TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Rainer Viirlaid 213102IAIB

Regulaaravaldiste mootori RE# vastete alguskohtade leidmise optimeerimine suurte sisendtekstide jaoks

Bakalaureusetöö

Juhendaja: Juhan-Peep Ernits
PhD
Kaasjuhendaja: Ian Erik Varatalu
MSc

Tallinn 2024

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Rainer Viirlaid

27.05.2024

Annotatsioon

RE# on uuenduslik regulaaravaldiste mootor, mis põhineb sümboolsetel olekumasinatel. Töö eesmärk on mootori otsingu kiirendamine suurte sisendtekstide puhul, keskendudes vastete alguskohtade otsingule ning kasutades sümbolite sagedusi otsingutekstis.

Töö käigus valmivad kaks põhilist optimisatsiooni, mis üritavad mustrist leida haruldasemaid sümbolihulki ning tekstist esmalt neid otsida.

Tulemusi mõõtsin eesti- ja ingliskeelsete tekstide peal, mis olid 6MB-1GB suurused, ning .NET Runtime koodibaasi peal, mis on 401MB. Parimal juhul sai mootor üle 50 korra kiiremaks, halvimal juhul 1,4 korda aeglasemaks. Tehtud muudatuste arvelt suurenes sageli mustri kompileerimiseks kuluv aeg.

Tulemustest järeldub, et otsinguteksti alamosa pealt arvutatud sümbolite sageduste abil on võimalik otsingu kiirust märgatavalt tõsta.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 47 leheküljel, 7 peatükki, 6 joonist, 7 tabelit.

Abstract

Optimizing Match Start Position Searching of RE# Regular Expression Engine for Large Input Texts

RE# is a novel regular expression engine based on symbolic automata and symbolic derivatives of regular expressions. Although its unique approach makes it fast for some patterns, it has not been optimized for most common regular expressions.

The aim of this thesis is to decrease the time it takes for the engine to find all the matches in a large input text. Specifically, the optimizations focus on finding the beginning of the match and use the frequency of symbols in the input text. The expected gain in speed is at most one order of magnitude.

To speed up the engine two main optimization algorithms are created, which use symbol frequencies in the input text to select an uncommon set of symbols from the pattern and use this set to find potential matches.

The results of the optimizations are measured on Estonian and English literary texts and Wikipedia dumps, with sizes between 6MB and 1GB, and the .NET Runtime codebase, which is 401MB.

As a result of the implemented optimizations RE# becomes at most 50 times faster and at worst 1.4 times slower.

Compared to the built-in engines of .NET, RE# becomes competitive, beating the built-in engines in some patterns but losing in others. More widely, RE# becomes comparable to PCRE2, but is still mostly slower than the Rust regular expressions engine.

From these results it can be concluded, that using character frequencies to find matches can be beneficial, even if those frequencies are calculated from a very small part of the input text.

This thesis is written in Estonian and is 47 pages long, including 7 chapters, 6 figures, and 7 tables.

Lühendite ja mõistete sõnastik

.NET	Microsofti loodud tarkvaraarenduse raamistik
JIT	*Just In Time* Käitusaegne
Minterm	Regulaaravaldise töötlemisel leitud sümbolihulk, millel puudub ühisosa ühegi teise mintermiga
Muster	Regulaaravaldis ehk otsingumuster, mida regulaaravaldiste mootor kasutab tekstist vastete otsimiseks
SFA	*Symbolic Finite Automaton* Sümboolne lõplik olekumasin ehk sümbolautomaat

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Regulaaravaldised on üks peamisi tehnoloogiaid, millega tekstist infot leitakse. Regulaaravaldiste mootori abil saab teksti seest leida tekstijuppe, mis vastavad mingile mustrile ehk regulaaravaldisele. Andmemahtude kasvades peavad regulaaravaldiste mootorid muutuma võimekamaks ja kiiremaks.

RE# (varasema nimega SBRE) [1], [2] on üks uuemaid regulaaravaldiste mootoreid. Erinevalt peaaegu kõigist teistest mootoritest, mis põhinevad üksikute sümbolitega töötaval lõplikul olekumasinal, on RE# aluseks SFA (*Symbolic Finite Automaton* Sümboolne lõplik olekumasin ehk sümbolautomaat), mis kasutab üksikute sümbolite asemel sümbolihulki [3]. Tänu sellele toetab mootor mustri koostamisel uusi laiendusi ning on osade mustritüüpide puhul teistest mootoritest palju kiirem. Siiski ei ole RE# veel enamike regulaaravaldiste puhul nii kiire, nagu ta saaks olla.

Töö eesmärk on RE# mootoriga vastete leidmine kiiremaks muuta, keskendudes peamiselt suurtele tekstidele (6MB-1GB). Seda just sellepärast, et siis langeb ühekordsete arvutuste osakaal otsingu ajast ning ilmneb otsingu algoritmi enda kiirus. Töö käigus tuli välja, et enamus aega kulub mootoril vastete alguskohtade leidmiseks, mitte vastete sisu kontrollimiseks, mistõttu on kõik optimisatsioonid suunatud alguskohtade leidmise kiirendamisele. Optimeerimisel lähtun sellest, et mõned sümbolid esinevad otsingutekstis sagedamini kui teised. Töö peamine uurimisküsimus on:

- Kas ja kuidas on võimalik kasutada otsinguteksti sümbolite esinemissagedust vastete alguskohtade otsingu kiirendamiseks?

Vastete leidmise kiirendamise arvelt võib aeglasemaks muutuda mustri kompileerimine. Kuna keskendun suurematele tekstidele ning kuna mootorit saab peale mustri kompileerimist kasutada mitme teksti peal, ei ole mustri kompileerimiseks kulunud aeg märkimisväärne võrreldes vastete otsimiseks kulunud ajaga.

Tulemuste mõõtmiseks võrdlen vastete leidmise kiirust enne ja peale muudatusi. Algseisuks võtan 12.03.2024 tehtud *commiti* (räsi „1d60b06e”[1]). Kiiruse mõõtmiseks kasutan BenchmarkDotNet teeki [4], mille abil saab mootorit võrrelda nii iseendaga kui ka .NET-i

[1] `https://github.com/ieviev/sbre/tree/1d60b06eeb9ba6b30c1d98509ca568b9997db942`

sisse ehitatud mootoriga. Regulaaravaldiste ehk mustrite esitamiseks kasutan käesolevas dokumendis ⌞sellist⌝ vormistust, eeskuju võtsin [5]. Mõõtmisel kasutan järgmisi tekste ja mustreid:

- Mark Twaini kogutud teosed (15MB) [6]. Kasutan [7] mustreid, kuid kuna tööd alustades ei olnud ankrud RE#-is veel täiesti toetatud, kasutan mustri ⌞\b\w+nn\b⌝ asemel mustrit ⌞\w+nn\W⌝.
- Ingliskeelse Vikipeedia esimene alamosa (1GB), eksporditud 20.03.2024 [8]. Mustrid koostan ise, võttes aluseks [7].
- Anton H. Tammsaare teosed, mis digitaalselt saadaval on (6MB) [9]. Mustrid koostan ise, võttes aluseks [7].
- Eestikeelne Vikipeedia (1GB), eksporditud 20.03.2024 [10]. Kasutan samu mustreid, mida kasutas [11], kuid lisasin sümbolihulkadesse ’ž’ ja ’š’ tähed ning mustrisse ⌞([A-Za-z]ina|[A-Za-z]ein)\s⌝ ka teised puuduvad täpitähed[1].
- .NET Runtime koodibaas (401MB) [12]. Võtsin kõik C# koodifailid ning panin need ühte faili kokku. Mustrid koostan ise.

Teiste mootoritega võrdlemiseks kasutan Rebar projekti [13], kus on pandud kokku enimkasutatavad ja kiiremad mootorid ning erinevad tekstid ja mustrid.

Töö eeldatav tulemus on kuni kümnekordne kiirusevõit mõnede mustrite puhul ning väiksem kiiruse tõus ülejäänud mustrite puhul. Mõni muster võib ka veidi aeglasemaks muutuda.

[1] Regulaaravaldistes tähistavad sümbolite vahemikud Unicode’i koodivahemikke. Seega vahemik ⌞a-z⌝ tähistab koodivahemikku 97-122, kus on ainult ladina väiketähed, mistõttu on eestikeelsetes tekstides vaja lisada täpitähed eraldi. Metasümbol ⌞\w⌝ aga juba sisaldab täpitähti.

2 .NET-i otsingu tööriistad

Sõnede seest kiire otsingu teostamiseks on .NET-is olemas mõned tööriistad, mis üritavad otsides kasutada vektoriseerimist. Peamised nendest, mida RE# kasutab, on `Span<T>` ja `SearchValues<T>`.

`Span<T>` objekt laseb andmeid vaadata kui mäluvahemikku [14]. Kuna sõne on lihtsalt sümbolite ehk `char`-ide jada, saab selle kergesti muuta `Span<char>` objektiks ehk sümbolitest koosnevaks mäluvahemikuks. Sellisel kujul hoiab RE# teksti, mille seest ta vasteid otsib.

`Span`-is oleva mälu seest otsimiseks saab kasutada bitioperatsioone, mida on võimalik vektoriseerida. `Span<T>` klassil on meetodid, kus selline otsinguviis on juba implementeeritud. Antud töös on neist tähtsaimad `LastIndexOf` ja `LastIndexOfAny`. Esimene võtab parameetriks teise `Span<T>` objekti, mida esimese seest otsida. Nii saab teksti seest otsida sõne, kui see samuti viia `Span<char>` kujule. Teine meetod võtab parameetriks kas `Span<T>` või `SearchValues<T>` objekti ning otsib mälust viimase elemendi, mis on olemas parameetri sees. Nii saab teksti seest otsida sümbolihulka.

`SearchValues<T>` objekt hoiab andmeid sellisel kujul, et neid saaks võimalikult kiiresti teiste andmete seest otsida [15]. .NET 8-s saab `SearchValues<T>` objekti koostada ainult `byte` või `char` objektidega. Kõige kiirem viis tekstist sümbolihulga otsimiseks on `Span<char>` ja `SearchValues<char>` koos kasutamine.

3 Vastete otsingu algseis

Kui RE# otsib teksti seest vasteid, siis esmalt otsib ta üles kõikide vastete alguskohad, liikudes tekstis paremalt vasakule. Seejärel käib ta need alguskohad üle ja leiab vastete lõppkohad, liikudes vasakult paremale. Selle käigus jäetakse välja sellised alguskohad, mis jäävad mõne eelneva vaste sisse.

Näiteks mustri `⌞a.*⌝` puhul leiab alguskoha otsing tekstist „abab" kaks alguskohta (esimene ja teine „a"), aga vastete lõppude leidmisel jääb teine alguskoht esimese vaste sisse ning tulemuseks on üks vaste, mis katab terve teksti. Suurte sisendtekstide puhul kulub peaaegu kogu otsingu aeg alguskohtade leidmisele.

Alguskohtade leidmise algoritm on tsükkel, mis liigub sümbolhaaval mööda teksti paremalt vasakule. Iga sümboli juures mootor kontrollib, kas see sobib otsitava mustri lõppu. Kui sümbol sobib mustri lõppu, siis võetakse mustrist tuletis praeguse sümboli järgi [16], et saada uus muster.

Näiteks kui otsitakse mustrit `⌞abc⌝` ja teksti seest leitakse sümbol „c", siis on mustri tuletis „c" järgi `⌞ab|abc⌝`, kuna järgmise sümbolina sobiks nii „b", mis jätkaks seda sama vastet, kui ka „c", mis alustaks uut vastet. Kui sümbol mustri lõppu ei sobi, siis liigub mootor algolekusse, st seab otsitavaks mustriks esialgse mustri.

Kuna suurtes tekstides esinevad vasted harva, veedaks täiesti optimeerimata mootor enamuse ajast algolekus. Selle vältimiseks on mõned optimisatsioonid RE#-il olemas. Kui mootor on algolekus, proovib ta kasutada mõnda alguskoha otsingu optimisatsiooni, et ebasobivad sümbolid vahele jätta.

3.1 Mustrist sufiksi leidmine

Iga olemasoleva optimisatsiooni aluseks on üks kahest mustri sufiksist. Need kaks sufiksit on:

- Kindel sufiks — jada sümbolihulki, mis peavad kindlasti vaste lõpus esinema. Näiteks mustri `⌞a[Bb]c⌝` kindel sufiks on $\langle\{a\},\{B,b\},\{c\}\rangle$ ja mustri `⌞(ABC|abc)de⌝` kindel sufiks on $\langle\{d\},\{e\}\rangle$.
- Potentsiaalne sufiks — jada sümbolihulki, kus iga sümbolihulga esinemine võib,

aga ei pruugi, tähendada mustrile sobiva sufiksi leidmist. Mustri `⌞a[Bb]c⌝` potentsiaalne sufiks on sama mis selle kindel sufiks, aga mustri `⌞(ABC|abc)de⌝` potentsiaalne sufiks on $\langle\{A, a\}, \{B, b\}, \{C, c\}, \{d\}, \{e\}\rangle$. See sufiks ei ole „kindel", kuna see sobitub ka sõnega „AbCde", mis algsele mustrile ei vasta.

3.2 Olemasolevad alguskoha otsingu optimisatsioonid

Mootoris on olemas kolm alguskoha otsingu optimisatsiooni, millest üks otsib sõne ja kaks otsivad sümbolihulki. Igal optimisatsioonil on kaks etappi. Esimene etapp on optimisatsiooni koostamine ning see toimub siis, kui mootorile antakse muster. Teine etapp on optimisatsiooni rakendamine ja see toimub siis, kui mootor otsib tekstist vasteid.

3.2.1 Sõne otsiv optimisatsioon

Sõne kasutav optimisatsioon põhineb kindlal sufiksil. Optimisatsiooni koostades võtab mootor kindla sufiksi ja leiab selle lõpust kõik järjestikused sümbolihulgad, milles on ainult üks sümbol. Nendest sümbolihulkadest paneb mootor kokku sõne, mille ta viib otsimiseks üle `Span<char>` kujule.

Optimisatsiooni rakendades saadakse sisendiks alustekst ja mootori praegune positsioon. Tekstis edasi liikumiseks kasutatakse .NET-i sisse ehitatud otsingumeetodeid, et leida tekstist järgmine koht, kus esineb optimisatsiooni koostamisel leitud sõne. Kuna optimisatsioon kasutab kindlat sufiksit, saab mootor ka need sümbolid vahele jätta, mis otsitava sõne sees on.

3.2.2 Sümbolihulki otsivad optimisatsioonid

Kui mustri lõpus sõne pole, siis kasutatakse ühte kahest sümbolihulkadega töötavatest optimisatsioonidest. Nende kahe peamine erinevus on see, et üks põhineb kindlal sufiksil ja teine potentsiaalsel sufiksil. Mõlemad optimisatsioonid võtavad koostamise hetkel sisse neile vastava sufiksi.

Optimisatsiooni rakendades võtavad nad mõlemad sufiksi viimase sümbolihulga ja otsivad seda teksti seest .NET-i otsingumeetoditega. Kui vastav sümbol leitakse, siis käiakse tsükliga läbi kõik ülejäänud sufiksi sümbolihulgad ning kontrollitakse, kas need esinevad järjest peale leitud positsiooni. Kui mõni sümbolihulk ei vasta tekstile, siis käivitatakse

uuesti viimase sümbolihulga otsing. Kui aga tsükkel lõpeb edukalt, siis tegutsevad optimisatsioonid veidi erinevalt. Nimelt saab kindlal sufiksil põhinev optimisatsioon sufiksis olevad sümbolid vahele jätta, aga potentsiaalsel sufiksil põhinev optimisatsioon peab laskma mootoril ka need sümbolid üle vaadata.

3.3 Profileerimine

Vaatamata optimisatsioonidele kulub siiski valdav enamus ajast alguskohtade leidmisele. Selle leidmiseks kasutasin profileerimist. Kuna .NET kasutab JIT (*Just In Time* Käitusaegne) kompileerimist, panin mootori vastete otsingu tsüklisse ja profileerisin seda.

Mark Twaini teksti peal kulus paljudel mustritel üle 98% ajast alguskohtade leidmiseks. Alguskoha otsingu optimisatsioonidele kulus 70%-99% ajast. Kohati teeb optimisatsioonide kasutamine isegi otsingu veidi aeglasemaks, näiteks mustriga `Huck[a-zA-Z]+|Saw[a-zA-Z]+`. See näitab, et olemasolevad optimisatsioonid on liiga aeglased.

4 Optimeerimine

Alguskoha otsingule uute optimisatsioonide tegemisel lähtusin sellest, et mõned sümbolid esinevad tekstis harvemini kui teised. Praegused optimisatsioonid otsivad vektoriseeritud tööriistadega neid sümboleid, mis on sufiksi lõpus. Minu optimisatsioonid valivad või koostavad regulaaravaldisest sellised sümbolihulgad, mis esinevad otsingutekstis harvemini ning otsivad tekstist esmalt neid hulki.

4.1 Sümbolite sageduse leidmine

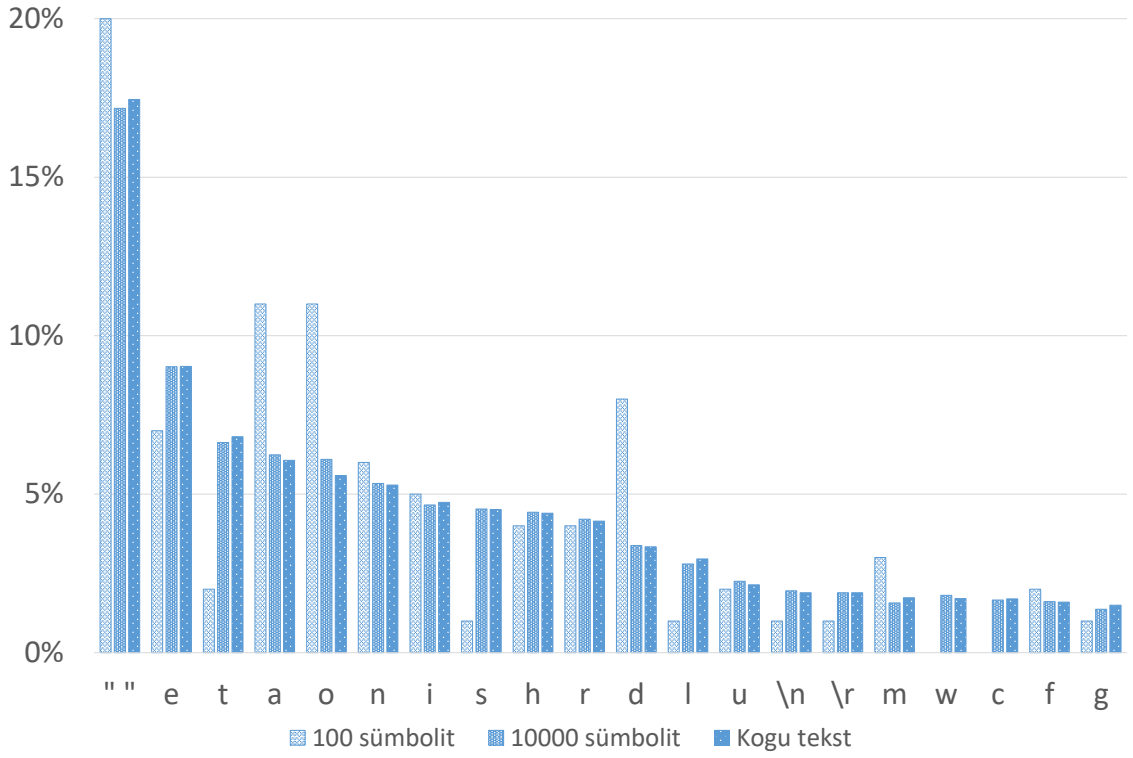
Sümbolihulkade esinemissageduse leidmiseks on esmalt vaja teada üksikute sümbolite esinemissagedusi ehk kaale. Need aga olenevad tekstist, mille seest vasteid otsitakse. Sümbolite kaalude leidmiseks tegin kolm varianti.

Esiteks võib mootori kasutaja määrata kaalud käsitsi. Selleks peab ta mootorile andma sõnastiku, kus võtmeteks on sümbolid ja sümboliga seotud väärtus on selle kaal ehk esinemissagedus tekstis (`Dictionary<char,float>`). Selleks, et mootor optimisatsioonide vahel õigesti valiks, peab sümboli kaal olema antud protsendina ehk vahemikus 0-100. Kasutasin kaaludeks protsente, kuna need on kasutajasõbralikumad kui normaliseeritud arvud.

Teine variant on anda mootorile ette mingi tekst ning lasta tal selle põhjal kaalud arvutada. Kuna suure teksti seest iga sümboli kokku lugemine on aeglane, peab ütlema ka kui mitut sümbolit on vaja vaadata. Kui see arv on teksti pikkusest väiksem, siis teksti paremaks iseloomustamiseks liigub kaalude arvutamine hüpetega üle terve teksti, mitte ei vaata ainult teksti alguses olevaid sümboleid.

Viimane variant on lasta mootoril automaatselt otsingutekstist kaalud arvutada. Selleks ei pea kasutaja midagi tegema. Kaalude automaatne arvutamine käivitub aga ainult siis, kui teksti pikkus on üle 50 000 sümboli. Katsetades leidsin, et lühemate tekstide puhul võtab uute kaalude leidmine ja nende põhjal optimisatsioonide uuesti koostamine rohkem aega kui nende kasutamine säästab. Kaalude arvutamiseks vaadatakse 10% teksti, kuid maksimaalselt 10 000 sümbolit. Vaadates joonist 1, on see sümbolite arv piisav, et saada üpris täpsed kaalud. Haruldasemate sümbolite puhul on nende kaalude suhteline viga suurem, kuid nii väiksed vahed ei mõjuta mootori kiirust märgatavalt.

Katsetades leidsin, et minu optimisatsioonid töötavad sageli peaaegu sama hästi 100 sümboli pealt arvutatud kaaludega kui terve teksti pealt arvutatud kaaludega. Jooniselt 1 on aga näha, et 100 sümboli pealt arvutatud kaalud erinevad märgatavalt terve teksti kaaludest. Kuigi väheste sümbolite pealt arvutatud kaaludega ei pruugi mootor teada, milline sümbolihulk on parim, saab selliste kaaludega siiski vältida halvimaid sümbolihulki. Mõne mustri puhul võivad paremad kaalud siiski aidata ning suuremate sisendtekstide puhul on rohkemate sümbolite pealt kaalude arvutamine suhteliselt kiire.



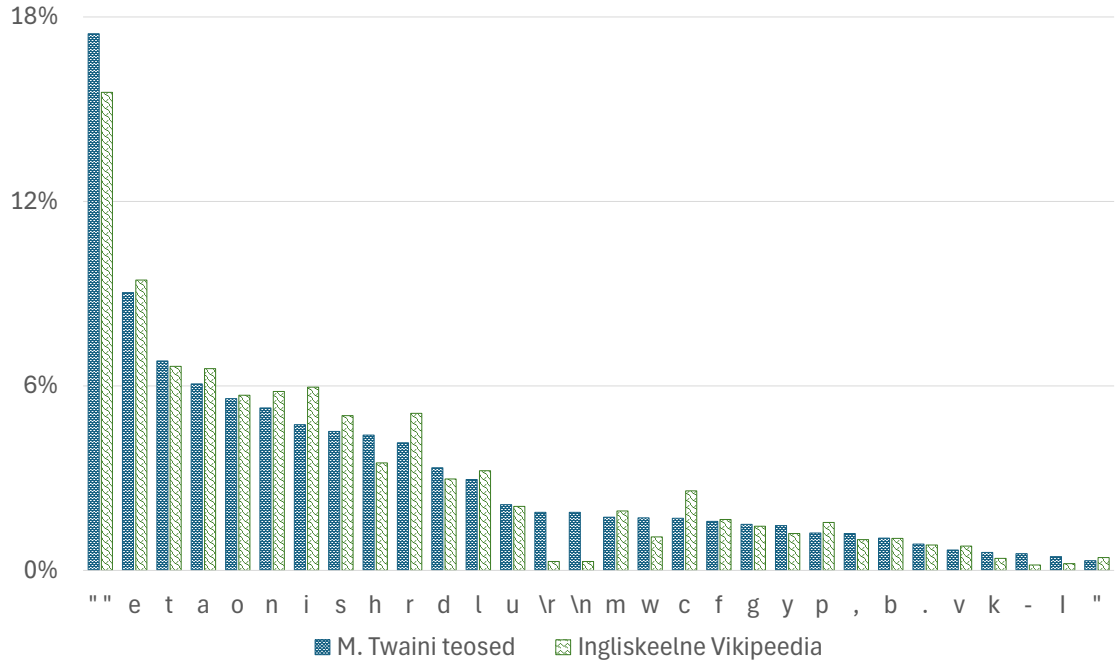
Joonis 1. Mark Twaini teostes 20 kõige sagedamini esineva sümboli osakaalud tekstist. Kaalud on arvutatud 100 sümboli, 10 000 sümboli ja terve teksti pealt. Esimene sümbol on tühik.

Kui mootor leiab sümboli, mille jaoks kaalu ei ole, siis võtab ta selle kaaluks 0. Seda sellepärast, et kui sümbol kaalude arvutamisel ei esinenud, siis on ta arvatavasti antud tekstis suhteliselt haruldane.

4.1.1 Sümbolite sagedused erinevates tekstides

Peaaegu igal tekstil on suuremal või väiksemal määral erinev sümbolite jaotus. Töös uuritud tekstidest on kõige sarnasemad samakeelsed tekstid, kuid ka neil on erinevusi.

Uuritud tekstidest ilmneb, et samakeelsetes teostes on enamike sümbolite sagedused üpris sarnased. Suuremad erinevused on tingitud tekstide vormistamisest. Jooniselt 2 on näha, et Twaini teostes on reavahetused[1] palju sagedasemad kui ingliskeelse Vikipeedia tekstis. Jooniselt 3 paistab, et Tammsaare teostes on rohkem tühikuid ning esinevad eestikeelsed jutumärgid („”).



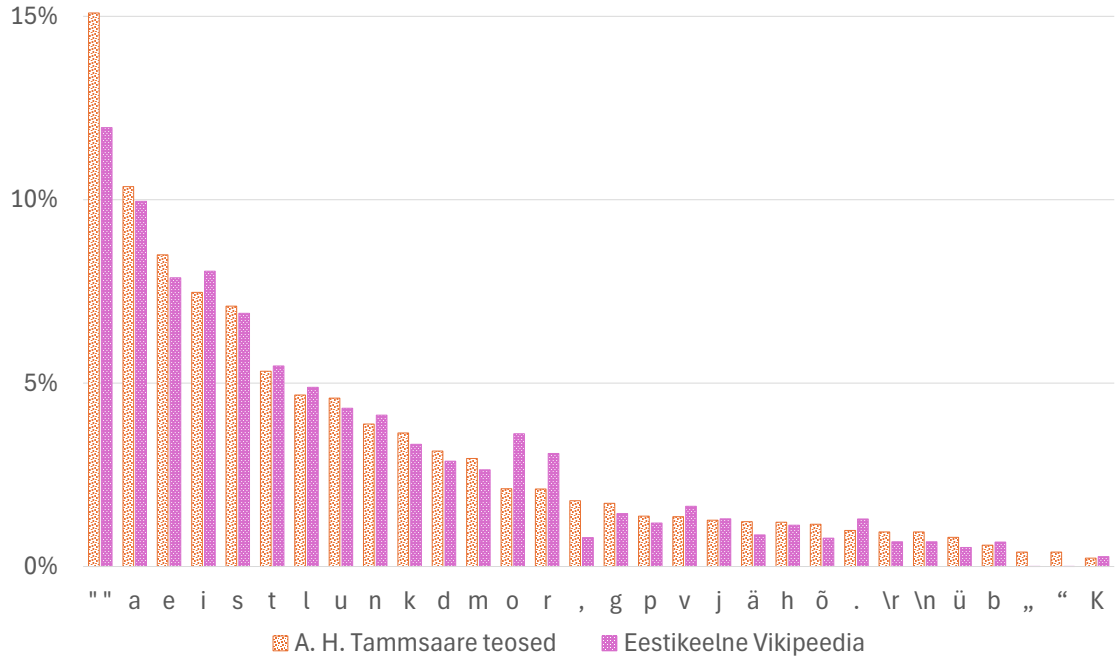
Joonis 2. M. Twaini teostes 30 kõige sagedamini esineva sümboli osakaalud tekstist ja nende samade sümbolite osakaalud ingliskeelse Vikipeedia tekstist.

On ka mõned erinevused, mis ilmselt ei tulene vormistamisest. Eestikeelses Vikipeedias esinevad Tammsaare teostega võrreldes „o” ja „r” palju sagedamini ning komasid esineb harvem.

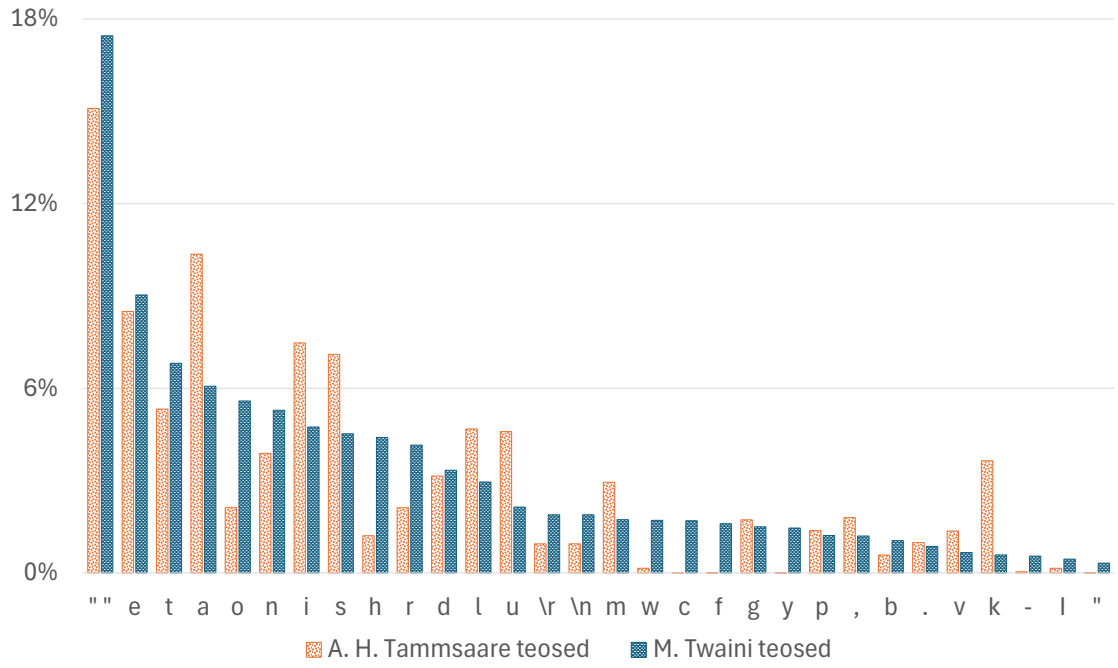
Nagu on näha jooniselt 4, on erikeelsetes tekstides kaalude erinevused suured, kuid leidub ka ühiseid jooni. Nii Tammsaare kui ka Twaini teostes on kõige enam esinev sümbol tühik ning mõlemas tekstis on väiketähed sagedasemad kui suured tähed. Suurimad erinevused tekivad selliste tähtede puhul, mida eestikeelsetes sõnades üldiselt ei leidu.

Kui võrrelda Twaini teoseid ingliskeelse koodiga, siis on erinevused väiksemad kui erikeelsetes tekstides, kuid siiski suured. Joonisel 5 osutub tühik jälle kõige sagedasemaks

[1]Reavahetusi tähistavad siin „\n” ja „\r”. Windows kasutab reavahetuseks neid kahte sümbolit koos, Unix-põhised süsteemid kasutavad ainult „\n” sümbolit.

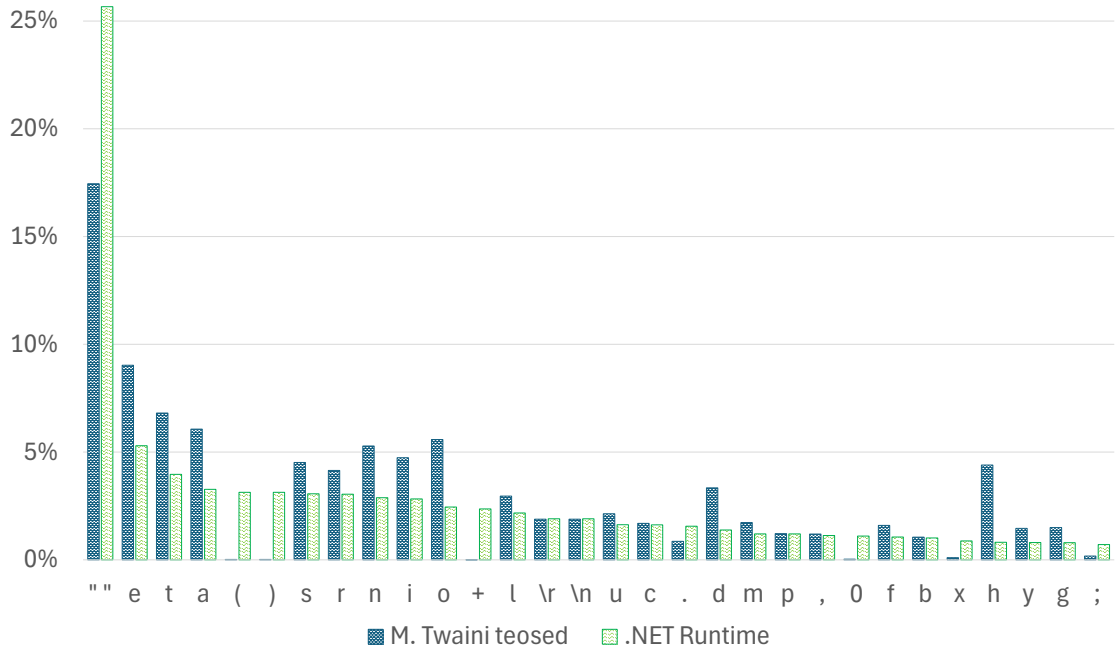


Joonis 3. A. H. Tammsaare teostes 30 kõige sagedamini esineva sümboli osakaalud tekstist ja nende samade sümbolite osakaalud eestikeelse Vikipeedia tekstist.



Joonis 4. M. Twaini teostes 30 kõige sagedamini esineva sümboli osakaalud tekstist ja nende samade sümbolite osakaalud A. H. Tammsaare teostes.

sümboliks, kuid koodis moodustab see lausa veerandi kogu tekstist. Koodis esinevad sageli ka mõned sümbolid, mida kirjanduses enamasti ei esine. Väiketähti vaadates on pilt üpris kirju. Osade tähtede sagedused on mõlemas tekstis peaaegu identsed, osade sagedused aga erinevad mitmekordselt.



Joonis 5. .NET Runtime koodibaasi 30 kõige sagedamini esineva sümboli osakaalud tekstist ja nende samade sümbolite osakaalud M. Twaini teostes.

4.2 Sümbolihulki kaaluv optimisatsioon

See uus optimisatsioon arvutab sufiksi sümbolihulkadele kaalud ning kasutab neid hulkade prioritiseerimiseks. Sellest optimisatsioonist on kaks varianti, millest üks kasutab kindlat sufiksit ja teine potentsiaalset sufiksit (vt peatükk 3.1). Mõlema variandi algoritm on peaaegu identne, ainus vahe on selles, et kindlat sufiksit kasutav variant saab sufiksi sümbolid vahele jätta, aga potentsiaalset sufiksit kasutav variant seda teha ei saa.

Nagu originaalsetel optimisatsioonidel, on ka sellel optimisatsioonil kaks etappi. Algoritmi esimene etapp võtab regulaaravaldise sufiksi ning järjestab selle elemendid ümber, kasutades sümbolite kaale. Kui mootor pannakse teksti seest vasteid otsima, siis käivitub algoritmi teine etapp, mis otsib tekstist sufiksile vastavat kohta.

Algoritmi esimene osa vajab sisendiks sufiksit ja sümbolite kaale. Olgu sümbolihulga

maksimaalne kaal k_m. Algoritmi esimene etapp toimib järgnevalt:

1. Võta sufiksist järjest iga sümbolihulk ψ_i[1] ja selle indeks i.
2. Arvuta sümbolihulga ψ_i kaal k_i:
   - Kui ψ_i on väike (vt peatükk 4.2.1), siis liida kokku kõigi hulgas olevate sümbolite kaalud.
   - Kui ψ_i täiend on väike, siis määra kaaluks $k_m - 1$.
   - Kui ψ_i ega selle täiend ei ole väiksed, siis määra kaaluks k_m.
3. Valmib järjend, kus iga element on kolmik $\langle i, \psi_i, k_i \rangle$.
4. Sorteeri järjendi elemendid kaalude järgi kasvavalt.
5. Viska elementidest kaalud ära, nii et iga element on paar $\langle i, \psi_i \rangle$.

Tulemuseks on järjend Ψ_k, kus elemendid on sorteeritud sümbolihulkade kaalude järgi kasvavalt ning iga element on paar $\langle i, \psi_i \rangle$, kus i on sümbolihulga indeks sufiksis ja ψ_i on sümbolihulk. Koos sümbolihulkadega on vaja hoida alles selle indeks sufiksis, kuna selle järgi teab algoritmi teine osa mis järjekorras sümbolihulgad tekstis esinema peavad.

Kui võtta näiteks muster `␣Huck[A-Za-z]+¬`, mille sufiks on $\langle \{H\}, \{u\}, \{c\}, \{k\}, \{A, B, ..., y, z\} \rangle$, ning kasutada Mark Twaini teoste peal arvutatud sümbolikaale, siis on algoritmi esimese osa tulemuseks järjend $\langle \langle 0, \{H\} \rangle, \langle 3, \{k\} \rangle, \langle 2, \{c\} \rangle, \langle 1, \{u\} \rangle, \langle 4, \{A, B, ..., y, z\} \rangle \rangle$.

Algoritmi teine osa vajab teksti t, mille seest vasteid otsitakse, mootori praegust positsiooni p ja algoritmi esimeses osas leitud sümbolihulkade järjendit Ψ_k. Olgu Ψ_k elementide arv n ja teksti pikkus p_m[2]. Algoritm toimib järgnevalt:

1. Võta Ψ_k esimene sümbolihulk ψ_0 ja selle indeks sufiksi sees i_0.
2. Alustades indeksilt p ja liikudes vasakule, leia t seest esimene sümbol s, mis rahuldab tingimust $s \in \psi_0$.
   - Kui sellist sümbolit ei leidu, siis algoritm lõpeb ja tagastab positsiooni 0.

[1] Tegelikult tähistab ψ (psii) sümbolihulka kuulumise predikaati, kuid lihtsuse eesmärgil kasutan seda sümbolite hulga enda tähistamiseks.

[2] Mootor liigub tekstis mööda tekstivahesid, seega p_m on ka mootori maksimaalne positsioon tekstis.

3. Kui sümbol leidub, siis olgu s indeks p_s. Sufiksi alguskoht tekstis on $p_{s0} \coloneqq p_s - i_0$ ja lõppkoht $p_{sn} \coloneqq p_{s0} + n$.
    - Kui $p_{s0} < 0$ või $p_m < p_{sn}$, siis ei mahu sufiks leitud positsioonile. Väärtusta $p \coloneqq p_s$ ja naase punkti 2.
4. Võta Ψ_k seest järjest iga sümbolihulk ψ_i ja selle indeks i.
5. Sümbolihulgale ψ_i vastab tekstis sümbol s_i, mis asub tekstis indeksil $p_i \coloneqq p_{s0} + i$.
    - Kui $s_i \notin \psi_i$, siis väärtusta $p \coloneqq p_s$ ja naase punkti 2.
6. Kui iga kontrollitud sümbol kuulus talle vastavasse sümbolihulka, siis tagasta p_{sn}.

Kui otsitakse mustrit `␣Huck[A-Za-z]+¬` ja algoritmi esimeses osas leitud järjend on $\langle\langle 0, \{H\}\rangle, \langle 3, \{k\}\rangle, \langle 2, \{c\}\rangle, \langle 1, \{u\}\rangle, \langle 4, \{A, B, ..., y, z\}\rangle\rangle$, siis otsib mootor esmalt tekstist „H”. Selle mustri puhul leiab mootor, et sufiksi alguskoht on sama mis „H” positsioon. Seejärel vaatab mootor, kas sufiksi alguskohast kolm sümbolit paremal on „k”. Nii kontrollib mootor kõik järjendis olevad sümbolihulgad läbi ning kui need sobitusid tekstiga, siis tagastab ta sufiksi lõpu indeksi.

Katsetades leidsin, et kõige rohkem mõjutab kiirust see, kas esimene sümboliklass ψ_0 on hästi valitud. Kui otsisin sufiksist ainult haruldaseima sümbolihulga välja ning jätsin ülejäänud algsesse järjekorda, siis enamus kiirusevõidust säilis. Seega on kõigi sümbolihulkade sorteerimine kasulik, kuid kõige tähtsam on esimesena otsida võimalikult head sümbolihulka.

4.2.1 Suurte sümbolihulkade käsitlemine

Mootor hoiab sümbolihulki mitmel kujul, et erinevates operatsioonides saaks kasutada kõige sobivamat varianti. Teksti seest otsimiseks on kõige parem sümbolihulki hoida `SearchValues<char>` objektidena, kuna nendega on võimalik vektoriseeritud otsingut teostada. Mõned sümbolihulgad on selleks aga liiga suured. Kui mootor sümbolihulki koostab, jaotab ta hulgad kolme kategooriasse:

- Väike sümbolihulk – selles hulgas on piisavalt vähe sümboleid, et sellega saab võibolla teostada vektoriseeritud otsingut.
- Väikse täiendiga sümbolihulk – on väike arv sümboleid, mis antud hulka ei kuulu. Hulka mittekuuluvatest sümbolitest saab koostada `SearchValues<char>` objek-

ti ning siis teksti peal teostada ümberpööratud otsingut, st otsida sümboleid, mis hulgas puuduvad. Selline otsing on iseenesest sama kiire kui väikse sümbolihulga otsing, aga tõenäoliselt leidub sellisele otsingule palju rohkem vasteid kui väikse sümbolihulga otsingule. See omakorda tähendab, et otsing peatub sagedamini ja mootor peab rohkem ülejäänud sümbolihulki kontrollima.

- Suur sümbolihulk – selle hulgaga vektoriseeritud otsingut ei ole peaaegu kindlasti võimalik teha. Mootoris on piiriks võetud 1024 sümbolit.

Suurt või väikse täiendiga sümbolihulka kasutan teksti seest otsimiseks ainult sel juhul, kui ühtegi paremat sümbolihulka ei leidu. Nende kahe vahel valides eelistan ümberpööratud sümbolihulka, kuna see toetab vektoriseeritud otsingut.

4.2.2 Ainult osade sümbolihulkade kontrollimine

Iga kord kui esimese sümbolihulga otsing peatub, kontrollitakse ka teisi sufiksi sümboleid. See on optimisatsioonis kõige rohkem jooksutatav koodi osa, mistõttu oleks kasulik see võimalikult kiireks teha. Üks võimalus selle tööd vähendada oleks osade sümbolihulkade kontrollimata jätmine. Võib arvata, et kui ära on kontrollitud 90% sümbolihulkadest, siis viimast 10%, milles on kõige sagedamini esinevad sümbolid, pole enam mõttekas kontrollida.

Tegelikult see aga kiirust märgatavalt ei tõsta. Praktikas ei ole sümboli hulka kuuluvuse kontroll eriti kulukas operatsioon, seega ei muutu tsükkel kiiremaks. Seevastu suureneb optimisatsioonis leitavate valepositiivide arv ning seeläbi mootori alguskoha otsingu tsükli töö. Testitud mustrite puhul muutub mootor veidi aeglasemaks, kui kontrollida ainult kahte sümbolihulka, ning tunduvalt aeglasemaks, kui kontrollida ainult ühte. Muul juhul kiirus märgatavalt ei muutunud.

4.3 Alternatsiooni optimisatsioon

Üks sageli esinev mustritüüp on alternatsioon, näiteks `⌞Finn|Tom⌝`, mistõttu on mõttekas teha selle jaoks eraldi optimisatsioon. Eelneva mustri sufiks on $\langle\{i, T\}, \{n, o\}, \{n, m\}\rangle$ ning tõenäoliselt oleks neist kõige haruldasem hulk $\{i, T\}$. Mustrile peale vaadates on aga näha, et alternatsiooni esimeses harus „Finn” on täht „F”, mis on tõenäoliselt haruldasem kui „i”. Seega oleks parem otsida hoopis sümbolihulka $\{F, T\}$, kuna see esineb tekstis

harvemini kui $\{i, T\}$.

Erinevalt teistest optimisatsioonidest ei kasuta alternatsioonide optimisatsioon olemasolevat sufiksit, vaid leiab ise mustri harud ning koostab nende põhjal otsimiseks sümbolihulga. Mustri `⌞Finn|Tom⌝` puhul on harudeks „Finn” ja „Tom”, keerulisematest mustritest harude leidmist selgitab peatükk 4.3.1.

Peale harude leidmist toimib see optimisatsioon kahes etapis, nagu ka teised optimisatsioonid. Esimeses etapis koostatakse optimisatsioon ning teises etapis kasutatakse seda teksti seest vastete leidmiseks. Algoritmi esimene etapp vajab mustri harude järjendit ja sümbolite kaale. See algoritmi osa toimib järgnevalt:

1. Loo tühi sümbolihulk ψ_0, kuhu hakkad koguma kõige haruldasemaid sümboleid ja tühi sõnastik M, kus võtmeks on sümbol ja väärtuseks on järjend.
2. Võta järjest iga mustri haru Ψ_h.
3. Leia Ψ_h seest minimaalse kaaluga sümbolihulk ψ_0. Sümbolihulkade kaalude arvutamiseks tegutse samamoodi, nagu sümbolihulki kaaluva optimisatsiooni esimese etapi 2. sammus.
4. Võta ψ_0 seest iga sümbol s.
   - Kui M võtmete hulgas s puudub, siis lisa see ja määra väärtuseks tühi järjend.
   - Võta M seest sümbolile s vastav järjend ning lisa sinna objekt, milles on Ψ_h, sümboli s indeks Ψ_h algusest i_a ning s kaugus Ψ_h lõpust i_l.

Valmib haruldaseimate sümbolite hulk ψ_0 ja sümboleid harudega siduv sõnastik M. Sõnastik M on vajalik selleks, et oleks võimalik teada millisele harule tekstist leitud sümbol viitab. Haruga koos on vaja salvestada s kaugused mõlemast haru otsast, et teksti seest oleks võimalik leida haru algus ja lõpp.

Erinevalt teistest optimisatsioonidest, ei liigu selle algoritmi teine osa tekstis koos mootoriga, vaid sellest eraldi. Teistes algoritmides on mustri lõpp leitud sümbolist alati sama kaugel, olenemata sellest mis sümbol leitakse. Näiteks kui `⌞Finn|Tom⌝` puhul otsitakse sümbolihulka $\{i, T\}$, siis vahet pole kas alguskoha optimisatsioon leiab „i” või „T”, mustri lõpp peab mõlemal juhul olema kaks sümbolit paremal.

Kui aga alternatsiooni optimisatsioon otsib teksti „baBXab” seest mustrit `⌞b.X|B.a⌝`, siis

on otsitav sümbolihulk $\{X, B\}$. Selle mustri esimesele harule tekstis vastet pole, aga teisele harule vastab „BXa". Optimisatsiooni otsing algab paremalt otsast ja peatub indeksil 3 tähe „X" peal. See positsioon antakse mootorile, mis astub kaks sammu vasakule, jõudes indeksile 1, enne kui avastab, et tegelikult seal vastet ei ole. Kui optimisatsioon alustaks uut otsingut sealt, kus mootor peatus, siis ei leiakski ta ühtegi vastet. Selle asemel peab ta jätkama positsioonilt 3, kus ta viimati peatus.

Algoritmi teine etapp vajab otsinguteksti t, esimeses etapis leitud sümbolihulka ψ_0 ja sõnastikku M ning alternatsiooni optimisatsiooniga viimati leitud sümboli positsiooni p_a. Olgu teksti pikkus p_m. Algoritmi teine etapp toimib järgnevalt:

1. Alustades indeksilt p_a ja liikudes vasakule, leia t seest esimene sümbol s, mis rahuldab tingimust $s \in \psi_0$.
2. Loo muutuja p_v algväärtusega -1.
3. Võta M seest s järgi harude järjend.
4. Käi läbi järjendi kõik elemendid. Järjendi elemendis on haru Ψ_h, sümboli s indeks haru vasakust otsast i_a ja kaugus paremast otsast i_l.
5. Olgu s indeks tekstis p_s, haru alguskoht tekstis $p_{s0} \coloneqq p_s - i_a$ ja haru lõpp tekstis $p_{sn} \coloneqq p_s + i_l$.
   - Kui $p_{s0} < 0$ või $p_m < p_{sn}$, siis ei mahu haru leitud positsioonile ja tuleb võtta järgmine haru.
6. Kontrolli, kas tekstis vahemikus p_{s0} kuni p_{sn} olevad sümbolid vastavad Ψ_h sümbolihulkadele.
   - Kui jah ning kui $p_{sn} > p_v$, siis väärtusta $p_v \coloneqq p_{sn}$ ja $p_a \coloneqq p_s$.
7. Kui järjendi kõik harud on läbi vaadatud, siis on p_v väärus kas -1, misjuhul tuleb algoritmi uuesti alustada, või mõni muu arv, misjuhul tuleb see arv tagastada.

See optimisatsioon annab kõige paremaid tulemusi siis, kui alternatsiooni harudes on olemas mingid haruldased sümbolid, mis varem esmase otsingu sümbolihulgast välja jäid. Avaldises `⌞Finn|Tom⌝` on haru „Finn", mille kõige haruldasem sümbol „F" jäi enne sufiksist üldse välja, kuid nüüd on ta kaasatud hulka, mille järgi mootor teostab vektoriseeritud otsingut. Kui aga võtta muster `⌞finn|tom⌝`, siis selle mustri puhul ei pruugi kiirusevõit nii märgatav olla, kuna kõik mustri sümbolid esinevad ingliskeelses tekstis üpris sageli.

4.3.1 Mustri harude leidmine

Mõne mustri puhul on harude leidmine lihtne, näiteks mustris `⌞Finn|Tom⌝`. Kui aga võtta muster `⌞([A-Za-z]awyer|[A-Za-z]inn)\s⌝`, siis on harude leidmine keerulisem.

Kui mootorile antakse muster, siis teeb ta sellest puu laadse struktuuri. Puu tipud on kas sümbolihulgad või sümbolihulki siduvad struktuurid. Puu kaared ühendavad sümbolihulki siduvate struktuuridega. Näiteks on mustril `⌞ab⌝` kolm tippu ja kaks kaart. Puu juur on *Concat* ehk aheldav struktuur, mis ütleb et kaks tippu peavad tekstis järjest esinema. Antud juhul on nendeks tippudeks sümbolihulgad $\{a\}$ ja $\{b\}$.

Siduvad struktuurid võivad viidata ka teistele siduvatele struktuuridele. Näiteks mustris `⌞abc⌝` aheldab juurtipp sümbolihulka $\{c\}$ ja teist ahelat, mis omakorda ühendab sümbolihulki $\{a\}$ ja $\{b\}$.

Mootoris on olemas 10 tüüpi tippe, kuid harude leidmiseks on kõige tähtsamad järgmised kolm:

- *Concat* ehk aheldamine seob kahte tippu.
- *Or* ehk alternatsioon määrab, et järgmisena sobib ükskõik milline järjendis olev tipp.
- *Loop* ehk tsükkel ütleb, et üks tipp peab korduma mingi arv kordi.

Mustrist harude leidmisel suurendavad harude arvu alternatsioonid ning sellised tsüklid, kus minimaalne ja maksimaalne korduste arv on erinevad. Sellised tsüklid võivad harude arvu väga järsult tõsta, mistõttu otsustasin harude leidmise nende juures peatada. Ainus erand selle puhul tekib siis, kui taoline tsükkel on mustri lõpus. Siis saab võtta tsükli minimaalse korduste arvu, kuna alguskohti otsides on vaja kontrollida ainult seda, et vaste minimaalne versioon oleks olemas. Mootori jaoks on ka „*” tegelikult tsükkel, kus minimaalne korduste arv on 0 ja maksimaalne arv lõpmatu. Haru sees ei saaks seda üldse kaasata, kuna siis suureneks harude arv lõpmatuseni, kuid kui see esineb mustri lõpus, siis saab selle tipu vahele jätta.

4.4 Sõne otsingu optimisatsioon

Üldjuhul on tekstist sõne parem otsida kui sümbolihulka, kuna sõne esineb tekstis harvemini. Algses mootoris oli sellega juba arvestatud ning kui mustri sufiksi lõpus oli sõne, siis

mootor kasutas seda vaste otsimiseks. Kui aga sõne esines sufiksi keskel või alguses, siis seda mootor ei leidnud. Täiendasin sõne leidmist nii, et näiteks mustrist `⌞Huck[A-Za-z]+⌝` leiab mootor sõne „Huck”.

On ka olukordi, kus sõne otsing ei ole kiirem. Näiteks eelneva näite puhul on „H” juba nii haruldane sümbol, et selle järgi otsimine on praktiliselt sama kiire kui sõne järgi otsimine. Kui see sümbol mustris ära vahetada suure sümbolihulga vastu, siis saadud mustris `⌞[A-Za-z]uck[A-Za-z]+⌝` on sõne otsimine kõige kiirem variant. Kui aga vaadata mustrit `⌞H[A-Za-z]ck[A-Za-z]+⌝`, siis on olukord keerulisem. Seda mustrit saab otsida kas sümbolihulga $\{H\}$ või sõne „ck” järgi. Kiirusi mõõtes tuleb välja, et antud juhul on $\{H\}$ järgi otsimine kiirem.

4.5 Optimisatsioonide vahel valimine

Paljude mustrite puhul on saadaval mitu erinevat alguskoha otsingu optimisatsiooni. Kuna mootor saab korraga kasutada ainult ühte optimisatsiooni, on vaja neid kuidagi võrrelda. Sümbolihulki kasutavate optimisatsioonide puhul saab võrrelda nende esimese sümbolihulga kaale, kuid sõnesid kasutavate optimisatsioonide puhul võrdlus nii lihtne ei ole.

Eelnevalt vaadatud mustrist `⌞H[A-Za-z]ck[A-Za-z]+⌝` tekib kaks otsingu varianti: otsinguks saab kasutada kas ühe sümboliga sümbolihulka $\{H\}$ või sõne „ck”. Antud juhul on sõne järgi otsing aeglasem, kuna „ck” esineb tekstis üpris sageli, mistõttu leidub selle järgi otsides palju valepositiive. Algoritmiliselt on aga nende kahe variandi vahel valiminine üpris keeruline.

Kõige lihtsam variant oleks käsitleda iga sümboli esinemist kui sõltumatut juhuslikku sündmust ning seeläbi arvutada sõnes olevate sümbolite järjest esinemise tõenäosus. Tegelikult aga sümboli esinemine tekstis ei ole sõltumatu sündmus ning sellise lihtsustuse tegemine ei anna head tulemust. See, et „ck” esineb tekstis sageli, ei kajastu otseselt üksikute sümbolite „c” ja „k” esinemise sagedustes. Seda on selgelt näha, kui pöörata sõne ümber ja vaadata selle esinemissagedust. Sõne „ck” esineb Mark Twaini teostes 17 293 korda, aga sõne „kc” esineb ainult 18 korda. Seega ei saa ainult üksikute sümbolite kaalude järgi sõnele täpset kaalu anda.

Sõnele „ck” täpse kaalu leidmiseks oleks vaja teada tingliku tõenäosust, et peale „c”-d esineb „k”. Siis saaks selle tõenäosuse ja „c” esinemise tõenäosuse abil leida „ck” esine-

mise tõenäosuse. Iga sümbolipaari tingliku tõenäosuse leidmine on aga üpris kulukas ning pikemate sõnede puhul oleks vaja veel keerulisemaid tõenäosusi. Ei ole praktiline otsida välja kõikide teksti sümbolite permutatsioonide esinemissagedused.

Seega on vaja leida ligikaudne heuristika üksikute sümbolite kaalude põhjal. See heuristika peab rahuldama kolme tingimust:

1. Selle väljund peab alati olema kindlas vahemikus, et seda oleks võimalik skaleerida sümbolite kaaludega samasse vahemikku.
2. Selle väljund peab olema sõne haruldaseima sümboli kaaluga võrdne või sellest väiksem. On ilmselge, et sõne „xaaa" peab tekstis esinema sama harva või harvemini kui sõne „x".
3. Heuristika peaks siiski arvestama ka ülejäänud sümbolite kaaludega, st „xxaa" peaks eestikeelses tekstis saama madalama kaalu kui „xaaa".

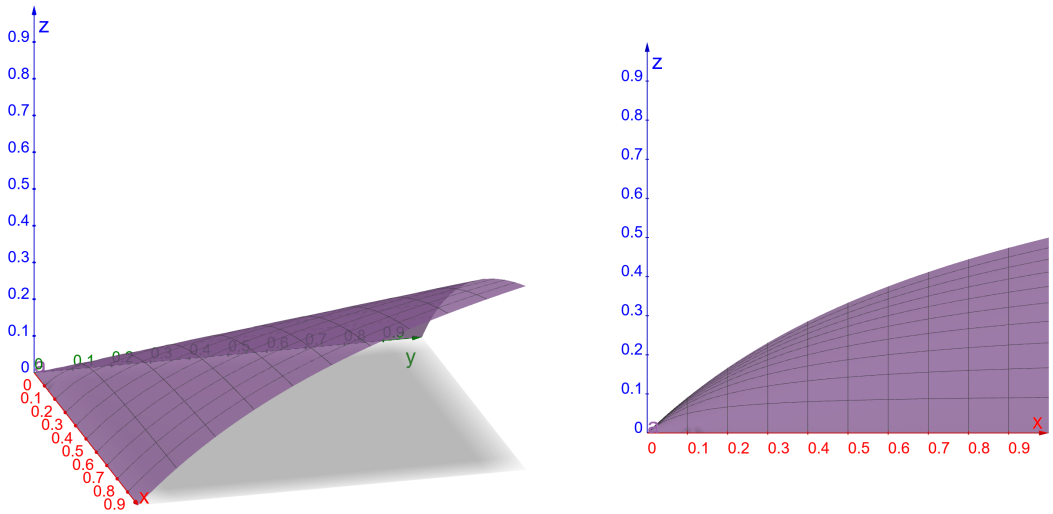
Neid tingimusi rahuldab Jaccardi indeksil [17] põhinev heuristika, kus sõne sümbolite kaalude korrutis jagatakse sümbolite kaalude summaga (valem 1). Selle kasutamiseks sümbolite kaaludega on vaja sisendid normaliseerida ehk jagada sajaga ja väljund sajaga korrutada, et see tagasi protsendiks muuta.

$$k_{str} = \frac{\prod_i^n k_i}{\sum_i^n k_i} \tag{1}$$

Joonisel 6 on graafiliselt kujutatud kahe sümboli pikkuse sõne puhul kasutatav heuristika. Selle parempoolses osas on näha, et iga x-teljel oleva sisendkaalu puhul on sellele vastav maksimaalne väljund (z-telje väärtus) sisendkaaluga võrdne või sellest väiksem. Samuti on näha, et valemi tulemus on alati üle nulli ja alla ühe. Tegelikult jääb tulemus isegi alla 0,5, kuid kui see võtta ülempiiriks ja tulemus selle järgi normaliseerida, siis võib väljund muutuda sisenditest suuremaks.

Selle heuristika abil saab võrrelda sõnesid omavahel, aga ka sõnesid sümbolihulkadega. Seega on võimalik omavahel võrrelda kõiki mootori optimisatsioone. Tuleb märkida, et sellist heuristikat ei saa kasutada sõnega, mis on ühe sümboli pikkune. Sellisel puhul tuleb seda käsitleda kui ühe sümboliga sümbolihulka.

On ka mustreid, mille puhul ei tasu nii põhjalikke kontrolle teha. Näiteks mustris `␣Twain¬`



Joonis 6. Jaccardi indeksil põhinev heuristika kahe sümboli pikkuse sõnega. Normaliseeritud sümbolite kaalud on x- ja y-telgedel ning väljund on z-teljel.

tasub ilmselgelt kasutada sõne otsingut. Selliste mustrite puhul ei ole sümbolite kaalude arvutamine ja teiste optimisatsioonide koostamine kasulik. Seetõttu tegin erandi, mis kontrollib mustri lõpus sõne olemasolu ja pikkust ning kui see sõne on neli või rohkem sümbolit pikk, siis sümbolite kaale ega teisi optimisatsioone mootor ei arvuta. Valisin miinimumiks nelja sümboliga sõne, kuna enamasti on selle kaal Jaccardi indeksi järgi võrreldav teksti haruldasemate sümbolite kaaludega.

5 Tulemuste analüüs

Mootori kiiruse mõõtmiseks kasutasin peamiselt BenchmarkDotNet teeki [4], mis muuhulgas likvideerib JIT kompileerimise mõju ning leiab tulemustest aritmeetilise keskmise, vea ja mediaani. Lisaks RE#-i algsele ja optimeeritud versioonile, mõõtsin võrdluseks ka .NET-i sisse ehitatud mootorit kahe konfiguratsiooniga. Sisseehitatud mootori *non-backtracking* variant väldib tekstis tagasi liikumist ning *compiled* variant kompileerib regulaaravaldised masinkoodile sarnasemasse vahekeelde. Need kiirustestid jooksutasin TalTechi AI Labi masinal, mille protsessor on 24-tuumaline AMD Threadripper 3960X ning millel on on 128GB mälu.

Teistes keeltes kirjutatud mootoritega võrdlemiseks kasutasin Rebar projekti [13]. Võtsin võrdluseks Rusti regulaaravaldiste mootori [18] ning PCRE2 mootori [19], kuna need on Rebari projektis koos .NET-i mootoriga kiireimate hulgas. Need kiirustestid jooksutasin masinal, mille protsessor on 6-tuumaline Intel Core I5-11600K ning millel on 32GB mälu.

Mark Twaini teoste mustrid võtsin [7] ja eestikeelse Vikipeedia mustrid võtsin [11]. Nende mustrite eeskujul koostasin teiste tekstide mustrid.

Tulemuste tabelites on kiireim tulemus **paksus kirjas**. Lisaks absoluutsele kiirusele on tabelites kordajaga välja toodud suhteline kiirus võrreldes optimeeritud RE#-ga.

5.1 M. Twaini ja A. H. Tammsaare teosed

Mark Twaini teoste [6] peal tehtud kiirustestide tulemused on Tabelis 1. Tabelis on tulemuste aritmeetiline keskmine, mille maksimaalne viga oli 2% ning kus mediaan erines keskmisest maksimaalselt 2,32%. Teksti maht on 15MB. A. H. Tammsaare teoste [9] peal jooksutatud kiirustestide tulemused on tabelis 2. Kirjas on aritmeetiline keskmine, mille maksimaalne viga oli 4,55% ja kus mediaan erines aritmeetilisest keskmisest maksimaalselt 8,14%. Teksti maht on 6MB.

Mõlema teksti puhul on minu optimeeritud mootor algsest versioonist enamike mõõdetud mustrite korral kiirem. Twaini tekstis on ainult üks muster algses mootoris kiirem, Tammsaare teostes on selliseid mustreid kolm.

Parimatel juhtudel on optimeeritud mootor mitukümmend korda kiirem. Üks

suurim kiirusevõit Tammsaare tekstis on peaaegu 50 kordne muutus mustris `˻Andr[A-ZŠŽÕÄÖÜa-zšžõäöü]+|Pear[A-ZŠŽÕÄÖÜa-zšžõäöü]+˺`. Siin otsis algne mootor mustri lõpus olevat suurt sümbolihulka, optimeeritud mootor aga otsib mustri alguses olevaid suuri tähti. Siit mustrist paremat sümbolihulka ei leidu, mistõttu ongi kiirusevõit nii suur. Mustris `˻Huck[a-zA-Z]+|Saw[a-zA-Z]+˺` on samal põhjusel 40 kordne kiirusevõit.

Mustris `˻\w+nn\W˺` otsib algne mootor tekstist väga suurt sümbolihulka „\W", optimeeritud mootor aga leiab mustrist sõne „nn" ja otsib seda.

Märkimisväärsed kiirusevõidud on ka alternatsioonide mustritel, mis on enamasti 7-8 korda kiiremad. Väiksem võit on tõstutundetu alternatsiooni puhul, kus parim sümbolihulk pole teisest nii palju parem kui tõstutundlikus mustris.

Mustrid, mis kasutavad algset sõne optimisatsiooni, minu muudatustest kiirust juurde ei saanud. Aeglasemaks jäänud mustrid on enamjaolt sellised, kus mustri lõpus olev sümbolihulk on juba kõige haruldasem.

.NET-i sisseehitatud mootorid on sageli RE#-i optimeeritud versioonist veidi kiiremad, kuid kohati ka palju aeglasemad. *Compiled* mootor on parimal juhul umbes 3,7 korda kiirem, aga halvimal juhul umbes 43 korda aeglasem. *Non-backtracking* variant on parimal juhul 3,1 korda kiirem, aga halvimal juhul 41 korda aeglasem. Kohati võib sisseehitatud mootoritele probleeme põhjustada see, et nad liiguvad tekstis vasakult paremale.

Üldiselt on optimeeritud RE#-i tulemused ühtlasemad, teistel on seevastu mõned üksikud mustrid teistest märgatavalt aeglasemad.

Tabel 1. Mark Twaini teoste kiirustestid, aritmeetiline keskmine.

Regulaaravaldis	.NET Non-backtracking (ms)	.NET Compiled (ms)	RE# Algne (ms)	RE# Optimeeritud (ms)
(?i)Tom|Sawyer|Huckleberry|Finn	55.740 (2.52x)	**20.902 (0.94x)**	42.025 (1.90x)	22.125 (1.00x)
(?i)Twain	2.782 (0.38x)	**2.640 (0.36x)**	17.277 (2.34x)	7.379 (1.00x)
([A-Za-z]awyer|[A-Za-z]inn)\s	12.092 (0.70x)	**9.620 (0.56x)**	40.717 (2.36x)	17.229 (1.00x)
.{0,2}(Tom|Sawyer|Huckleberry|Finn)	43.147 (8.35x)	225.115 (43.54x)	36.898 (7.14x)	**5.170 (1.00x)**
.{2,4}(Tom|Sawyer|Huckleberry|Finn)	42.748 (8.41x)	220.205 (43.35x)	37.640 (7.41x)	**5.080 (1.00x)**
Huck[a-zA-Z]+|Saw[a-zA-Z]+	2.093 (0.66x)	**1.885 (0.59x)**	127.765 (40.01x)	3.193 (1.00x)
Tom.{10,25}river|river.{10,25}Tom	7.735 (1.00x)	**6.294 (0.81x)**	25.580 (3.29x)	7.773 (1.00x)
Tom|Sawyer|Huckleberry|Finn	3.659 (0.80x)	**3.504 (0.77x)**	37.025 (8.09x)	4.575 (1.00x)
Twain	2.064 (0.97x)	**1.985 (0.93x)**	2.147 (1.01x)	2.133 (1.00x)
[”’][^”’]{0,30}[?!\.][”’]	8.653 (1.25x)	**5.515 (0.80x)**	7.207 (1.04x)	6.911 (1.00x)
[a-q][^u-z]{13}x	2.398 (0.61x)	**1.745 (0.45x)**	2.782 (0.71x)	3.910 (1.00x)
[a-zA-Z]+ing	107.518 (8.37x)	98.174 (7.64x)	13.707 (1.07x)	**12.853 (1.00x)**
[a-z]shing	2.247 (1.06x)	**2.076 (0.98x)**	2.287 (1.08x)	2.116 (1.00x)
\s[a-zA-Z]{0,12}ing\s	56.260 (4.00x)	113.119 (8.05x)	43.135 (3.07x)	**14.052 (1.00x)**
\w+nn\W	99.713 (37.70x)	106.657 (40.32x)	119.654 (45.24x)	**2.645 (1.00x)**

Tabel 2. A. H. Tammsaare teoste kiirustestid, aritmeetiline keskmine.

Regulaaravaldis	.NET Non-backtracking (ms)	.NET Compiled (ms)	RE# Algne (ms)	RE# Optimeeritud (ms)
(?i)Karl|Andres|Pearu|Vanapagan	27.142 (2.69x)	15.449 (1.53x)	19.232 (1.91x)	**10.087 (1.00x)**
(?i)Tammsaare	1.056 (0.32x)	**0.904 (0.27x)**	7.114 (2.16x)	3.294 (1.00x)
([A-ZŠŽÕÄÖÜa-zšžõäöü]earu|[A-ZŠŽÕÄÖÜa-zšžõäöü]ndres)\s	3.774 (0.74x)	**2.530 (0.50x)**	18.755 (3.70x)	5.072 (1.00x)
.{0,2}(Karl|Andres|Pearu|Vanapagan)	17.368 (7.68x)	84.643 (37.45x)	17.687 (7.82x)	**2.260 (1.00x)**
.{2,4}(Karl|Andres|Pearu|Vanapagan)	17.614 (7.82x)	90.160 (40.03x)	17.688 (7.85x)	**2.252 (1.00x)**
Andr[A-ZŠŽÕÄÖÜa-zšžõäöü]+|Pear[A-ZŠŽÕÄÖÜa-zšžõäöü]+	0.984 (0.77x)	**0.613 (0.48x)**	60.758 (47.28x)	1.285 (1.00x)
Karl|Andres|Pearu|Vanapagan	1.505 (0.71x)	**1.196 (0.56x)**	17.787 (8.38x)	2.123 (1.00x)
Madis.{10,25}kraav|kraav.{10,25}Madis	3.574 (1.02x)	**2.697 (0.77x)**	9.068 (2.58x)	3.512 (1.00x)
Tammsaare	**0.387 (0.96x)**	0.672 (1.66x)	0.443 (1.09x)	0.405 (1.00x)
[A-ZŠŽÕÄÖÜa-zšžõäöü]+ne	43.790 (11.72x)	40.442 (10.82x)	3.844 (1.03x)	**3.737 (1.00x)**
[a-q][^u-z]{13}w	0.753 (0.59x)	**0.525 (0.41x)**	0.960 (0.75x)	1.282 (1.00x)
[a-zšžõäöü]mine	1.065 (1.19x)	0.927 (1.03x)	**0.850 (0.95x)**	0.898 (1.00x)
[„”’][^“”’]{0,30}[?!\.][“”’]	4.615 (1.18x)	**1.843 (0.47x)**	3.606 (0.92x)	3.916 (1.00x)
\s[A-ZŠŽÕÄÖÜa-zšžõäöü]{0,12}ne\s	26.776 (9.97x)	49.282 (18.34x)	18.919 (7.04x)	**2.687 (1.00x)**
\w+nn\W	41.533 (41.52x)	41.037 (41.02x)	48.381 (48.36x)	**1.000 (1.00x)**

5.2 Vikipeedia

Eestikeelse Vikipeedia teksti [10] peal tehtud kiirustestide tulemused on tabelis 3 ja ingliskeelse Vikipeedia alamosa [8] peal tehtud kiirustestide tulemused on tabelis 4. Eestikeelse teksti peal oli maksimaalne viga 2,9% ja mediaan erines keskmisest maksimaalselt 2,25%. Ingliskeelse teksti peal oli maksimaalne viga 1,99% ja mediaan erines keskmisest maksimaalselt 1,14%. Mõlemad tekstid on 1GB suurused.

Mõlema teksti peal on optimeeritud RE# iga mustriga kas sama kiire või kiirem kui mootori algne versioon. Ilmselt amortiseeruvad nii pika teksti puhul minu optimisatsioonidest tulenevad ühekordsed kulud ka nende mustrite peal, kus neist vähem kasu on.

Võrreldes teiste tekstidega on optimeeritud RE#-i võidud algse mootori üle kohati veidi väiksemad. Paari mustriga on optimeeritud mootor küll mitukümmend korda kiirem, kuid neist järgmine suurim võit on 8 ja 6 kordne.

.NET-i mootorite parimad tulemused on nende tekstide puhul RE#-ist veidi kaugemal ees kui eelmistes tekstides. Parimal juhul on nad 5,9 korda kiiremad. Halvimal juhul on *compiled* versioon 68 korda aeglasem ja *non-backtracking* versioon 41 korda aeglasem.

Kui arvestada mustrit `␣["'][^"']{0,30}[?!\.]["']¬`, siis on ingliskeelse Vikipeedia peal optimeeritud RE# kiireim 7-l korral 15-st, mis on vaadatud tekstidest parim tulemus.

Tabel 3. Eestikeelse Vikipeedia kiirustestid, aritmeetiline keskmine.

Regulaaravaldis	.NET Non-backtracking (ms)	.NET Compiled (ms)	RE# Algne (ms)	RE# Optimeeritud (ms)
(?i)Eesti	220.000 (0.20x)	**184.900 (0.17x)**	1207.900 (1.11x)	1089.900 (1.00x)
(?i)Toomas|Margus|Rein|Jaan	3195.700 (2.91x)	**881.400 (0.80x)**	1673.500 (1.52x)	1098.800 (1.00x)
([A-Za-zšžüõöäŠŽÜÕÖÄ]ina|[A-Za-zšžüõöäŠŽÜÕÖÄ]ein)\s	1291.700 (1.02x)	**975.800 (0.77x)**	2388.700 (1.89x)	1262.600 (1.00x)
.{0,2}Toomas|Margus|Rein|Jaan	3235.900 (10.50x)	15492.400 (50.25x)	1757.700 (5.70x)	**308.300 (1.00x)**
.{2,4}Toomas|Margus|Rein|Jaan	3115.500 (10.10x)	15694.100 (50.86x)	1749.700 (5.67x)	**308.600 (1.00x)**
Eesti	199.800 (0.97x)	**159.900 (0.78x)**	227.500 (1.11x)	205.700 (1.00x)
Eesti.{10,25}jõgi|jõgi.{10,25}Eesti	308.100 (0.91x)	**227.600 (0.67x)**	1327.300 (3.94x)	337.300 (1.00x)
Heli[a-zA-ZšžüõöäŠŽÜÕÖÄ]+|Aja[a-zA-ZšžüõöäŠŽÜÕÖÄ]+	156.900 (0.74x)	**138.700 (0.66x)**	10330.800 (49.01x)	210.800 (1.00x)
Rootsi	143.700 (1.02x)	**134.000 (0.95x)**	158.400 (1.12x)	141.400 (1.00x)
Toomas|Margus|Rein|Jaan	267.400 (0.86x)	**232.300 (0.75x)**	1763.000 (5.70x)	309.200 (1.00x)
["'][^"']{0,31}[?!\.]["']	364.700 (1.60x)	314.300 (1.38x)	243.900 (1.07x)	**227.700 (1.00x)**
[a-q][^u-z]{12}x	231.800 (0.66x)	**155.600 (0.44x)**	390.500 (1.11x)	350.600 (1.00x)
[a-zA-ZšžüõöäŠŽÜÕÖÄ]+tud	5656.200 (17.68x)	4888.300 (15.28x)	351.000 (1.10x)	**320.000 (1.00x)**
[a-zšžüõöä]ee	348.100 (0.85x)	**227.100 (0.55x)**	434.200 (1.06x)	410.500 (1.00x)
\p{Sc}	2542.600 (0.83x)	**788.100 (0.26x)**	3077.900 (1.00x)	3074.100 (1.00x)
\s[a-zA-ZšžüõöäŠŽÜÕÖÄ]{0,12}tud\s	4248.600 (14.55x)	4766.700 (16.33x)	2345.400 (8.03x)	**291.900 (1.00x)**

Tabel 4. Ingliskeelse Vikipeedia kiirustestid, aritmeetiline keskmine.

Regulaaravaldis	.NET Non-backtracking (ms)	.NET Compiled (ms)	RE# Algne (ms)	RE# Optimeeritud (ms)
(?i)Lincoln	484.100 (0.77x)	**413.000 (0.66x)**	947.600 (1.52x)	624.900 (1.00x)
(?i)Lincoln|Washington|Roosevelt|Jefferson	3352.500 (2.68x)	**1086.400 (0.87x)**	1639.400 (1.31x)	1251.300 (1.00x)
([A-Za-z]incoln|[A-Za-z]oosevelt)\s	613.300 (0.86x)	**519.900 (0.73x)**	1982.400 (2.79x)	709.900 (1.00x)
.{0,2}(Lincoln|Washington|Roosevelt|Jefferson)	2927.300 (11.45x)	17188.100 (67.25x)	1699.800 (6.65x)	**255.600 (1.00x)**
.{2,4}(Lincoln|Washington|Roosevelt|Jefferson)	2879.200 (11.26x)	17625.600 (68.96x)	1688.600 (6.61x)	**255.600 (1.00x)**
Linc[a-zA-Z]+|Roo[a-zA-Z]+	160.700 (0.75x)	**145.800 (0.68x)**	9037.600 (42.04x)	215.000 (1.00x)
Lincoln	139.600 (0.99x)	**132.400 (0.94x)**	149.600 (1.06x)	140.700 (1.00x)
Lincoln|Washington|Roosevelt|Jefferson	230.300 (0.92x)	**203.700 (0.81x)**	1689.800 (6.72x)	251.400 (1.00x)
Roosevelt.{10,25}river|river.{10,25}Roosevelt	699.200 (1.90x)	589.600 (1.60x)	1761.200 (4.79x)	**367.900 (1.00x)**
["'][^"']{0,30}[?!\.]["']	334.100 (1.59x)	276.700 (1.32x)	**209.100 (1.00x)**	209.900 (1.00x)
[a-q][^u-z]{13}x	3823.000 (14.31x)	**135.200 (0.51x)**	288.100 (1.08x)	267.100 (1.00x)
[a-zA-Z]+ing	6588.300 (10.18x)	5502.000 (8.50x)	696.300 (1.08x)	**647.200 (1.00x)**
[a-z]shing	156.600 (0.99x)	**147.500 (0.93x)**	160.100 (1.01x)	158.100 (1.00x)
\s[a-zA-Z]{0,12}ing\s	4175.000 (7.08x)	4595.400 (7.79x)	2009.800 (3.41x)	**589.800 (1.00x)**
\w+nn\W	6443.200 (41.20x)	6497.300 (41.54x)	8570.900 (54.80x)	**156.400 (1.00x)**

5.3 .NET Runtime koodibaas

.NET Runtime koodibaasi [12] peal tehtud kiirustestid on tabelis 5. Tabelis on aritmeetiline keskmine, maksimaalne viga oli 1,99% ja mediaan erines keskmisest maksimaalselt 1,47%. Teksti maht on 401MB.

Nendes kiirustestides muutus RE# peale optimeerimist peaaegu iga mustriga kas kiiremaks või jäi sama kiireks nagu varem. Halvimal juhul langes kiirus mõne protsendi võrra. Parimal juhul tõusis kiirus 28,5 korda.

Suurim kiirusevõit on mustrites `⌞\WI[A-Z][A-Za-z0-9_]+⌝` ja `⌞\\u[A-Fa-f0-9]{4,4}⌝`, kus jällegi on mustri lõpus suur sümbolihulk, kuid mustri sees leidub palju väiksem ja harvemini esinev sümbolihulk.

Regulaaravaldises `⌞[^.]\.\.[^.]⌝` otsis mootor algselt mustri lõpus olevat sümbolihulka. Nüüd aga otsib ta kahest punktist koosnevat sõne.

Alternatsioonide puhul on suurim võit kõige pikemas mustris. Seal leiab alternatsiooni optimisatsioon, et igas harus on „R” kõige haruldasem sümbol. Seega on esimeses otsinguhulgas ainult üks sümbol, mis on vektoriseeritud otsingu jaoks kasulik. Sama alternatsiooni lühemas versioonis valitakse aga igast harust erinev sümbol. Tõstutundetus alternatsioonis ning mustris `⌞public|private|protected|internal⌝` esineb olukord, kus alternatsiooni optimisatsioon leiab küll algseisuga võrreldes parema sümbolihulga, kuid see ei ole nii palju parem kui teistes alternatsioonides.

Nende mustrite puhul on enamasti kiireim .NET-i *compiled* mootor. Parimal juhul on see optimeeritud RE#-ist üle 8 korra kiirem. Siiski on ka siin paar mustrit, kus sisseehitatud mootorid on palju aeglasemad. Muster `⌞[A-Za-z0-9_]*__[A-Za-z0-9_]*⌝` on sümmeetriline, seega ei ole siin üht või teist pidi tekstis liikumisel vahet. Paistab, et sisseehitatud mootorid lihtsalt ei suuda otstes olevatest tsüklitest edasi vaadata, et näha keskel olevaid alakriipse. Seetõttu on nad selle mustriga kas 19 või 22 korda aeglasemad.

Tabel 5. .NET Runtime koodibaasi kiirustestid, aritmeetiline keskmine.

Regulaaravaldis	.NET Non-backtracking (ms)	.NET Compiled (ms)	RE# Algne (ms)	RE# Optimeeritud (ms)
(?i)Regex	51.740 (0.54x)	**50.380 (0.52x)**	93.970 (0.98x)	96.330 (1.00x)
(?i)\.Compiled|\.CultureInvariant|\.None| \.NonBacktracking	109.810 (0.25x)	**88.400 (0.20x)**	707.290 (1.58x)	447.240 (1.00x)
Regex	**51.030 (0.95x)**	51.800 (0.96x)	51.770 (0.96x)	53.880 (1.00x)
RegexOptions\.Compiled|RegexOptions\.CultureInvariant| RegexOptions\.None|RegexOptions\.NonBacktracking	51.740 (0.65x)	**47.060 (0.59x)**	603.170 (7.58x)	79.590 (1.00x)
String[A-Za-z]+|Regex[A-Za-z]+	81.030 (0.68x)	**66.030 (0.56x)**	2034.330 (17.18x)	118.420 (1.00x)
[A-Za-z0-9_]*__[A-Za-z0-9_]*	980.770 (19.91x)	1101.960 (22.37x)	**48.950 (0.99x)**	49.250 (1.00x)
[A-Za-z0-9_]+Attribute	1542.360 (23.22x)	1079.930 (16.26x)	72.740 (1.10x)	**66.410 (1.00x)**
[^.]\.\.[^.]	**42.460 (0.12x)**	42.720 (0.12x)	2590.470 (7.39x)	350.480 (1.00x)
\.Compiled|\.CultureInvariant|\.None|\.NonBacktracking	79.860 (0.51x)	**66.770 (0.43x)**	602.880 (3.85x)	156.620 (1.00x)
\WI[A-Z][A-Za-z0-9_]+	80.390 (0.96x)	**56.720 (0.68x)**	2412.070 (28.78x)	83.800 (1.00x)
\\u[A-Fa-f0-9]{4,4}	51.060 (0.94x)	**42.620 (0.78x)**	1067.580 (19.64x)	54.350 (1.00x)
_[A-Za-z0-9_]+	229.080 (1.34x)	**88.950 (0.52x)**	2282.360 (13.40x)	170.320 (1.00x)
public|private|protected|internal	306.140 (0.56x)	**196.860 (0.36x)**	725.470 (1.32x)	548.220 (1.00x)

5.4 Mustri kompileerimine

Tabelis 6 on Mark Twaini teoste peal kasutatud regulaaravaldiste kompileerimiseks kulunud aeg erinevate mootoritega. Kompileerimine tähendab siinpuhul mootori objekti loomist, kuna siis kompileerib mootor talle antud mustri sellisele kujule, mida ta saab otsimiseks kasutada.

Kompileerimise mõõtmiseks lasin igal mootoril ka lühikeses sõnes vaste olemasolu kontrollida. See oli vajalik sellepärast, et osad mootorid ei kompileeri mustrit täielikult sel hetkel, kui see neile antakse, vaid teevad seda jooksvalt otsingu ajal.

Tulemustest paistab, et minu muudatused enamasti aeglustasid RE# mustri kompileerimise kiirust. Halvimal juhul võtab mustri kompileerimine optimeeritud mootoris 3,33 korda rohkem aega. Enamasti on aga kiiruste vahe palju väiksem.

Võrreldes .NET-i mootoritega on RE#-i optimeeritud versioon sageli *compiled* versioonist kiirem, aga *non-backtracking* versioonist aeglasem.

Tabel 6. Mark Twaini teoste mustritega mootorite ehitamiseks kulunud aeg, aritmeetiline keskmine.

Regulaaravaldis	.NET Non-backtracking (ms)	.NET Compiled (ms)	RE# Algne (ms)	RE# Optimeeritud (ms)
(?i)Tom|Sawyer|Huckleberry|Finn	1.397 (0.43x)	**1.226 (0.37x)**	3.324 (1.01x)	3.280 (1.00x)
(?i)Twain	**0.112 (0.37x)**	0.534 (1.77x)	0.188 (0.62x)	0.302 (1.00x)
([A-Za-z]awyer|[A-Za-z]inn)\s	**0.427 (0.40x)**	1.315 (1.24x)	1.026 (0.97x)	1.060 (1.00x)
.{0,2}(Tom|Sawyer|Huckleberry|Finn)	1.560 (0.39x)	**1.136 (0.28x)**	3.989 (1.00x)	3.998 (1.00x)
.{2,4}(Tom|Sawyer|Huckleberry|Finn)	1.575 (0.38x)	**1.187 (0.28x)**	4.222 (1.01x)	4.195 (1.00x)
Huck[a-zA-Z]+|Saw[a-zA-Z]+	**0.286 (0.39x)**	1.178 (1.59x)	0.764 (1.03x)	0.742 (1.00x)
Tom.{10,25}river|river.{10,25}Tom	**0.308 (0.10x)**	1.523 (0.47x)	3.245 (1.01x)	3.224 (1.00x)
Tom|Sawyer|Huckleberry|Finn	1.430 (0.44x)	**1.359 (0.41x)**	3.362 (1.02x)	3.281 (1.00x)
Twain	**0.113 (0.37x)**	0.352 (1.16x)	0.188 (0.62x)	0.302 (1.00x)
["'][^"']{0,30}[?!\.]["']	**0.095 (0.07x)**	0.831 (0.61x)	1.316 (0.96x)	1.369 (1.00x)
[a-q][^u-z]{13}x	**0.098 (0.14x)**	0.741 (1.07x)	0.208 (0.30x)	0.694 (1.00x)
[a-zA-Z]+ing	**0.101 (0.38x)**	0.986 (3.72x)	0.192 (0.72x)	0.265 (1.00x)
[a-z]shing	**0.183 (0.42x)**	0.567 (1.29x)	0.293 (0.67x)	0.438 (1.00x)
\s[a-zA-Z]{0,12}ing\s	**0.180 (0.20x)**	1.321 (1.48x)	0.739 (0.83x)	0.891 (1.00x)
\w+nn\W	1.297 (0.79x)	**1.184 (0.72x)**	1.625 (0.98x)	1.652 (1.00x)

5.5 Rebar

Mootori kiiruse võrdlemiseks teistes keeltes kirjutatud mootoritega kasutasin Rebari projekti [13]. Rebaris tehtud kiirustestide tulemused on tabelis 7. Kõige kiirem mootor on peaaegu iga mustriga Rusti mootor, kuid mõne mustri puhul on teised mootorid esikohale üpris lähedal. Lisas 2 on optimeeritud RE#-i võrreldud veel teiste mootoritega.

Võrreldes algseisuga, on optimeeritud RE# esikohale palju lähemal. Vaadeldud mustrite hulgas ei ole selle jaoks enam eriliselt aeglaseid mustreid, erinevalt PCRE2 ja .NET-i *compiled* mootoritest. Halvimal juhul on see esikohast 9 korda aeglasem ja parimal juhul ainult 1,4 korda aeglasem

Mustri `␣[a-q][^u-z]{13}x␣` puhul on RE# kiirem kui Rusti mootor. See võib tuleneda sellest, et Rusti mootor ei oska esmalt otsida mustri lõpus olevat „x”-i, või on tal keerulisem sümbolihulkadega tegutseda.

Sõnede alteratsiooni mustris `␣Tom|Sawyer|Huckleberry|Finn␣` on Rusti mootor kiire tänu Teddy [20] ja Aho-Corasic [21] algoritmidele, mis võimaldavad korraga vektoriseeritult mitut sõne otsida. Tõstutundetus alternatsioonis `␣(?i)Tom|Sawyer|Huckleberry|Finn␣` on nendest algoritmidest aga vähem kasu, mistõttu Rusti edumaa on seal väiksem.

Tabel 7. Vastete leidmise kiirus erinevates mootorites, aritmeetiline keskmine.

Regulaaravaldis	PCRE2	Rust	.NET Compiled	RE# Algne	RE# Optimeeritud
Twain	895.23 us (0.56x)	**798.09 us (0.50x)**	1.52 ms (0.95x)	1.63 ms (1.02x)	1.60 ms (1.00x)
(?i)Twain	**1.03 ms (0.14x)**	1.44 ms (0.20x)	2.17 ms (0.30x)	13.53 ms (1.86x)	7.26 ms (1.00x)
[a-z]shing	1.38 ms (0.79x)	**1.10 ms (0.63x)**	1.65 ms (0.95x)	1.81 ms (1.04x)	1.74 ms (1.00x)
Huck[a-zA-Z]+|Saw[a-zA-Z]+	**1.05 ms (0.38x)**	1.07 ms (0.38x)	1.88 ms (0.67x)	130.54 ms (46.79x)	2.79 ms (1.00x)
\w+nn\W	42.43 ms (20.70x)	**1.46 ms (0.71x)**	96.27 ms (46.96x)	101.37 ms (49.45x)	2.05 ms (1.00x)
[a-q][^u-z]{13}x	90.91 ms (30.61x)	3.79 ms (1.28x)	**1.46 ms (0.49x)**	2.29 ms (0.77x)	2.97 ms (1.00x)
Tom|Sawyer|Huckleberry|Finn	9.88 ms (2.00x)	**1.14 ms (0.23x)**	2.91 ms (0.59x)	35.93 ms (7.26x)	4.95 ms (1.00x)
(?i)Tom|Sawyer|Huckleberry|Finn	36.19 ms (1.58x)	**15.84 ms (0.69x)**	21.12 ms (0.92x)	41.54 ms (1.82x)	22.88 ms (1.00x)
.{0,2}(Tom|Sawyer|Huckleberry|Finn)	119.42 ms (24.22x)	**1.26 ms (0.26x)**	188.45 ms (38.23x)	36.98 ms (7.50x)	4.93 ms (1.00x)
.{2,4}(Tom|Sawyer|Huckleberry|Finn)	139.14 ms (30.51x)	**1.31 ms (0.29x)**	190.89 ms (41.86x)	37.63 ms (8.25x)	4.56 ms (1.00x)
Tom.{10,25}river|river.{10,25}Tom	2.52 ms (0.33x)	**1.41 ms (0.19x)**	6.54 ms (0.86x)	20.45 ms (2.70x)	7.57 ms (1.00x)
[a-zA-Z]+ing	42.79 ms (3.48x)	**7.07 ms (0.58x)**	92.98 ms (7.57x)	12.05 ms (0.98x)	12.29 ms (1.00x)
\s[a-zA-Z]{0,12}ing\s	54.90 ms (4.46x)	**7.38 ms (0.60x)**	102.10 ms (8.29x)	45.62 ms (3.70x)	12.32 ms (1.00x)
([A-Za-z]awyer|[A-Za-z]inn)\s	8.86 ms (0.63x)	**1.48 ms (0.11x)**	12.22 ms (0.87x)	41.24 ms (2.93x)	14.08 ms (1.00x)
[\"'][^\"']{0,30}[?!\.][\"']	6.66 ms (0.99x)	**4.43 ms (0.66x)**	5.47 ms (0.82x)	6.82 ms (1.02x)	6.71 ms (1.00x)

6 Võimalikud edasiarendused

On veel mitmeid meetodeid, millega minu tehtud optimisatsioone täiendada või mootorit muul viisil kiiremaks teha.

Sümbolihulki kaaluva optimisatsiooni puhul saaks sümbolihulkade järjestamisel arvestada sellega, kuidas `SearchValues` objekt sümboleid otsima hakkab. Näiteks kui objekt luuakse sümbolite vahemikuga, siis võib selle otsimine olla palju kiirem kui mitme erineva sümboli otsimine.

Alternatsioonide optimisatsioonis tasuks üritada otsida harudest korduvaid sümboleid, selle asemel et igast harust eraldi kõige haruldasem sümbol valida. Mõne mustri puhul võib see anda märgatava kiirusetõusu. Veel saaks proovida mustris olevaid tsükleid harude leidmisel teisiti käsitleda.

.NET 9 laiendab `SearchValues`-it, nii et seda saab luua ka sõnedega. Selleks implementeeriti Teddy algoritm [20] ja Aho Corasick algoritm [21], millega saab teksti seest vektoriseeritult mitut sõne otsida. Seda saaks üpris kergelt kasutada sõnede alternatsioonide otsimiseks. Veidi rohkema tööga saaks seda integreerida ka minu alternatsiooni optimisatsiooni sisse, kuid siis peab harud sümbolihulkade jadadest ümber sõnedeks tegema, mis ei ole alati võimalik. Saaks ka ise implementeerida teisi sarnaseid algoritme, näiteks uuema Harry algoritmi [22].

Otsingut kiirendaks võib-olla ka see, kui otsigutekst viia üle teisele kujule. Praegu töötab mootor .NET-i `char`-idega, mis on 16-bitised, kuid enamjaolt ASCII sümbolitest koosneva teksti puhul oleks parem kasutada 8-bitilisi sümboleid.

Kuna mootor põhineb sümbolihulkadel, saaks teksti sümbolid muuta mintermideks. Minterm on regulaaravaldisest leitud minimaalne sümbolihulk, millel pole ühisosa teiste mintermidega. Näiteks on mustril `⌞ab[a-z]⌝` neli mintermi: $\{a\}$, $\{b\}$, $\{c, d, ..., y, z\}$ ja viimasena kõikide teiste sümbolite hulk, mis eelnevatesse hulkadesse ei kuulu. Enamike mustrite puhul on mintermide arv alla 64, seega saaks sümboleid mälus üpris kompaktselt hoida. Sellise konverteeritud teksti peal on vektoriseerimine palju efektiivsem ja lihtsam, kuna ei pea tõlkima sümbolihulkade ja sümbolite vahel. Sellise lähenemise miinus on aga see, et teksti konverteerimine võtaks tõenäoliselt kauem aega kui otsing muidu võtaks, kuid seda ajakulu oleks võimalik vähendada konverteerimist paralleliseerides.

7 Kokkuvõte

Töö eesmärk oli regulaaravaldiste mootori RE# vastete leidmise kiiruse tõstmine. Ootus oli, et kiirus tõuseb mõne mustriga parimal juhul 10 korda, enamike mustritega vähem.

Töö alguses ilmnes, et valdav enamus vastete otsingu ajast kulus vastete alguskohtade leidmiseks. Seega keskendusin just selle osa optimeerimisele. Selleks kasutasin edukalt otsinguteksti põhjal leitud sümbolite sagedusi ning tegin järgmised tähelepanekud:

- Suurtest tekstidest kõigi vastete leidmisel on tähtis see, millist sümbolihulka või sõne esimesena teksti seest otsida.
- Esimest sümbolihulka valides on kõige tähtsam on vältida halvimaid (kõige sagedamini esinevaid) sümbolihulki. See ilmnes hästi sümbolihulki kaaluvas optimisatsioonis.
- Mõnes mustris tasub otsida ka parim sümbolihulk. See ilmnes alternatsioonide optimisatsioonis.
- Sümbolite ja sümbolihulkade võrdlemiseks tasub lugeda otsingutekstist sümbolite sagedusi ning isegi ainult 100 sümboli vaatamine võimaldab suuri kiirusevõite.

Nende punktide põhjal tehtud optimisatsioonidega suutsin mootori kiirust tõsta kohati üle 50 korra. Enamike mustrite puhul oli kiirusevõit väiksem ning mõne mustri puhul kiirus veidi langes.

Uurimisküsimuse vastuseks saab öelda, et otsinguteksti sümbolite sageduste järgi saab prioritiseerida mustri sufiksis olevaid sümbolihulki või koostada nende põhjal uusi sümbolihulki, mille järgi saab vasteid kiiremini otsida. Sümbolite sageduste järgi saab ka erinevaid optimisatsioone prioritiseerida.

Töös tehtud optimisatsioonid töötavad kõige paremini sellistes olukordades, kus otsingutekst on suur ning vasted esinevad harva. Sobib näiteks suure hulga raamatute seest mustri otsimine või suure koodibaasi refaktoreerimine. Suurimad kiirusevõidud tekivad mootori algversiooniga võrreldes selliste regulaaravaldistega, mille lõpus on suur sümbolihulk või mille sees leidub haruldasi sümboleid.

Katsetatud olukordades on optimeeritud RE# konkurentsivõimeline .NET-i sisse ehitatud

mootoriga ning on algseisuga võrreldes palju lähemal Rusti mootorile, mis on mõõdetud mootoritest kiireim.

Kasutatud kirjandus

[1] I. E. Varatalu, M. Veanes ja J.-P. Ernits, "Derivative Based Extended Regular Expression Matching Supporting Intersection, Complement and Lookarounds", 2023, https://arxiv.org/abs/2309.14401 (vaadatud 11.05.2024). arXiv: 2309.14401 [cs.FL].

[2] I. E. Varatalu, SBRE - Symbolic Boolean Regular Expressions, https://doi.org/10.5281/zenodo.8369590 (vaadatud 11.05.2024), 2023.

[3] L. D'Antoni ja M. Veanes, "The Power of Symbolic Automata and Transducers", teoses Computer Aided Verification, R. Majumdar ja V. Kunčak, toim., https://link.springer.com/chapter/10.1007/978-3-319-63387-9_3 (vaadatud 11.05.2024), Cham: Springer International Publishing, 2017, lk. 47–67, ISBN: 978-3-319-63387-9.

[4] A. Akinshin, BenchmarkDotNet, https://benchmarkdotnet.org/ (vaadatud 11.05.2024).

[5] J. E. F. Friedl, Mastering regular expressions. O'Reilly Media Inc., 2006.

[6] M. Twain, The Entire Project Gutenberg Works of Mark Twain, https://www.gutenberg.org/ebooks/3200 (vaadatud 11.05.2024).

[7] Z. Herczeg, Performance comparison of regular expression engines, https://zherczeg.github.io/sljit/regex_perf.html (vaadatud 11.05.2024), 2015.

[8] Ingliskeelse Vikipeedia arhiivi 1. osa, loodud 20.03.2024, https://wikimedia.bringyour.com/enwiki/ (vaadatud 11.05.2024).

[9] A. H. Tammsaare, Tammsaare digiteeritud teosed, https://et.wikisource.org/wiki/Autor:A._H._Tammsaare (vaadatud 11.05.2024).

[10] Eestikeelse Vikipeedia arhiiv, loodud 20.03.2024, https://dumps.wikimedia.org/etwiki/ (vaadatud 11.05.2024).

[11] B. Yanovich, "Optimisation of Symbolic Automata Based Regular Expression Library SRM Using SIMD Intrinsics", Tallinna Tehnikaülikool, 2021, https://digikogu.taltech.ee/et/Item/d4a9a0af-f8d2-4e57-8da4-2a41c59f9571 (vaadatud 11.05.2024).

[12] Microsoft, .NET Runtime, https://github.com/dotnet/runtime (vaadatud 11.05.2024).

[13] A. Gallant, Rebar, https://github.com/BurntSushi/rebar (vaadatud 11.05.2024).

[14] System.Span<T> struct, https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-span%7Bt%7D (vaadatud 11.05.2024).

[15] SearchValues<T> Class, https://learn.microsoft.com/en-us/dotnet/api/system.buffers.searchvalues-1?view=net-8.0 (vaadatud 11.05.2024).

[16] J. A. Brzozowski, "Derivatives of Regular Expressions", J. ACM, köide 11, nr 4, lk. 481–494, 1964-10, https://doi.org/10.1145/321239.321249 (vaadatud 11.05.2024), ISSN: 0004-5411. DOI: 10.1145/321239.321249.

[17] Jaccard index, https://en.wikipedia.org/wiki/Jaccard_index (vaadatud 11.05.2024).

[18] Rust Foundation, Rust regex, https://github.com/rust-lang/regex (vaadatud 11.05.2024).

[19] P. Hazel, Perl Compatible Regular Expressions, http://www.pcre.org/ (vaadatud 11.05.2024).

[20] K. Qiu, H. Chang, Y. Hong, W. Zhu, X. Wang ja B. Li, "Teddy: An Efficient SIMD-based Literal Matching Engine for Scalable Deep Packet Inspection", teoses Proceedings of the 50th International Conference on Parallel Processing, seeria ICPP '21, https://doi.org/10.1145/3472456.3473512 (vaadatud 11.05.2024), Lemont, IL, USA: Association for Computing Machinery, 2021, ISBN: 9781450390682. DOI: 10.1145/3472456.3473512.

[21] A. V. Aho ja M. J. Corasick, "Efficient string matching: an aid to bibliographic search", Commun. ACM, köide 18, nr 6, lk. 333–340, 1975-06, https://doi.org/10.1145/360825.360855 (vaadatud 11.05.2024), ISSN: 0001-0782. DOI: 10.1145/360825.360855.

[22] H. Xu, H. Chang, W. Zhu et al., "Harry: A Scalable SIMD-based Multi-literal Pattern Matching Engine for Deep Packet Inspection", teoses IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, https://doi.org/10.1109/INFOCOM53939.2023.10229022 (vaadatud 11.05.2024), IEEE, 2023, lk. 1–10. DOI: 10.1109/INFOCOM53939.2023.10229022.

Lisa 1 – Litsents

Mina, Rainer Viirlaid

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose “Regulaaravaldiste mootori RE# vastete alguskohtade leidmise optimeerimine suurte sisendtekstide jaoks”, mille juhendajad on Juhan-Peep Ernits ja Ian Erik Varatalu
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

27.05.2024

Lisa 2 – Optimeeritud RE# võrdlus veel teiste mootoritega

Regulaaravaldis	RE2	JavaScript	Python	Java	RE# Optimeeritud
Twain	**1.19 ms (0.74x)**	1.73 ms (1.08x)	10.29 ms (6.43x)	16.43 ms (10.27x)	1.60 ms (1.00x)
(?i)Twain	**5.77 ms (0.79x)**	9.92 ms (1.37x)	11.95 ms (1.65x)	78.54 ms (10.82x)	7.26 ms (1.00x)
[a-z]shing	17.43 ms (10.02x)	8.54 ms (4.91x)	12.95 ms (7.44x)	117.02 ms (67.25x)	**1.74 ms (1.00x)**
Huck[a-zA-Z]+|Saw[a-zA-Z]+	17.26 ms (6.19x)	8.14 ms (2.92x)	64.85 ms (23.24x)	163.38 ms (58.56x)	**2.79 ms (1.00x)**
\w+nn\W	17.28 ms (8.43x)	106.89 ms (52.14x)	902.87 ms (440.42x)	413.39 ms (201.65x)	**2.05 ms (1.00x)**
[a-q][^u-z]{13}x	73.82 ms (24.86x)	266.75 ms (89.81x)	10.52 ms (3.54x)	219.88 ms (74.03x)	**2.97 ms (1.00x)**
Tom|Sawyer|Huckleberry|Finn	17.60 ms (3.56x)	19.62 ms (3.96x)	76.62 ms (15.48x)	317.18 ms (64.08x)	**4.95 ms (1.00x)**
(?i)Tom|Sawyer|Huckleberry|Finn	**17.82 ms (0.78x)**	42.99 ms (1.88x)	320.45 ms (14.01x)	345.10 ms (15.08x)	22.88 ms (1.00x)
.{0,2}(Tom|Sawyer|Huckleberry|Finn)	17.91 ms (3.63x)	50.35 ms (10.21x)	2.61 s (529.41x)	1.17 s (237.32x)	**4.93 ms (1.00x)**
.{2,4}(Tom|Sawyer|Huckleberry|Finn)	19.00 ms (4.17x)	53.63 ms (11.76x)	2.54 s (557.02x)	1.36 s (298.25x)	**4.56 ms (1.00x)**
Tom.{10,25}river|river.{10,25}Tom	17.45 ms (2.31x)	12.08 ms (1.60x)	89.96 ms (11.88x)	156.31 ms (20.65x)	**7.57 ms (1.00x)**
[a-zA-Z]+ing	32.31 ms (2.63x)	105.82 ms (8.61x)	891.06 ms (72.50x)	238.37 ms (19.40x)	**12.29 ms (1.00x)**
\s[a-zA-Z]{0,12}ing\s	27.05 ms (2.20x)	89.02 ms (7.23x)	46.74 ms (3.79x)	251.31 ms (20.40x)	**12.32 ms (1.00x)**
([A-Za-z]awyer|[A-Za-z]inn)\s	17.27 ms (1.23x)	**13.31 ms (0.95x)**	637.03 ms (45.24x)	397.69 ms (28.25x)	14.08 ms (1.00x)
[\"'][^\"']{0,30}[?!\.][\"']	18.76 ms (2.80x)	14.98 ms (2.23x)	98.12 ms (14.62x)	50.73 ms (7.56x)	**6.71 ms (1.00x)**