TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Andre Ivan 233318IAIB
Marcus-Kevin Otto 233074IAIB

Tekstiotsingu kiirendamine DEFLATE-algoritmiga pakitud failides

Bakalaureusetöö

Juhendaja: Ian Erik Varatalu
MSc

Kaasjuhendaja: Juhan-Peep Ernits
PhD

Tallinn 2026

Autorideklaratsioon

Kinnitame, et oleme 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.

Autorid: Andre Ivan ja Marcus-Kevin Otto

01.06.2026

Lühikokkuvõte

Suuri tekstandmete kogumeid, sealhulgas veebiarhiive, serverite logisid, Git-pakkfaile ja suuri veebikorpuseid nagu Common Crawl, säilitatakse `DEFLATE`-vormingus kokkupakituna. Levinud tööriistad nagu `zgrep` ja `ripgrep` pakivad sisendi lahti olemasolevate lahtipakkijate abil.

Käesolevas töös võrreldakse olemasolevaid tööriistu, analüüsitakse lahtipakkijate sisemist töötamist ning hinnatakse kirjanduses kirjeldatud otsingutehnikaid ja nende implementatsioone.

Grammatikapõhine otsing, mis on akadeemilises kirjanduses tuntud lähenemine kokkupakitud andmetest otsimiseks, osutub `DEFLATE` jaoks halvasti sobivaks: `LZ77` tagasiviidete struktuurist grammatika taastamine nõuab rohkem arvutustööd kui otsing ise, mida see peaks kiirendama.

Samuti testitakse ja analüüsitakse Twins algoritmi, mille eesmärk on vältida otsingumootori olekute uuesti arvutamist. Katsed näitavad, et tugevalt optimeeritud `DEFLATE` lahtipakkijate korral ei anna see jõudluseelist, kuna suurendab lahtipakkimise kulu.

Töö käigus arendatakse voogtöötlusel põhinev implementatsioon, mis ühendab lahtipakkimise ja mustriotsingu samasse protsessi ning saavutab hinnatud testide korral parema jõudluse kui tuntud tööriistad nagu `ripgrep`, `zgrep` ja `ugrep`.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 35 leheküljel, 8 peatükki, 8 joonist, 18 tabelit.

Abstract

Accelerating text search in files compressed with the DEFLATE algorithm

Large volumes of text data, including web archives, server logs, Git packfiles, and large web crawls such as Common Crawl, are stored in `DEFLATE`-compressed form. Common tools such as `zgrep` and `ripgrep` decompress the input using available decompression tools.

The thesis compares existing tools, examines decompressor internals, and evaluates several search techniques and their implementations from the literature.

Grammar-based search, a well-established academic approach to searching compressed data, is shown to be a poor fit for `DEFLATE`: reconstructing a grammar from the `LZ77` backreference structure costs more than the search it is meant to accelerate.

Twins algorithm, a method to skip calculating search engine states is tested and analyzed for performance gain. It is tested to not be beneficial in highly optimized DEFLATE decompressors due to increased cost of unpacking.

A streaming implementation is developed that overlaps decompression with matching and outperforms well-known utilities such as `ripgrep`, `zgrep`, `ugrep` on the evaluated workloads.

The thesis is in Estonian and contains 35 pages of text, 8 chapters, 8 figures, 18 tables.

Lühendite ja mõistete sõnastik

API	rakendusliides;
CVE	Common Vulnerabilities and Exposures: avalike infoturbe haavatavuste identifikaatorite süsteem;
HTTP	Hypertext Transfer Protocol: veebis andmete edastamiseks kasutatav protokoll;
Huffmani kodeering	prefikskoodide süsteem;
JSON	JavaScript Object Notation: tekstipõhine andmevorming;
LZ77	kadudeta sõnastikupõhine pakkimisalgoritm, mis kasutab literaale ja tagasiviiteid;
NVD	National Vulnerability Database: haavatavuste andmebaas;
PCRE	Perl Compatible Regular Expressions: regulaaravaldiste otsingumootor võimsa süntaksiga;
RE#	resharp: automaatidel töötav regulaaravaldiste otsingumootor;
regex	regulaaravaldis ehk märkidest koosnev otsingumuster;
SIMD	Single Instruction, Multiple Data: protsessori käsustik mitme andmeelemendi paralleelseks töötlemiseks;
SLP	Straight-Line Program: grammatikapõhine esitus, mis genereerib täpselt ühe sõne;
WARC	Web ARChive: veebiarhiivide salvestamiseks kasutatav failivorming, mis talletab HTTP-päringuid, vastuseid ja nendega seotud metaandmeid.

Sisukord

Jooniste loetelu 8

Tabelite loetelu 9

1 Sissejuhatus 11

2 Analüüs 13

2.1 Olemasolevad lahendused 13

2.1.1 ripgrep 13

2.1.2 ugrep 14

2.1.3 zgrep 14

2.1.4 zcat 14

2.2 DEFLATE 15

2.2.1 LZ77 15

2.2.2 Huffmani kodeering 15

2.3 Lahtipakkijate implementatsioonid 16

2.3.1 zlib 16

2.3.2 zlib-ng 16

2.3.3 zlib-rs 16

2.3.4 libdeflate 17

2.3.5 zune-inflate 17

2.4 Lahtipakkijate võrdlus 17

2.4.1 Metoodika 18

2.4.2 Tulemused 18

2.5 Otsingumootor 19

2.5.1 RE# 19

2.5.2 Rust Regex 19

3 LZ77 teisendamine grammatika kujule 21

3.1 Grammatika kui andmestruktuur 21

3.2 Chomsky normaalkuju 22

3.3 Straight-Line Program ... 22
3.4 gzip faili teisendus grammatikaks ... 22
3.5 Mõõtmismetoodika ... 24
3.6 Tulemused ... 25
3.7 Teisenduskulu ... 26
3.8 Järeldused ... 27
4 Vooglahendus ... 28
4.1 Twins algoritm ... 28
4.1.1 Lihtsustatud prototüüp ... 29
4.1.2 Integreerimine optimeeritud lahtipakkijasse ... 29
4.2 Lõplik vooglahendus ... 30
4.2.1 Puhvri suuruse mõju ... 31
5 Tulemuste analüüs ... 34
5.1 Eestikeelse Vikipeedia pealkirjad ... 35
5.2 Ingliskeelse Vikipeedia pealkirjad ... 37
5.3 Debiani faililoend ... 39
5.4 Common Crawli WARC andmestik ... 40
5.5 Tulemuste kokkuvõte ... 41
6 Kokkuvõte ... 43
Kasutatud kirjandus ... 46
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks ... 49
Lisa 2 – Tulemuste tabelid ... 50

Jooniste loetelu

Joonis 1. Töötlemise voog gzipist mustriotsinguni.. 23

Joonis 2. Grammatika ajakulu faaside kaupa. 27

Joonis 3. Twins algoritmi näide korduva sisendi ja automaadi olekute võrdlemisest. 29

Joonis 4. Voogtöötluse lahendus. 32

Joonis 5. Eestikeelse Vikipeedia pealkirjade andmestiku peal `defgrep`i suhtelised kiirused. Kiirus on arvutatud kiireima võrdlustööriista keskmise aja ja `defgrep`-i keskmise aja suhtena; väärtus üle $1\times$ tähendab, et `defgrep` oli kiirem. 37

Joonis 6. Ingliskeelse Vikipeedia pealkirjade andmestiku peal `defgrep`i suhtelised kiirused. 38

Joonis 7. Debiani faililoendi andmestiku peal `defgrep`i suhtelised kiirused. 40

Joonis 8. Common Crawli andmestiku peal `defgrep`i suhtelised kiirused. 41

Tabelite loetelu

Tabel 1. Erinevate pakkimisteekide jõudlus väikese, keskmise ja suure sisendi korral. 18

Tabel 2. Grammatika mõõtmise sisendfailid. 24

Tabel 3. Grammatika implementatsiooni korrektsuse eelkontroll. 25

Tabel 4. Sünteetilise `abc` 10 miljoni korduse olemasolukontroll, aeg sekundites. Sulgudes on aeg võrreldes sama rea kiireima tööriistaga: $1.00x$ tähistab kiireimat ning suurem kordaja aeglasemat käivitust. Kiireim aeg on märgitud paksus kirjas. 25

Tabel 5. Sünteetilise `abc` 1 miljardi korduse olemasolukontroll, aeg sekundites. Sulgudes on aeg võrreldes sama rea kiireima tööriistaga: $1.00x$ tähistab kiireimat ning suurem kordaja aeglasemat käivitust. Kiireim aeg on märgitud paksus kirjas. 26

Tabel 6. NVD CVE kirjete olemasolukontroll, aeg sekundites. Sulgudes on aeg võrreldes sama rea kiireima tööriistaga: 1.00x tähistab kiireimat ning suurem kordaja aeglasemat käivitust. Kiireim aeg on märgitud paksus kirjas. 26

Tabel 7. Mini-gzip ja twins meetodite võrdlus sisendfailidel. 29

Tabel 8. Regex mustrite võrdlus Twins implementatsiooni ja ripgrep vahel. c = vastete koguarv, l = ridade arv. 30

Tabel 9. Puhvri suuruse mõju keskmisele ajale. 32

Tabel 10. Võrdluses kasutatud sisendfailid. 35

Tabel 11. Eestikeelse Vikipeedia pealkirjade vastega ridade loendamine, aeg millisekundites. Sulgudes on tööriista kiirus `defgrep`i suhtes. 50

Tabel 12. Eestikeelse Vikipeedia pealkirjade vastete koguarvu loendamine, aeg millisekundites. Sulgudes on tööriista kiirus `defgrep`i suhtes. 51

Tabel 13. Ingliskeelse Vikipeedia pealkirjade vastega ridade loendamine, aeg sekundites. Sulgudes on tööriista kiirus `defgrep`i suhtes. 52

Tabel 14. Ingliskeelse Vikipeedia pealkirjade vastete koguarvu loendamine, aeg sekundites. Sulgudes on tööriista kiirus defgrepi suhtes. 53
Tabel 15. Debiani faililoendi vastega ridade loendamine, aeg millisekundites. Sulgudes on tööriista kiirus defgrepi suhtes. 54
Tabel 16. Debiani faililoendi vastete koguarvu loendamine, aeg millisekundites. Sulgudes on tööriista kiirus defgrepi suhtes. 55
Tabel 17. Common Crawli WARC-faili vastega ridade loendamine, aeg sekundites. Sulgudes on tööriista kiirus defgrepi suhtes. 56
Tabel 18. Common Crawli WARC-faili vastete koguarvu loendamine, aeg sekundites. Sulgudes on tööriista kiirus defgrepi suhtes. 57

1 Sissejuhatus

Suurandmete ajastul salvestatakse ja edastatakse tohutul hulgal tekstilist informatsiooni kokkupakitud kujul. Lisaks kasutatakse pakkimisalgoritme mitme otstarbega kõikjal internetis. Kasutajad pakivad faile, et säästa potentsiaalselt kuni 80% algsest failisuurusest [1]. Veebiserverid ja HTTP-kliendid kasutavad andmete pakkimist, et vähendada ülekantavate andmete mahtu, mis omakorda parendab andmeedastuskiirust ja lühendab veebilehtede laadimisaega [2]. Samuti on sellel oluline roll arhiveerimises, näiteks mittetulundusühing Common Crawl kasutab kokkupakkimist, et säästa mälu 10 petabaidi suurusel arhiivil [3]. Pakkimine vähendab salvestusruumi ja võrguliikluse kulu, kuid muudab andmete vahetu töötlemise keerulisemaks: otsingumootorid töötavad tavaliselt järjestikuse tekstivoo peal, samas kui pakitud fail sisaldab koode, tagasiviiteid ja vorminguspetsiifilisi metaandmeid.

Käesolev töö keskendub `gzip` pakkimisvormingule ja selle sees kasutatavale `DEFLATE` algoritmile. Töö ei püüa katta kõiki pakkimisvorminguid ega võrrelda kõiki kaasaegseid algoritme. Eksisteerib ka mitmeid alternatiive, nagu `Brotli` ja `zstd`, mis on teatud stsenaariumites kuni 90% efektiivsemad lahtipakkimisel, kui `gzip` [4], kuid peamine motivatsioon `gzip`-i valikuks seisneb selle laialdases kasutuses. W3Techs hinnangul on 2026. aasta 5. aprilli seisuga `gzip` endiselt kõige levinum pakkimisalgoritm veebilehtede seas, seda kasutab 49,0% veebilehtedest, millele järgneb `Brotli` 41,1%-ga [5]. Lisaks leiab `gzip` kasutust erinevates arhiivides, nagu näiteks Wikipedia, pakihaldurites nagu npm või apt, Giti hoidlates ja ka Common Crawl, kus on salvestatud üle 10 petabaiti andmeid.

Pakkimisalgoritmid koosnevad tihti mitmest iseseisvast osast. `DEFLATE` ühendab `LZ77`-põhise korduste asendamise Huffmani kodeeringuga, kus korduvad tekstilõigud esitatakse pikkuse ja tagasiviite kauguse paaridena ning saadud sümbolid kodeeritakse prefikskoodidega. Seetõttu on `DEFLATE`-i sees olemas kordusi kirjeldav struktuur, mida saab otsingu optimeerimisel potentsiaalselt ära kasutada. Samuti katsetatakse ja tehakse analüüs alternatiivlahendusele, mis kasutab `LZ77` omadust teisenduda SLP grammatika reegliteks, mille peal on juba võimalik teha otsingut ilma kogu teksti tavapärasel kujul lahti pakkimata.

Töö keskendub järgmistele uurimisküsimustele.

1. Kuidas saaks `DEFLATE` andmevoost `LZ77` viidete põhjal ehitada SLP-laadse grammatika, millel teostatud otsing annab sisenditel sama tulemuse kui lahtipakitud tekstil tehtud otsing?
2. Kuidas `LZ77` tagasiviiteid kasutada otsinguautomaadi töö vähendamiseks viisil, mis annab praktilise kiirusevõidu?
3. Kui hästi toimib loodud vooglahenduse prototüüp võrreldes võrdlusbaasiga?

2 Analüüs

Selles peatükis kirjeldatakse `DEFLATE` algoritmi, olemasolevaid otsingu tööriistu ning lahtipakkijaid ja regulaaravaldise mootorite valikut. Analüüsi eesmärk on leida praktiline lähtekoht prototüübile: lahtipakkija peab toetama voogtöötlust ning otsingumootor peab sobima järjestikuse tekstivoo töötlemiseks.

Voogtöötlemise vajadus tuleneb praktilistest piirangutest. Ilma voogtöötluseta tuleb kogu fail enne otsingut täielikult lahti pakkida, mis tähendab kas suuremat põhimälu kasutust või ajutise väljundi kirjutamist kettale. Suurte failide korral võib lahtipakitud sisu suurus erineda kokkupakitud sisendist mitmekordselt.

Praeguse töö eesmärk on saavutada kiiruse võit lahtipakkimise ja otsimise peale kokku. Sellepärast on vaja analüüsida mõlemat osa ja teha järeldused, mis võimaldavad meil võimalikult kiiresti tekst lahti pakkida ja sellest otsida.

2.1 Olemasolevad lahendused

Olemasolevad lahendused võimaldavad teostada otsingut otse pakitud andmetel. Eksisteerib lahendusi, mis kasutavad andmete voogtöötlust, kuid ka lahendusi, mis pakivad terve faili lahti. Meie teadmiste järgi pole lahendust, mis kasutaks ära `DEFLATE` algoritmi unikaalseid metaandmeid või üritaks optimeerida just selle peal tehtud otsingut.

2.1.1 ripgrep

`ripgrep` on regexi otsingu tööriist, mis töötab rekursiivselt kataloogide peal. `ripgrep` on tänapäevane lahendus ja see toetab mitmesuguseid otsinguvõimalusi, sealhulgas võimalust otsida kokkupakitud failide peal. `ripgrep`i otsing toetab enamus lahtipakkimisvorminguid, mille hulgas on näiteks `gzip`, `bzip2`, `xz`, `lz4`, `Brotli` ja `zstd` [6].

Otsingu jaoks kasutab `ripgrep` `Rust Regex` mootorit ja literaalide jaoks SIMD algoritmi Teddy. `ripgrep`i põhiline otsingumootor `Rust Regex` ei toeta kõiki PCRE-laadseid

konstruktsioone, nagu näiteks tagasivaateid (*lookbehind*), edasivaateid (*lookahead*) ja tagasiviiteid (*backreferences*), mida kasutatakse keerukamate mustrite kirjeldamiseks. Selliste mustrite jaoks on võimalik kasutada PCRE otsingumootorit `-P`/`--pcre2` võtmega. `ripgrep`i võib pidada käesolevas töös üheks tugevamaiks üldotstarbeliseks pakitud andmetest otsingu tööriistaks [7], mis on enamikes testides võrdlusbaasiks. Tuleb märkida, et `ripgrep`i ja teiste otsingumootorite jõudluse hindamiseks kasutatava võrdlustööriista Rebari autor on sama isik. Sellest hoolimata on Rebar piisavalt tuntud ja erapooletu, et seda enda hinnangu alusena kasutada.

2.1.2 ugrep

`ugrep` on sarnane `ripgrep`ile. `ugrep` toetab samuti kõiki laialt levinud vorminguid nagu `gzip`, `bzip2`, `xz`, `lz4`, `Brotli` ja `zstd`. Peamine erinevus `ripgrep`i ja `ugrep`i vahel tuleb regulaaravaldise raamistikust, kus `ugrep` kasutab `RE-flex`-i, mis on tõhus C++ regulaaravaldiste teek ja lekseri generaator [8]. `ugrep`il on sama probleem, mis `ripgrep`il: kutsutakse välja välist lahtipakkijat nagu `gzip`, mitte parimat implementatsiooni. Kokkupakitud failide peal tehtud otsingu optimeerimisele `ugrep` ei keskendu [9].

2.1.3 zgrep

Kokkupakitud failide peal otsingu jaoks eksisteerib `grep`i versioon nimega `zgrep`. `grep` tugineb automaatidele ja algoritmidele nagu Aho-Corasick ja Boyer–Moore. Lahtipakkimiseks kutsub `zgrep` välja `gzip` lahtipakkija ja söödab lahtipakitud sisu edasi grepile [10].

2.1.4 zcat

`zcat` on lihtne ja optimeerimata lähenemine, millega saab kuvada tervet pakitud faili sisu. `zcat` kasutab olemasolevat lahtipakkijat ja kuvab lahtipakitud sisu käsureale [11]. `zcat` ei paku otsingu võimalust, kuid toome selle siiski välja, kui lahenduse, millega on võimalik kuvada kokkupakitud faili.

2.2 DEFLATE

DEFLATE on kadudeta pakkimisalgoritm, mis koosneb LZ77 algoritmist ja Huffmani kodeeringust [12]. DEFLATE algoritm leiab tänapäeval kasutust mitmetes populaarsetes vormingutes, nagu näiteks gzip, zlib, PNG ja zip. Kuna kokkupakkimisalgoritm on neil vormingutel sama, siis nad erinevadki just metaandmetes, näiteks faili päises ja sabas. Sisu identsuse omaduse tõttu saab selles töös keskenduda ühele implementatsioonile lahtipakkimisest ja lisada vormingutele tugi meie nõuete järgi. Kuigi vorminguid on triviaalne juurde lisada, siis esmajärguline tähelepanu on suunatud just gzip vormingule.

gzip ja DEFLATE ei ole samad mõisted. gzip on failivorming, mis lisab DEFLATE andmevoole päise ja saba, sealhulgas kontrollsumma ja algse faili suuruse. Käesolevas töös loetakse sisendiks gzip faile, kuid otsingu seisukohalt on kõige olulisem nende sees olev DEFLATE andmevoog.

2.2.1 LZ77

LZ77 ehk Lempel-Ziv 1977. aasta algoritm on järjestikune, kadudeta ja sõnastikul põhinev kokkupakkimisalgoritm [13]. LZ77 on suure kokkupakkimisvõimega ning sellest tingitult on selle kasutus väga levinud ka väljaspool DEFLATE algoritmi. Algoritm kasutab tavaolukorras 32KiB suurust tahapoole vaatavat liugakent, et leida korduvaid mustreid, millele viidata. LZ77 lahtipakkimine on palju odavam kui selle kokkupakkimine. Odavam kulu on tingitud sellest, et lahtipakkimiseks peab ainult tekstis tagasi liikuma ja kopeerima. LZ77 viited võivad viidata tekstile, mis ise on samuti tagasiviite poolt loodud. See omapära nõuab, et kogu tekst enne viidet oleks lahtipakitud, et sisu oleks võimalik teada saada.

2.2.2 Huffmani kodeering

Huffmani kodeering on prefikskoodide süsteem, kus sagedamini esinevad sümbolid kodeeritakse lühemate bitijadadega ning harvemini esinevad sümbolid pikematega [14]. Huffmani kodeeringust lahtisaamine on võrreldes LZ77 ja otsinguga triviaalne kulu. See on tingitud sellest, et DEFLATE kasutab kas eelarvutatud tabeleid või dünaamilisi tabeleid [12]. Nendest tabelitest on võimalik minimaalse kuluga dekodeerida LZ77 viited ja literaalide väärtused.

2.3 Lahtipakkijate implementatsioonid

`DEFLATE` algoritm on tänaseks üle 30 aasta vana. Aastate jooksul on loodud mitmeid erinevaid implementatsioone andmete pakkimiseks ja lahtipakkimiseks `DEFLATE` algoritmi abil. Need lahendused erinevad oma eesmärkide poolest – osa keskendub maksimaalsele jõudlusele, ohverdades paindlikkust ja kasutatavust, samas kui teised eelistavad universaalsust ja laialdast ühilduvust.

Lahtipakkija valikul arvestati kolme tingimust: voogtöötluse tugi, jõudlus suurte failide korral ning integreeritavus Rustis kirjutatud otsingumootoriga. Kuna töö eesmärk ei ole kirjutada uut `DEFLATE` dekoodrit nullist, võrreldakse olemasolevaid teeke ning valitakse neist prototüübi jaoks sobivaim. Lisaks voogtöötlust toetavatele teekidele käsitletakse ka lahendusi, mis seda ei toeta, et saada võrdluseks ajaline lähtepunkt.

2.3.1 zlib

`zlib` on üks enimlevinud teeke `DEFLATE` algoritmi realiseerimiseks. Kuigi tegemist ei ole kõige suurema jõudlusega lahendusega, pakub see täielikku ja stabiilset implementatsiooni, mida kasutatakse laialdaselt erinevates süsteemides ja tööriistades. Mitmed utiliidid kasutavad `zlib`i selle laialdase toe ja töökindluse tõttu.

2.3.2 zlib-ng

`zlib-ng` on moderniseeritud alternatiiv `zlib`ile. See säilitab `zlib`iga ühilduva API, kuid on optimeeritud tänapäevaste protsessoriarhitektuuride jaoks. `zlib-ng` kasutab SIMD-operatsioone, optimeeritud `LZ77` tagasiviidete käsitlemist ning vähendab harude arvu kriitilistes koodilõikudes, et saavutada kõrgem läbilaskevõime [15].

2.3.3 zlib-rs

`zlib-rs` on Rusti programmeerimiskeeles kirjutatud zlibi implementatsioon. See saavutab võrreldava jõudluse `zlib-ng`-ga, kasutades kompilaatori automaatset vektorisatsiooni ning optimeerides vahemälu kasutust ja tingimuslausete arvu. Lisaks on `zlib-rs`i lihtne integreerida Rusti-põhistesse projektidesse [16].

2.3.4 libdeflate

`libdeflate` on kõrge tõhususega teek, mida peetakse sageli üheks kiireimaks `DEFLATE` lahtipakkimise implementatsiooniks. Erinevalt `zlib`ist ei toeta see voogtöötlust, vaid on mõeldud terviklike andmeplokkide töötlemiseks.

`libdeflate` saavutab oma suure jõudluse alternatiivse lähenemisega Huffmani kodeeringu dekodeerimisel. See kasutab suuri eelnevalt arvutatud tabeleid, võimaldades dekodeerida korraga mitut sümbolit ning vältides bititasemel iteratiivseid operatsioone. Lisaks vähendab see ettearvamatute harude arvu ning optimeerib mälukasutust, mis võimaldab saavutada väga kõrge läbilaskevõime [17].

2.3.5 zune-inflate

`zune-inflate` on Rustis kirjutatud `DEFLATE` algoritmi implementatsioon, mille eesmärk on saavutada kõrge lahtipakkimise jõudlus. See keskendub eelkõige lahtipakkimise kiirusele ning madalale latentsusele, olles sobilik rakendustes, kus andmeid tuleb töödelda kiiresti ja suurtes mahtudes [18].

Erinevalt traditsioonilistest lahendustest, nagu `zlib`, on `zune-inflate` optimeeritud tänapäevaste protsessorite jaoks. See vähendab tingimuslausete arvu kriitilistes kooditeedes, parandab vahemälu lokaalsust ning kasutab efektiivseid andmestruktuure `DEFLATE` voogude töötlemiseks. Selline lähenemine võimaldab saavutada suuremat läbilaskevõimet võrreldes klassikaliste implementatsioonidega.

2.4 Lahtipakkijate võrdlus

Kuna uue lahtipakkija implementeerimine oleks käesoleva töö raames liiga mahukas, tuleb valida olemasolev implementatsioon, mida võtta katsetuste aluseks. Kuigi `libdeflate` ja `zune-inflate` saavutavad väga suure läbilaskevõime, on need käesoleva töö jaoks ebasobilikud, kuna puudub voogtöötluse tugi ning kogu sisu tuleb enne töötlust lahti pakkida. Sobiva implementatsiooni valimiseks viiakse läbi lihtne võrdlusanalüüs kandidaatlahenduste ning võrdluspunktina ka `libdeflate`-i vahel.

2.4.1 Metoodika

Lahtipakkijate implementatsioonide võrdluseks viiakse läbi kolm erinevat testi. Iga testi aluseks on võetud erineva suurusega failid: Common Crawli sisu ehk suur fail (900MB), JSON fail ehk keskmine fail (100MB) ja man3 fail ehk väike fail (3MB). Tulemuste võrdlemisel mõõdetakse aega, mis kulub kogu faili lahtipakkimiseks. Aja mõõtmiseks kasutatakse programmi nimega `hyperfine` [19], kusjuures tulemuseks võetakse kolme käivituse keskmine. Väikese faili puhul on tulemused esitatud millisekundites, keskmise ja suure fail puhul sekundites.

Katsed viidi läbi Ryzen 5 5600X protsessori peal. Samuti tuleks mainida, et lahtipakkijad ise ei toeta erinevaid päiseid ja variante, mis tähendab, et implementatsioone testiti läbi Rust programmeerimiskeeles kasutades `flate2-rs` teeki. Teekide versioonideks võeti töö kirjutamise hetkel `flate2` versioon 1.1.9, `libdeflate` versioon 1.25.

2.4.2 Tulemused

Tabel 1. Erinevate pakkimisteekide jõudlus väikese, keskmise ja suure sisendi korral.

Meetod	Väike fail (ms)	Keskmine fail (s)	Suur fail (s)
`zlib`	0.562	1.211	7.500
`zlib-rs`	0.443	0.806	4.559
`zlib-ng`	0.534	0.853	4.786
`libdeflate`	0.740	0.412	2.589

Tabelis 1 on näha, et väikese faili korral saavutab parima tulemuse `zlib-rs`, samas kui `libdeflate` jääb teistest lahendustest maha. Tõenäoliselt on see tingitud `libdeflate`-i suuremast käivituslisakulust, mis väikese sisendi korral moodustab märkimisväärse osa kogu töötlusajast.

Keskmise ja suure faili puhul on näha, et `libdeflate` jõuab lahtipakkida märgatavalt kiiremini kui ülejäänud valikud, mis on tingitud madala taseme optimisatsioonidest ja SIMD-operatsioonide kasutamisest.

Samas on `libdeflate` võrreldud lahendustest ainus, mis ei toeta voogtöötlust. See piirang muutub oluliseks olukordades, kus sisendfaili suurus ületab süsteemi põhimälu mahu või

kus andmeid tuleb töödelda järk-järgult.

Kuigi tabeli põhjal saavutab `libdeflate` parima tulemuse kahes katses, ei ole see käesoleva töö jaoks kõige sobivam valik. `libdeflate` on väga kiire suurte tervikfailide lahtipakkimisel, kuid selle API ei sobi voogtöötlusel põhineva lahenduse implementeerimiseks. `zlib-rs` sobib voogtöötluseks paremini, kuna seda saab kasutada Rusti ökosüsteemis ning integreerida otsingumootoriga samasse protsessi.

2.5 Otsingumootor

`DEFLATE` andmevoo peal otsimise teine oluline komponent on otsingumootor. Olemasolevate lahtipakkijate ja otsingumootorite analüüsimisel joonistuvad domineerivate programmeerimiskeeltena välja C++ ja Rust. Mõlema keele jaoks leidub kvaliteetseid `DEFLATE` lahtipakkimise implementatsioone ning nende jõudlus on sageli võrreldav.

Rusti peamine eelis seisneb suure jõudlusega otsingumootorites. Nii `Rust Regex` kui ka `RE#` on kirjutatud Rustis ning kuuluvad tänapäevaste regulaaravaldiste mootorite seas kõrge jõudlusega lahenduste hulka. Käesolevas töös keskendutakse mustrite leidmisele vasakult paremale, mitte kõige pikema vaste leidmisele.

2.5.1 RE#

`RE#` on kõrge jõudlusega automaatidel põhinev regulaaravaldiste otsingumootor. Lisaks toetab `RE#` täiend- ja ühisosaoperaatoreid [20]. Autori analüüsi põhjal võib `RE#` teatud tingimustel olla kuni 71% kiirem kui `Rust Regex` ning saavutada võrreldava jõudluse teiste kaasaegsete otsingumootoritega.

2.5.2 Rust Regex

Kuna käesolev töö on valdavalt kirjutatud Rustis, on loomulikuks valikuks `Rust Regex` otsingumootor. `RE#` uurimustest nähtub, et `Rust Regex` on konkurentsivõimeline teiste kaasaegsete otsingumootoritega ning saavutab mitmes olukorras väga häid tulemusi.

`Rust Regex`i ja selle tõttu ka `ripgrep`i oluline eelis on SIMD-operatsioonidega tehtud literaaliotsing, mis annab lihtsate mustrite korral märkimisväärse jõudluseelise. Kuigi teegil on tugev põhi literaalide otsinguks, ei ole olemas tuge kõikide PCRE-laadsete

konstruktsioonide jaoks, nagu näiteks tagasivaade, mis on keerukam muster.

3 LZ77 teisendamine grammatika kujule

Käesolev peatükk käsitleb grammatikapõhist alternatiivlahendust kokkupakitud tekstist mustri otsimiseks. Lahenduse lähtekohaks on tähelepanek, et `LZ77`-tüüpi pakkimine ei kirjelda teksti üksnes baidijadana, vaid ka korduvate lõikude ja tagasiviidete struktuurina. Seda struktuuri saab omakorda tõlgendada grammatikana: tekst esitatakse reeglite kogumina, mille algussümbol genereerib täpselt algse sõne. Grammatikapõhine otsing kokkupakitud tekstilt on akadeemilises kirjanduses pikalt uuritud suund, millest võetakse järgnevalt aluseks üks hilisem tulemus.

Peatüki teoreetiline alus on Ganardi ja Gawrychowski artikkel *Pattern Matching on Grammar-Compressed Strings in Linear Time*, kus näidatakse, et kui tekst on juba antud SLP-grammatikana, saab mustri olemasolu otsustada ajas $O(n + m)$, kus n on grammatika suurus ja m mustri pikkus [21]. Meie töö kontekstis ei ole sisend aga valmis grammatika, vaid `gzip`i fail. Seetõttu uurime, kas `gzip` sisendist taastatav `LZ77` kirjeldus on praktiliselt teisendatav SLP-laadseks grammatikaks ning kas sellisel kujul otsing võiks olla konkurentsivõimeline tavapärasele lahtipakkimisele ja otsimisele.

3.1 Grammatika kui andmestruktuur

Andmete esitamist reeglite kogumina nimetatakse grammatikapõhiseks pakkimiseks ehk *grammar-based compression* [22]. Sellisel juhul on pakitud faili sisuks reeglid ehk produktsioonid. Reeglites saame defineerida terminaale ja mitteterminaale. Terminaalid on märgid, mis esinevad tekstis (näiteks sümbolid $a, b, 1, *$). Mitteterminaalid on abisümbolid, mis tähistavad mingit alamsõne. Iga mitteterminaali jaoks on antud reegel, mis ütleb, kuidas see mitteterminaal asendub teiste sümbolitega. Nende abil konstrueeritakse samm-sammult tekst.

Reegel on kuju $A \to BC$, kus vasakul on mitteterminaal A ja paremal sümbolite jada BC. Reeglite korduv rakendamine algussümbolile annab lõpuks terminaalidest koosneva sõne.

Näiteks sõne $banaanbanaan$ saab esitada reeglitega $S \to AA$ ja $A \to banaan$. Selles näites on S ja A mitteterminaalid ning tähed b, a ja n terminaalid [23].

3.2 Chomsky normaalkuju

Selleks, et grammatika peal oleks võimalik teostada efektiivset ja kiiret otsingut, nagu kirjeldab artikkel, peavad reeglid alluma kindlale struktuurile, milleks on Chomsky normaalkuju ehk *Chomsky normal form*.

Chomsky normaalkuju lihtsustab grammatikat selliselt, et iga reegel on alati täpselt üks kahest:

- Mitteterminaal jaguneb kaheks: $A \to BC$ (asendub kahe mitteterminaaliga);
- Mitteterminaal jaguneb sümboliks: $A \to a$ (asendub terminaaliga).

Selline piirang tagab, et grammatikast moodustuv puu on alati binaarne, mis teeb selle arvutuslikult efektiivseks [24].

3.3 Straight-Line Program

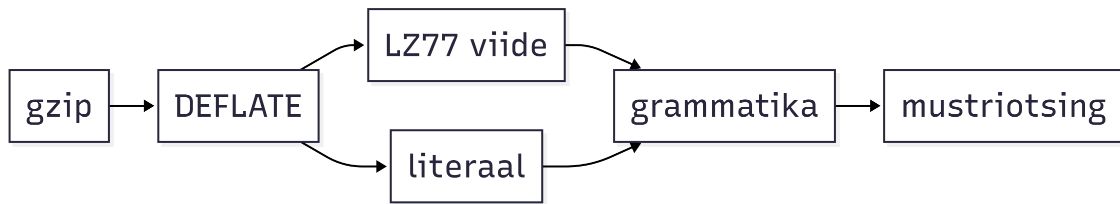
Straight-Line Program ehk SLP on formaalse grammatika erijuht, kus iga mitteterminaal defineeritakse täpselt üks kord ja reeglid on kujul:

- $A \to BC$ (kahe mitteterminaali konkatenatsioon),
- $A \to a$ (terminaal).

Selline puu vorm võimaldab kirjeldada väga pikki sõnesid kompaktselt. Grammatikapuu tasakaal on eelduseks, et oleks võimalik otsing teostada lineaarse kuluga.

3.4 gzip faili teisendus grammatikaks

Artiklis kirjeldatud algoritm eeldab, et sisendtekst on juba antud SLP-grammatikana. Meie töö praktiline küsimus oli seetõttu veidi teistsugune: kas tavalist `gzip` faili on võimalik teisendada sellisesse vormi, et selle peal saaks mustriotsingut teha ilma kogu faili tavapärasel kujul lahti pakkimata. Selleks ehitati prototüübis järgmine töötlusahel, mida on kujutatud joonisel 1.



Joonis 1. Töötlemise voog gzipist mustriotsinguni.

`gzip` faili sees olev `DEFLATE` andmevoog koosneb Huffmani kodeeringust ja `LZ77` viidetest. Dekoodri esimene ülesanne on lugeda bitivoogu, taastada Huffmani tabelid ning teisendada sisend ühtseks `LZ77` tokenite jadaks. Token on prototüübis kujul (val, len). Kui $len = 0$, tähistab val ühte literaalbaidi väärtust. Kui $len > 0$, tähistab val kaugust varem genereeritud tekstis ning len kopeeritava lõigu pikkust.

`LZ77` tokenijada teisendatakse grammatikaks vasakult paremale. Literaaltokeni korral luuakse või taaskasutatakse vastav terminaalireegel. Viitetokeni korral otsitakse juba konstrueeritud grammatika juurest alamsõne, millele viide osutab. See alamstring lisatakse jooksvale juurele binaarse konkatenatsioonina. Kui viide kirjeldab sama lõigu korduvat kopeerimist, kasutatakse korduste konstrueerimisel *repeated squaring* ideed: kõigepealt ehitatakse lõik, siis lõigu kaks koopiat, siis neli koopiat ning edasi järjest kahekordistuva pikkusega kordused. Nii ei ole vaja igat kordust eraldi reeglina lisada.

Grammatikat hoitakse puustruktuuris. Struktuuris hoitakse reegleid ja literaale. Iga mitteterminaal on kas literaal või reegel kujul $A \to BC$. Binaarsete reeglite juures salvestatakse lisaks vasaku ja parema järglase indeksile genereeritava sõne pikkus ning puu kõrgus. Et sama alamstruktuuri mitte korduvalt luua, kasutatakse kanoniseerimist: kui paar (B, C) on juba olemas, saab uut reeglit mitte luua ja kasutada olemasolevat mitteterminaali.

Oluline tehniline probleem on grammatikapuu tasakaal. Kui iga uus token lihtsalt eelmise juure paremale külge lisada, muutuks tuletuspuu sügavaks ning alamsõnede väljavõtmine kulukaks. Seetõttu kasutatakse heuristilist tasakaalustamist: vasaku ja parema alampuu genereeritud pikkusi võrreldakse ning liiga suure ebatasakaalu korral kombineeritakse alampuid ümber. Tegemist ei ole artiklis kirjeldatud täielikult optimaalse andmestruktuuriga, vaid praktilise kompromissiga, mis aitab vältida olukordi, kus tuletuspuu muutub liiga sügavaks.

3.5 Mõõtmismetoodika

Grammatikapõhine prototüüp väljastab iga mustri kohta väärtuse `true` või `false`. Võrdlustööriistade käsud olid `rg -F -z -q` ja `zgrep -a -F -q`, mis kontrollivad literaalmustri olemasolu ning võivad esimese vaste leidmisel töö lõpetada. Kõik mustrid on literaalid, sest prototüüp ei toeta regulaaravaldise süntaksit.

`ripgrep`i puhul tähendab `-F`, et mustrit käsitletakse literaalina, mitte regulaaravaldisena; `-z` lubab otsida pakitud failist; `-q` lülitab sisse vaikse režiimi, kus vasteid ei kirjutata väljundisse. `zgrep`i puhul tähendab `-a`, et lahtipakitud sisu käsitletakse tekstina; `-F` kasutab literaalmustrit; `-q` kasutab vaikset olemasolukontrolli.

Kasutatud sisendfailid on toodud tabelis 2. NVD CVE andmestik [25] on JSON vormingus hiljutiste CVE-de fail, kus faili alguses on metaandmed ning põhiosa moodustab massiiv CVE-kirjetega. Selle jaoks valiti 15 literaalmustrit: neli faili alguses esinevat mustrit, viis keskel esinevat mustrit, viis lõpuosas esinevat mustrit ning üks puuduv kontrollmuster. Lisaks kasutati kahte sünteetilist andmestikku, kus tekst on abc kordus. Nendel testiti ainult mustreid abc ja abcd, kus esimene on väga sage vaste ning teine puudub.

Tabel 2. Grammatika mõõtmise sisendfailid.

Fail	Pakitud suurus	Lahtipakitud suurus	Sisu
`NVD CVE recent`	479,53 KiB	5,07 MiB	NVD CVE JSON 2.0 hiljutised kirjed
`abc10000000.txt.gz`	28,48 KiB	28,61 MiB	`abc` kordus 10 miljonit korda
`abc-1-billion.txt.gz`	2,78 MiB	2,79 GiB	`abc` kordus 1 miljard korda

Enne ajamõõtmist kontrolliti, et grammatikapõhine implementatsioon, `ripgrep` ja `zgrep` annaksid iga mustri korral sama tõeväärtuse. Tabel 3 kujutab olemasolukontrolli tulemusi. Aja mõõtmiseks kasutati `hyperfine` tööriista. Iga käsu kohta tehti kolm mõõdetud jooksu ilma soojenduseta. Tabelites on esitatud aritmeetiline keskmine ning kiireim aeg on märgitud paksus kirjas. Võrdluspunkti põhjaks on võetud grammatika ajalised tulemused.

Tabel 3. Grammatika implementatsiooni korrektsuse eelkontroll.

Andmestik	Teste	Vaste olemas	Vaste puudub	Kattuvus
NVD CVE recent	15	14	1	15/15
abc x 10M	2	1	1	2/2
abc x 1B	2	1	1	2/2

3.6 Tulemused

Tulemused on jagatud andmestike kaupa. CVE andmestiku tabelis on esitatud ka esimese vaste suhteline asukoht lahtipakitud failis. Sünteetilistel andmestikel on vastega mustri esimene vaste faili alguses ning abcd on puuduv kontrollmuster.

CVE andmestikul kulub prototüübil ligikaudu 3,2-3,5 sekundit sõltumata sellest, kas esimene vaste on faili alguses, keskosas, lõpus või puudub täielikult. See on ootuspärane, sest prototüüp ehitab enne otsingut terve `LZ77` põhise grammatika. `ripgrep` ja `zgrep` seevastu töötavad voona ja `-q` režiimis saavad vaste leidmisel lõpetada. Seetõttu kasvab nende aeg faili algusest lõppu liikudes mõnest millisekundist ligikaudu 7-10 millisekundini.

Sünteetilistel andmestikel on vahe väiksem, sest abc kordus on `LZ77` jaoks väga soodne. 30 MB lahtipakitud teksti korral võtab grammatika tee umbes 0,125 sekundit, samal ajal kui `ripgrep` ligikaudu 0,034 sekundit. 3 GB lahtipakitud teksti korral võtab grammatika tee umbes 20 sekundit, `ripgrep` aga umbes 3,0-3,2 sekundit. Seega aitab korduv struktuur grammatika esitust, kuid praegune teisendus ja andmestruktuuride ehitamine jääb siiski kalliks.

Tabel 4. Sünteetilise `abc` 10 miljoni korduse olemasolukontroll, aeg sekundites. Sulgudes on aeg võrreldes sama rea kiireima tööriistaga: $1.00x$ tähistab kiireimat ning suurem kordaja aeglasemat käivitust. Kiireim aeg on märgitud paksus kirjas.

Muster	Stsenaarium	Esimene vaste	Grammatika	`ripgrep`	`zgrep`
`abc`	algus	0.00%	0.1247 (3.72x)	**0.0335 (1.00x)**	0.0390 (1.16x)
`abcd`	vaste puudub	puudub	0.1252 (3.61x)	**0.0347 (1.00x)**	0.0400 (1.15x)

Tabel 5. Sünteetilise `abc` 1 miljardi korduse olemasolukontroll, aeg sekundites. Sulgudes on aeg võrreldes sama rea kiireima tööriistaga: $1.00x$ tähistab kiireimat ning suurem kordaja aeglasemat käivitust. Kiireim aeg on märgitud paksus kirjas.

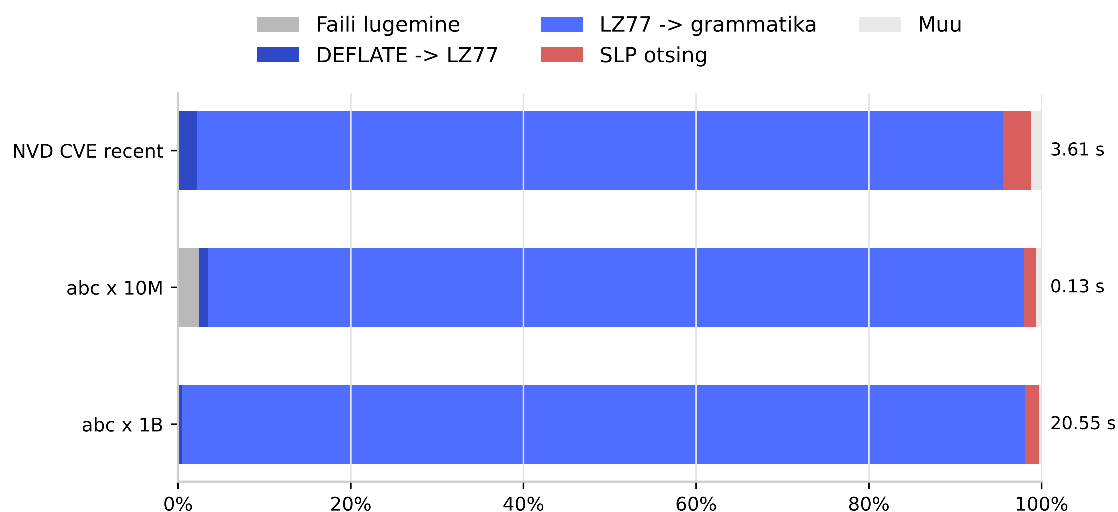
Muster	Stsenaarium	Esimene vaste	Grammatika	`ripgrep`	`zgrep`
`abc`	algus	0.00%	19.8197 (6.58x)	**3.0142 (1.00x)**	4.9804 (1.65x)
`abcd`	vaste puudub	puudub	20.5573 (6.39x)	**3.2170 (1.00x)**	6.9617 (2.16x)

Tabel 6. NVD CVE kirjete olemasolukontroll, aeg sekundites. Sulgudes on aeg võrreldes sama rea kiireima tööriistaga: 1.00x tähistab kiireimat ning suurem kordaja aeglasemat käivitust. Kiireim aeg on märgitud paksus kirjas.

Muster	Stsenaarium	Esimene vaste	Grammatika	`ripgrep`	`zgrep`
`resultsPerPage`	algus	<0.01%	3.4149 (1392.16x)	**0.0025 (1.00x)**	0.0052 (2.12x)
`CVE-2026-3867`	algus	<0.01%	3.2759 (1286.86x)	**0.0025 (1.00x)**	0.0053 (2.10x)
`Secure Router`	algus	0.01%	3.4406 (1242.00x)	**0.0028 (1.00x)**	0.0058 (2.09x)
`buffer overflow`	algus	0.08%	3.2289 (1309.76x)	**0.0025 (1.00x)**	0.0051 (2.06x)
`Cisco`	keskosa	21.60%	3.1815 (916.73x)	**0.0035 (1.00x)**	0.0061 (1.77x)
`Chrome`	keskosa	30.79%	3.1980 (855.27x)	**0.0037 (1.00x)**	0.0068 (1.81x)
`Microsoft`	keskosa	34.62%	3.1811 (805.81x)	**0.0039 (1.00x)**	0.0068 (1.73x)
`CVE-2026-7503`	keskosa	53.58%	3.2109 (705.30x)	**0.0046 (1.00x)**	0.0075 (1.65x)
`CVE-2026-28909`	keskosa	53.78%	3.3210 (704.04x)	**0.0047 (1.00x)**	0.0077 (1.63x)
`OpenSSL`	lõpp	66.20%	3.1847 (569.24x)	**0.0056 (1.00x)**	0.0084 (1.50x)
`CVE-2026-38431`	lõpp	99.83%	3.2924 (442.68x)	**0.0074 (1.00x)**	0.0101 (1.35x)
`CVE-2026-43002`	lõpp	99.86%	3.5487 (502.10x)	**0.0071 (1.00x)**	0.0096 (1.36x)
`CVE-2026-7847`	lõpp	99.89%	3.2114 (457.02x)	**0.0070 (1.00x)**	0.0098 (1.39x)
`Langchain-Chatchat`	lõpp	99.10%	3.2415 (467.80x)	**0.0069 (1.00x)**	0.0096 (1.38x)
`NO_SUCH_CVE_PATTERN_2026`	vaste puudub	puudub	3.4129 (484.43x)	**0.0070 (1.00x)**	0.0099 (1.41x)

3.7 Teisenduskulu

Mõõtmised näitasid, et põhiline kulu ei ole mustri olemasolukontroll ise, vaid `LZ77` tokenite teisendamine grammatikaks. Artikli $O(n + m)$ tulemus kirjeldab otsingut juhul, kui grammatika on juba sisendina olemas. `gzip` faili puhul tuleb aga enne seda teha mitu lisasammu: lugeda fail, eemaldada `gzip`i päis ja saba, dekodeerida `DEFLATE` algoritm, taastada `LZ77` tokenid, ehitada grammatika ning alles seejärel teha otsing. Joonis 2 kujutab otsinguks kulunud ajakulu faaside kaupa.



Joonis 2. Grammatika ajakulu faaside kaupa.

3.8 Järeldused

Grammatika põhine lähenemine kinnitas, et `gzip` failist on võimalik taastada `LZ77` kirjeldus, teisendada see SLP-laadseks grammatikaks ning teha mustriotsingut selle kuju peal. Kontrollitud sisenditel kattusid grammatika põhise otsingu tulemused `ripgrep` tulemustega. Seega on idee funktsionaalselt teostatav.

Samas näitasid mõõtmised, et antud prototüüp ei ole praktiliselt konkurentsivõimeline. Peamine kulu ei ole mustriotsing, vaid `gzip` sisendi teisendamine grammatikaks. Artikli lineaaraja tulemus jääb seetõttu pigem teoreetiliseks sihiks: see kehtib olukorras, kus SLP on juba olemas. `gzip` faili puhul tuleb kogu teisendamisprotsess eraldi arvesse võtta.

Selles töös on grammatika põhine lähenemine eelkõige teoreetilise võimaluse ja tehniliste piirangute kaardistus. See ei ole praegusel kujul kiirem otsingumootoritest nagu `ripgrep` või `zgrep`, kuid suures osas on tegemist arvutuslikult kuluka teisendusega, mitte niivõrd mustriotsingu loogikaga.

4 Vooglahendus

Eelmiste peatükkide põhjal valiti lõplikuks lahenduseks voogtöödeldav arhitektuur, mis ühendab DEFLATE algoritmi ja regulaaravaldise otsingu samas protsessis. Grammatikapõhine prototüüp näitas, et LZ77 kirjeldust saab teisendada SLP-laadseks grammatikaks, kuid selle konstrueerimise kulu on liiga kulukas.

4.1 Twins algoritm

Kõige kulukam ja suurem töö, mis tehakse terve DEFLATE algoritmi peale, on just LZ77 mustrite tekitamine tekstist. Teades, et LZ77 mustrite tekitamiseks on kulunud tohutu arvutusmaht, tekib küsimus, kas saab neid ära kasutada töö vahele jätmiseks või kiirendamiseks.

Twins on kokkupakitud võrguliikluse analüüsi jaoks loodud algoritm, mille autorid väidavad, et on võimalik jätta vahele kuni 90% eksisteerivatest baitidest kokkupakitud tekstiotsingul [26]. Üldine idee on üpris lihtne. Deterministlikus olekumasinas viib korduv sisend samasse olekusse, kui algolek on identne. LZ77 viide on alati tekstilõigule, mis on juba olekumasinast literaalidena läbi käinud. Need kaks asja kokku pannes saame teha väga lihtsa järelduse. Kui olekumasin on viite alguses samas olekus, kui originaaltekstis, lõpetab olekumasin täpselt identses olekus kui originaalis. See väide võimaldab meil jätta vahele nii-öelda olekumasinast läbi söötmist ja tekitab võimaluse ülejäänud kopeerida. Arvestama peab samuti sellega, et viide võib viidata tekstile, mis on olnud viite sees. See tähendab, et me peame hoidma sama suurt puhvrit kui LZ77 distants. Ajaline võit algoritmist tuleb sellest, et olekuid korraga kopeerida on kiirem kui kõik ükshaaval läbi arvutada. Joonis 3 on kujutatud kaks näidet, kus viidet saab vahele jätta, sest olekumasin jääb algolekusse ja näidet, kus olekumasin saab hüppata teatud olekusse. Igat lõiku näites võib võtta kui eraldi LZ77 viitena.

Näide 1	abcd	fad	abcd
Olek:	0000	120	0000

Näide 2	abcdef	bgag	abcdef
Olek:	123456	0010	123456

Joonis 3. Twins algoritmi näide korduva sisendi ja automaadi olekute võrdlemisest.

4.1.1 Lihtsustatud prototüüp

Esialgse katsena implementeeriti Twinsi algoritm lihtsustatud `DEFLATE` algoritmi lahtipakkijal. Sellise lähenemisega oli Twinsi algoritmi lihtsam implementeerida, sest optimeeritud lahtipakkijad nagu `zlib-rs` implementeerivad mitu viisi, kuidas viiteid lahti kirjutada, eraldi abstraktsioonikihi taga ja kasutades SIMD-operatsioone suurte kopeerimiste jaoks.

Tabel 7. Mini-gzip ja twins meetodite võrdlus sisendfailidel.

Sisend	Mini-gzip aeg (s)	Twins aeg (s)	Kiirendus
`enwiki-latest-langlinks.sql.gz`	13.849	13.111	1.06×
`employees_1GB.json.gz`	4.658	3.202	1.45×

Tulemustest on näha, et antud testidel on suurema enwiki faili peal 1,06-kordne kiirendus ning eriti märkimisväärne on employees JSON-fail, kus on 1,45-kordne kiirendus võrreldes lihtsa lahtipakkijaga. Twinsi algoritmi ajaline võit sõltub palju sellest, millist mustrit otsitakse. Kui `LZ77` viite alguses on juba õige olek, siis saab kõige rohkem tööd vahele jätta. Muidu peab viidet läbi töötama kuni olek on klappiv originaalse tekstiga.

4.1.2 Integreerimine optimeeritud lahtipakkijasse

Nähes positiivseid ajalisi võite naiivse lahenduse peal, implementeerisime Twinsi algoritmi `zlib-rs`i lahtipakkija peal. Selle implementatsiooni jaoks peab mitme asjaga arvestama, kui otsing toimub lahtipakkija sees. Twinsi implementeerimiseks peab andmestruktuure ümber kujundama, lisama puhvreid ja säilitama kiirust, mis tuleb optimeeritud harudest ja SIMD-operatsioonidest. `zlib-rs` teegi puhul otsustasime implementeerida Twinsi algoritmi teksti lahtikirjutamise hetkel, koos terve ülejäänud `RE#` otsingumootori haldamisega.

Tabel 8. Regex mustrite võrdlus Twins implementatsiooni ja ripgrep vahel. c = vastete koguarv, l = ridade arv.

Muster	Twins-c (ms)	rg-c (ms)	Twins-l (ms)	rg-l (ms)
`_response`	113	79	137	69
`_WARC-Type:\sresponse`	113	76	135	69
`_Mozilla|_CCBot`	114	92	135	90
`_cloudflare|_nginx|_apache`	113	91	137	83
`_200\s_OK|_404\s_Not\s_Found`	123	95	146	89
`_http://[a-z]+\.com`	112	83	137	69
`(?<=WARC-Type:\s)response`	112	67	135	59
`(?<=HTTP/1\.1\s)[0-9]{3}`	112	92	137	82
`(?<=Server:\s)cloudflare`	113	63	136	58
`_sha1:ABCDEF[0-9]{8}`	111	76	135	69

Tulemustest on näha, et kui implementeerida Twinsi algoritm optimeeritud lahtipakkija sisse, siis varasem hüpotees paika ei pea. Optimeeritud lahtipakkijad eeldavad, et enamik tööst saab teha väikestes tsüklites paralleelsete SIMD-operatsioonide abil. Twinsi algoritmi implementeerides selgus, et otsingult saavutatud kokkuhoid muutis lahtipakkimise enda piisavalt aeglaseks, et kogutulemus oli halvem. Huvitav tulemus on, et iga mõõdetud aeg Twinsi lahendusel saavutab sarnase aja. Sellest võib järeldada, et implementatsioonist tekib uus pudelikael kusagil loendamisel.

Suur ajakulu, mis tuleb arvestada Twinsi algoritmiga on, kui isegi saame kopeerida kõik olekud viite algusest lõpuni, siis selle kopeeritud osa peab ikkagi läbi käima, et teada saada, kus kõik vasted on.

4.2 Lõplik vooglahendus

Twins algoritmi katse näitas, et `LZ77` viidete vahelejätmine võib lihtsustatud lahtipakkijas vähendada otsingumootorile söödetavate baitide arvu. Samas ei kandunud see võit üle optimeeritud lahtipakkijasse. Seetõttu loobuti lõplikus prototüübis `LZ77` viidete vahelejätmisest ning keskenduti väiksema üldkuluga voogtöötlusele: kiire lahtipakkija ja otsingumootor samas protsessis ning vahefaili kirjutamise vältimine.

Lõpplahenduse töövoog on järgmine. Sisendfail loetakse tükkidena, `zlib-rsi` `Inflate`

algoritm dekodeerib `DEFLATE` andmevoo ning toodetud tekstiplokid antakse edasi `RE#` otsingumootorile. Programm ei paki lahti tervet faili mällu, vaid hoiab korraga ainult puhvrit ning otsingumootori olekut.

Lahendus toetab kahte mõõtmisviisi: vastete arvu loendamist ja vastega ridade arvu loendamist. Vastete loendamisel saab otsingumootor töödelda terveid väljundplokke. Ridade loendamisel tuleb säilitada vaste ja reavahetuse järjekord, mistõttu töödeldakse väljundit baitide kaupa. Seetõttu võib sama muster olla `--count-matches` režiimis märgatavalt kiirem kui `--count-lines` režiimis.

Failide sisselugemisel andmete lahtipakkimiseks on kaks võimalust, kas kasutada mälukaardistust või puhverdatud lugejat. Selles töös otsustasime lugeja kasuks järjestikuse töövoo ja suurema maksimaalse kiiruse tõttu. Lahtipakkimisel ja otsingul peab faili lineaarselt algusest lõpuni läbi lugema. Arvestades olemasolevaid tulemusi, saavutab mälukaardistus koos soojendusega läbilaskevõime ligi 2,25 GB/s, võrreldes tavalise lugejaga lineaarsete failide peal, mille läbilaskevõime 2^{16}-baidise puhvri korral on 2,5 GB/s [27]. Arvestades töö eesmärki saavutada maksimaalne kiirus, ei ole praegu meile teadaolevat põhjust kasutada mälukaardistust üle tavalise puhverdatud lugeja.

Vooglahendust ehitades otsustasime integreerida `zlib-rs`i teegi otse meie koodibaasi [16]. Selline lähenemine võimaldas meile kahte asja. Esiteks oli võimalik teha katsetusi Twinsi algoritmi implementeerimisega, mida analüüsisime alguses. Teine ja tähtsam saavutus oli see, et sisseehitatud lahtipakkija võimaldas vähendada protsessi alustamise kulu, tänu millele oli väikeste failide peal teatud olukordades võimalik saavutada ajaline võit. Protsessi alustamise kulu vähendamine tuli sellest, et olemasolevad tööriistad kutsuvad välja väliseid lahtipakkimistööriistu.

4.2.1 Puhvri suuruse mõju

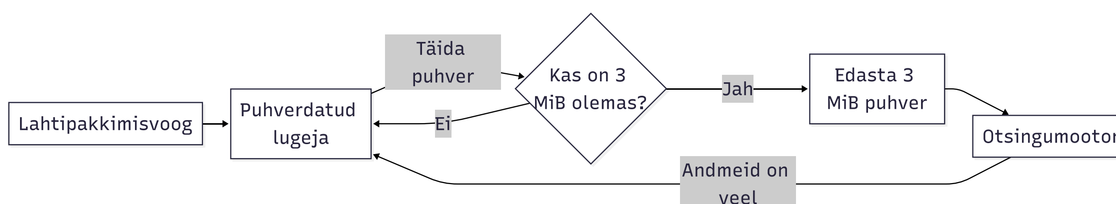
Kuna otsustasime teha voogtöötluse lahenduse, siis tuleb arvestada ka puhvri suurusega. Puhvri suurusest on tingitud kas on kasutuses vahemälu või muutmälu. Katsetamiseks võeti fail, mis oli 100MB kokkupakituna ja 1GB lahtisena. Testina mõõdetakse lahtipakkija puhvri suuruse mõju tulemusele. Kasutatud muster testis oli `spring|java|boot`.

Tabel 9. Puhvri suuruse mõju keskmisele ajale.

Puhvri suurus	Keskmine aeg (ms)	Puhvri suurus	Keskmine aeg (ms)
512B	1713.5	256KiB	1200.6
1KiB	1542.4	512KiB	1193.7
2KiB	1445.6	1MiB	1193.7
4KiB	1405.3	2MiB	1187.9
8KiB	1391.2	4MiB	1182.4
16KiB	1370.0	8MiB	1183.3
32KiB	1311.4	16MiB	1178.2
64KiB	1250.8	**32MiB**	**1177.2**
128KiB	1212.7	64MiB	1182.3

Tabelist 9 järeldub, et liiga väike väljundpuhver suurendab halduskulu, kuid pärast 1 MiB piiri muutub mõju väikeseks. Seetõttu valiti lõplikus prototüübis vaikimisi 3 MiB suurune lahtipakitud väljundpuhver. See hoiab plokkide arvu väiksemana, kuid ei suurenda mälu kasutust määravalt.

Lõplik prototüüp `defgrep` loeb sisendfaili 1 MiB suuruste tihendatud plokkidena. `zlib-rs` põhine `Inflate` dekooder toodab vaikimisi kuni 3 MiB suuruseid lahtipakitud plokke, mis antakse kohe `RE#` voogotsijale. Programm ei kirjuta lahtipakitud faili kettale ega hoia kogu teksti mälus. Mälus hoitakse sisendpuhvrit, väljundpuhvrit, lahtipakkija olekut ja otsingumootori olekut. Töö voogu on näha joonisel 4.



Joonis 4. Voogtöötluse lahendus.

Vastete koguarvu loendamine on prototüübis odavam, sest iga leitud vaste suurendab ainult loendurit. Vastega ridade loendamisel tuleb lisaks teada, kas samal real on juba vaste loendatud. Selleks tuleb võrrelda vastepositsioone reavahetuste positsioonidega

ning säilitada ridade olekut üle plokipiiride. See lisatöö selgitab, miks sama muster võib loendamise režiimis olla kiirem kui ridade loendamisel.

5 Tulemuste analüüs

Prototüübi võrdlemiseks kasutasime `ripgrep`i ja `zgrep`i. `ripgrep` esindab tänapäevast optimeeritud otsingutööriista, mis oskab `gzip` failidest otsida. `zgrep` esindab klassikalist lahendust, kus `gzip` pakitakse lahti andmevoona ja antakse edasi grepile.

`ripgrep` käsud moodustati kujul `rg -z`, literaalide korral lisati `-F` ja PCRE-laadsete mustrite korral `-P`, mis kasutab PCRE otsingumootorit. `zgrep` jaoks lisati literaalide korral `-F` ja muude mustrite korral `-P`. `ugrep` käsud moodustati kujul `ug -z`, literaalide korral lisati `-F` ja PCRE-laadsete mustrite korral `-P`, mis kasutab PCRE otsingumootorit. Kõigi kolme puhul `-F` teostab literaalide otsingut ja ei otsi regulaaravaldist.

Mõõtmisel kasutasime kahte väljundirežiimi: vastega ridade arvu ja vastete koguarvu. Neid režiime tuleb eraldi käsitleda, sest need ei pruugi olla otsingu mõttes sarnaselt optimeeritud. Näiteks reavahetuste otsimist saab teha efektiivsemalt SIMD-operatsioonidega. defgrep puhul vastavad neile käsurea argumendid `--count-lines` ja `--count-matches`. `ripgrep` puhul kasutati vastavalt `-c` ja `-count-matches`. `zgrep` puhul kasutati ridade loendamiseks `-c`, vastete loendamiseks kasutati kuju `zgrep -a -P -o <muster> <fail> | wc -l`.

Aja mõõtmiseks kasutasime käsureatööriista `hyperfine`. Iga käsu kohta tehti üks soojenduskatse ja kolm mõõdetud katset. Tabelites on esitatud aritmeetiline keskmine. Kiireim aeg on märgitud paksus kirjas. Enne mõõtmist kontrollisime, et kõigi tabelites olevate mustrite korral annaksid `defgrep`, `ripgrep`, `ugrep` ja `zgrep` sama loenduse. Lisaks on `RE#` süntaksis alakriips _ erimärk, mistõttu tuli alakriipsuga literaalid `defgrep` käsus põgeneda kujul `_`; tabelites on mustrid loetavuse huvides esitatud tavapärasel kujul. Paremal on tabelites märgitud, milline tööriist on kiirem, juhul kui `defgrep` on aeglasem.

PCRE tulemuste võrdluses tuleb arvestada, et antud otsingumootor teeb `ripgrep` ja `zgrep` otsingud aeglasemaks, sest see on iseseisev ja teisiti optimeeritud otsingumootor võrreldes `Rust Regex`-i ja `RE-flex`-ga.

Mõõtmised tehti arvutil, mille protsessor on AMD Ryzen 5 5600X ning millel on 16 GB mälu. Vastavate programmide versioonid olid `hyperfine 1.20.0`, `ripgrep 15.1.0` PCRE2 toega ning `gzip/zgrep 1.14`.

Iga andmestiku jaoks valiti vähemalt 15 mustrit. Mustrikomplekt sisaldab literaale, alternatsioone, tühja tulemusega kontrollmustreid, suurema vastete arvuga mustreid ning *lookaround* mustreid. Detailsed tulemused iga andmestiku ja tööriista jaoks on toodud välja eraldi tabelites Lisa 2 all. Tabelis 10 on näha erinevate failide suurusi ja sisude kirjeldusi.

Tabel 10. Võrdluses kasutatud sisendfailid.

Fail	Pakitud suurus	Lahtipakitud suurus	Sisu
`etwiki all-titles`	3,51 MiB	11,46 MiB	eestikeelse Vikipeedia pealkirjad
`enwiki all-titles`	361,34 MiB	1,45 GiB	ingliskeelse Vikipeedia pealkirjad
`Contents-amd64.gz`	12,03 MiB	161,06 MiB	Debiani pakettide failiteed
`CC-MAIN-2026-08 WARC`	846,02 MiB	3,74 GiB	Common Crawli WARC-kirjed

5.1 Eestikeelse Vikipeedia pealkirjad

Eestikeelse Vikipeedia fail [28] on mõõdetud andmestikest väikseim, umbes 3,5 MiB kokkupakitult ja 11,5 MiB lahtipakitult. Iga rida sisaldab nimeruumi ja lehe pealkirja, mistõttu sobib see eelkõige literaalide, alternatsioonide ja lihtsate sõnakujude testimiseks.

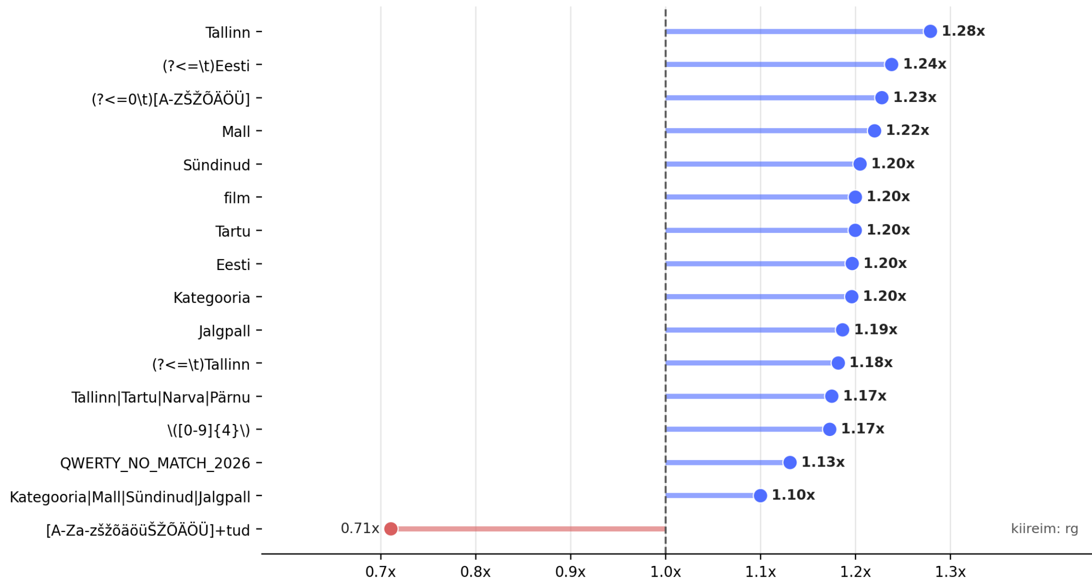
Jooniselt 5 tuleb välja, et `defgrep` on kiireim 15 mustri puhul 16-st nii ridade kui ka vastete loendamisel. Keskmine kiirustegur parima tööriista suhtes oli ridade loendamisel 1,16x ja vastete loendamisel 1,43x.

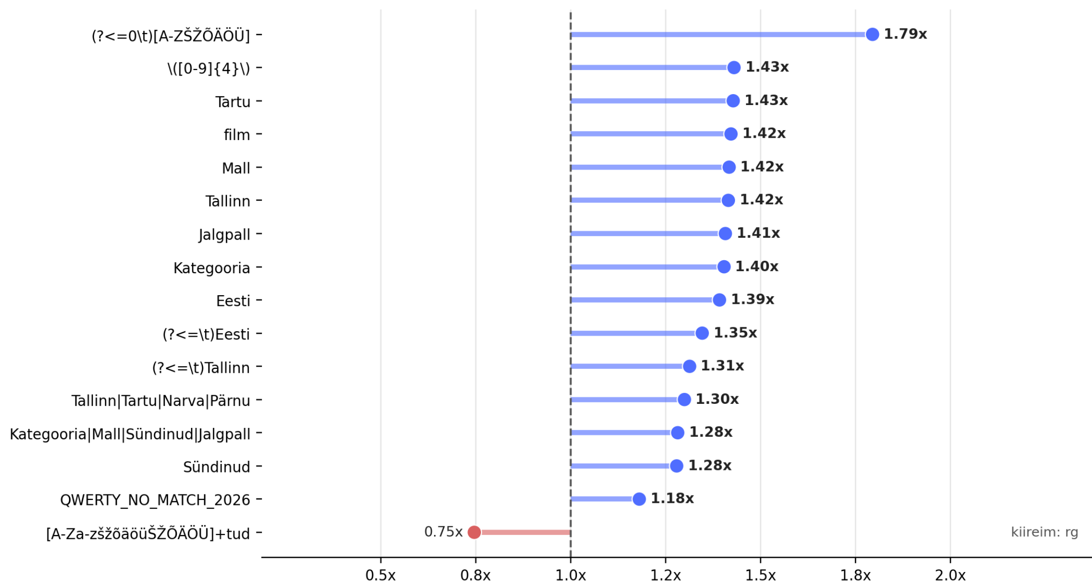
Selles suurusjärgus on absoluutajad kümnetes millisekundites, mistõttu mõjutab tulemust lisaks otsingule ka protsessi käivitamise lisakulu. `defgrep` kasutab sisseehitatud lahtipakkijat, kuid `ripgrep` tugineb välisele lahtipakkimisprogrammile, näiteks `gzip`ile. See tähendab, et väikese faili korral võib `ripgrep`i mõõdetud aega mõjutada ka eraldi

lahtipakkimisprotsessi käivitamine ja selle väljundi edasiandmine otsingumootorile.

Vastete loendamisel on defgrepi eelis suurem kui ridade loendamisel. Vaste loendamine on lihtne loenduri suurendamine, kuid rea loendamiseks tuleb lisaks salvestada vastepositsioone ja skaneerida reavahetusi. Kuna etwiki failis on palju lühikesi ridu, on ridade loendamise lisatöö suhteliselt hästi nähtav.

Erandiks oli muster `[A-Za-zžõäöüŠŽÕÄÖ]+tud`, kus defgrep oli aeglasem kui teised otsingutööriistad.



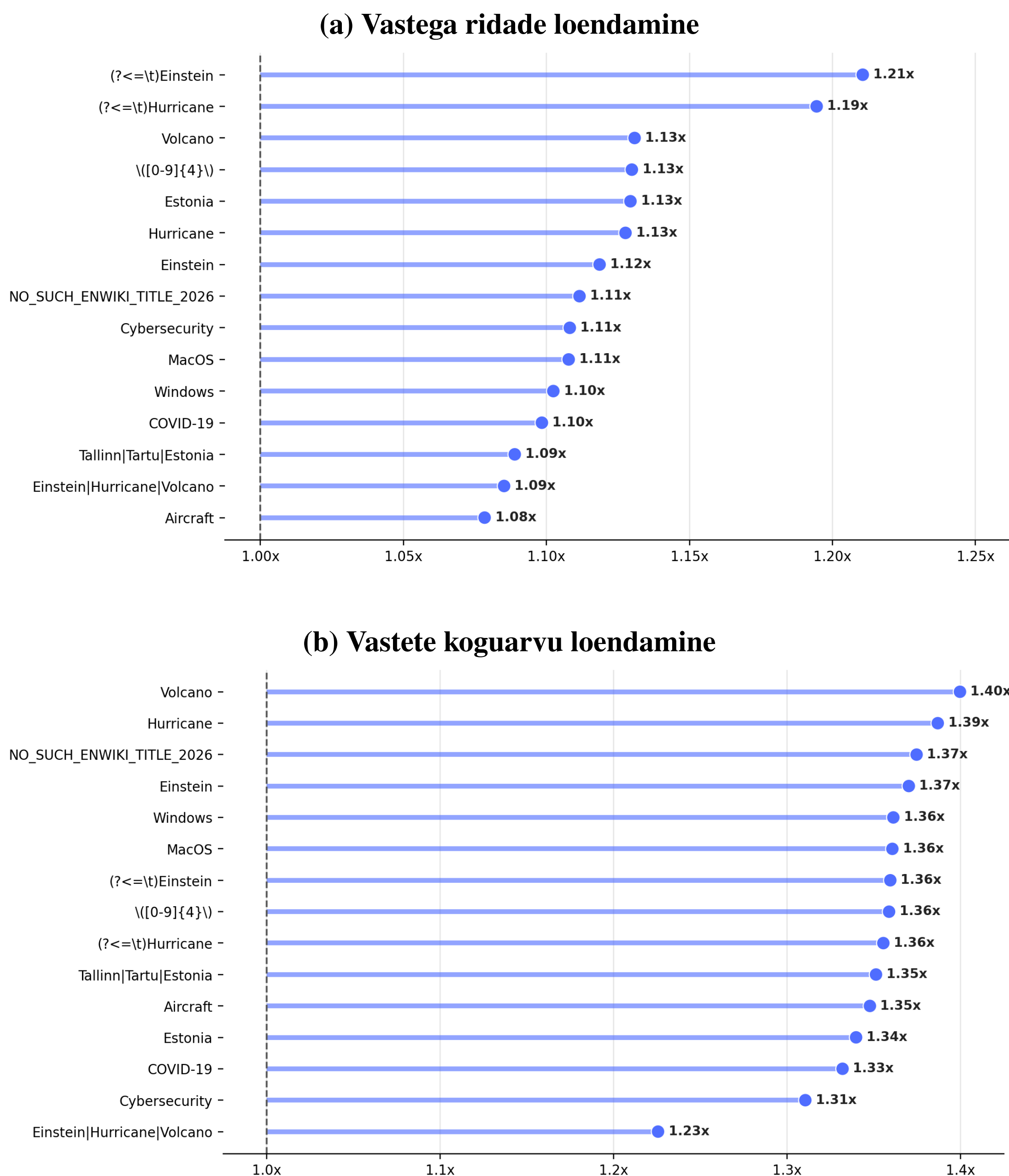


Joonis 5. Eestikeelse Vikipeedia pealkirjade andmestiku peal `defgrep`i suhtelised kiirused. Kiirus on arvutatud kiireima võrdlustööriista keskmise aja ja `defgrep`-i keskmise aja suhtena; väärtus üle $1\times$ tähendab, et `defgrep` oli kiirem.

5.2 Ingliskeelse Vikipeedia pealkirjad

Ingliskeelse Vikipeedia pealkirjade [29] fail on sama tüüpi andmestik, kuid oluliselt suurem: 361 MiB kokkupakitult ja 1,45 GiB lahtipakitult. Võrreldes väiksema failiga mõõdab see paremini voogtöötluse kiirust, sest protsessi alustamise aeg on kogutöö suhtes väiksem. Mustriteks valiti keskmise sagedusega literaalid ja üks alternatsioon.

Joonisel 6 on `defgrep` kõigi 15 mustri ja mõlema väljundirežiimi puhul kiireim. Ridade loendamisel kõige suurem võit tuli tagasivaate mustritel, kus oli võit kuni 21%. Peale selle oli saavutatud võit literaalide otsingul vahemikus 8–13%. Tulemusteta mustri peal oli võit 11%. Koguarvu loendamisel oli kõige suurem võit 40%, mis saavutati literaalide loendamisel. Samuti sama palju kiirem oli vasteteta literaali otsimine. Vastega ridade loendamisel on eelis `ripgrep` ja `zgrep` ees kokku 8–21%. Vastete koguarvu loendamisel kasvab vahe 23–40%-ni.



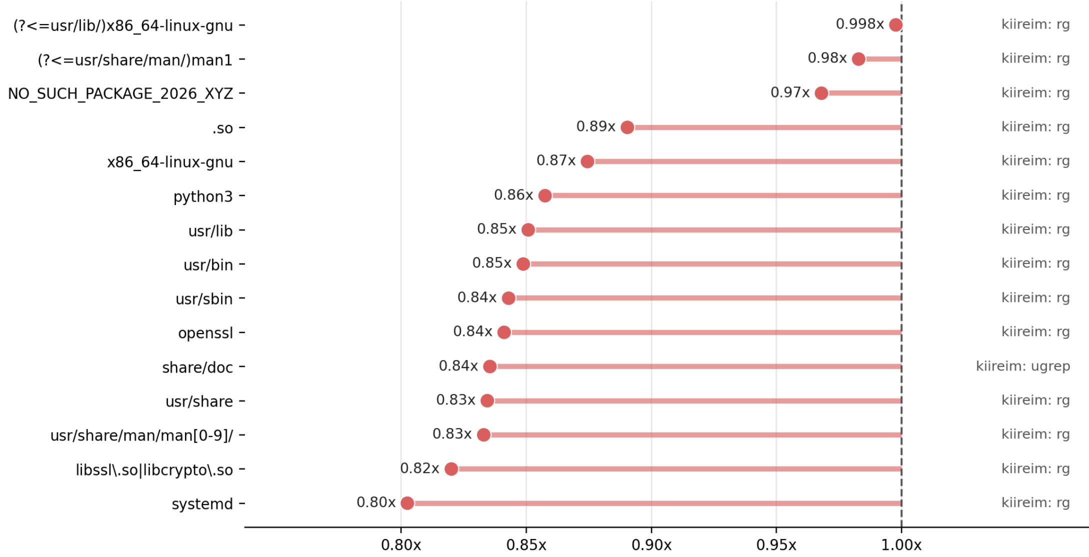
Joonis 6. Ingliskeelse Vikipeedia pealkirjade andmestiku peal `defgrep`i suhtelised kiirused.

5.3 Debiani faililoend

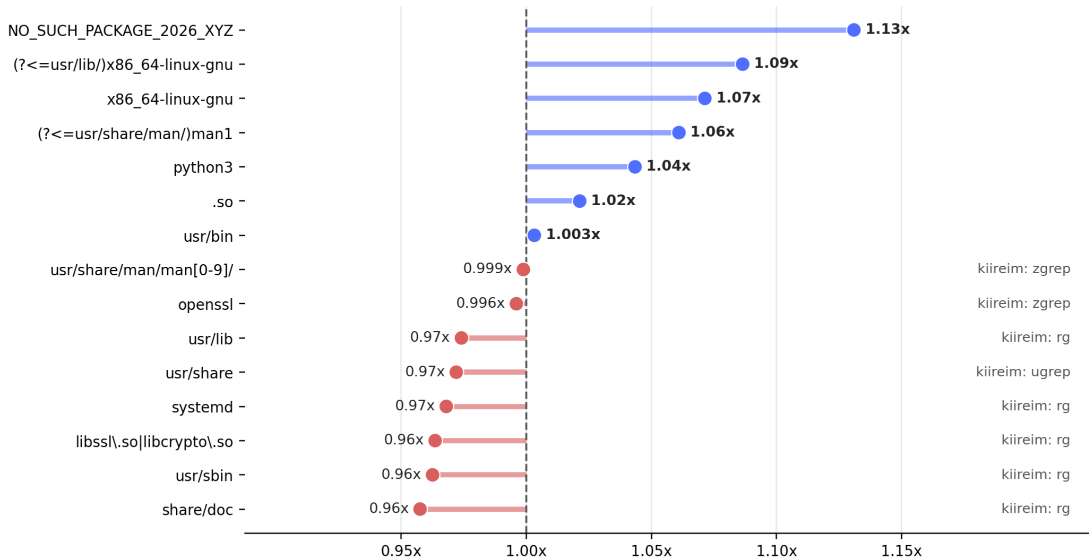
Debiani faililoend [30] on indeks, mis näitab, millise Debiani paketi sees iga failitee asub. Igal real on kõigepealt failitee ja seejärel paketi kategooria koos paketinimega. Mustriteks valiti sagedased teeprefiksid, paketisõned, manuaalilehtede teed, arhitektuuri teeosa, harv teeginimede alternatsioon ning kaks tagasivaate mustrit.

Joonisel 7 on tulemus kõige segasem. Vastega ridade loendamisel on `ripgrep` või `zgrep` kõigi mustrite puhul kiirem; `defgrep` konkureerib ainult arhitektuuri tagasivaate ja tühja mustri peal. Vastete koguarvu loendamisel muutub pilt paremaks: `defgrep` on kiireim 7 mustri puhul 15-st ning mitme ülejäänud mustri puhul on erinevus väike. Koguarvu loendamisel on literaalide tulemus kõigil üpris sarnane. Märkimisväärne võit tuleb ainult tagasivaatel ja tühjal mustril.

(a) Vastega ridade loendamine



(b) Vastete koguarvu loendamine



Joonis 7. Debiani faililoendi andmestiku peal `defgrep`i suhtelised kiirused.

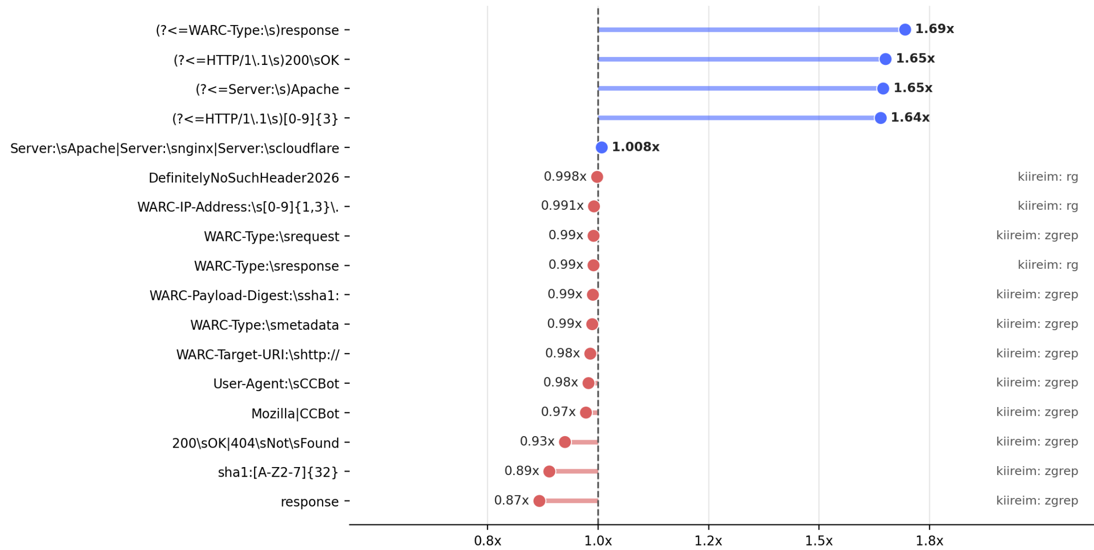
5.4 Common Crawli WARC andmestik

Common Crawli andmestik [31] on mõõdetud failidest suurim: 3,74 GiB lahtipakitult. Fail koosneb WARC-kirjetest ja HTTP-sõnumitest, mistõttu sisaldab see palju korduvaid päiseid, näiteks `WARC-Type`, `WARC-Payload-Digest`, lisaks HTTP päiseid.

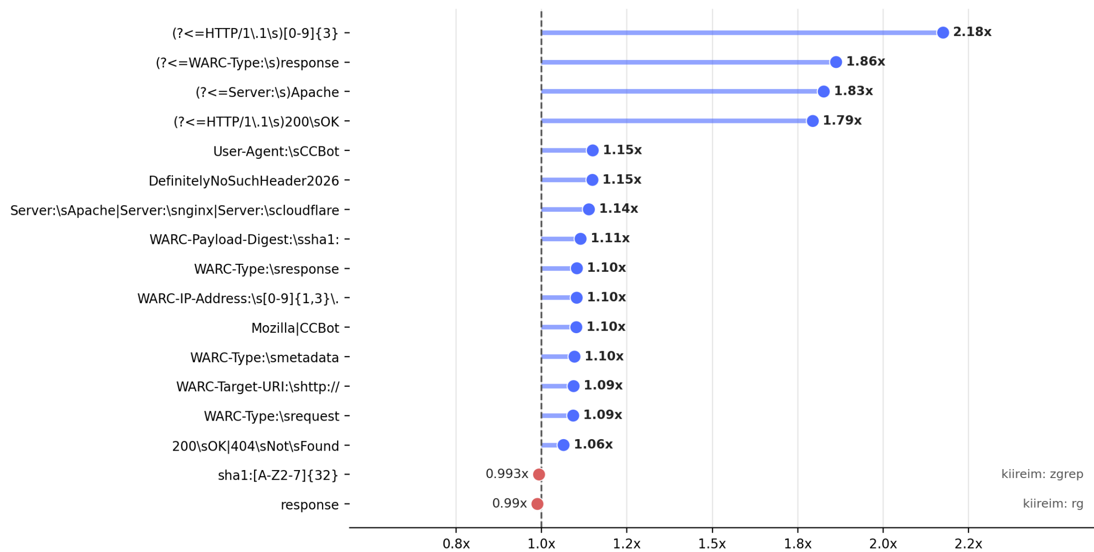
Joonisel 8 eristuvad lihtsad päisemustrid ja tagasivaate mustrid. Lihtsate WARC-päiste ridade loendamisel on `zgrep` või `ripgrep` sageli napilt kiirem. Vastete koguarvu loendamisel on `defgrep` kiireim 15 mustri puhul 17-st. Kõige suurem eelis ilmneb tagasivaate

mustritel: `HTTP 200`, HTTP staatusekood, `Server: Apache` ja `WARC-Type` tagasivaate mustrid on oluliselt kiiremad kui `rg -P` ja `zgrep -P`.

(a) Vastega ridade loendamine



(b) Vastete koguarvu loendamine



Joonis 8. Common Crawli andmestiku peal `defgrep`i suhtelised kiirused.

5.5 Tulemuste kokkuvõte

Mõõtmistest saab mitu üldistust välja lugeda. Kõige suurem võit on üldjoontes mustritel, milles sisaldub tagasivaade. Nende puhul on võit tingitud `RE#` otsingumootorist. Samuti on näha, et defgrep suudab saavutada parema aja kui `ripgrep` ja `zgrep` enamus vastete loendamiste testidel. Ainus muster, kus prototüüp märgatavalt alla jäi, oli `[A-Za-zšžõäöüŠŽÕÄÖÜ]+tud`, kus saavutati ainult 0.75x kiirus, võrreldes konkurentsiga.

Literaalide otsingul oli enamus testidel ligi 1.10x kiiruse võit, selle põhjuseks oleks nii otsingumootor kui ka lahtipakkija. Koht, kus oleks vaja veel arendust teha, on vastega ridade leidmine: selles on prototüüp kiirem tagasivaate mustritega, kuid kui vaadata Debiani faililoendit, langevad kõik tulemused samasse auku ning ridade loendamisel on see isegi viimasel kohal.

6 Kokkuvõte

Töö eesmärk oli uurida, kas on võimalik DEFLATE vormingus kokkupakitud failidel otsida kiiremini. Selle eesmärgi saavutamiseks uuriti kolme erinevat lähenemist: LZ77 viidetest SLP-grammatika ehitamist, otsinguautomaadi töö vahelejätmist Twins algoritmi abil ning otsingumootori tihedamat integratsiooni lahtipakkijaga.

Grammatikapõhine lähenemine näitas, et kokkupakitud faili LZ77 viidetest saab ehitada SLP-laadse esituse ning kontrollitud sisenditel säilis otsingu korrektsus. Samas osutus grammatika konstrueerimise kulu liiga suureks, et antud lähenemine oleks otstarbekas.

Twinsi algoritmist inspireeritud lähenemine andis lihtsustatud lahtipakkijas mõõdetava minimaalse kiirusevõidu, kuid optimeeritud zlib-rs lahtipakkijasse integreerimisel koos RE# otsingumootoriga võit kadus: korrektsuse hoidmiseks vajalik lisainfo, automaadi olekute säilitamine ja lisakontrollid suurendasid töövoo kulu rohkem, kui LZ77 viidete vahelejätmine seda vähendas.

Lõplik voogtöötluse prototüüp, kus pakitakse lahti, kuni puhver on täis, ja söödetakse RE# otsingumootorile, on lootustandev. Katsete põhjal oli näha, et keerukamate mustrite puhul nagu tagasivaated suudab RE#-il töötav prototüüp saavutada üle 2-kordse kiirusevõidu. Samuti on prototüüp lihtsate mustrite peal, kus on vaja loendada kõik vasted, üldjuhul kiirem. Nõrgaks kohaks jäid ridade loendamine ja Debiani faililoend, kus prototüüp oli võrdväärse või kehvema tulemusega.

Tulemused toetavad järeldust, et gzip-/DEFLATE-failide otsingut saab kiirendada, kui kiire lahtipakkimine ühendada tõhusa voogtöödeldava regulaaravaldise mootoriga. Samas ei tõestanud tulemused universaalset kiirusevõitu kõikides stsenaariumites.

Vastused uurimisküsimustele

1. Kuidas saaks DEFLATE andmevoost LZ77 viidete põhjal ehitada SLP-laadse grammatika, millel teostatud otsing annab sisenditel sama tulemuse kui lahtipakitud tekstil tehtud otsing?

Funktsionaalselt on see võimalik. Töö raames ehitatud prototüüp teisendab DEFLATE-voost taastatud LZ77 tokenid SLP-laadseks grammatikaks ning sellel teostatud fikseeritud mustri olemasolukontroll langeb kontrollitud sisenditel kokku tööriista `ripgrep` tulemusega. Korrektsus on kinnitatud reaalsete andmestike ja sünteetiliste sisendite peal, kuid täielikku formaalset tõestust ega kombineeritud piirjuhtude (LZ77 viite, DEFLATE-ploki ja grammatikareegli piiri ületavate vastete) ammendavat testimist töös ei tehtud.

2. Kuidas LZ77 tagasiviiteid kasutada otsinguautomaadi töö vähendamiseks viisil, mis annab praktilise kiirusevõidu?

Kasutades Twinsi algoritmi on lihtsustatud lahtipakkijas võimalik tööd vähendada, optimeeritud lahtipakkijas ei ole. Twinsi algoritmi naiivses implementatsioonis saavutati enwiki failil 1,06-kordne ja employees JSON-failil 1,45-kordne kiirendus. Optimeeritud `zlib-rs` lahtipakkijasse integreerimisel see võit kadus, sest olekute säilitamise ja lisakontrollide kulu ületas säästetud töö mahu. Lisaks tuleb vastete asukohtade leidmiseks viite kopeeritud osa siiski läbi käia. Järeldus on, et SIMD-ga optimeeritud lahtipakkijate kontekstis ei anna LZ77 viidete vahelejätmine praktilist kiirusevõitu.

3. Kui hästi toimib loodud vooglahenduse prototüüp võrreldes võrdlusbaasiga?

Prototüüp `defgrep` on kiirem järgmistes olukordades:

- Tagasivaate mustrid kõigil andmestikel – kiirusevõit on kuni 2-kordne, mis tuleneb RE# otsingumootorist.
- Ingliskeelse Vikipeedia pealkirjade fail – kõigi 15 mustri ja mõlema väljundirežiimi puhul kiireim (10–40% eelis).
- Eestikeelse Vikipeedia pealkirjad – kiireim 15/16 mustri puhul, keskmine kiirustegur 1,16–1,43x.
- Common Crawli WARC-andmestik vastete koguarvu loendamisel – kiireim 15/17 mustri puhul.

Prototüüp **ei ole kiirem** Debiani faililoendi ridade loendamisel (välja arvatud arhitektuuri tagasivaate mustril) ning mustri `[A-Za-zšžõäöüŠŽÕÄÖÜ]+tud` peal etwiki andmestikul. Universaalset kiirusevõitu ei saavutatud.

Kasutatud kirjandus

[1] Jean-Loup Gailly ja Mark Adler. *zlib Technical Details*. zlib. 2022. URL: https://zlib.net/zlib_tech.html (vaadatud 21.04.2026).

[2] MDN. *Compression in HTTP*. MDN Blog RSS. 4. juuli 2025. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Compression (vaadatud 24.04.2026).

[3] Common Crawl. *About*. Common Crawl - Open Repository of Web Crawl Data. 2026. URL: https://commoncrawl.org/about (vaadatud 25.04.2026).

[4] Matt Mehra *et al. New standards for a faster and more private internet*. The Cloudflare Blog. 25. september 2024. URL: https://blog.cloudflare.com/new-standards/ (vaadatud 22.04.2026).

[5] W3Techs. *Usage statistics of compression for websites*. W3Techs. 5. aprill 2026. URL: https://w3techs.com/technologies/details/ce-compression (vaadatud 05.04.2026).

[6] Andrew Gallant. *GitHub - BurntSushi/ripgrep: ripgrep recursively searches directories for a regex pattern while respecting your gitignore — github.com*. https://github.com/burntsushi/ripgrep. (Vaadatud 31.03.2026).

[7] Andrew Gallant. *Ripgrep is faster than grep, AG, Git Grep, UCG, PT, sift*. 2016. URL: https://burntsushi.net/ripgrep//#searching (vaadatud 31.03.2026).

[8] Genivia. *Genivia/RE-flex: Yet another high-performance C++ Regex Library and Lexical Analyzer Generator like flex*. 2019. URL: https://github.com/Genivia/RE-flex (vaadatud 18.05.2026).

[9] Genivia. *Genivia/ugrep: ugrep 7.8 file pattern searcher*. GitHub. URL: https://github.com/Genivia/ugrep (vaadatud 30.04.2026).

[10] zgrep. 2026. URL: https://linux.die.net/man/1/zgrep (vaadatud 31.03.2026).

[11] zcat. 2026. URL: https://linux.die.net/man/1/zcat (vaadatud 31.03.2026).

[12] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. Mai 1996. DOI: 10.17487/RFC1951. URL: https://www.rfc-editor.org/info/rfc1951 (vaadatud 01.03.2026).

[13] J. Ziv ja A. Lempel. „A universal algorithm for sequential data compression“. *IEEE Transactions on Information Theory* 23.3 (1977), lk. 337–343. DOI: 10.1109/TIT.1977.1055714. (Vaadatud 25.03.2026).

[14] David A. Huffman. „A Method for the Construction of Minimum-Redundancy Codes“. *Proceedings of the IRE* 40.9 (1952), lk. 1098–1101. DOI: 10.1109/JRPROC.1952.273898. (Vaadatud 25.03.2026).

[15] Hans Kristian Rosbach. *Zlib-ng/zlib-ng: Zlib replacement with optimizations for "Next Generation" systems*. GitHub. URL: `https://github.com/zlib-ng/zlib-ng` (vaadatud 25.04.2026).

[16] Trifecta Tech Foundation. *Trifectatechfoundation/Zlib-Rs: A zlib implementation in Rust available as a C dynamic library and as a Rust crate*. GitHub. URL: `https://github.com/trifectatechfoundation/zlib-rs` (vaadatud 25.04.2026).

[17] Eric Biggers. *Ebiggers/Libdeflate: Heavily optimized library for Deflate/zlib/gzip compression and decompression*. GitHub. 2026. URL: `https://github.com/ebiggers/libdeflate` (vaadatud 06.05.2026).

[18] Caleb Etemesi. *ETEMESI254/Zune-Image: A fast and memory efficient image library in Rust*. GitHub. 2026. URL: `https://github.com/etemesi254/zune-image` (vaadatud 06.05.2026).

[19] Sharkdp. *SHARKDP/Hyperfine: A command-line benchmarking tool*. GitHub. 2018. URL: `https://github.com/sharkdp/hyperfine` (vaadatud 19.04.2026).

[20] Ian Erik Varatalu, Margus Veanes ja Juhan Ernits. „RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds". *Proc. ACM Program. Lang.* 9.POPL (jaanuar 2025). DOI: `10.1145/3704837`. URL: `https://doi.org/10.1145/3704837` (vaadatud 22.04.2026).

[21] Moses Ganardi ja Paweł Gawrychowski. „Pattern Matching on Grammar-Compressed Strings in Linear Time". Teoses: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial ja Applied Mathematics, jaanuar 2022, lk. 2833–2846. ISBN: 9781611977073. DOI: `10.1137/1.9781611977073.110`. URL: `http://dx.doi.org/10.1137/1.9781611977073.110` (vaadatud 01.03.2026).

[22] Sebastian Maneth. „Grammar-Based Compression". Teoses: *Encyclopedia of Big Data Technologies*. Toim. Sherif Sakr ja Albert Y. Zomaya. Cham: Springer International Publishing, 2019, lk. 801–808. ISBN: 978-3-319-77525-8. DOI: `10.1007/978-3-319-77525-8_56`. URL: `https://doi.org/10.1007/978-3-319-77525-8_56` (vaadatud 24.04.2026).

[23] Jaak Henno. *Grammatikad*. 2013. URL: `https://staff.ttu.ee/~jaak.henno/transl/grammatikad.htm` (vaadatud 25.04.2026).

[24] Michael Sipser. „Ch 2: Context-Free Languages". Teoses: *Introduction to the Theory of Computation*. 3. väljaanne. Cengage Learning, 2012, lk. 101–110. (Vaadatud 24.04.2026).

[25] National Institute of Standards and Technology. *NVD CVE 2.0 recent JSON data feed*. URL: `https://nvd.nist.gov/feeds/json/cve/2.0/nvdcve-2.0-recent.json.gz` (vaadatud 29.04.2026).

[26] Xiuwen Sun *et al.* „Efficient regular expression matching over compressed traffic". *Computer Networks* 168 (2020), lk. 106996. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2019.106996`. URL: `https://www.sciencedirect.com/science/article/pii/S1389128618311939` (vaadatud 19.04.2026).

[27] Gagarine Yaikhom. *Comparing throughput of memory mapped file to buffered read*. 2024. URL: https://yaikhom.com/2024-05-03-comparing-memory-mapped-file-to-buffered-read.html (vaadatud 06.05.2026).

[28] Wikimedia Foundation. *Estonian Wikipedia all-titles dump*. 1. aprill 2026. URL: https://dumps.wikimedia.org/etwiki/20260401/etwiki-20260401-all-titles.gz (vaadatud 25.04.2026).

[29] Wikimedia Foundation. *English Wikipedia all-titles dump*. 1. aprill 2026. URL: https://dumps.wikimedia.org/enwiki/20260401/enwiki-20260401-all-titles.gz (vaadatud 03.05.2026).

[30] Debian Project. *Debian stable main Contents-amd64 package path index*. URL: https://ftp.debian.org/debian/dists/stable/main/Contents-amd64.gz (vaadatud 03.05.2026).

[31] Common Crawl. *CC-MAIN-2026-08 WARC segment file*. 6. veebruar 2026. URL: http://data.commoncrawl.org/crawl-data/CC-MAIN-2026-08/segments/1770395505396.36/warc/CC-MAIN-20260206181458-20260206211458-00000.warc.gz (vaadatud 03.05.2026).

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Meie, Andre Ivan ja Marcus-Kevin Otto

1. Anname Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Tekstiotsingu kiirendamine DEFLATE-algoritmiga pakitud failides", mille juhendajad on Ian Erik Varatalu ja Juhan-Peep Ernits
   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. Oleme teadlikud, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autoritele.
3. Kinnitame, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

01.06.2026

[1]Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Tulemuste tabelid

Tabel 11. Eestikeelse Vikipeedia pealkirjade vastega ridade loendamine, aeg millisekundites. Sulgudes on tööriista kiirus `defgrepi` suhtes.

<table>
<tr><th>Muster</th><th>Tulemus</th><th>ripgrep (ms)</th><th>zgrep (ms)</th><th>ugrep (ms)</th><th>defgrep (ms)</th></tr>
<tr><td>Eesti</td><td>13 001</td><td>35,6 (0.84x)</td><td>39,9 (0.74x)</td><td>38,4 (0.77x)</td><td>**29,7 (1.00x)**</td></tr>
<tr><td>Tallinn</td><td>4 195</td><td>38,6 (0.78x)</td><td>39,8 (0.76x)</td><td>38,7 (0.78x)</td><td>**30,2 (1.00x)**</td></tr>
<tr><td>Tartu</td><td>3 548</td><td>35,8 (0.83x)</td><td>38,8 (0.77x)</td><td>37,5 (0.80x)</td><td>**29,8 (1.00x)**</td></tr>
<tr><td>film</td><td>1 877</td><td>34,7 (0.83x)</td><td>37,7 (0.77x)</td><td>37,8 (0.77x)</td><td>**28,9 (1.00x)**</td></tr>
<tr><td>Kategooria</td><td>82</td><td>35,8 (0.84x)</td><td>40,3 (0.74x)</td><td>37,2 (0.80x)</td><td>**29,9 (1.00x)**</td></tr>
<tr><td>Mall</td><td>325</td><td>35,8 (0.82x)</td><td>40,2 (0.73x)</td><td>38,1 (0.77x)</td><td>**29,4 (1.00x)**</td></tr>
<tr><td>Sündinud</td><td>2 514</td><td>36,8 (0.83x)</td><td>41,2 (0.74x)</td><td>38,3 (0.80x)</td><td>**30,6 (1.00x)**</td></tr>
<tr><td>Jalgpall</td><td>420</td><td>35,9 (0.84x)</td><td>38,6 (0.78x)</td><td>38,5 (0.79x)</td><td>**30,2 (1.00x)**</td></tr>
<tr><td>Tallinn|Tartu|Narva|Pärnu</td><td>9 901</td><td>36,0 (0.85x)</td><td>41,7 (0.74x)</td><td>36,9 (0.83x)</td><td>**30,6 (1.00x)**</td></tr>
<tr><td>Kategooria|Mall|Sündinud|Jalgpall</td><td>3 336</td><td>36,0 (0.91x)</td><td>38,8 (0.85x)</td><td>38,4 (0.85x)</td><td>**32,8 (1.00x)**</td></tr>
<tr><td>\([0-9]{4}\)</td><td>3 746</td><td>37,3 (0.85x)</td><td>39,4 (0.80x)</td><td>37,1 (0.85x)</td><td>**31,6 (1.00x)**</td></tr>
<tr><td>[A-Za-zšžõäöüŠŽÕÄÖÜ]+tud</td><td>3 052</td><td>**35,8 (1.41x)**</td><td>38,7 (1.30x)</td><td>38,2 (1.32x)</td><td>50,4 (1.00x)</td></tr>
<tr><td>(?<=\t)Eesti</td><td>9 744</td><td>37,2 (0.81x)</td><td>39,4 (0.76x)</td><td>37,2 (0.81x)</td><td>**30,0 (1.00x)**</td></tr>
<tr><td>(?<=\t)Tallinn</td><td>2 930</td><td>36,5 (0.85x)</td><td>39,7 (0.78x)</td><td>38,1 (0.81x)</td><td>**30,9 (1.00x)**</td></tr>
<tr><td>(?<=0\t)[A-ZŠŽÕÄÖÜ]</td><td>438 265</td><td>36,6 (0.81x)</td><td>38,8 (0.77x)</td><td>39,0 (0.76x)</td><td>**29,8 (1.00x)**</td></tr>
<tr><td>QWERTY_NO_MATCH_2026</td><td>0</td><td>36,0 (0.88x)</td><td>41,4 (0.77x)</td><td>37,4 (0.85x)</td><td>**31,8 (1.00x)**</td></tr>
</table>

Tabel 12. Eestikeelse Vikipeedia pealkirjade vastete koguarvu loendamine, aeg millisekundites. Sulgudes on tööriista kiirus `defgrep`i suhtes.

<table>
<tr><th>Muster</th><th>Tulemus</th><th><code>ripgrep</code> (ms)</th><th><code>zgrep</code> (ms)</th><th><code>ugrep</code> (ms)</th><th><code>defgrep</code> (ms)</th></tr>
<tr><td><code>Eesti</code></td><td>13 040</td><td>35,8 (0.72x)</td><td>38,7 (0.66x)</td><td>39,0 (0.66x)</td><td>25,7 (1.00x)</td></tr>
<tr><td><code>Tallinn</code></td><td>4 199</td><td>36,2 (0.71x)</td><td>40,4 (0.63x)</td><td>36,9 (0.69x)</td><td>25,6 (1.00x)</td></tr>
<tr><td><code>Tartu</code></td><td>3 630</td><td>36,4 (0.70x)</td><td>39,4 (0.65x)</td><td>38,3 (0.67x)</td><td>25,5 (1.00x)</td></tr>
<tr><td><code>film</code></td><td>1 881</td><td>35,7 (0.70x)</td><td>39,3 (0.64x)</td><td>37,3 (0.67x)</td><td>25,1 (1.00x)</td></tr>
<tr><td><code>Kategooria</code></td><td>82</td><td>35,6 (0.71x)</td><td>39,6 (0.64x)</td><td>37,4 (0.68x)</td><td>25,4 (1.00x)</td></tr>
<tr><td><code>Mall</code></td><td>325</td><td>36,7 (0.71x)</td><td>39,2 (0.66x)</td><td>36,9 (0.70x)</td><td>25,9 (1.00x)</td></tr>
<tr><td><code>Sündinud</code></td><td>2 514</td><td>36,7 (0.78x)</td><td>40,7 (0.70x)</td><td>37,9 (0.76x)</td><td>28,7 (1.00x)</td></tr>
<tr><td><code>Jalgpall</code></td><td>424</td><td>35,7 (0.71x)</td><td>39,1 (0.65x)</td><td>38,1 (0.67x)</td><td>25,3 (1.00x)</td></tr>
<tr><td><code>Tallinn|Tartu|Narva|Pärnu</code></td><td>10 126</td><td>35,6 (0.77x)</td><td>41,6 (0.66x)</td><td>37,8 (0.73x)</td><td>27,4 (1.00x)</td></tr>
<tr><td><code>Kategooria|Mall|Sündinud|</code>
<code>Jalgpall</code></td><td>3 345</td><td>36,3 (0.78x)</td><td>40,1 (0.71x)</td><td>37,1 (0.76x)</td><td>28,3 (1.00x)</td></tr>
<tr><td><code>\([0-9]{4}\)</code></td><td>3 748</td><td>37,3 (0.70x)</td><td>40,2 (0.65x)</td><td>37,1 (0.70x)</td><td>25,9 (1.00x)</td></tr>
<tr><td><code>[A-Za-zšžõäöüŠŽÕÄÖÜ]+tud</code></td><td>3 064</td><td>35,7 (1.34x)</td><td>41,4 (1.15x)</td><td>37,4 (1.28x)</td><td>47,8 (1.00x)</td></tr>
<tr><td><code>(?<=\t)Eesti</code></td><td>9 744</td><td>36,4 (0.74x)</td><td>40,5 (0.67x)</td><td>38,2 (0.71x)</td><td>27,0 (1.00x)</td></tr>
<tr><td><code>(?<=\t)Tallinn</code></td><td>2 930</td><td>38,3 (0.74x)</td><td>40,0 (0.71x)</td><td>37,3 (0.76x)</td><td>28,4 (1.00x)</td></tr>
<tr><td><code>(?<=0\t)[A-ZŠŽÕÄÖÜ]</code></td><td>438 265</td><td>73,2 (0.37x)</td><td>75,8 (0.36x)</td><td>49,1 (0.56x)</td><td>27,4 (1.00x)</td></tr>
<tr><td><code>QWERTY_NO_MATCH_2026</code></td><td>0</td><td>37,1 (0.85x)</td><td>40,0 (0.79x)</td><td>39,0 (0.81x)</td><td>31,4 (1.00x)</td></tr>
</table>

Tabel 13. Ingliskeelse Vikipeedia pealkirjade vastega ridade loendamine, aeg sekundites. Sulgudes on tööriista kiirus defgrepi suhtes.

Muster	Tulemus	ripgrep (s)	zgrep (s)	ugrep (s)	defgrep (s)
Cybersecurity	311	3,351 (0.90x)	3,508 (0.86x)	3,531 (0.86x)	**3,024 (1.00x)**
MacOS	415	3,317 (0.90x)	3,344 (0.90x)	3,516 (0.85x)	**2,995 (1.00x)**
Einstein	2 405	3,355 (0.89x)	3,374 (0.89x)	3,526 (0.85x)	**2,999 (1.00x)**
Volcano	3 326	3,400 (0.88x)	3,408 (0.88x)	3,486 (0.86x)	**3,007 (1.00x)**
Windows	8 173	3,365 (0.91x)	3,410 (0.90x)	3,496 (0.87x)	**3,053 (1.00x)**
Aircraft	9 069	3,358 (0.92x)	3,344 (0.93x)	3,489 (0.89x)	**3,101 (1.00x)**
COVID-19	10 329	3,472 (0.88x)	3,350 (0.91x)	3,526 (0.87x)	**3,050 (1.00x)**
Hurricane	14 373	3,395 (0.88x)	3,375 (0.89x)	3,509 (0.85x)	**2,993 (1.00x)**
Estonia	18 144	3,403 (0.88x)	3,395 (0.89x)	3,509 (0.86x)	**3,006 (1.00x)**
Tallinn|Tartu|Estonia	20 892	3,431 (0.90x)	3,361 (0.92x)	3,496 (0.88x)	**3,086 (1.00x)**
Einstein|Hurricane|Volcano	20 104	3,350 (0.92x)	3,403 (0.91x)	3,527 (0.88x)	**3,088 (1.00x)**
\([0-9]{4}\)	123 285	3,411 (0.88x)	3,392 (0.89x)	3,501 (0.86x)	**3,002 (1.00x)**
(?<=\t)Einstein	1 233	3,435 (0.82x)	3,420 (0.83x)	3,507 (0.81x)	**2,825 (1.00x)**
(?<=\t)Hurricane	7 063	3,517 (0.82x)	3,431 (0.84x)	3,500 (0.82x)	**2,872 (1.00x)**
NO_SUCH_ENWIKI_TITLE_2026	0	3,351 (0.90x)	3,341 (0.90x)	3,526 (0.85x)	**3,006 (1.00x)**

Tabel 14. Ingliskeelse Vikipeedia pealkirjade vastete koguarvu loendamine, aeg sekundites. Sulgudes on tööriista kiirus defgrepi suhtes.

Muster	Tulemus	ripgrep (s)	zgrep (s)	ugrep (s)	defgrep (s)
Cybersecurity	311	3,308 (0.76x)	3,339 (0.76x)	3,506 (0.72x)	**2,525 (1.00x)**
MacOS	415	3,345 (0.73x)	3,360 (0.73x)	3,541 (0.69x)	**2,458 (1.00x)**
Einstein	2 413	3,380 (0.72x)	3,337 (0.73x)	3,521 (0.69x)	**2,436 (1.00x)**
Volcano	3 332	3,399 (0.71x)	3,385 (0.71x)	3,484 (0.69x)	**2,419 (1.00x)**
Windows	8 209	3,368 (0.73x)	3,396 (0.73x)	3,496 (0.71x)	**2,475 (1.00x)**
Aircraft	9 070	3,397 (0.74x)	3,438 (0.73x)	3,506 (0.72x)	**2,520 (1.00x)**
COVID-19	10 352	3,331 (0.75x)	3,344 (0.75x)	3,543 (0.71x)	**2,501 (1.00x)**
Hurricane	14 469	3,330 (0.72x)	3,324 (0.72x)	3,525 (0.68x)	**2,397 (1.00x)**
Estonia	18 153	3,384 (0.74x)	3,349 (0.75x)	3,494 (0.72x)	**2,500 (1.00x)**
Tallinn|Tartu|Estonia	20 957	3,392 (0.74x)	3,401 (0.74x)	3,500 (0.72x)	**2,510 (1.00x)**
Einstein|Hurricane|Volcano	20 214	3,314 (0.82x)	3,473 (0.78x)	3,514 (0.77x)	**2,704 (1.00x)**
\([0-9]{4}\)	123 321	3,318 (0.74x)	3,518 (0.69x)	3,522 (0.69x)	**2,442 (1.00x)**
(?<=\t)Einstein	1 233	3,606 (0.71x)	3,491 (0.74x)	3,509 (0.73x)	**2,568 (1.00x)**
(?<=\t)Hurricane	7 063	3,496 (0.73x)	3,456 (0.74x)	3,490 (0.73x)	**2,550 (1.00x)**
NO_SUCH_ENWIKI_TITLE_2026	0	3,347 (0.73x)	3,394 (0.72x)	3,517 (0.69x)	**2,435 (1.00x)**

Tabel 15. Debiani faililoendi vastega ridade loendamine, aeg millisekundites. Sulgudes on tööriista kiirus defgrepi suhtes.

Muster	Tulemus	ripgrep (ms)	zgrep (ms)	ugrep (ms)	defgrep (ms)
libssl\.so|libcrypto\.so	8	**215,0 (1.22x)**	220,3 (1.19x)	225,4 (1.16x)	262,2 (1.00x)
openssl	1 837	**214,8 (1.19x)**	219,1 (1.17x)	220,3 (1.16x)	255,4 (1.00x)
usr/sbin	3 192	**214,0 (1.19x)**	220,3 (1.15x)	221,8 (1.15x)	253,9 (1.00x)
systemd	6 324	**216,3 (1.25x)**	221,7 (1.22x)	223,1 (1.21x)	269,6 (1.00x)
usr/bin	33 034	**218,7 (1.18x)**	220,4 (1.17x)	224,3 (1.15x)	257,7 (1.00x)
(?<=usr/share/man/)man1	29 249	**210,7 (1.02x)**	216,4 (0.991x)	237,3 (0.90x)	214,4 (1.00x)
usr/share/man/man[0-9]/	65 214	**213,6 (1.20x)**	219,3 (1.17x)	219,8 (1.17x)	256,4 (1.00x)
.so	64 645	**218,5 (1.12x)**	226,4 (1.08x)	222,8 (1.10x)	245,4 (1.00x)
python3	77 452	**216,1 (1.17x)**	219,2 (1.15x)	219,4 (1.15x)	252,1 (1.00x)
(?<=usr/lib/)x86_-64-linux-gnu	229 197	**214,0 (1.002x)**	218,2 (0.98x)	225,5 (0.95x)	214,5 (1.00x)
share/doc	261 328	223,3 (1.19x)	222,8 (1.19x)	**222,1 (1.20x)**	265,9 (1.00x)
x86_64-linux-gnu	282 049	**211,4 (1.14x)**	218,5 (1.11x)	231,6 (1.04x)	241,7 (1.00x)
usr/share	713 107	**216,2 (1.20x)**	220,2 (1.18x)	222,2 (1.17x)	259,1 (1.00x)
usr/lib	789 164	**217,2 (1.18x)**	219,6 (1.16x)	221,5 (1.15x)	255,3 (1.00x)
NO_SUCH_PACKAGE_2026_XYZ	0	**217,5 (1.03x)**	220,7 (1.02x)	218,5 (1.03x)	224,7 (1.00x)

Tabel 16. Debiani faililoendi vastete koguarvu loendamine, aeg millisekundites. Sulgudes on tööriista kiirus defgrepi suhtes.

Muster	Tulemus	ripgrep (ms)	zgrep (ms)	ugrep (ms)	defgrep (ms)
libssl\.so|libcrypto\.so	8	**220,7 (1.04x)**	224,6 (1.02x)	223,0 (1.03x)	229,1 (1.00x)
openssl	2 542	222,2 (0.98x)	**217,5 (1.004x)**	222,9 (0.98x)	218,3 (1.00x)
usr/sbin	3 192	**212,7 (1.04x)**	222,5 (0.993x)	220,4 (1.002x)	221,0 (1.00x)
systemd	9 827	**214,7 (1.03x)**	218,3 (1.02x)	222,8 (0.996x)	221,8 (1.00x)
usr/bin	33 034	214,8 (0.997x)	219,6 (0.98x)	220,6 (0.97x)	**214,1 (1.00x)**
(?<=usr/share/man/)man1	29 249	218,3 (0.94x)	219,8 (0.94x)	238,5 (0.86x)	**205,8 (1.00x)**
usr/share/man/man[0-9]/	65 214	221,9 (0.999x)	**221,3 (1.001x)**	222,3 (0.997x)	221,6 (1.00x)
.so	64 647	215,4 (0.98x)	219,4 (0.96x)	223,3 (0.94x)	**210,9 (1.00x)**
python3	144 256	217,4 (0.96x)	223,4 (0.93x)	224,9 (0.93x)	**208,3 (1.00x)**
(?<=usr/lib/)x86_-64-linux-gnu	229 197	262,7 (0.79x)	229,6 (0.90x)	225,1 (0.92x)	**207,1 (1.00x)**
share/doc	261 328	**219,8 (1.04x)**	224,2 (1.02x)	224,2 (1.02x)	229,6 (1.00x)
x86_64-linux-gnu	282 677	215,4 (0.93x)	222,4 (0.90x)	221,2 (0.91x)	**201,0 (1.00x)**
usr/share	713 117	221,9 (1.03x)	230,6 (0.99x)	**221,7 (1.03x)**	228,1 (1.00x)
usr/lib	789 185	**215,7 (1.03x)**	225,4 (0.98x)	223,3 (0.992x)	221,4 (1.00x)
NO_SUCH_PACKAGE_2026_XYZ	0	221,3 (0.88x)	225,7 (0.87x)	222,9 (0.88x)	**195,7 (1.00x)**

Tabel 17. Common Crawli WARC-faili vastega ridade loendamine, aeg sekundites. Sulgudes on tööriista kiirus `defgrep`i suhtes.

<table>
<tr><th>Muster</th><th>Tulemus</th><th><code>ripgrep</code> (s)</th><th><code>zgrep</code> (s)</th><th><code>ugrep</code> (s)</th><th><code>defgrep</code> (s)</th></tr>
<tr><td><code>DefinitelyNoSuchHeader2026</code></td><td>0</td><td>6,370 (1.002x)</td><td>6,389 (0.999x)</td><td>7,160 (0.89x)</td><td>6,383 (1.00x)</td></tr>
<tr><td><code>(?<=HTTP/1\.1\s)200\sOK</code></td><td>5 073</td><td>6,458 (0.60x)</td><td>6,357 (0.61x)</td><td>7,817 (0.49x)</td><td>3,851 (1.00x)</td></tr>
<tr><td><code>(?<=Server:\s)Apache</code></td><td>2 147</td><td>6,448 (0.59x)</td><td>6,300 (0.61x)</td><td>7,361 (0.52x)</td><td>3,830 (1.00x)</td></tr>
<tr><td><code>(?<=HTTP/1\.1\s)[0-9]{3}</code></td><td>22 033</td><td>7,877 (0.49x)</td><td>6,364 (0.61x)</td><td>11,259 (0.34x)</td><td>3,881 (1.00x)</td></tr>
<tr><td><code>(?<=WARC-Type:\s)response</code></td><td>22 031</td><td>6,489 (0.58x)</td><td>6,376 (0.59x)</td><td>8,926 (0.42x)</td><td>3,762 (1.00x)</td></tr>
<tr><td><code>200\sOK|404\sNot\sFound</code></td><td>5 181</td><td>6,354 (1.07x)</td><td>6,270 (1.08x)</td><td>7,307 (0.93x)</td><td>6,777 (1.00x)</td></tr>
<tr><td><code>WARC-Target-URI:\shttp://</code></td><td>5 706</td><td>6,324 (1.01x)</td><td>6,304 (1.02x)</td><td>7,322 (0.88x)</td><td>6,416 (1.00x)</td></tr>
<tr><td><code>User-Agent:\sCCBot</code></td><td>22 034</td><td>6,320 (1.02x)</td><td>6,298 (1.02x)</td><td>7,154 (0.90x)</td><td>6,437 (1.00x)</td></tr>
<tr><td><code>WARC-Type:\sresponse</code></td><td>22 031</td><td>6,321 (1.01x)</td><td>6,332 (1.009x)</td><td>7,176 (0.89x)</td><td>6,387 (1.00x)</td></tr>
<tr><td><code>WARC-Type:\srequest</code></td><td>22 031</td><td>6,401 (0.995x)</td><td>6,305 (1.01x)</td><td>7,178 (0.89x)</td><td>6,370 (1.00x)</td></tr>
<tr><td><code>WARC-Type:\smetadata</code></td><td>22 031</td><td>6,340 (1.010x)</td><td>6,316 (1.01x)</td><td>7,164 (0.89x)</td><td>6,400 (1.00x)</td></tr>
<tr><td><code>WARC-Payload-Digest:\ssha1:</code></td><td>22 031</td><td>6,430 (1.003x)</td><td>6,374 (1.01x)</td><td>7,155 (0.90x)</td><td>6,449 (1.00x)</td></tr>
<tr><td><code>Mozilla|CCBot</code></td><td>22 876</td><td>6,635 (1.02x)</td><td>6,576 (1.03x)</td><td>7,250 (0.93x)</td><td>6,761 (1.00x)</td></tr>
<tr><td><code>Server:\sApache|Server:\snginx|Server:\scloudflare</code></td><td>3 817</td><td>6,559 (0.98x)</td><td>6,477 (0.992x)</td><td>7,142 (0.90x)</td><td>6,423 (1.00x)</td></tr>
<tr><td><code>WARC-IP-Address:\s[0-9]{1,3}\.</code></td><td>43 628</td><td>6,374 (1.009x)</td><td>6,391 (1.007x)</td><td>7,185 (0.90x)</td><td>6,433 (1.00x)</td></tr>
<tr><td><code>sha1:[A-Z2-7]{32}</code></td><td>44 062</td><td>6,390 (1.12x)</td><td>6,353 (1.12x)</td><td>7,161 (0.997x)</td><td>7,140 (1.00x)</td></tr>
<tr><td><code>response</code></td><td>63 051</td><td>6,396 (1.15x)</td><td>6,386 (1.15x)</td><td>7,225 (1.02x)</td><td>7,363 (1.00x)</td></tr>
</table>

Tabel 18. Common Crawli WARC-faili vastete koguarvu loendamine, aeg sekundites. Sulgudes on tööriista kiirus defgrepi suhtes.

Muster	Tulemus	ripgrep (s)	zgrep (s)	ugrep (s)	defgrep (s)
DefinitelyNoSuchHeader2026	0	6,406 (0.87x)	6,376 (0.87x)	7,177 (0.77x)	**5,549 (1.00x)**
(?<=HTTP/1\.1\s)200\sOK	5 073	6,458 (0.55x)	6,379 (0.56x)	7,810 (0.46x)	**3,555 (1.00x)**
(?<=Server:\s)Apache	2 147	6,450 (0.54x)	6,398 (0.55x)	7,360 (0.48x)	**3,503 (1.00x)**
(?<=HTTP/1\.1\s)[0-9]{3}	22 033	7,832 (0.45x)	7,672 (0.46x)	11,255 (0.31x)	**3,526 (1.00x)**
(?<=WARC-Type:\s)response	22 031	6,449 (0.53x)	6,370 (0.54x)	8,913 (0.38x)	**3,420 (1.00x)**
200\sOK|404\sNot\sFound	5 188	6,320 (0.94x)	6,360 (0.93x)	7,279 (0.82x)	**5,938 (1.00x)**
WARC-Target-URI:\shttp://	5 706	6,323 (0.91x)	6,332 (0.91x)	7,338 (0.79x)	**5,780 (1.00x)**
User-Agent:\sCCBot	22 034	6,363 (0.87x)	6,385 (0.87x)	7,145 (0.77x)	**5,534 (1.00x)**
WARC-Type:\sresponse	22 031	6,369 (0.91x)	6,392 (0.90x)	7,181 (0.80x)	**5,772 (1.00x)**
WARC-Type:\srequest	22 031	6,347 (0.92x)	6,370 (0.91x)	7,185 (0.81x)	**5,808 (1.00x)**
WARC-Type:\smetadata	22 031	6,344 (0.91x)	6,388 (0.91x)	7,166 (0.81x)	**5,786 (1.00x)**
WARC-Payload-Digest:\ssha1:	22 031	6,422 (0.90x)	6,406 (0.90x)	7,188 (0.80x)	**5,749 (1.00x)**
Mozilla|CCBot	23 273	6,391 (0.90x)	6,359 (0.91x)	7,248 (0.80x)	**5,772 (1.00x)**
Server:\sApache|Server:\snginx|Server:\scloudflare	3 817	6,362 (0.88x)	6,347 (0.88x)	7,152 (0.78x)	**5,575 (1.00x)**
WARC-IP-Address:\s[0-9]{1,3}\.	43 628	6,408 (0.91x)	6,397 (0.91x)	7,195 (0.81x)	**5,800 (1.00x)**
sha1:[A-Z2-7]{32}	44 062	6,314 (1.001x)	**6,277 (1.007x)**	7,186 (0.88x)	6,323 (1.00x)
response	73 475	**6,418 (1.01x)**	6,802 (0.96x)	7,218 (0.90x)	6,496 (1.00x)