Markus Lahtonen

Regexp-hakujen toteuttaminen äärellisillä automaateilla

Tietotekniikan kandidaatintutkielma

29. huhtikuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Markus Lahtonen

Yhteystiedot: markus.h.lahtonen@student.jyu.fi

Ohjaaja: Sanna Mönkölä

Työn nimi: Regexp-hakujen toteuttaminen äärellisillä automaateilla

Title in English: Implementing regular expression matching using finite automata

Työ: Kandidaatintutkielma

Sivumäärä: 21+0

Tiivistelmä: Tässä kirjallisuuskatsauksessa tarkastellaan, miten säännöllisiä lausekkeita käyttävät merkkijonohaut voidaan toteuttaa äärellisten automaattien avulla. Sen lisäksi muutamia kirjallisuudesta löytyviä toteutustapoja tarkastellaan yksityiskohtaisemmin.

Avainsanat: regexp, säännöllinen lauseke, äärellinen automaatti

Abstract: This literature review explains how regular expression matching can be implemented using finite-state automata. Moreover, some implementations found in the literature will be reviewed in greater detail.

Keywords: regexp, regular expression, finite-state automaton

Kuviot

Sisältö

1 Johdanto

Säännölliset lausekkeet (engl. *regular expressions*) kuvaavat joukon merkkijonoja, eli säännöllisen kielen, joka voidaan tunnistaa äärellisen automaatin avulla (Mitkov 2003). Äärellisillä automaateilla on useita käytännön sovelluksia, mutta tässä tutkielmassa keskitytään niiden käyttöön merkkijonohauissa.

Useimmat suosituimmista ohjelmointikielistä tarjoavat mahdollisuuden säännöllisiä lausekkeita käyttäviin merkkijonohakuihin (Berglund ym. 2014). Näistä suurin osa käyttää toteutuksessaan ns. rekursiivista peruutusta (engl. *recursive backtracking*) (Cox 2007). Tämä mahdollistaa hakujen ilmaisuvoimaa lisäävien ominaisuuksien, kuten takaisinviittausten, käytön. Näitä ominaisuuksia sisältävät *laajennetut säännölliset lausekkeet* eivät (välttämättä) ole tunnistettavissa pelkästään äärellisten automaattien avulla (Hopcroft ym. 2001).

Sekaannusten välttämiseksi laajennetuista säännöllisistä lausekkeista käytetään tässä tekstissä jatkossa termiä *regexp*, joka kattaa sekä määritelmän mukaiset säännölliset lausekkeet että niiden laajennokset. Sen lisäksi termillä *regexp-haku* viitataan tässä tekstissä regexp-hakukoneisiin ja ohjelmointikielten regexp-kirjastoihin — ei yksittäisen haun suorittamiseen.

Tämän kirjallisuuskatsauksen tarkoituksena on tarkastella äärellisten automaattien käyttöä regexp-hakujen toteutuksessa. Mahdollisten toteutustapojen ominaisuuksia vertaillaan keskenään. Lisäksi tutkielmassa tutustutaan tarkemmin muutaman erityyppisen regexp-toteutuksen toimintaan.

Tutkielman toisessa luvussa kerrotaan äärellisten automaattien ja säännöllisten lausekkeiden toiminnasta yleisellä tasolla. Kolmannessa luvussa tarkastellaan mahdollisia regexp-hakujen toteutusperiaatteita ja vertaillaan niiden vahvuuksia ja heikkouksia. Neljäs luku sisältää muutamien toteutustapojen yksityiskohtaisemman tarkastelun. Viidennessä luvussa on tutkielman yhteenveto.

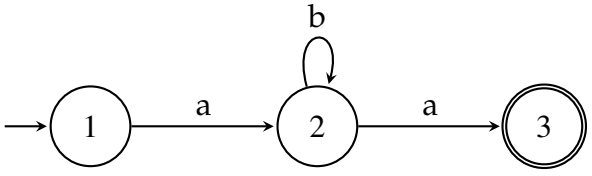
2 Äärelliset automaatit ja säännölliset lausekkeet

Tässä luvussa kerrotaan yleisluontoisesti äärellisten automaattien ja säännöllisten lausekkeiden ominaisuuksista. Formaalit matemaattiset määritelmät sivuutetaan, sillä ne eivät ole tämän tutkielman kannalta oleellisia.

2.1 Äärelliset automaatit

Yksinkertaistettu äärellisen automaatin (engl. *finite-state automaton*) määritelmä voi olla seuraavanlainen: Automaatti on laite, joka koostuu tiloista ja niiden välisistä siirtymistä. Automaatti lukee sille annettua syötettä ja vaihtaa tilaansa jokaisen syötemerkin perusteella. Syötteen päätyttyä se joko hyväksytään tai hylätään, riippuen siitä, onko automaatin silloinen tila hyväksyvä vai hylkäävä. Automaatti on äärellinen, jos sillä on äärellinen määrä tiloja ja niiden välisiä siirtymiä. Jos lisäksi jokaisen luetun syötemerkin kohdalla automaatin seuraava tilasiirtymä on yksikäsitteinen, kyseessä on deterministinen äärellinen automaatti (engl. *deterministic finite automaton*) eli DFA.

Automaatteja voidaan havainnollistaa tilakaavioiden avulla. Aloitustila on merkitty tyhjällä nuolella. Muut nuolet kuvaavat automaatin tilasiirtymiä annetulla syötemerkillä. Hylkäävät tilat esitetään ympyröimällä ne kerran, ja hyväksyvät tilat on ympyröity kahdesti.

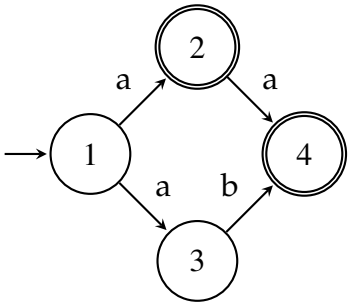


Kuvio 1. Esimerkki DFA:sta, joka lukee syötemerkkejä `a` ja `b`.

Kuvion 1 automaatti hyväksyy kaikki seuraavanlaiset merkkijonot: ensimmäinen merkki on `a`, jota seuraa nolla tai useampi merkki `b`, jota seuraa merkki `a`. Hyväksyttyjä merkkijonoja ovat siis `aa`, `aba`, `abba`, `abbba` ja niin edelleen. Merkkijonoja,

jotka automaatti hyväksyy, sanotaan sen tunnistamaksi kieleksi — äärellisen automaatin tapauksessa on kyse ns. säännöllisestä kielestä (Feldman 1993).

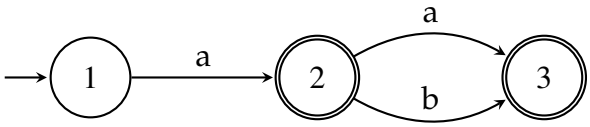
DFA:ssa jokaiselle syötemerkille on jokaisesta tilasta tasan yksi siirtymä (Feldman 1993). Kuvioihin selkeyden vuoksi piirtämättä jätetyt siirtymät johtavat suoraan hylkäykseen — esimerkiksi syötemerkin `b` lukeminen kuvion 1 automaatin ollessa tilassa 1 johtaa hylkäykseen. Äärellinen automaatti voi kuitenkin sisältää useampia siirtymiä samasta tilasta samalla syötemerkillä — tällöin kyseessä on epädeterministinen äärellinen automaatti (engl. *non-deterministic finite automaton*) eli NFA. Tämän lisäksi NFA voi sisältää siirtymiä, joihin ei tarvita syötettä (Feldman 1993).



Kuvio 2. Esimerkki NFA:sta, joka lukee syötemerkkejä `a` ja `b`.

Kuvion 2 automaatti hyväksyy ainoastaan merkkijonot `a`, `aa` ja `ab`. Sen tilasta 1 on syötemerkillä `a` kaksi mahdollista siirtymää, joten automaatti on epädeterministinen. NFA hyväksyy kaikki merkkijonot, jotka lukemalla voidaan päätyä hyväksyvään tilaan vähintään yhdellä tavalla (Hopcroft ym. 2001).

DFA:t ja NFA:t ovat ilmaisuvoimaltaan samanarvoisia (Feldman 1993). Toisin sanoen jokainen NFA on mahdollista muuntaa saman kielen tunnistavaksi DFA:ksi, ja koska NFA:n määritelmä sisältyy DFA:n määritelmään, jokainen DFA on aina myös NFA.



Kuvio 3. DFA:ksi muunnettu kuvion 2 NFA, joka lukee syötemerkkejä `a` ja `b`.

Kuvion 3 DFA tunnistaa saman kielen kuin kuvion 2 NFA. Toisin kuin NFA:n tapauksessa, DFA ei joudu minkään syötemerkin kohdalla "arvaamaan", mihin tilaan sen pitää siirtyä (Feldman 1993).

2.2 Säännölliset lausekkeet

Säännöllinen kieli voidaan kuvata sekä äärellisellä automaatilla, että säännöllisellä lausekkeella (Hopcroft ym. 2001). Säännölliset lausekkeet ovat ilmaisuvoimaltaan samanarvoisia äärellisten automaattien kanssa — jokaista säännöllistä lauseketta vastaa ainakin yksi äärellinen automaatti, ja päinvastoin (Feldman 1993).

Hopcroft ym. (2001) määrittelevät säännöllisille lausekkeille seuraavat kolme laskuoperaatiota:

Olkoot r_1 ja r_2 säännöllisiä lausekkeita. Näille ovat voimassa

1. Toisto: r_1* kuvaa merkkijonot, jotka saadaan toistamalla r_1:n kuvaamia merkkijonoja nolla tai useampi kertaa
2. Yhteen liittäminen: $r_1 r_2$ kuvaa merkkijonot, jotka saadaan yhdistämällä yhden r_1:n kuvaaman merkkijonon perään yksi r_2:n kuvaama merkkijono
3. Kielten yhdiste: $r_1 \mid r_2$ kuvaa merkkijonot, jotka kuuluvat r_1:n tai r_2:n kuvaamiin merkkijonoihin.

Laskuoperaatiot on listattu järjestyksessä vahvimmasta heikoimpaan — järjestystä on mahdollista muuttaa suluttamalla lausekkeen osia (Hopcroft ym. 2001). Näitä operaatioita käyttämällä saadaan kuvion 1 automaattia vastaava säännöllinen lauseke `ab*a`. Kuvioiden 2 ja 3 automaatteja vastaa säännöllinen lauseke `a|aa|ab`.

Käytännön sovelluksissa säännölliset lausekkeet sisältävät paljon laajemman notaation kuin kolme edellä määriteltyä laskuoperaatiota — tarkoituksena on saada säännöllisistä lausekkeista lyhyempiä ja selkeämpiä, tai laajentaa niiden ominaisuuksia (Hopcroft ym. 2001). Esimerkiksi UNIXin notaatio sisältää mm. operaattorit `?` (nolla tai yksi esiintymää), `+` (yksi tai useampi esiintymä) ja merkinnän `[A-Z]` (isot kirjaimet A:sta Z:aan) (Hopcroft ym. 2001). Lisättyjä ominaisuuksia ovat esimerkiksi

takaisinviittaukset, merkkijonon selaaminen ja osittainen jäsentäminen. Edellä mainittuja operaattoreita ja ominaisuuksia sisältävistä lausekkeista käytetään nimitystä *laajennetut säännölliset lausekkeet*, tai yleisemmin *regexp*.

Regexpien avulla kuvioita 2 ja 3 vastaava säännöllinen lauseke `a|aa|ab` saadaan muotoon `a(a|b)?`. Monimutkaisemmissa tapauksissa saavutettu hyöty on helposti huomattavissa: esimerkiksi regexp `([01]?[0-9]|2[0-3]):[0-5][0-9]`, joka tunnistaa kellonaikoja — mm. merkkijonot `17:32` ja `9:45` hyväksytään, mutta `25:86` hylätään — kirjoitettaisiin puhtaiden säännöllisten lausekkeiden notaatiolla muodossa `((()|0|1)(0|1|2|3|4|5|6|7|8|9)|2(0|1|2|3)):(0|1|2|3|4|5)(0|1|2|3|4|5|6|7|8|9)`.

On huomattava, että regexpit eivät kaikissa tapauksissa kuvaa säännöllisiä kieliä (Hopcroft ym. 2001). Tällöin regexpistä on mahdotonta muodostaa äärellistä automaattia, joka tunnistaisi regexpin kuvaaman kielen. Esimerkiksi takaisinviittauksen sisältävä regexp `(.*)\1` kuvaa sellaiset merkkijonot, jotka voidaan jakaa kahteen identtiseen osaan — mm. merkkijonot `aa`, `1010` ja `poniponi` kuuluvat regexpin kuvaamaan kieleen. Tällainen kieli ei kuitenkaan ole säännöllinen. Tämä voidaan todistaa ns. *säännöllisten kielten pumppauslemman* avulla (Hopcroft ym. 2001).

Käytännön regexp-haut kuitenkin harvoin perustuvat pelkästään äärellisiin automaatteihin. Laajentamalla automaattien ominaisuuksia voidaan tunnistaa myös muita kuin säännöllisiä kieliä.

3 Regexp-hakujen toteuttaminen

Tässä luvussa tarkastellaan mahdollisia menetelmiä toteuttaa regexp-haku käyttäen äärellisiä automaatteja. Lisäksi vertaillaan eri menetelmien mahdollistamia toimintoja ja menetelmien suorituskykyä.

3.1 Toteutusmenetelmät

Regexp-haku voidaan toteuttaa äärellisiä automaatteja käyttäen kahdella eri tavalla: DFA:han perustuen tai NFA:han perustuen (Berglund ym. 2014). Nämä kaksi tapaa ovat ominaisuuksiltaan hyvin erilaisia.

DFA:han perustuvat toteutukset muuntavat regexpin ensin NFA:ksi, joka puolestaan muunnetaan DFA:ksi (Berglund ym. 2014). Tällaiset toteutukset ovat erittäin nopeita, sillä ne lukevat haun aikana jokaisen syötemerkin enintään yhden kerran (Microsoft 2016).

NFA:han perustuvat toteutukset muuntavat regexpin NFA:ksi ja haku tapahtuu syötteen ohjaamana syvyyshakuna NFA:sta. NFA:han perustuvia regexp-hakuja kutsutaan niiden yleisen toimintaperiaatteen vuoksi usein rekursiiviseen peruutukseen perustuviksi. (Berglund ym. 2014.) Se tarkoittaa, että kaikki vaihtoehdot käydään läpi rekursiivisesti yksi kerrallaan.

Rekursiivista peruutusta voidaan soveltaa esimerkiksi luvun 2 kuvion 2 NFA:han. Sen tilasta 1 on useita mahdollisia siirtymiä syötemerkillä `a`, joten eri reittejä kokeillaan yksi kerrallaan. Mikäli haku ei ensimmäisellä reitillä tuota tulosta, peruutetaan takaisin tilaan 1 ja kokeillaan muita mahdollisia reittejä.

DFA-toteutusten käyttö on NFA-toteutuksiin verrattuna huomattavasti harvinaisempaa (Cox 2007). Laurikarin (2001) mukaan NFA-toteutusten etuja ovat toteutuksen yksinkertaisuus ja ominaisuuksien lisäämisen helppous. Lisäksi NFA:han perustuva regexp-haku mahdollistaa useita sellaisia ominaisuuksia, joihin DFA:han perustuvat toteutukset eivät kykene (Berglund ym. 2014).

3.2 Hakuominaisuudet

Osa regexpien käyttämistä ominaisuuksista on sellaisia, joita pelkkä äärellinen automaatti ei riitä käsittelemään. Rekursiiviseen peruutukseen perustuviin toteutuksiin sisältyy yleensä ainakin pino, johon säilötään tieto mahdollisista peruutuspisteistä (Laurikari 2001). Pinon avulla peruutuspisteeseen voidaan palata välittömästi, mikä vähentää rekursiosta aiheutuvaa ylimääräistä laskentaa (engl. *overhead*) (Hoff 1997).

NFA-toteutukset kykenevät peruuttamaan syötemerkkijonoa taaksepäin — tämä mahdollistaa uusien hakuominaisuuksien, kuten takaisinviittausten ja merkkijonon selaamisen, käytön (Microsoft 2016). DFA-toteutukset eivät yleensä ole toiminnallisuudeltaan yhtä monipuolisia kuin NFA-toteutukset.

3.3 Suorituskyky

Eri toteutustapojen väliset suorituskykyerot voivat olla suuria. Wangin ym. (2013) mukaan DFA:han perustuvat toteutukset ovat nopeita, mutta käyttävät usein tarpeettoman paljon muistia — NFA:han perustuvat toteutukset puolestaan käyttävät vähemmän muistia, mutta voivat tietyissä tilanteissa olla hyvin hitaita. Cox'n (2007) mukaan NFA-toteutuksille jotkin regexpit ovat *patologisia* — niiden haku kestää joko äärimmäisen kauan, tai muisti ei riitä niiden suorittamiseen. DFA-toteutukset eivät kärsi patologisista regexpeistä.

Becchin ja Crowleyn (2008) mukaan käytännön regexpit sisältävät usein kolmentyyppisiä ominaisuuksia, jotka tekevät äärellisiin automaatteihin perustuvista regexptoteutuksista ongelmallisia:

1. Piste–tähti-lausekkeet (`.*`), jotka hyväksyvät mitä tahansa syötemerkkejä nolla tai useampi kappaletta
2. Laskentarajoitteet (esimerkiksi `a{3,9}`), joissa merkkijonokuvion toistojen määrä on annetulla välillä
3. Takaisinviittaukset (esimerkiksi `(a|b)\1`), jotka etsivät viittauksen kohteena olevan osalausekkeen sisältämää merkkijonoa.

Laskentarajoitteet ovat ongelmallisia muistinkäytön kannalta, ja takaisinviittaukset ovat ongelmallisia aikavaativuuden kannalta (Becchi ja Crowley 2008). Näiden ongelmien ratkaisua tarkastellaan luvussa 4.3.

3.3.1 Aikavaativuus

Cox'n (2007) mukaan rekursiiviseen peruutukseen perustuvien regexp-toteutusten aikavaativuus on pahimmassa tapauksessa eksponentiaalinen. Tämä johtuu siitä, että pahimmassa mahdollisessa tapauksessa NFA joutuu jokaisen syötemerkin kohdalla kokeilemaan kaikkia automaatin tiloja ja peruuttamaan niistä pois.

Rekursiivinen peruutus ei kuitenkaan ole ainoa tapa NFA:n simuloimiseen. Cox (2007) esittää keinon, joilla NFA:han perustuva haku voidaan suorittaa tehokkaammin: Sen sijaan, että NFA:n kaikki mahdolliset reitit käydään tarvittaessa läpi yksitellen, voidaan NFA:n kaikki mahdolliset reitit käydä läpi samanaikaisesti. Tällöin syöte luetaan ainoastaan yhden kerran.

Cox'n (2007) menetelmän aikavaativuus on lineaarinen, sillä peruutusta ei tapahdu. Sellaiset regexpien ominaisuudet, jotka peruutus mahdollistaa, eivät Cox'n menetelmässä ole mahdollisia. Menetelmän avulla voidaan kuitenkin merkittävästi parantaa NFA-toteutusten tehokkuutta.

DFA:han perustuvien regexp-toteutusten aikavaativuus on lineaarinen, sillä syöte luetaan ainoastaan kerran. Lähes kaikki nykyaikaisten regexpien ominaisuudet on mahdollista toteuttaa siten, että toteutuksen aikavaativuus on polynominen — lukuun ottamatta takaisinviittauksia. (Cox 2007.)

3.3.2 Muistinkäyttö

Yksinkertaisten hakujen tapauksessa muistinkäyttö ei yleensä ole merkittävä suorituskykytekijä minkään toteutustavan kohdalla. Käyttösovelluksissa, joissa muistia on rajallinen määrä, puhutaan kuitenkin ns. muistiräjähdysongelmasta — tämä ongelma koskee erityisesti DFA:ita (Wang ym. 2013).

Becchin ja Crowleyn (2008) mukaan DFA:n vaatiman muistin määrä voi olla eksponentiaalinen NFA:han verrattuna — tämä voi tehdä DFA-toteutuksista käyttökelvottomia sovelluksissa, jotka käyttävät paljon monimutkaisia regexpejä. Pisolkarin ja Lahanen (2015) mukaan DFA:ita on kuitenkin mahdollista optimoida ja tiivistää muistinkäytön vähentämiseksi.

4 Erilaisia regexp-toteutuksia

Tässä luvussa tarkastellaan joitakin regexp-hakujen sovellusalueita ja esitellään niihin liittyviä toteutustapoja. Kuten luvussa 3 havaittiin, rekursiiviseen peruutukseen pohjautuvat regexp-haut saattavat joissakin tapauksissa kärsiä suorituskykyongelmista. Luvussa esiteltävät regexp-toteutukset eivät tarjoa uusia ominaisuuksia, vaan tavoitteena on suorituskyvyn parantaminen heikentämättä toiminnallisuutta.

4.1 Osittainen jäsentäminen

Regexpejä käyttävän haun avulla voidaan selvittää, sisältääkö syöte regexpin kuvaamia merkkijonoja. Käytännössä olisi usein tarpeellista myös saada selville, missä kohtaa syötettä haun tulokset esiintyvät. Esimerkiksi normaali tekstihaku tekstidokumentista olisi melko hyödytön, jos sen tuloksena olisi vain tieto merkkijonon mahdollisesta löytymisestä mutta ei sijainnista tekstissä.

Edellä mainittu ongelma voidaan ratkaista *osittaisen jäsentämisen* (engl. *submatch addressing*) avulla. Osittainen jäsentäminen tarkoittaa regexpin osalausekkeita vastaavien merkkijonojen etsimistä (Laurikari 2001). Osalauseke on nimensä mukaisesti vain osa kokonaista regexpiä. Laajassa käytössä olevien osittaisen jäsentämisen algoritmien aikavaativuus pahimmassa tapauksessa on eksponentiaalinen ja vaativat muistia suoraan verrannollisesti syötteen pituuteen nähden (Laurikari 2001).

4.1.1 Merkityt siirtymät

Laurikarin (2000) ehdotus tehokkaaseen osittaiseen jäsentämiseen ovat *merkityt siirtymät* (engl. *tagged transitions*). Kyseessä on laajennus perinteisiin äärellisiin automaatteihin, jonka avulla automaatin siirtymiin voidaan lisätään ns. merkintöjä. Merkinnät esitetään muodossa t_x, jossa x on järjestysnumero. Jokaista merkintää vastaa muuttuja, johon tallennetaan syötemerkkijonon lukukohta silloin, kun kyseisen merkinnän sisältävä siirtymä tapahtuu. (Laurikari 2000.)

Esimerkiksi regexpissä `a*b`t_0`a*` merkintää t_0 vastaavaan muuttujaan tallennetaan syötteen lukukohta silloin, kun `b`-merkki luetaan. Jos koko syöte kuuluu regexpin kuvaamaan kieleen, voidaan t_0:aa vastaavasta muuttujasta lukea merkin `b` sijainti syötteessä. Jotta merkittyjä siirtymiä voidaan käyttää käytännön regexpeissä, täytyy symboleille t_x määritellä sopiva notaatio.

4.1.2 Haberin ym. algoritmi

Haber ym. (2013) ovat luoneet algoritmin, joka mahdollistaa osittaisen jäsentämisen suorittaminen tehokkaasti. Algoritmissa muodostetaan yhtä regexpiä kohden neljä automaattia: DFA:t M_1, M_3, M_4 ja NFA M_2.

Regexpin pohjalta muodostettu automaatti M_1 sisältää informaation etsittävistä osamerkkijonoista. Seuraavaksi M_1 muunnetaan useita alkutiloja sisältävksi NFA:ksi M_2. Se puolestaan muunnetaan DFA:ksi M_3, jonka tarkoitus on lukea syötemerkkijono takaperin — mahdollistaen merkkijonon selaamisen, joka yhden automaatin tapauksessa vaatisi rekursiivista peruutusta käyttävän algoritmin. (Haber ym. 2013.)

Automaattien M_1, M_2 ja M_3 pohjalta muodostetaan automaatti M_4. Lopullisessa haussa käytetään vain automaatteja M_3 ja M_4. Automaatti M_3 lukee syötteen takaperin ja selvittää, täsmääkö syöte regexpin kuvaamaan kieleen, sekä tallentaa tiedon tiloista, joissa se on ollut. Tätä tietoa ja automaattia M_4 käytetään jäsennettyjen merkkijonojen määrittämiseen. (Haber ym. 2013.)

Haberin ym. (2013) mukaan heidän algoritminsa on noin kaksi kertaa nopeampi verrattuna Java-ohjelmointikielen rekursiiviseen peruutukseen perustuvaan regexphakuun. Erityisen tehokas se on heidän mukaansa tilanteissa, joissa samaa regexpiä käytetään toistuvasti eri syötteillä — useiden automaattien luominen yhtä regexpiä varten on vaativaa, mutta valmiilla automaateilla haku on erittäin tehokas. Algoritmin aikavaativuus pahimmassa tapauksessa on eksponentiaalinen, mutta testien perusteella tällaiset tapaukset ovat harvinaisia (Haber ym. 2013).

4.2 Tietoturvauhkien havaitseminen

Wang ym. (2013) toteavat, että regexp-hakuja käytetään nykyään laajasti virusten ja hyökkäysten tunnistamiseen. Tähän tarkoitukseen käytetään Pisolkarin ja Lahanen (2015) mukaan yleensä DFA:han perustuvia regexp-hakuja niiden nopeuden takia, mutta DFA-toteutusten suuri muistinkäyttö rajoittaa niiden toimintakykyä.

Regexp-haut voivat myös aiheuttaa tietoturvauhkia. Berglundin ym. (2014) mukaan huolimattomasti toteutetuissa sovelluksissa rekursiiviseen peruutukseen perustuvia regexp-hakuja voidaan käyttää muun muassa palvelunestohyökkäyksiin.

4.2.1 StriFA

Wangin ym. (2013) mukaan *pakettien syvätutkinta* (engl. *deep packet inspection, DPI*) on olennainen osa nykyaikaisia tietoverkkojen tunkeutumisentunnistusjärjestelmiä. Niitä käytetään datapakettien tarkastamiseen mahdollisen uhkaavan sisällön varalta. Saapuva data tarkastetaan normaalisti tavu kerrallaan, joko normaalia merkkijonohakua tai regexpejä hyödyntävää hakua käyttäen. Regexpien monipuolisuuden johdosta ne ovat korvanneet merkkijonohaut useissa palveluissa. (Wang ym. 2013.)

Menetelmän pääidea on syötteen muuntaminen kokonaisluvuiksi, joita käsitellään Wangin ym. (2013) kehittämällä *StriFA*-automaatilla (*stride finite automaton*). Näin löydetään potentiaaliset regexpejä vastaavat merkkijonot, mutta menetelmä saattaa tuottaa myös ns. vääriä hälytyksiä — niiden osuus saadaan kuitenkin laskettua alle puoleen prosenttiin. Kaikki potentiaaliset hakutulokset kuitenkin tarkistetaan tutkimalla pätkää alkuperäisestä syötteestä. (Wang ym. 2013.)

Wang ym. (2013) kertovat, että StriFA kykenee saavuttamaan jopa kymmenenkertaisen nopeuden ja pienemmän muistinkäytön — verrattuna aikaisempiin DFA- ja NFA-perustaisiin tunkeutumisentunnistusjärjestelmiin — toimintakyvyn pysyessä samana.

4.2.2 Pakatut DFA:t

Pisolkar ja Lahane (2015) ovat kehittäneet menetelmän DFA:iden pakkaamiseen, jonka avulla niiden käyttämän muistin määrää on mahdollista vähentää. Menetelmässä regexp muunnetaan aluksi normaalisti DFA:ksi, jonka jälkeen DFA pakataan luomalla sille ns. pakkaussäännöt (Pisolkar ja Lahane 2015).

Pakatut DFA:t vievät merkittävästi vähemmän muistia pakkaamattomiin DFA:ihin verrattuna — tämä on tärkeää erityisesti regexpejä hyödyntävissä tietoturvasovelluksissa. Menetelmän avulla on mahdollista muodostaa pienin mahdollinen DFA mistä tahansa DFA:sta. (Pisolkar ja Lahane 2015.)

4.3 Laskentarajoitteet ja takaisinviittaukset

Becchi ja Crowley (2008) ovat luoneet *laajennetun äärellisen automaatin* (engl. *extended finite automaton*), jonka tavoitteena on laskentarajoitteita ja takaisinviittauksia sisältävien regexpien haun tehostaminen.

Laskentarajoitteiden ongelma ratkaistaan luomalla erityinen laskuriautomaatti, joka laskee merkkijonojen toistojen määrää laskureiden avulla sen sijaan, että jokaista lukumäärää vastaisi yksi automaatin tila. Takaisinviittausten tehokkuutta saadaan parannettua suorittamalla hakua syvyyshaun sijasta leveyshakuna NFA:sta aina kuin mahdollista. (Becchi ja Crowley 2008.)

Puhdas syvyyshaku tarkoittaisi käytännössä samaa kuin rekursiiviseen peruutukseen perustuva haku, mutta leveyshaun avulla tehokkuutta voidaan parantaa. Sekä laskuri-NFA:n että takaisinviittauksia käsittelevän NFA:n muuntaminen DFA:ksi tapahtuu samalla tavalla, joten molemmat toiminnot on mahdollista sisällyttää yhteen automaattiin. (Becchi ja Crowley 2008.)

Becchin ja Crowleyn (2008) mukaan heidän automaattinsa on ensimmäinen huippunopea automaatti, joka mahdollistaa kaikkien nykyaikaisista regexpeistä löytyvien ominaisuuksien käytön.

4.4 RE2

Googlen kehittämä *RE2* on avoimen lähdekoodin regexp-kirjasto. RE2 luotiin Googlen *Code Search* -palvelua varten, koska rekursiiviseen peruutukseen perustuvat regexp-haut olisivat olleet alttiita palvelunestohyökkäyksille (Cox 2010).

Hakuja voidaan luokitella sen mukaan, mitä toimintoja niiltä vaaditaan. Esimerkiksi osittaista jäsentämistä suorittavat haut ovat vaativampia kuin haut, jotka pelkästään etsivät merkkijonoja (Cox 2010).

Vaativuudeltaan erilaisiin hakuihin tarvitaan erilaisia automaatteja. RE2 pyrkii käyttämään ensisijaisesti DFA:ta jokaisen haun suorittamiseen, ja usein haku on mahdollista suorittaa pelkän DFA:n avulla. Joskus DFA kykenee suorittamaan haun vain osittain, jolloin tarvitaan lisäksi NFA haun suorittamiseen loppuun. (Cox 2010.)

DFA:t vaativat enemmän muistia NFA:hin verrattuna, mikä saattaa aiheuttaa ongelmia. RE2 kuitenkin rajoittaa yksittäisen haun muistinkäytön yhteen megatavuun. Rajan ylittyessä muisti vapautetaan ja hakua yritetään uudelleen luomalla uusi DFA. Mikäli haku epäonnistuu riittävän monta kertaa, RE2 turvautuu viimeisenä keinona hitaampaan hakuun NFA:n avulla. (Cox 2010.)

Yksinkertaiset regexpit RE2 suorittaa lähes yhtä nopeasti kuin muut sen kilpailijat. Koska RE2 ei joudu käyttämään rekursiivista peruutusta, se soveltuu erityisen hyvin suurten tekstimäärien läpikäyntiin. RE2 on aikavaativuudeltaan polynomiaalinen. (Cox 2010.)

RE2:sta puuttuu paljon ominaisuuksia, joihin useat muut laajasti käytetyt regexp-haut, kuten PCRE, tarjoavat käyttömahdollisuuden (Google 2016). Puuttuvia ominaisuuksia ovat kaikki ne, joita ei ole mahdollista toteuttaa tehokkaasti äärellisten automaattien avulla (Cox 2010).

5 Yhteenveto

Tässä kirjallisuuskatsauksessa tarkasteltiin erilaisia tapoja toteuttaa regexp-haku äärellisten automaattien avulla ja todettiin, että erilaiset toteutustavat mahdollistavat erilaisten regexp-toimintojen käytön. Sopivaa toteutustapaa valittaessa joudutaan tasapainottelemaan tehokkuuden ja toiminnallisuuden välillä.

Berglundin ym. (2014) mukaan regexpien kehitys on kulkenut käytännöllisyys ja toiminnallisuus edellä, ja teoreettinen puoli ei ole pysynyt vauhdissa mukana. Cox'n (2007) mukaan regexpien teorian heikko ymmärrys ja huomiotta jättäminen on johtanut siihen, että rekursiiviseen peruutukseen perustuvia toteutustapoja pidetään laajasti ainoina vaihtoehtoina NFA:n simuloimiseen. Luvussa 3 todettiin, että tämä ei pidä paikkaansa.

NFA:han perustuvat regexp-haut ovat suosittuja, sillä ne tarjoavat enemmän mahdollisuuksia monipuolisten hakujen suorittamiseen (Microsoft 2016). DFA:han perustuvia toteutuksia ei kuitenkaan pidä aliarvioida, sillä niiden nopeus ja turvallisuus ovat välttämättömiä tietyillä sovellusalueilla.

Kirjallisuutta

Becchi, M. & Crowley, P. 2008. *Extending finite automata to efficiently match Perl-compatible regular expressions*. CoNEXT '08 Proceedings of the 2008 ACM CoNEXT Conference. Artikkeli no. 25.

Berglund, M., Drewes, F. & van der Merwe, B. 2014. *Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching*. `arXiv:1405.5599v1 [cs.FL]`

Cox, R. 2007. *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)*. Saatavilla internetissä <URL: `https://swtch.com/~rsc/regexp/regexp1.html`>. Viitattu 9.2.2016.

Cox, R. 2010. *Regular Expression Matching in the Wild*. Saatavilla internetissä <URL: `https://swtch.com/~rsc/regexp/regexp3.html`>. Viitattu 10.3.2016.

Feldman, Y. 1993. *Finite-State Machines*. Encyclopedia of Computer Science and Technology, s. 387–418.

Google 2016. *RE2 Syntax*. Saatavilla internetissä <URL: `https://github.com/google/re2/wiki/Syntax`>. Viitattu 28.2.2016.

Haber, S., Horne, W., Manadhata, P., Mowbray, M. & Rao, P. 2013. *Efficient Submatch Extraction for Practical Regular Expressions*. Language and Automata Theory and Applications. Berlin, Heidelberg: Springer, s. 323–344.

Hoff, B. 1997. *High-speed finite-state machines*. Dr. Dobb's Journal. Volume 22, no. 11, s. 54–61. ProQuest.

Hopcroft, J., Motwani, R. & Ullman, J. 2001. *Introduction to Automata Theory, Languages, and Computation*. 2. painos. Addison-Wesley.

Laurikari, V. 2000. *NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions*. Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, s. 181–187.

Laurikari, V. 2001. *Efficient submatch addressing for regular expressions*. Saatavilla internetissä <URL: `http://laurikari.net/ville/regex-submatch.pdf`>. Viitattu 14.2.2016.

Microsoft 2016. *Details of Regular Expression Behavior*. Saatavilla internetissä <URL: https://msdn.microsoft.com/en-us/library/e347654k(v=vs.110).aspx>. Viitattu 10.3.2016.

Mitkov, R. 2003. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, s. 754.

Pisolkar, U. & Lahane, S. 2015. *A Memory Efficient Regular Expression Matching by Compressing Deterministic Finite Automata*. ProQuest.

Wang, X., Xu, Y., Jiang, J., Ormond, O., Liu, B. & Wang, X. 2013. *StriFA: Stride Finite Automata for High-Speed Regular Expression Matching in Network Intrusion Detection Systems*. IEEE Systems Journal. Volume 7, Issue 3, s. 374–384. doi:10.1109/JSYST.2013.2244791