byla použita klasická implementace Pythonu, tedy CPython. Ten nenabízí téměř žádné možnosti optimalizace, které by zásadně ovlivnily výkon, optimalizace tedy nebyly použity. Nabízí se ale možnost použít jinou implementaci Pythonu, například PyPy, která v testu na malém vzorku automatů často dosahovala velmi dobrých výsledků – pro některé algoritmy byla až 7krát rychlejší (relace simulace, test prázdnosti) nebo 4krát rychlejší (minimalizace), pro jiné byla ovšem rychlejší jen asi 1,1krát (determinizace, zbytečné stavy). Zdá se tedy, že čím náročnější výpočet, tím lepší má PyPy efekt. Dá se tedy předpokládat, že pro práci s automaty by tato alternativní implementace Pythonu měla být výrazně lepší. Jako možné budoucí rozšíření této práce se tedy nabízí otestovat více různých implementací Pythonu (například PyPy, Jython nebo IronPython).

C#

Třetím vybraným jazykem je další z nových vysokoúrovňových general-purpose jazyků, tentokrát silně staticky typovaný a semi-kompilovaný (kód je přeložen do CIL formátu a následně je interpretován). Vyznačuje se jeho výraznou objektovou orientací, neexistují globální funkce a proměnné, vše musí být definováno ve třídě. Netají se tím, že inspiraci čerpá hlavně z C++ a Javy, od kterých přejímá mnoho prvků jako garbage collector, referenční a hodnotový typ, syntaxi atd. Byl navržen Andersem Hejlsbergem. Vyvíjen je společností Microsoft pro platformu .NET. To je zároveň i jeho nevýhodou, jelikož stále slouží hlavně pro vývoj aplikací pro operační systém Windows, což se ale pomalu mění.

Stejně jako jazyk Python se snaží o jednoduchost a přehlednost kódu, zároveň však od C++ přebírá robustnost a syntaxi. Dá se tedy z tohoto pohledu považovat za kompromis

mezi Pythonem a C++. Rychlostně by měl být také někde mezi těmito dvěma jazyky. Informace o C# byly získány z [13].

Pro překlad tohoto jazyka byl použit program Visual Studio 2019 a v něm zabudovaný překladač. Byly povoleny optimalizace pomocí přepínače -optimize. Tento jazyk má velké výkyvy v rychlosti provádění programu, takže je těžké určit, jak moc optimalizace zlepšila výkon. Naměřené rozdíly byly extrémně malé, neoptimalizovaná verze byla pouze asi 1,15krát pomalejší než optimalizovaná.

OCaml

Posledním jazykem je OCaml, který je objektově orientovanou verzí jazyka Caml (Categorical Abstract Machine Language). Jeho předchůdcem je jazyk ML (meta-language), který výrazně ovlivnil mnoho dnešních funkcionálních jazyků. Vývoj Camlu začal v osmdesátých letech francouzským institutem pro informatiku a automatizaci (Inria), vedoucím vývoje byl Gérard Huet. V devadesátých letech se potom jazyk z důvodu jejich rostoucí popularity rozšířil o třídy a objekty a vznikl OCaml. Ten si postupně získal značnou popularitu a dnes nachází využití hlavně jako výzkumný jazyk na univerzitách.

OCaml je zástupce funkcionálního deklarativního programovacího jazyka, avšak umožňuje i zjednodušené imperativní konstrukce a jejich kombinování s funkcionálními prvky. Tento přístup tak nelimituje programátora v rozhodování, jak vyřeší daný problém, a umožňuje implementovat jednotlivé části programu nejvhodnějším způsobem. Jinak se jedná o kompilovaný dynamicky typovaný general-purpose jazyk. Hlavní důraz je podle jeho tvůrců kladen na bezpečnost, vyjadřovací schopnost (expressiveness) a rychlost. Dále se jazyk chlubí propracovaným typovým systémem, modulovým a objektovým systémem, pokročilým vyhledáváním vzorců (pattern matching) a automatickou správou paměti. Informace o OCamlu byly převzaty z [11] a [12].

Jazyk OCaml byl přeložen pomocí programu ocamlopt. Tento program implicitně optimalizuje pro rychlost, překlad tedy nebyl upraven žádným přepínačem. Existují sice možnosti, například -inline nebo -match-context-rows a další, ale žádná z nich neměla při testování zásadní vliv na rychlost výpočtu. Tyto parametry zlepšily rychlost jen 1,06krát.

Kapitola 4

Měření efektivity programovacích jazyků a výsledky

V této kapitole jsou prezentovány výsledky měření pro jednotlivé algoritmy, metodika měření a závěrečné porovnání a zhodnocení efektivity vybraných programovacích jazyků při práci s konečnými automaty.

4.1 Popis měření

Žádné z měření neskončilo nestandardním způsobem, tedy chybou. Implementace byly průběžně testovány při tvorbě na sadě malých pokusných automatů, díky čemuž byla odstraněna většina chyb. Dále byly namátkově porovnávány vzájemné výsledky jednotlivých jazyků při hlavním měření. Výsledky byly vždy stejné a dávaly smysl, což lze vypozorovat i z výsledných grafů.

Automatové algoritmy pro měření byly uchovávány v textových souborech, ze kterých byly postupně načítány do příslušných datových struktur. Tento proces nebyl součástí měření. Měřeným algoritmům byly automaty následně předávány jako parametry funkce. Měření probíhalo na počítači s 64-bitovým operačním systémem Fedora 31, procesorem Intel s frekvencí 3.2GHz a 4096MB operační paměti. Měření bylo prováděno pomocí unixového bash skriptu, přičemž každý automat byl zpracován všemi čtyřmi jazyky a poté se přešlo na další. Celkem bylo měřeno pět statistik, a to počet stavů automatu, počet přechodů automatu, procesorový čas, reálný čas a nejvyšší spotřeba paměti. Reálný čas a počet přechodů automatu nakonec ale nepřinesly žádné zajímavé výsledky a nebyly tedy použity. Aby měření netrvalo příliš dlouhou dobu, byla délka zpracovávání automatu pro každý jazyk omezena pomocí unixového programu timeout u rychlejších algoritmů (test prázdnosti, zbytečné stavy a determinizace) na dvě minuty, u ostatních na jednu minutu. Toto nutné opatření způsobilo, že u některých algoritmů pro velké automaty nebylo dosaženo výsledku. Pro účely následujících statistik nebyly tyto výsledky použity.

Měřena byla jen doba běhu algoritmů uvedených v kapitole 3.1, načítání automatů do datových struktur a závěrečné výpisy výsledků nejsou zahrnuty. Dále byl řešen problém jazyka C#, který spočíval v tom, že je jen velmi obtížné získat opravdu přesné výsledky pro krátká měření. Výchozí vzorkovací frekvence pro funkce měřící čas je totiž 16 milisekund. Údajně je možné ji sice komplikovaným způsobem zpřesnit až na 1 milisekundu, to je ovšem pro účely tohoto typu měření stále příliš nepřesné. Z tohoto důvodu probíhalo měření ve všech jazycích, aby byly zachovány férové podmínky, v cyklu, který opakovaně spouštěl daný

algoritmus po dobu minimálně 500 milisekund a počítal počet provedení. Výsledný čas pro jeden průchod byl poté vypočítán jako (celková doba měření) / (počet provedení). Za tuto dobu byl čas aktualizován alespoň 31 krát, což po vydělení počtem provedení podle autora dostatečně minimalizuje chybu. Minimální počet měření byl stanoven na dva pro jakkoli velké automaty. Toto řešení je vhodné i z toho důvodu, že funguje jak pro malé, tak pro obrovské automaty. Druhým možným řešením bylo spouštět algoritmus v cyklu s pevným počtem opakování, měření velkých automatů by však trvalo věčnost. Měřena byla přesnost až na nanosekundy, přičemž platí, že čím rychlejší byl jeden průchod algoritmem, tím menší je chyba. V ostatních jazycích tento problém nebyl.

Nejvyšší spotřeba paměti byla měřena pro celý proces pomocí GNU utility time.

K měření byla využita kolekce celkem dvou set automatů o velikostech od 3 až do 802 stavů, přičemž pro zajímavost bylo přidáno šest automatů o velikostech 3759 až 3781 stavů. Automaty byly získány z náhodných regulárních výrazů nebo byly vygenerovány z jednotlivých kroků verifikace algoritmů (byla použita podmnožina stejných automatů jako v [2]). Průměrný počet stavů byl 372 a přechodů 1502. U některých algoritmů ovšem může být počet stavů zpracovávaných automatů o hodně vyšší. Například algoritmus minimalizace DKA vyžaduje předchozí determinizaci automatů, která může počet stavů výrazně navýšit. Kolekce občas obsahuje dvojice nebo trojice podobných verzí automatů, například nedeterministickou a deterministickou nebo verzi s několika přidanými stavy, přechody nebo znaky abecedy. Pro čtyři jazyky, deset různých verzí algoritmů a dvě stě automatů bylo získáno 8000 výsledků. Jelikož výpočet pro velké automaty často trval příliš dlouho a nebyl pro některé algoritmy dokončen, bylo později přidáno ještě 10 automatů o velikostech 2400 až 2600 stavů, pro které byl zároveň i upraven postup měření – bylo umožněno, aby byl průchod algoritmem měřen pouze jednou. Z toho důvodu nejsou tato doplňující měření tak přesná, a tudíž jsou výsledky uváděny samostatně. Tato měření, která byla prováděna jen pro ty algoritmy, pro které nejsou k dispozici výsledky velkých automatů z hlavního měření, by čtenáři měla dát hrubou představu o výkonech jazyků při provádění takových algoritmů s velkými automaty.

4.2 Výsledky

Následuje výčet výsledků měření pro všechny vybrané algoritmy.

4.2.1 Test prázdnosti

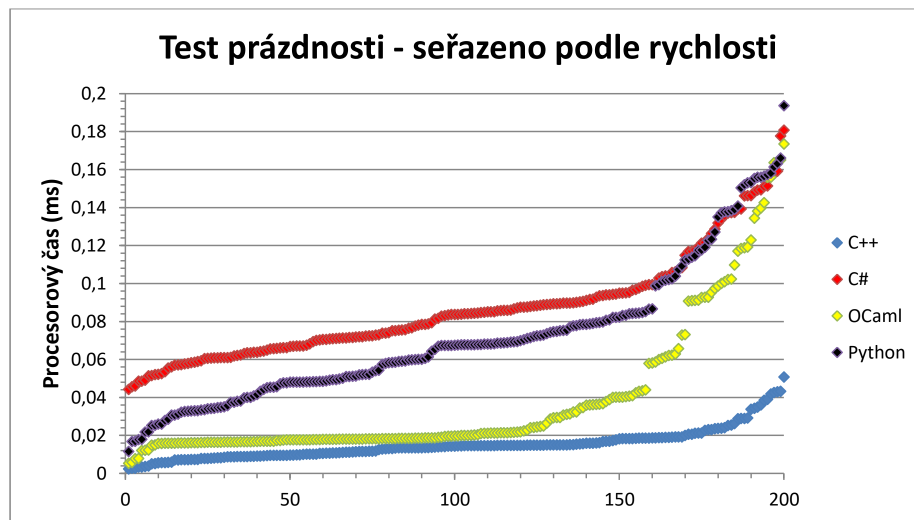
Test prázdnosti není nijak náročný algoritmus, obzvlášť, když je implementován prohledáváním do hloubky. Pro každý jazyk tedy úspěšně proběhlo všech 200 měření. V tabulce 4.1 jsou uvedeny maximální, minimální a průměrné časy v milisekundách pro jednotlivé jazyky, vypočítané ze všech 200 výsledků. Jsou v nich započítány všechny automaty všech velikostí. Některé jazyky jsou lepší pro menší automaty, jiné naopak. Jelikož má ale tato práce za cíl celkově porovnat vybrané jazyky, může tato tabulka sloužit jako hrubý indikátor pořadí jazyků.

Obecně se dá říct, že C++ je jednoznačně nejrychlejší ve všech kategoriích. Pro menší automaty mu relativně dobře konkuruje OCaml, který zaostává průměrně o 0,023 milisekund, je tedy přibližně 2,56krát pomalejší. Dalším v pořadí je Python, kde je rozdíl 0,056 milisekund, je tedy 4,74krát pomalejší než C++. Možná trochu překvapivě je poslední C#, což může být dáno tím, že je primárně určen pro operační systém Windows, kdežto měření proběhlo na unixovém OS. Za C++ zaostal průměrně o 0,071 milisekund (5,74krát

Tabulka 4.1: **Test prázdnosti:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	0,050743	0,180766	0,173460	0,193624
Minimální čas:	0,002366	0,044311	0,005325	0,011519
Průměrný čas:	0,015066	0,086496	0,038634	0,071448
Medián:	0,014447	0,083752	0,019739	0,067586

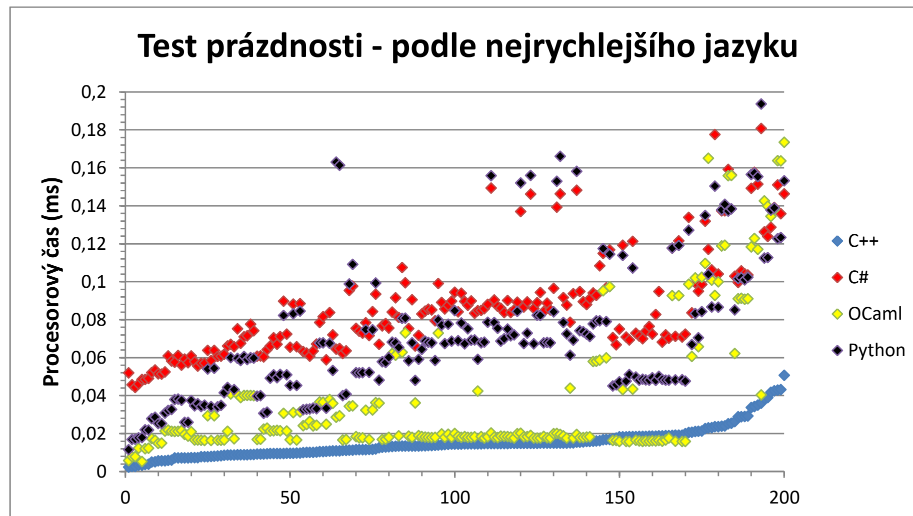
pomalejší). Graf na obrázku 4.1 zobrazuje všechny provedené výpočty pro každý jazyk, seřazené podle procesorového času bez ohledu na velikost. Osa x zobrazuje pořadí automatu.



Obrázek 4.1: **Test prázdnosti: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.

Obrázek 4.1 pěkně doplňuje předchozí tabulku, neboť ukazuje vývoj časů od minimálních až po maximální. Z grafu je na první pohled patrné, že C++ je i zde vítězem. Nejenže má konstantně na každém místě pořadí nejrychlejší časy, ale zároveň má i nejmenší nárůst trvání výpočtu. Nejpomalejší výpočet byl jen o trochu pomalejší než nejrychlejší výpočet C#. Další zajímavý fakt je, že OCaml byl zprvu schopen C++ dobře konkurovat, ale se zvyšujícím se trváním výpočtu se výrazně propadl až na úroveň Pythonu a C#. Zároveň má mezi 150. a 200. dobře zřetelné shluky výsledků. Algoritmus prochází stavy a zároveň si názvy již navštívených stavů ukládá do hashovací tabulky. Tyto shluky automatů tedy odpovídají přepočítávání při zvětšování této tabulky. Byly provedeny testy pro potvrzení tohoto výkladu a ukázalo se, že při výrazném zvýšení velikosti tabulky při inicializaci tyto shluky zmizely a graf se "vyhladil", avšak zvýšila se paměťová náročnost. Z toho vyplývá, že pokud jsou potřeba konzistentní výsledky a mírné zrychlení, je dobrý nápad přizpůsobit velikost hashovací tabulky počtu stavů zpracovávaných automatů a optimalizovat nebo zakázat zvětšování tabulky (například v případě kolize uložit prvek do zřetězeného seznamu příslušného pole tabulky). To stejné je patrné v menší míře u Pythonu, jehož slovník takto také funguje. Dále lze z grafu zjistit, že se C# se zvyšující dobou výpočtu ve srovnání s ostatními jazyky výrazně zlepšuje a její nárůst je po C++ nejmenší. U tohoto algoritmu doba výpočtu příliš nezávisí na počtu stavů a přechodů, ale spíše na tom, jak moc má automat od počátečního stavu slepých větví a jestli má koncový stav. Automaty s vyšším

časem výpočtu tedy byly často ty, které neměly koncový stav, a algoritmus tedy musel projít všechny dostupné stavy a uložit je do hashovací tabulky.



Obrázek 4.2: **Test prázdnosti: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

Obrázek 4.2 ukazuje jiný pohled na řazení dle rychlosti, konkrétně podle nejrychlejšího jazyka. Pro dané x jsou tedy zobrazeny výsledky všech čtyř jazyků pro stejný automat.

Porovnáme-li dobu výpočtu a počet stavů, zjistíme, že OCaml má problém s automaty, které pravděpodobně nemají koncový stav nebo vyžadují dlouhé hledání. U automatů, u kterých je koncový stav nalezen hned na začátku se vyrovná C++. Naproti tomu C# a pro více stavů i Python mají vyšší základní dobu výpočtu i pokud je koncový stav nalezen rychle.

Tabulka 4.2 ukazuje maximální spotřebu paměti (peak memory usage) procesu všech výpočtů. Podezřelá je vysoká spotřeba paměti u C#, která může být způsobena měřením na unixovém OS, ale i potřebou spustit spoustu dalších programů pro běh C# aplikace jako například JIT compiler nebo Common language runtime (CLR) a další. Zde je vidět, že C++ je průměrně nejlepší, má však vyšší maximální spotřebu paměti než OCaml. To by z části vysvětlovalo bídné výkony OCamlu při delších výpočtech, kdy tento jazyk pravděpodobně příliš šetří pamětí na úkor rychlosti. Průměrně potřebuje OCaml 1,34krát, C# 7,86krát a Python 2,82krát více paměti než C++.

Tabulka 4.2: **Test prázdnosti:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	12908	45332	10444	17232
Minimální:	3624	33048	5416	12288
Průměrná:	4471	35171	6023	12620
Medián:	4148	35570	5852	12500

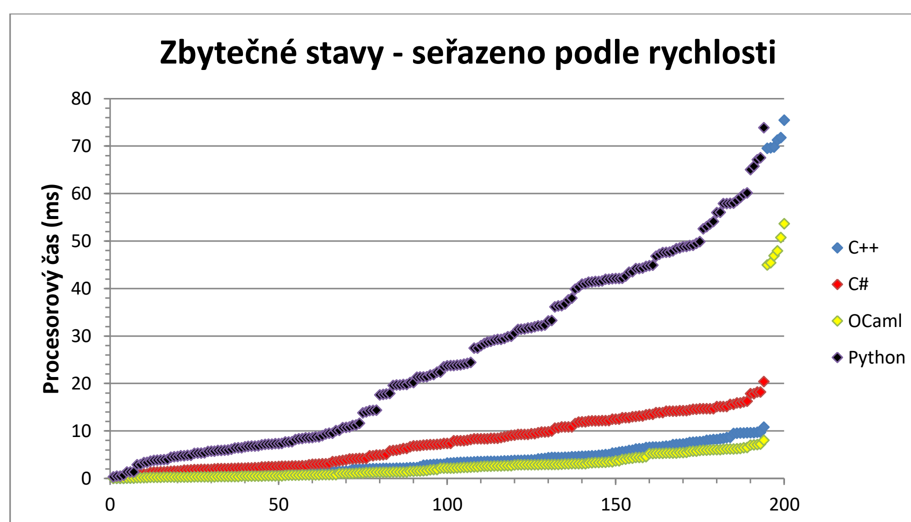
4.2.2 Odstranění zbytečných stavů

Odstranění zbytečných stavů, tedy neukončujících a nedostupných, má v první fázi velmi podobný průběh jako test prázdnosti s tím rozdílem, že je třeba projít téměř všechny stavy dvakrát (od počátečních a koncových stavů) a uložit jejich názvy do dvou hashovacích tabulek. Ve druhé fázi jsou poté odstraňovány stavy, které se nenachází v obou hashovacích tabulkách. Tato fáze byla poměrně dobře optimalizována (kapitola 3.1.2), pro automat s nižším počtem odstraňovaných stavů by měla odpovídat době třetího průchodu přes stavy automatu. Z tohoto důvodu i zde úspěšně proběhlo všech 200 měření pro každý jazyk.

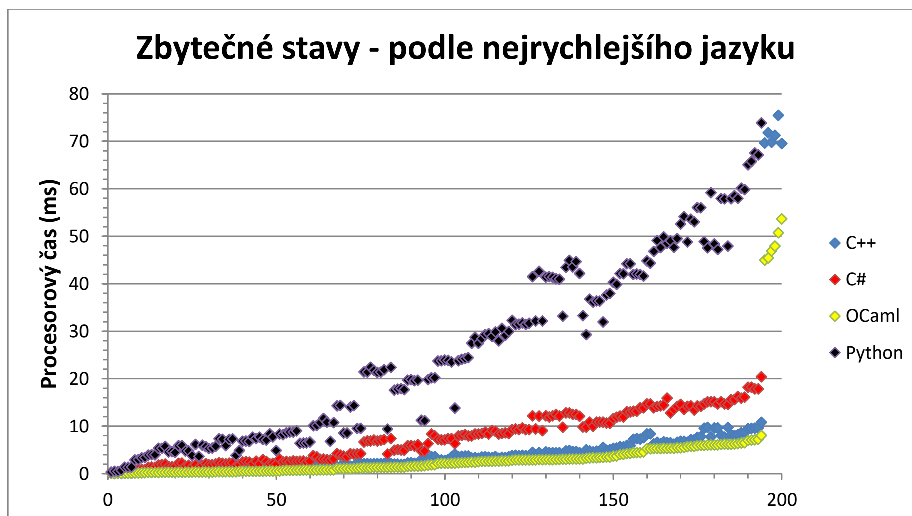
Z tabulky 4.3 a obrázků 4.3 a 4.4 je patrné, že nejrychlejším jazykem byl ve všech případech OCaml. Druhé C++ za ním průměrně zaostalo s časem 1,44krát horším, třetí C# byl 2,93krát pomalejší a poslední byl 9,38krát pomalejší Python. Důvod, proč tentokrát nebylo nejrychlejší C++, může být drobný rozdíl v implementaci (kapitola 2.2) a také to, že C++ nemá rozlišeny hodnotové a referenční typy tak, jako ostatní tři jazyky. Bylo tedy nutné oproti ostatním jazykům přidat operace udržující integritu ukazatelů při odstraňování stavů a náročném kopírování automatů pro opakovaný výpočet. S větším odstupem zaostával C#, který ovšem opět prokázal vlastnost, že pro menší automaty je pomalejší (v min. čase byl 6,95krát pomalejší než OCaml), ale pro náročnější výpočty vůči ostatním jazykům mírně zrychluje (v max. čase byl 2,38krát pomalejší než OCaml). Z obrázku 4.3 je vidět, že nejhorším jazykem byl s výrazným odstupem Python, který také ukázal velmi strmý nárůst doby výpočtu se zvyšující se náročností.

Tabulka 4.3: **Zbytečné stavy:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	C++	C#	OCaml	Python
Maximální čas:	75,450429	128,000000	53,667091	377,647316
Minimální čas:	0,015607	0,105197	0,015131	0,346691
Průměrný čas:	5,473448	11,076046	3,780147	35,479227
Medián:	3,253491	7,391304	2,171821	23,739694

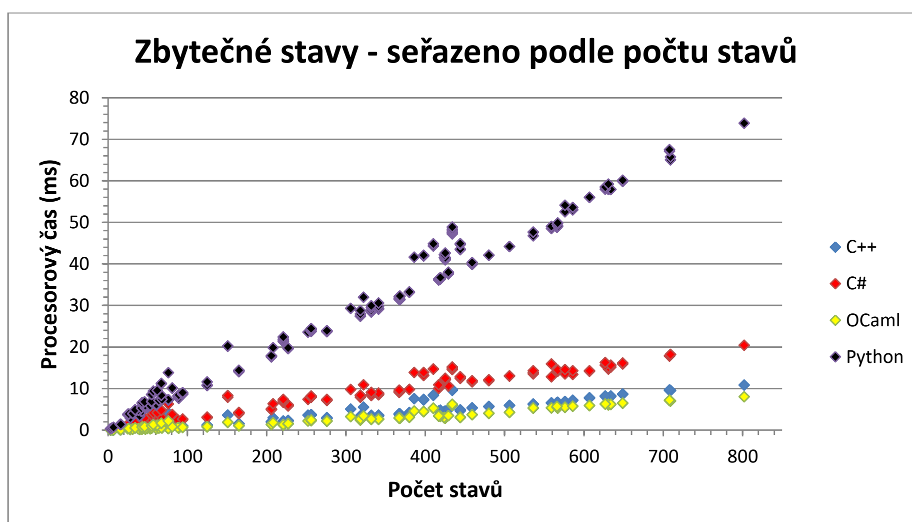


Obrázek 4.3: **Zbytečné stavy: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu. Měření pro šest velkých automatů nejsou pro Python a C# z důvodu čitelnosti grafu uvedena.



Obrázek 4.4: **Zbytečné stavy: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky. Měření pro šest velkých automatů nejsou pro Python a C# z důvodu čitelnosti grafu uvedena.

Obrázek 4.5 ukazuje závislost procesorového času na počtu stavů. V tomto případě je vidět, že body jednotlivých jazyků jsou přibližně lineárně seřazeny, doba výpočtu je tedy poměrně dost závislá na počtu stavů u všech jazyků. Dává to smysl, protože obvykle se příliš moc stavů neodstraňuje, a tím pádem nejsou v průměru výrazné rozdíly mezi automaty s podobným počtem stavů. Nejpatrnější jsou tyto rozdíly u Pythonu, nejméně patrné pak u OCamlu.



Obrázek 4.5: **Zbytečné stavy: závislost rychlosti na počtu stavů.** Výsledky jsou seřazeny podle počtu stavů automatu.

Tabulka 4.4 obsahuje souhrn nejvyšší spotřeby paměti v kB. Zajímavá jsou opět obrovská čísla u jazyka C#. Průměrně nejšetrnější bylo C++, které mělo větší spotřebu paměti než OCaml pro velké automaty, ale naopak menší spotřebu pro malé. Ovšem vzhledem k tomu,

že s malými automaty obvykle problémy nejsou a cení se hlavně nízké nároky a vysoká rychlost u velkých automatů, má OCaml výsledky ve finále lepší. Následuje Python, který potřebuje průměrně 3,42krát více paměti než C++ a nejhorší je opět výrazně C#, který byl 9,74krát horší než C++ a maximální spotřeba činila těžko pochopitelných 581MB.

Tabulka 4.4: **Zbytečné stavy:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	21832	581648	14032	50268
Minimální:	3616	32996	5464	12296
Průměrná:	5163	50338	6849	17693
Medián:	4580	40870	6224	17016

4.2.3 Produkt

Algoritmus pro produkt je sice ještě poměrně jednoduchý, obsahuje frontu dvojic stavů a hashovací tabulku nově vytvořených stavů, ale v jednom je jiný než předchozí dva algoritmy – vyžaduje dva automaty. To znamená, že počet stavů uváděný v následujících grafech je součtem stavů obou automatů, což jej výrazně zvyšuje. Z toho důvodu všechny výpočty proběhly pouze v jazycích C++, C# a OCaml, v jazyce Python výpočet produktu největších automatů trval příliš dlouho a byl ukončen (celkem pět výpočtů). Ve statistikách se tedy s těmito pěti výsledky v žádném jazyce nepočítá a nepromítají se ani do údajů v tabulkách. Pro zajímavost průměrná rychlost těchto výpočtů v C++ byla 297,892 milisekund, v C# 1484 milisekund a v OCamlu 701,810 milisekund. Tabulka 4.5 zobrazuje statistiky procesorového času. Průměrně nejrychlejší pro tento algoritmus bylo poměrně výrazně C++, druhý byl 2,58krát pomalejší OCaml, třetí byl C# (4,84krát pomalejší) a poslední s obrovským odstupem Python (14,84krát pomalejší), aniž by k tomu měl z pohledu implementace nějaký zvláštní důvod. Opět lze sledovat, že C# má vysoký minimální čas, v tomto případě stejný jako Python, ale maximální čas má přibližně třikrát nižší. Tento nižší nárůst doby trvání je dobrá vlastnost při práci s automaty.

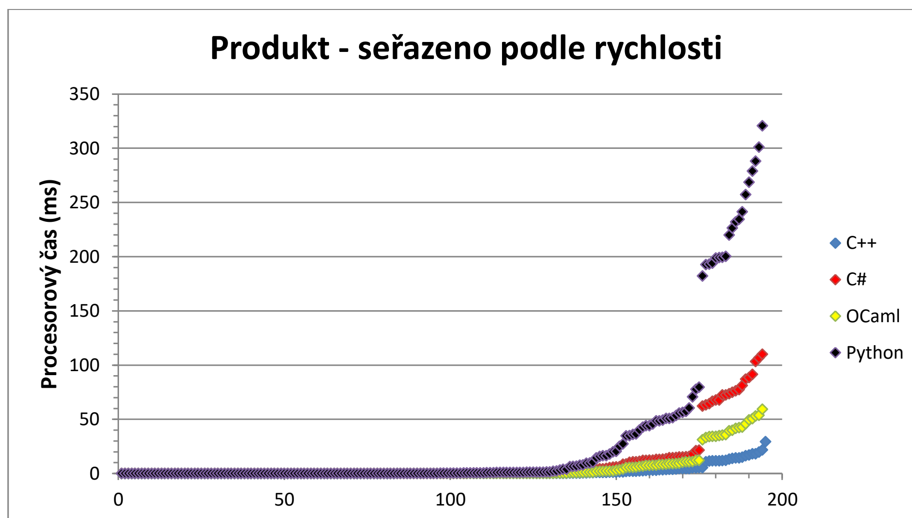
Tabulka 4.5: **Produkt:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	29,303579	110,000000	59,382700	320,751138
Minimální čas:	0,003588	0,095859	0,017983	0,099387
Průměrný čas:	2,046983	9,926258	5,299202	30,389332
Medián:	0,020169	0,208632	0,055974	0,279431

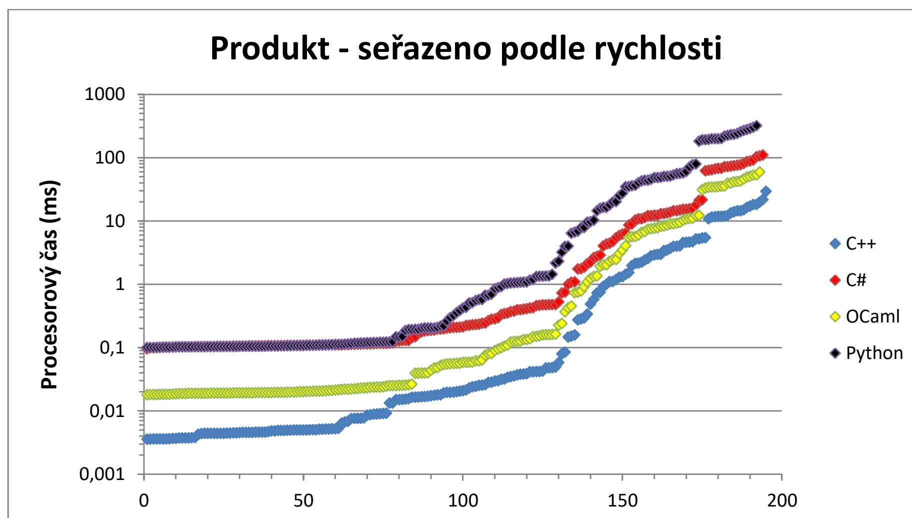
Tabulka 4.6: **Produkt – doplňující měření:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	153,845000	700,000000	372,125500	1781,310289
Minimální čas:	131,271400	610,000000	332,081500	1620,438562
Průměrný čas:	139,650756	635,555567	348,028722	1693,233731
Medián:	139,411600	630,000000	350,057000	1669,730811

Obrázky 4.6, 4.7 a 4.8 potvrzují informace získané z tabulky, tedy obrovský nárůst doby výpočtu u Pythonu a ostatní tři jazyky víceméně udržující poměr trvání. Druhý graf zobrazuje osu y v násobcích 10, jelikož jsou velké rozdíly mezi rychlými a pomalými výpočty.

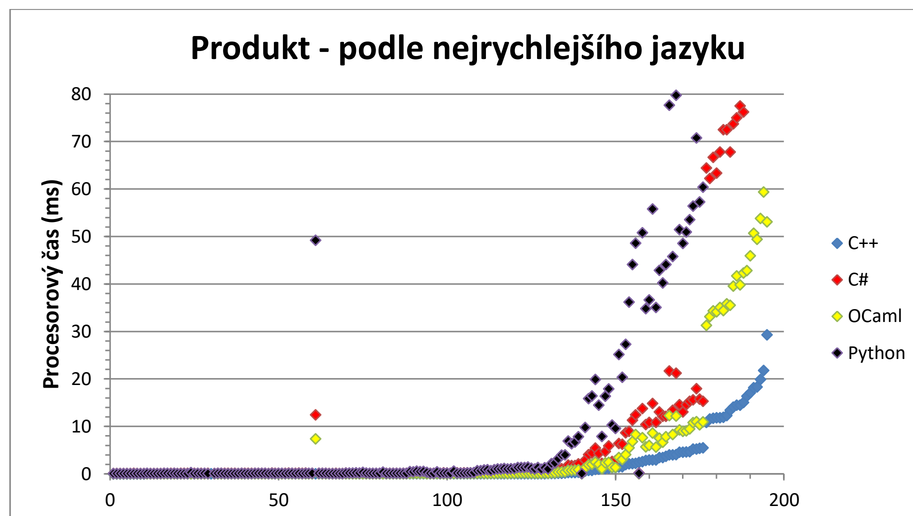


Obrázek 4.6: **Produkt: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu. Výsledky pro největší automaty nejsou z důvodu čitelnosti grafu uvedeny.



Obrázek 4.7: **Produkt: srovnání jazyků s logaritmickou stupnicí osy y.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu. Výsledky pro největší automaty nejsou z důvodu čitelnosti grafu uvedeny.

U tohoto algoritmu ve všech jazycích doba výpočtu příliš nezávisí na počtu stavů, záleží hlavně na tom, jak moc podobné mají automaty abecedy a jak velký je průnik jejich jazyků. Jako poslední je uvedena tabulka 4.7 paměťových nároků jazyků. Nejnižší je v průměru má C++, následuje OCaml (1,51krát více), Python (2,90krát více) a konečně C# (8,01krát více). Opět lze vidět ve srovnání s ostatními jazyky obrovské nároky jazyka C#.



Obrázek 4.8: **Produkt: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

Tabulka 4.7: **Produkt:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	15072	210396	18632	27716
Minimální:	3548	33052	5636	12444
Průměrná:	5012	40188	7585	14579
Medián:	4640	35858	6488	12656

Tabulka 4.8: **Produkt – doplňující měření:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	31152	980496	39464	55592
Minimální:	30344	745240	38432	54860
Průměrná:	30702	856123	38701	55299
Medián:	30608	868368	38524	55448

4.2.4 Determinizace

Determinizace je svou strukturou velmi podobná algoritmu pro výpočet produktu. Místo fronty dvojic stavů obsahuje frontu množiny spojených stavů, následně pro každý stav této množiny nalezne všechny přechody pro dané písmeno abecedy (vyhledání v hashovací tabulce) a uloží nový stav. Navíc ovšem obsahuje optimalizace pomocí dalších dvou hashovacích tabulek a jedné seřazené množiny. Oproti algoritmu pro produkt ale pracuje s polovičním počtem stavů. Pro každý jazyk tedy úspěšně proběhlo všech 200 výpočtů.

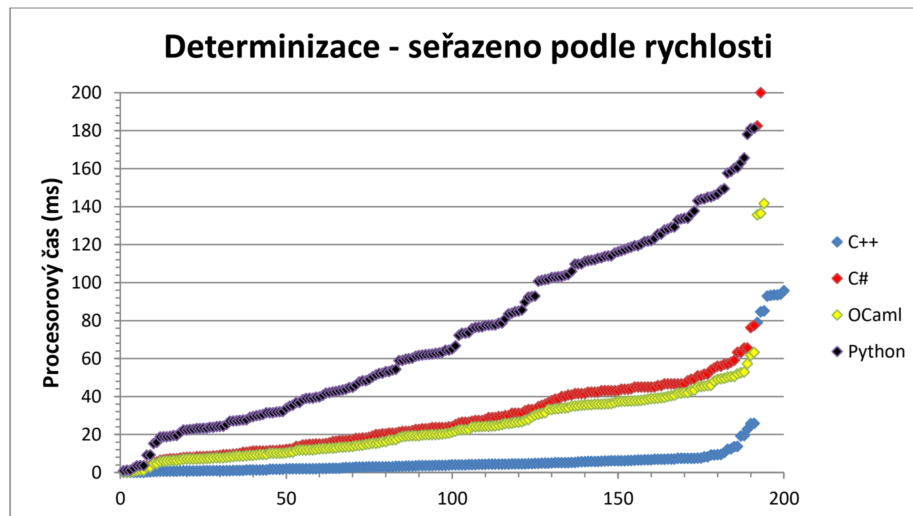
Z tabulky doby výpočtu 4.9 lze zjistit, že nejrychlejší bylo velmi výrazně C++, za ním poté OCaml (průměrně 4,26krát pomalejší), C# (5,22krát pomalejší) a nakonec Python (13,48krát pomalejší). Tento algoritmus úplně nevyhovoval OCamlu, který se jindy blíží C++, ale zde je spíše na úrovni C#. To lze přičíst tomu, že tento algoritmus často tvoří nové stavy, které ukládá do hashovacích tabulek. Algoritmus testu prázdnosti naznačuje, že tato datová struktura by mohla být v OCamlu pomalejší. Dále C# je opět pro menší automaty pomalejší, ale s rostoucí obtížností výpočtů se ve srovnání s ostatními jazyky zlepšuje. Další

zajímavostí je to, že minimální průchod algoritmem produktu měli C# a Python stejný, ale zde má Python minimální průchod více než dvakrát pomalejší. Maximální průchod produktem měl Python třikrát pomalejší, kdežto tady je přibližně dvakrát pomalejší.

Tabulka 4.9: **Determinizace:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	95,667571	515,000000	394,265000	1198,572741
Minimální čas:	0,027070	0,469925	0,267594	1,008975
Průměrný čas:	8,260864	43,177005	35,215732	111,405966
Medián:	4,041324	24,664502	21,491920	65,908563

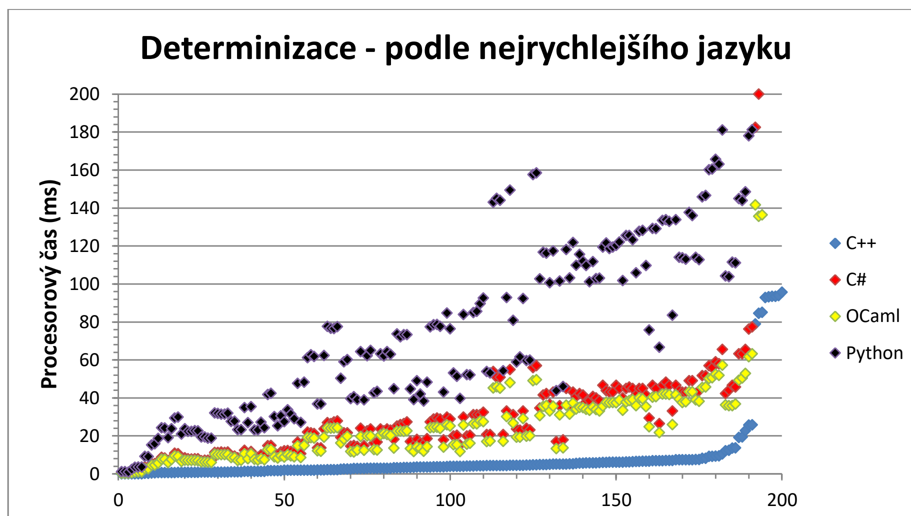
Uvedeny jsou obrázky 4.9 a 4.10, první je klasický graf jednotlivých výpočtů seřazených podle rychlosti pro každý jazyk zvlášť, druhý obsahuje výpočty seřazené podle nejrychlejšího jazyku. Z prvního grafu je vidět poměrně výrazný nárůst Pythonu hned od nízkých hodnot procesorového času. Naopak C++ ukazuje obdivuhodně nízký nárůst doby výpočtu. Nejobtížnější automaty dokonce dokázalo determinizovat do 100 milisekund. OCaml a C# mají velmi podobné průběhy.



Obrázek 4.9: **Determinizace: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu. Výsledky pro největší automaty nejsou z důvodu čitelnosti grafu uvedeny.

Python u většiny algoritmů projevuje velkou chaotičnost výsledků, jak je možné vidět v obrázku 4.10. Zatímco výsledky OCamlu a C# lze víceméně proložit křivkou (jsou tedy dost podobné C++), o Pythonu se to moc říci nedá.

Závěrem je uvedena tabulka spotřeby paměti 4.10. Nejméně paměti potřebuje C++, dále OCaml (1,56krát více), Python (2,96krát více) a C# (10,04krát více). Opět se ukazuje, že Ocaml je možná až zbytečně šetrný s pamětí při náročných průbězích, což může být na úkor rychlosti výpočtu. Jeho maximální spotřeba byla jen o 1264KB vyšší, než v případě C++. Minimální spotřebu měl ovšem téměř dvakrát větší než C++.



Obrázek 4.10: **Determinizace: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

Tabulka 4.10: **Determinizace:** srovnání jazyků podle spotřeby paměti v kB.

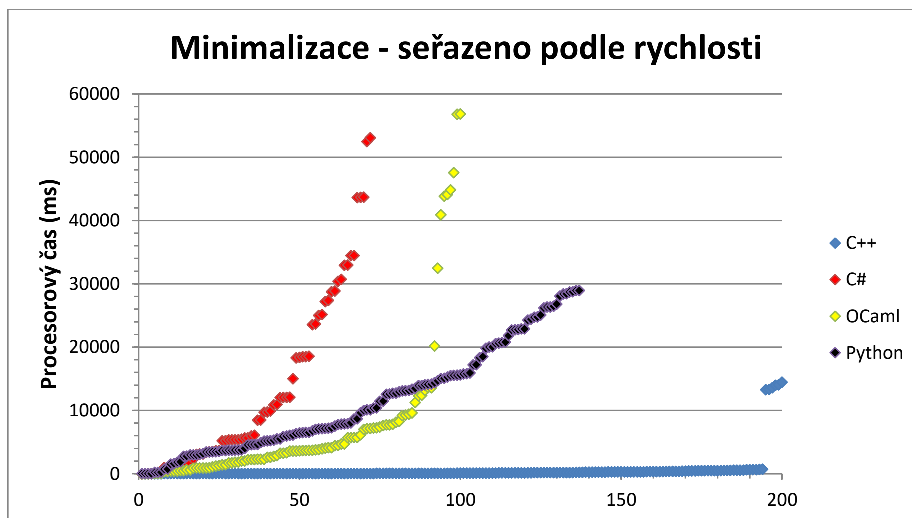
Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	16208	577720	17472	36828
Minimální:	3604	32808	5992	12300
Průměrná:	4774	47968	7460	14177
Medián:	4346	37198	6764	13148

4.2.5 Minimalizace

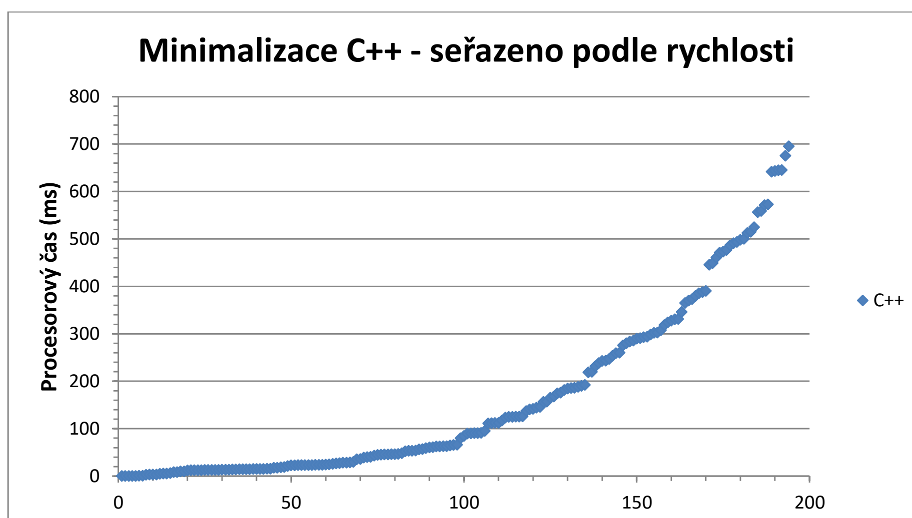
Algoritmus pro minimalizaci deterministického automatu se ukázal být velmi problematický. Automaty v sadě pro měření totiž nebyly všechny deterministické, před samotnou minimalizací bylo tedy třeba každý automat determinizovat. To u spousty automatů poměrně výrazně zvýšilo už tak velký počet stavů. Dále rychlost ovlivňuje i to, že tento algoritmus nebyl v zadání považován za pokročilý, nebyl tedy použit nejefektivnější způsob implementace. To nemusí úplně vadit, neboť tak tento algoritmus může sloužit jako extrémně náročný test. Výsledkem je to, že pro C++ proběhlo všech 200 měření, ale OCaml jich ve stanoveném čase zvládl jen 100, C# 72 a Python 137. Jelikož jsou tyto počty velmi malé, nejsou zde vypočítány některé statistiky. Sluší se ale zmínit, že C++ v tomto opravdu náročném testu podalo skvělé výkony, kdy i největší automaty o 3700 stavech před determinizací zvládlo minimalizovat za přibližně 14,5 vteřin a potřebovalo k tomu pouze 17568 KB paměti. Pro srovnání, ostatním jazykům trvala minimalizace automatů o 200-300 stavech mezi 10-50 vteřinami. Tento dominantní výsledek naznačuje, že pro opravdu velké automaty a náročné operace je C++ pravděpodobně jedinou smysluplnou volbou. Dále také velmi překvapil zatím rychlostně nejhorší jazyk – Python, který v tomto testu výrazně předčil jak OCaml, tak C#. Nejsou uvedena ani doplňková měření, neboť pouze C++ bylo schopno provést výpočty do stanoveného časového limitu.

Obrázky 4.11 a 4.12 zobrazují jednotlivé výsledky seřazené podle rychlosti. První srovnává všechny jazyky, druhý ukazuje pouze C++ bez šesti největších automatů. Nejhůře si

v tomto testu vedl C#, který ukazuje nejstrmější nárůst, Python byl trochu překvapivě o hodně lepší. OCaml si v minulých měřeních vždy vedl celkem dobře a mnohdy se C++ vyrovnával, ale nyní naprosto propadl.

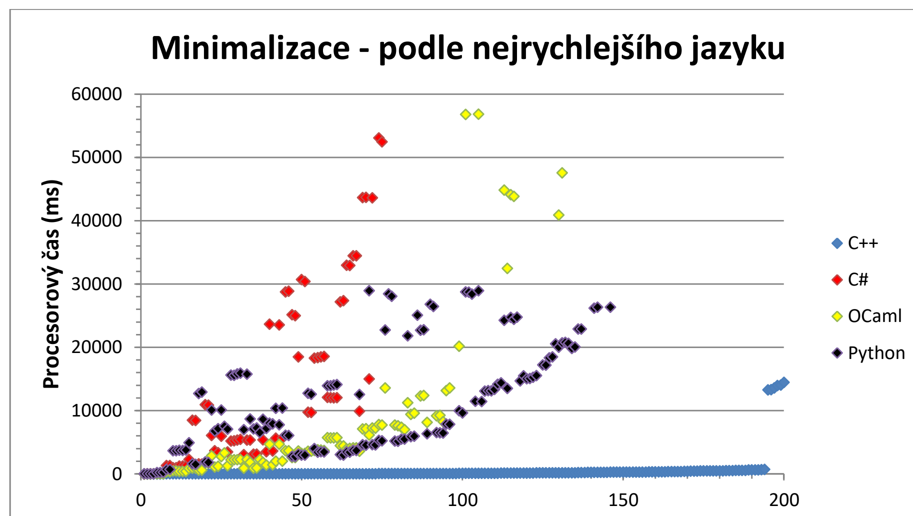


Obrázek 4.11: **Minimalizace: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.



Obrázek 4.12: **Minimalizace: průběh jazyka C++.** Jelikož bylo C++ zdaleka nejrychlejší a v ostatních grafech se nezobrazuje přehledně, je zde vyobrazen průběh pouze tohoto jazyka.

Obrázek 4.13 řadí výsledky podle nejrychlejšího jazyka. V případě C++ není po celou dobu vidět prakticky žádný nárůst procesorového času, C# a OCaml však strmě rostou. Python si udržuje pro nízkou náročnost výpočtů celkem vysokou rychlost, problém nastává u komplikovanějších případů, které vybočují z jeho hlavní křivky.



Obrázek 4.13: **Minimalizace: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

4.2.6 Výpočet relace simulace

Algoritmus výpočtu relace simulace byl dalším těžkým testem, protože pro velké automaty je tato relace obvykle velmi velká, a je třeba přes všechny páry této relace iterovat. Zároveň je potřeba vyhledávat informace o tom, jestli daný pár už byl do relace vložen. Dalším náročným místem výpočtu je získání množinového doplňku a tím i finální verzi relace simulace. Z toho důvodu žádný jazyk nezvládl výpočet pro 6 největších automatů a Python měl dokonce jen 176 úspěšných výsledků. Ostatní jazyky jich měly 194. Celkem vysoký počet výsledků je způsoben implementací nejefektivnější verze algoritmu, ne jako v případě minimalizace.

Tabulka 4.11 zobrazuje statistiky doby výpočtu, je nutno brát v potaz, že v případě Pythonu je započítáno jen 176 výsledků. Průměrně nejrychlejším jazykem se ukázalo být C++, druhým v pořadí byl OCaml (2,30krát pomalejší), dále C# (2,59krát pomalejší) a nakonec Python (7,32krát pomalejší). Z pohledu minimálního času Python výrazně zaostal, C# opět potvrdil, že je pro jednoduché výpočty pomalejší a čas OCamlu byl oproti C++ přibližně dvakrát pomalejší. V průměru se ale časy začaly výrazně srovnávat a při pohledu na maximální časy je zřejmé, že rozdíl mezi C# a OCamlem jsou minimální. Zejména u C# je to zajímavé, neboť v tomto výpočtu algoritmus umožnil, aby byla použita třída Array, která je údajně rychlejší než List pro rozsáhlé výpočty.

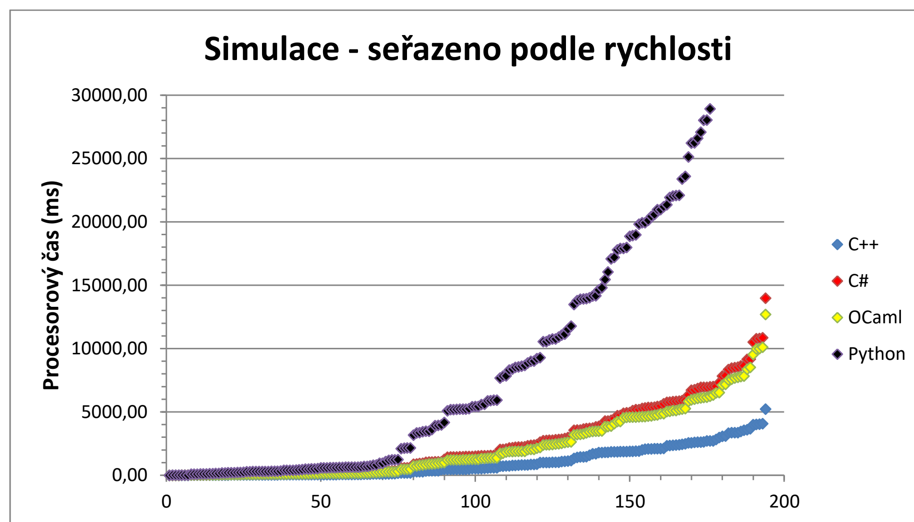
Tabulka 4.11: **Relace simulace:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	C++	C#	OCaml	Python
Maximální čas:	5222,208500	13970,000000	12697,648500	28910,860956
Minimální čas:	0,057512	0,222916	0,109111	0,599035
Průměrný čas:	1003,269458	2602,300243	2316,808883	7351,768550
Medián:	481,698167	1485,000000	1239,724000	3963,836094

Tabulka 4.12: **Relace simulace – doplňující měření:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	73423,109000	150910	147990,920000	###
Minimální čas:	62166,018000	132470	130030,038000	###
Průměrný čas:	67172,940000	140908	137722,783100	###
Medián:	65693,272500	139565	136086,784000	###

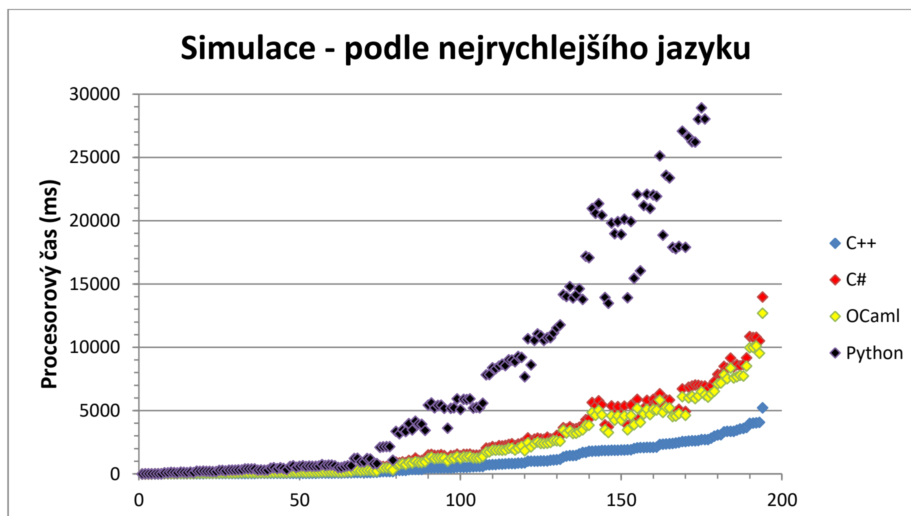
Graf seřazených výpočtů podle doby trvání 4.14 potvrzuje, že C# a OCaml vykazují velmi podobný průběh. C++ je výrazně rychlejší než ostatní jazyky. Python se drží ze začátku, poté ale strmě roste. Ve druhém grafu všechny jazyky udržují velmi podobné výsledky.



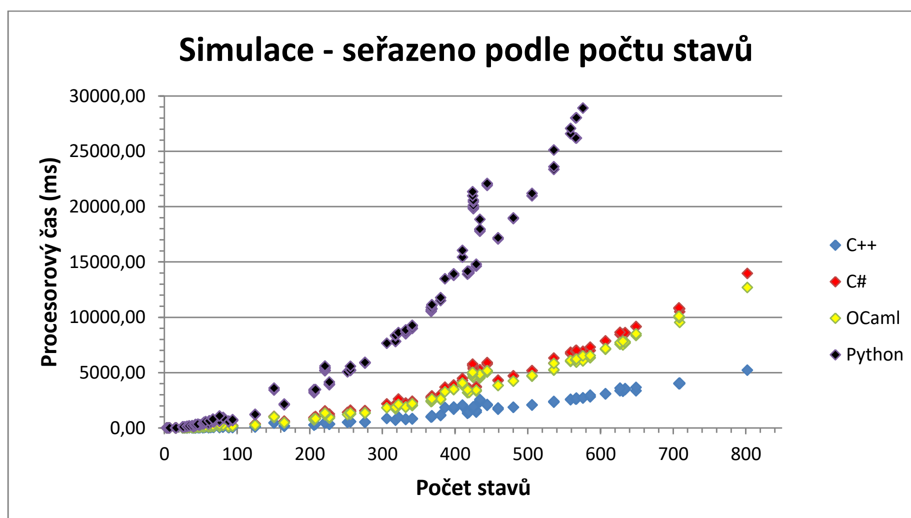
Obrázek 4.14: **Relace simulace: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.

Pokud si vytvoříme graf závislosti doby výpočtu na počtu stavů 4.16, zjistíme, že pro každý jazyk je tato závislost velmi silná. Počet stavů velmi spolehlivě určí i výslednou rychlost výpočtu. Jedinou výjimkou je Python mezi 400-500 stavy, kdy vykazuje rozdíly až 8 vteřin pro automaty o přibližně stejném počtu stavů.

Tabulka spotřeby paměti 4.13 je také nebývale vyrovnaná. Nejmenší průměrnou spotřebu mělo C++, dále pak Python (1,72krát více), OCaml (1,96krát více) a C# (3,01krát více). Poměrně menší spotřeba paměti C# při déle trvajících algoritmech (nestihne se tedy tolik opakování toho stejného algoritmu v rámci měření) autora vede k myšlence, že je toto velké číslo způsobeno fungováním garbage collectoru. Pro výpočet relace simulace, který trval 14 vteřin, bylo spotřebováno 329MB paměti, ale pro výpočet zbytečných stavů, trvající 128 milisekund, potřebuje 581MB. Dále je zajímavé, že jindy velmi šetrný OCaml potřeboval pro výpočet vysoké množství paměti. Jelikož je relace simulace většinou opravdu obrovská a byla v tomto jazyce uchovávána v hashovací tabulce, je možné, že od určitého vysokého počtu prvků OCaml začne zvyšovat alokovaný rozsah více než obvykle.



Obrázek 4.15: **Relace simulace: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.



Obrázek 4.16: **Relace simulace: závislost rychlosti na počtu stavů.** Výsledky jsou seřazeny podle počtu stavů automatu.

Tabulka 4.13: **Relace simulace:** Srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	129208	329472	222040	139916
Minimální:	3788	32884	6020	12092
Průměrná:	27400	82496	53785	47310
Medián:	17026	54506	33276	32872

4.2.7 Test univerzality

Algoritmus pro test univerzality je svou strukturou hodně podobný níže uvedenému algoritmu pro test inkluze. Má však oproti němu jednu velkou nevýhodu. Inkluze se dá poměrně

Tabulka 4.14: **Relace simulace – doplňující měření:** Srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	1220648	2624316	1764220	###
Minimální:	1167000	2003224	1676112	###
Průměrná:	1201496	2418748	1733732	###
Medián:	1210524	2564254	1747850	###

dobře testovat pro jakoukoli dvojici automatů se stejnými abecedami, a tudíž nebyl problém s univerzální testovací sadou algoritmů, jelikož všechny stejnou abecedu měly. Pro důkladné testování univerzality bylo zjištěno, že je potřeba, aby automat měl většinu stavů koncových – tedy aby algoritmus chvíli běžel, než rozhodne o výsledku. Pokud toto není pravda, algoritmus v drtivé většině případů rozhodne hned na svém začátku a okamžitě skončí. Není však běžné, aby náhodně získané automaty měly tolik koncových stavů, což se projevilo ve výsledcích, kdy naprostá většina výpočtů trvala stejně dlouho. Uvedeny jsou pouze statistiky testu univerzality s relací identity, protože oproti klasickému testu univerzality proběhly téměř všechny výpočty (194) a výsledky jsou stejně v obou případech shodné.

Tabulka 4.15 ukazuje, že C# je pro lehké výpočty opravdu extrémně pomalý, za průměrně nejrychlejším C++ zaostal 40,14krát, OCaml se již klasicky C++ poměrně těsně drží (1,35krát pomalejší) a Python splnil svůj pomalý standard (9,16krát pomalejší), ačkoliv byl stále o hodně rychlejší než C#.

Tabulka 4.15: **Test univerzality:** srovnání jazyků podle procesorových časů v milisekundách.

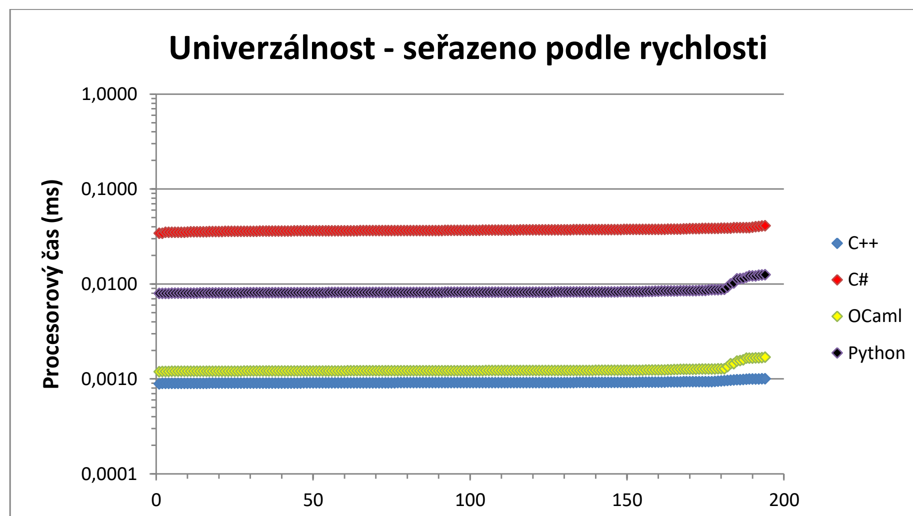
Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	0,001007	0,041132	0,001702	0,012553
Minimální čas:	0,000890	0,034297	0,001196	0,007949
Průměrný čas:	0,000920	0,036931	0,001250	0,008430
Medián:	0,000914	0,036750	0,001223	0,008189

Tabulka 4.16: **Test univerzality – doplňující měření:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	0,000921	0,039167	0,001291	0,008464
Minimální čas:	0,000867	0,036203	0,001202	0,007925
Průměrný čas:	0,000888	0,037659	0,001230	0,008188
Medián:	0,000885	0,037418	0,001213	0,008208

Z grafu seřazených výsledků podle rychlosti 4.17 toho tentokrát moc vyčíst nejde, odklon z jinak precizně rovných přímek demonstroval pouze OCaml a Python.

Graf závislosti rychlosti na počtu stavů není v tomto případě třeba ukazovat, z principu algoritmu se ale dá usuzovat, že ve všech jazycích bude záležet hlavně na poměru koncových a nekoncových stavů. Pokud bude hodně stavů koncových, bude pravděpodobně výpočet trvat déle. Poslední je uvedena tabulka spotřeby paměti 4.17. Nejšetrnější bylo C++, dále OCaml (1,38krát víc), Python (2,98krát víc) a C# (8,26krát víc). Všechna tato čísla jsou



Obrázek 4.17: **Test univerzality: srovnání jazyků s logaritmickou stupnicí osy y.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.

pravděpodobně blízko minimální spotřebě paměti pro načtení automatu a provádění nebo interpretování programu v daném jazyce.

Tabulka 4.17: **Test univerzality:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	5320	37444	6508	12844
Minimální:	3644	33096	5448	12392
Průměrná:	4240	35029	5888	12635
Medián:	4124	35622	5822	12636

Tabulka 4.18: **Test univerzality – doplňující měření:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	9516	44068	8856	15064
Minimální:	9136	43108	8588	14796
Průměrná:	9333	43817	8752	14945
Medián:	9334	43966	8784	14936

4.2.8 Test inkluze

Test inkluze je poměrně těžko předvídatelný algoritmus, jelikož je téměř nemožné při pohledu na automat odhadnout, jak dlouho bude výpočet trvat, nebo která část bude trvat nejdéle. Rozebrány jsou zde dva rozdílné průchody algoritmu. V jednom je relace simulace klasická, ve druhém je místo relace simulace použita relace identity (vedoucí na antichain verzi). Klasická množina simulačního předuspořádání obecně vytváří a prochází méně makrostavů, a je tedy pro komplikované případy efektivnější, na druhou stranu je

daleko větší než pouhá relace identity, a tak pro extrémně nenáročné výpočty může být naopak méně efektivní, jelikož je třeba v ní hodně vyhledávat. Pro práci s automaty je ale daleko přínosnější. Zde jsou uvedeny výsledky obou verzí.

Nejtěžší (neměřenou) částí tohoto algoritmu je výpočet relace simulace ze sjednocení dvou vstupních automatů. Sjednocením se totiž výrazně zvýší počet stavů, a tak měl někdy příslušný algoritmus problémy výpočet stihnout. Následná měřená část algoritmu už je velmi rychlá. Proto ve všech jazycích kromě Pythonu úspěšně proběhlo 193 měření, v případě Pythonu pak jen 146. V následujících tabulkách a grafech jsou tedy statistiky pro Python vypočítávány pouze ze 146 měření. Z tabulek 4.19 a 4.20 lze vypozorovat, že průměrně nejrychlejší bylo pro klasickou inkluzi C++, druhý byl OCaml (2,07krát pomalejší), dále C# (3,02krát pomalejší) a Python (6,71krát pomalejší). Pro inkluzi s relací identity bylo pořadí: C++, OCaml (1,90krát pomalejší), C# (1,93krát pomalejší), Python (3,24krát pomalejší). Velmi rychlý se ukázal být C#, který byl průměrně pomalejší než OCaml, ale v maximálním čase se mu vyrovnal a ve druhém případě byl dokonce rychlejší. Pythonovská implementace tohoto algoritmu výrazně trpěla nemožností upravovat množinu, přes kterou je souběžně iterováno, což výrazně snížilo efektivitu operace Minimize. Žádný z ostatních jazyků podobný problém neměl. Je nutno mít na paměti, že Python oproti ostatním jazykům nezvládl 47 nejtěžších výpočtů.

Tabulka 4.19: **Test inkluze:** srovnání jazyků podle procesorových časů v milisekundách. Algoritmus inkluze s relací simulace.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	6,209110	11,860465	9,071632	15,162524
Minimální čas:	0,002625	0,078027	0,003779	0,025496
Průměrný čas:	0,364016	1,102051	0,754004	2,445789
Medián:	0,156941	0,525210	0,264814	1,512605

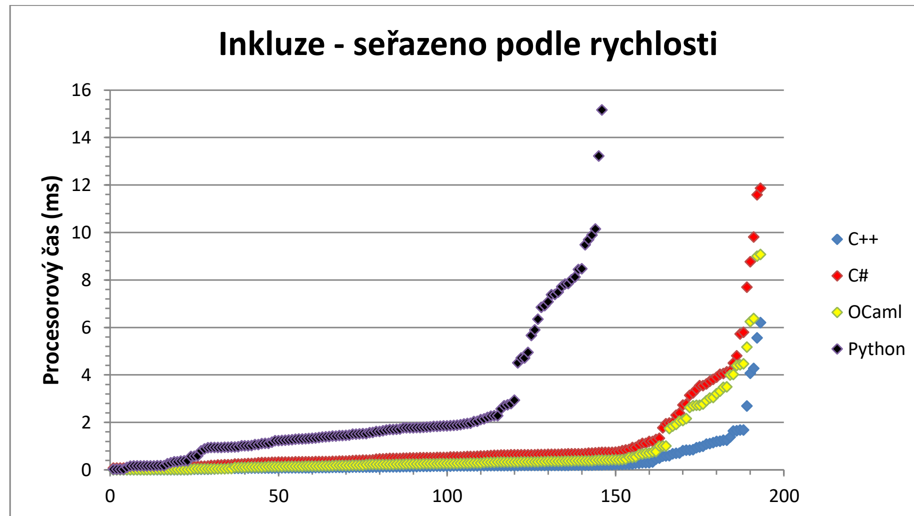
Tabulka 4.20: **Test inkluze:** srovnání jazyků podle procesorových časů v milisekundách. Algoritmus inkluze s relací identity.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	881,081500	1665,000000	1833,109500	2729,502265
Minimální čas:	0,005761	0,101174	0,009672	0,103589
Průměrný čas:	20,231086	39,226270	38,636880	65,608387
Medián:	0,271126	1,050420	0,473360	2,048815

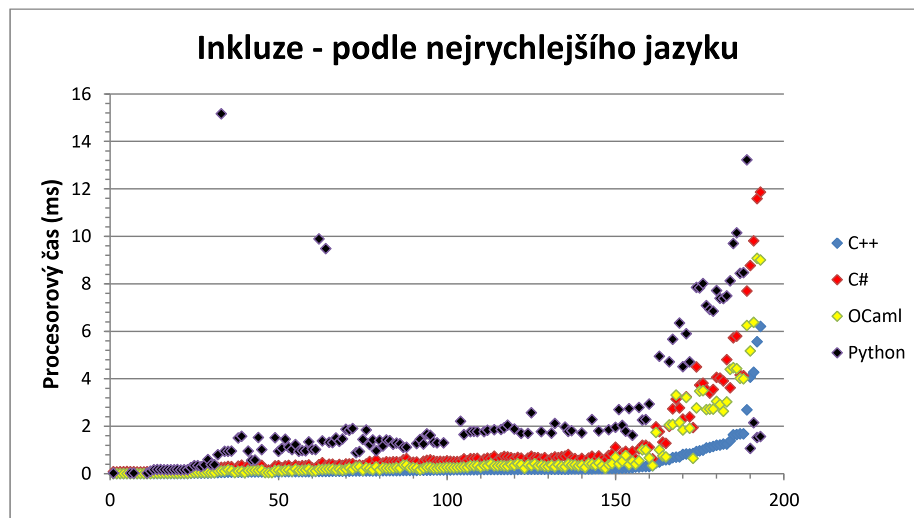
Tabulka 4.21: **Test inkluze – doplňující měření:** srovnání jazyků podle procesorových časů v milisekundách. Algoritmus inkluze s relací identity.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	17188,404500	31070,000000	36250,845000	49931,456212
Minimální čas:	382,986000	280,000000	29,871588	411,760129
Průměrný čas:	7881,460870	13924,444433	15937,430850	22320,157849
Medián:	1261,780000	1289,999900	1562,948000	1838,078872

Grafy 4.18, 4.19, 4.20 a 4.21 potvrzují naprostou neefektivitu Pythonu díky svým interním limitacím. Průběhy ostatních jazyků jsou si velmi podobné.

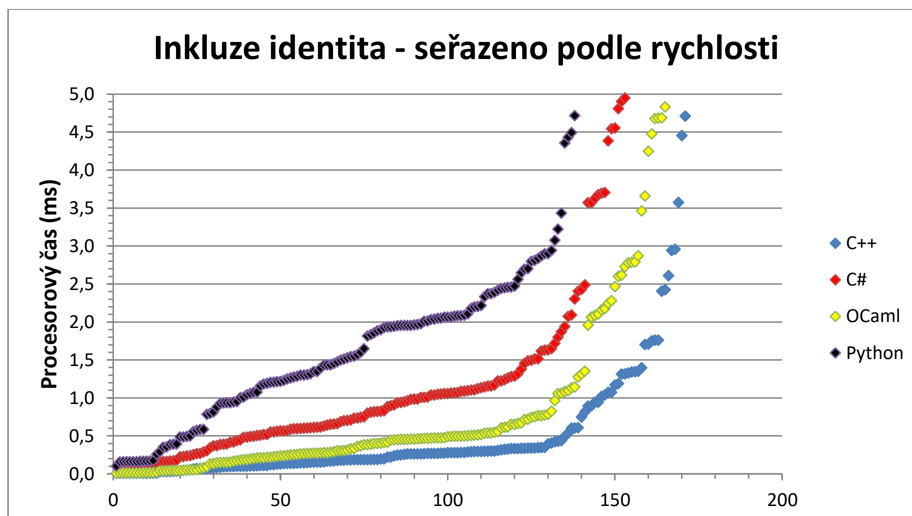


Obrázek 4.18: **Test inkluze: srovnání jazyků pro test inkluze s relací simulace.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.

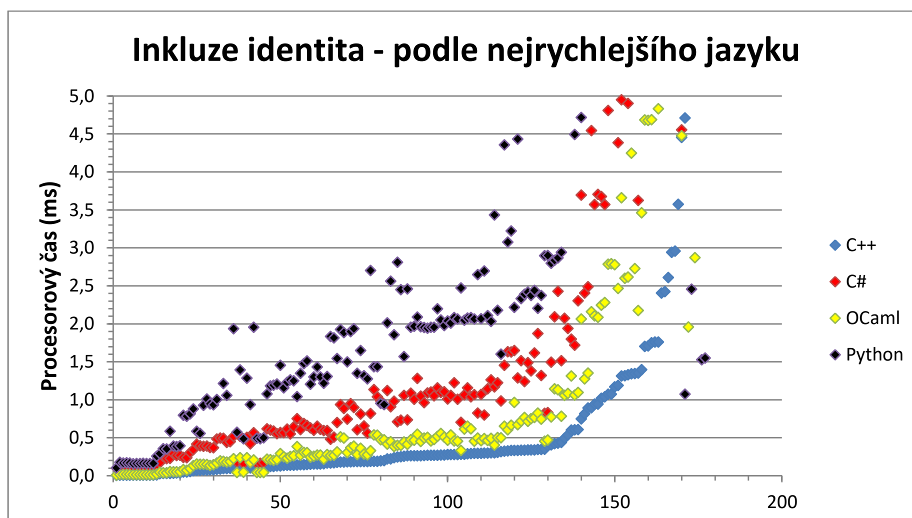


Obrázek 4.19: **Test inkluze: srovnání jazyků pro test inkluze s relací simulace.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

Neexistuje zde prakticky žádná závislosti rychlosti na počtu stavů, v žádném jazyce se z počtu stavů dvou automatů nedá usuzovat, jak náročný bude výpočet jejich inkluze. Nakonec jsou uvedeny tabulky paměťové náročnosti 4.22 a 4.23. V klasické inkluzi nejméně paměti potřebovalo C++, dále překvapivě C# (1,25krát víc), téměř stejně Python (1,25krát víc) a nejnáročnější byl výjimečně jindy velmi šetrný OCaml (1,51krát víc). V inkluzi s relací identity nejméně paměti potřebovalo taktéž C++, dále OCaml (1,22krát víc), Python (2,43krát víc) a C# (6,91krát víc). Rozdíl mezi těmito dvěma verzemi je v tom, že v první verzi musel před výpočtem inkluze proběhnout výpočet relace simulace, který je daleko náročnější než výpočet relace identity. To vysvětluje velké rozdíly v paměťové náročnosti. Dále je zajímavé, že C# byl při velkém množství potřebné paměti druhý nejšetrnější, ačkoli



Obrázek 4.20: **Test inkluze: srovnání jazyků pro test inkluze s relací identity.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.



Obrázek 4.21: **Test inkluze: srovnání jazyků pro test inkluze s relací identity.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

jindy nejvíce “plýtvá”. Ve všech algoritmech i při primitivních výpočtech potřeboval alespoň 30MB paměti, budeme-li tedy předpokládat, že toto množství je přibližně potřeba pro běh všech programů důležitých pro provádění kódu (např. JIT compiler a další), vyplyne ze srovnání obou verzí, že C# byl v tomto případě vlastně velmi nenáročný a přesto hodně rychlý.

Tabulka 4.22: **Test inkluze:** srovnání jazyků podle spotřeby paměti v kB. Algoritmus inkluze s relací simulace.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	464804	460316	653256	253988
Minimální:	3884	33352	5680	12592
Průměrná:	93432	117424	141957	117640
Medián:	78080	99272	120196	120672

Tabulka 4.23: **Test inkluze:** srovnání jazyků podle spotřeby paměti v kB. Algoritmus inkluze s relací identity.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	7992	38900	9728	15368
Minimální:	3720	33124	5648	12296
Průměrná:	5225	36108	6412	12703
Medián:	5064	36200	6280	12636

Tabulka 4.24: **Test inkluze – doplňující měření:** srovnání jazyků podle spotřeby paměti v kB. Algoritmus inkluze s relací identity.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	22188	47348	18420	30228
Minimální:	21312	46680	11812	20840
Průměrná:	21779	47089	14869	25333
Medián:	21868	47156	13808	22080

4.2.9 Posloupnost operací

Jako bonus ke standardním operacím a finální komplexní test jazyků bylo přidáno měření posloupnosti několika operací za sebou. Na vstupu byly tři automaty A1, A2 a A3. Vzorec pro posloupnost operací byl (doplněk (determinizace ((A1 sjednocení A2) průnik A3))). Celkem v každém jazyce proběhlo 192 úspěšných měření. Nejrychlejší zde bylo C++, druhý byl OCaml (2,47krát pomalejší), třetí C# (3,51krát pomalejší) a čtvrtý Python (9,52krát pomalejší).

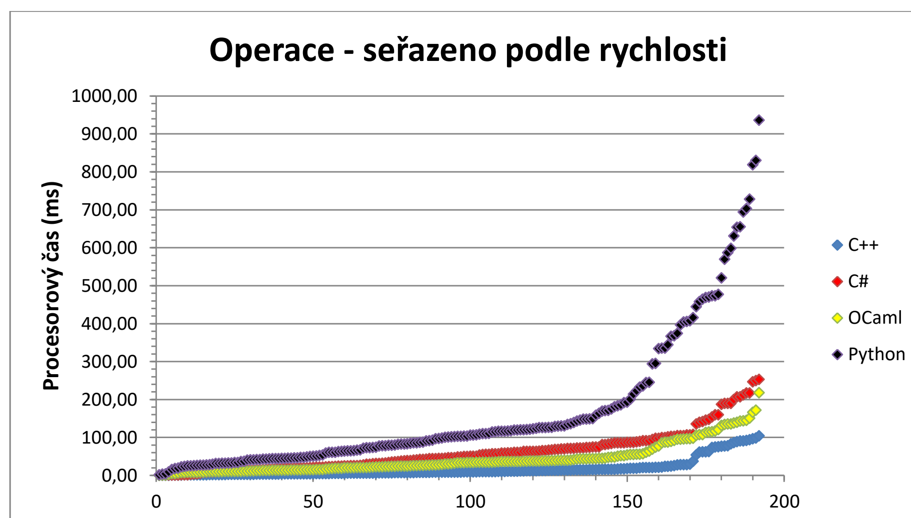
Tabulka 4.25: **Posloupnost operací:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	104,581500	253,333333	217,740250	936,034743
Minimální čas:	0,048717	1,000000	0,368918	1,439358
Průměrný čas:	17,429827	61,336761	43,096801	165,981516
Medián:	8,930376	49,090909	33,374679	103,121220

Seřadíme-li si výsledky podle rychlosti 4.22, vidíme, že výrazněji vybočuje pouze Python, který zaostává. OCaml a C# jsou si výkonnostně podobné s přihlédnutím k faktu, že C# je vždy pomalejší při lehčích výpočtech, což nijak moc nevadí a ovlivňuje to celkový průměr. C++ je, jako ve většině případů, stabilně nejrychlejší.

Tabulka 4.26: **Posloupnost operací – doplňující měření:** srovnání jazyků podle procesorových časů v milisekundách.

Časy (ms)	**C++**	**C#**	**OCaml**	**Python**
Maximální čas:	760,737000	1520,000000	790,523000	3861,602234
Minimální čas:	649,095500	1340,000000	724,842000	3676,224886
Průměrný čas:	672,019500	1428,750000	748,197625	3779,302832
Medián:	665,435000	1430,000000	747,711500	3791,220387



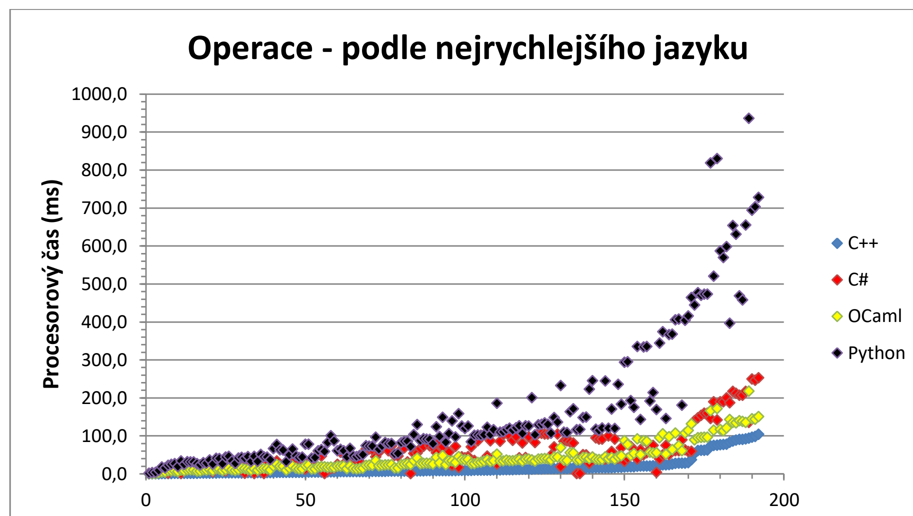
Obrázek 4.22: **Posloupnost operací: srovnání jazyků.** Výsledky každého jazyka jsou seřazeny zvlášť. Body se stejnou x-hodnotou nemusí odpovídat stejnému automatu.

Při seřazení výsledků podle nejrychlejšího jazyka 4.23 ukazuje Python opět největší rozdíly, v tomto případě oproti C++. Naopak výsledky OCamlu jsou těm z C++ výrazně podobné. V každém jazyce pravděpodobně nejvíce záleží na tom, jak velký je průnik A3 a výsledku sjednocení. To výrazně ovlivní dobu výpočtu.

Co se týče paměti, nejefektivnějším jazykem bylo těsně C++, za ním OCaml (1,49krát víc), Python (2,85krát víc) a C# (7,45krát víc).

Tabulka 4.27: **Posloupnost operací:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	14628	166252	15148	36392
Minimální:	3892	34396	6016	12460
Průměrná:	6171	46026	9238	17642
Medián:	5502	42478	9646	15046



Obrázek 4.23: **Posloupnost operací: srovnání jazyků.** Výsledky jsou seřazeny podle nejrychlejšího jazyka. Body se stejnou x-hodnotou odpovídají stejnému automatu pro všechny čtyři jazyky.

Tabulka 4.28: **Posloupnost operací – doplňující měření:** srovnání jazyků podle spotřeby paměti v kB.

Paměť (kB)	**C++**	**C#**	**OCaml**	**Python**
Maximální:	62436	583280	31596	102880
Minimální:	61548	256188	31272	101560
Průměrná:	62048	390888	31453	102437
Medián:	62072	386814	31458	102558

4.3 Zhodnocení jazyků

Druhým kritériem hodnocení jazyků v této práci bylo to, jak příjemně, efektivně a rychle se v nich algoritmy zapisují. Za tímto účelem byly vybrány dva objektivní parametry, a to velikost komprimovaného zdrojového kódu (metodou zip) a počet řádků. Dále byly jazyky subjektivně zhodnoceny z pohledu autora této práce. Na závěr pak proběhlo shrnutí výsledků měření. Následující text porovnává jazyky na základě autorových zkušeností, nabytých při implementování algoritmů pro tuto práci. Věnuje se tedy výhradně popisu pozitiv či nedostatků jazyků, se kterými se setkal při tvorbě této práce.

Tabulka 4.29: Výsledky velikosti a počtu řádků zdrojového kódu.

Jazyk	**Velikost v kB**	**Počet řádků**
C++	15,7	2189
C#	16,7	2173
OCaml	12,7	1742
Python	12,1	1383

Výsledky v tabulce 4.29 mohou vytvořit určitou představu o tom, jak náročné a dlouhé je psaní programů v jednotlivých jazycích, ale nesmí se brát příliš vážně. Například OCaml měl druhý nejmenší počet řádků a velikost zdrojového souboru, avšak jelikož se jedná

o funkcionální jazyk, je tento výsledek předpokládatelný. Funkcionální jazyky totiž mívají kratší zápisy algoritmů, ale náročnost programování se pro člověka zvyklého na imperativní jazyky mnohonásobně zvyšuje. Dále tyto statistiky také výrazně ovlivní to, jak moc se v jazyce využívají závorky. To se projevilo v případě Pythonu, který závorky téměř nepoužívá a navíc má spoustu způsobů a konstrukcí, jak úkon napsat na jeden řádek. Sice se tak může zdát, že programy se v Pythonu píší "samy", což je částečně pravda, ale pro spoustu lidí včetně autora tohoto textu může být absence závorek naopak dezorientující, obzvláště při velkém zanoření a dlouhých funkcích. C# a C++ jsou jazyky syntakticky velmi podobné, což se projevilo i v uvedených výsledcích. C# však po syntaktické stránce působil jako lepší verze C++, řešení většiny problémů bylo kompaktnější s menším počtem kroků. Příkladem budiž třeba vyhledání prvků v datových strukturách, sjednocení názvů metod datových struktur a samozřejmě možnost programování bez manuální správy ukazatelů.

Ze subjektivního pohledu autora této práce vyšel z pohledu programátorské přívětivosti nejlépe jazyk C#. Výbornou vlastností tohoto jazyka je, že není příliš limitující, čímž se dost podobá C++. Nemá snahu lidem vnucovat určitý styl programování, jako to dělá například Python, kde se programátor chtě nechtě musí podřídit představě jeho tvůrců o jediném správném způsobu. Jazyk umožňuje moderní vysokoúrovňové programování s prvky jako automatická správa paměti, výrazná objektová orientace, listové generátory a další, zároveň však nabízí možnost práce s ukazateli, jednoduchou možnost úpravy datové struktury v průběhu iterace, struktury ve stylu C++, zkrátka více možností. Další velká výhoda C# je fakt, že je staticky typovaný. Autorův názor je takový, že neexistuje žádné pozitivum dynamického typování, naopak je původcem mnoha chyb a popírá základní programovací návyky – každá proměnná má jeden účel, který se nemění a musí být známý již při její tvorbě. Nemožnost určit datový typ proměnných byla při práci s Pythonem a částečně i s OCamlem velmi obtěžující. Obzvláště OCaml občas projevoval velkou neochotu správně určit datový typ argumentů funkcí, naštěstí však umožňuje jeho manuální vynucení. Mezi nevýhody C# patří hlavně až nepochopitelné problémy s měřením procesorového a celkového času běhu funkce (platí pro .NET core). Problém byl popsán v kapitole 4.1. Představa Microsoftu, že měření s vzorkovací frekvencí 16 milisekund je přesné, je přinejmenším zarážející. Tento problém přetrvává jak na operačním systému Windows, tak na unixových OS a způsobil při tvorbě práce mnoho komplikací. Podivuhodné je, že všechny ostatní jazyky byly schopny měření s přesností přibližně na nanosekundy.

Pokud někomu nevadí absence závorek a statického typování v Pythonu, může být tento jazyk také velmi příjemnou volbou. Algoritmy v něm napsané jsou krátké a kompaktní, programy s nízkým počtem řádků a rychle napsané. Práci velmi ulehčuje i schopnost automatického výpisu datových struktur bez nutnosti pro tento účel vytvářet speciální funkce nebo metody. Jazyk je z pohledu sémantiky velmi podobný C#. Programy lze tedy velice snadno převádět z jednoho jazyka do druhého. Poněkud nepříjemná je však malá velikost standardní knihovny. Tvůrci Pythonu, zdá se, předpokládají, že v případě potřeby například větší rozmanitosti datových struktur (seřazená množina prvků, lineárně zřetězený seznam, ...) je postačující, když si programátor doinstaluje potřebné balíčky (nebo je pracně naimplementuje sám). V takovém případě ovšem vyvstává otázka, jestli bude takový program spustitelný i na jiné instalaci Pythonu, například na nějakém školním serveru. I z tohoto důvodu nebyly v implementaci použity žádné nestandardní prvky. Škoda pro Python, avšak pravda je taková, že je možné si velmi dobře vystačit i bez toho.

Pokud se zaměříme čistě na pohodlnost programování, není jediný důvod si vybrat C++. Většina výhod zmíněných u C# nebo Pythonu se dá použít zde jako argument proti C++. Speciální pozornost autorovy kritiky si žádá jeho překladač, konkrétně chybové

výpisy, kdy jeden malý překlep dokáže vygenerovat nekončící stěnu textu a je výzvou pro nejotrlejší, aby sdělení dešifrovali. Dá se říci, že C++ má velmi Erbenovský charakter, neboť za jakoukoli chybu následuje neúměrně vysoký trest. Na druhou stranu se tento jazyk ukázal být poměrně odolný vůči hloupým a neefektivním řešením, obzvláště se zapnutými optimalizacemi, což lze považovat za výhodu.

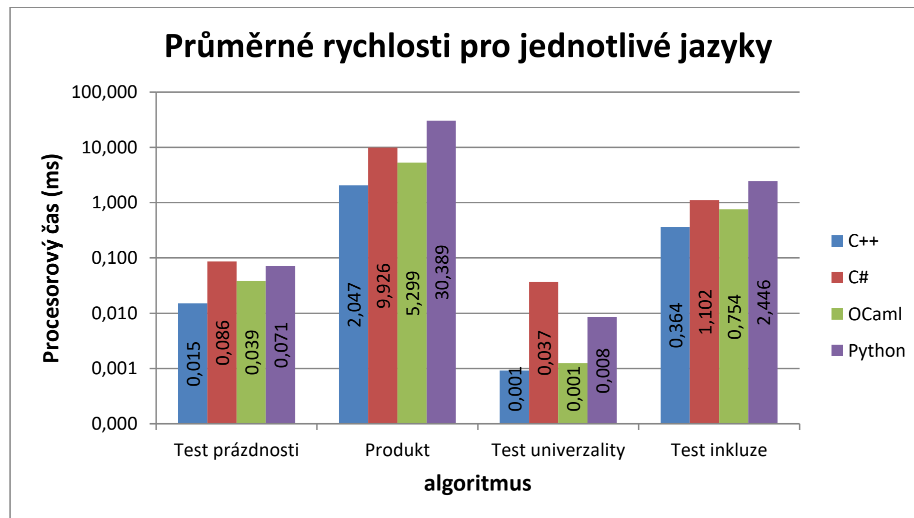
K OCamlu lze říci jen málo, neboť je to jazyk jiného paradigma než ostatní. Pokud má tedy čtenář v oblibě funkcionální jazyky nebo je vášnivým milovníkem rekurze, je OCaml jistě nejlepší volbou. Pokud ne, ostatní tři možnosti budou pravděpodobně lákavější. Mezi největší problémy, které byly zaznamenány, patří občasná neschopnost OCamlu správně určit datové typy argumentů funkcí a extrémně stručné chybové výpisy překladače, tedy pravý opak C++ s přibližně stejnou sdělovací schopností. Jako velice pozitivní je považován fakt, že v tomto jazyce lze kombinovat funkcionální a procedurální programování. Je tedy možné využít maximální potenciál obou paradigmat k nalezení nejpohodlnějšího řešení. V této práci se sice autor snažil používat téměř výhradně prvky funkcionálního programování, je však přesvědčen, že se v kombinování těchto dvou přístupů může skrývat velký potenciál. Obzvláště přínosnými jsou například metody filter, fold a iter pro práci s Listem. Princip, kdy jsou všechny konstrukce zároveň výrazem, je také velmi zajímavý.

Pokud se budeme dívat pouze na rychlost, nejlepším jazykem bylo suverénně C++. Z devíti různých algoritmů bylo osmkrát první ve všech kategoriích (maximální, minimální i průměrný procesorový čas). V drtivé většině případů projevovalo nejmenší nárůst trvání výpočtu ze všech jazyků a zaznamenalo také nejmenší výkyvy, například oproti Pythonu. Ze všech jazyků také zvládlo nejvíce výpočtů. Konkrétně při výpočtu minimalizace, která byla implementována poněkud neefektivně, zvládlo všech 200 měření včetně největších automatů o 3700 stavech, k čemuž se žádný jiný jazyk ani nepřiblížil. To vše při nízkých paměťových nárocích. Výjimkou byl pouze algoritmus odstranění zbytečných stavů, kde se ukázala nevýhoda C++ oproti ostatním jazykům, kterou je neexistence rozlišení na hodnotové a referenční datové typy a s tím spojená existence garbage collectoru, jako je tomu v ostatních třech jazycích. Bylo tedy nutné použít ukazatele, což vedlo k velmi obtížnému zajišťování referenční integrity dat ve strukturách při odstraňování stavů, které byly uchovávány ve vektoru. Následkem toho se v tomto případě C++ umístilo až na druhém místě za OCamlem.

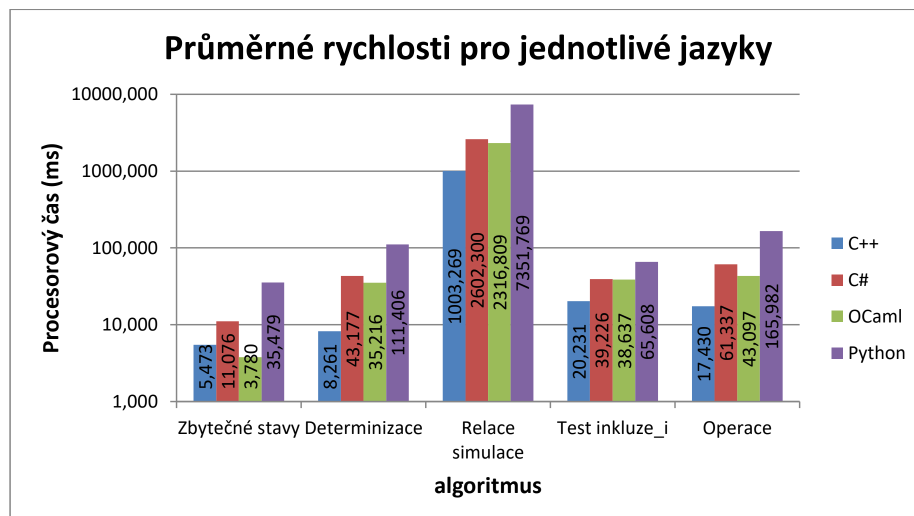
Další přijatelnou možností může být za určitých okolností OCaml, například při preferenci funkcionálního jazyku. Celkově se tento jazyk umístil výrazně za C++, při některých výpočtech se C++ vyrovnal a výjimečně ho i předčil, ovšem postrádal konzistentnost výsledků. Při některých algoritmech, např. testu prázdnosti, determinizaci a minimalizaci, výrazně zaostal. Bylo by tedy nutné použít nejefektivnější možné algoritmy a výrazně implementaci optimalizovat, aby měl jazyk šanci se přiblížit C++. Mezi jeho přednosti však patří nízká paměťová náročnost.

Jazyk C# se svou rychlostí často blížil OCamlu, většinou byl ale pomalejší. Po celou dobu poměrně stabilně prokazoval vzrůstající výkonnost se zvyšující se obtížností výpočtů, což je při práci s automaty velmi pozitivní fakt. Jako zanedbatelná se jeví jeho tristní rychlost při lehkých výpočtech. Znepokojující je ovšem jeho extrémní paměťová náročnost. Tento jev může být způsoben optimalizacemi garbage collectoru, pokud však není, činí to ze C# jazyk pro práci s automaty naprosto nepoužitelný. Stejně jako pro OCaml tedy platí, že pro použití na velmi velké automaty by bylo třeba použít nejefektivnější algoritmy a věnovat spoustu času optimalizaci, aby existovala šance se C++ přiblížit.

Python ze všech měření vyšel jako nejhorší s jedinou výjimkou, a to algoritmus minimalizace. Jinak se jeví jako nepoužitelný pro jakékoli serióznější výpočty spojené s velkými automaty. C++ nebo poněkud bližší C# by měly být daleko lepší a jistější volby.



Obrázek 4.24: **Souhrnné srovnání průměrných procesorových časů všech jazyků.** Zobrazeny jsou čtyři algoritmy s nejmenšími hodnotami.



Obrázek 4.25: **Souhrnné srovnání průměrných procesorových časů všech jazyků.** Zobrazených je dalších pět algoritmů.

Obrázky 4.24 a 4.25 shrnují průměrné rychlosti pro každý jazyk ve všech testech. Celkově bylo nejrychlejší C++, druhý OCaml byl průměrně 2,25krát pomalejší, třetí C# 7,77krát pomalejší a čtvrtý Python 8,72krát pomalejší než C++. Tato čísla ale spíše matou, protože průměrují výsledky jazyků se stejnou váhou u každého algoritmu. Fungují relativně dobře u OCamlu a Pythonu, ale C# velmi špatně vystihují díky jeho velmi slabým výkonům při krátkých a jednoduchých výpočtech. Například při testu univerzality, který se vyznačoval extrémně rychlými výpočty, byl C# 40krát pomalejší než C++. S průměrem 0,037

milisekund. Pokud bychom nebrali v úvahu tyto krátké výpočty, odhadem by C# byl přibližně 3–4krát pomalejší než C++.

Z těchto údajů lze konstatovat, že pokud je cílem práce s velkými automaty a komplikovanými algoritmy, například při tvorbě knihoven pro práci s automaty nebo zátěžových testech algoritmů, je C++ pravděpodobně jedinou smysluplnou volbou pro imperativní jazyky. Při preferenci funkcionálního jazyku se OCaml jeví jako přijatelně efektivní, hlavně díky jeho nízké paměťové spotřebě. Při použití velmi efektivních algoritmů, omezení jeho paměťové náročnosti a důkladné optimalizaci by se dal použít i C#, ale ten se díky své modernosti a vysokoúrovňovému zaměření spíše hodí pro pohodlnou a rychlou tvorbu prototypů implementace algoritmů při jejich vývoji nebo testování, kdy se nepředpokládá jeho použití na obrovských automatech. Pro seriózní práci s velmi velkými automaty existují lepší volby. Pro Python toto platí několikanásobně, neboť se tento jazyk vůbec nehodí pro jakékoli výpočty, kde jsou vysoké požadavky na rychlost nebo paměťovou šetrnost.

Kapitola 5

Závěr

Cílem této práce bylo experimentálně zjistit a porovnat efektivitu programovacích jazyků C++, C#, OCaml a Python při práci s automatovými algoritmy. Tento cíl byl splněn, stejně jako se podařilo splnit všechny tři body zadání této práce. Na základě prvního bodu byla nastudována literatura k automatovým algoritmům, výstupem čehož je kapitola 3.1, kde byly nabyté informace přehledně shrnuty. Následně byly v souladu s druhým bodem navrženy datové struktury a algoritmy byly implementovány v zadaných jazycích. Navíc byl implementován jeden algoritmus a vybrán jeden jazyk nad rámec minimálního znění zadání. Pro splnění třetího bodu byl vybrán testovací soubor automatů, byla provedena měření a na závěr autor nabídl své zhodnocení, interpretaci výsledků a prezentoval srovnání jednotlivých jazyků.

V práci se podařilo ukázat, že C++ je dominantním jazykem pro práci s automaty, a potvrdilo se, že se jedná o rychlý a efektivní jazyk i v tomto kontextu. Dále byl navržen OCaml jako snesitelně rychlá alternativa pro funkcionální programování. C# a Python se příliš nehodí pro práci s automaty, kde je kritická rychlost nebo šetrné zacházení s pamětí, avšak díky jejich příjemným vlastnostem byla zdůrazněna vhodnost těchto jazyků pro případy, kdy rychlost není zásadní a je třeba hlavně přehlednost, možnost snadné modifikace zdrojových kódů a rychlost implementace.

Práce mi jako studentovi dala příležitost si výrazně rozšířit své znalosti o zajímavé problematice konečných automatů a s nimi spojených netriviálních operacích. Dále jsem se mohl naučit čtyři populární programovacími jazyky a vyzkoušet si programování v různých paradigmatech, což považuji za velmi přínosné vzhledem k tomu, že jsem za celé studium poznal téměř výhradně jazyk C. To mi umožnilo lépe pochopit rozdíly v principech jazyků a následně si na ně vytvořit názor i v praxi. V neposlední řadě jsem se seznámil s problematikou efektivního programování a automatizovaného měření výkonnosti programů.

Práci je samozřejmě dále možné rozšířit o další algoritmy nebo programovací jazyky tak, aby bylo srovnání co nejobsáhlejší. Dále spatřuji možné zdokonalení ve výběru testovacích automatů, kdy by bylo ideální použít jinou sadu automatů ušitou každému algoritmu na míru, a zajistit tak rovnoměrné a předvídatelné rozložení obtížností výpočtů pro každý algoritmus. Přesnosti srovnání by jistě přispělo i zvýšení počtu a maximální velikosti testovacích automatů, což bohužel nebylo možné z časových důvodů – měření by trvalo velmi dlouho. Bylo by však například zajímavé zjistit, zdali se výkon C# bude nadále zlepšovat i pro větší automaty. A konečně, jednotlivé jazyky nabízejí více různých způsobů implementace, například řešení pomocí indexů v C++ nebo imperativní řešení v OCamlu.

Literatura

[1] *CpPreference.com* [online]. [cit. 2020-04-09]. Dokumentace k C++. Dostupné z: https://en.cppreference.com/w/.

[2] Abdulla, P. A., Chen, Y.-F., Holík, L., Mayr, R. a Vojnar, T. When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In: Esparza, J. a Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* 1. vyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 158–174. Lecture Notes in Computer Science, č. 6015. ISBN 978-3-642-12002-2. TACAS 2010. Dostupné z: https://doi.org/10.1007/978-3-642-12002-2_14.

[3] Aruoba, S. B. a Fernández Villaverde, J. *A Comparison of Programming Languages in Economics.* Working Paper 20263. National Bureau of Economic Research, June 2014. Dostupné z: http://www.nber.org/papers/w20263.

[4] Champarnaud, J.-M. a Coulon, F. NFA Reduction Algorithms by Means of Regular Inequalities. In: Ésik, Z. a Fülöp, Z., ed. *Developments in Language Theory.* Berlin, Heidelberg: Springer Berlin Heidelberg, červenec 2003, sv. 2710, s. 194–205. Lecture Notes in Computer Science. ISBN 978-3-540-45007-8. Dostupné z: https://doi.org/10.1007/3-540-45007-6_15.

[5] De Wulf, M., Doyen, L., Henzinger, T. A. a Raskin, J. F. Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T. a Jones, R. B., ed. *Computer Aided Verification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, sv. 4144, s. 17–30. Lecture Notes in Computer Science. ISBN 978-3-540-37411-4.

[6] Esparza, J. *Automata Theory: An Algorithmic Approach* [online]. Mnichov: Institut für Informatik, Technische Universität München, srpen 2017, revidováno 26.8.2017 [cit. 2019-12-20]. 321 s. Dostupné z: https://www7.in.tum.de/~esparza/automatanotes.html.

[7] Fourment, M. a Gillings, M. R. A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics.* Springer, Springer Nature. Únor 2008, sv. 9, s. 1–9. DOI: 10.1186/1471-2105-9-82. ISSN 1471-2105. Dostupné z: https://doi.org/10.1186/1471-2105-9-82.

[8] Holík, L., Lengál, O., Síč, J., Veanes, M. a Vojnar, T. Simulation Algorithms for Symbolic Automata. In: Lahiri, S. K. a Wang, C., ed. *Proc. of 16th International Symposium on Automated Technology for Verification and Analysis.* Heidelberg, DE: Springer International Publishing, 2018, s. 109–125. Lecture Notes

in Computer Science, č. 11138. ISBN 978-3-030-01090-4. Dostupné z: http://dx.doi.org/10.1007/978-3-030-01090-4_7.

[9] Ilie, L., Navarro, G. a Yu, S. On NFA Reductions. In: Karhumäki, J., Maurer, H., Păun, G. a Rozenberg, G., ed. *Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday.* 1. vyd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 112–124. Lecture Notes in Computer Science, č. 3113. ISBN 978-3-540-27812-2. Dostupné z: https://doi.org/10.1007/978-3-540-27812-2_11.

[10] INRIA. Comparison of Standard Containers. *OCaml Website* [online]. [cit. 2020-04-09]. Dostupné z: https://ocaml.org/learn/tutorials/comparison_of_standard_containers.html.

[11] INRIA. What is OCaml. *OCaml Website* [online]. [cit. 2020-04-09]. Dostupné z: https://ocaml.org/learn/description.html.

[12] INRIA. A History of OCaml. *OCaml Website* [online]. [cit. 2020-04-09]. Dostupné z: https://ocaml.org/learn/history.html.

[13] Microsoft Corporation. *C# documentation* [online]. [cit. 2020-04-09]. Dostupné z: https://docs.microsoft.com/en-gb/dotnet/csharp/.

[14] Naveen Reddy, K. P., Geyavalli, Y., Sujani, D. a Rajesh, M. Comparison of Programming Languages: Review. In: Dionnie Fahour, J. et al., ed. *International Journal of Computing Science and Communication.* Bangalore, India: Department of Computer science and Engineering, Gandhi Institute of Technology and Management, červenec 2018, sv. 9, č. 2, s. 113–122. ISSN 0973-7391. Dostupné z: https://www.researchgate.net/publication/326672199_Comparison_of_Programming_Languages_Review.

[15] Python Software Foundation. *Python documentation* [online]. [cit. 2020-04-09]. Dostupné z: https://docs.python.org/3/.

[16] Stroustrup, B. A history of C++. In: *ACM SIGPLAN Notices.* Assn for Computing Machinery, Březen 1993, sv. 28, s. 271–297. DOI: 10.1145/154766.155375. ISBN 0897915704. Dostupné z: https://doi.org/10.1145/155360.155375.