ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

| | |
|---|---|
| **Název:** | Měření v automatové knihovně |
| **Student:** | Radovan Červený |
| **Vedoucí:** | Ing. Jan Trávníček |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Teoretická informatika |
| **Katedra:** | Katedra teoretické informatiky |
| **Platnost zadání:** | Do konce letního semestru 2016/17 |

## Pokyny pro vypracování

Nastudujte současnou architekturu a implementaci Automatové knihovny [1].
Navrhněte potřebné rozšíření knihovny pro měření času běhu a paměti spotřebované algoritmem.
Pro Automatovou knihovnu navrhněte možnosti automatizovaného měření spotřebovaného času a paměti různými algoritmy, které řeší stejnou úlohu.
Implementujte měření spotřebovaného času a paměti v Automatové knihovně.
Zvolte, nastudujte a v Automatové knihovně implementujte vhodné algoritmy řešící stejnou úlohu, na kterých provedete ukázková měření.
Tyto implementace otestujte a proměřte pomocí Vašeho rozšíření Automatové knihovny, diskutujte výsledky.

## Seznam odborné literatury

[1] Martin Žák: Automatová knihovna – vnitřní a komunikační formát. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 5. ledna 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

# Měření v Automatové knihovně

***Radovan Červený***

Vedoucí práce: Ing. Jan Trávníček

10. května 2016

# Poděkování

Především děkuji Ing. Janu Trávníčkovi za jeho neocenitelné rady a pozitivní energii, kterou vnášel do této práce. Dále bych rád poděkoval své rodině a přátelům za jejich podporu při mých studiích.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2016 .....

České vysoké učení technické v Praze
Fakulta informačních technologií


### Odkaz na tuto práci

Červený, Radovan. *Měření v Automatové knihovně*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

# Abstrakt

Předmětem této práce je rozšíření Automatové knihovny pro měření času běhu algoritmu, paměti spotřebované algoritmem, a obecných událostí pro profilování algoritmu. Dále jsou navrženy a implementovány aplikace pro automatizaci měření a základní zpracování výsledků měření. Automatová knihovna je rozšířena o tři nové implementace algoritmů pro řetězcové vyhledávání. Nakonec jsou prezentovány výsledky experimentálního měření nových i některých již existujících algoritmů Automatové knihovny.

**Klíčová slova** rozšíření Automatové knihovny, profilování algoritmů, automatizace měření, řetězcové vyhledávání, experimentální měření

# Abstract

This theses introduces new extension of the Automata library, which allows measuring running time of an algorithm, memory used by an algorithm, and general events for algorithm profiling. Next, applications for measurement automation and basic processing of measurement results are designed and implemented. Three new implementations of string matching algorithms are added to the Automata library. Finally, the results of an experimental measurement of the new algorithms as well as some already existing algorithms in the Automata library are presented.

**Keywords** Automata library extension, algorithm profiling, measurement automation, string matching, experimental measurements

# Obsah

**Úvod** **1**
Cíle práce . . . . . 1
Struktura práce . . . . . 2

**1 Automatová knihovna a existující řešení** **3**
1.1 Automatová knihovna . . . . . 3
1.2 SMART . . . . . 4

**2 Měření v Automatové knihovně** **7**
2.1 Způsob měření . . . . . 7
2.2 Výsledek měření . . . . . 13
2.3 Výstup měření . . . . . 13

**3 Automatizované měření** **17**
3.1 Způsob měření . . . . . 17
3.2 Definice pojmů . . . . . 18
3.3 Konfigurace automatizovaného měření . . . . . 19
3.4 Zpracování vstupních dat . . . . . 20
3.5 Měření programových rour . . . . . 23
3.6 Výsledek a výstup automatizovaného měření . . . . . 24

**4 Zpracování výsledků** **25**
4.1 Způsob zpracování . . . . . 25
4.2 Filtrování . . . . . 25
4.3 Volba datové domény a datového pohledu . . . . . 26
4.4 Agregace . . . . . 27
4.5 Formát výpisu . . . . . 27

**5 Algoritmy pro řetězcové vyhledávání** **29**
5.1 Definice problému a pojmů . . . . . 29

5.2 Principy řešení problému . . . . . 29
5.3 Sufixový automat . . . . . 30
5.4 Faktorové orákulum . . . . . 31
5.5 Backward DAWG Matching . . . . . 31
5.6 Backward Nondeterministic DAWG Matching . . . . . 32
5.7 Backward Oracle Matching . . . . . 33
5.8 Implementace v Automatové knihovně . . . . . 34

**6 Testování** **35**
6.1 Měření v Automatové knihovně . . . . . 35
6.2 Automatizované měření . . . . . 35
6.3 Algoritmy pro řetězcové vyhledávání . . . . . 35

**7 Experimentální měření** **37**
7.1 Datasety . . . . . 37
7.2 Metodika . . . . . 37
7.3 Měření . . . . . 38

**Závěr** **43**

**Literatura** **45**

**A Seznam použitých zkratek** **47**

**B Ukázky** **49**
B.1 Výsledek měření Automatové knihovny . . . . . 49
B.2 Konfigurační soubor automatizovaného měření . . . . . 50
B.3 Výsledek automatizovaného měření . . . . . 51

**C Výsledky měření** **53**
C.1 Automatová knihovna . . . . . 53
C.2 Nástroj SMART . . . . . 54

**D Uživatelská příručka** **55**
D.1 Požadavky . . . . . 55
D.2 Instalace . . . . . 55
D.3 Měření v Automatové knihovně . . . . . 55
D.4 Automatizované měření . . . . . 55
D.5 Zpracování výsledků měření . . . . . 56

**E Obsah přiloženého CD** **57**

# Seznam obrázků

5.1 Faktorový přístup vyhledávání . . . . . 30
5.2 Sufixový automat pro reverzi řetězce „*deed*“ . . . . . 31
5.3 Nedeterministický sufixový automat pro reverzi řetězce „*deed*“ . . 32
5.4 Faktorové orákulum pro reverzi řetězce „*deed*“ . . . . . 33

# Seznam grafů

7.1 Automatová knihovna vs. SMART – BF, BDM, BOM . . . . . 38
7.2 Poměr dob běhu fází algoritmu BF . . . . . 39
7.3 Paměť alokovaná při běhu jader algoritmů . . . . . 40

# Úvod

Pokud máme vybrat mezi dvěma algoritmy nebo mezi dvěma implementacemi jednoho algoritmu, největší váhu dáváme měřítkům výkonnosti jako jsou doba běhu algoritmu a paměťové nároky algoritmu. Následně se také můžeme ptát, kolik pro nás důležitých operací algoritmy provádějí (např.: kolikrát se porovnají dvě struktury).

Logickým krokem směrem k učinění rozhodnutí je uskutečnění experimentálních měření našich algoritmů a získání relevantních výsledků popisující jejich výkonnost. Tento experiment je třeba několikrát zopakovat, abychom omezili možnost chyby a minimalizovali rozptyl měření.

Záměrem této práce je dát uživateli možnosti měřit jednotlivé výkonnostní metriky, své experimenty lehce definovat, měření jednoduchým způsobem automatizovat a výsledky měření základním způsobem zpracovávat. To vše realizované v rámci Automatové knihovny.

## Cíle práce

Prvním cílem této práce je nastudování architektury Automatové knihovny, načež je navrženo a implementováno rozšíření Automatové knihovny pro měření algoritmem spotřebovaného času a paměti spolu s čítáním obecných událostí, které při běhu algoritmu nastanou.

Druhým cílem je navržení a implementace aplikací pro automatizaci měření a pro základní strojové zpracování výsledků měření.

Třetím cílem je obohatit Automatovou knihovnu o nové implementace algoritmů z oblasti textového vyhledávání, které poslouží pro testování výše implementovaných rozšíření. Zvětšený repertoár algoritmů textového vyhledávání je nutnou podmínkou pro úspěšné vykonání posledního cíle.

Posledním cílem je experimentální měření algoritmů Automatové knihovny a prezentace závěrů založených na výsledcích těchto měření. Dále je Automatová knihovna z hlediska výkonnosti srovnána s nástrojem SMART.

## Struktura práce

Kapitola 1 je věnována představení Automatové knihovny a nástroje SMART, popisuje jejich hlavní myšlenky a komentuje jejich vliv na tuto práci. Kapitola 2, 3 a 4 se zaobírají návrhem a realizací jednotlivých rozšíření Automatové knihovny. V kapitole 5 jsou rozebrány implementované algoritmy. V kapitole 6 je zběžně popsáno testování obou rozšíření a implementací algoritmů. Nakonec jsou v kapitole 7 prezentovány výsledky experimentálního měření Automatové knihovny spolu s výsledky srovnání Automatové knihovny s nástrojem SMART.

Jisté sekce jsou doplněny o nečíslované podsekce nazvané „Implementace“ nebo „Závěr“. V těchto podsekcích jsou vysvětlovány implementační záležitosti dané sekce či závěry z dané sekce vyvozené.

Kapitola 1

# Automatová knihovna a existující řešení

Existuje mnoho nástrojů a knihoven podobného zaměření jako má Automatová knihovna [1], relevantní pro tuto práci jsou ale jen ty, které poskytují pokročilejší možnosti měření. Tato kapitola představuje Automatovou knihovnu a nástroj SMART, srovnává jejich přístup k měření a komentuje jejich vliv na tuto práci.

## 1.1 Automatová knihovna

Automatová knihovna je rozsáhlý projekt vyvíjený zde na ČVUT FIT pod vedením Ing. Jana Trávníčka. Projekt v sobě zahrnuje jak samotnou programovou knihovnu, tak připravené aplikace obalující funkcionalitu programové knihovny. Implementačním jazykem je C++11.

Programová knihovna implementuje struktury popisující formalismy teorie formálních jazyků a teorie grafů spolu s algoritmy, které nad těmito strukturami operují. Implementace jak struktur, tak algoritmů se velmi blíží svému matematickému zápisu, není kladen důraz na jejich efektivitu.

Aplikace jsou ve své podstatě jednoúčelové utility, které poskytují rozhraní k malé části programové knihovny. Komplexních úkonů dosáhneme jejich spojováním do programových rour. Aplikace pro komunikaci mezi sebou používají protokol ve formátu XML.

Základní kamen pro knihovnu byl položen v [1], ale v této chvíli z původního návrhu zůstaly jen některé myšlenky, jádro bylo skoro celé přepsáno.

### 1.1.1 Měření

V Automatové knihovně před započetím této práce již existovala základní implementace měření.

Umožňuje přímo ve zdrojovém kódu určit oblasti, které se mají měřit, skrze jednoduché API. Měřené oblasti lze do sebe zanořovat, čímž můžeme dosáhnout jemné granularity měření.

Měřit lze pouze čas. Výstup měření je v textovém formátu, který není moc vhodný pro další strojové zpracování. Neexistuje žádná podpora pro automatizované měření, ani pro základní zpracování výsledků měření.

### 1.1.2 Algoritmy pro vyhledávání v textu

K dispozici jsou implementace čtyř algoritmů pro vyhledávání v textu: naivní algoritmus, Boyer-Moore-Horspool, obrácený Boyer-Moore-Horspool a Dead-Zone.

### 1.1.3 Vliv na tuto práci

Z existující implementace měření přebíráme API a myšlenku přímého určování měřených oblastí ve zdrojovém kódu spolu s možností tyto oblasti zanořovat.

Implementované algoritmy využijeme pro experimentální porovnání s nástrojem SMART.

## 1.2 SMART

SMART (String Matching Research Tool) [2] je nástroj pro výzkum v oblasti algoritmů pro řetězcové vyhledávání. Nástroj je napsán v jazyce C a nabízí jednoduché aplikační rozhraní pro implementaci těchto algoritmů.

### 1.2.1 Měření

SMART umožňuje lehce měřit jednotlivé algoritmy a získávat snadno interpretovatelné výsledky měření.

Neexistuje programové rozhraní měření, kterým lze určit, jaké části algoritmu se mají měřit, lze měřit pouze celý algoritmus. Měřenou doménou je pouze čas. Výstup měření není k dispozici v textové podobě – surové výsledky se ukládají do sdílené paměti, odkud je nelze snadným způsobem dostat.

Pro získání výsledků měření je třeba provést automatizovaný experiment. Tento experiment konfigurujeme skrze terminálovou aplikaci, kde vybereme jaké algoritmy se mají měřit, na jakých vstupních datech a vybereme velikost hledaného vzorku. SMART jednotlivá měření v experimentu několikrát zopakuje a výsledky zprůměruje, čímž dosahuje statisticky přesnějších výsledků. Výsledek experimentu je k dispozici v několika formátech např.: XML nebo HTML.

Vnitřně automatizované měření funguje tak, že skrze systémová volání spouští jednotlivé utility, ve kterých jsou algoritmy implementovány. Tyto

utility sami vystaví výsledky měření do sdílené paměti, odkud si je program pro automatizované měření vyzvedne a dále zpracuje.

### 1.2.2 Algoritmy pro vyhledávání v textu

Obsahuje 86 efektivně implementovaných algoritmů. Dále obsahuje korpus různých textů pro testování těchto algoritmů.

### 1.2.3 Vliv na tuto práci

Využijeme myšlenku obslužného programu, průměrování výsledků jednotlivých měření a možnost volby formátu výsledků experimentu.

Korpus textů použijeme pro porovnání výkonnosti algoritmů Automatové knihovny a algoritmů SMARTu.

Kapitola 2

# Měření v Automatové knihovně

Tato kapitola se zabývá principy, návrhem a implementací měření v Automatové knihovně. Dále ukazuje, jaké domény lze měřit a jaké informace z nich získáme. Nakonec popisuje, jak se k těmto informacím dostaneme.

## 2.1 Způsob měření

Měření je navrženo tak, aby programátor měl možnost měřit jen ty části programu, které chce. Skrze API měření přímo v kódu určuje *oblasti*, které se mají měřit. Oblast má své jméno a typ, sloužící zejména k sémanticky vhodnému popisu této oblasti a pro pozdější zpracování.

Takto definované oblasti lze do sebe zanořovat do libovolné hloubky. To nám dovoluje měřit zároveň velkou oblast (např.: běh celého algoritmu) a její menší části (např.: předzpracování dat a výpočetní jádro algoritmu). Zanořováním oblastí do sebe namísto jednoduché lineární struktury dostáváme stromovou strukturu.

V oblastech se měří čas, paměť a obecné události, které definuje sám programátor. Vždy se měří všechny tři domény, nelze si vybrat jen některé – cílem je získat co nejvíce dat z běhu programu, jejichž pozdějším zpracováním si uživatel vybere data pro něj relevantní.

Jakýkoli požadavek, který má být v měření vykonán, musí projít následujícími kroky, které představují jednotlivé úrovně abstrakce návrhu měření:

1. API měření
2. Jádro měření
3. Rámec měření
4. Datový rámec měření

### 2.1.1 API měření

Veškeré struktury a funkce související s měřením se nacházejí v knihovně `alib2common` a jsou součástí jmenného prostoru `measurements`. Pro použití měření v kódu je třeba inkludovat hlavičku `<measure>`. Celé měření se z hlediska programátora ovládá pouze šesti funkcemi, které definují API měření.

Funkce `start`, `end` slouží k označení začátku, konce oblasti měření. Funkce `reset` slouží k zahození aktuálního měření. Funkce `results` vrací výsledky aktuálního měření. Funkce `counterInc`, `counterDec` slouží ke zvýšení a snížení čítače o danou hodnotu. Jedná se o aliasy funkce `hint`, která slouží k notifikaci měření, že nastala nějaká programátorem definovaná událost.

Celé API měření tvoří fasádu pro volání metod *jádra měření*. Důvody pro jeho existenci jsou kompatibilita s původním API měření a kratší, výstižnější zápis v kódu.

### 2.1.2 Jádro měření

Funkcí jádra je udržovat dvě struktury – *strom měření* a *zásobník referencí*. Tyto struktury popisují stav měření vztahující se k nějakému konkrétnímu právě prováděnému místu v programu a jsou klíčové pro propagaci měřených hodnot mezi zanořenými oblastmi.

Uzlům stromu říkáme *rámce měření* a odpovídají každé dosud otevřené oblasti. Tyto rámce obsahují výsledky měření jednotlivých oblastí nebo aktuální hodnoty měření, pokud daná oblast ještě nebyla uzavřena.

Zásobník obsahuje reference na rámce ve stromu měření. Rámce referované zásobníkem korespondují oblastem, které jsou právě otevřeny a ještě nebyly uzavřeny, v pořadí, ve kterém byly otevřeny. Ve stromu měření se jedná o uzly, které leží na cestě od kořene stromu do uzlu, jenž je referován vrcholem zásobníku. Speciálně vrchol zásobníku referuje rámec v jehož oblasti se právě nacházíme.

Konstrukce stromu měření a udržování stavu zásobníku se děje následujícím způsobem:

- Při volání funkce API `start` je vytvořen nový rámec $r$. Nechť rámec $p$ je rámec referovaný vrcholem zásobníku. Rámec $r$ je zařazen mezi potomky rámce $p$, rámci $r$ je rámec $p$ nastaven jako rodič. Na zásobník je vložena reference na rámec $r$.

- Při volání funkce API `end` je zavřena oblast referovaná vrcholem zásobníku, ze zásobníku je odebrán jeho vrchol.

Lze nahlédnout, že strom měření generovaný výše popsaným způsobem je průchod do hloubky implicitního stromu měření, který je určen samotným průběhem vykonávání programu. Pro nás důležitým důsledkem je, že při

opakovaném spuštění daného deterministického programu na daných datech dostaneme vždy stejný strom měření.

Při inicializaci jádra je vytvořen kořen stromu měření a na zásobník je na něj vložena reference. Tento kořen je logickým uzlem, který nelze odebrat. Zajišťuje fakt, že strom měření má vždy pouze jeden kořen (nevzniká les měření).

#### Implementace

Jádro je implementováno třídou `MeasurementEngine`, na kterou je aplikován návrhový vzor *singleton* [3]. Tento vzor zabezpečuje, že za běhu programu bude existovat pouze jedna instance této třídy s jedním globálním přístupovým bodem. Tato instance je vytvořena ihned po spuštění programu a s jejím vytvořením dochází k inicializaci jádra.

Udržování stromu měření a zásobníku referencí je realizováno metodami jádra `pushMeasurementFrame` a `popMeasurementFrame`, které odpovídají funkcím API `start` a `end`. Metoda `resetMeasurements` odpovídá funkci API `reset` a je použita jak pro resetování jádra, tak pro jeho inicializaci, protože tyto dvě operace mají stejnou sémantiku.

### 2.1.3 Rámec měření

Rámec je datová struktura obsahující jméno a typ oblasti, kterou reprezentuje, referenci na rodiče, množinu referencí na potomky ve stromu měření, a datové rámce.

Sémantika typu oblasti je dána programátorem a je možno vybrat z předdefinovaných typů: `OVERALL`, `INIT`, `FINALIZE`, `PREPROCESS`, `ALGORITHM`, `MAIN` a `AUXILIARY`. Typ `ROOT` označuje kořen stromu měření a programátorovi je nepřístupný.

#### Implementace

Rámec je implementován strukturou `MeasurementFrame`. Typy jsou definované ve výčtovém typu `Type`.

### 2.1.4 Datový rámec měření

Existují tři druhy datového rámce – časový, paměťový a čítačový. Datový rámec definuje tři věci: data, jenž chceme měřením získat, způsob získání těchto dat a způsob agregace těchto dat, kterou lze sloučit více datových rámců do jednoho.

#### Implementace

Jednotlivé datové rámce jsou implementovány strukturami `TimeDataFrame`, `MemoryDataFrame` a `CounterDataFrame`. Každý z rámců implementuje logiku

ve své verzi tří statických funkcí `init`, `update` a `hint`. Všem těmto funkcím je parametrem předáván aktuální strom měření a odkaz na rámec, který má být zpracován. Funkci `hint` je ještě předána struktura popisující událost, která nastala.

Funkce `init` je volána při vytvoření rámce (otevření oblasti) a inicializuje měřená data. Funkce `update` je volána při odebrání rámce ze zásobníku (zavření oblasti) a nějakým způsobem propaguje měřená data do předků rámce, který byl právě zavřen. Volání funkce `hint` je svázáno s voláním stejnojmenné funkce API a nějakým způsobem zpracovává a propaguje data události na svém vstupu.

### 2.1.5 Měření času

Měříme dvě různé hodnoty: délku pobytu v daném rámci včetně jeho potomků a délku pobytu pouze v daném rámci. Jako jednotky jsou použity mikrosekundy.

Existují dva typy času, které se dají měřit – *wall clock time* a *CPU time*.

Wall clock time odpovídá času, který měří hodiny visící na zdi, tedy opravdovou délku běhu programu.

CPU time je čas, který je součtem časů, který strávila jednotlivá jádra procesoru počítáním našeho programu. Neformálně se jedná o čas, který by jednojádrový procesor potřeboval pro výpočet celého programu.

Pro nás je relevantní pouze wall clock time. Pro jeho měření má STL k dispozici několik druhů hodin, ze kterých používáme hodiny s vysokou přesností `chrono::high_resolution_clock`.

Agregace časových datových rámců je aritmetický průměr z naměřených časů.

### 2.1.6 Měření paměti

Program může dynamicky alokovat paměť buď na zásobníku, nebo na haldě.

Na zásobníku jsou typicky alokovány lokální proměnné a informace související s voláním funkcí. Z pohledu měření lze zásobník považovat za konstantně velkou paměť (bývá často omezen velikostí řádově desítek megabytů), tedy alokace paměti na zásobníku neměříme.

Halda je místo, odkud lze explicitně požádat o přidělení určitého objemu paměti. Takto přidělená paměť zůstává ve vlastnictví programu, dokud není explicitně uvolněna. V jazyce C++ se pro zažádání o paměť používají operátory `new`, `new[]` a pro uvolnění paměti operátory `delete`, `delete[]`. Verze operátorů s hranatými závorkami se používají při alokaci, dealokaci polí.

Alokace paměti na haldě již sledujeme. Získáváme informace o velikosti alokované paměti na haldě v okamžiku vytvoření rámce a v okamžiku jeho zavření. Dále sledujeme tzv. *high watermark*, což je největší objem paměti,

který byl v daném rámci najednou alokován. Počítáme ho pro daný rámec včetně jeho potomků i pouze pro daný rámec. Použité jednotky jsou byty.

Agregace pro paměťové datové rámce momentálně pouze vrací nezměněné hodnoty jednoho z rámců, který se má agregovat.

Pro korektní implementaci měření paměti na haldě bylo třeba využít dvou principů jazyka C++ a STL.

#### 2.1.6.1 Operátory `new` a `delete`

Jazyk C++ umožňuje nahradit globální operátory `new` a `delete` uživatelskou implementací.

Pokud se v jakémkoli zdrojovém souboru, který bude přilinkován do výsledného programu, nachází uživatelem definované funkce se stejnou signaturou jako mají globální operátory, dojde k jejich nahrazení [4].

Pro naše měření definujeme dvě sady operátorů `new` a `delete` – *proxy sadu* a *měřící sadu*.

Proxy sada operátorů má stejnou signaturu jako globální verze, tudíž je tyto operátory nahradí. Jsou definovaný ve zdrojovém souboru, který je přilinkován do výsledné knihovny pouze v debugovacím módu (tedy k nahrazení globálních operátorů dochází pouze v debugovacím módu). Tyto operátory pouze volají jejich odpovídající operátory v měřící sadě.

Měřící sada operátorů již nemá stejnou signaturu jako globální verze, obsahuje navíc jeden booleovský parametr `measure`. Operátory z proxy sady při volání operátorů z měřící sady parametru `measure` předávají hodnotu `true`. Operátory z měřící sady jsou implementovány následovně:

- V operátoru `new` při požadavku na alokaci $n$ bytů alokujeme $n + s$ bytů, kde $s$ je počet bytů nutný k uložení neznaménkového celočíselného typu *size_t* (typicky 4 nebo 8 bytů). K alokaci je použita funkce `malloc`, která vrací ukazatel $p$ na začátek alokovaného bloku paměti. Na prvních $s$ bytů tohoto bloku je uložena velikost požadavku a ukazatel $p$ je posunut dopředně o $s$ bytů, čímž dostaneme ukazatel $p'$. Pokud alokace proběhla úspěšně a parametr `measure` má hodnotu `true`, je jádro měření notifikováno, že došlo k alokaci $n$ bytů. Nakonec je vrácen posunutý ukazatel $p'$.

- V operátoru `delete` při požadavku na uvolnění paměti referované ukazatelem $p'$ nejdříve ukazatel $p'$ posuneme zpět o $s$ bytů, čímž dostaneme ukazatel $p$. Je přečtena velikost požadavku na alokaci $n$, ze které ukazatel $p$ vznikl. Pokud parametr `measure` má hodnotu `true`, je jádro měření notifikováno o dealokaci $n$ bytů. Na ukazatel $p$ je volána funkce `free`.

#### Implementace

Na uživatelskou implementaci obou operátorů jsou standardem definovány jisté požadavky. Dále je třeba dbát, aby chování nových operátorů bylo konzistentní s chováním jejich defaultních verzí. Informace vedoucí ke korektní implementaci byly čerpány z [5].

S příchodem jazyka C++11 není třeba nahrazovat operátory `new[]` a `delete[]`, protože jejich implementace interně používá operátory `new` a `delete` [4].

#### 2.1.6.2 STL kontejnery a `allocator`

Pro přesné měření paměti na haldě je třeba odfiltrovat alokace, které dělá samotné měření. K těmto alokacím dochází při ukládání prvků do STL kontejnerů jako jsou `std::string`, `std::vector` a `std::map`.

STL umožňuje změnit způsob, jakým se alokuje paměť v těchto kontejnerech, skrze uživatelem definovaný `allocator`. Při vytváření nového kontejneru lze uživatelský `allocator` specifikovat skrze argument šablony daného kontejneru, čímž vzniká nový typ kontejneru.

#### Implementace

Pro měření definujeme nový `stealth_allocator`. Jeho minimální funkční implementace definuje typový alias `value_type` na typ, nad kterým bude operovat, a metody `allocate` a `deallocate`. Tyto metody volají přímo operátory `new` a `delete` z měřící sady operátorů a parametru `measure` předávají hodnotu `false`, tedy se alokace a dealokace skrze `stealth_allocator` nepromítnou do měření.

Pro vybrané kontejnery STL odvozujeme jejich měření nezkreslující protějšky: `stealth_string`, `stealth_vector` a `stealth_map`.

### 2.1.7 Měření čítačem

Měření čítačem slouží pro sledování obecných událostí, které definuje sám programátor (např.: počet porovnání znaků nebo počet stavů v automatu).

Čítač je reprezentován celočíselným datovým typem. Je identifikovaný svým jménem, které je unikátní napříč celým měřením, a skrze API měření k němu můžeme přičíst nebo odečíst nějakou konstantu. Ve zdrojovém kódu je není třeba definovat, jsou automaticky vytvořeny při prvním použití.

Udržujeme dvě verze čítačů: první počítá události pro daný rámec včetně jeho potomků a druhá pouze pro daný rámec.

Agregace pro časové datové rámce stejně jako v případě paměťových datových rámců vrací nezměněné hodnoty jednoho z rámců, který se má agregovat.

## 2.2 Výsledek měření

Funkce API `results` vrací instanci struktury `MeasurementResults`. V této instanci je uložena kopie stromu měření ve stavu, v jakém se nacházel při volání funkce `results`. Pokud v okamžiku tohoto volání existují ještě neuzavřené rámce (s výjimkou kořene stromu měření), měřená data nebudou odpovídat skutečnosti.

Účelem struktury `MeasurementResults` je vypisování stromu měření a agregace stromů měření, čehož využíváme v automatizaci měření.

### 2.2.1 Výpis a formátování

Pro výpis této struktury lze použít libovolný stream (abstrakce obalující V/V) odvozený od třídy `ostream`. Lze ji vypsat ve třech různých formách – *list*, *strom* a *XML*. Z těchto forem vybíráme použitím manipulátoru výstupu (globální funkce měnící vlastnosti daného streamu), kterému je předán odpovídající formát `LIST`, `TREE` nebo `XML` definované ve výčtovém typu `MeasurementFormat`. Defaultně je použit formát XML. Překlad struktury `MeasurementResults` z a do formátu XML obstarává třída `MeasurementResultsXml`.

Formáty list a strom slouží uživateli pouze pro ověření, že strom měření má očekávanou strukturu. Pokud data chceme dále zpracovávat, je třeba použít formát XML. Pro příklad výpisu výsledků měření ve formátu XML viz přílohu B.1.

### 2.2.2 Agregace

Agregaci lze použít pouze pro stromy měření se stejnou strukturou, kterou si agregovaný strom měření zachová. Vytvořme očíslování uzlů vstupních stromů a agregovaného stromu, které odpovídá pořadí jejich otevření při průchodu do hloubky. Rámce v agregovaném stromě potom vzniknou agregací rámců vstupních stromů se stejným číslem.

## 2.3 Výstup měření

Program po svém startu má k dispozici tři otevřené speciální soubory `stdin`, `stdout` a `stderr`, které používá pro vstup, výstup a chybový výstup. V jazyce C++ jsou na tyto soubory standardně navázány streamy: `std::cin` na `stdin`, `std::cout` na `stdout` a `std::cerr` spolu s `std::clog` na `stderr`.

Programy automatové knihovny na svůj standardní výstup vypisují výsledek svého běhu. Na svůj chybový výstup potom zapisují vše ostatní: některé chyby programu, logovací zprávy a výsledky měření.

Pro potřeby automatizace měření je nutné oddělit výstup měření od ostatních výpisů. Této funkcionality dosáhneme implementací nových streamů,

které umožňují uživateli přesměrovat vypisování výsledků měření i logovacích zpráv mimo `stderr`.

### 2.3.1 Mechanismy UNIXových operačních systémů

K implementaci nových streamů je třeba znát jisté mechanismy, které se týkají souborových deskriptorů, vytváření procesů a přesměrování V/V v UNIXových systémech. Podkladem této sekce je [6].

#### Soubory a souborové deskriptory

Koncepce souboru v UNIXových systémech popisuje kromě obyčejných souborů (textové, binární) i soubory speciální jako jsou paměť, zařízení, pojmenované roury atp.

Procesy se soubory pracují skrze souborové deskriptory. Souborový deskriptor je nezáporné celé číslo, které v rámci daného procesu slouží k identifikaci konkrétního souboru. Každý proces si udržuje vlastní sadu souborových deskriptorů, které jsou mezi procesy nezávislé. Proces může používat více deskriptorů k identifikaci jednoho souboru. Skrze souborové deskriptory lze přímo provádět V/V operace nad souborem, který identifikuje (např.: funkce `read`, `write`).

Souborům `stdin`, `stdout` a `stderr` standardně odpovídají deskriptory 0, 1 a 2.

#### Vytváření procesů a `fork`

Procesy vznikají (kromě několika úplně prvních) pouze klonováním již existujících. Používáme k tomu systémové volání `fork`. Proces, který je předlohou pro klonování, nazýváme *rodičovský proces* a proces, který je výsledkem klonování, nazýváme *synovský proces*. Synovský proces je přesnou kopii svého rodiče (s několika výjimkami popsanými v [7]).

Pro nové streamy je důležitý fakt, že synovský proces bude mít k dispozici kopie souborových deskriptorů rodičovského procesu, které identifikují stejné soubory.

Tohoto faktu využívá shell při přesměrování V/V nějakého programu. Než shell program spustí skrze dvojici systémových volání `fork` a `exec`, připraví pro něj souborové deskriptory, které program poté transparentně používá. Typickým využitím je přesměrování výstupu jednoho programu na vstup druhého programu (tzv. roura). Shell dává uživateli možnost určit pro spouštěný program, jaké souborové deskriptory se pro něj mají otevřít a na jaké soubory tyto deskriptory budou ukazovat.

### 2.3.2 Nové streamy

Pro výstup měření je vytvořen nový stream `cmeasure`. Dále je upraveno chování streamu `std::clog`. Oběma streamům je přidělen nový souborový deskriptor: 5 pro `cmeasure` a 4 pro `std::clog`.

K vytvoření streamu `cmeasure` a k úpravě streamu `std::clog` dochází při startu programu. Pokud jsou v tento okamžik otevřeny jejich odpovídající souborové deskriptory, budou je používat pro své V/V operace, jinak se uchýlí k použití souborového deskriptoru 2 – standardního chybového výstupu.

#### Implementace

Standardní streamy jsou instance tříd `std::ostream` a `std::istream`. Tyto třídy obalují instanci třídy `std::streambuf`, což je abstrakce nějakým způsobem zajišťující V/V operace.

Pro streamy pro souborové deskriptory implementujeme třídu `fdstreambuf`, která dědí ze třídy `std::streambuf`. Třída `fdstreambuf` obaluje daný souborový deskriptor, nad kterým provádí V/V operace.

V GCC existuje nestandardní rozšíření standardní knihovny plnící stejnou funkci jako `fdstreambuf`: `stdio_filebuf`, nicméně bylo rozhodnuto raději nestandardní rozšíření nepoužívat.

Třídu `fdstreambuf` defaultně využívají třídy `ofdstream` a `ifdstream`. Třídy `ofdstream` a `ifdstream` dědí ze tříd `std::ostream` a `std::istream`.

Stream `cmeasure` je instance třídy `ofdstream`.

Úprava streamu `std::clog` spočívá v náhradě jeho interní instance třídy `std::streambuf` naší novou instancí třídy `fdstreambuf`.

Vytvoření streamu `cmeasure` a úprava streamu `std::clog` jsou spolu s třídami `fdstreambuf`, `ofdstream` a `ifdstream` součástí knihovny `alib2std` a rozšiřují jmenný prostor `std`.

Kapitola 3

# Automatizované měření

V této kapitole představujeme řešení automatizovaného měření v Automatové knihovně. Toto řešení zkoumáme jak z hlediska návrhu, tak z hlediska implementace.

## 3.1 Způsob měření

Automatizace měření umožňuje uživateli snadno vykonat měřící experiment, jehož účastníky jsou programy Automatové knihovny.

Uživatel svůj experiment popisuje konfiguračním souborem automatizovaného měření.

V tomto souboru zadává na jakých vstupních datech se má měřit a jaké programy se mají měřit.

Vstupní data se zadávají buď jako předpřipravený soubor v souborovém systému, nebo jako shellový příkaz, který je vygeneruje buď sám o sobě, nebo z již vygenerovaných dat, jejichž cesta k nim je příkazu předána parametrem na uživatelem určené místo v příkazu (typické je např.: generování vzorku z daného textu).

Programy jsou zadávány formou shellového příkazu, ze kterých lze vytvořit programovou rouru – konstrukci, kde se výstup příkazu přesměrovává na vstup následujícího příkazu. Vstupní data tyto programy dostávají opět parametrem na uživatelem zadaná místa ve formě cesty k souboru vstupních dat.

Výstupem experimentu je strukturovaný soubor výsledků všech uskutečněných měření, který lze dále strojově zpracovávat.

Automatizované měření spouštíme programem `ameasure2`, kterému parametrem předáváme cestu ke konfiguračnímu souboru.

### Implementace

Rozšíření je implementováno v knihovně `alib2measurepp` ve jmenném prostoru `measurements`.

Základy tohoto rozšíření jsou položeny ve třídě `MeasurementProvisioner` a její statické metodě `runConfiguration`.

## 3.2 Definice pojmů

Pro potřeby této kapitoly definujeme a upřesňujeme některé pojmy pro jednoznačný popis problematiky automatizovaného měření.

### 3.2.1 Obecné pojmy

*Shellový příkaz* je textový řetězec, který lze vykonat standardním shellem skrze systémové volání `system`. Příkaz může obsahovat libovolné konstrukce shellu. Výjimku tvoří proměnné označující prvních deset vstupních argumentů příkazu ($0, $1, ..., $9), které mají v automatizovaném měření speciální význam a jsou zpracovány ještě před předáním příkazu shellu (viz *substituční proměnná*).

*Shellová cesta* je textový řetězec, který lze expandovat standardním shellem. Po expanzi by cesta měla ukazovat na alespoň jeden soubor v souborovém systému.

*Pracovní adresář* procesu je adresář, vůči kterému jsou vypočítávány relativní cesty v rámci tohoto procesu.

### 3.2.2 Pojmy automatizovaného měření

*Iterační počet* je kladné celé číslo, které určuje, kolikrát se má zopakovat měření programové roury pro jedny konkrétní vstupní data. Jednotlivé výsledky jsou poté agregovány.

*Zdroj dat* definuje konkrétní způsob jakým jsou vstupní data získána. Tento způsob je určen jeho typem. Existují dva typy zdrojů dat – *file* a *generator*.

*Vstupní dávka* je logické sdružení alespoň jednoho zdroje dat. Způsob zpracování dané dávky určuje její typ. Existují dva typy vstupních dávek – *simple* a *dependency*.

### 3.2.3 Pojmy zpracování vstupních dat

*Substituční skupina* je soubor zdrojů dat v rámci jedné dávky. Substituční skupiny označujeme celým jednociferným číslem z rozsahu 0–9. Do jaké skupiny je zdroj dat zařazen určuje jeho *identifikátor substituční skupiny*.

*Substituční proměnná* je speciální symbol, který se může vyskytovat v shellovém příkazu. Formát symbolu je dvojznak, kde prvním znakem je dolarový znak: `$` a druhým znakem je identifikátor substituční skupiny. Substituční proměnná v shellovém příkazu označujeme místo, na které bude za substituční proměnnou dosazena cesta k souboru se vstupními daty.

*Substituční vzor* je množina identifikátorů substitučních skupin. U vstupní dávky definujeme *zdrojový substituční vzor* jako sjednocení identifikátorů zdrojů dat, které se v dané vstupní dávce nachází. U shellového příkazu definujeme *cílový substituční vzor* jako sjednocení identifikátorů substitučních proměnných, které se v daném příkazu vyskytují.

*Substituční mapa* je množina dvojic identifikátor substituční skupiny – cesta k souboru vstupních dat.

## 3.3 Konfigurace automatizovaného měření

Konfigurace je řízena konfiguračním souborem, který je ve formátu XML.

Kořenovým elementem konfiguračního souboru je element **MeasurementProvisioner**. V tomto elementu se nachází definice prostředí, vstupních dat a programových rour.

### 3.3.1 Prostředí

Definice prostředí je uvozena elementem **Environment**. V tomto elementu definujeme nastavení pracovního adresáře a nastavení iteračního počtu.

Nastavení pracovního adresáře uvozujeme elementem **WorkingDirectory**. Obsahem tohoto elementu je shellová cesta k adresáři, který má být nastaven jako pracovní. Toto nastavení je nepovinné, při jeho absenci je použita cesta k aktuálnímu pracovnímu adresáři.

Nastavení iteračního počtu uvozujeme elementem **PipelineIterations**. Jeho obsahem je kladné celé číslo. Toto nastavení je nepovinné s defaultní hodnotou 1.

### 3.3.2 Vstupní data

Definice vstupních dat je uvozena elementem **InputData**. V tomto elementu následují definice alespoň jedné vstupní dávky.

Definice vstupní dávky je uvozena elementem **InputBatch**. Tento element má jeden nepovinný atribut **dependency** typu boolean. Při hodnotě `true` je typ dávky vynucen jako typ *dependency*. Při hodnotě `false` nebo při absenci tohoto atributu se použije automatické rozpoznání typu dávky.

V elementu **InputBatch** se nacházejí jednotlivé definice zdrojů dat. Podle typu zdroje dat pro jeho uvození použijeme element **File** nebo element **Generator**. Oba elementy mají jeden povinný atribut **id** a dva nepovinné atributy **count** a **alias**.

Atribut **id** je jednociferné celé číslo z rozsahu 0–9 a jedná se o identifikátor substituční skupiny daného zdroje dat.

Atribut **count** je celé kladné číslo, které říká kolikrát je daný zdroj dat uvažován. Tento atribut je zkratka pro uživatele, aby nemusel několikrát opakovat tu samou definici zdroje dat. Defaultní hodnota je 1.

Atribut **alias** je textový řetězec, který slouží uživateli pro sémantické označení daného zdroje dat. Při absenci tohoto atributu je alias vygenerován z obsahu definice zdroje dat.

Obsah elementu **File** je shellová cesta a obsah elementu **Generator** je shellový příkaz.

### 3.3.3 Programové roury

Element **Pipelines** obaluje jednotlivé definice programových rour, které uvozujeme elementem **Pipeline**.

Jednotlivé shellové příkazy v programové rouře uvozujeme elementem **Command**. Tento element má dva nepovinné atributy **measure** a **alias**.

Atribut **measure** je typu boolean. Při hodnotě `false` je příkaz spuštěn, ale není měřen. Při hodnotě `true` nebo při jeho absenci je příkaz jak spuštěn, tak i měřen.

Atribut **alias** má stejný význam jako u zdrojů dat, tedy sémanticky popis daného příkazu. Při jeho absenci je použit obsah definice daného příkazu.

#### Implementace

Struktura `MeasurementProvisionerConfiguration` popisuje obsah konfiguračního souboru, její instance je použita pro přístup k jednotlivým položkám konfigurace. Třída `MeasurementProvisionerConfigurationXml` zajišťuje parsování konfiguračního souboru.

Pro příklad konfiguračního souboru viz přílohu B.2.

## 3.4 Zpracování vstupních dat

Účelem zpracování vstupních dat je připravit soubor substitučních map pro proces měření programových rour.

Prvním krokem zpracování vstupních dat je nalezení globálního cílového substitučního vzoru. Jedná se o sjednocení cílových substitučních vzorů všech příkazů všech programových rour. Tento vzor určuje, za jaké různé substituční proměnné budeme v programových rourách dosazovat.

Následuje zpracování jednotlivých vstupních dávek. Pokud zdrojový substituční vzor dané vstupní dávky neodpovídá globálnímu cílovému substitučnímu vzoru, je ohlášena chyba a zpracování je ukončeno. Vstupní dávky se zpracovávají izolovaně od ostatních vstupních dávek.

#### Implementace

Zpracování vstupních dat je implementováno ve třídě `MPInputData`. Skrze metody této třídy se dostáváme k výsledkům zpracování vstupních dat.

### 3.4.1 Zpracování zdrojů dat

U zdroje typu *file* je expandována shellová cesta v jeho definici. Výsledkem expanze je množina cest k souborům v souborovém systému, která je vrácena jako výstup zdroje. Pokud má být použita pro zpracování tohoto zdroje substituční mapa, daná substituční mapa je ignorována.

U zdroje typu *generator* je nejdříve vytvořen nový soubor s náhodným unikátním jménem, nazvěme ho $f$. Soubor $f$ je vytvořen v operační paměti systému (ne na jeho disku), abychom urychlili V/V operace nad ním. Pokud má být použita pro zpracování tohoto zdroje substituční mapa, daná substituční mapa je použita pro nahrazení substitučních proměnných v shellovém příkazu zdroje. Pokud žádná substituční mapa nemá být použita, je shellový příkaz zdroje ponechán beze změny.

Následně je shellový příkaz zdroje spuštěn, jeho výstup je přesměrován do souboru $f$. Cesta k souboru $f$ je vrácena jako výstup zdroje.

#### Implementace

Expanze shellových cest se vykonávají skrze volání funkce `wordexp`.

Pro vytváření a mazaní souborů v operační paměti (tzv. objekty sdílené paměti) se používají funkce `shm_open` a `shm_close`. Na typickém UNIXovém systému se tyto soubory vytváří ve složce `/dev/shm`.

Náhodná jména souboru se skládají z náhodného čísla a z hodnoty čítače, který počítá, kolik takových souborů bylo vytvořeno. V rámci běhu jednoho programu tedy není možné dvakrát vygenerovat stejné jméno.

Cesty z výstupu zdrojů dat jsou obaleny strukturou `MPInputDatum`. Tato struktura si udržuje jaký typ zdroje tuto cestu vytvořil, což určuje způsob, jakým je cesta uvolněna: u typu *file* se neděje nic, u typu *generator* je soubor smazán.

Pomocné funkce, které obalují výše popsanou funkcionalitu, jsou implementovány ve třídě `MPUtils`.

### 3.4.2 Zpracování vstupních dávek

Nejprve je třeba určit typ vstupní dávky, který určuje způsob jejího dalšího zpracování.

Vstupní dávka je typu *simple* tehdy a jen tehdy, pokud její zdroje dat neobsahují ve svých definicích žádné substituční proměnné a pokud u dávky nebyl atributem vynucen typ *dependency*.

Ve všech ostatních případech se jedná o typ *dependency*.

Výstup zpracování jedné vstupní dávky je soubor substitučních map. Tyto soubory kumulujeme v globálním souboru substitučních map.

#### 3.4.2.1 Vstupní dávka typu *simple*

Pro každou substituční skupinu ve zdrojovém substitučním vzoru dané vstupní dávky je vytvořena fronta, do kterých budeme postupně přidávat jednotlivé výstupy zdrojů dat.

Po zpracování určitého zdroje jsou jeho výstupy zařazeny do fronty odpovídající jeho identifikátoru substituční skupiny.

Po zpracování všech zdrojů dat v dané dávce je zkontrolováno, zda-li je počet prvků ve všech frontách stejný, pokud není, je ohlášena chyba a zpracování je ukončeno.

Vytvořme prázdný soubor substitučních map $S$.

Ze všech front naráz odebereme první prvky, ze kterých vytvoříme jednu substituční mapu, kterou přidáme do souboru $S$. Tento proces opakujeme, dokud nejsou fronty prázdné.

Jakmile jsou fronty prázdné, soubor $S$ obsahuje výsledné substituční mapy a je vrácen jako výstup zpracování dané dávky.

#### 3.4.2.2 Vstupní dávka typu *dependency*

Zpracování postupuje po jednotlivých substitučních skupinách ve zdrojovém substitučním vzoru dané dávky vzestupně podle jejich identifikátoru. Tento proces má dvě fáze, během kterých postupně konstruuje výsledné substituční mapy.

- První fáze zpracovává pouze substituční skupinu s nejnižším identifikátorem. Používá se stejný způsob jako u dávky typu *simple*. Výsledkem je soubor částečných substitučních map $S$.

- Druhá fáze zpracovává všechny ostatní substituční skupiny.

  Předpokládejme, že zpracováváme substituční skupinu $K$ a z předešlé iterace máme k dispozici soubor částečných substitučních map $S$. Vytvoříme prázdný soubor substitučních map $S'$. Pro každou substituční mapu $s$ ze souboru $S$ a pro každý zdroj dat $z$ ve skupině $K$ proveď následující krok:

  – Zpracuj zdroj $z$ za použití substituční mapy $s$. Pro každý jeho výstup $v$ vytvoř novou substituční mapu $s'$, která vznikne rozšířením substituční mapy $s$ o výstup $v$. Substituční mapu $s'$ přidej do souboru $S'$.

  Následně dojde k náhradě obsahu v souboru $S$ obsahem souboru $S'$ a pokračuje se zpracováním substituční skupiny následující po substituční skupině $K$. Pokud žádná taková neexistuje, soubor $S$ obsahuje výsledné substituční mapy a je vrácen jako výstup zpracování dané dávky.

## 3.5 Měření programových rour

Měření jednotlivých programových rour se uskuteční zvlášť pro jednotlivé substituční mapy z globálního souboru substitučních map stejným způsobem, předpokládejme tedy po zbytek této sekce, že měříme danou programovou rouru $p$ pro danou substituční mapu $s$.

Měření programové roury $p$ spočívá v simulování jejího běhu postupným spouštěním jednotlivých shellových příkazů.

Před spuštěním shellového příkazu v něm nahradíme substituční proměnné skrze substituční mapu $s$. Pokud v definici daného příkazu nebyl atribut **measure** nastaven na hodnotu `false`, je k shellovému příkazu doplněn přepínač `-m`, který zapíná měření daného příkazu.

Při úspěšném vykonání příkazu jsou zaznamenány výsledky měření (pokud existují) a jeho výstup je přesměrován na vstup příkazu následujícího. V případě, že žádný takový neexistuje, je tento výstup zahozen.

Pokud nastane během vykonávání příkazu chyba, je simulace programové roury zastavena a chyba je zaznamenána ve statusu programové roury ve výsledcích měření.

Měření programové roury $p$ opakujeme tolikrát, kolik je hodnota iteračního počtu. Jednotlivé výsledky odpovídající jednomu příkazu jsou agregovány do jednoho finálního výsledku.

### Implementace

Simulace programové roury probíhá skrze soubory v operační paměti systému (jedná se o ten samý typ jako u zdrojů dat typu *generator*). Těmito soubory simulujeme standardní vstup, výstup, chybový výstup a výstup pro měření. Těmto souborům nastavíme souborové deskriptory 0, 1, 2 a 5 použitím systémových funkcí `close`, `dup` a `dup2` ve správném pořadí – souborové deskriptory se přiřazují od prvního nejmenšího možného. Tyto funkce slouží k přímé manipulaci se souborovými deskriptory a umožňují nám je zavírat a duplikovat.

Využíváme faktu, že proces, který vznikne spuštěním shellového příkazu skrze systémové volání `system`, bude mít takto připravené deskriptory 0, 1, 2 a 5 otevřené a k dispozici pro V/V operace.

Pro zaznamenání výsledků měření shellového příkazu (pokud se jedná o program Automatové knihovny) využíváme nového rozšíření popsané v kapitole 2, kdy jsou výsledky měření zapisovány na souborový deskriptor 5, pokud je otevřen v době inicializace streamu `std::cmeausre`.

Po vykonání příkazu vymažeme soubory pro vstup, chybový výstup a výstup pro měření pomocí funkce `ftruncate` a přetočíme je na začátek pomocí funkce `fseek`. Soubor pro výstup pouze přetočíme na začátek, data v něm budou přesměrována na vstup následujícího příkazu. Toto přesměrování potom pouze spočívá v prohození souborových deskriptorů 0 a 1.

## 3.6 Výsledek a výstup automatizovaného měření

Výsledky automatizovaného měření jsou uloženy v instanci hierarchické struktury `MeasurementProvisionerResults`. Jednotlivé úrovně této struktury bráno od nejvyšší po nejnižší jsou: vstupní data, programové roury a příkazy programových rour.

V úrovni vstupních dat ukládáme výčet identifikátorů substitučních skupin a aliasů zdrojů dat odpovídající substituční mapě, která byla použita pro nahrazovaní substitučních proměnných při spouštění programových rour.

V úrovni programových rour je status dané programové roury. Pokud programová roura skončí chybou, je ve statusu uložen návratový kód chyby, která nastala, její popis a příkaz, který tuto chybu vyvolal. Pokud chyba nenastala, status pouze obsahuje návratový kód 0.

V úrovni příkazů programových rour zaznamenáváme alias příkazu a výsledný strom měření daného příkazu.

Strukturu `MeasurementProvisionerResults` lze vypisovat ve formátu XML skrze libovolný stream odvozený od třídy `std::ostream`, její překlad z a do formátu XML zajišťuje třída `MeasurementProvisionerResultsXml`. Pro příklad výsledků automatizovnaého měření viz přílohu B.3.

Kapitola 4

# Zpracování výsledků

V této kapitole rozebíráme možnosti strojového zpracování výsledků automatizovaného měření.

## 4.1 Způsob zpracování

Výstup z automatizovaného měření může snadno dosahovat i několika desítek tisíců řádků formátovaného XML. Takové množství informací nelze rozumně zpracovávat ručně, proto automatizované měření doplňujeme o možnost strojového zpracování jeho výstupu.

Strojové zpracování v sobě zahrnuje filtrování výsledků podle měřených příkazů nebo podle typů a jmen rámců měření, výběr datové domény, datového pohledu, možnost agregace a volbu formátu výstupu.

Tyto volby uživatel předává programu `ameasurep2` skrze parametry příkazové řádky spolu s cestou k souboru s výsledky z automatizovaného měření.

### Implementace

Zpracování výsledků je implementováno v knihovně `alib2measurepp` ve jmenném prostoru `measurements`.

Hlavními účastníky zpracování výsledků jsou třída `MeasurementProcessor` a její statické metody `process` a `output`.

## 4.2 Filtrování

Filtrování operuje nad strukturou `MeasurementProvisionerResults`. Vstupní instanci nazvěme $R_{in}$ a výstupní $R_{out}$.

Filtr definujeme jako pravidlo, které určuje, jak má vypadat určitá vlastnost nějaké entity v $R_{in}$. Pokud daná entita projde všemi uživatelem zadanými filtry, je ponechána v $R_{out}$, jinak je vyřazena.

Momentálně můžeme filtrovat dvě entity – příkazy programových rour a rámce měření.

### 4.2.1 Filtr příkazů programových rour

Příkazy lze filtrovat podle jejich aliasů. Aby byly výsledky měření příkazu zahrnuty v $R_{out}$, musí jeho alias přesně odpovídat hodnotě zadaného filtru. Na příkazové řádce ho definujeme skrze přepínač `--filterCommand`.

### 4.2.2 Filtr rámců měření

Rámce měření filtrujeme podle jejich aliasů nebo jejich typů. Aby daný rámec měření zůstal ve svém stromu měření v $R_{out}$ musí buď jeho alias, nebo typ (záleží na druhu filtru) přesně odpovídat hodnotě zadaného filtru.

Pokud rámec měření filtrem neprojde, je odstraněn ze svého stromu měření a jeho potomci jsou zařazeni mezi potomky jeho rodiče. Pokud po jeho odebrání nastane situace, že ve stromu měření, odkud byl daný rámec odebrán, zůstane pouze kořen, je z $R_{out}$ odebrán celý příkaz, kterému tento strom měření odpovídal.

Na příkazové řádce definujeme filtr aliasu přepínačem `--filterFrameName` a filtr typu přepínačem `--filterFrameType`.

## 4.3 Volba datové domény a datového pohledu

V jeden čas lze zpracovávat pouze jednu ze tří možných datových domén – čas, paměť, nebo čítače. Volbu uskutečníme přepínačem `--engine` s hodnotou `time`, `memory`, nebo `counter`. Defaultní hodnota je `time`.

Datový pohled určuje hodnotu, která nás z dané domény zajímá, a vybíráme ho skrze atributy dané domény použitím přepínačů `--engineAttr`.

### 4.3.1 Obecné datové pohledy

Pokud je zadán atribut `inFrame`, pohled se omezí pouze na hodnoty uvnitř rámce bez jeho potomků. Pokud zadán není, pohled zahrnuje hodnoty rámce včetně jeho potomků.

### 4.3.2 Časové datové pohledy

V časové doméně je k dispozici pouze jeden pohled: délka setrvání v rámci měření udávaná v mikrosekundách. Pro výběr použijeme atribut `duration`, což je zároveň výchozí hodnota.

### 4.3.3 Paměťové datové pohledy

Vybíráme jeden pohled ze čtyř možných. Hodnoty všech paměťových pohledů jsou udávaný v jednotkách bytů.

- Atribut `startHeapUsage` – objem alokované paměti při vstupu do rámce.
- Atribut `endHeapUsage` – objem alokované paměti při výstupu z rámce.
- Atribut `highWatermark` – nejvyšší alokovaný objem paměti během pobytu v rámci.
- Atribut `delta` – objem paměti alokovaný během pobytu v rámci. Tento pohled odpovídá výrazu: `delta = highWatermark - startHeapUsage`.

### 4.3.4 Čítačové datové pohledy

Zadaný atribut určuje přímo jméno čítače, jehož hodnotu chceme na výstupu. Jednotky čítačů jsou nespecifikované.

## 4.4 Agregace

V aktuální implementaci existuje pouze jedna možnost agregace – agregování přes stejné kombinace vstupů.

Dvě kombinace vstupů jsou stejné, pokud se shodují v identifikátorech substitučních skupin a jím příslušných aliasů zdrojů dat, neboli jsou stejné, pokud odpovídají stejné substituční mapě.

Výsledky jednotlivých měření každého příkazu jsou agregovány v rámci skupiny stejných kombinací vstupů.

## 4.5 Formát výpisu

Pro výpis zpracované instance struktury `MeasurementProvisionerResults` máme k dispozici čtyři různé formáty: XML, CSV, HTML a statistika. Formát vybíráme přepínačem `--output` s hodnotou `xml`, `csv`, `html`, nebo `stats`.

### 4.5.1 XML

Tento formát je stejný jako výpis programu `ameasure2`, tedy je i opět validním vstupem programu `ameasurep2`.

### 4.5.2 Tabulkové formáty

Každý rámec měření (kromě kořenů stromu měření) je nejdříve transformován do maticové reprezentace $M$.

Počet řádků této matice odpovídá počtu substitučních map, na kterých byla měření uskutečněna, a počet sloupců odpovídá počtu příkazů, které byly měřeny. Prvek $M_{i,j}$ této matice potom odpovídá naměřené hodnotě pro $j$-tý příkaz na $i$-té substituční mapě v daném rámci. O jaké hodnoty konkrétně jde je určeno datovým pohledem.

Tabulka pro výpis z matice $M$ vznikne anotováním každého řádku matice za sebe spojenými aliasy zdrojů dat odpovídající substituční mapě v daném řádku a každého sloupce matice aliasem příkazu odpovídajícího danému sloupci.

Následně jsou tabulky jednotlivých rámců vypsány ve zvoleném formátu CSV nebo HTML.

### 4.5.3 Statistika

Tento formát vypisuje základní informace o experimentu, ze kterého vznikly dané výsledky.

Tyto informace jsou: na kolika substituční mapách se měřilo, jaké programové roury jsme měřili a zda-li nastala během vykonávání experimentu nějaká chyba.

Kapitola 5

# Algoritmy pro řetězcové vyhledávání

Tato kapitola představuje jeden z možných přístupů k řetězcovému vyhledávání. Dále se zaměřujeme na tři algoritmy řešící tento problém, jejichž implementacemi rozšiřujeme Automatovou knihovnu. Definice pojmů a popisy algoritmů jsou dle [8, 9, 10].

## 5.1 Definice problému a pojmů

Problém řetězcového vyhledávání spočívá v nalezení všech výskytů *vzorku* $P = p_1p_2...p_m$ v dlouhém *textu* $T = t_1t_2...t_n$, kde $P$ i $T$ jsou sekvence znaků z konečné množiny znaků $\Sigma$, kterou nazýváme *abeceda*.

Nechť máme řetězce $x$, $y$, $z$ a $w$. Řetězec $x$ nazveme *sufixem* řetězce $y$, pokud platí $y = zx$. Řetězec $x$ nazveme *faktorem* řetězce $y$, pokud platí $y = zxw$.

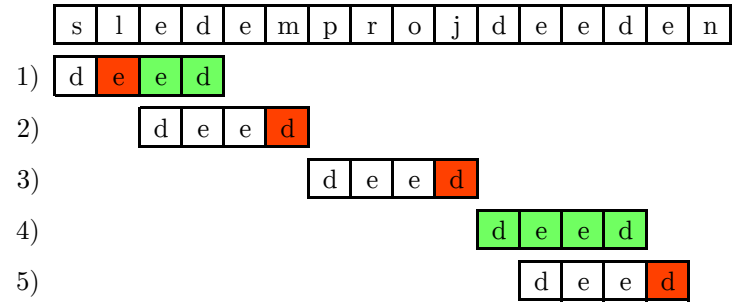
Mějme dán řetězec $x = a_1a_2...a_n$, potom řetězec $x^R = a_na_{n-1}...a_1$ nazýváme *reverze* řetězce $x$.

## 5.2 Principy řešení problému

Omezíme se na tři principy, jak řetězcové vyhledávání řešit: plovoucí okénko, protisměrné vyhledávání a faktorový přístup.

*Plovoucí okénko* je oblast textu délky $m$, která je možným kandidátem na výskyt vzorku. Okénko budeme posouvat zleva doprava a algoritmus vyhledávání končí jakmile se okénko ocitne mimo text. Rozlišovacím znakem algoritmů založených na posuvu plovoucího okénka je způsob, jakým toto okénko posouvají.

*Protisměrné vyhledávání* určuje, že vzorek v plovoucím okénku vyhledáváme odzadu. Protisměrné vyhledávání nám dovoluje dosahovat sublineární složitosti v průměrném případě – při protisměrném vyhledávání mohou na-




Obrázek 5.1: Faktorový přístup vyhledávání

stat situace, že přeskočíme několik ještě nezkontrolovaných znaků aktuálního okénka, protože jsme si jisti, že se na nich vzorek nemůže vyskytovat.

*Faktorový přístup* určuje způsob jakým je posunováno plovoucí okénko. Myšlenka spočívá v tom, že hledáme faktory vzorku v aktuálním okénku. Předpokládejme, že jsme přečetli faktor $u$ vzorku $P$, a že se nám na dalším písmeni již faktor nepodařilo přečíst, tedy $\sigma u$ už faktorem vzorku $P$ není. Potom můžeme tvrdit, že vzorek $P$ určitě neobsahuje faktor $\sigma u$, a proto můžeme bezpečně posunout plovoucí okénko za výskyt písmene $\sigma$.

Obrázek 5.1 ukazuje aplikaci tohoto přístupu s použitím protisměrného vyhledávání pro vyhledávání vzorku „deed“ v textu.

Předpokladem pro úspěšnou aplikaci faktorového přístupu je skutečnost, že umíme rychle rozpoznat situaci, že nalezený faktor již není faktorem vzorku $P$. K tomuto účelu lze použít mnoho struktur, ze kterých nás zajímají *sufixový automat* a *faktorové orákulum*.

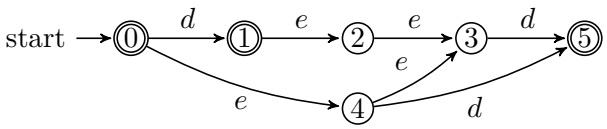
## 5.3 Sufixový automat

Sufixový automat (též nazývaný DAWG) postavený pro řetězec $s$ je deterministický automat, u kterého posloupnost přechodů zakončená v koncovém stavu odpovídá nějakému sufixu řetězce $s$.

Pokud sufixový automat upravíme tak, že všechny jeho stavy označíme jako koncové, bude potom rozpoznávat všechny faktory řetězce $s$.

Rozpoznání, zda-li je řetězec $u$ sufixem (nebo faktorem) řetězce $s$, lze uskutečnit v čase $\mathcal{O}(|u|)$. Zkonstruovat sufixový automat pro řetězec $s$ lze v čase $\mathcal{O}(|s|)$.

Konstrukce sufixového automatu pracuje *on-line*, tedy automat inkrementálně vytváříme postupným přidáváním znaků řetězce $s$.




Obrázek 5.2: Sufixový automat pro reverzi řetězce „*deed*“

## 5.4 Faktorové orákulum

Faktorové orákulum (z angl. factor oracle) pro řetězec $s$ je deterministický automat, který rozpoznává všechny faktory řetězce $s$ a může rozpoznat i některé řetězce, které faktorem řetězce $s$ nejsou. Je ale zaručeno, že rozpozná právě jeden řetězec délky $|s|$ a to samotný řetězec $s$.

Podobně jako u sufixového automatu, rozpoznání faktoru $u$ lze učinit v čase $\mathcal{O}(|u|)$ a zkonstruovat faktorové orákulum pro řetězec $s$ lze on-line v čase $\mathcal{O}(|s|)$.

## 5.5 Backward DAWG Matching

BDM, též nazývaný Reverse factor, je založen na použití sufixového automatu, jehož příklad vidíme na obrázku 5.2.

### Algoritmus

Nechť máme na vstupu vzorek $P = p_1p_2...p_m$ a text $T = t_1t_2...t_n$ z abecedy $\Sigma$.

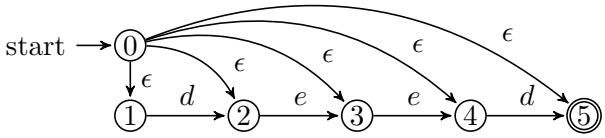
Předzpracování se skládá ze sestavení sufixového automatu $A$ pro řetězec $P^R$.

Prohledávání aktuálního okénka provádíme zprava doleva tak, že okénko zpracováváme automatem $A$. Pokud se během tohoto prohledávání ocitneme v koncovém stavu automatu $A$, který neodpovídá celému $P^R$, poznamenáme si aktuální pozici v okénku do proměnné *last*. Pozorujeme, že tato situace odpovídá nalezení zatím nejdelšího prefixu $P$, protože sufixy $P^R$ jsou reverze prefixů $P$ a tyto sufixy rozpoznáváme zprava doleva.

Prohledávání okénka může skončit dvěma způsoby:

1. Narazili jsme na symbol $\sigma$, pro který v automatu $A$ v daném stavu není definován přechod. V tomto případě posuneme okénko tak, že začíná na pozici uloženou v proměnné *last*.

2. Dosáhli jsme začátku okénka, tedy jsme našli výskyt vzorku $P$. Oznámíme tento výskyt a posuneme okénko stejným způsobem jako v předchozím případě.




Obrázek 5.3: Nedeterministický sufixový automat pro reverzi řetězce „*deed*“

## 5.6 Backward Nondeterministic DAWG Matching

BNDM funguje na podobném principu jako BDM s tím, že pouze simuluje nedeterministickou verzi sufixového automatu $A$ pro řetězec $P^R$ pomocí bitového paralelismu. Na obrázku 5.3 vidíme příklad takového automatu.

Bitový paralelismus je inherentní vlastnost bitových operací nad počítačovým slovem. V jednom slově můžeme mít zabaleno více hodnot, které lze upravovat všechny najednou pomocí bitových operací. Typickou šířkou slova je 32 nebo 64 bitů.

Myšlenka algoritmu spočívá v tom, že při zpracování aktuálního okénka, kdy zprava doleva načítáme řetězec $u$, si budeme udržovat pozice všech výskytů řetězce $u$ ve vzorku $P$. Umožní nám to snadno rozeznat situaci, kdy se nám podařilo načíst prefix vzorku $P$, a tedy i přesně určit, jak má být okénko posunuto.

Algoritmus předpokládá, že délka vzorku není větší než je velikost počítačového slova. Pokud by byl vzorek delší, jedním z možných řešení je vyhledávat pouze prefix vzorku o délce počítačového slova a při nalezení výskytu tohoto prefixu spustit z pozice výskytu jiný algoritmus, který zkontroluje, zda-li byl nalezen výskyt celého vzorku.

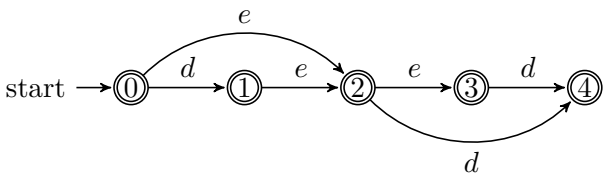
### Algoritmus

Nechť máme na vstupu vzorek $P = p_1p_2...p_m$ a text $T = t_1t_2...t_n$ z abecedy $\Sigma$. Definujme typ bitové masky $D = d_md_{m-1}...d_1$ jako posloupnost bitů, kde vlevo je nejvýznamnější bit, vpravo je nejméně významný bit.

Předzpracování se skládá z výpočtu tabulky $B$, ve které máme pro každý symbol $\sigma$ abecedy $\Sigma$ uloženou bitovou masku $B_\sigma$. V bitové masce $B_\sigma$ je bit $b_{m-i+1}$ nastaven, pokud se symbol $\sigma$ vyskytuje ve vzorku $P$ na pozici $i$ pro $i \in \{1, ..., m\}$.

Aktuální okénko zpracováváme zprava doleva. Nejprve provedeme inicializaci proměnných: bitová maska $S$ naplněná samými jedničkami reprezentuje všechny výskyty aktuálně zpracovávaného řetězce $u$ (v inicializaci je $u$ prázdným řetězcem, tedy se ve vzorku vyskytuje všude), proměnnou $last = m + 1$ reprezentující pozici posledního nálezu prefixu $P$ v aktuálním okénku.

Nechť zpracováváme symbol $\sigma$ a v proměnné *pos* máme uloženou aktuální pozici v okénku. Bitovou masku $S$ aktualizujeme výrazem $S = S \ \& \ B_\sigma$.




Obrázek 5.4: Faktorové orákulum pro reverzi řetězce „*deed*“

Otestujeme, zda-li je nejvýznamnější bit $S$ nastaven, pokud ano, nastávají dva případy:

1. Pozice *pos* odpovídá začátku okénka, tato situace odpovídá nalezení výskytu celého vzorku, tedy ohlašujeme výskyt na pozici *pos*.

2. Pozice *pos* neodpovídá začátku okénka, tato situace odpovídá nalezení prefixu vzorku $P$ začínajícího na pozici *pos*. Proměnnou *last* aktualizujeme na pozici dle *pos*.

Následně je $S$ bitově posunuta doleva o jeden bit, čímž končí zpracování symbolu $\sigma$.

Pokud po zpracování symbolu $\sigma$ je bitová maska $S$ nulová, nalezli jsme faktor, který není faktorem vzorku $P$. V důsledku toho ukončujeme zpracovávání symbolů v aktuálním okénku a okénko je posunuto tak, že jeho začátek odpovídá pozici uložené v proměnné *last*.

## 5.7 Backward Oracle Matching

BOM pro vyhledávání faktorů ve vzorku používá faktorové orákulum, jehož příklad vidíme na obrázku 5.4.

### Algoritmus

Nechť máme na vstupu vzorek $P = p_1p_2...p_m$ a text $T = t_1t_2...t_n$ z abecedy $\Sigma$.

Předzpracování se skládá z konstrukce faktorového orákula $A$ pro řetězec $P^R$.

Prohledávání aktuálního okénka provádíme zprava doleva tak, že okénko zpracováváme orákulem $A$.

Pokud při zpracovávání narazíme na symbol $\sigma$, pro který v orákulu $A$ v daném stavu není definován přechod, znamená to, že jsme nalezli řetězec, který určitě není faktorem vzorku $P$. Ukončujeme zpracování aktuálního okénka a posouváme ho za výskyt symbolu $\sigma$.

Pokud zpracujeme celé okénko, nalezli jsme výskyt vzorku $P$ na pozici začátku okénka. Ohlásíme tento výskyt a okénko posuneme o jednu pozici vpravo.

## 5.8 Implementace v Automatové knihovně

On-line konstrukce sufixového automatu a faktorového orákula spolu s algoritmy BDM, BNDM a BOM jsou implementovány v knihovně `alib2algo`. Lze je spouštět programem `astringology2`.

Kapitola 6

# Testování

V této kapitole stručně popisujeme postupy při testování nových rozšíření a implementovaných algoritmů.

## 6.1 Měření v Automatové knihovně

V měřícím rozšíření se testuje správnost výsledků měření jednotlivých domén. Testy jsou složeny z posloupnosti otevírání, zanořování a zavírání oblastí, ve kterých je simulován předmět měření.

Časová doména je testována pozastavením vykonávání aktuálního vlákna procesu skrze volání funkce `std::this_thread::sleep_for`, které předáváme dobu pozastavení v řádech desítek milisekund. Test dovoluje jistou odchylku od očekávaných hodnot z důvodu výkyvů doby pozastavení způsobené operačním systémem.

Paměťová doména je testována prováděním alokací a dealokací různě velkých paměťových bloků, výsledky měření jsou porovnány s referenčními hodnotami.

Čítačová doména je obdobně testována inkrementováním a dekrementováním různých čítačů s následným porovnáním s referenčními výsledky.

## 6.2 Automatizované měření

Testujeme možnost vytvářet objekty sdílené paměti a generování náhodných názvů souborů.

## 6.3 Algoritmy pro řetězcové vyhledávání

U algoritmů pro konstrukci sufixového automatu a faktorového orákula jsou ručně připraveny referenční automaty. Testujeme rovnost mezi zkonstruovanými automaty a jejich referenčními protějšky.

Algoritmy BDM, BNDM a BOM testujeme na několika ručně vytvořených testovacích textech a vzorcích. Dále je testujeme na náhodném řetězci i vzorku, jejich výsledky porovnáváme s referenčním naivním algoritmem.

Kapitola 7

# Experimentální měření

V této kapitole prezentujeme výsledky experimentálního měření provedených v Automatové knihovně a v nástroji SMART.

## 7.1 Datasety

Využili jsme volně dostupného korpusu v nástroji SMART a vytvořili čtyři datasety: `bible_1M`, `genom_1M`, `rand32_1M` a `rand250_1M`.

- Dataset `bible_1M` je anglický překlad bible. Jedná se o strukturovaný text s abecedou o velikosti 60 symbolů, který dobře vystihuje vlastnosti přirozeného textu.
- Dataset `genom_1M` je sekvence DNA bakterie *E. coli*. DNA kódujeme čtyřmi symboly. Tento dataset je typickým zástupcem textů zpracovávaných v bioinformatice.
- Datasety `rand32_1M` a `rand250_1M` jsou náhodně vygenerované texty. Pravděpodobnosti výskytů jednotlivých symbolů jsou z rovnoměrného rozdělení. Velikost abeced je 32 a 250 symbolů.

Všechny datasety obsahují jeden milion znaků. Na každém datasetu jsme prováděli měření pro náhodné vzorky délky $\{2, 4, 8, 16, \ldots, 1024, 2048, 4096\}$. Náhodný vzorek je vždy volen jako nějaký podřetězec daného datasetu (tedy je zaručen alespoň jeden výskyt).

## 7.2 Metodika

Všechna měření jsou vykonávána na stejném nezatíženém stroji.

Automatovou knihovnu lze zkompilovat v módech *release* a *debug*. Mód release používá všechny dostupné optimalizace kompilátoru, mód debug nepoužívá žádné. Čas měříme v realease módu a paměť v debug módu. Pro daný

dataset a danou délku vzorku vygenerujeme 10 instancí náhodných vzorků. Pro každou z těchto instancí opakujeme měření třikrát. Výsledek měření pro daný dataset a danou délku vzorku je potom průměr z 30 vykonaných měření.

V nástroji SMART je pro daný dataset a danou délku vzorku vygenerováno 500 instancí náhodných vzorků. Výsledkem je průměr jednotlivých měření.

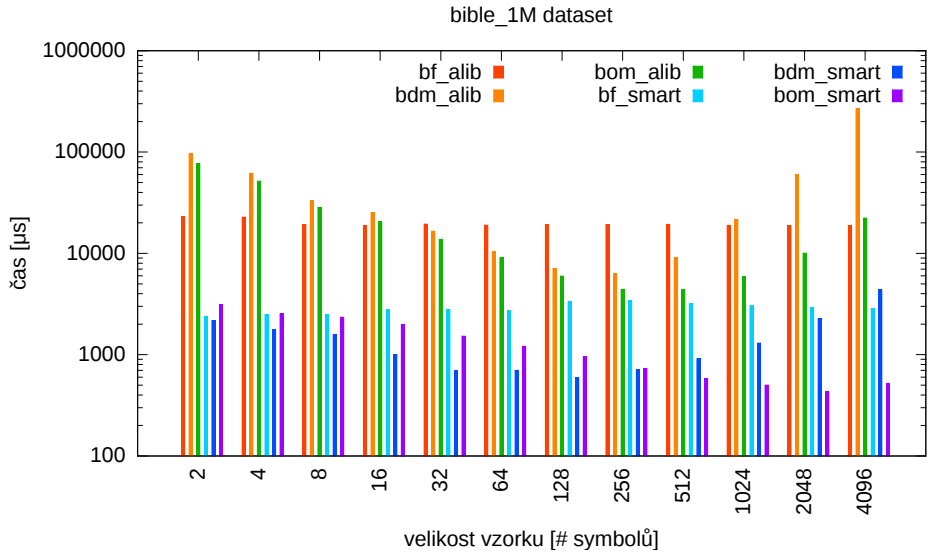
Náhodné instance vzorků a opakování měření používáme pro redukci datové citlivosti a nepřesností měření. Rozdíl v počtu měřených instancí mezi Automatovou knihovnou a nástrojem SMART je z výkonnostních důvodů.

## 7.3 Měření

V této sekci budeme interpretovat výsledky měření z datasetu `bible_1M`. Kompletní výsledky z tohoto datasetu jsou k dispozici v příloze C, výsledky ze všech datasetů jsou k dispozici v přiloženém CD.

### 7.3.1 Čas

Při srovnání Automatové knihovny a nástroje SMART se očekávalo, že SMART bude řádově rychlejší, což se i potvrdilo. V grafu 7.1 je vidět srovnání dob běhů algoritmů BF, BDM a BOM. Lze pozorovat jisté trendy u jednotlivých algoritmů:



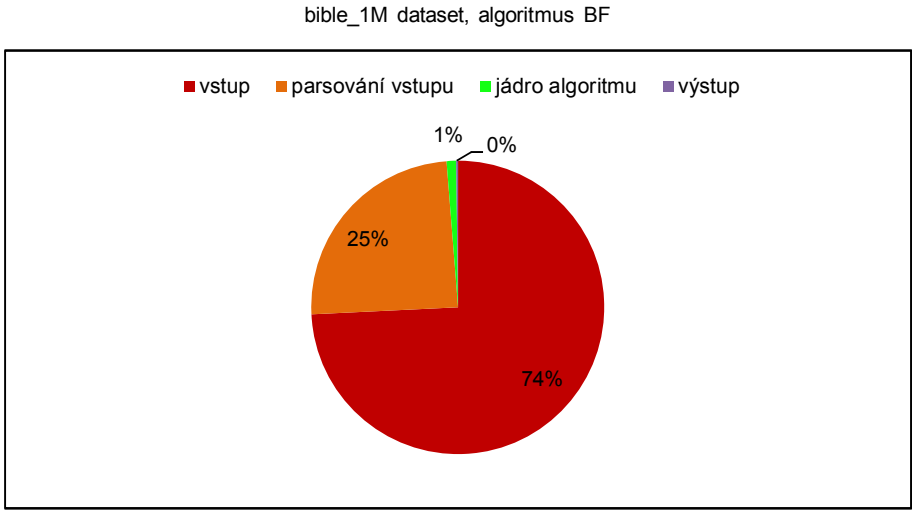

Graf 7.1: Automatová knihovna vs. SMART

Algoritmus BF (naivní algoritmus) se u obou přes všechny vzorky pohybuje pořád na stejné časové hladině.

Algoritmus BDM u obou dosahuje nejvyšší výkonnosti okolo délek vzorku $128, 256, 512$. Pro malé vzorky je znatelná režie spojená se spouštěním automatu, pro velké vzorky se projeví delší doba konstrukce sufixového automatu.

Algoritmus BOM u SMARTu vykazuje klesající tendenci dob běhů, což je očekávané chování vzhledem k tomu, že konstrukce faktorového orákula je znatelně jednodušší než konstrukce sufixového automatu. U Automatové knihovny toto nepozorujeme.

Časy Automatové knihovny v grafu 7.1 jsou izolované doby běhu jader algoritmů, tedy bez načítání a parsování vstupu a generování výstupu. Graf 7.2 ukazuje podíl běhu výpočetního jádra algoritmu BF na jeho celkovém běhu. Vidíme, že celé zpracování vstupu zabírá 99 % celkového času běhu. V našem případě je doba výstupu téměř nulová, protože nenastává mnoho výskytů náhodného vzorku v textu.




Graf 7.2: Poměr dob běhu fází algoritmu BF

### Závěr

Automatová knihovna platí daň za její obecnost, která se negativně popisuje na její výkonnosti. Z hlediska doby běhů algoritmů je největší překážkou komplexní objektový model a používání kontejnerů STL.

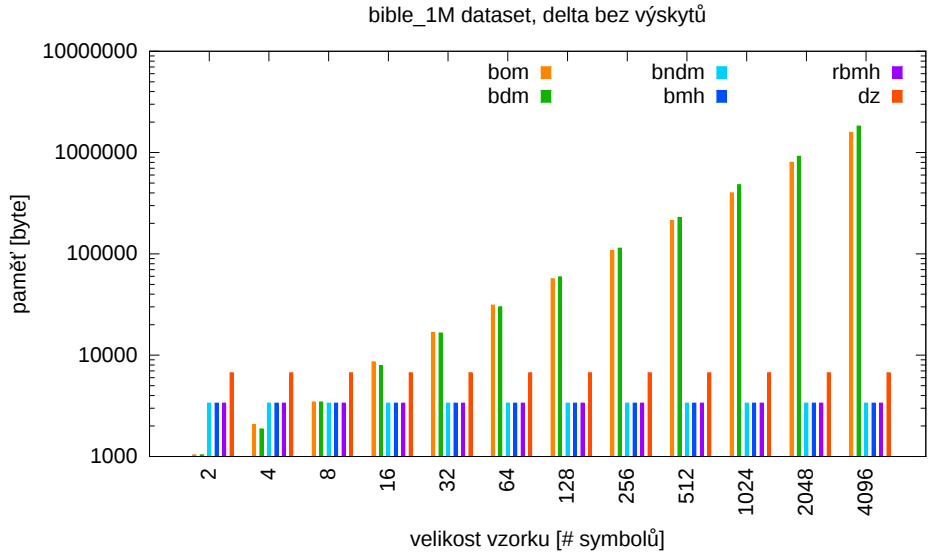
Zdaleka největším problémem Automatové knihovny je ale načítání a parsování vstupu. Je to důsledek použitého komunikačního protokolu, který je navržen tak, aby byl čitelný pro obyčejného uživatele a odpovídal matematickému zápisu. Z toho důvodu je poněkud neúsporný a jeho přečtení a zparsování je velmi drahá operace.

Na poli textového vyhledávání se Automatová knihovna již z principu nemůže rovnat nástroji SMART, který podporuje pouze tuto jednu oblast. Vý-

početní doba jádra algoritmu je v Automatové knihovně pomalejší řádově stokrát. Na druhou stranu ale pozorujeme, že Automatová knihovna zachovává trendy ve výkonnosti algoritmů na různých vstupních datech, tedy ji lze použít pro porovnávání algoritmů v ní implementovaných z hlediska času.

### 7.3.2 Paměť

Začneme opět s výsledky pro izolovaná výpočetní jádra algoritmů. Graf 7.3 ukazuje objem alokované paměti výpočetními jádry algoritmů při jejich běhu. Hodnoty v grafu nezahrnují paměť alokovanou pro report výsledků. Ukazují se vlastnosti paměťových složitostí algoritmů: u BNDM, BMH, RMBH a DZ paměť závisí pouze na velikosti abecedy, u BOM a BDM paměť závisí na velikosti vzorku.




Graf 7.3: Paměť alokovaná při běhu jader algoritmů

Konstrukce automatů pro algoritmy BOM a BDM vykazují lineární růst použité paměti vzhledem k velikosti vzorku, což odpovídá jejich teoretické asymptoticky lineární paměťové složitosti. Z naměřených dat lze vyčíst, že multiplikativní konstanta tohoto růstu je v průměru 455.

Při zkoumání paměťové složitosti celého běhu algoritmu (včetně načtení a zpracování vstupu a výstupu), dojdeme k podobnému pozorování jako u časové složitosti: paměťová složitost jádra algoritmu je v porovnání se spotřebou paměti během načtení a zpracování vstupu zanedbatelná. U datasetu `bible_1M` dosahujeme nejvyšší spotřeby 195 `MB` paměti. Mimo experiment se uskutečnilo měření na datasetech o dvou milionech a čtyřech milionech symbolů a spotřeba paměti se vyšplhala na 412 `MB` a 798 `MB`. Z toho lze vyvodit, že spotřeba paměti s velikostí datasetu roste lineárně. Pro zpracování jednoho

symbolu, který v surové podobě má maximálně 4 byty dle kódování, je potřeba v Automatové knihovně v jeden okamžik zhruba 200 bytů.

#### Závěr

Z experimentu vyplývá, že zpracovávání velkých datasetů Automatovou knihovnou se může ukázat jako problematické z důvodu velké paměťové náročnosti. Vyřešením problémů s pamětí by mohlo mít pozitivní dopad i na časovou složitost, protože alokování paměti (zejména po malých částech) je drahá operace.

# Závěr

Hlavním cílem této práce bylo navrhnout a implementovat rozšíření Automatové knihovny pro měření algoritmem spotřebovaného času a paměti spolu s čítáním obecných událostí nastávajících při běhu algoritmu. Dalším cílem bylo nastudovat a implementovat algoritmy pro řetězcového vyhledávání. Posledním cílem bylo experimentálně proměřit algoritmy Automatové knihovny a srovnat Automatovou knihovnu s nástrojem SMART z hlediska výkonnosti.

Všech cílů bylo úspěšně dosaženo. Měření automatizujeme programem `ameasure2`, výsledky zpracováváme programem `ameasurep2`.

Nově implementovanými algoritmy jsou Backward DAWG Matching, Backward Nondeterministic DAWG Matching a Backward Oracle Matching, pro jejichž potřeby byly ještě implementovány algoritmy pro on-line konstrukci sufixového automatu a faktorového orákula.

Experimentální měření ukázalo, že Automatová knihovna se potýká s výkonnostními problémy jak časovými, tak paměťovými při zpracování většího objemu dat. Nicméně se ale ukázalo, že algoritmy implementované v Automatové knihovně si uchovávají své teoretické vlastnosti, tedy je lze porovnávat. V porovnání s nástrojem SMART se ukazuje, že časová i paměťová složitost algoritmů Automatové knihovny jsou zatížené vysokou multiplikativní konstantou.

Do budoucna by bylo dobré přidat lepší možnosti agregování výsledků, pokročilejší nástroje pro zpracování výsledků a obecně nová rozšíření vylepšit z hlediska uživatelské zkušenosti.

O výsledky této práce se bude možno opřít při řešení výkonnostních problémů Automatové knihovny nebo při optimalizování stávajících implementací algoritmů.

# Literatura


[1] Žák, M.: *Automatová knihovna – vnitřní a komunikační formát.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[2] Faro, S.; Lecroq, T.: *SMART – String matching research tool* [online]. [cit. 2015-12-11]. Dostupné z: `http://www.dmi.unict.it/~faro/smart/`

[3] Rybola, Z.: *Software Engineering 1, lecture 8. Database model, GRASP, GoF* [online]. Praha, České vysoké učení technické v Praze, Fakulta informač-ních technologií [cit. 2016-05-05]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-SI1/_media/en/lectures/lecture8.pdf`

[4] *C++ reference* [online]. [cit. 2016-05-05]. Dostupné z: `http://en.cppreference.com/w/`

[5] Meyers, S.: *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition).* Addison-Wesley Professional, 2005, ISBN 0321334876.

[6] Kotal, V.; Pechanec, J.; Beran, M.: *Programování v UNIXu* [online]. Praha, SISAL MFF UK [cit. 2016-05-05]. Dostupné z: `http://mff.devnull.cz/pvu/slides/programovani-v-unixu.pdf`

[7] *Linuxová manuálová stránka: FORK(2)* [online]. [cit. 2016-05-05]. Dostupné z: `http://man7.org/linux/man-pages/man2/fork.2.html`

[8] Navarro, G.; Raffinot, M.: *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences.* New York, NY, USA: Cambridge University Press, 2002, ISBN 0-521-81307-7, 9780521813075.

[9] Holub, J.: *Automaty a gramatiky, přednáška 1. Základní pojmy* [online]. Praha, České vysoké učení technické v Praze, Fakulta informač-ních technologií [cit. 2016-05-05]. Dostupné z: `https://edux.fit.cvut.cz/courses/BI-AAG/lectures/01/start`

[10] Charras, C.; Lecroq, T.: *ESMA – Exact string matching algorithms* [online]. [cit. 2016-05-05]. Dostupné z: `http://www-igm.univ-mlv.fr/~lecroq/string/`

Příloha **A**

# Seznam použitých zkratek

**Obecné zkratky**

**V/V** Vstupně výstupní operace

**API** Application programming interface

**XML** Extensible markup language

**HTML** HyperText markup language

**CSV** Comma separated values

**CPU** Central processing unit

**STL** Standard template library

**GCC** GNU compiler collection

**DAWG** Directed acyclic word graph

**Zkratky použitých algoritmů**

**BF** Brute force

**BDM** Backward DAWG matching

**BNDM** Backward nondeterministic DAWG matching

**BOM** Backward oracle matching

**BM** Boyer-Moore

**BMH** Boyer-Moore-Horspool

**RBMH** Reversed Boyer-Moore-Horspool

**DZ** Deadzone

Příloha B

# Ukázky

## B.1 Výsledek měření Automatové knihovny

```
<?xml version="1.0"?>
<MeasurementResults>
  <MeasurementFrame>
    <Name>Root</Name><Type>ROOT</Type>
    <SubFrames>
      <MeasurementFrame>
        <Name>Overal</Name><Type>OVERALL</Type>
        <TimeData>
          <Duration>666</Duration>
          <InFrameDuration>666</InFrameDuration>
        </TimeData>
        <MemoryData>
          <StartHeapUsage>0</StartHeapUsage>
          <EndHeapUsage>0</EndHeapUsage>
          <HighWatermark>1337</HighWatermark>
          <InFrameHighWatermark>1337</InFrameHighWatermark>
        </MemoryData>
        <CounterData>
          <Counters>
            <Counter>
              <Name>testInc</Name><Value>5</Value>
            </Counter>
          </Counters>
          <InFrameCounters>
            <Counter>
              <Name>testInc</Name><Value>5</Value>
            </Counter>
          </InFrameCounters>
        </CounterData>
        <SubFrames/>
      </MeasurementFrame>
    </SubFrames>
  </MeasurementFrame>
</MeasurementResults>
```

## B.2 Konfigurační soubor automatizovaného měření

```
<?xml version="1.0"?>
<MeasurementProvisioner>
    <Environment>
        <WorkingDirectory>~/alib_binaries</WorkingDirectory>
        <PipelineIterations>10</PipelineIterations>
    </Environment>
    <InputData>
    <InputBatch>
        <File id="1" alias="bible_1M">~/alib_corpora/
            alib_bible_1M.xml</File>
        <Generator id="2" alias="pat_4096" count="10">./arand2 -t
             SST --length 4096 -i $1</Generator>
    </InputBatch>
    <InputBatch dependency="true">
        <File id="1" alias="genom">~/alib_corpora/alib_genom_
            [1-4]M.xml</File>
        <File id="2" alias="DNA_pat" count="4">~/alib_corpora/
            DNA_pat.xml</File>
    </InputData>
    <Pipelines>
        <Pipeline>
            <Command alias="bdm" >./astringology2 -a
                backwardDAWGMatching -s $1 -p $2</Command>
        </Pipeline>
        <Pipeline>
            <Command measure="false">cat $1</Command>
            <Command alias="bndm">./astringology2 -a
                backwardDAWGMatching -p $2</Command>
        </Pipeline>
    </Pipelines>
</MeasurementProvisioner>
```

## B.3 Výsledek automatizovaného měření

```
<?xml version="1.0"?>
<MeasurementProvisionerResults>
  <MeasurementProvisionerResult>
    <Inputs>
      <Input id="1">bible_1M</Input>
      <Input id="2">pat_4096</Input>
    </Inputs>
    <Pipelines>
      <Pipeline>
        <PipelineStatus>
          <ExitCode>0</ExitCode>
          <ErrorOrigin/>
          <ErrorValue/>
        </PipelineStatus>
        <Commands>
          <Command>
            <Name>bndm</Name>
            <MeasurementResults>
              <MeasurementFrame>
                <Name>Root</Name>
                <Type>ROOT</Type>
                <SubFrames>
                  <MeasurementFrame>
                    <Name>Overal</Name>
                    <Type>OVERALL</Type>
                    <TimeData>
                      <Duration>3864285</Duration>
                      <InFrameDuration>81</InFrameDuration>
                    </TimeData>
                    <MemoryData>
                      <StartHeapUsage>0</StartHeapUsage>
                      <EndHeapUsage>0</EndHeapUsage>
                      <HighWatermark>206323504</HighWatermark>
                      <InFrameHighWatermark>0</
                          InFrameHighWatermark>
                    </MemoryData>
                    <CounterData>
                      <Counters/>
                      <InFrameCounters/>
                    </CounterData>
                    <SubFrames/>
                  </MeasurementFrame>
                </SubFrames>
              </MeasurementFrame>
            </MeasurementResults>
          </Command>
        </Commands>
      </Pipeline>
    </Pipelines>
  </MeasurementProvisionerResult>
</MeasurementProvisionerResults>
```

Příloha C

# Výsledky měření

## C.1 Automatová knihovna

Tabulka C.1: Celková doba běhu algoritmů [$\mu$s]

| \|P\| | BF | BOM | BDM | BNDM | BMH | RBMH | DZ |
|---|---|---|---|---|---|---|---|
| 2 | 1628444 | 1683020 | 1702261 | 1661897 | 1663500 | 1659348 | 1695423 |
| 4 | 1605218 | 1663962 | 1657995 | 1596875 | 1601305 | 1605685 | 1658774 |
| 8 | 1357548 | 1377133 | 1373480 | 1359851 | 1364797 | 1357442 | 1365455 |
| 16 | 1356320 | 1359392 | 1361281 | 1350781 | 1347970 | 1349055 | 1357051 |
| 32 | 1372533 | 1380349 | 1379675 | 1407890 | 1415943 | 1367103 | 1373956 |
| 64 | 1352527 | 1345483 | 1346091 | 1344451 | 1342779 | 1341387 | 1348758 |
| 128 | 1352317 | 1340922 | 1345252 | 1341746 | 1341307 | 1341232 | 1345777 |
| 256 | 1355412 | 1340555 | 1340574 | 1344012 | 1342529 | 1343347 | 1348720 |
| 512 | 1371312 | 1351327 | 1361760 | 1362002 | 1355573 | 1352317 | 1379984 |
| 1024 | 1355655 | 1343440 | 1361192 | 1344795 | 1341666 | 1341618 | 1349109 |
| 2048 | 1356408 | 1345525 | 1397714 | 1345114 | 1341493 | 1338268 | 1346188 |
| 4096 | 1360838 | 1364064 | 1606631 | 1349641 | 1342632 | 1340807 | 1352880 |

Tabulka C.2: Doba běhu výpočetního jádra algoritmů [$\mu$s]

| \|P\| | BF | BOM | BDM | BNDM | BMH | RBMH | DZ |
|---|---|---|---|---|---|---|---|
| 2 | 23148 | 76732 | 95564 | 54744 | 55797 | 54501 | 91642 |
| 4 | 22737 | 51578 | 61610 | 29955 | 29534 | 29135 | 47911 |
| 8 | 19212 | 28372 | 33148 | 15155 | 14780 | 13954 | 26009 |
| 16 | 18770 | 20469 | 25125 | 11936 | 10315 | 11625 | 19662 |
| 32 | 19430 | 13732 | 16556 | 8014 | 8691 | 8741 | 15856 |
| 64 | 18953 | 8989 | 10337 | 7719 | 7070 | 6833 | 13124 |
| 128 | 19229 | 5950 | 7101 | 7815 | 5813 | 6045 | 12672 |
| 256 | 19192 | 4391 | 6300 | 7793 | 4871 | 4914 | 11879 |
| 512 | 19302 | 4377 | 9003 | 7953 | 4336 | 3999 | 12034 |
| 1024 | 18856 | 5899 | 21515 | 7903 | 3749 | 3427 | 11175 |
| 2048 | 18720 | 10044 | 59464 | 7745 | 3196 | 2839 | 11349 |
| 4096 | 18869 | 22273 | 266009 | 7817 | 3236 | 2845 | 12461 |

Tabulka C.3: Paměť alokovaná výpočetním jádrem algoritmů bez výskytů [byte]

| \|P\| | BF | BOM | BDM | BNDM | BMH | RBMH | DZ |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 1040 | 1040 | 3360 | 3360 | 3360 | 6720 |
| 4 | 0 | 2064 | 1872 | 3360 | 3360 | 3360 | 6720 |
| 8 | 0 | 3456 | 3456 | 3360 | 3360 | 3360 | 6720 |
| 16 | 0 | 8616 | 7904 | 3360 | 3360 | 3360 | 6720 |
| 32 | 0 | 16696 | 16560 | 3360 | 3360 | 3360 | 6720 |
| 64 | 0 | 31176 | 30104 | 3360 | 3360 | 3360 | 6720 |
| 128 | 0 | 56984 | 59376 | 3360 | 3360 | 3360 | 6720 |
| 256 | 0 | 107816 | 113752 | 3360 | 3360 | 3360 | 6720 |
| 512 | 0 | 213400 | 228776 | 3360 | 3360 | 3360 | 6720 |
| 1024 | 0 | 400664 | 481816 | 3360 | 3360 | 3360 | 6720 |
| 2048 | 0 | 801752 | 921288 | 3360 | 3360 | 3360 | 6720 |
| 4096 | 0 | 1582872 | 1823848 | 3360 | 3360 | 3360 | 6720 |

## C.2 Nástroj SMART

Tabulka C.4: Celková doba běhu algoritmů [$\mu$s]

| \|P\| | BF | BOM | BDM | BNDM | BM |
|---|---|---|---|---|---|
| 2 | 2390 | 3120 | 2160 | 2260 | 4520 |
| 4 | 2490 | 2530 | 1770 | 1740 | 2570 |
| 8 | 2480 | 2340 | 1580 | 1290 | 1580 |
| 16 | 2780 | 1990 | 1000 | 900 | 1170 |
| 32 | 2780 | 1520 | 690 | 560 | 850 |
| 64 | 2710 | 1210 | 700 | 560 | 670 |
| 128 | 3330 | 950 | 590 | 620 | 710 |
| 256 | 3380 | 730 | 710 | 680 | 630 |
| 512 | 3160 | 580 | 910 | 690 | 600 |
| 1024 | 3040 | 500 | 1300 | 620 | 540 |
| 2048 | 2910 | 430 | 2250 | 620 | 520 |
| 4096 | 2830 | 520 | 4340 | 630 | 500 |

Příloha D

# Uživatelská příručka

## D.1 Požadavky

Pro kompilaci zdrojového kódu Automatové knihovny je potřeba mít kompilátory `g++` nebo `clang++` podporující jazyk C++11. Dále je potřeba program `make` a nainstalovaná knihovna `libxml2`.

## D.2 Instalace

Kompilaci Automatové knihovny spouštíme příkazem `make debug` nebo `make release` v kořenovém adresáři projektu. Pro kompilaci přes `clang++` je třeba použít příkaz `CXX=clang++ make debug` (obdobně pro `release`). Výsledné binární soubory budou ve složkách `bin-debug` nebo `bin-release`.

## D.3 Měření v Automatové knihovně

U programů binární knihovny zapínáme měření přepínačem `-m`. Výstup měření lze přesměrovat použitím shellového přesměrování:

```
$ ./astringology2 -a factorOracleAutomaton -i vstup.xml \
      -m 5> vysledky_mereni.xml
```

## D.4 Automatizované měření

Program `ameasure2` očekává na vstupu pouze konfigurační soubor. Předáváme ho buď jako cestu k němu skrze argument příkazové řádky, nebo jeho obsah dáme k dispozici na standardní vstup. Příklad použití:

```
$ ./ameasure2 konfigurace.xml

$ cat konfigurace.xml | ./ameasure2
```

## D.5 Zpracování výsledků měření

Program `ameasurep2` očekává na vstupu pouze výsledky automatizovaného měření, které předáváme stejně jako v případě programu `ameasure2`. Program `ameasurep2` konfigurujeme přepínači:

- `-o, -output`
- `-e, -engine`
- `-engineAttr`
- `-filterFrameType`
- `-filterFrameName`
- `-filterCommandName`
- `-aggregate`

Pro jejich popis viz kapitolu 4 o zpracování výsledků. Příklady použití:

```
$ ./ameasurep2 --output csv --aggregate \
      --engine memory --engineAttr highWatermark \
      --filterFrameType MAIN \
      --filterFrameType OVERALL \
      --filterCommandName bndm \
       vysledky_auto_mereni.xml

$ ./ameasurep2 --aggregate vysledky_auto_mereni2.xml
```

Příloha **E**

# Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `automata-library` ..... zdrojové kódy Automatové knihovny
- `corpora` ..... datasety pro experimentální měření
- `measurement-results` ..... výsledky experimentálního měření
- `text`
  - `BP_Cerveny_Radovan_2016.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojové soubory práce ve formátu LaTeX