Säännöllisten lausekkeiden tulkki

Riku Oja

13. tammikuuta 2015

Toteutus koostuu kolmesta luokasta: RORegEx, RONFA ja ROState.

0.1 RoRegEx-luokka

RORegEx on rajapintaluokka, joka sisältää kirjastokäyttöliittymän säännöllisen lausekkeen matchausta varten. Luokka konstruoi halutusta regexistä ja halutusta stringistä NSTextCheckingResult-olion, joka sisältää kaikki stringin sisältämät regex-matchit eli aloitusindeksit ja pituudet järjestyksessä.

0.2 RONFA-luokka

RONFA on luokka, joka sisältää itse regex-tulkin konstruointilogiikan sekä myös matchauslogiikan. Luokka initialisoidaan halutulla regexillä, jonka jälkeen findMatch:-kutsulla voidaan saada halutusta stringistä ensimmäinen regexiä vastaava match.

RONFA-luokan sisäinen toteutus jakautuu vastaavasti initialisointi- ja matchausmetodeihin. Koska RONFA on toteutus epädeterministiselle tilakoneelle, joka vastaa haluttua regexiä, itse olion initialisointi eli tilakoneen konstruointi on eniten koodirivejä sisältävä metodi. initWithRegEx: käy läpi regexin merkki kerrallaan ja lisää RONFA:n alkutilaan uusia seuraavia tiloja sen mukaan, mikä merkki tai operaattori lausekkeessa tulee vastaan. Näin ollen koko lausekkeen tunnistamislogiikka synnytetään heti olion initialisointivaiheessa. Metodi on sitä monimutkaisempi, mitä monimutkaisempia regex-operaattoreita halutaan toteuttaa.

RONFA-luokan findMatch:-metodi ajaa tilakonetta alkutilasta lähtien annetulla stringillä. Koska RONFA-luokka käyttää matchien tallentamiseen itse tilaolioita, tämän metodin alussa tilakone joudutaan konstruoimaan uudestaan edellisten tulosten poistamiseksi.

0.3 ROState-luokka

RONFA käyttää ROState-luokkaa esittämään yhtä mahdollista tilaa. Näin ollen tilan olennaiset ominaisuudet ovat matchaava kirjain (matchingCharacter), seuraavat mahdolliset tilat (nextState sekä alternateState) ja lopputila (finality). Näistä matchingCharacter voi olla nil, jos halutaan matchata mikä tahansa kirjain; alternateState taas on nil, mikäli tila ylipäänsä on matchaava tila, ja toinen tila, mikäli tila haarautuu. Tällöin matchingCharacter-arvoa ei lueta. Lisäksi luokka sisältää joka kierroksella päivitettävät startIndex ja nextStartIndex-kentät, joihin tilakone tallentaa kulloisenkin matchin aloitusindeksit.

ROState-luokkaan ei ollut järkevää lisätä tilakoneen toteutuslogiikkaa, sillä mitkä tahansa operaattorit on toteutettavissa kolmea tilatyyppiä yhdistelemällä; näin ollen yksittäinen ROState on lähinnä olio, joka sisältää muutaman kentän sekä viittauksen muihin ROState-olioihin. Näistä konstruoituu RONFA:n initialisaatiossa puu, joka sisältää viittaukset kaikkiin mahdollisiin tilakoneen tiloihin. Olemassaoleviin tiloihin ei ole muita viittauksia kuin ensimmäisestä tilasta lähtevä, haarautuva linkitetty lista. Operaattoreista riippuen listaan syntyy myös silmukoita ja haarojen yhdistymisiä. Se, mitkä tilat päätyvät kulloinkin tilakoneen currentStates-joukkoon, riippuu siitä, mihin tilakoneen haaroihin stringin matchaus etenee.

1 Paralleelin tilakoneen toteutus

Olennaisesti tilakone koostuu kolmesta erityyppisestä tilasta: 1) kirjaimenmatchaustila, josta siirrytään kirjaimesta riippuen seuraavaan tilaan tai takaisin alkutilaan, 2) haarautuva tila, jota tarvitaan operaattorien toteutukseen ja josta siirrytään automaattisesti kahteen uuteen tilaan, sekä 3) lopputila, joka indikoi stringin matchausta ja jonka kohtaaminen (tässä toteutuksessa) lopettaa tilakoneen toiminnan ennen kuin string on käsitelty loppuun. Näistä haarautuva tila eroaa kirjaimenmatchaustilasta sikäli, että se vastaa säännöllisessä kielessä ns. epsilon-siirtymää, eli siitä siirrytään eteenpäin ilman, että luettavassa stringissä siirrytään eteenpäin.

Lineaarisen suorituskyvyn kannalta olennaisin ominaisuus on se, että tilakone käy läpi stringiä merkki merkiltä siirtyen *kaikkiin* niihin tiloihin, joihin regexin perusteella on mahdollista siirtyä. Tällöin kukin merkki on sekä mahdollinen matchin aloitusmerkki että matchin seuraava merkki, mikäli match on alkanut jo aiemmin. Aloitustiloja on aina yksi, mutta heti matchin alkaessa tilakone haarautuu.

Näiden tilojen tallennus on toteutettu NSMutableSet-joukkoon currentStates, joka on siis järjestämätön joukko olioita. Mikäli sama tila tulee vastaan saman merkin kohdalla tilakoneen eri haaroissa, tilaa *ei* lisätä useampaan kertaan seuraavaksi käsi-

teltäviin tiloihin, vaan haarat sulautuvat takaisin yhteen; tämä on olennaista eksponentiaalisen skaalautumisen estämiseksi. Samalla kuitenkin unohtuu se, mitä haaroja pitkin kuhunkin tilaan on päästy. Matchin alkukirjaimen muistamiseksi tilakone tallentaa matchin alkuindeksin matchattuun ROState-tilaan ja kopioi alkuindeksiä eteenpäin siirryttäessä. Ns. ahneen matchauksen toteuttamiseksi pienin alkuindekseistä on aina se, joka jää voimaan seuraavassa tilassa.

Ahne algoritmitoteutus edellyttäisi tarkkaan ottaen sitä, että myös matchin loppupituutta maksimoitaisiin eli käytäisiin läpi pitemmät mahdolliset matchit, vaikka lyhyin match on jo löytynyt. Tämä on tavallisimpien regex-operaattorien oletusominaisuus, ja olisi helposti toteutettavissa sillä, että matchausta ei katkaista ensimmäisen matchin löytyessä; nykyinen toteutus kuitenkin preferoi mahdollisimman aikaisia ja lyhyitä matcheja.

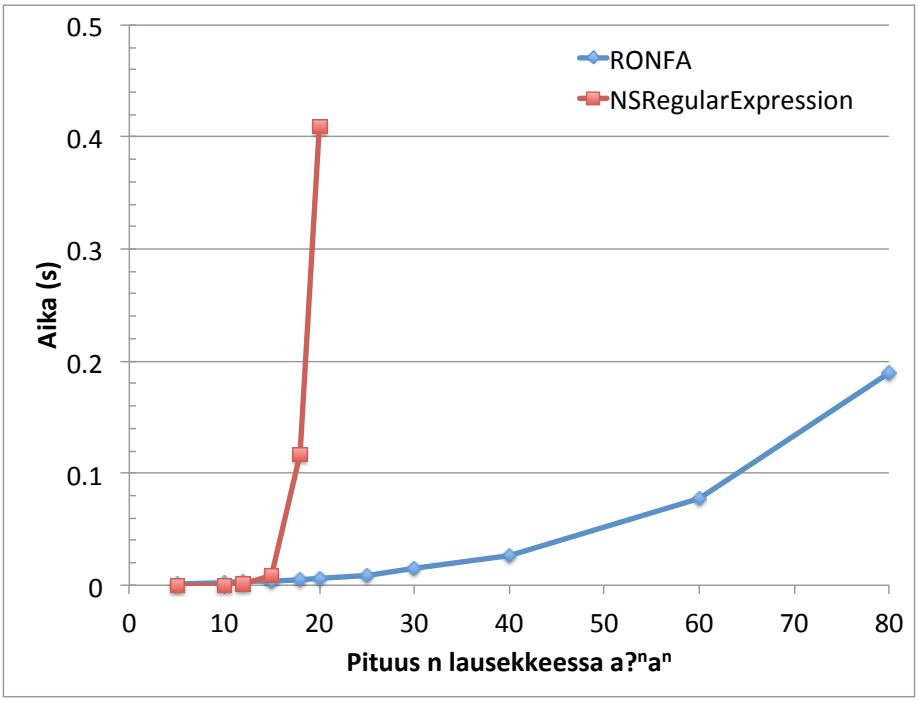
Koska matchaavien tilojen lisäksi kone saattaa sisältää myös haarautuvia tiloja, joista edetään ilman etenemistä stringissä, nämä tilat on otettava huomioon ennen kuin stringissä edetään seuraavaan kirjaimeen. Tämä tapahtuu jokaisen kirjaimen kohdalla pruneForks-metodilla. Metodi käy kaikki currentStatesin sisältämät tilat läpi ja haaran löydettyään lisää tämän haarat currentStatesiin sekä poistaa alkuperäisen tilan. Näin kirjaimen matchaukseen päädytään vasta siinä vaiheessa, kun kaikki haarautuvat tilat sekä niiden alla olevat haarautuvat tilat on poistettu ja currentStates sisältää ainoastaan sellaisia tiloja, joita matchataan stringin nykyiseen kirjaimeen. Itse kirjaimen matchaus on silmukassa eroteltu matchCharacter-metodiin, joka huolehtii matchauksen aloitusindeksin tallentamisesta.

2 Testaustulokset

Lineaarisen skaalautuvuuden testaus on syytä tehdä lausekkeella, joka tuottaa ongelmia backtrackaus-algoritmeille eksponentiaalisesti kasvavan vaativuuden vuoksi. Tällainen lauseke on Coxin mallin mukaan $(a?)^n(a)^n$ eli lauseke, jossa on n kappaletta valinnaista a-kirjainta sekä tämän jälkeen n kappaletta välttämättömiä a-kirjaimia. Mikäli tämä lauseke halutaan matchata stringiin, jossa on täsmälleen n a-kirjainta, eksponentiaalisesti kasvavalle algoritmille ongelmia tulee melko nopeasti; algoritmi käy läpi kaikki mahdollisuudet, joissa valinnaiset a-kirjaimet voivat olla mukana stringissä, ja näiden mahdollisuuksien määrä skaalautuu eksponentiaalisesti.

Lineaarisesti stringin alusta loppuun etenevä algoritmi sen sijaan haarautuu m:nnellä askeleella korkeintaan (ja tämän lausekkeen tapauksessa täsmälleen!) m:ään osaan, sillä m:n pituinen lauseke synnyttää aina tilakoneen, jossa on korkeintaan m kappaletta erilaisia tiloja. Lisätyötä aiheuttaa se, että käyttämässäni matchauksen alkupisteen tallennusmenetelmässä ennen matchausta kaikki tilat tulee käydä läpi ja

nollata aloitusindeksit, eli käytännössä rakentaa tilakone uudestaan jokaista matchia varten. Tämä operaatio vaatii $O(m^2)$-ajan, sillä jokaisella askeleella tilakone saattaa luoda korkeintaan yhden uuden tilan. Itse matchauksessa jokainen askel vaatii $O(m)$-ajan, joten n:n pituisen stringin matchaus vaatii ajan $O(mn)$ tai koneen konstruointi mukaan lukien $O(m(m+n))$-ajan.



Kuva 1: n:n kirjaimen pituisen testistringin matchauksen nopeus n:n funktiona toteutetulla algoritmilla (RONFA) sekä Applen NSRegularExpression-algoritmilla.

Lausekkeen $(a?)^n(a)^n$ matchauksen skaalautuvuus on osoitettu testeillä testPerformancePathologicaln, missä stringin pituus on $n = 5, 10, 12, 15, 18, 20, 25, 30, 40, 60, 80$. Tällöin itse regexin pituudeksi tulee $3n$. Koko findMatch-metodin suoritus tulisi kestää ajan $O(3n(3n+n))$ eli $O(n^2)$. Tulosten perusteella skaalautuminen on aavistuksen verran huonompaa kuin neliöllinen; kolmannen asteen polynomi sopii dataan merkittävästi paremmin (korrelaatiokerroin $R^2 > 0.999$).

Kuutiollisen skaalautumisen syitä pitää etsiä findMatch-kutsun tarkemmasta toteutuksesta, joka vaatii matchin aloitusindeksien tallennuksen ja päivityksen jokaisella askeleella. Koska jokaisen käsitellyn ja tulevan tilan aloitusindeksit on päivitettävä tasan kerran riippumatta siitä, kuuluuko tila jompaankumpaan vai kumpaankin joukoista currentStates ja nextStates, jokainen askel edellyttää currentStates- ja

nextStates-joukkojen yhdistämistä, eli käytännössä n:n kokoisen joukon läpikäymistä kertaalleen duplikaattien eliminoimiseksi.

Tämä voi pahimmillaan johtaa ylimääräiseen n-kertaiseen hidastumiseen jokaisella kierroksella. Yleisillä regexeillä kerrallaan käsiteltävien joukkojen määrä ei välttämättä ole yhtä suuri kuin regexin kokonaispituus, mutta testitapauksessa, jossa joukon koko kasvaa joka askeleella yhdellä, kokonaisvaikutus on efektiivisesti n-kertainen hidastuminen. Tällöin aloitusindeksit ROState-olioon tallentava algoritmi skaalautuu itse asiassa nopeudella $O(n^3)$, ei $O(n^2)$. Stringin pituuden n ja regexin pituuden m funktiona lopullinen skaalautuminen on siis indeksit tallennettaessa $O(m(m+n)n)$.

Vertailun vuoksi saman lausekkeen matchausta mitattiin myös Applen sisäänrakennetulla NSRegularExpression-oliolla. Tämä on tehty testeillä testNSPerformancePathologicaln, missä stringin pituus on $n = 5, 10, 12, 15, 18, 20$. Yli kahdenkymmenen testejä ei ajettu loppuun, sillä ne eivät valmistuneet siedettävässä ajassa. Tuloksissa on selvästi nähtävillä tavanomaisen regex-toteutuksen eksponentiaalinen skaalautuminen huonosti valituilla lausekkeilla.

3 Puutteet ja parannusehdotukset

Vaikeimmaksi kysymykseksi työn toteutuksessa osoittautui matchin aloitusindeksien tallentaminen ymmärrettävään tietorakenteeseen. ROState-olioon tallentaminen ei välttämättä ole toteutuksellisestikaan tehokkain vaihtoehto; toisaalta haaroja ei ole mahdollista erottaa toisistaan lineaarisessa toteutuksessa, joten tallennettavia indeksejä on lopulta vain yksi tilaa kohden, ja tämä valinta tuntuu luonnolliselta. On hyvinkin mahdollista, että toisella aloitusindeksin tallennustavalla algoritmi voisi olla nopeampi ja näin ollen lineaarinen eikä neliöllinen stringin pituuden funktiona myös vaikeiden regexien tapauksissa; en ole kuitenkaan selvittänyt tarkemmin, millä tavalla esimerkiksi Coxin toteutuksessa matchin paikan tallennus on tehty.

Toinen vaikeus on itse koodin saaminen helposti hahmotettavaksi ja luettavaksi. Regex-operaattorien suuri määrä johtaa siihen, että jokaisella askeleella suoritussilmukassa on runsaasti erilaisia vaihtoehtoja sekä tilakoneen konstruointi- että suoritusvaiheissa, joten näiden vaihtoehtojen eristäminen silmukan ulkopuolelle algoritmin hahmottamisen helpottamiseksi ei ole aivan suoraviivaista. Lisäksi juuri päivitettävien tilojen sekä aloitusindeksien päivittäminen silmukan aikana on vaikeasti hahmotettava prosessi. Tämä kuitenkin kuuluu paralleelin tilakoneen luonteeseen: sama tila voi olla sekä yhdellä kierroksella suoritettavien tilojen että toisaalta seuraavalla kierroksella jonkin muun tilan vuoksi suoritettavien tilojen joukossa, joten tiloja ei voi aina käsitellä ja päivittää toisistaan riippumattomina.

Viimeinen ja luontevin parannusehdotus on tietysti uusien regex-operaattoreiden

ja -ominaisuuksien toteuttaminen. Keskeisin puuttuva regex-ominaisuus on matchaus stringin loppuun asti; tämänhetkinen toteutus palauttaa ensimmäisen matchin eikä maksimaalista matchia. Matchaus stringin loppuun asti edellyttäisi olemassaolevien matchien tallennusta erilliseen tietorakenteeseen ja algoritmin jatkamista myös tilakoneen lopputilasta eteenpäin.

Uusien operaattorien toteuttaminen on varsin helppoa insertFork-metodin avulla, sillä mikä tahansa operaattori voidaan toteuttaa sulkulausekkeiden, character matchin sekä haarautumisen avulla. Mitä monimutkaisempi operaattori on kyseessä, sitä useamman koodirivin kuitenkin vaatii operaattorin kuvailu yksinkertaisten operaattoreiden yhdistelmänä sekä regexin lukeminen. Regex luetaan edelleen merkki kerrallaan, mutta useat monimutkaisemmat regex-lausekkeet ovat useamman merkin mittaisia, joten lukeminen tulee tehdä samaan tapaan kuin sulkulausekkeiden käsittely on tässä implementoitu, tarpeen mukaan myös rekursiota käyttäen.