TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

IT40LT

Bahdan Yanovich 134456IAPB

Optimisation of Symbolic Automata Based Regular Expression Library SRM Using SIMD Intrinsics

Bachelor's thesis

Supervisor: Juhan-Peep Ernits
PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

IT40LT

Bahdan Yanovich 134456IAPB

Sümbolautomaatidel põhineva regulaaravaldiste teegi SRM optimeerimine SIMD käsustiku abil

Bakalaureusetöö

Juhendaja: Juhan-Peep Ernits
PhD

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, the literature and the work of others have been referenced. This thesis has not been presented for examination anywhere else.

Author: Bahdan Yanovich

14.06.2021

Abstract

To the best of my knowledge, SRM is currently the only regular expression engine based on Symbolic Automata. It has shown promising results in performance compared to state of the art regex engines.

The goal of the current thesis is to explore how SIMD intrinsics can be used to speed up finding matches in SRM from large strings. The expected outcome was 5-10% gain in performance.

The optimisations were performed in a function that was identified by profiling to be frequently called.

The experiments were performed on the English text of the Entire Project Gutenberg Works of Mark Twain, a dataset often used to compare regular expression engines. The text contains mostly ASCII symbols, but also a few extended ASCII characters. In addition similar regular expressions were created and run on the entire contents of Wikipedia written in Estonian that was provided in UTF-8 encoding.

The results of the experiments showed that with some regular expression patterns it was possible achieve a speed up of 48%. Additionally, the experiments highlighted how Intel Core I9 10th generation and AMD Threadripper 3960X processors perform differently while using SIMD instructions.

This thesis is written in English and is 33 pages long, including 5 chapters, 7 figures, and 4 tables.

Annotatsioon

SRM on teadaolevalt ainus sümbolautomaatidele tuginev regulaaravaldiste teek, mis on näidanud paljulubavaid tulemusi võrdluses alternatiivsete regulaaravaldiste teekidega.

Käesoleva bakalaureusetöö eesmärgiks on uurida, kuidas saab SIMD käsustikke kasutada SRMis vastete leidmise kiirendamiseks suurtest stringidest. Eesmärgiks seadsime 5-10%-lise kiiruse võidu.

Optimiseerimised tehti tihti väljakutsutavas funktsioonis, mis identifitseeriti profileerimise teel.

Eksperimentideks kasutati inglise keelset Projekt Gutenbergi raames koondatus Mark Twaini kogutud teoste teksti, mida kasutatakse tihti ka teiste regulaaravaldiste mootorite jõudluse uurimisel. Mark Twaini teoste inglise keelne versioon sisaldab enamasti ASCII sümboleid ja üksikuid laiendatud ASCII sümboleid. Lisaks koostasime sarnased regulaaravaldised ka eestikeelse vikipeedia korpusele, mis oli UTF-8 kodeeringus.

Eksperimentide tulemused varieerusid väikesest kiiruse kaotusest kuni 48%-lise kiiruse võiduni sõltuvalt regulaaravaldise mustrist. Katsed tõid välja ka Intel Core i9 protsessori ja AMD Threadripper 3960x protsessorite erinevused SIMD käsustike käivitamisel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 33 leheküljel, 5 peatükki, 7 joonist, 4 tabelit.

List of abbreviations and terms

ASCII American Standard Code for Information Interchange

BST Binary Search Tree

CPU Central Processing Unit

CsA Counting-Set Automata

DFA deterministic finite automata

JIT just-in-time compiler

NFA nondeterministic finite automata

ReDoS Regular expression Denial of Service

SDK Software Development Kit

SIMD Single Instruction Multiple Data

SRM Symbolic Regex Matcher

UTF-8 Universal Coded Character Set (unicode) Transformation Format – 8-bit

Contents

1 Introduction **11**

2 Optimisation **13**

2.1 Symbolic Regex Matcher (SRM) matching algorithm 13

2.2 Profiling 13

2.3 Challenges and constraints 15

2.4 Optimisation 17

3 Benchmarking **21**

3.1 Datasets 21

3.2 Benchmark setup 22

3.3 Experiment results and discussion 23

4 Related work **29**

5 Summary **30**

5.1 Future Work 30

References **32**

Appendix 1 Non-exclusive licence for reproduction and publication of a graduation thesis **34**

Appendix 2 Automata Introduction **35**

2.1 Classical Automata Theory 35

2.2 Symbolic Automata 37

2.3 Counting-Set Automata 38

Appendix 3 Acknowledgements 39

List of Figures

1 Profiling results for ”[a-z]shing” regular expression 14

2 Profiling results for ”Tom|Sawyer|Huckleberry|Finn regular expression . . 14

3 Documentation for `testC` intrinsic 16

4 State diagram of a finite automaton that accepts strings that represent 3-digit integers that are greater than or equal to 900 and divisible by 5 . . . 36

5 Regular expression ”9[0-9][05]” represented as a symbolic automaton . . 38

6 Regular expression ”k{5}” represented as a finite automaton 38

7 Regular expression ”k{5}” represented as a counting-set automaton . . . 38

List of Tables

1 Benchmark results: Mark Twain dataset on Intel processor 25

2 Benchmark results: Mark Twain dataset on AMD processor 26

3 Benchmark results: Estonian wikipedia dataset on Intel processor 27

4 Benchmark results: Estonian wikipedia dataset on AMD processor 28

1 Introduction

Regular expressions are broadly used for solving different problems related to text data processing: a regular user might need to simply *find a word* on a web page or in a document; developers of any interactive application have to *validate user input*; data specialists use them for *data scraping* and *data cleaning*. Internally they are typically based on *deterministic finite automata (DFA)*, or *nondeterministic finite automata (NFA)* or their hybrids [1]. Applying a regular expression under some circumstances might yield an *exponential-time* computation, which can cause problems in the era of big data, when the amount of data that needs to be interpreted is growing tremendously [2]. Moreover, inappropriate use of regular expressions or choosing an inefficient regular expression engine can lead to wasted time and energy or unexpected bottlenecks. Recent research shows that up to 10% of regular expressions are applied to unreliable input, which might lead to so-called *Regular expression Denial of Service (ReDoS)* [3] attacks.

When starting to process the input, in the DFA and NFA based approaches the regular expressions in string format are compiled into some internal automata representation where each transition is labelled with a single character. While such approach might be acceptable in the case of *American Standard Code for Information Interchange (ASCII)* and extended ASCII character sets[1], where the number of different characters are 127 and 255 accordingly, modern character sets such as *Universal Coded Character Set (unicode) Transformation Format – 8-bit (UTF-8)* are capable of encoding more than 10^6 characters and a blow up may occur when compiling regular expressions. To handle large alphabets, *symbolic automata* were introduced as an extension of classical automata [4], [5]. The first and, to the best of my knowledge, the only open source symbolic automata based regular expression engine *SRM* was introduced by Veanes et al. According to the authors, the overall algorithm complexity of match generation is *linear*; it supports bounded quantifiers and Unicode categories; it is safe to use in multi-threaded development environments [6]. The source code of the library is available at [7] [2].

There are various ways to optimise computations. The typical approach is to parallelise the computations to utilise multiple cores of processors. The regular expression matching algorithms are complex and do not lend themselves easily to parallelisation. One of the ways to optimise single threaded code is to use *Single Instruction Multiple Data (SIMD)*

[1] `https://en.wikipedia.org/wiki/Extended_ASCII`

[2] SRM is currently in the process of becoming a standard library in future versions of .Net SDKs. [8]

vector instructions. In .NET it is possible to use SIMD instructions via the *intrinsics* library `System.Runtime.Intrinsics`. This mechanism allows to use longer processor registers for computation to perform multiple operations in one processor cycle. So in case of applying a command to a vector of 4 numbers, instead of 4 separate cycles (load a number into the processor's register, do the operation, save the result) for each of the numbers, the operation will be applied to all four numbers once on the processor level [9], [10]. Some research show that using SIMD intrinsics in the context of regular expression might reduce the running time up to 50% in some specific cases [11].

Thus, the **goal** of the current thesis is to optimise SRM using SIMD intrinsics to enhance performance in the context of large inputs.

Research questions:

- How is it possible to apply SIMD intrinsics to SRM?
- What is the impact of applying SIMD intrinsics on the performance of SRM?

In order to achieve the goal, the process will be split into **three phases:**

1. Preparing a data set valid for searching regular expressions in large strings (as one of the possible big data scenarios) (Section 3.1)
2. Profiling and optimisation of SRM (Section 2)
3. Experiment: benchmarking with other regular expression engines (Section 3.2 - 3.3)

Validation of the potential optimisation will be done during the experiment. To validate the experiment, the results can be compared to the similar experiments found in the literature.

Expected outcomes of the thesis are:

- 5-10% gain in SRM performance
- benchmark dataset for the regular expression performance evaluation

2 Optimisation

When processing regular expressions the algorithm works in multiple stages. In the first stage the regular expression string is compiled and the resulting internal structure will constitute the inner workings of the matching algorithm. For the purpose of the current thesis we assume large input strings and thus we consider the regular expression compilation to take negligible time compared to the time required to perform the matches on the input. Thus we concentrate our optimisation efforts to the SRM matching algorithm. Profiling confirms that most of the time is spent in various steps of the SRM matching algorithm.

2.1 SRM matching algorithm

The following SRM matching algorithm as presented in [6]:

1. Initially $i = 0$ is the start position of the first symbol u_0 of u.
2. Let $i_{\text{orig}} = i$. Find the *earliest* match starting from i and $q = .{*}R$: Compute $q := \partial_{u_i} q$ and $i := i + 1$ until q is nullable. **Terminate** if no such q exists.
3. Find the *start position* for the above match closest to i_{orig}: Let $p = R^r$. While $i > i_{\text{orig}}$ let $p := \partial u_{u_i} p$ and $i := i - 1$, if p is nullable let $i_{\text{start}} := i$.
4. Fine the *end position* for the match: Let $q = R$ and $i = i_{\text{start}}$. Compute $q := \partial_{u_i} q$ and $i := i + 1$ and let $i_{\text{end}} := i$ if q is nullable; repeat until $q = \bot$
5. **Return** the match from i_{start} to i_{end}.
6. Repeat step 2 from $i := i_{\text{end}} + 1$ for the next *nonoverlapping* start position.

where: u - the input text, R - the original regex, R^r - the reversal of regex R, $\partial_x R$ - Brzozowski *x-derivative of R* [12], . - the *true* predicate, $\bot$ - the *false* predicate.

2.2 Profiling

In order to figure out what part of the library needs to be optimised, first, the library was profiled using Visual Studio 2019 Preview [13]. The Preview version of Visual Studio 2019 supports .NET 5, which is the current version of the Software Development Kit

Current View: Call Tree | Expand Hot Path | Show Hot Path

Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module	Category
katse.exe (PID: 24648)	12229 (100,00%)	0 (0,00%)	Multiple modules	
[Native]	12227 (99,98%)	307 (2,51%)	Multiple modules	IO | Networking |...
katse.Program.Main(System.String[])	11906 (97,36%)	9 (0,07%)	katse.dll	Networking | JIT |...
katse.Program.SRMDotnet5()	10074 (82,38%)	2 (0,02%)	katse.dll	JIT | File System |...
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.Matches(string, int, int, int)	9911 (81,05%)	0 (0,00%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.MatchesSafe(string, int, i...	9911 (81,05%)	6 (0,05%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.FindFinalStatePosition(...	9905 (81,00%)	7303 (59,72%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.IndexOfStartset(strin...	2409 (19,70%)	2409 (19,70%)	srm-dotnet5.dll	
0x007ffab3890128	182 (1,49%)	182 (1,49%)		
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.CreateNewTransition...	8 (0,07%)	0 (0,00%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.GetWatchdog(Micro...	2 (0,02%)	2 (0,02%)	srm-dotnet5.dll	
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.GetWatchdog(Micro...	1 (0,01%)	1 (0,01%)	srm-dotnet5.dll	
Microsoft.SRMDotnet5.Regex..ctor(string)	161 (1,32%)	4 (0,03%)	srm-dotnet5.dll	JIT | File System |...
katse.Program.LoadData()	1822 (14,90%)	2 (0,02%)	katse.dll	JIT | File System |...
System.Console.dll!0x007ffbaec14a9a	1 (0,01%)	0 (0,00%)	System.Console.dll	
System.Private.CoreLib.dll!0x007ffb129bcfa7	6 (0,05%)	0 (0,00%)	System.Private.Co...	JIT | Kernel
System.Private.CoreLib.dll!0x007ffb129d5a23	3 (0,02%)	0 (0,00%)	System.Private.Co...	JIT | Kernel
System.Private.CoreLib.dll!0x007ffb129ee4e0	2 (0,02%)	1 (0,01%)	System.Private.Co...	JIT | Kernel
System.Private.CoreLib.dll!0x007ffb129a64ae	1 (0,01%)	1 (0,01%)	System.Private.Co...	
System.Private.CoreLib.dll!0x007ffb129d563c	1 (0,01%)	0 (0,00%)	System.Private.Co...	
System.Private.CoreLib.dll!0x007ffb129d59ee	1 (0,01%)	0 (0,00%)	System.Private.Co...	
[Unwalkable]	2 (0,02%)	0 (0,00%)	[External Code]	

Figure 1. Profiling results for ”[a-z]shing” regular expression.

Current View: Call Tree | Expand Hot Path | Show Hot Path | Search

Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module	Category
katse.exe (PID: 29196)	3509 (100,00%)	0 (0,00%)	Multiple modules	
[Native]	3507 (99,94%)	258 (7,35%)	Multiple modules	UI | IO | Networki...
katse.Program.Main(System.String[])	3235 (92,19%)	3 (0,09%)	katse.dll	Networking | JIT |...
katse.Program.SRMDotnet5()	1712 (48,79%)	0 (0,00%)	katse.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.Matches(string, int, int, int)	1595 (45,45%)	1 (0,03%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.MatchesSafe(string, int, i...	1594 (45,43%)	3 (0,09%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.FindFinalStatePosition(...	1591 (45,34%)	132 (3,76%)	srm-dotnet5.dll	JIT | Kernel
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.IndexOfStartset(strin...	1453 (41,41%)	1453 (41,41%)	srm-dotnet5.dll	
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.CreateNewTransition...	3 (0,09%)	0 (0,00%)	srm-dotnet5.dll	Kernel
0x007ffab4450128	1 (0,03%)	1 (0,03%)		
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.GetWatchdog(Micro...	1 (0,03%)	1 (0,03%)	srm-dotnet5.dll	
Microsoft.SRMDotnet5.SymbolicRegex<ulong>.CreateNewTransition...	1 (0,03%)	0 (0,00%)	srm-dotnet5.dll	JIT
Microsoft.SRMDotnet5.Regex..ctor(string)	116 (3,31%)	1 (0,03%)	srm-dotnet5.dll	JIT | Kernel
System.Collections.Generic.List<Microsoft.SRMDotnet5.Match>.MoveNext()	1 (0,03%)	1 (0,03%)	System.Private.Co...	
katse.Program.LoadData()	1519 (43,29%)	2 (0,06%)	katse.dll	Networking | JIT |...
System.Console.dll!0x007ffb98ee4a9a	1 (0,03%)	0 (0,00%)	System.Console.dll	Kernel
System.Private.CoreLib.dll!0x007ffb0f0fcfa7	8 (0,23%)	0 (0,00%)	System.Private.Co...	JIT | Kernel
System.Private.CoreLib.dll!0x007ffb0f115a23	3 (0,09%)	0 (0,00%)	System.Private.Co...	JIT
System.Private.CoreLib.dll!0x007ffb0f1159ee	2 (0,06%)	0 (0,00%)	System.Private.Co...	Kernel
System.Private.CoreLib.dll!0x007ffb0f12e4e0	1 (0,03%)	0 (0,00%)	System.Private.Co...	JIT
[Unwalkable]	2 (0,06%)	0 (0,00%)	[External Code]	

Figure 2. Profiling results for ”Tom|Sawyer|Huckleberry|Finn regular expression.

(SDK) at the time of writing the thesis. SIMD intrinsics were introduced into the .NET SDK starting from .NET Core version 3.0, but as the just-in-time compiler (JIT) of the runtime is under constant development, using the current version of SDK was considered appropriate. I used the dataset and regexes from the chosen benchmark (See dataset subsection). The results of profiling for two different kinds of regular expressions (without sets and with sets) are shown on figures 1 and 2. It corresponds to the step 2 of the matching algorithm.

As one might notice the most time-consuming (Central Processing Unit (CPU) consuming) part was `SymbolicRegex<S>.IndexOfStartset()` method in

SymbolicRegexMatcher.cs file. It took $19 - 41\%$ of the runtime. See the original method implementation in the listing 1.

```
int IndexOfStartset(string input, int i)
{
    int k = input.Length;
    while (i < k)
    {
        var input_i = input[i];
        if (input_i < A_StartSet.precomputed.Length ?
            A_StartSet.precomputed[input_i] :
                A_StartSet.bst.Find(input_i) == 1)
            break;
        else
            i += 1;
    }
    if (i == k)
        return -1;
    else
        return i;
}
```

Listing 1. Original implementation of IndexOfStartset() method.

2.3 Challenges and constraints

While trying to apply SIMD intrinsics on the mentioned method, I faced a few challenges regarding both SRM and SIMD intrinsics:

a As one might see, the operations in the loop are not identical for each loop iteration and depend on the input: in some cases precomputed array is used, in other cases – bst binary search tree.

b SIMD intrinsics are designed to work on primitive data types such as integers and floating point numbers that are processed as vectors. Unfortunately, there is no direct way to apply any of the instructions to variables of type String. .NET provides an approach called Span<T> that "provides a type- and memory-safe representation of a contiguous region of arbitrary memory"[1]. Using Span<T> it is possible to ac-

[1] https://docs.microsoft.com/en-us/dotnet/api/system.span-1?view=net-5.0

cess .NET strings as arrays of characters. Due to the fact that strings are immutable in .NET, it is necessary to use the read only version ReadOnlySpan<T>.

c In order to use SIMD intrinsic instructions, it is necessary to work with arrays of ushort rather than arrays of char, i.e. it is necessary to cast the span to ushort array. The number of applicable intrinsic instructions is quite limited. Unlike instructions availble for vectors of type sbyte, long, short, int data types, there is no strict comparison operator (e.g. CompareGreaterThan or CompareLessThan) for vectors of ushort.

d The output of the intrinsics is usually vectors. It is not convinient when it comes to using the results in conditionals. It would require extra Equals comparison with a specific reference value (e.g. 111...111), which be preferable to avoid.

e There are some intrinsics that return boolean values. However, their documentation is not straightforward. .NET documentation provides only with the instrinsic instruction. The documentation of the intrinsics shows bitwise pseudocode and sometimes it is not obvious what problem the intrinsics solve. See an example of documentation of testC intrinsic on the figure 3.

TestC(Vector256<UInt16>, Vector256<UInt16>)

int _mm256_testc_si256 (__m256i a, __m256i b)

VPTEST ymm, ymm/m256

C# Copy

```
public static bool TestC (System.Runtime.Intrinsics.Vector256<ushort> left,
System.Runtime.Intrinsics.Vector256<ushort> right);
```

Parameters

left Vector256<UInt16>

right Vector256<UInt16>

Returns

Boolean

(a) .NET documentation [14].

```
int _mm256_testc_si256 (__m256i a, __m256i b)
```

vptest

Synopsis

```
int _mm256_testc_si256 (__m256i a, __m256i b)
#include <immintrin.h>
Instruction: vptest ymm, ymm
CPUID Flags: AVX
```

Description

Compute the bitwise AND of 256 bits (representing integer data) in a and b, and set ZF to 1 if the result is zero, otherwise set ZF to 0. Compute the bitwise NOT of a and then AND with b, and set CF to 1 if the result is zero, otherwise set CF to 0. Return the CF value.

Operation

```
IF ((a[255:0] AND b[255:0]) == 0)
	ZF := 1
ELSE
	ZF := 0
FI
IF (((NOT a[255:0]) AND b[255:0]) == 0)
	CF := 1
ELSE
	CF := 0
FI
RETURN CF
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	-	1
Skylake	3	1

(b) Intel documentation [15].

Figure 3. Documentation for testC intrinsic.

f Since the very begininng .NET and the C# have supported *unsafe* code blocks where it is possible to write code that bypasses the built in memory management and perform operations directly with pointers. But in widely used libraries unsafe code blocks are preferably avoided due to added risks. Thus the preferred way is to write C# in the default managed segment where the built in memory manager and garbage collector are responsible for managing memory resources. It is also sometimes referred to as *safe* code.

Next subsection covers, step by step, how those challenges were tackled and how I reached the optimised version of the method.

2.4 Optimisation

First of all, for backward compatibility, it is necessary to ensure that the processor supports SIMD intrinsics. .NET provides with the corresponding properties. In case the processor does not support the intrinsics, JIT [16] will not include this branch at all [17].

```
if (Avx2.IsSupported && Avx.IsSupported)
{
    ...
}
```

To represent `String` object as a contiguous memory, `ReadOnlySpan` is used. Besides using only safe code, it allows slicing (method, similar to `String.Substring()`, but without allocating memory and it has almost no overhead) [18]–[19].

```
ReadOnlySpan<char> inputSpan = input;
```

As we use only safe code, after dividing the input string into `Vector256` chunks there will be a part less than `Vector256` size. In that case the initial code will be used as it is.

```
ReadOnlySpan<Vector256<ushort>> avx2Span =
    MemoryMarshal.Cast<char, Vector256<ushort>>(inputSpan.Slice(i));
```

As it is not possible to apply intrinsics for user-defined operations (accessing random indices of a string (span) and calling Binary Search Tree (BST) methods), the only option was to try to avoid checking the condition at each iteration. To figure out which branch of *if-else* is called more often, the code was run along the regexes and dataset, mentioned in the Benchmark chapter. It appeared that the most of the cases SRM did not call *else* branch with BST.

So the next step was to compare `A_StartSet.precomputed.Length` with all the characters in the current vector (`input_i < A_StartSet.precomputed.Length`).

As it was mentioned, strict *greater than* and *less than* operations offered by intrinsics are available only for `sbyte`, `long`, `short`, `int` data types. A combination of other intrinsics was used to overcome this constraint. First, the maximum value of two ar-

guments is found, and then, it is compared with the argument which is supposed to be greater. This way, I got the *greater or equal* operator. To make the comparison strict, the greater value was decreased by 1. So, essentially I got the following operation `input_i <= A_StartSet.precomputed.Length - 1`, which is the equivalent of the necessary operation over integers.

```
Vector256<ushort> comp =
    Vector256.Create((ushort)(A_StartSet.precomputed.Length - 1));
Vector256<ushort>res =
    Avx2.CompareEqual(Avx2.Max(v, comp), comp);
```

Finally, to ensure the condition is met for all the vector elements and the result value is of the boolean type (so that it could be used in the *if* condition), I used `testC` operation. In fact, it might function as operator that checks whether the first argument's bits are all ones when you pass all ones as the second argument.

```
Vector256<ushort> e = Vector256.Create((ushort)0xFFFF);
if (Avx.TestC(res, e))
{
    ...
}
```

Final version of the code is available on the listing 2.

Additionally, to ensure that code works correctly, I wrote a code snippet which would count the number of matches of the original version of SRM and the optimised version.

Listing 2. Optimised implementation of IndexOfStartset() method.

```
int IndexOfStartset(string input, int i)
{
    int k = input.Length;

    if (Avx2.IsSupported && Avx.IsSupported)
    {
        ReadOnlySpan<char> inputSpan = input;
        ReadOnlySpan<Vector256<ushort>> avx2Span =
            MemoryMarshal.Cast<char, Vector256<ushort>>
                (inputSpan.Slice(i));
        Vector256<ushort> comp =
            Vector256.Create(
                (ushort)(A_StartSet.precomputed.Length - 1));
        ushort maxShort = (ushort)0xFFFF;
        Vector256<ushort> e = Vector256.Create(maxShort);
        Vector256<ushort> res = Vector256.Create((ushort)0);
        foreach (Vector256<ushort> v in avx2Span)
        {
            res = Avx2.CompareEqual(Avx2.Max(v, comp), comp);
            if (Avx.TestC(res, e))
            {
                for (int j = 0; j < Vector256<ushort>.Count; j++)
                {
                    if (A_StartSet.precomputed[inputSpan[i]])
                        goto Done;
                    else i += 1;
                }
            }
            else
            {
                for (int j = 0; j < Vector256<ushort>.Count; j++)
                {
                    ushort x = res.GetElement(j);
                    if (x == maxShort ?
                        A_StartSet.precomputed[inputSpan[i]] :
                            A_StartSet.bst.Find(inputSpan[i]) == 1)
                        goto Done;
                    else
                        i += 1;
                }
            }
```

```
        }
    }

    while (i < k)
    {
        var input_i = input[i];
        if (input_i < A_StartSet.precomputed.Length ?
            A_StartSet.precomputed[input_i] :
                A_StartSet.bst.Find(input_i) == 1)
            break;
        else
            i += 1;
    }

    Done:
        if (i == k)
            return -1;
        else
            return i;
}
```

3 Benchmarking

To evaluate the impact of the changes the modified SRM was benchmarked along side with the original version compiled as .NET standard 2.1 binary (serving as a baseline) and as a .NET version 5.0 binary. The .NET current default regex library was used as a reference baseline.

3.1 Datasets

In order to validate the performance of SRM against default .NET regex engine, I used Mark Twain corpus [20] also used in [6]. The following regular expressions were benchmarked:

- `Mark`
- `Twain`
- `(?i)Twain`
- `[a-z]shing`
- `Huck[a-zA-Z]+|Saw[a-zA-Z]+`
- `[a-q][^u-z]{13}x`
- `Tom|Sawyer|Huckleberry|Finn`
- `(?i)Tom|Sawyer|Huckleberry|Finn`
- `.{0,2}Tom|Sawyer|Huckleberry|Finn`
- `.{2,4}Tom|Sawyer|Huckleberry|Finn`
- `Tom.{10,25}river|river.{10,25}Tom`
- `[a-zA-Z]+ing`
- `\s[a-zA-Z]{0,12}ing\s`
- `([A-Za-z]awyer|[A-Za-z]inn)\s`
- `["'][^"']{0,30}[?!\.][\"']`
- `\p{Sm}`

However, Mark Twain corpus contains mostly ASCII characters and only a few extended ASCII characters. As SRM claims to be efficient with UTF-8 inputs and regexes, I compiled similar list of regexes for Estonian wikipedia (as of 01.03.2021) [21]:

- `Eesti`
- `Rootsi`
- `(?i)Eesti`
- `[a-züõöä]ee`
- `Heli[a-zA-ZüõöäÜÕÖÄ]+|Aja[a-zA-ZüõöäÜÕÖÄ]+`
- `[a-q][^u-z]{12}x`
- `Toomas|Margus|Rein|Jaan`
- `(?i)Toomas|Margus|Rein|Jaan`
- `.{0,2}Toomas|Margus|Rein|Jaan`
- `.{2,4}Toomas|Margus|Rein|Jaan`
- `Eesti.{10,25}jõgi|jõgi.{10,25}Eesti`
- `[a-zA-ZüõöäÜÕÖÄ]+tud`
- `\s[a-zA-ZüõöäÜÕÖÄ]{0,12}tud\s`
- `([A-Za-z]ina|[A-Za-z]ein)\s`
- `["'][^"']{0,31}[?!\.][\"']`
- `\p{Sc}`

3.2 Benchmark setup

For benchmarking, I used *BenchmarkDotNet* library [22]. It runs the experiment the necessary amount of times and works out the statistics. Additionally, it takes into consideration the fact that .NET applications run in virtual environment and are optimised at runtime by JIT. Hence, the experiments, first, are run as a warm-up and only then taken into account.

Experiments were run on two AI-Lab [23] machines with different configurations:

- Intel® Core™ i9-10900X CPU [1] @ 3.70GHz 10-Core/20-thread Processor, 128 GB of memory, 2 x NVidia 2080Ti GPU with 11 GB of graphics memory
- AMD Threadripper 3960X 24-Core/48-thread Processor [2], 128 GB of memory, AMD RX6900XT GPU with 16 GB of graphics memory

The code of the benchmark is available in the project repository [24]. Building and execution steps are available in `README.md` file.

3.3 Experiment results and discussion

The results of the benchmarking are shown in tables 1-4. Tables 1 and 2 cover the benchmarks run on The Entire Project Gutenberg Works of Mark Twain dataset (on Intel and AMD processors, correspondingly). Tables 3 and 4 cover the benchmarks run on Estonian wikipedia dataset (on Intel and AMD processors, correspondingly).

Although for some regular expressions the application has become slower (up to 9%), in some cases the gain in performance has been up to 48%.

SRM outperforms the default .NET regex engine due to Symbolic Automata. I have managed to improve it for the cases when a regex starts with a fixed prefix (no sets) or with a small set. I have to mention that optimisation was done only for the case when BST was not called. It means that for the majority of the symbols the values were precomputed. On the other hand, the precomputed array is limited by ASCII symbols, so the optimisation might not have effect, for example, for the inputs and regular expressions which use Cyrillic alphabets. This scenario needs to be benchmarked separately.

Another regex component aspect which needs to be researched separately is the number of repetitions. In the current datasets, there were only a few such regular expressions and half of them contained very low number of repetitions.

Regarding using the intrinsics, even though intrinsics have their own limitations and are not well documented, it is still possible to use them for some cases. The code now has duplications and it is less readable. However, in my opinion, it is a reasonable trade off for such a gain in performance.

[1] https://ark.intel.com/content/www/us/en/ark/products/198019/intel-core-i9-10900x-x-series-processor-19-25m-cache-3-70-ghz.html

[2] https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3960x

Interestingly, the experiments reveal that AMD Threadripper 39XX and Intel Core i9 10th generation processors perform quite differently. Although their turbo boost frequencies are comparable (4.5 GHz) and the experiments were carried out with a single thread running per processor the effects of the SIMD optimisations increase the performance of regexes such as ”Tom|Sawyer|Huckleberry|Finn” on AMD, but seem to have no effect on Intel Core i9. In general it appears that for string processing the Ryzen cores seem to outperform Intel Core i9 architecture. It should be noted that the computers used identical 16 GB memory modules with all memory slots populated.

<table>
<tr><th>Regex</th><th>.NET</th><th>SRM</th><th>SRM (.NET5)</th><th>SRM (.NET5) Avx2</th><th>SRM (.NET5) Avx2 / SRM</th></tr>
<tr><td>(?i)Tom|Sawyer|Huckleberry|Finn</td><td>526.586 ms</td><td>100.021 ms</td><td>99.822 ms</td><td>90.848 ms</td><td>0.91</td></tr>
<tr><td>(?i)Twain</td><td>29.320 ms</td><td>54.933 ms</td><td>55.061 ms</td><td>55.039 ms</td><td>1.00</td></tr>
<tr><td>([A-Za-z]awyer|[A-Za-z]inn)\s</td><td>1,452.229 ms</td><td>145.075 ms</td><td>145.181 ms</td><td>150.195 ms</td><td>1.04</td></tr>
<tr><td>.{0,2}Tom|Sawyer|Huckleberry|Finn</td><td>2,747.648 ms</td><td>115.292 ms</td><td>114.795 ms</td><td>123.565 ms</td><td>1.07</td></tr>
<tr><td>.{2,4}Tom|Sawyer|Huckleberry|Finn</td><td>2,892.666 ms</td><td>114.699 ms</td><td>121.068 ms</td><td>122.862 ms</td><td>1.07</td></tr>
<tr><td>Huck[a-zA-Z]+|Saw[a-zA-Z]+</td><td>81.827 ms</td><td>24.358 ms</td><td>24.591 ms</td><td>18.881 ms</td><td>0.78</td></tr>
<tr><td>Mark</td><td>19.667 ms</td><td>3.382 ms</td><td>3.591 ms</td><td>3.542 ms</td><td>1.05</td></tr>
<tr><td>Tom.{10,25}river|river.{10,25}Tom</td><td>134.394 ms</td><td>54.717 ms</td><td>54.513 ms</td><td>44.339 ms</td><td>0.81</td></tr>
<tr><td>Tom|Sawyer|Huckleberry|Finn</td><td>91.790 ms</td><td>26.593 ms</td><td>26.660 ms</td><td>27.346 ms</td><td>1.03</td></tr>
<tr><td>Twain</td><td>20.275 ms</td><td>4.295 ms</td><td>4.184 ms</td><td>4.338 ms</td><td>1.01</td></tr>
<tr><td>[”’][^”’]{0,30}[?!\.][\”’]</td><td>138.377 ms</td><td>50.076 ms</td><td>49.937 ms</td><td>40.877 ms</td><td>0.82</td></tr>
<tr><td>[a-q][^u-z]{13}x</td><td>1,167.907 ms</td><td>9,202.485 ms</td><td>9,278.862 ms</td><td>9,323.441 ms</td><td>1.01</td></tr>
<tr><td>[a-zA-Z]+ing</td><td>1,088.468 ms</td><td>153.892 ms</td><td>153.377 ms</td><td>158.846 ms</td><td>1.03</td></tr>
<tr><td>[a-z]shing</td><td>834.930 ms</td><td>143.442 ms</td><td>143.365 ms</td><td>148.494 ms</td><td>1.04</td></tr>
<tr><td>\p{Sm}</td><td>77.347 ms</td><td>35.412 ms</td><td>35.407 ms</td><td>24.958 ms</td><td>0.70</td></tr>
<tr><td>\s[a-zA-Z]{0,12}ing\s</td><td>712.263 ms</td><td>130.446 ms</td><td>130.241 ms</td><td>140.467 ms</td><td>1.08</td></tr>
</table>

Table 1. Benchmark results: Mark Twain dataset on Intel processor.

Regex	.NET	SRM	SRM (.NET5)	SRM (.NET5) Avx2	SRM (.NET5) Avx2 / SRM
(?i)Tom|Sawyer|Huckleberry|Finn	596.116 ms	85.901 ms	86.079 ms	**81.439 ms**	**0.95**
(?i)Twain	**30.595 ms**	**58.554 ms**	58.622 ms	58.679 ms	1.00
([A-Za-z]awyer|[A-Za-z]inn)\s	1,727.670 ms	**137.339 ms**	137.414 ms	138.016 ms	1.00
.{0,2}Tom|Sawyer|Huckleberry|Finn	3,140.863 ms	110.315 ms	**109.150 ms**	109.824 ms	1.00
.{2,4}Tom|Sawyer|Huckleberry|Finn	3,752.531 ms	113.839 ms	111.692 ms	**110.465 ms**	**0.97**
Huck[a-zA-Z]+|Saw[a-zA-Z]+	62.279 ms	34.419 ms	33.639 ms	**22.625 ms**	**0.66**
Mark	18.051 ms	2.974 ms	**2.971 ms**	2.991 ms	1.01
Tom.{10,25}river|river.{10,25}Tom	117.618 ms	50.134 ms	48.803 ms	**35.201 ms**	**0.70**
Tom|Sawyer|Huckleberry|Finn	72.998 ms	36.744 ms	35.518 ms	**20.185 ms**	**0.52**
Twain	18.344 ms	4.004 ms	**3.473 ms**	3.883 ms	0.98
[”’][^”’]{0,30}[?!\.][\”’]	124.251 ms	45.833 ms	46.668 ms	**37.129 ms**	**0.81**
[a-q][^u-z]{13}x	**1,223.610 ms**	7,936.593 ms	8,091.623 ms	8,622.955 ms	1.09
[a-zA-Z]+ing	1,144.840 ms	**144.449 ms**	147.008 ms	148.725 ms	1.03
[a-z]shing	953.279 ms	135.576 ms	**131.955 ms**	135.815 ms	1.00
\p{Sm}	57.335 ms	33.899 ms	33.831 ms	**22.194 ms**	**0.65**
\s[a-zA-Z]{0,12}ing\s	729.039 ms	**121.866 ms**	123.747 ms	126.368 ms	1.04

Table 2. Benchmark results: Mark Twain dataset on AMD processor.

<table>
<tr><th>Regex</th><th>.NET</th><th>SRM</th><th>SRM (.NET5)</th><th>SRM (.NET5) Avx2</th><th>SRM (.NET5) Avx2 / SRM</th></tr>
<tr><td>(?i)Eesti</td><td>**1,721.966 ms**</td><td>2,618.421 ms</td><td>2,618.510 ms</td><td>2,610.830 ms</td><td>1.00</td></tr>
<tr><td>**(?i)Toomas|Margus|Rein|Jaan**</td><td>18,423.398 ms</td><td>3,407.311 ms</td><td>3,501.442 ms</td><td>**3,251.826 ms**</td><td>**0.95**</td></tr>
<tr><td>([A-Za-z]ina|[A-Za-z]ein)\s</td><td>57,506.827 ms</td><td>5,701.311 ms</td><td>**5,691.508 ms**</td><td>5,911.740 ms</td><td>1.04</td></tr>
<tr><td>.{0,2}Toomas|Margus|Rein|Jaan</td><td>127,574.429 ms</td><td>5,109.237 ms</td><td>**5,108.021 ms**</td><td>5,475.085 ms</td><td>1.07</td></tr>
<tr><td>.{2,4}Toomas|Margus|Rein|Jaan</td><td>128,910.704 ms</td><td>**5,130.630 ms**</td><td>5,140.269 ms</td><td>5,471.042 ms</td><td>1.07</td></tr>
<tr><td>Eesti</td><td>1,177.946 ms</td><td>205.314 ms</td><td>205.013 ms</td><td>**204.422 ms**</td><td>1.00</td></tr>
<tr><td>**Eesti.{10,25}jõgi|jõgi.{10,25}Eesti**</td><td>4,352.452 ms</td><td>1,296.753 ms</td><td>1,298.791 ms</td><td>**1,122.132 ms**</td><td>**0.86**</td></tr>
<tr><td>**Heli[a-zA-ZüõöäÜÕÖÄ]+|Aja[a-zA-ZüõöäÜÕÖÄ]+**</td><td>3,871.276 ms</td><td>1,111.153 ms</td><td>1,112.767 ms</td><td>**927.567 ms**</td><td>**0.84**</td></tr>
<tr><td>**Rootsi**</td><td>752.357 ms</td><td>172.583 ms</td><td>171.862 ms</td><td>**170.157 ms**</td><td>**0.99**</td></tr>
<tr><td>Toomas|Margus|Rein|Jaan</td><td>4,401.658 ms</td><td>**1,181.965 ms**</td><td>1,716.256 ms</td><td>1,257.119 ms</td><td>1.06</td></tr>
<tr><td>**[”’][^”’]{0,31}[?!\.][\”’]**</td><td>5,946.567 ms</td><td>1,928.788 ms</td><td>1,930.675 ms</td><td>**1,540.108 ms**</td><td>**0.80**</td></tr>
<tr><td>[a-q][^u-z]{12}x</td><td>45,699.453 ms</td><td>**15,747.063 ms**</td><td>15,861.119 ms</td><td>15,926.193 ms</td><td>1.01</td></tr>
<tr><td>[a-zA-ZüõöäÜÕÖÄ]+tud</td><td>51,283.276 ms</td><td>5,560.944 ms</td><td>**5,558.053 ms**</td><td>5,742.833 ms</td><td>1.03</td></tr>
<tr><td>[a-züõöä]ee</td><td>33,031.723 ms</td><td>5,593.717 ms</td><td>**5,583.875 ms**</td><td>5,657.266 ms</td><td>1.01</td></tr>
<tr><td>**\p{Sc}**</td><td>3,691.848 ms</td><td>1,598.289 ms</td><td>1,597.292 ms</td><td>**1,179.714 ms**</td><td>**0.74**</td></tr>
<tr><td>\s[a-zA-ZüõöäÜÕÖÄ]{0,12}tud\s</td><td>19,163.914 ms</td><td>4,248.384 ms</td><td>**4,267.430 ms**</td><td>4,369.826 ms</td><td>1.03</td></tr>
</table>

Table 3. Benchmark results: Estonian wikipedia dataset on Intel processor.

Regex	.NET	SRM	SRM (.NET5)	SRM (.NET5) Avx2	SRM (.NET5) Avx2 / SRM
(?i)Eesti	**1,619.947 ms**	2,762.118 ms	2,586.767 ms	2,602.085 ms	0.96
(?i)Toomas|Margus|Rein|Jaan	20,975.642 ms	3,135.517 ms	3,124.506 ms	**2,776.571 ms**	**0.89**
([A-Za-z]ina|[A-Za-z]ein)\s	69,774.342 ms	5,196.090 ms	**5,186.411 ms**	5,270.636 ms	1.01
.{0,2}Toomas|Margus|Rein|Jaan	151,889.328 ms	**4,690.700 ms**	4,784.250 ms	4.852.462 ms	1.03
.{2,4}Toomas|Margus|Rein|Jaan	181,856.643 ms	4,770.527 ms	**4,747.014 ms**	4,897.799 ms	1.03
Eesti	907.431 ms	136.907 ms	**136.843 ms**	136.998 ms	1.00
Eesti.{10,25}jõgi|jõgi.{10,25}Eesti	3,452.557 ms	1,668.475 ms	1,676,121 ms	**1,017,001 ms**	**0.61**
Heli[a-zA-ZüõöäÜÕÖÄ]+|Aja[a-zA-ZüõöäÜÕÖÄ]+	2,993,605 ms	1,562.012 ms	1,543.735 ms	**1,068.857 ms**	**0.68**
Rootsi	684.201 ms	117.980 ms	**117.134** ms	117.965 ms	1.00
Toomas|Margus|Rein|Jaan	3,665,899 ms	1,580.230 ms	1,591.544 ms	**861.836 ms**	**0.55**
[”’][^”’]{0,31}[?!\.][\”’]	4,938.891 ms	1,805.495 ms	1,803.828 ms	**1,413.222 ms**	**0.78**
[a-q][^u-z]{12}x	48,691.007 ms	15,344.857	15,079.095 ms	**15,030.052 ms**	**0.98**
[a-zA-ZüõöäÜÕÖÄ]+tud	55,795.835 ms	**5,244.481 ms**	5,287.474 ms	5,297.134 ms	1.01
[a-züõöä]ee	34,791.855 ms	5,242.730 ms	5,208.587 ms	**5,026.472 ms**	**0.96**
\p{Sc}	2,762.207 ms	1,511.756 ms	1,529.579 ms	**1,105.293 ms**	**0.73**
\s[a-zA-ZüõöäÜÕÖÄ]{0,12}tud\s	20,793.314 ms	4,129.517 ms	4,171.502 ms	**3,996.533 ms**	**0.97**

Table 4. Benchmark results: Estonian wikipedia dataset on AMD processor.

4 Related work

While using SIMD intrinsics is more general method, regex matching problem can be approached considering the knowledge in the domain.

Resolving very specific aspects of regular expressions. For example, similarly to Symbolic Automata and solving sets matching problem, it is possible to speed up matching regular expressions with bounded repetition, e.g. `k{5}`, by using Counting-Set Automata (CsA), introduced in [25]. The idea is briefly explained in the appendix.

Using different string matching algorithms. Fixed parts of regular expressions can be considered as strings. In this case, different string matching algorithms can be used. Boyer-Moore preprocesses the pattern and early skips the parts of the input string, which will not match [26]. [11] claims it is possible to speed up the algorithm with SIMD intrinsics. Some modern algorithms could be used, too. DFC string matching algorithm, which outperforms one of the most popular string matching algorithm, Aho-Corasic, twice and also uses SIMD intrinsics, was introduced in 2016 [27].

Comprehensive approach. Hyperscan, a high-performance regex matching system, partially converts NFA to DFA and decomposes regular expressions as strings and subregexes. Additionally, it does dominant path and dominant region analysis and uses SIMD intrinsics [28].

5 Summary

The goal of this thesis was to optimise SRM, a symbolic automata based regular expression engine, using SIMD intrinsics. SRM is very promising as it outperforms default .NET regular expressions engine and its performance is getting close to the performance of RE2 [29]. It is also included in future .NET release as a feature [8].

To limit the scope of preparing data sets, I utilised one widely-used data set for ASCII encodings and compiled my own data set for UTF-8 encoding, based on the Estonian Wikipedia.

While working on the thesis, I faced some limitations of using SIMD intrinsics and found solutions for some of them. This knowledge might be useful for those who want to apply SIMD intrinsics in their software.

Also, I succeeded in finding a way to apply SIMD intrinsics to SRM and increased the performance by up to 48% for the regular expressions with prefixes. The results were empirically proved and validated by establishing the experiments.

Therefore, the goal of the thesis was achieved and the real outcomes exceeded the expected ones for the known scenarios.

5.1 Future Work

This work sets the ground for further work and research:

- In order to improve the performance of SRM, Counting-Set Automata can be implemented in SRM.
- To evaluate the performance of SRM for non-latin alphabets, new data sets need to be compiled.
- Regarding regex engines' benchmarking, this work covers only matching regular expressions in large strings. To observe the behaviour and performance of SRM and other engines for other scenarios, a new, more comprehensive benchmark should be created.
- Modern processors are already provided with 512-bit registers, but .NET SIMD

intrinsics currently support only 256-bit vectors. Once 512-bit vectors support is available, current solution might be improved even more.

References

[1] J. E. Friedl, *Mastering regular expressions*. " O'Reilly Media, Inc.", 2006.

[2] R. Chowdhury, M. R. Babu, V. Mishra, and H. Jain, "Regular expressions in big data analytics," in *2017 International Conference on Intelligent Computing and Control (I2C2)*, IEEE, 2017, pp. 1–10.

[3] J. C. Davis, "On the impact and defeat of regular expression denial of service," Ph.D. dissertation, Virginia Tech, 2020.

[4] M. Veanes, "Applications of symbolic finite automata," in *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*, S. Konstantinidis, Ed., ser. Lecture Notes in Computer Science, vol. 7982, Springer, 2013, pp. 16–23. DOI: `10.1007/978-3-642-39274-0_3`. [Online]. Available: `https://doi.org/10.1007/978-3-642-39274-0%5C_3`.

[5] H. Tamm and M. Veanes, "Theoretical aspects of symbolic automata," in *International Conference on Current Trends in Theory and Practice of Informatics*, Springer, 2018, pp. 428–441.

[6] O. Saarikivi, M. Veanes, T. Wan, and E. Xu, "Symbolic regex matcher," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 372–378.

[7] "Github - automatadotnet/srm." (), [Online]. Available: `https://github.com/AutomataDotNet/srm` (visited on 04/10/2021).

[8] "Dotnet/runtimelab · github." (), [Online]. Available: `https://github.com/dotnet/runtimelab/tree/feature/regexsrm/src/libraries/System.Text.RegularExpressions/src/System/Text/RegularExpressions/srm` (visited on 06/08/2021).

[9] "Intel intrinsics guide." (), [Online]. Available: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/` (visited on 05/14/2021).

[10] A. Stojanov, I. Toskov, T. Rompf, and M. Püschel, "Simd intrinsics on managed language runtimes," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 2–15.

[11] E. Sitaridi, O. Polychroniou, and K. A. Ross, "Simd-accelerated regular expression matching," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 1–7.

[12] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.

[13] "Visual studio preview." (), [Online]. Available: `https://visualstudio.microsoft.com/vs/preview/` (visited on 05/26/2021).

[14] "Avx.testc method (system.runtime.intrinsics.x86) | microsoft docs." (), [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/api/system.runtime.intrinsics.x86.avx.testc?view=net-5.0#System_Runtime_Intrinsics_X86_Avx_TestC_System_Runtime_Intrinsics_Vector256_System_UInt16__System_Runtime_Intrinsics_Vector256_System_UInt16__` (visited on 04/27/2021).

[15] "Intel intrinsics guide." (), [Online]. Available: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=3296,766,766,5909,5945,5939,5931,5200,5945,339,766,5909%5C&text=VPTEST%5C&techs=AVX,AVX2` (visited on 04/27/2021).

[16] "Using .net hardware intrinsics api to accelerate machine learning scenarios | .net blog." (), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process#compiling_msil_to_native_code (visited on 05/25/2021).

[17] "Using .net hardware intrinsics api to accelerate machine learning scenarios | .net blog." (), [Online]. Available: https://devblogs.microsoft.com/dotnet/using-net-hardware-intrinsics-api-to-accelerate-machine-learning-scenarios/ (visited on 03/14/2021).

[18] "Span - adam sitnik - .net performance and reliability." (), [Online]. Available: https://adamsitnik.com/Span/ (visited on 03/20/2021).

[19] "Readonlyspan<t> struct (system) | microsoft docs." (), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.readonlyspan-1?view=net-5.0 (visited on 05/14/2021).

[20] "Mark twain corpus." (), [Online]. Available: http://www.gutenberg.org/files/3200/old/mtent12.zip (visited on 03/17/2021).

[21] "Estonian wikipedia backup as of 01.03.2021." (), [Online]. Available: https://dumps.wikimedia.org/etwiki/20210301/etwiki-20210301-pages-articles-multistream.xml.bz2 (visited on 04/29/2021).

[22] "Benchmarkdotnet." (), [Online]. Available: https://benchmarkdotnet.org/ (visited on 05/23/2021).

[23] "Intro - taltech ai-lab." (), [Online]. Available: https://ai-lab.pages.taltech.ee/posts/intro/ (visited on 05/02/2021).

[24] "Bahdan.yanovich / regex-experiments gitlab." (), [Online]. Available: https://gitlab.cs.ttu.ee/bahdan.yanovich/regex-experiments (visited on 06/14/2021).

[25] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar, "Regex matching with counting-set automata," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[26] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[27] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "Dfc: Accelerating string pattern matching for network applications," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 551–565.

[28] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern cpus," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 631–648.

[29] "Github - google/re2." (), [Online]. Available: https://github.com/google/re2 (visited on 06/03/2021).

[30] M. Sipser, *Introduction to the Theory of Computation*. Cengage learning, 2012.

[31] P. Hooimeijer and M. Veanes, "An evaluation of automata algorithms for string analysis," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2011, pp. 248–262.

[32] "Introduction to symbolic automata." (), [Online]. Available: https://compose.ioc.ee/alice/videos/hendrik20210219.mp4 (visited on 02/19/2021).

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis

I, Bahdan Yanovich,

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis ”Optimisation of Symbolic Automata Based Regular Expression Library SRM Using SIMD Intrinsics” , supervised by Juhan-Peep Ernits
   a. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   b. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

14.06.2021

Appendix 2 – Automata Introduction

2.1 Classical Automata Theory

A **finite automaton** M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states,**
2. Σ is a finite set called the **alphabet,**
3. $\delta : Q \times \Sigma \to Q$ is the **transition function,**
4. $q_0 \in Q$ is the **start state,**
5. $F \subset Q$ is the **set of final states.** [30]

Consider the following finite automaton M_1:

1. $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}\}$,
2. $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
3. δ described as

	0	1	2	3	4	5	6	7	8	9
q_0										q_1
q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
q_2	q_{12}					q_{12}				
q_3	q_{12}					q_{12}				
q_4	q_{12}					q_{12}				
q_5	q_{12}					q_{12}				
q_6	q_{12}					q_{12}				
q_7	q_{12}					q_{12}				
q_8	q_{12}					q_{12}				
q_9	q_{12}					q_{12}				
q_{10}	q_{12}					q_{12}				
q_{11}	q_{12}					q_{12}				
q_{12}										

4. q_0 is the start state, and
5. $F = \{q_{12}\}$.

It can be visualised by a state diagram shown on Figure 4[1]. If A is the set of all string that automaton M accepts, we say that A is the **language of automaton *M*** and write $L(M) = A$. [30]

This finite automaton accepts the string of a 3-digit integer greater than or equal to 900 and divisible by 5. Another word, $A = \{w|\ w$ is a string that represents a 3-digit integer greater than or equal to 900 and divisible by 5 $\}$ and $L(M_1) = A$.

Let A and B be languages. We define the regular operations **union, contatenation,** and **star** as follows:

- **Union**: $A \cup B = \{x|\ x \in A$ or $x \in B\}$,
- **Concatenation**: $A \circ B = \{xy|\ x \in A$ and $y \in B\}$,
- **Union**: $A^* = \{x_1x_2 \cdots x_k|\ k \geq 0$ and each $x_i \in A\}$. [30]

[1] Technically each $0, 5$ transition consists of two transitions: 0 and 5.

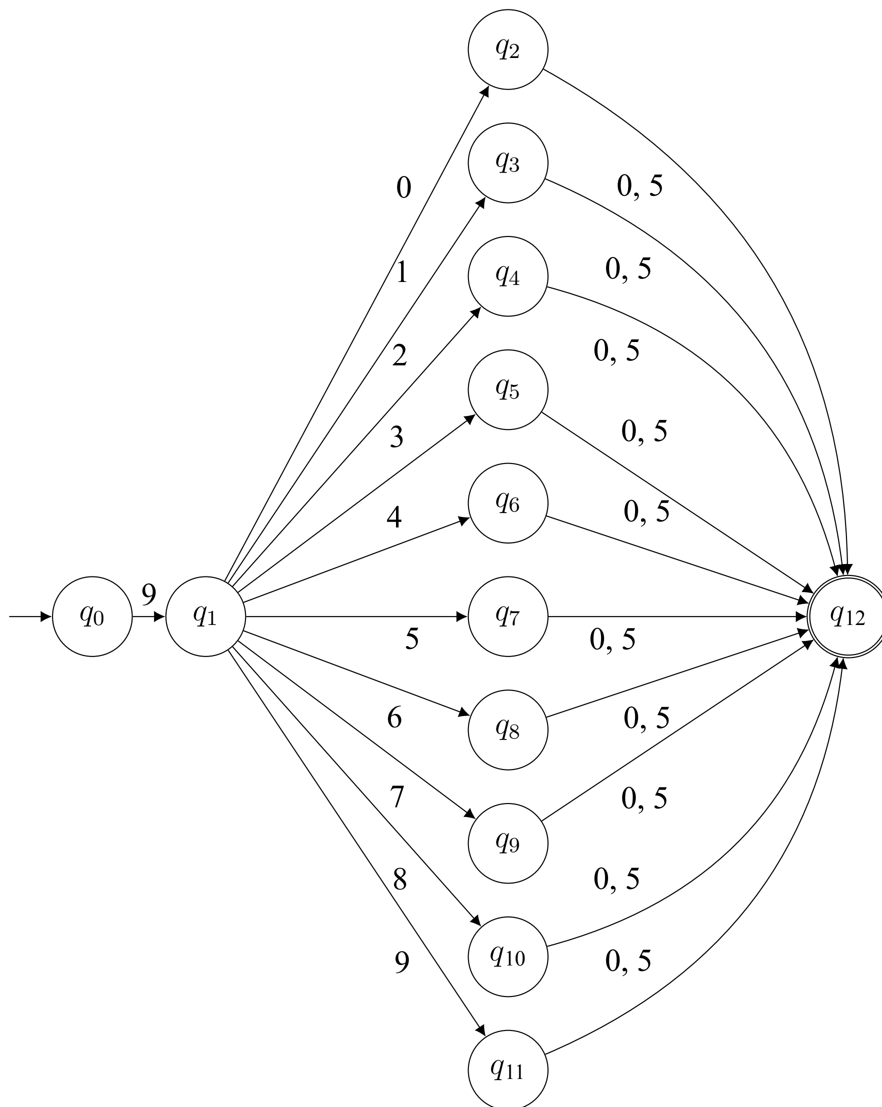

Figure 4. State diagram of a finite automaton that accepts strings that represent 3-digit integers that are greater than or equal to 900 and divisible by 5.

Say that R is a **regular expression** if R is

1. a for some a in the alphabet Σ,
2. ε (empty symbol),
3. $\emptyset$ (empty language),
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression. [30]

Regular expressions and finite automata are equivalent as any finite automaton can be converted to a regular expression that recognises the language it describes, and vice versa. More details can be found in [30].

Therefore, the automaton M_1, also shown as state diagram at figure 4, could be represented as the regular expression `"9[0-9][05]"`.

As one might have noticed from the example above, automata of quite trivial regular expressions can grow very quickly. Moreover, DFA and NFA work with finite alphabets.

2.2 Symbolic Automata

As I mentioned above, one of the problems of finite automata is that they work with finite alphabets. It becomes unreasonably complex with large alphabets (e.g. UTF-8 encoding in the context of regular expressions). There is another approach of handling transitions and large or even infinite languages – using effective Boolean algebras as alphabets and predicates as transitions, so that several transitions may be combined into a single symbolic move [31]. That is the main idea of Symbolic Automata.

Here is the formal definition.

A **symbolic finite automaton** is a tuple $M = (Q, \Sigma, \Delta, q_0, F)$, where

1. Q is a finite set of **states,**
2. Σ is the **input alphabet (effective Boolean algebra),**
3. $\Delta : Q \times 2^{\Sigma} \times Q$ is the **move relation,**
4. $q_0 \in Q$ is the **start state,**
5. $F \subseteq Q$ is the **set of final states.** [31]

$(p, \varphi, q) \in \Delta$ is a transition from state p to state q, where φ is the predicate of the transition. [32]

Figure 5 shows how the same regex (`"9[0-9][05]"`) can be represented as a symbolic automaton. As one can notice, the number of states and transitions has been reduced significantly.

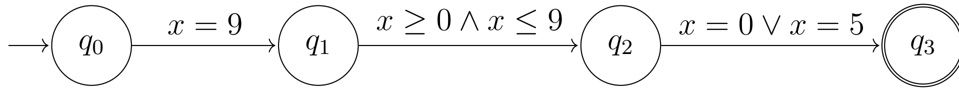

Figure 5. Regular expression ”9[0-9][05]” represented as a symbolic automaton.

2.3 Counting-Set Automata

Another potential bottleneck of regexes is bounded repetition, e.g. `k{5}`. To reduce the number of repetitive states in an automaton with bounded repetition, Turoňová et al. introduced Counting-Set Automata (CsA), automata with registers that can hold sets of bounded integers and can be manipulated by a limited set of constant-time operations. For more details, see [25]. Comparison of the same regex (`"k{5}"`) represented by finite automaton and counting-set automaton is shown at figures 6 and 7. Again, the number of states has reduced when using a new approach.

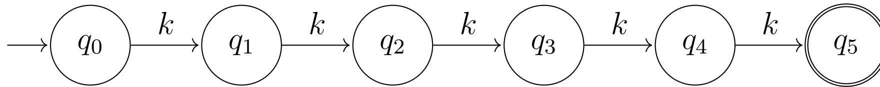

Figure 6. Regular expression ”k{5}” represented as a finite automaton.

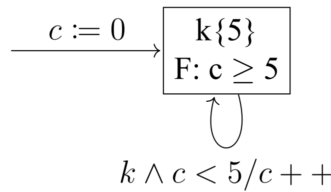

Figure 7. Regular expression ”k{5}” represented as a counting-set automaton.

Appendix 3 – Acknowledgements

While writing this thesis, I have been lucky to receive a lot of support.

I would like to thank my supervisor, Juhan-Peep Ernits, for showing me the real research. Your expertise and desire of constant moving forward has been an example for me. I admire how easily you bring people together and establish collaboration between them. It has been a pleasure to work with you.

I would also like to thank my colleagues, Marko Kääramees and Ago Luberg, for letting me in to the university world and kindly reminding me about the graduation.

Additionally, I would like to thank my friends, Valeriia Shpychka, Karina Nikitina and Seyidahmad Alibayov, for being ready to help any time by words and deeds and being able to share in my joy.