TARTU ÜLIKOOL

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse Instituut

Informaatika eriala

Kristo Tammeoja

Regulaaravaldiste ligikaudne otsimine bitt-paralleelse algoritmi abil

Semestritöö

Juhendaja: Jaak Vilo, PhD

Tartu 2006

Sisukord

1 Sissejuhatus

Regulaarsed keeled on formaalsete keelte alamjaotus, mis on määratud Chomsky hierarhia madalaima, kolmanda[1] tasemega. Nende esitamiseks on kolm võimalust: regulaaravaldised, determineeritud lõplikud automaadid ning mitte-determineeritud lõplikud automaadid. Väljendusvõimsuselt on need kolm varianti võrdsed. Kuid oma mugava esitlusviisi tõttu kasutatakse praktikas peamiselt regulaaravaldisi. Lõplikud automaadid on sellisel juhul lihtsalt üks võimalikke variante regulaaravaldisega esitatud keelde kuuluvate sõnade leidmiseks.

Esimesena võttis regulaaravaldised kasutusele Stephen Kleene 1950-ndatel. Peagi integreeris Ken Thompson esitatud regulaaravaldiste notatsiooni toe tollal levinud tekstiredaktoritesse QED ja ed. Hiljem on regulaaravaldiste süntaksisse lisatud mitmeid teksti töötlemist hõlbustavaid vahendeid, millest mõned on muutnud regulaaravaldised võimsamaks kui regulaarsed keeled.

Käesolev semestritöö annab lühikese ülevaate regulaaravaldistele aja jooksul lisandunud võimalustest, kuid põhiliselt keskendutakse bioinformaatikas huvi pakkuvale alamhulgale. Bioinformaatikas kasutatavad mustrid on suhteliselt lihtsad (tüüpilised näited on `[ILV]...SG.{0,10}R`, `A.{3,6}V[ILV][RK]`, `R[FWY].[AGS][ILV].{0,7}A[ILV]` jms). Seetõttu piirdutakse algoritme käsitlevas osas vaid traditsioonilisemate regulaaravaldiste süntaksisse lisatud võimalustega – kvantorite ning sümboliklassidega.

Lähemalt käsitleme üht uut laiendust kus regulaaravaldist (automaati) sobitatakse vigadega. Võimsuselt on ka see käsitletav lõpliku automaadina. Vastav determineeritud automaat oleks liiga suur, kuid mitte-determineeritud automaadi suurus kasvab lineaarselt lubatud vigade arvuga.

Töö praktilise osana valminud C++-is tehtud ligikaudset regulaaravaldiste otsimist võimaldavas teegis ongi kasutatud ühte mitte-determineeritud automaadil põhinevat algoritmi. Efektiivsuse tagamiseks kasutab algoritm biti-paralleelsust. Ning kuna regulaaravaldise ligikaudseks otsimiseks kasutatav lõplik automaat on konstrueeritav täpset otsimist

[1] Chomsky hierarhia kõrgeim tase on 0.

võimaldavate automaadi koopiatest, siis lihtsuse ja mälu kokkuhoiu huvides on täiendatud täpsel otsimisel kasutatavat algoritmi, mitte automaati ennast.

Regulaaravaldisest mitte-determineeritud lõpliku automaadi koostamisel on regulaarseid keeli käsitlevates õppematerjalides enam levinud Thompson'i algoritmi asemel kasutusel Glushkov'i automaat. Tänu Glushkov'i algoritmi poolt koostatud automaadi omadusele oli võimalik ligikaudse otsimise algoritmile omalt poolt kergesti lisada ka vigadeta piirkondade tugi.

Töö teises peatükis esitame regulaaravaldiste mõiste ja erijuhud. Kolmandas peatükis kirjeldame kahte regulaaravaldiste otsimiseks kasutatavat algoritmi. Lähemalt tutvustame bitt-paralleelset Glushkov'i automaadil põhinevat ligikaudset regulaaravaldiste otsimise algoritmi. Samas peatükis on kirjeldatud ka muudatusi, mis on vaja algoritmis teha, toetamaks vigadeta piirkondi ligikaudsetes regulaaravaldises. Töö neljandas peatükis toome ära teiste regulaaravaldiste teekidega võrdlemiseks tehtud testide tulemused.

2 Regulaaravaldised

Käesolev peatükk on sisu poolest jagatav kolme ossa. Esmalt toome regulaaravaldiste formaalse definitsiooni. Seejärel tutvume põhiliste rakendusprogrammide poolt lisatud laienduste ja võimalustega. Lõpuks pakume välja bioinformaatika rakendustes vajaliku alamhulga süntaksist.

2.1 Definitsioon

Definitsioon 1 (Regulaaravaldis): Tähistagu ε tühja sõnet (või stringi, i.k. *string*) ning olgu fikseeritud kasutatav lõplik mittetühi tähestik Σ. Sellisel juhul on regulaaravaldis rekursiivselt defineeritav järgmiselt

- ε ja $\alpha \in \Sigma$ on regulaaravaldised,
- kui RE_1 ja RE_2 on regulaaravaldised, siis on ka (RE_1), $(RE_1 \cdot RE_2)$, $(RE_1 \mid RE_2)$ ja (RE_1^*) regulaaravaldised.

□

Näiteks `(((A·T)|(G·A))·((A·G)|((T·G)·(A*))*))` on korrektne regulaaravaldis. Enne regulaaravaldise interpretatsioonini jõudmist vaatame veel, kuidas regulaaravaldisi mõnevõrra mugavamalt ülesse kirjutada..

Esituse kohmakuse tõttu seatakse enamasti regulaaravaldistes kastutatavatele erisümbolitele järgmised prioriteedid kahanevas järjekorras

- (ja)
- *
- ·
- |

ning ‚·' jäetakse kirjutamata. Niimoodi saame me toodud regulaaravaldise lihtsustada märksa loetavamale kujule `(AT|GA)(AG|TGA*)*`.

Igale regulaaravaldisele RE seatakse vastavusse stringide hulk (sageli lõpmatu), mida see regulaaravaldis ära tunneb. Seda hulka nimetatakse regulaaravaldise poolt määratud keeleks ning tähistatakse L(RE).

Definitsioon 2: Etteantud regulaaravaldise RE poolt määratud keele defineerime rekursiivselt:

- kui RE on ε, siis L(RE) = {ε}, ehk tühi string,
- kui RE on $\alpha \in \Sigma$, siis L(RE) = {α}, ehk RE poolt määratud täht,
- kui RE on kujul (RE_1), siis L(RE) = L(RE_1), st ümbritsevate sulgude äravõtmine jätab keele samaks,
- kui RE on kujul ($RE_1 \cdot RE_2$), siis L(RE) = L(RE_1) ·L(RE_2), st uude keelde kuuluvad stringid w = $w_1 \cdot w_2$, kus $w_1 \in L(RE_1)$ ja $w_2 \in L(RE_2)$ ning · tähistab stringide liitmist,
- kui RE on kujul (RE_1 | RE_2), siis $L(RE) = L(RE_1) \cup L(RE_2)$, ehk siis uude keelde kuuluvad kõik stringid, mis kuuluvad kas keelde $L(RE_1)$ või keelde $L(RE_2)$ ning
- kui RE on kujul (RE_1*), siis $L(RE) = L(RE_1)^* = \bigcup_{i=0}^{+\infty} L(RE_1)^i$, seejuures $L^0 = \{\varepsilon\}$ ning $L^i = L \cdot L^{i-1}$, seega uude keelde kuuluvad stringid, mis on saadud nulli või rohkema keelde $L(RE_1)$ kuuluva stringi üksteise otsa liitmisel.

□

Näidetena kasutame eelpool toodud regulaaravaldist `(AT|GA)(AG|TGA*)*` ning tema alamavaldisi (rasvast ja kaldkirja on kasutatud regulaaravaldiste erinevate osade märkimiseks ning mingit semantilist tähendust neil pole).

- L(`AT`) = {AT}
- L(`AT|GA`) = {AT, GA}
- L(`TG`**`A*`**) = {TG, TG**A**, TG**AA**, TG**AAA**, ...}

- L(*AG* | TG**A***) = {*AG*, TG, TG**A**, TG**AA**, TG**AAA**, ...}
- L((*AG* | TG**A***) (*AG* | TG**A***)) =

  {*AGAG*, *AG*TG, *AG*TG**A**, *AG*TG**AA**, ...,
  TG*AG*, TGTG, TGTG**A**, ...,
  TG**A***AG*, TG**A**TG, TG**A**TG**A**, TG**A**TG**AA**, ...}
- L((*AG* | TG**A***) *) = {ε,

  AG, TG, TG**A**, TG**AA**, TG**AAA**, ...,
  AGAG, *AG*TG, *AG*TG**A**, *AG*TG**AA**,...,
  TG*AG*, TGTG, TGTG**A**, ...,
  TG**A***AG*, TG**A**TG, TG**A**TG**A**, TG**A**TG**AA**, ...
  ...}

  Paneme tähele, et toodud keeles sisalduvad muuhulgas keeltes

  $L(\varepsilon) = L(\texttt{AG | TGA*})^0$,

  $L(\texttt{AG | TGA*}) = L(\texttt{AG | TGA*})^1$ ning

  $L(\texttt{(AG | TGA*) (AG | TGA*)}) = L(\texttt{AG | TGA*})^2$

  lubatud stringid.

Teooriast veel niipalju, et keeli, mis on kirjeldatavad regulaaravaldiste abil, nimetatakse regulaarseteks keelteks.

Praktikas on üheks regulaaravaldiste kasutusvõimaluseks kasutajalt saadud tekstiliste andmete kontrollimine. Näiteks saab regulaaravaldiste abil lihtsalt kontrollida e-maili aadressi, numbri ja kuupäeva formaati.

Teine praktikas oluline koht, kus regulaaravaldisi kasutatakse, on regulaaravaldiste abil esitatava info otsimine tekstist. Neid kahte kasutusjuhtu eristab peamiselt see, et info otsimisel huvitab meid lisaks teadmisele, kas otsitav asi esineb tekstis, ka selle asja täpne asukoht. Edaspidi nimetame teksti alamstringi, mis kuulub etteantud regulaaravaldise poolt määratud keelde, vasteks (i.k *match*). On selge, et tekstis võib esineda ka mitu vastest, kusjuures vastete pikkused võivad olla erinevad ning osad vasted võivad ka kattuda. Reeglina keskenduvad regulaaravaldisi kasutavad vahendid just sellele regulaaravaldiste rakendusele. Enamasti on neil sellisel juhul süntaksis eriline koht regulaaravaldise algu-

ses asuval katusel ‚^' ja regulaaravaldise lõpus asuval dollari märgil ‚$', mis vastavalt tähistavad, et regulaaravaldise vaste peab algama teksti algusest ning regulaaravaldise vaste peab lõppema teksti lõpus. Neid kahte erisümbolit koos kasutades saamegi jälle andmete formaadi kasutamiseks kõlbuliku regulaaravaldise.

2.2 Regulaaravaldiste lisavõimalused

Nagu juba eelpool öeldud võivad praktikas kasutatavad regulaaravaldised erineda oluliselt regulaaravaldise definitsioonis toodust. Erinevused võib põhiliselt jagada kolme klassi [Friedl].

- Esiteks, regulaaravaldiste esituse lihtsustamiseks kasutusele võetud võimalused. Neid kõiki on võimalik definitsioonis toodud võimaluste abil väljendada, kuid pahatihti oleks niimoodi saadud regulaaravaldis inimsilmale märksa segasem (peatükk 2.3).
- Teiseks võimalused, mis lubavad mitme vaste korral sobivat vastet täpsemini spetsifitseerida (peatükk 2.4).
- Viimaseks võimalused, mis lasevad regulaaravaldiste abil määrata ka mitteregulaarseid keeli. Sellised täiendused on andnud alust regulaaravaldisi teinekord teatud mõttes programmeerimiskeelteks nimetada. Üheks äärmuseks selles vallas võib pidada Perl-i[2], mis lubab regulaaravaldistes teatud kohtadesse suisa Perl-i koodi kirjutada. Uute võimaluste tõttu soovitatakse ingliskeelses terminoloogias täiendatud võimalustega regulaaravaldisi nimetada lihtsalt kui *regex*[3] või *pattern* (e.k muster) (peatükk 2.5).

Kõige tuntumad on ilmselt esimesse klassi kuuluvad lisavõimalused.

---

[2] Alates versioonist 5.0. Selles versioonis on sisse toodud ka võimalus kirjutada regulaaravaldisi selguse huvides mitmele reale (sisemiselt eemaldatakse enne otsingut kõik tühikud ja reavahetused ning teostatakse otsing järjelejäänud regulaaravaldisega) ning kasutada sealjuures ka kommentaare.

[3] Erinevalt klassikalist regulaaravaldist tähistavast *regexp*-ist ei ole see lühend inglise keelsest väljendist *regular expression*.

2.3 Regulaaravaldiste lisasümbolid

2.3.1 Kvantorid

Kvantorite ülesandeks on määrata, mitu korda mingi regulaaravaldise alamosa võib *vastes* esineda. Regulaaravaldise definitsioonis oli juba üks kvantor - * - ära toodud. Ja me teame, et näiteks $L((ab)^*) = \{\varepsilon, ab, abab, ...\}$.

Esmalt tuleks ilmselt ära märkida, et põhimõtteliselt kõigis regulaaravaldiste realisatsioonides on definitsiooniga võrreldes siiski ka midagi ära jäätud – reeglina puudub regulaaravaldiste süntaksis võimalus kasutada sümbolit ε. Selle kompenseerimiseks on sisse toodud kvantor ?, mis lubab alamavaldisel esineda 0 või 1 korda. Nii näiteks, `a?` = `(ε|a)`. Ainukene, ja praktikas mitteoluline, puudus on antud asenduse korral see, et ε regulaaravaldise süntaksist välja jätmisel ei ole võimalik esitada keelt $\{\varepsilon\}$.

Kolmas levinud kvantor on +, mis lubab alamavaldisel esineda 1 kuni lõpmatu arv kordi. Nii näiteks `a+` = `aa*`. Erinevalt *-ga määratud keelest jääb +-i puhul välja ainult tühistring ε (seda küll tingimusel, et alamavaldise enda poolt määratud keeles ei esine tühistringi). Seega $L((ab)+) = \{ab, abab, ...\}$.

Neljanda liigi moodustavad kvantorid kujul {n}, {n, m} ja {n,}, mis vastavalt nõuavad alamavaldise esinemist täpselt n, n kuni m ja n kuni lõpmatu arv korda. Seega $L((ab)\{2\}) = \{abab\}$, $L((ab)\{2,3\}) = \{abab, ababab\}$ ja $L((ab)\{2,\}) = \{abab, ababab, ...\}$.

Regulaaravaldise üleskirjutuse on ka kõik teised kvantorid sama prioriteediga nagu *.

2.3.2 Sümbolite klassid

Sümbolite klass määrab ära mingi hulga sümboleid, mis kõik võivad vastes antud kohal esineda. Näiteks Eestis üldkasutatavaid autonumbreid otsides tahame me, et vaste kolmel esimesel positsioonil oleksid numbrid.

Sümbolite klassi märgitakse kandiliste sulgudega ning lihtsamal juhul loetletakse seal sees kõik lubatud sümbolid. Näiteks `ka[mn]a` tähistab nii sõna `kama` kui ka sõna `kana`. Mugavuse huvides lubatakse määrata ka sümbolite vahemikke. Näiteks `[0-9]` tähis-

tab ühte suvalist numbrit ning `[0-9A-Z]` ühte suvalist numbrit või ladina tähestiku suurtähte.

Kui sümbolite klass algab katuse sümboliga ‚^', siis tähistatakse sümbolite klassiga hoopis keelatud sümboleid. Näiteks `[^xy]` ei luba antud kohal väikest `x` ja väikest `y`.

Kui välja arvata see, et sümbolite klass lubab antud kohal suvalise klassi kuuluva sümboli esinemist, siis muus osas käitub sümbolite klass täpselt samamoodi nagu üksik täht. Nii on võimalik kasutada sümbolite klasse koos kvantoritega. Näiteks `[0-9]+` tähistab ühte või enamat järjestikust numbrit, kusjuures vastes võivad erinevatel kohtadel olla erinevad numbrid (st viimati toodud regulaaravaldise vasteks sobivad nii `111` kui ka `123`).

Sümbolite klassi kõrval kasutatakse tihti ka erisümboleid. Levinum neist on punkt (‚**.**'), mis tähistab suvalist sümbolit. Näiteks `a.b` tähistab kolmest sümbolist koosnevat teksti, mille alguses on a ning lõpus b, vahepealne sümbol pole oluline.

Põhimõtteliselt on võimalik sümbolite klass ka ühendina lahti kirjutada (`[abc]` on sama, mis `(a|b|c)`), kuid selline tähistus on oluliselt tülikam, seda eriti lubamatuid sümboleid näitava sümbolite klassi (nagu `[^xy]`) korral. Pealegi suudavad regulaaravaldiste mootorid sümbolite klasse ühenditest märksa efektiivsemalt kasutada.

2.4 Vaste täpsem määramine

2.4.1 Ahned ja laisad kvantorid

Vaikimisi on regulaaravaldistes kasutatavad kvantorid ahned (i.k *greedy*), mis lihtsustatud kujul tähendab seda, et nad püüavad leida võimalikult pikka vastet. Reeglina on see kasulik, sest näiteks positiivseid täisarve otsiv regulaaravaldis `[0-9]+` leiab sellisel juhul tekstist „`Stephen Cole Kleene was born in 1909`" numbri 1909, mitte 190 või 19. Samas klassikaline näide, kus ahne kvantor soovitud tulemust ei anna, on HTML-i tag-ide otsimine. Võiks ju arvata, et selleks tööks sobib regulaaravaldis „`<.+>`", kuid kui seda kasutada tekstist „`<EM>first</EM> test`" esimese tag-i otsimiseks, siis me saame tulemuseks „`<EM>first</EM>`". Põhjuseks on siis asjaolu, et tag-i sisu tähistav „`.+`" on ahne ning püüab kaasata võimalikult palju sümboleid.

Ahnete kvantorite alternatiiv oleks laisad kvantorid (i.k *lazy quantifiers*), mis vastupidiselt ahnetele kvantoritele püüavad leida võimalikult lühikest vastet. Regulaaravaldiste süntaksis lisatakse tavalisele kvantori tähisele „?" lõppu. Meie regulaaravaldise korral saame „`<.+?>`", kusjuures „`.+?`" tähendaks siis seda, et me otsime lühimat alamstringi, mis jääb < ja > vahele. Toodud näites kasutatud teksti korral saaksime nüüd tulemuseks „`<EM>`".

Seega kui tekstis on vaste üheselt määratud, siis tagastavad ahned ja laisad kvantorid mõlemad sellesama ainsa vaste. Nende mõte on hoopis mitme vaste olemasolul õige vaste täpsem kirjeldamine.

2.4.2 Atomaarne grupp

Pisut teise tähendusega on aga atomaarne grupp (i.k *atomic grouping*). Atomaarsel grupil on kaks funktsiooni. Esiteks võimaldab ta teatud juhtudel osad potentsiaalsed vasted välja filtreerida. Teiseks aitab ta mõningate regulaaravaldiste otsimise kiiremaks teha. Atomaarse grupi süntaks on **`(>? ... )`**.

Atomaarse grupi töö mõistmiseks on kasulik teada regulaaravaldiste mootori tööpõhimõtet. Seetõttu selgitame enne näidete juurde minemist lühidalt otsimisalgoritmi.

Regulaaravaldiste mootorites, kus atomaarne grupp üldse lubatud on, toimub vaste leidmine põhimõtteliselt katsetamise teel (pisut pikemalt on sellest kirjas peatükis 3.1). Selleks võetakse esmalt ette tekst ning regulaaravaldis. Seejärel hakatakse võimalusel tähthaaval nii tekstis kui ka regulaaravaldises edasi liikuma. Niimoodi regulaaravaldises edasi liikudes tekivad üldjuhul paratamatult kohad, kus edasiliikumine pole üheselt määratud. Sellisteks kohtadeks võivad olla näiteks kvantorid, kus alamavaldise täpne arv tuleb alles vaste leidmise ajal välja, või siis `|`-ga eraldatud regulaaravaldiste osad. Regulaaravaldist tekstile sobitades tuleb kõik sellised valikukohad meeles pidada, sest kui mõnes kohas sai tehtud alguses vale valik, siis hiljem saame me samast kohast mõnda teist võimalust edasi proovida. Selline katsetamine võib aga teatud puhkudel osutuda mittesoovitavaks. Esiteks on see aeglane ning mõnikord võiks otsimise varem ära lõpetada. Teiseks on mõnikord vaste määramisel tähtis, et teatud variandid vaatamata jäetaks. Atomaarne grupp ongi võimalus sellise vajaduse regulaaravaldiste mootorile selgeks tegemiseks. Si-

suliselt tähendab atomaarne grupp seda, et kui regulaaravaldist tekstile sobitades oleme me suutnud atomaarsesse gruppi kuuluvale alamavaldisele kuidagi vaste leida, siis vajadusel võetakse tagasi kogu alamavaldise poolt sobitatud tekst ning atomaarsesse gruppi jäänud läbivaatamata valikukohti enam edasi ei uurita.

Vaatleme ühte lihtsat näidet. Meil on vaja leida komadega eraldatud failist read, mille esimesel väljal on tekst A ning neljandal väljal P. Seega sobib näiteks rida „A,B,C,P,E", kuid ei sobi rida „A,B,C,E,P", sest P asub valel väljal. Esmapilgul tundub, et sellise ülesande lahendamiseks sobib regulaaravaldis[4] `^A,(.*?,){2}P`. Kuid tegelikult sobiks sulgudes olevale laisale kvantorile vaatamata toodud regulaaravaldise korral ka rida „A,B,C,E,P". Regulaaravaldist selle reaga sobitades tegutsetakse esimesel korral täpselt nii nagu me seda soovime – enne regulaaravaldise täheni P jõudmist on tekstist kasutatud (alamavaldise `(.*?,)` poolt leitud alamosad on sulgudega tähistatud) „A,(B,)(C,)...". Kuna teksti järgmine täht on E, mitte P nagu regulaaravaldises oodataks, siis loetakse valitud tee tupikuks ning minnakse tagasi viimase kvantori (antud juhul on see viimane otsustuskoht) juurde. Järgmisel katsel on enne regulaaravaldise P-ni jõudmist sobitatud tekst „A,(B,)(C,E,)..." ning kuna edasi tuleb tekstis täht P, siis loetakse rida sobivaks.

Ülesande saaks lahendatud, kui vahepealseid väljasid leidev alamavaldis panna atomaarsesse gruppi: `^A,<?(.*?,){2}>P`. Nii vaadataks sulge sobitades läbi ainult esimene võimalus ning kui niimoodi käitudes kogu regulaaravaldist sobitada ei suudeta kuulutataks ka rida mittesobivaks[5].

Atomaarse grupi alaliik on possessiivsed kvantorid (i.k *possessive quantifiers*), mis annavad lihtsama süntaksi üksiku atomaarsesse gruppi kuuluva (ahne[6]) kvantori üleskirjutamiseks. Possessiivseid kvantoreid tähistatakse prefiksiga „+". Nii näiteks on „`x++`" se-

---

[4] ^ tagab, et vaste algaks rea algusest

[5] Korrektne oleks ära mainida, et antud ülesande puhul saaks hakkama ka atomaarset gruppi kasutamata `^A,([^,]*?,){2}P`.

[6] Paneme tähele, et atomaarsesse gruppi pole mõtet panna üksikut laiska kvantorit, sest selline konstruktsioon sobitaks kvantori poolt hõlmatud avaldist tekstile parajasti vähim lubatud arv kordi (laiskuse tõttu eelistatakse võimalikult lühikesi vasteid ning atomaarsuse tõttu enam pikemaid vasteid ei vaadataks).

mantiliselt sama mis „`(?>x+)`” ning seda süntaksit saab kasutada ka kõigi teiste kvantorite puhul.

2.5 Keerukamad lisavõimalused

Viimaseks vaatleme regulaaravaldiste laiendusi, mis muudavad regulaaravaldised võimsamaks kui regulaarsed keeled.

2.5.1 Vaata konteksti

Perl 5.0-is võeti kasutusele kaks võimsat konstruktsiooni: *lookahead* (e.k vaata edasi), mis võimaldab regulaaravaldise suvalisse kohta seada täiendavaid kontrolle sellele kohale järgnevale tekstile, ja *lookbehind* (e.k vaata tagasi), mis võimaldab regulaaravaldise suvalisse kohta seada täiendavaid kontrolle sellele kohale eelnevale tekstile. Nende konstruktsioonide ühine nimetus ongi *lookaround* (e.k vaata ringi).

Mõlemast konstruktsioonist on olemas nii positiivne kui ka negatiivne variant. Positiivne variant nõuab meie poolt näidatud tingimuse kehtimist. Näiteks kui me tahame tekstist leida nime John esinemisi, millele järgneb inglise keeles omastavat käänet tähistav „’s”, siis me võime kirjutada regulaaravaldise `John(?=\'s)`. Oluline on ära märkida, et vasteks on sellisel regulaaravaldisel „John”, mitte „John’s”, sest *lookahead*-is leitud teksti ei lisata kunagi vastele juurde. Analoogiliselt, kui me tahaksime leida nime John, millele ei järgne lõppu „’s”, siis võiksime me kasutada negatiivse *lookahead*-iga regulaaravaldist `John(?!\'s)`.

Lookbehind-il on sarnane süntaks: positiivset *lookbehind*-i tähistatakse `(?<=...)` ja negatiivset `(?<!...)`. *Lookbehind*-i juures tuleks tähele panna, et enamasti ei lubata kontrolliks kasutada fikseerimata vaste pikkusega regulaaravaldisi (sest regulaaravaldiste mootorid töötavad tekstis edaspidi liikudes, *lookbehind* nõuaks aga regulaaravaldise tagurpidi tekstile sobitamist). Tavaliselt on *lookbehind*-is kasutatavates regulaaravaldistes lubatud kasutada vaid tähti, sümbolite klasse ning erisümbolit „|”.

2.5.2 Tagasi viide

Paljudes regulaaravaldiste mootorites on võimalik lisaks vastele endale teada saada ka sulgudega tähistatud alamavaldiste poolt leitud tekst. Sellist funktsionaalsust nimetatakse

tagasi viiteks (i.k *backreference*). Reeglina on need alamavaldised nummerdatud alates 1-st ning nad on järjestatud vastavalt alustavale sulule. Vaatleme ühte eelpool toodud regulaaravaldist pisut muudetud kujul `^(A,(.*+,){2})P`. Kui seda rakendada tekstile „A,B,C,P,E", siis me saame esimese alamavaldise väärtuseks „A,B,C," ning teise alamavaldise väärtuseks „C,". Paneme tähele, et teiste sulgude sees olev alamavaldist sobitatakse tekstile kaks korda, esimest korda „B,"-le ning teist korda „C,"-le, kuid alamavaldise väärtuseks võetakse viimane vaste.

Huvitavam on aga võimalus kasutada alamavaldiste poolt leitud väärtusi juba regulaaravaldise enda sees. Eelpool vaatlesime, kuidas leida regulaaravaldise abil ühte HTML-i tag-i. Seekord üritame aga leida HTML-i tag-i koos tema lõpetava tag-iga. Arusaadavalt peab alustavas tag-is olema sama märgend nagu lõpetavas tag-is. Tavapäraste regulaaravaldistega pole sellist tulemust võimalik saavutada, kuid tagasi viite abil saab selle kirjutada näiteks nii `<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>`. Lahtiseletatuna tähendab see seda, et HTML-i parsides leitakse kõigepealt „<"-le järgnev tähega algav ning tähtedest ja numbritest koosnev märgendi nimetus. Kuna vastav alamavaldis on esimene sulgudega eraldatud alamavaldis selles regulaaravaldises, siis on võimalik selle alamavaldise poolt leitud märgendit lõpetava tag-i leidmisel `\1` abil regulaaravaldises uuesti kasutada. Selliselt koostatud regulaaravaldis leiab näiteks vaste tekstist „`<A alt="a.html"> link </A>`" kuid ei leia vastet tekstist „`<B>tekst</C>`", sest alustavas ja lõpetavas tag-is on erinev märgend.

2.6 Ligikaudne otsimine

Tekstialgoritmides on lisaks teksti täpsele esinemisele kasutusel ligikaudse otsimise mõiste. Kõige tüüpilisem on teisenduskaugus ehk Levenshteini kaugus. Käesolevas peatükis laiendame seda mõistet regulaaravaldistele.

Mitteformaalselt võib ütelda, et regulaaravaldise ligikaudse otsimise, teisisõnu ka vigadega otsimise, eesmärgiks on leida vasted, mis „peaaegu" vastavad etteantud regulaaravaldisele. Antud töös vaatleme me „peaaegu" vastavuse mõõduna tekstide võrdlemisest tuntud Levenshteini kaugust ehk sõnede teisenduskaugust. Levenshteini kauguse korral loetakse sõnede kauguseks teisenduste arvu, mis on vajalik sõnede võrdseks saamiseks,

kusjuures teisendustena on lubatud tähe kustutamine, lisamine ning asendamine. Seega regulaaravaldiste ligikaudsel otsimisel tahame me leida vasteid, mis on võimalik ülimalt lubatud arvu paranduste järel etteantud regulaaravaldisega täpselt sobitada.

Praktikas on suhteliselt palju rakendusi, kus ligikaudne otsimine vajalikuks võib osutuda. Näiteks tavalisest tekstist mingi märksõna otsimisel tahaksime me leida ka need kohad, kus otsitav sõna trükiveaga esineb. Nii leiaksime me sõna `regulaaravaldis` otsides ka „`...on lisatud` **`regulaalavaldis`**`te süntaksisse...`" (allajoonitult on toodud koht, kus ühe tähe asendamisega oleks võimalik saada täpselt otsitav sõna). Bioinformaatikas on sageli vaja otsida mustreid, mis võivad sekventsidel esineda teatavate mutatsioonidega, mis on omandatud näiteks evolutsiooni käigus.

Mustrite otsimisel pole sageli probleemiks mitte niivõrd kõigi mustriga sobivate vastete leidmine kuivõrd just oluliste vastete leidmine. Täpse märksõna järgi otsimise korral on vaste ühesuse tõttu olukord suhteliselt lihtne. Regulaaravaldiste korral muutub olukord pisut keerulisemaks seetõttu, et ka samast alguspositsioonist võib mõnikord mitu erinevat vastet leida, näiteks regulaaravaldis **`N..R.`**`{1,4}`**`RR`** ja tekst „`...`**`N`**`AC`**`R`**`KV`**`RR`**`R...`". Kuid ligikaudsuse lubamine regulaaravaldiste otsimisel tekitab juba üsnagi olulisel määral mitterelevantseid tulemusi. Näiteks otsides regulaaravaldist `babb*bab` tekstist „`...aba`**`babbbab`**`aaa...`" leidub meil üks täpne vaste. Samas kui me lubaksime otsimisel kuni kahte viga, siis saaksime me täpse vaste otsa tähti lisades ning tähti kustutades 12 triviaalset vastet **`babbb__`**, `a`**`babbba_`**, **`babbba_`**, **`_abbba_`**, `ba`**`babbbab`**, `a`**`babbbab`**, **`_abbbab`**, **`__bbbab`**, `a`**`babbbab`**`a`, **`babbbab`**`a`, **`_abbbab`**`a` ja **`babbbab`**`aa`. Käesolevas töös me sellise müra filtreerimisele tähelepanu ei pööra, kuna see jääb rohkem mustrite kaevandamise ning olemasolevate mustrite parendamise valdkonda [MOG98].

Ligikaudse otsimise üheks laienduseks on nn vigadeta piirkondade lubamine regulaaravaldises. See võimaldab meil vigu lubaval otsimisel määrata regulaaravaldise piirkondi, mida on vaja kindlasti täpselt sobitada. Kuna juba ligikaudne otsimine ise on suhteliselt haruldane võimalus regulaaravaldiste mootorite juures, siis puudub ka ühtne süntaks nii lubatud vigade kui ka vigadeta piirkondade tähistamiseks. TRE-nimelises regulaaravaldiste mootoris [Laurikari3], mida me hiljem testides kasutame, märgitakse lubatud vigade

arv alamavaldise järele loogelistesse sulgudesse[7]. Käesoleva töö raames realiseeritud regulaaravaldiste mootoris on piirdutud vähem paindliku süntaksiga, kus kogu regulaaravaldises lubatud vigade arv märgitakse kooloniga eraldatult regulaaravaldise ette (see konstruktsioon lubatakse ka ära jätta, sellisel juhul teostatakse lihtsalt regulaaravaldise täpne otsing) ning vigadeta piirkond tähistatakse suurem-väiksem märkidega[8]. Näiteks `2:e<f>ektiivne` leiab tekstist sõnad `effekdiivne` ja `efektiivse`, kuid mitte sõna `selektiivne`.

2.7 Ülesande detailsem kirjeldus

Käesoleva semestritöö eesmärgiks on realiseerida bioinformaatikas mustrite otsimiseks vajaliku regulaaravaldiste alamhulgaga töötav algoritm. Bioinformaatikas kasutatavad mustrid on suhteliselt lihtsad (tüüpilised näited on `[ILV]...SG.{0,10}R`, `A.{3,6}V[ILV][RK]`, `R[FWY].[AGS][ILV].{0,7}A[ILV]` jms). Seetõttu on süntaksis piirdutud vaid traditsioonilisemate regulaaravaldiste lisavõimalustega – kvantorite ning sümboliklassidega.

Võrreldes olemasolevate vahenditega on käesoleva töö raames tehtud programmil kaks eripära.

- Esiteks võimaldab see otsida regulaaravaldisi ligikaudselt. Mõningatel olemasolevatel programmide on samuti olemas ligikaudse otsimise võimalus, kuid nad kas töötavad seejuures suhteliselt aeglaselt või tagastavad ainult tekstiread (mitte vaste enda), kus sellised vasted asusid.
- Teiseks tagastatakse kõik võimalikud vasted. Ülejäänud vahendid leiavad iga teksti positsiooni kohta ülimalt ühe vaste (st kui teksti mingist positsioonist alates leidub kaks erineva pikkusega vastet siis nendest tagastatakse vaid üks) ning järgmist hakatakse otsima alles pärast eelmisena leitud vaste lõppu. Seega ei tagastata

[7] Näiteks {~3} määrab lubatud vigade arvuks 3-e, {+2~5} aga ütleb, et tähe lisamisi tohib teha ülimalt 2, kuid kogu vigade arv võib olla ülimalt 5.

[8] Selline tähistus on kasutusel ka stringide ligikaudsel otsimisel

kattuvaid vasteid, mis mustrite otsimise juures võib kaotada mõningad huvitavad tulemused.

Kasutatud on järgmises peatükis lähemalt tutvustatavat algoritmi, millele on omalt poolt lisatud vigadeta piirkondade kasutamise võimalus.

3 Regulaaravaldiste otsimisel kasutatavad algoritmid

Regulaaravaldiste otsimise kaks enamtuntud algoritmi on:

- *backtracking* ja [Friedl]
- lõplikud automaadid. [Navarro]

Esimene neist on oluliselt paindlikum, lubades näiteks peatükis 2.5 tutvustatud tagasi viitamist ja konteksti vaatamise funktsionaalsust. Kõik nimetatud võimalusi lubavad regulaaravaldiste mootorid kasutavad tõenäoliselt just seda algoritmi. Algoritmi põhiidee on käia tekst sümbolhaaval läbi ning üritada sobitada puu kujule teisendatud regulaaravaldist igale teksti positsioonile.

Teine, vanem, piiratumate võimalustega, kuid kiirem nendest põhineb otseselt formaalsel keeleteoorial. Nimelt saab igast regulaaravaldisest konstrueerida talle vastava mittedeterminееritud lõpliku automaadi (ing. k *Nondeterministic Finite Automaton*, e. NFA) ning sellest omakorda determineeritud lõpliku automaadi (ing. k *Deterministic Finite Automaton*, e. DFA). Regulaaravaldiste otsimist on võimalik teostada nii NFA kui ka DFA-ga. NFA-ga otsimise keerukus on halvimal juhul $O(nm)$ ning DFA-l alati $O(n)$, kus *n* on teksti pikkus ning *m* regulaaravaldise pikkus. Samas mäluvajadus on NFA-l alati $O(m)$, kuid DFA-l halvimal juhul $O(2^m)$.

Võrreldes lõplike automaatidega on *backtracking*-ul kaks suuremat eelist.

- Esiteks on võimalik lihtsasti kindlaks määrata mingi regulaaravaldise alamosa poolt leitud tekst. See on ka tema täiendavate omaduste, näiteks tagasi viite, kasutamise eelduseks.
- Teiseks võimaldab selline lähenemine saada regulaaravaldiste vasted kätte nende tekstis esinemise järjekorras.

Nende põhjuseks on asjaolu, et automaadid on mõeldud vastamaks küsimusele, kas antud regulaaravaldis esineb otsitavas tekstis. Täpsemalt, automaadi abil saab kätte parajasti kõik vastete lõpud. Vasted ise on võimalik teksti teistkordsel läbimisel kindlaks teha, kuid ilma täiendava töötlemiseta saame me vasted kätte nende lõppemise järjekorras, mis ei pruugi olla päris ootuspärane tulemus.

Bactracking-u peamiseks probleemiks on aga tema halb ajaline keerukus. Nimelt võib isegi sellise lihtsa avaldise nagu `(a|aa)*b` korral osutuda *backtracking*-u algoritmi tööaeg teksti pikkuse suhtes eksponentsiaalseks (reeglina realiseerub halb ajaline keerukus just vaste puudumisel). Seetõttu üritavad kõik praktikas kasutatavad *backtracking*-ul põhinevad regulaaravaldiste mootorid optimeerida mõningaid olukordi, kus algoritm muidu aeglased oleks. *Bactracking*-u kaitseks tuleks siiski ära märkida, et praktikas kasutatavate regulaaravaldistega töötab ta piisavalt hästi (viimati toodud regulaaravaldis on ilmselgelt kunstlikult konstrueeritud, sama keele määraks ka märksa lihtsam regulaaravaldis `a*b`). Lisaks on *backreference*-dega täiendatud regulaaravaldisese otsimine üldisemal juhul NP-täielik ülesanne.

Neid kahte algoritmi kasutavate regulaaravaldiste mootorite kõrval on tehtud ka realisatsioone, mis kasutavad mõlema algoritmi paremaid omadusi. Selleks leitakse DFA abil kiiresti potentsiaalsed vastete asukohad ning aeglasemat *backtracking*-ut kasutatakse alles täpse vaste kindlaksmääramisel.

Enne algoritmide kirjeldamist märgime veel ära, et inglise keelses kirjanduses kasutatakse sageli pisut eksitavat terminoloogiat, kus automaatidel põhinevale algoritmile viidatakse sageli kui DFA-le ning *backtracking*-u algoritmile kui NFA-le (kuigi *backtracking*-ul ja mittedetermineeritud lõplikul automaadil ehk NFA-l pole põhimõtteliselt midagi ühist).

3.1 Backtracking

Backtracking-u tööpõhimõte on väga lihtne. Selle tundmine aitab esiteks mõista milliseid vasteid ja miks *backtracking*-ul põhinevad regulaaravaldiste mootorid annavad. Lisaks on võimalik tööpõhimõtet tundes aimata ära kuidas ning miks just sellised lisavõimalused regulaaravaldistele juurde on tekkinud.

Backtracking-u korral toimub otsimine sisuliselt positsioonhaaval regulaaravaldise tekstile sobitamise abil. Kõigepealt üritatakse regulaaravaldist sobitada teksti esimesele positsioonile. Kui see ebaõnnestub, siis nihutatakse otsimise algus (millest hiljem saab ka tegelikult vaste algus) järgmisele sümbolile jne.

Regulaaravaldise tekstile sobitamise illustreerimiseks kasutame regulaaravaldist `hommi-ku?(ni|l|ks)?`, mis on võimeline tuvastama sõnu „hommik”, „hommiku”, „hommi-

kuni", „hommikul", „hommikuks", aga paraku ka „hommikni", „hommikl" ja „hommikks", kuid ärme lase ennast viimasel puudusel häirida. Üritame seda regulaaravaldist sobitada tekstile „tulen hommikul". Põhimõtteliselt sobiks vasteks nii „hommik", „hommiku" kui ka „hommikul", kuid vaatame, miks enamus *backtracking*-ul põhinevaid mootoreid just viimase variandi valivad.

Vaste otsimist alustatakse sellega, et regulaaravaldist üritatakse sobitada tekstile alates teksti esimesest tähest ‚t'. Ilmselgelt viib see tupikuni, sest meie regulaaravaldisele sobib esimeseks täheks vaid ‚h'. Järgmisena üritataks regulaaravaldist sobitada tähest ‚u', kuid ka see ei vii kuhugi. On lihtne näha, et midagi hakkab hargnema alles tähest ‚h'. Kuna ‚h' sobis, siis jäetakse vaste algus ‚h' meelde ning liigutakse nii tekstis kui ka regulaaravaldises ühe sammu võrra edasi. Regulaaravaldises on edasi liikumisel ainuke võimalus lugeda sisse täht ‚o'. Kuna see ka õnnestub, siis liigutakse edasi regulaaravaldise kolmanda täheni jne.

Esimene valikukoht tekib alles regulaaravaldise tähe ‚u' juures, kuna kvantori tõttu on võimalik ‚u' nii sisse lugeda kui ka lugemata jätta. Eelnevalt kvantoreid kirjeldades märkisime me ära, et vaikimisi on kvantorid ahned, st nad üritavad ennast tekstile sobitada võimalikult palju kordi. Seetõttu eelistatakse praegu ‚u' sisse lugeda. Sisemiselt aga jäetakse meelde, et selle valiku ebaõnnestumise korral võime me alati samasse kohta tagasi tulla ning ‚u' sisselugemise asemel ‚u' lugemata jätta (sellisest tagasi tulemisest ongi nimi *backtracking*).

Pärast ‚u' sisselugemist on jälle valiku tegemise koht. Variante on seekord neli ning neid hakatakse läbi vaatama just sellises järjekorras: „ni", „l", „ks" ning kogu selle sulu vahelejätmine. Valik „ni" osutub sobimatuks juba esimese tähe tõttu. Edasi minnakse variandi „l" juurde ning kuna ‚l'-i sisselugemise järel jõutakse regulaaravaldise lõppu, siis loetakse vaste leituks ning katkestatakse kogu otsing.

Juba nii lihtsa näite korral saab selgeks, et ahned ja laisad kvantorid on tegelikult kasutusele võetud just regulaaravaldise mootori valikute juhtimiseks valikuvõimaluste korral. Teiseks saab selgeks, miks on *backtracking*-u algoritmi korral kerge kätte saada nii vastet ennast kui ka erinevate alamavaldiste poolt leitud piirkondi tekstist, selleks piisab vaid meelde jätta, millal me viimati antud alamavaldist regulaaravaldise tekstile sobitamisel

kasutasime. Siit edasi tagasi viidete kasutamiseni regulaaravaldises endas (peatükk 2.5.2) on ainult väike samm.

Sageli tavatsetakse ütelda, et *backtracking*-ul töötavad mootorid tagastavad „pikima vasakpoolseima vaste". Vasakpoolsusega probleeme pole. See tuleneb ilmselgelt sellest, et teksti vaadatakse läbi vasakult paremale ning tagastatakse esimene leitud vaste. „Pikim" tuleneb aga sellest, et vaikimisi on kvantorid ahned (ja eelistavad võimalikult pikki vasteid). Kuid tegelikult on „pikim" mõnevõrra eksitav termin, sest regulaaravaldise mootor ei vaata kõiki võimalikke variante läbi, vaid valib oma kindla järjekorra variantide läbivaatamiseks ning lõpetab töö esimese sobiva variandini jõudmisel. Nii näiteks leiavad regulaaravaldised `kala(l|le)` ja `kala(le|l)` tekstile „kalale minnes..." sobitades vastavalt „kalal" (mis ei ole pikim võimalik vaste) ja „kalale".

Nüüd, kui me juba tunneme *backtrack*-ingu tööd ning teame, et selle taustal toimub tegelikult täiesti tavaline rekursiivne läbivaatus, oskame me ka ütelda, miks osade regulaaravaldiste mootorite korral võib regulaaravaldise `(a|aa)*b` pikemale ‚a'-dest koosnevale tekstile rakendades väga kaua aega võtta. Suur vahe erinevate regulaaravaldiste mootorite töökiiruse vahel võib tekkida sellest, et *backtracking*-u algoritmi on võimalik nii mõnegi erijuhu jaoks optimeerida. Nii näiteks suudab Perl 5.0 erinevalt regulaaravaldiste algusaegadel tehtud regulaaravaldiste mootoritele toodud regulaaravaldisega väga edukalt toime tulla.

3.2 Lõplik automaat

Enne automaatidel põhineva algoritmi juurde minemist toome ära pisut teooriat.

3.2.1 Definitsioon

Definitsioon. Lõplik automaat on järjestatud viisik

$A = (\Sigma, S, I, F, \delta)$, kus

- Σ on tähestik (lõplik mittetühi sümbolite hulk),
- S on lõplik ja mittetühi seisundite hulk,
- $I \in S$ on automaadi algseisund,

- $F \subset S$ on automaadi lõppseisundite hulk (võib olla tühi) ning
- $\delta \subset S \times \Sigma \times S$ on üleminekuseos.

Juhul kui üleminekuseos on esitatav funktsioonina $\delta : S \times \Sigma \to S$ (st igale seisundist ja tähest koosnevale paarile vastab ülimalt üks seisund), siis nimetatakse automaati lõplikuks determineeritud automaadiks (i.k. DFA – *Deterministic Finite-State Automaton*), vastasel juhul lõplikuks mitte-determineeritud automaadiks (i.k. NFA – *Nondeterministic Finite-State Automaton*).

Kõige levinum on lõpliku automaadi (nii determineeritud kui ka mitte-determineeritud) esitus suunatud graafina, kus tippudeks on seisundid ning kaarteks on tähtedega märgendatud üleminekuseosed. Sellisel juhul kuulub etteantud sõna *x* lõplik automaadi A poolt määratud keelde parajasti siis, kui sellele sõnale vastav tee automaadi A graafis algab algolekust ja lõpeb lõppolekus.

3.3 Mittedetermineeritud lõplik automaat (NFA)

Selle. eatüki kirjutamisel kasutasin peamiselt Gonzalo Navarro ja Mathieu Raffinot raamatut „Flexible Pattern Matching in Strings" ([Navarro]). Kuna me võtsime käesoleva töö raames eesmärgiks teha programm, mis toetaks ligikaudset otsimist, siis piirdume me siinkohal vaid mittedetermineeritud automaatidel põhinevate bitt-paralleelsete algoritmide vaatamisega. Põhjuseid on kaks

- Nagu me hiljem näeme, siis meetod, kuidas vigadeta regulaaravaldiste otsingu algoritmist saadakse vigasid lubav algoritm toob paratamatult kaasa selle, et automaat muutub mittedetermineerituks. Seega automaadi determineerimine ei annaks meile sisulist võitu.
- Nagu me eelpool nägime, siis bioinformaatikas läheb reeglina vaja just lühikesi regulaaravaldisi. Samas praktika näitab, et lühikeste regulaaravaldiste korral osutuvad kiiremaks just bitt-paralleelsusel põhinevad lahendused.

Regulaaravaldisest NFA konstrueerimiseks on olemas mitmeid meetodeid. Nendest tuntumad on Thompson'i ja Glushkov'i algoritmid.

Lihtsam ja formaalset algoritmiteooriat käsitlevates raamatutes enam käsitletud on nendest Thompson'i algoritm, mis annab tulemuseks NFA, kus on ülimalt $2m$ seisundit ja $4m$ üleminekuseost. Mõnevõrra tülikaks asjaoluks on see, et Thompson'i algoritm tekitab ka ε-üleminekuid.

Samas Glushkov'i algoritm annab tulemuseks täpselt $m+1$ seisundist koosneva ε-üleminekuteta automaadi, kuid paraku võib üleminekuseoste arv ulatuda selles halvimal juhul $O(m^2)$-ni. Lihtsam algoritm Glushkov'i automaadi konstrueerimiseks annab ajaliseks keerukuseks halvimal juhul $O(m^3)$, kuid on näidatud, et seda saab vähendada ka $O(m^2)$ peale. Kuna reeglina on regulaaravaldised suhteliselt lühikesed, siis ei ole Glushkov'i automaadi suurus ning tema konstrueerimise keerukus eriliseks probleemiks. Küll aga huvitab meid bitt-paralleelsete algoritmide juurest tema Thompson'i automaadiga võrreldes mõnevõrra väiksem seisundite arv. Nimelt koostavad kõik regulaaravaldiste otsimisel kasutatavad bitt-paralleelsusel põhinevad algoritmid oma töös üleminekuteisenduste haldamiseks 2^q elemendist koosneva tabeli, kus q on automaadi seisundite arv (tõsi küll, seda tabelit võib mälu kokkuhoiu huvides tükeldada, kuid see toob paratamatult omakorda kaasa algoritmi kiiruse langemise), ning seetõttu võib olla automaati seisundite lisamine küllaltki kallis.

3.3.1 Glushkov'i automaat

Põhiidee Glushkov'i automaadi konstrueerimise juures on tekitada iga regulaaravaldises oleva tähe kohta üks seisund ning lisaks veel üks seisund, mis tähistab regulaaravaldise algust. Üleminekud näitavad sellisel juhul, millised tähed võivad regulaaravaldise tekstile sobitamisel antud tähele järgneda. Algoritmi kirjeldamiseks kasutame me näitena regulaaravaldist `(AT|GA)((AG|AAA)*)`. Selle regulaaravaldise korral saame näiteks, et T-le vastavast seisundist on võimalik minna edasi tärniga varustatud alamavaldise alguses olevate A-deni (**A**`G` ja **A**`AA`). Aga vaatame algoritmi täpsemalt.

Algoritmi esimese sammuna nummerdame me kõik regulaaravaldises olevad tähed nende järjekorranumbri alusel. Nii saame me oma näitena toodud regulaaravaldisest $(A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)$. Regulaaravaldisest R moodustatud märgendatud regulaaravaldist tähistatakse $\overline{R}$ ning tema poolt määratud keelt, kus igal tähel on ka sobiv

indeks, tähistatakse $L(\overline{R})$. Meie näite korral L((A$_1$T$_2$|G$_3$A$_4$)((A$_5$G$_6$|A$_7$A$_8$A$_9$)*)) = {A$_1$T$_2$, G$_3$A$_4$, A$_1$T$_2$A$_5$G$_6$, G$_3$A$_4$A$_5$G$_6$, A$_1$T$_2$A$_7$A$_8$A$_9$, G$_3$A$_4$A$_7$A$_8$A$_9$, A$_1$T$_2$A$_5$G$_6$A$_5$G$_6$, ...}. Toome nüüd veel sisse järgmised tähistused. Olgu $Pos(\overline{R}) = \{1,...,m\}$ regulaaravaldises kasutatud indeksite hulk ning $\overline{\Sigma}$ indeksitega varustatud Σ tähtedest koosnev tähestik.

Esmalt ehitatakse Glushkov'i automaat keelt $L(\overline{R})$ määrava regulaaravaldise $\overline{R}$ jaoks. Keele $L(R)$ äratundmiseks kasutatav automaat saadakse siis $\overline{R}$ põhjal ehitatud automaadist indeksite ärakustutamisel.

Automaadi ehitamise esimese sammuna tehakse $m+1$ seisundit märgenditega 0 kuni *m*. Seisundid 1 kuni *m* tähistavad vastava indeksiga tähti regulaaravaldises ning märgendiga 0 seisund on automaadi algseisundiks. Niimoodi seisundeid märgendades saavutame me selle, et kui automaati tekstile sobitades on saanud aktiivseks seisund *j*, siis loeti viimati sisse regulaaravaldise *j*-ndal positsioonil olnud täht. Lugedes sisse uue tähe α peame me nüüd teadma, millisesse seisundisse me *j*-ndast seisundist selle tähe abil edasi saame minna.

Et üleminekute konstrueerimist oleks lihtsam selgitada toome kõigepealt sisse neli uut mõistet. Regulaaravaldise $\overline{R}$ *y*-ndal positsioonil olevat indeksiga tähte tähistame edaspidi α_y-ga.

Definitsioon. $First(\overline{R}) = \{x \in Pos(\overline{R}), \exists u \in \overline{\Sigma}^*, a_x u \in L(\overline{R})\}$

Seega $First(\overline{R})$ näitab ära kõik tähed, millega keelde $L(\overline{R})$ kuuluvad sõnad võivad alata, või teisisõnu, kõik positsioonid regulaaravaldises $\overline{R}$, millest võib regulaaravaldise tekstile sobitamine alata. Meie näite korral First((**A$_1$**T$_2$|**G$_3$**A$_4$)((A$_5$G$_6$|A$_7$A$_8$A$_9$)*)) = {1, 3}.

Definitsioon. $Last(\overline{R}) = \{x \in Pos(\overline{R}), \exists u \in \overline{\Sigma}^*, u a_x \in L(\overline{R})\}$

Seega $Last(\overline{R})$ näitab ära kõik tähed, millega keelde $L(\overline{R})$ kuuluvad sõnad võivad lõppeda, või teisisõnu, kõik positsioonid regulaaravaldises $\overline{R}$, kus regulaaravaldise tekstile so-

bitamine võib lõppeda. Meie näite korral Last((A$_1$**T$_2$** | G$_3$**A$_4$**) ((A$_5$**G$_6$** | A$_7$A$_8$**A$_9$**)*)) = {2, 4, 6, 9}.

Definitsioon. $Follow\left(\overline{R},x\right)=\left\{y\in Pos\left(\overline{R}\right),\exists u,v\in\overline{\Sigma}^{*},ua_{x}a_{y}v\in L\left(\overline{R}\right)\right\}$

Hulk $Follow\left(\overline{R},x\right)$ näitab kõiki tähti, mis võivad keeles $L\left(\overline{R}\right)$ tähe α_x järel tulla, või teisisõnu, kõiki positsioone regulaaravaldises, mis võivad *x*-nda positsiooni järel aktiivseks muutuda. Meil näiteks *Follow*((A$_1$T$_2$ | G$_3$A$_4$) ((**A$_5$**G$_6$ | **A$_7$**A$_8$A$_9$)*), 6) = {7, 5}.

Viimase uue mõistena defineerime hulga $Empty_R$

Definitsioon. $Empty_R$ väärtuseks on $\{\varepsilon\}$ kui tühi string ε kuulub keelde $L(R)$, ning Ø vastasel juhul. Selle hulga väärtuse saab analoogiliselt regulaaravaldise definitsioonis tooduga ülesse kirjutada rekursiivselt.

$Empty_{\varepsilon} = \{\varepsilon\}$

$Empty_{\alpha\in\Sigma} = \emptyset$

$Empty_{\mathrm{RE1}\,|\,\mathrm{RE2}} = Empty_{\mathrm{RE1}} \cup Empty_{\mathrm{RE2}}$

$Empty_{\mathrm{RE1}\,\cdot\,\mathrm{RE2}} = Empty_{\mathrm{RE1}} \cap Empty_{\mathrm{RE2}}$

$Empty_{\mathrm{RE}^*} = \{\varepsilon\}$

Keelt $L\left(\overline{R}\right)$ äratundva determineeritud Glushkov'i automaadi $\overline{GL}$ saab nüüd järgmiselt

$\overline{GL}=\left(\overline{\Sigma},S,I,F,\overline{\delta}\right),$

kus

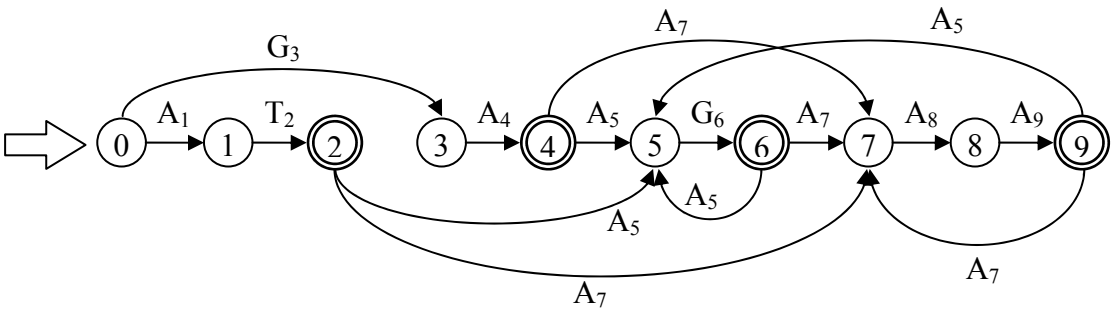
(i) S on seisundite hulk, $S=\{0,1,...,m\}$ ning algseisund $I=0$

(ii) F on lõppseisundite hulk, $F=Last\left(\overline{R}\right)\cup\left(Empty_{R}\cdot\{0\}\right)$. Ehk teisisõnu, seisund *i* on lõppseisund, kui ta kuulub hulka $Last\left(\overline{R}\right)$. Erandiks on seisund 0, mis kuulub lõppseisundite hulka parajasti siis, kui $\varepsilon\in L(R)$

(iii) $\overline{\delta}$ on üleminekute hulk, mis on defineeritud kui
$\forall x\in Pos\left(\overline{R}\right),\ \forall y\in Follow\left(\overline{R},x\right),\ \overline{\delta}\left(x,a_{y}\right)=y$

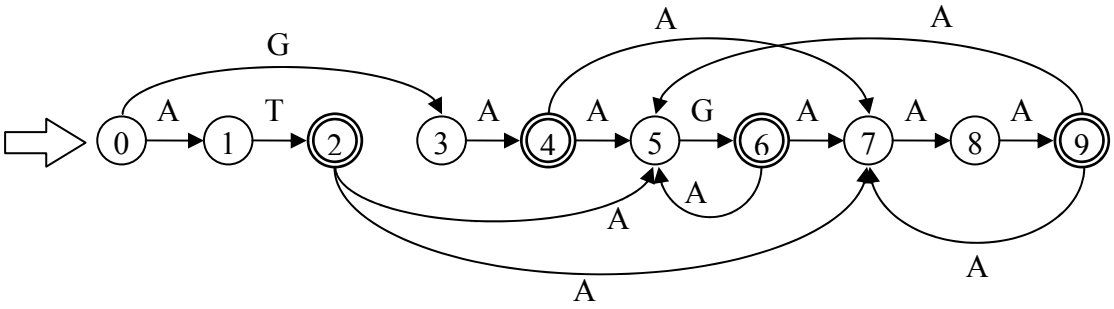
Algseisust väljuvad üleminekud on defineeritud kui

$\forall y \in First(\overline{R}),\ \overline{\delta}(0, a_y) = y$.

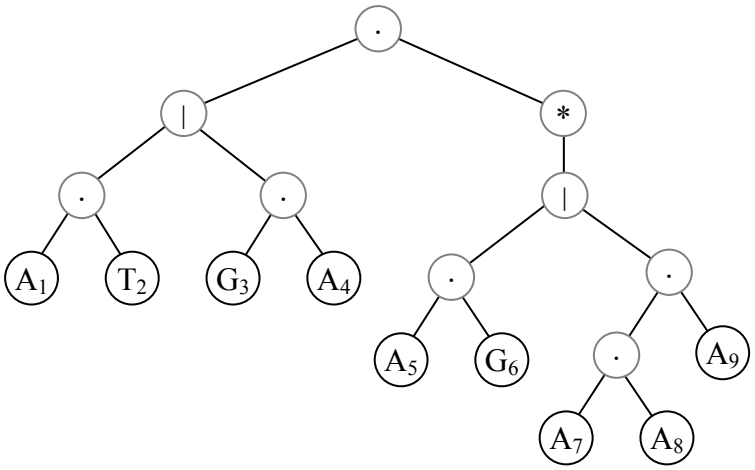


(A1 T2 | G3 A4) (A5 G6 | A7 A8 A9)*

Nagu eelpool öeldud, et saada Glushkov'i automaat algse regulaaravaldise jaoks, tuleb meil lihtsalt saadud automaadis indeksid ära kustutada. Selle sammu tegemisel muutub automaat enamasti mittedeterminеerituks.



Nüüd jääb veel üle näidata, kuidas konstrueeritakse automaadi tegemiseks vajalikud hulgad *First*, *Last*, *Follow* ja *Empty*. Eeldame, et etteantud regulaaravaldis on parsitud ning puukujul.



Vajalikud hulgad leitakse lõppjärjestusega parsimispuud läbides. Seega iga tipu korral rakendatakse algoritmi eelnevalt alluvatele ning alles seejärel tipule endale. Algoritmi jälgides tuleks silmas pidada, et erinevalt eelpool toodud notatsioonist on algoritmis hulk *Follow*(*x*) globaalne muutuja, kuhu kogutakse positsioonist *x* väljuvaid üleminekuid kogu algoritmi töö käigus. Hulk *Follow*(*x*) algväärtustatakse siis, kui algoritm jõuab puu tähega märgendatud leheni. Algoritmis vastab sellele juhtum $\alpha_x \in \overline{\Sigma}$. Edaspidi võidakse hulka *Follow*(*x*) elemente ainult lisada.

Regulaaravaldise RE jaoks kirjeldatud hulki leidev algoritm ise on järgmine

- kui RE on ε, siis
  - $First(RE) = \varnothing$
  - $Last(RE) = \varnothing$
  - $Empty_{RE} = \{\varepsilon\}$
- kui RE on $\alpha_x \in \overline{\Sigma}$, siis
  - $First(RE) = \{x\}$
  - $Last(RE) = \{x\}$
  - $Empty_{RE} = \varnothing$
  - $Follow(x) = \varnothing$
- kui RE on kujul (RE_1 | RE_2), siis
  - $First(RE) = First(RE_1) \cup First(RE_2)$
  - $Last(RE) = Last(RE_1) \cup Last(RE_2)$
  - $Empty_{RE} = Empty_{RE_1} \cup Empty_{RE_2}$
- kui RE on kujul ($RE_1 \cdot RE_2$), siis

- $First(RE) = First(RE_1) \cup (Empty_{RE_1} \cdot First(RE_2))$
- $Last(RE) = (Empty_{RE_2} \cdot Last(RE_1)) \cup Last(RE_2)$
- $Empty_{RE} = Empty_{RE_1} \cap Empty_{RE_2}$
- iga $x \in Last(RE_1)$ korral $Follow(x) = Follow(x) \cup First(RE_2)$

- kui RE on kujul (RE_1*), siis
  - $First(RE) = First(RE_1)$
  - $Last(RE) = Last(RE_1)$
  - $Empty_{RE} = \{\varepsilon\}$
  - iga $x \in Last(RE_1)$ korral $Follow(x) = Follow(x) \cup First(RE_1)$

Nagu me juba eelpool mainisime, siis Glushkov'i automaadi üheks heaks omaduseks on tema seisundite vähesus. Lisaks on Glushkov'i automaadi omadusteks, et

- puuduvad ε-üleminekud ning
- seisundisse y on võimalik saada vaid sümboli a_y sisselugemisel.

Kahte viimast omadust on mugav kasutada Glushkov'i automaadi töö simuleerimiseks.

3.3.2 Bitt-paralleelne Glushkov

NFA-de tööd simuleerides peame me meeles hoidma aktiivsete seisundite loetelu. Bitt-paralleelsed algoritmid kasutavad selleks bitivektorit, kus igale seisundile vastab üks bitt. Aktiivsete seisundite märkimiseks on nüüd põhimõtteliselt kaks võimalust, siin toodud algoritmis tähistab aktiivset seisundit bitt väärtusega[9] 1. Bitivektoreid hoitakse reeglina täisarvudena. Kuna tüüpiliselt on protsessori poolt toetatud täisarvu pikkus 32 baiti, siis teoreetiliselt on võimalik ühe protsessori sammu ajal teostada operatsioon kuni 32 seisundi jaoks. Sellest tuleneb ka bitt-paralleelsete algoritmide eelis nende tavaversioonidest analoogide ees. Kuid nagu me hiljem näeme, siis vähemalt selles peatükis käsitletud regulaaravaldise otsimise algoritmi puhul tuleb korraga töödeldavate seisundite arvule piir ette enne 32 seisundit.

[9] Teine võimalus tähistada aktiivseid seisundeid biti väärtusega 0. Kuigi esmapilgul tundub et nendel kahel lähenemisel vahet pole, siis osade bitt-paralleelsete algoritmide puhul võimaldab aktiivse seisundi 0-ga tähistamine algoritmi sammu paari masinkoodi käsu võrra lühemalt kirja panna.

Algoritmi koostamisel konstrueerisime me üleminekuteisendused nii, et igale seisundi ning tähe paarile seati vastavusse hulk uusi seisundeid, kuhu automaati simuleerides on võimalik jõuda. Kuna bitt-paralleelsuse korral tahame me töödelda mitut seisundit korraga, siis on meie jaoks kriitiline leida meetod, mis leiaks uued aktiivsed seisundid hoopis aktiivsete seisundite hulga ning tähe abil. Enne sellise funktsiooni juurde jõudmist tuleb meil sisse tuua tähistus (Q on automaadi seisundite hulk ning m regulaaravaldises olnud tähtede arv)

$$B_n[i,a] = \left|_{(s_i,a,s_j)\in\delta} 0^{|Q|-1-j}10^j\right. ,$$

mis määrab ära bitivektori seisunditest, mis on saavutatavad seisundist i tähe a abil.

Olgu meil nüüd D bitivektorina esitatud aktiivsete seisundite hulk ning a sisseloetud täht. Arvestades, et Glushkov'i automaadis kasutavad kõik ühte seisundisse suubuvad üleminekud ühte ja sedasama tähte, siis võime seisundite hulgaga töötava üleminekufunktsiooni avaldada järgmiselt

$$\delta(D,a) = T_d[D] \,\&\, B[a],$$

kus

$$B[a] = \left|_{i\in 0\ldots m} B_n[i,a]\right.$$

on bitivektor kõikidest seisunditest, kuhu on võimalik saada tähe a lugemisel, ning

$$T_d[D] = \left|_{i\in 0..m, D\&0^{m-i}10^i\neq 0^{m+1}, a\in\Sigma} B_n[i,a]\right.$$

on bitivektor seisunditest, mis on saavutatavad ükskõik millisest D poolt määratud seisundist ükskõik millise tähe abil.

Tabelite B ja T_d täitmise algoritm (Algoritm 1) on toodud allpool . Tabeli T_d täitmise lihtsustamiseks on kasutusel ka abitabel $A[i] = \left|_{a\in\Sigma} B[i,a]\right.$, mis on sisuliselt hulga $Follow(i)$ esitlus bitivektorina. Algoritmi ajaline keerukus on $O(2^m + m\cdot|\Sigma|)$.

BuildTran (N = (Q_n, Σ, I_n, F_n, B_n))
For i ∈ 0 ... m **Do** A[i] ← 0^{m+1}
For σ ∈ Σ **Do** B[σ] ← 0^{m+1}
For i ∈ 0 ... m, σ ∈ Σ **Do**
    A[i] ← A[i] | B_n[i, σ]
    B[σ] ← B[σ] | B_n[i, σ]
End of For
/* hulgad B ja A on valmis, nüüd konstrueerib hulga T_d */
T_d[0] ← 0^{m+1}
For i ∈ 0 ... m **Do**
    For j ∈ 0 ... 2^i-1 **Do**
        T_d[2^i + j] ← A[i] | T_d[j]
    End of For
End of For
Return (B, T_d)

Algoritm 1 Bitt-paralleelse Glushkov-i algväärtustamine

Regulaaravaldise otsimiseks vajalik algoritm (Algoritm 2) on toodud allpool.

BPGlushkov (N = (Q_n, Σ, I_n, F_n, B_n), T = $t_1t_2...t_n$)
/* Eeltöötlus */
For σ ∈ Σ **Do** B_n[0, σ] ← B_n[0, σ] | 0^m1
(B, T_d) ← BuildTrans(N)
/* Otsimine */
D ← 0^m1 /* algseisund aktiivne */
If D & F_n ≠ 0^{m+1} **Then** leiti vaste, mille lõpu asukoht on 0
For pos ∈ 1 ... n **Do**
    D ← T_d[D] & B[t_{pos}] | 0^m1
    If D & F_n ≠ 0^{m+1} **Then** leiti vaste, mille lõpu asukoht on pos
End of For

Algoritm 2 Bitt-paralleelne Glushkov

Olenevalt masinsõna suurusest võib tabel T_d osutuda mälus hoidmiseks liiga suureks. Selle vältimiseks võime me tabelit „horisontaalselt" tükeldada. Olgu meil masinsõna pikkus 32, kuid me otsustame, et me tahame suure tabeli T_d asendada nelja väiksema tabeliga T_{d0}, T_{d1}, T_{d2}, ja T_{d3}, mis vastavalt näitavad bitivektoreid seisunditest, mis on võimalik saavutada seisundite 0-7, 8-15, 16-23 ja 24-31 mingi alamhulga abil. Formaalsemalt tähistades[10]

[10] >> tähistab kahendarvu nihutamist etteantud arv bittide võrra paremale (sisuliselt kaotatakse etteantud arv parempoolseid bitte arvust)

$$T_{dx}[D] = \Big|_{i\in 0..7,(D>>8\cdot i)\&0^{m-i}10^i\neq 0^{m+1},a\in\Sigma} B_n[i,a]$$

Avaldise $T_d[D]$ saame me sellisel juhul asendada avaldisega[11]

$$T_d[D] = T_{d0}[D \,\&\, 1^8]T_{d1}[(D >> 8)\,\&\, 1^8]T_{d2}[(D >> 16)\,\&\, 1^8]T_{d3}[(D >> 24)\,\&\, 1^8]$$

Sellise „horisontaalse" tükeldamisega võidame me tabeli T_d suuruses, kuid peame andma lõivu teisenduse $T_d[D]$ arvutamise kiiruses.

3.3.3 Ligikaudne otsimine

Bitt-paralleelse algoritmi täiendamine ligikaudse otsimise toega on üsna lihtne. Selleks kasutatakse korraga mitut samasugust automaati, iga veataseme jaoks üks. Näiteks kui me lubame tulemuses kuni k viga, siis on meil paralleelselt töös k+1 identset automaati indeksitega 0 kuni k. Indeksiga 0 automaat näitab sellisel juhul seisundeid, kuhu on võimalik jõuda vigu tegemata, indeksiga 1 automaat seisundeid, kuhu on võimalik jõuda ülimalt ühe veaga jne. Algoritmi kirjelduses tähistame nende automaatide aktiivseid seisundeid hoidvaid bitivektoreid vastavalt R_0, R_1, ... R_k. Näitame nüüd, kuidas leida bitivektorite uued väärtused R_0', R_1', ... R_k' pärast tähe c sisse lugemist.

Samal veatasemel liikumiseks kasutatakse täpselt samasuguseid üleminekuid nagu vigadeta otsingu korral. Seega 0-nda taseme korral

$$R_0' = T_d[R_0] \,\&\, B[c] \mid 0^m1$$

ning kõigi ülejäänud tasemete jaoks

$$R_i' = T_d[R_i] \,\&\, B[c] \ldots$$

Ligikaudseks otsimiseks tuleb nüüd vaid lisada vigu tähistavad üleminekud veatasemete vahele. Tuletame meelde, et ligikaudse otsimise juures oli meil lubatud kolme liiki teisendusi – tähe kustutamine, tähe lisamine ning tähe asendamine. Vaatame neid kõiki ükshaaval.

---

[11] 1^8 tähistab kahendarvu, kus 8 madalaimat bitti on seatud 1-ks. $X\&1^8$ tähistab sellisel juhul kahendarvu, mis on saadud X-st vaid viimase 8 biti allesjätmisel (ülejäänud bitid nullitakse)

Kui me kustutaksime tekstist loetud tähe, siis automaadis jäävad kõik aktiivsed seisundid aktiivseteks

$$R_i' \models R_{i-1} \ldots$$

Kui me lisaksime teksti pärast (seda on kasulik teada algoritmi algväärtustamise juures) loetud tähte veel mingi tähe, siis automaadi jaoks tähendaks see loetud tähe töötlemist (R_{i-1}') ning pärast seda lisatud tähe (mis võis olla suvaline täht, sest me ise lisame selle sinna eesmärgiga tekst regulaaravaldisega sobima panna) võrra automaadis edasi liikumist

$$R_i' \models T_d[R_{i-1}'] \ldots$$

Kolmas vea tüüp on tähe asendamine tekstis. Sellisel juhul liigume me automaadis ühe sammu võrra edasi sõltumata loetud tähest (seega erinevalt samal veatasemel edasi liikumisest enam B[c] abil aktiivseid seisundeid filtreerida pole vaja)

$$R_i' \models T_d[R_{i-1}]$$

Arvestades, et $T_d[R_{i-1}] \mid T_d[R_{i-1}'] = T_d[R_{i-1} \mid R_{i-1}']$, saame algoritmiks

$$R_0' = T_d[R_0] \,\&\, B[c] \mid 0^m1$$

For $i \in 1 \ldots k$ **Do** $R_i' = (T_d[R_i] \,\&\, B[c]) \mid R_{i-1} \mid T_d[R_{i-1} \mid R_{i-1}']$

Algoritmi täielikuks kirjeldamiseks tuleb veel ära näidata bitivektorite R_0, R_1, ... R_k algväärtustamine ning vastete asukohtade leidmine.

Vigadeta otsimise korral oli jooksval positsioonil mõne vaste lõpp parajasti siis kui bitivektoris oli märgitud aktiivseks mõni lõppseisund ($D \,\&\, F_n \neq 0^{m+1}$). Ligikaudse otsimise korral on bitivektoreid k+1 tükki. Kindlasti saaks vaste tuvastada, kui kontrollida kõiki k+1 bitivektorit. Kuid osutub, et algoritmi sobivalt algväärtustades piisab vaste tuvastamiseks vaid viimase ehk k+1 veataseme bitivektori kontrollimisest. Selleks on vaja algoritmi täitmise ajal tagada olukord, kus suvalise i korral oleks bitivektoris R_{i-1} aktiivseks märgitud seisund aktiivne ka bitivektoris R_i. Märgime seda tingimust edaspidi $R_{i-1} \subset R_i$.

Induktsiooni abil on lihtne näidata, et selline sisaldumise tingimus kehtib enne tähe töötlemist, st $R_0 \subset R_1 \subset \ldots \subset R_k$, siis see tingimus jääb kehtima ka pärast tähe töötlemist, st $R_0' \subset R_1' \subset \ldots \subset R_k'$. Me ei too siin ära ranget tõestust, kuid illustratsiooniks näitame, et

$R_1' \subset R_2'$. Enne detailide juurde minemist paneme tähele, et teisenduse T_d omaduse tõttu kehtib järeldus $A \subset B \Rightarrow T[A] \subset T[B]$. Induktsiooni eelduse põhjal teame, et

- $R_0 \subset R_1 \subset R_2$
- $R_0' \subset R_1'$

Selle põhjal saame, et

- $R_1 \subset R_2 \Rightarrow T_d[R_1] \subset T_d[R_2] \Rightarrow T_d[R_1]$ & $B[c] \subset T_d[R_2]$ & $B[c]$
- $R_0 \subset R_1$
- $R_0 \subset R_1, R_0' \subset R_1' \Rightarrow R_0 \mid R_0' \subset R_1 \mid R_1' \Rightarrow T[R_0 \mid R_0'] \subset T[R_1 \mid R_1']$

Kuna

- $R_1' = (T_d[R_1]$ & $B[c]) \mid R_0 \mid T_d[R_0 \mid R_0']$ ja
- $R_2' = (T_d[R_2]$ & $B[c]) \mid R_1 \mid T_d[R_1 \mid R_1']$,

ning kõigi bitikaupa VÕI-ga seotud bitivektorite jaoks on meid huvitav sisaldumine näidatus, siis on ilmne, et $R_1' \subset R_2'$.

Väikeseks erandiks induktsiooni rakendamisele oleks R_0' ja R_1' võrdlemine, kuna R_0' arvutamisel seatakse bitivektoris kindlasti aktiivseks ka 0-is seisund (algoritmis „$\mid 0^m1$”). Õnneks, nagu me nägime vigasid mittelubava algoritmi korral, seatakse bitivektoris R_0 0-is bitt juba algväärtustamise ajal ning seega sisaldab algoritmis $R_i' = \ldots \mid R_{i-1} \mid \ldots$ varjatud kujul ka 0-ndale seisule vastava biti ülesse seadmist.

Vigadeta otsingu korral piisas algoritmi algväärtustamiseks automaadi 0-nda seisu aktiivseks muutmisest. Vigadega otsingu korral tuleb aga leida sobivad algseisundid ka teiste veatasemete jaoks. Peamine põhjus, miks teistel veatasemetel ei piisa vaid 0-nda seisundi aktiveerimisest, on see, et meie algoritmis toimub teksti tähe lisamine pärast loetud sümboli töötlemist. Seega algoritmi algväärtustamisega on meil vaja saavutada olukord, mis vastaks teksti algusesse (st enne esimest sümbolit) nõutud arvu tähtede lisamisele. Algoritmis vastaks siis sellele

$$R_i = T_d[R_{i-1}] \ldots$$

Kuid vaste lõpu leidmise kirjeldamisel saime me algoritmi algväärtustamise jaoks veel kaks tingimust – kõrgemal veatasemel pidid olema aktiivsed kõik need seisundid, mis madalamal veatasemel, ning 0-is seisund pidi olema aktiivne kõigil veatasemetel. Tegelikult esimese tingimuse täitmine tagab meile automaatselt ka teise tingimuse täitmise. Toome siinkohal ära vaid täpse ja ligikaudse otsingu erinevused

/* Otsimine */
$R_0 \leftarrow 0^m1$ /* algseisund aktiivne */
For i ∈ 1 … k **Do** $R_i = R_{i-1} \mid T_d[R_{i-1}]$
If D & $F_n \neq 0^{m+1}$ **Then** leiti vaste, mille lõpu asukoht on 0
For pos ∈ 1 ... n **Do**
    $R_0' = T_d[R_0]$ & $B[t_{pos}] \mid 0^m1$
    For i ∈ 1 … k **Do** $R_i' = (T_d[R_i]$ & $B[t_{pos}]) \mid R_{i-1} \mid T_d[R_{i-1} \mid R_{i-1}']$
    If R_k' & $F_n \neq 0^{m+1}$ **Then** leiti vaste, mille lõpu asukoht on pos
    For i ∈ 0 … k **Do** $R_i = R_i'$ /* järgmise sümboli töötlemiseks */
End of For

3.3.4 Vigadeta piirkonnad

Ligikaudset algoritmi kirjeldades vaatlesime me lubatud teisendusi kui operatsioone tekstiga. Kuna vigadeta piirkonnad regulaaravalises näitavad kohti, kus regulaaravalised ei tohi tekstiga sobitamiseks muuta, siis tuleks teisendused sõnastada ümber operatsioonideks regulaaravalisega. Et Glushkov'i automaadis vastas igale seisundile konkreetne täht regulaaravalises, siis võib teisendusi vaadata ka kui operatsioone automaadiga.

Ilmselt jääb algoritmi see osa, mis puudutab samal veatasemel liikumist, samaks. Üle tuleks aga vaadata tekstil tehtavad operatsioonid.

Vigadeta otsingu korral oli esimene operatsioon tähe kustutamine tekstist. Automaadis vastaks sellele tähe lisamine. Loomulikult on „automaati tähe lisamine" operatsiooni olemuse selgitamiseks kasutusele võetud mõiste ning ei tähenda automaadi (seisundite, üleminekute jms) tegelikku muutmist.

Ilmselt ei ole tähe lisamine lubatud automaadis vigadeta piirkonna sisse, see tähendab siis kahe samasse vigadeta piirkonda kuuluva tähe vahele. Toome näite. Regulaaravaldises `<AB><CD>` on ainsad kohad, kuhu tähe automaati lisamine on keelatud, `A` ja `C` järel. Samas `B` järele on tähe lisamine lubatud, sest see muudatus jääks vigadeta piirkonnast välja.

Seega lihtsate vigadeta piirkondadega saaksime seega hakkama – lisamine on lubatud vigadeta piirkondadest välja jäävate tähtede järel ning ka vigadeta piirkondade viimaste tähtede (lõppseisundite) järele.

Vaatame aga järgmist kahte näidet, olgu meil regulaaravaldis `A<BC+>B`. Kui seda sobitada tekstidele `ABCBCXB` ja `ABCXBCB` siis esimene tekst sobiks vasteks, teine aga mitte. Seega olgugi et `C` on meil vigadeta piirkonna viimane sümbol, siis ühel korral ei ole tema järele automaati tähe lisamine lubatud, teisel korral aga on. Kui vaste leidmist automaadi tasemel vaadata, siis esimese teksti puhul peab automaat minema pärast `C`-d esimese, see tähendab samas vigadeta piirkonnas oleva, `B` juurde, samas teise teksti puhul tuleb `X` lisada pärast vigadeta piirkonnast väljumist. Sellise vigadeta piirkondade lõppseisundite kahese olemusega tegelemiseks tuleks nendest teha ilmselt kaks koopiat. Esimeses koopia järel me tähe automaati lisamist ei luba ning temast väljuvad ainult samas vigadeta piirkonnas lõppevad üleminekud. Teise koopia järel aga lubame tähe lisamist automaati ning temast väljuvad ainult temaga samasse vigadeta piirkonda mitte-suunduvad teisendused.

Sellise seisundi kaheks jaotamise juures on kasulik märgata kolme asja. Esiteks on selge, et kui me võtame mingi kaheks jaotatud seisundi ning vaatame tema suhtes kõiki automaadi seisundeid (muuhulgas siis ka teda ennast), siis kõik nad kas kuuluvad valitud seisundiga samasse vigadeta piirkonda või siis ei kuulu sinna. See aga tähendab seda, et kõik algses automaadis olnud teisendused jagatakse kahe koopia vahel ära ning ühtegi teisendust ära ei kaotata. Teiseks, automaadi lihtsuse huvides võib seisundi esimese koopia, mille järele automaati tähe lisamist ei lubata, jaoks jätta alles ikkagi kõik teisendused, ka need, mis suunduvad samasse vigadeta piirkonda. Tähti automaati lisamata võime me ju tegelikult kõikidesse algses automaadis lubatud seisunditesse minna. Kolmandaks ka üks tehnilisem märkus. Seisundeid kaheks koopiaks jagades tuleb dubleerida ka kõik nendesse tulevad teisendused.

Märgime siinjuures ära, et vigadeta piirkondade lõppseisundid on tõesti ainsad seisundid, mida dubleerida tuleb. On ju vigadeta piirkondadest välja jäävate seisundite järele tähe lisamine alati lubatud ning vigadeta piirkondade mitte-lõppseisundite järel saab tulla vaid samasse vigadeta piirkonda kuuluv seisund ning seega ei ole tähe lisamine nende järele kunagi lubatud.

Seega tähe tekstist kustutamiseks saame me vigadeta piirkondade puhul

$R_i' \models R_{i-1}$ & I …

kus I on bitivektor märkimaks seisundeid, mille järel võib automaati tähe lisada. Ning bitivektoris I on ära märgitud vigadeta piirkondadest välja jäävad seisundid ning vigadeta piirkondade lõppseisundite teised, vigadeta piirkonnast välja viivad koopiad.

Ülejäänud kahe teisendusega läheb juba lihtsamalt. Teksti tähe lisamisele vastab automaadi puhul tähe automaadist kustutamine või siis täpsemini öeldes vahele jätmine. Vahele ei saa aga seisundit jätta parajasti siis kui ta kuulub vigadeta piirkonda. Seega tähe automaadist kustutamise realiseerimiseks oleks meil teisendusega T_d sarnast teisendust. Ainuke vahe oleks see, et uus seisund ei sisaldaks üleminekuid, mis suunduvad vigadeta piirkonnas olevatele seisunditele (sest ülemineku lõppseisund näitab tähte, mis automaadis vahele jäetakse). Tähistame seda uut üleminekut T_e. Seega teksti tähe lisamise jaoks saame

$R_i' \models T_e[R_{i-1}']$ …

Kolmas ja viimane operatsioon on tähe asendamine tekstis, mis automaadi korral vastaks tähe asendamisele automaadis. Analoogselt automaadist tähe vahele jätmisele ei tohi ka siin muuta automaadi tähte parajasti siis kui ta kuulub vigadeta piirkonda. Kuna ka siin tähistab muudetavat tähte just ülemineku lõppseisnud, siis saame

$R_i' \models T_e[R_{i-1}]$

Uut teisendust T_e läheb vaja veel ka algoritmi algväärtustamisel, sest mäletatavasti oli algväärtustamine sisuliselt teksti algusesse tähtede lisamine. Seega vigadeta piirkondi lubava algoritmi väärtustamiseks

$R_0 \leftarrow 0^m1$ /* algseisund aktiivne */

For $i \in 1 \ldots k$ **Do** $R_i = R_{i-1} \mid T_e[R_{i-1}]$

4 Algoritmide realisatsioon

Ligikaudne regulaaravaldiste otsimine sai realiseeritud kui üks mustrite liik plaanitavast SALLY-nimelisest C++-i tekstialgoritmide teegist. Selles peatükis toome ära eelmises peatükis kirjeldatud bitt-paralleelsel Glushkov-i automaadil põhineva algoritm realisatsiooni võrdluse mõningate teiste C++-is kirjutatud regulaaravaldiste mootoritega.

4.1 Võrdluses osalenud regulaaravaldiste mootorid

Vigadeta otsingus sai realiseeritud regulaaravaldise mootorit võrrelda kolme teise mootoriga

- Boost.Regex (http://www.boost.org/libs/regex/doc/index.html, versioon 1.33.1). Tegemist on ühe kiirema[12] ja populaarsema C++-i jaoks kasutatava regulaaravaldiste mootoriga.
- GRETA (http://research.microsoft.com/projects/greta/regex_perf.html, versioon 2.6.4). GRETA on mõeldud just lühemates tekstides kiirete otsingute tegemiseks. Seetõttu on GRETA-s regulaaravaldise eeltöötlemine paljude teiste regulaaravaldiste mootoritega võrreldes mõnevõrra vähemmahukas.
- TRE (http://laurikari.net/tre/, versioon 0.7.5). Selle teegi suureks plussiks on võimalus otsida ka ligikaudselt.

Esimese kahe teegi populaarsust tõestab asjaolu, et mõlemad on saadavad Windowsi platvormil laialt levinud tasuta C++-i arenduskeskkonna Dev-C++ lisa-paketina.

Kahjuks jäi võrdlusest välja oma kiirusega tuntud AGREP (http://www.tgries.de/agrep/). Tegemist on Linux-i keskkonnas paljukasutatava utiliidi GREP paindlikuma, ka ligikaudset otsingut võimaldava, versiooniga. Paraku on aga AGREP-i koodis kasutatavad meetodid mõeldud just tekstis vaste olemasolu kontrolliks, mitte kõikide tekstis esinevate vastete leidmiseks, ning seetõttu teda antud testide puhul kasutada ei saanud.

[12] http://cvs.sourceforge.net/viewcvs.py/*checkout*/boost/boost/libs/regex/doc/gcc-performance.html

4.2 Testid

Täpse otsingu võrdlemiseks sai tehtud viis testi. Otsitavad regulaaravaldised on bioinformaatika spetsiifilised ning võetud veebilehelt http://ep.ebi.ac.uk/GPCR/. Tekstid on kunstlikult koostatud ning nad kõik koosnevad 10-st miljonist sümbolist. Varieeritud on vastete arvu. Testide lühikirjeldus on toodud allolevas tabelis.

Nr	Regulaaravaldis	Boost-i, GRETA ja TRE poolt leitud vastete arv	SALLY leitud vastete arv
1.	[ILV]...SG.{0,10}R	89 840	209 217
2.	V...[RK]....R	517 552	695 527
3.	R[FWY].[AGS][ILV].{0,7}A[ILV]	673 037	956 724
4.	T..[RK].{0,10}S..T|A.{3,6}V[ILV][RK]|P..[AGS]T.{0,10}S|[AGS][ILV][ILV][RK].{2,10}S	456 586	796 149
5.	[ILV].....A.T|S...L.{1,11}Y|S...L.{2,9}TL|[RK]F....K	601 033	1 049 569

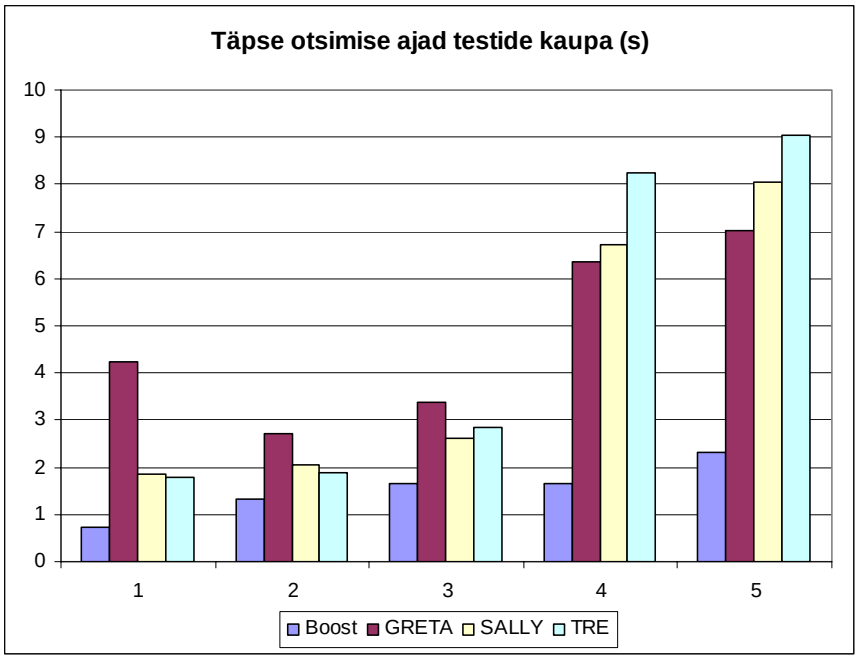
Nagu juba ülesandepüstituses mainitud sai, siis vastete arvud erinevad just seetõttu, et SALLY leiab ka kattuvad vasted, Boost, GRETA ja TRE aga mitte.

Kui 1. kuni 3. testi regulaaravaldised on piisavalt lühikesed, et nendega töötamisel jätkub ühest masinasõnast, siis 4. ja 5. testi regulaaravaldised on saadud mitme regulaaravaldise „|"-ga kokku liitmisel ning nendega töötamiseks läheb vaja juba kahte masinasõna.

Ligikaudse otsingu võrdlemiseks sai kasutatud käesoleva töö ühte pisut muudetud vaheversiooni suurusega umbes 66 tuhat märki, kust otsiti 0 kuni 4 veaga sõna „regulaaravaldis". Muudatused seisnesid üksikutesse vastetesse trükivigade lisamises.

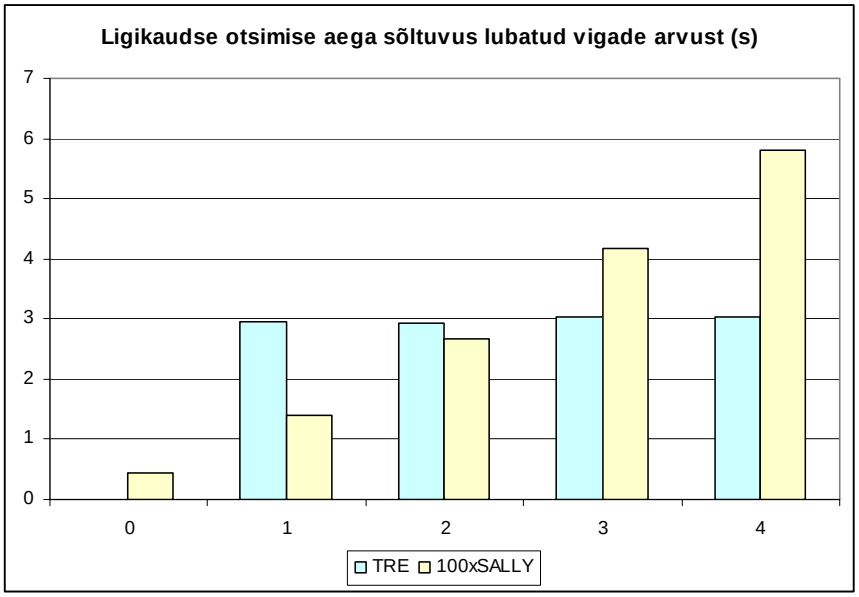
4.3 Tulemused

Täpse otsingu võrdlemiseks tehtud testide tulemused on toodud järgneval joonisel



Nagu näha on Boost-i regulaaravaldiste teek teistest märgatavalt kiirem, eriti väikese vastete arvu ning pikemate regulaaravaldiste korral. GRETA, SALLY ja TRE töökiirused on enam-vähem võrdsed. Kusjuures SALLY ja TRE on pisut kiiremad lühikeste regulaaravaldiste ning GRETA pikemate regulaaravaldiste korral. Ja nagu arvata võis mõjutab automaadi hoidmiseks vajalik masinasõnade arv tugevasti SALLY töökiirust.

Ligikaudse otsingu tulemusi iseloomustab järgmine joonis.



Vigadeta otsingu korral töötasid TRE ja SALLY antud testi puhul mõlemad alla 0,01 sekundi[13]. Kuid kuna vigade lubamine muutis TRE oluliselt aeglasemaks, siis on SALLY-t testitud 100 korda suurema faili peal.

Mõlemad teegid kasutavad oma töös lõplikku automaati. Kuid SALLY-s realiseeritud algoritm kasutab bitt-paralleelsust, mistõttu vigade arvu suurendamiseks tuleb ka automaat suuremaks ehitada. Seetõttu on SALLY töökiirus lineaarselt seotud lubatud vigade arvuga. TRE puhul emuleeritakse automaadi täitmist bitt-paralleelsust kasutamata. See võimaldab iga seisu juures hoida rohkem infot ning seetõttu saab vigu lubavat otsingut teha ka automaati muutmata. Küll aga nõuab ligikaudne otsing rohkemate automaadi seisundite töötlemist. Seetõttu muudab antud testi korral vigade lubamine TRE üle kahe suurusjärgu aeglasemaks, kuid edasine vigade lubamine algoritmi kiirust enam oluliselt ei mõjuta.

Võrreldes ülejäänud kolme teegiga on SALLY miinuseks suur regulaaravaldise parsimiseks ning automaadi ehitamiseks kuluv aeg, mis oli testides toodu regulaaravaldiste korral umbes 0,2 sekundit (teiste teekide korral oli see alla kasutatud mõõtmisviisi täpsuse, st vähem kui 0,01 sekundit). Seetõttu on praegusel kujul mõtet SALLY-t kasutada vaid pikematest tekstidest otsimisel. Kuna täpse otsingu korral olid otsitavad tekstid suhteliselt pikad, siis nendes testides mõõdeti otsinguaega koos regulaaravaldise ettevalmistamiseks kuluva ajaga, kuid ligikaudse otsingu korral jäeti lühikese teksti tõttu SALLY algoritmi kiiruslike omaduste paremaks välja toomiseks ettevalmistusaeg mõlema teegi korral otsinguajast välja.

[13] Kasutatud mõõtmisviisi korral oli 0,1 sekundit vähim mõõdetav aeg

5 Kokkuvõte

Alates sellest ajast, kui Kleene regulaaravaldised 50-ndatel kasutusele võttis [WiKleA], on need leidnud teksti töötlemisel laialdast kasutust. Üheks alaliigiks on ka mustrite otsimine bioinformaatikas. Tänaseks päevaks on lisatud Kleene poolt väljapakutud regulaaravaldistele hulgaliselt lisavõimalusi, kuid bioinformaatika seisukohalt pole nendel võimalustel väga suurt praktilist väärtust. Bioinformaatika ülesannete spetsiifilisust arvesse võttes on märksa huvitavam hoopis lihtsate regulaaravaldiste ligikaudse otsimise võimalus.

Käesoleva semestritöö raames valmis Glushkov-i automaadi bitt-paralleelsel simuleerimisel põhinev C++-is kirjutatud regulaaravaldiste teek. Realiseeritud algoritm sisaldab kõiki regulaaravaldise definitsioonis toodud võimalusi ning lubab ka Levenshteini kaugusel põhinevat ligikaudset otsimist. Omalt poolt sai algoritmi lisatud vigadeta piirkondade tugi. Kuna olemasolevatel teekidel on funktsionaalsus realiseeritud teegist mõnevõrra erinev, siis päris täpset võrdlust teistega teha ei saa. Kuid üldiselt võib ütelda, et vigadeta otsingu korral jääb kirjutatud teek kiiruse poolest parematele olemasolevatele teekidele mõnevõrra alla, aga ligikaudse otsimise jaoks on tegemist üsnagi arvestatava realisatsiooniga.

Ligikaudne regulaaravaldiste otsimine on vaid üks mustrite liik plaanitavast SALLY-nimelisest C++-i tekstialgoritmide teegist. Edaspidise tööna võiks vaadelda realiseeritud koodi optimeerimist, regulaaravaldiste ligikaudsel otsimisel põhinevaid mustrite kaevandamise meetodeid ning SALLY täiendamist uute võimalustega.

Abstract

Approximate Search of Regular Expressions Using Bit-Parallel algorithms

Term paper

Kristo Tammeoja

Since Kleene introduced regular expressions in 1950's, they have become a powerful tool for text-based practical tasks, like defining lexical constructs of programming languages, text searches, and others. Nowadays there are many practical extensions to the abstract definition of regular expression proposed by Kleene. In practical areas like bioinformatics those new features may not be so widely used since the concept of formal language may not be so clearly known. Hence, in bioinformatics, often finding simple regular expressions approximately is more interesting.

In this term paper a C++ regular expression library based on bit-parallel simulation of Glushkov's automaton has been implemented. The library supports all the possibilities introduced in the definition of regular expressions plus approximate search using Levenshtein's distance as a measure. Besides implementing the basic version of the algorithm, support for error-free regions was added.

We compared the new implementation with other matching libraries. Since its functionality differs slightly from popular C++ libraries, we can not compare them exactly. For exact matching the new library is 1,5-3 times slower than the existing ones,. For approximate matching the comparison was harder. We found one other implementation TRE, and our library performed 100-fold faster than that.

The approximate search of regular expressions is just one type of patterns for a planned text-algorithms library SALLY. The other algorithms deal with probabilistic position weight matrix matching, for example. Future work will include optimizing of the current code, studying the methods for mining patterns using approximate regular expressions and adding new features to the SALLY.

Kasutatud kirjandus

[Friedl] Jeffrey Friedl, 2002. Mastering Regular Expressions, 2nd Edition

[Laurikari] http://laurikari.net/ville/regex-submatch.pdf

[MOG98] Eugene W. Myers, Paulo Oliva, Katia Guimares, 1998. Reporting Exact and Approximate Regular Expression Matches (www.dcs.qmul.ac.uk/~pbo/papers/paper001.pdf)

[Navarro] Gonzalo Navarro, Mathieu Raffinot, 2002. Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences

[regex.inf] http://www.regular-expressions.info/

[WiKleA] http://en.wikipedia.org/wiki/Kleene_algebra