Collatzin konjektuuri ja algoritmien analysointi

Antti Laaksonen
Aalto-yliopisto ja Helsingin yliopisto
ahslaaks@cs.helsinki.fi

Jokaiselle tietokoneen käyttäjälle ovat tuttuja erilaiset ongelmat ohjelmien kanssa, kuten jumiutuminen ja virheellinen toiminta. Ohjelmien virheitä korjataan pikkuhiljaa, mutta aina löytyy myös uusia. Herää kysymys, miksi tällaisia ongelmia ei korjata järjestelmällisesti. Eikö olisi hyvä idea ottaa avuksi matematiikka ja *todistaa* ennen ohjelman julkaisua, että ohjelma toimii aina halutulla tavalla?

Tämä on hieno tavoite, mutta tehtävä on vaikeampi kuin voisi kuvitella. Itse asiassa osoittautuu, että yleisessä tapauksessa on *mahdotonta* analysoida automaattisesti ohjelmien toimintaa. Tässä kirjoituksessa tutustumme aiheeseen sekä teorian että käytännön kautta.

Collatzin konjektuuri

Collatzin konjektuuri vuodelta 1937 liittyy seuraavaan algoritmiin: Valitaan aluksi positiivinen kokonaisluku n. Jos n on parillinen, se jaetaan kahdella, ja jos n on pariton, se kerrotaan kolmella ja tulokseen lisätään yksi. Tätä jatketaan, kunnes n on 1. Esimerkiksi valinta $n = 6$ tuottaa seuraavan ketjun:

$$6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Collatzin konjektuurin mukaan millä tahansa n:n valinnalla algoritmin tuottama ketju päättyy lukuun 1 eli algoritmi *pysähtyy* aina.

Algoritmin voi toteuttaa käytännössä esimerkiksi seuraavasti Python-kielellä:

```
def collatz(n):
    while n > 1:
        if n%2 == 0:
            n = n/2
        else:
            n = 3*n+1
```

On helppoa varmistaa, että Collatzin algoritmi pysähtyy pienillä n:n arvoilla, mutta voisiko olla jokin suuri n:n arvo, jolla näin ei tapahdu?

Suoraviivainen tapa tutkia algoritmin toimintaa on antaa sille järjestyksessä n:n arvoja $1, 2, 3, \ldots$ ja katsoa, mitä tapahtuu. Yksi kiinnostava tieto algoritmiin liittyen on *askelten määrä* ketjussa tietyllä n:n arvolla. Voimme laskea tämän muuttamalla algoritmia näin:

```
def collatz(n):
    c = 0
    while n > 1:
        c += 1
        if n%2 == 0:
            n = n/2
        else:
            n = 3*n+1
    return c
```

Nyt esimerkiksi funktiokutsu `collatz(6)` palauttaa arvon 8, koska tapauksessa $n = 6$ ketjussa on 8 askelta.

Voimme laskea lämmittelynä vaikkapa askelten määrät, kun alkuarvo n on välillä $1 \ldots 10$:

alkuarvo n	askelten määrä
1	0
2	1
3	7
4	2
5	5
6	8
7	16
8	3
9	19
10	6

Taulukosta voi havaita, ettei askelten määrälle ole ainakaan kovin yksinkertaista sääntöä, joka pistäisi silmään heti ensimmäisistä taulukon arvoista.

Nyt on aika ottaa järeämmät keinot käyttöön ja tutkia algoritmia suuremmilla n:n arvoilla. Seuraava taulukko kertoo suurimman askelten määrän, kun algoritmia testataan kaikilla n:n arvoilla tiettyyn rajaan asti.

suurin n:n arvo	suurin askelten määrä
10	19, kun $n = 9$
10^2	118, kun $n = 97$
10^3	178, kun $n = 871$
10^4	261, kun $n = 6171$
10^5	350, kun $n = 77031$
10^6	524, kun $n = 837799$

Tiedämme nyt, että algoritmi pysähtyy ainakin alkuarvoon $n = 10^6$ asti ja suorittaa enimmillään 524 askelta tapauksessa $n = 837799$ ennen pysähtymistä. Näyttää siltä, että n:n kasvaessa löytyy aina uusia tapauksia, joissa askelten määrä on aiempaa suurempi, mutta askelten määrä kasvaa hitaasti verrattuna n:n kokoon.

Voisimme jatkaa algoritmin tutkimista vielä suuremmilla n:n arvoilla, joskin ennen pitkää laskenta alkaisi viedä paljon aikaa. Kokeilemalla kaikkia arvoja on selvitetty [3], että algoritmi pysähtyy ainakin aina, kun n on korkeintaan $5 \cdot 2^{60}$. Tämän laskeminen vaati hyvin optimoidun algoritmin ja useita vuosia aikaa.

Tämän aineiston perusteella *vaikuttaa* siltä, että algoritmi pysähtyy aina. Kukaan ei kuitenkaan tiedä tapaa *todistaa*, että näin olisi, vaikka lukuisat matemaatikot ovat pohtineet ongelmaa.

On helppoa keksiä erilaisia muunnelmia algoritmista. Esimerkiksi Solmun numerossa 1/2017 Juhani Fiskaali esitti seuraavan algoritmin [1]:

```
def fiskaali(n):
    while n > 1:
        if n%3 == 0:
            n = n/3
        elif n%3 == 1:
            n = 4*n-1
        else:
            n = 5*n-7
```

Tämän algoritmin analysointi on huomattavasti helpompaa, koska arvolla $n = 1352$ algoritmi ei pysähdy koskaan, vaan ketju on muotoa

$$1352 \to 6753 \to 2251 \to 9003 \to \cdots \to 9003 \to \cdots,$$

missä luvusta 9003 alkaa 194 askeleen ketju, joka palaa takaisin lukuun 9003.

Collatzin konjektuuri antaa näytteen siitä, miten vaikeaa voi olla selvittää, pysähtyykö algoritmi. Vaikka algoritmissa on vain muutama rivi koodia, kukaan ei osaa sanoa varmasti, pysähtyykö se. Vertailun vuoksi tietokoneen käyttöjärjestelmässä voi olla miljoonia rivejä koodia. Ei ihme, että tietokone voi jäädä jumiin, eikä kukaan osaa selittää asiaa.

Pysähtymisongelma

Pysähtymisongelmassa on tehtävänä selvittää, pysähtyykö annettu algoritmi. Esimerkiksi seuraava algoritmi laskee summan $1 + 2 + \ldots + n$ ja selvästi pysähtyy kaikilla n:n arvoilla:

```
def summa(n):
    s = 0
    for i in range(1,n+1):
        s += i
    return s
```

Seuraava algoritmi taas ei pysähdy syötteellä $n = 13$, koska algoritmi tulostaa silloin loputtomasti rivejä, joissa lukee ”Heippa!”:

```
def hassu(n)
    if n == 13:
        while True:
            print "Heippa!"
```

Pysähtymisongelma on varmasti vaikea ongelma, koska Collatzin konjektuuri on vain yksi ongelman erikoistapaus. Itse asiassa jos meillä olisi varma keino selvittää, pysähtyykö annettu algoritmi aina, voisimme ratkaista saman tien monia matematiikan avoimia ongelmia. Esimerkiksi lukuteorian suuria kysymyksiä on, onko olemassa äärettömästi alkulukupareja eli pareja muotoa $(p, p+2)$, missä p ja $p+2$ ovat alkulukuja. Esimerkiksi $(5, 7)$, $(11, 13)$ ja $(17, 19)$ ovat tällaisia pareja. Voimme tutkia ongelmaa seuraavalla algoritmilla:

```
def haku(n):
    while True:
        if alkuluku(n) and alkuluku(n+2):
            break
        n += 1
```

Algoritmille annetaan luku n ja se alkaa etsiä paria $(p, p+2)$, jossa $p \geq n$ ja p ja $p+2$ ovat alkulukuja.

Heti jos tällainen pari löytyy, algoritmi pysähtyy, mutta jos paria ei löydy koskaan, haku jatkuu ikuisesti. Alkulukupareja on äärettömästi tarkalleen silloin, kun algoritmi pysähtyy millä tahansa syötteellä, joten on yhtä vaikeaa selvittää algoritmin pysähtyminen kuin ratkaista alkulukuparien ongelma.

Vuonna 1936 Alan Turing osoitti, että pysähtymisongelmaa ei ole mahdollista ratkaista: ei ole olemassa algoritmia, joka pystyisi tutkimaan luotettavasti ohjelman koodista, pysähtyykö se vai ei. Tällaista algoritmia ei ole mahdollista toteuttaa, koska voimme aina löytää tilanteen, jossa algoritmi erehtyisi. Tarkastellaan esimerkkinä seuraavaa funktiota, joka *yrittää* tutkia ohjelman pysähtymistä:

```
def pysahtyy(x, y):
    ...
```

Parametri x on tutkittavan ohjelman koodi, parametri y on ohjelman syöte ja funktion tulisi palauttaa `True`, jos ohjelma pysähtyy, ja `False`, jos ohjelma ei pysähdy. Emme tiedä tarkalleen, miten funktio toimisi, joten funktion sisältönä on vain kolme pistettä. Voimme kuitenkin laatia seuraavan ohjelman, jonka tarkoituksena on harhauttaa funktiota:

```
def huijaus(x):
    if pysahtyy(x, x):
        while True:
            print "Enkä pysähdy!"
    else:
        print "Pysähdynpäs!"
```

Harhautus onnistuu silloin, kun parametrina x on ohjelman oma koodi. Ideana on, että ohjelma kysyy ensin, pitäisikö sen pysähtyä tässä tilanteessa vai ei, ja toimii sitten täsmälleen päinvastoin. Jos funktion `pysahtyy` mukaan ohjelman pitäisi pysähtyä, se jää silmukkaan, ja jos taas ohjelman pitäisi jäädä silmukkaan, se pysähtyy. Niinpä funktio `pysahtyy` antaa väärää tietoa pysähtymisestä riippumatta siitä, miten se on toteutettu, eli ei voi olla olemassa toimivaa funktiota ohjelman pysähtymisen tarkastamiseen.

Tässä on kysymys suunnilleen samasta asiasta kuin vanhassa vitsissä, jossa henkilö sanoo ”Minä valehtelen”. Jos henkilö puhuu totta, hän kuitenkin valehtelee, ja jos taas henkilö valehtelee, hän puhuukin totta.

Ricen lause

Pysähtyminen ei ole ainoa vaikeasti selvitettävä asia ohjelmasta, vaan sama koskee *kaikkia* muitakin ohjelman ominaisuuksia. Esimerkiksi yhtä vaikea tehtävä on selvittää, tulostaako ohjelma kirjainta ”a”. Tämä tulos tunnetaan nimellä Ricen lause.

Esimerkiksi jos voisimme selvittää ohjelman koodista, tulostaako se aina kirjaimen ”a”, voisimme ratkaista Collatzin konjektuurin yhtä helposti kuin ennenkin. Riittää muuttaa ohjelmaa niin, että se tulostaa pysähtyessään kirjaimen ”a”:

```
def collatz(n):
    while n > 1:
        if n%2 == 0:
            n = n/2
        else:
            n = 3*n+1
    print "a"
```

Tämän ansiosta kirjaimen ”a” tulostaminen ilmaisee meille, milloin ohjelma pysähtyy. Koska pysähtymisongelmaa ei ole mahdollista ratkaista, ei ole myöskään mahdollista ratkaista, tulostaako ohjelma kirjainta ”a”.

Voimme tietenkin korvata kirjaimen ”a” tulostamisen millä tahansa koodilla. Esimerkiksi seuraava ohjelma soittaa Tapio Rautavaaran *Isoisän olkihattu* -kappaleen tarkalleen silloin, kun algoritmi pysähtyy:

```
def collatz(n):
    while n > 1:
        if n%2 == 0:
            n = n/2
        else:
            n = 3*n+1
    soita("olkihattu.mp3")
```

Emme pysty siis edes selvittämään ohjelmasta, soittaako se Isoisän olkihatun.

Entä käytännössä?

Tässä vaiheessa tavoitteemme todistaa algoritmien toimivuus matemaattisesti vaikuttaa jäävän haaveeksi, koska näyttää olevan mahdotonta tarkastaa *mitään* ohjelman ominaisuutta automaattisesti. Mutta onko asia todella näin?

Tässä täytyy kuitenkin muistaa, että äskeisten tulosten mukaan ohjelman analysointia yrittävä algoritmi erehtyy ainakin *yhdellä* syötteellä. Tämä ei kuitenkaan tarkoita, että algoritmi olisi välttämättä täysin käyttökelvoton. Hieman vastaava tilanne on jakolaskun laskeminen. Tunnetusti jakolasku ei ole mahdollinen, jos jakajana on nolla, mutta tämä ei tarkoita, ettei jakolaskua voisi käyttää missään tilanteessa.

Käytännössä voi olla hyvinkin mahdollista varmistaa monista ohjelmista, että ne toimivat oikealla tavalla. Esimerkiksi Microsoftin tutkimusyksikössä oli joitakin vuosia sitten käynnissä Terminator-projekti, jonka tavoitteena oli tutkia ohjelmien pysähtymistä. Tehtävä on kuitenkin vaikea ja olemassa olevat työkalut soveltuvat lähinnä lyhyiden algoritmien tutkimiseen.

Vuonna 2015 tutkijat koettivat todistaa harjoituksen vuoksi, että Java-kielen standardikirjaston järjestämisalgoritmi toimii oikein. Todistaminen ei kuitenkaan ottanut onnistuakseen, ja lopulta tutkijat tekivät yllättävän havainnon: Javan algoritmi on *virheellinen*, minkä vuoksi ei olekaan mahdollista todistaa, että se toimisi oikein [2]. Kukaan Javan käyttäjä ei ollut huomannut asiaa, koska virhe on hyvin harvinainen: taulukko, jossa virhe esiintyy, sisältää 67108864 alkiota. Tämä osoitti, että jo nykyisillä menetelmillä on mahdollista saada tuloksia, joilla on merkitystä käytännössä.

Ei näytä kuitenkaan siltä, että kokonaisten ohjelmien analysointi olisi mahdollista vielä pitkään aikaan. Mittava haaste on pelkästään se, että ennen analyysia täytyy kuvata matemaattisesti, mikä ohjelman haluttu toiminta on missäkin tilanteessa. Tämä on helppoa tarkasti määritellylle algoritmille mutta hyvin vaikeaa vaikkapa nettiselaimelle tai tekstinkäsittelyohjelmalle.

Lopuksi

Yllättävää kyllä, on olemassa algoritmi, joka kertoo, päteekö Collatzin konjektuuri vai ei. Kyseinen algoritmi on jompikumpi seuraavista:

```
def totuus1():
    print "Collatzin konjektuuri pätee"

def totuus2():
    print "Collatzin konjektuuri ei päde"
```

Huonona puolena on vain se, että emme tiedä, *kumpi* algoritmeista toimii oikein.

Viitteet

[1] J. Fiskaali: Eräs Collatz-Kakutanin otaksuman analogia. *Solmu* 1/2017. `http://matematiikkalehtisolmu.fi/2017/1/otaksuma.pdf`

[2] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel ja R. Hähnle: OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. *27th International Conference on Computer Aided Verification*, 2015.

[3] T. O. e Silva: Computational verification of the 3x+1 conjecture. `http://sweet.ua.pt/tos/3x+1.html`, 2009.

Verkko-Solmun oppimateriaalit

Osoitteesta `matematiikkalehtisolmu.fi/oppimateriaalit.html` löytyvät oppimateriaalit:

Suppeaa suhteellisuusteoriaa alusta alkaen (Lasse Pantsar)

Lukion matemaattisen analyysin mestarikurssi (Markku Halmetoja ja Jorma Merikoski)

Ensiaskeleet Einsteinin avaruusaikaan, osa 1: Kinematiikka: aika, paikka ja liike (Teuvo Laurinolli)

Ensiaskeleet Einsteinin avaruusaikaan, osa 2: Dynamiikka: liikelait, liikemäärä ja energia (Teuvo Laurinolli)

Kilpailumatematiikan opas (Matti Lehtinen)

Geometrian perusteita (Matti Lehtinen)

Geometria (K. Väisälä)

Lukualueiden laajentamisesta (Tuomas Korppi)

Jaksolliset desimaaliesitykset algebrallisesta näkökulmasta (Jaska Poranen ja Pentti Haukkanen)

Algebra (Tauno Metsänkylä ja Marjatta Näätänen)

Algebra (K. Väisälä)

Matemaattista fysiikkaa lukiolaiselle 1: Mekaniikkaa (Markku Halmetoja ja Jorma Merikoski)

Matemaattista fysiikkaa lukiolaiselle 2: Sähköoppia (Markku Halmetoja ja Jorma Merikoski)

Lukuteorian helmiä lukiolaisille (Jukka Pihko)

Matematiikan peruskäsitteiden historia (Erkki Luoma-aho)

Matematiikan historia (Matti Lehtinen)

Reaalianalyysiä englanniksi (William Trench)