Al-Khwarizmin perintö - mitä on algoritmitutkimus?

Esko Ukkonen
Helsingin yliopisto

April 1, 1996

1 Johdanto

Algoritmien ja laskennallisen kompleksisuuden tutkimus on tietojenkäsittelytieteen päävirtauksia. Sillä on kahdenlaisia tavoitteita. Konstruktiivisena tavoitteena on kehittää erilaisille tietojenkäsittelyongelmille tehokkaita ratkaisualgoritmeja. Analyyttisenä tavoitteena on arvioida eri algoritmien vaatimia laskentaresursseja kuten laskenta-aikaa ja muistitilaa sekä sitä, onko jokin algoritmi tehokkain mahdollinen ongelman ratkaisu.

Algoritmien optimaalista tehokkuutta tarkasteltaessa tulee tietojenkäsittelyongelmista analyysin varsinaisia kohteita. Tällöin puhutaan vaativuus- tai kompleksisuusteoriasta. Ongelman analysointi verrattuna yksittäisen algoritmin analyysiin on oleellisesti vaikeampaa, koska silloin pyritään arvioimaan ongelman kaikkien mahdollisten ratkaisualgoritmien käyttäytymistä. Toisaalta kysymyksenasettelutkin ovat kiintoisia: tavoitteena on ymmärtää, miksi joidenkin ongelmien algoritminen ratkaisu vaatii radikaalisti enemmän laskentaresursseja kuin joidenkin toisten ongelmien ratkaisu.

Tässä kirjoituksessa esitetään ensin pieni katsaus algoritmitutkimuksen historiaan, minkä jälkeen esitellään tutkimusalan peruskäsitteitä ja ongelmanasetteluja. Lopussa on joitakin huomautuksia algoritmitutkimuksen luonteesta ja kehitysnäkymistä.

2 Eukleides, al-Khwarizmi, Babbage

Vanhin tunnettu nimenomaan algoritmiksi kutsuttu algoritmi lienee Eukleideen nimiin laitettu menetelmä kahden luvun suurimman yhteisen tekijän määräämiseksi.

Sana *algoritmi* tulee kuitenkin persialais-arabialaisen matemaatikon al-Khwarizmin ("Khwarizmilainen") nimestä. Al-Khwarizmi, jonka täydellinen nimi oli Muhammed ibn Musa Abu Abdullah al-Khwarizmi al-Madjusi al-Qutrubulli, eli noin vuosina 780-850. Hän syntyi Aral-järven itäpuolella sijainneessa muinaisessa Khwarizmissa. Elämäntyönsä hän kuitenkin teki Bagdadissa, "Viisauden taloksi" kutsutun tutkimuslaitoksen palveluksessa. Laitosta ylläpiti kalifi al-Mamun, joka oli Tuhannen ja yhden yön kertomuksista tunnetun kalifi Harun ar-Rashidin poika.

Al-Khwarizmi osallistui intialaisen matematiikan välittämiseen länsimaihin. Sen keskeistä sisältöä olivat desimaalijärjestelmän käyttö lukujen merkitsemiseen, aritmetiikka ja trigonometria. Näitä kaikkia al-Khwarizmi käsitteli kirjoituksissaan, jotka sisältävät esimerkiksi myös hyvin selkeitä algoritminomaisia menetelmiä toisen asteen yhtälöiden ratkaisemiseksi. On kuitenkin epäselvää, mikä on intialaista perintöä ja missä näkyy al-Khwarizmin oma luova panos.

Keskiajan matematiikassa al-Khwarizmi eli "Algorismus" näkyy koulukuntajaossa abakistit - algoristit. Edelliset käyttivät laskuapuna helmitaulua, jälkimmäiset turvautuivat desimaalijärjestelmään ja aritmeettisiin menetelmiin.

Myöhempää kehitystä kuvattaessa on erityisesti mainittava englantilainen Charles Babbage (1792-1871). Hän oli aikaansa edellä oleva näkijä, jonka suunnittelema "analyyttinen kone" on nykyisten tietokoneiden varhainen edeltäjä. Analyyttinen kone, joka tosin ei koskaan valmistunut, oli suunniteltu kutomakoneen inspiroimana. Sen vallankumouksellinen piirre oli ohjelmoitavuus. Ohjelma oli tarkoitus antaa reikäkorteilla. Samaan tapaan kuin reikäkortit ohjasivat kutomakonetta, piti korttien ohjata myös analyyttisen koneen matemaattisia toimintoja. Vaihtamalla kortit sama kone saataisiin suorittamaan erilaisia ohjelmia kuten kutomakonekin pystyi tuottamaan erikuosisia kankaita.

Vaikka kone jäi rakentamatta, Babbage tajusi kirkkaasti ideansa merkityksen. Nykyisen algoritmitutkimuksen huoneentauluksi sopiikin Babbagen omaelämäkerrassaan hiukkasen mahtipontisesti muotoilema arvio: *As soon as an Analytical Engine exists, it will necessarily guide the course of the sci-*

ence. Whenever any result is sought by its aid, the question will then arise - By what course of calculation can these results be arrived at by the machine in the shortest time?

Algoritmikäsitteen matemaattisesti täsmällinen määrittely tehtiin 1930-luvulla. Näistä erästä, Turingin konetta, käsitellään tässä kirjoituksessa vielä tarkemmin. Turingin koneen ohella kehitettiin useita muita määrittelytapoja, jotka osoittautuivat keskenään samanarvoisiksi.

Elektronisen tietokoneen synty 1940-luvulla ja sitä seurannut tietokonelaitteistojen ripeä kehitys johti algoritmien kysynnän räjähdysmäiseen kasvuun. Yhtäkkiä algoritmitutkimus oli sovelluskelpoista, olihan onnistuttu rakentamaan laite, joka pystyi ainakin periaatteessa suorittamaan kuinka työläitä ja monimutkaisia algoritmeja tahansa.

Samalla alkoi laaja algoritmisen ajattelun läpimurto kaikkialla tutkimuksessa ja tekniikassa. Tämä on johtanut siihen, että yhä useammilla aloilla tavoitteeksi on tullut tiedon saattaminen algoritmeiksi tai ainakin algoritmisesti käsiteltävään muotoon. Sen, missä määrin tässä onnistutaan, voi katsoa monessa tapauksessa jokseenkin suoraan mittaavan asianomaisen alan saavuttamaa tiedon syvyyttä.

3 Algoritmin käsite ja Turingin kone

Koulusivistykseen kuuluu useita algoritmeja. Hyvä esimerkki on moninumeroisten lukujen kertolaskun suorittamiseksi opittu menettely. Jos on laskettava '49 kertaa 18', muodostuu tämän algoritmin kirjanpito seuraavaksi:

$$\begin{array}{r} 18 \\ 49 \\ \hline 162 \\ 720 \\ \hline 882 \end{array}$$

Ensin 18 kerrotaan kertojan 49 ykkösillä, siis 9:llä, mistä saadaan 162, ja sitten kymmenillä, mistä saadaan 72 (oikeastaan 720, mutta 0 on tapana jättää kirjoittamatta). Lopuksi lasketaan välitulokset yhteen. Näin on syötteistä 49 ja 18 saatu tuloste 882.

Yleisesti algoritmeja voi luonnehtia toteamalla, että kukin algoritmi määrittelee äärellisen, täsmällisen operaatiojonon, joka tuottaa algoritmin saamista syötteistä halutut tulosteet. Jonon kunkin operaation täytyy olla äärellisessä ajassa suoritettavissa. Algoritmin määrittelemää toimintaa voi ajatella symbolien manipulointina, jonka voi periaatteessa aina suorittaa täysin tarkasti käsityönä äärellisessä ajassa kynällä ja paperilla.

Sama ongelma voidaan usein ratkaista monella eri algoritmilla, jotka saattavat poiketa syvällisesti toisistaan. Esimerkiksi kertolaskua varten tunnetaan myös 'venäläinen' algoritmi. Edellisen esimerkin tapauksessa sen kirjanpito näyttäisi seuraavalta:

$$
\begin{array}{rrl}
49 & 18 & + \\
24 & 36 & \\
12 & 72 & \\
6 & 144 & \\
3 & 288 & + \\
1 & 576 & + \\
\hline
 & 882 &
\end{array}
$$

Ensimmäisellä sarakkeella olevat luvut on saatu jakamalla kertoja 49 toistuvasti luvulla 2 ja ottamalla tuloksen kokonaisosa. Toisella sarakkeella olevat luvut on puolestaan saatu kertomalla kerrottava 18 toistuvasti luvulla 2. Lopputulos 882 saadaan laskemalla yhteen ne kerrottava-sarakkeen luvut (merkitty +:lla), joita vastaa kertoja-sarakkeella pariton luku. Tämä algoritmi, joka käyttää perusoperaationaan luvulla 2 jakamista ja kertomista sekä yhteenlaskua, sopii hyvin tietokoneille, koska ne käyttävät 2-kantaista lukujärjestelmää.

Algoritmi saa täsmällisen muodon kun se kirjoitetaan ohjelmaksi jollakin ohjelmointikielellä. Ohjelmointikielten joukko on tunnetusti laaja ja koko ajan kasvava. Eri kielet voivat poiketa ulkoasunsa, perusoperaatioidensa ja kieleen liittyvän ohjelmointiajattelun osalta erittäin syvällisestikin toisistaan; eroilla on käytännöllistä merkitystä lähinnä siinä, miten helppoa ja luontevaa erityyppisten tietojenkäsittelymenetelmien ja algoritmien muotoilu kussakin kielessä on. Kielten teoreettinen ilmaisuvoima on yleensä kuitenkin sama: kaikilla pystytään lopulta laskemaan täsmälleen samat asiat.

Koska kielet ovat tässä mielessä samanarvoisia, algoritmiteoreettisten tarkastelujen perustana kannattaa usein käyttää mahdollisimman primitiivistä ohjelmointikieltä, jolla ohjelmoidaan äärimmilleen pelkistettyä abstraktia tietokoneen mallia. Turingin kone ja sen ohjelmointikieli sopivat tähän tarkoitukseen.

Yksinkertaisin Turingin kone käyttää muisti-, syöttö- ja tulostuslaitteena yhtä “nauhaa”. Nauha on jaettu muistipaikkoihin. Näitä on rajattoman paljon, koska nauhan ajatellaan jatkuvan äärettömiin. Kussakin paikassa voi olla talletettuna yksi merkki, tai paikka voi olla tyhjä. Koneessa on lisäksi tietysti “prosessori”, jolla voidaan käsitellä nauhan sisältöä.

Käsittely tapahtuu koneeseen kuuluvan ohjelman mukaan. Prosessori pystyy kunakin ajanhetkenä tarkastelemaan yhtä nauhapaikkaa. Yhdessä ohjelma-askeleessa prosessori voi ensin muuttaa tarkastelemansa paikan sisällön kirjoittamalla sinne jonkin merkin ja siirtyä sitten tarkastelemaan välittömästi vasemmalla tai oikealla puolella olevaa nauhapaikkaa. Lisäksi prosessorilla on oma sisäinen äärellistilainen muisti, jonka sisältöä eli prosessorin “tilaa” ohjelma-askel voi myös muuttaa. Koneen koko toiminta muodostuu kuvatuista hyvin yksikertaisista askelista. Toiminnan alkaessa koneen saama syöte on kirjoitettu nauhan alussa oleviin peräkkäisiin muistipaikkoihin, toiminnan päätyessä koneen antama tuloste voidaan lukea nauhan muistipaikoista.

Kuvassa 1 on eräs Turingin kone. Se tarkastaa, onko syötteenä annetussa bittijonossa 0010110 parillinen vai pariton määrä merkkejä 0. Ohjelman ensimmäinen lause (A,0) $\rightarrow$ (B,0,R) voitaisiin lukea: “Jos kone on tilassa A ja tarkasteltu nauhapaikka sisältää merkin 0, siirry tilaan B, kirjoita nauhapaikkaan 0 ja siirry tarkastelemaan seuraavana oikealla puolella olevaa nauhapaikkaa." Kuvan tilanteessa tämä askel otettaisiin seuraavaksi, koska koneen tila todella on A ja tarkasteltu nauhamerkki 0.

On selvää, että Turingin kone on teoreettinen konstruktio, jolla ei ole varsinaista käytännön merkitystä. Edellä kuvattujen kertolaskualgoritmien ohjelmoiminen Turingin koneeksi olisi kyllä mahdollista, mutta jokseenkin työlästä. Se kelpaa kuitenkin algoritmikäsitteen matemaattiseen määrittelyyn: algoritmisesti laskettavissa olevaa on täsmälleen se mitä pystytään laskemaan Turingin koneilla.

Onko näin asetettu laskettavuuden määritelmä oikein? Toisin sanoen, vastaako Turingin koneina formalisoitu laskettavuus intuitiivista, esimerkiksi kynällä ja paperilla tapahtuvaan symbolien käsittelyyn tai mikroprosessoreiden ominaisuuksiin perustuvaa käsitystämme algoritmisista menetelmistä? *Church - Turing teesi* lausuu sen, että näin on. Teesiä ei tietenkään voi matemaattisesti todistaa. Käytännön kokemus kuitenkin tukee sitä. Tämä voi tuntua yllättävältä, merkitseehän teesi esimerkiksi sitä, että kehittyneimmänkin digitaalisen tietokoneen toimintaa voidaan tietyssä mielessä tarkasti simuloida yksinkertaisella Turingin koneella.

Algoritmikäsitteen vaihtoehtoisista formalisointitavoista Turingin kone ei ehkä ole elegantein. Algoritmitutkimuksen kannalta se on kuitenkin käyttökelpoinen, koska siinä on luonnolliset mitat laskennan vaatimille resursseille: laskenta-ajan mittana voidaan käyttää laskennan ottamien ohjelma-askelien lukumäärää, tilan mittana niiden muistipaikkojen määrää, joiden kautta laskenta on kulkenut. Usein tarvitaan kuitenkin kehittyneempiä malleja, koska Turingin kone on liian kaukana todellisista tietokoneista. Näiden laskenta-askeleet vastaavat paremmin oikeiden tietokoneiden perusoperaatioita.

4 Algoritmien analysointi

Miten keksitään hyviä algoritmeja? Kaiken kattavia suunnittelusääntöjä on arvatenkin mahdotonta antaa, siinä määrin yksilöllisiä monet merkittävät algoritmit ovat. Joitakin hyviksi havaittuja periaateita on kuitenkin eristetty. Näitä ovat muiden muassa hajoita ja hallitse -tekniikaksi kutsuttu menettely ja dynaaminen ohjelmointi. Esimerkiksi hajoita ja hallitse -algoritmit jakavat ratkaistavan tehtävän keskenään samanlaisiin osatehtäviin, jotka ratkaistaan erikseen ja osaratkaisuista kootaan kokonaisratkaisu. Tietorakenteiden kehittely on usein oleellinen osa hyvän algoritmin muodostamista.

Kun algoritmi on konstruoitu, teoreettisen analyysin keinoin pyritään arvioimaan sen tehokkuutta, lähinnä algoritmin toiminnan vaatimaa aikaa ja muistitilaa. Tehtävä voi tuntua hankalalta: on ilmeistä, että sama algoritmi voi käyttäytyä eri syötteillä hyvinkin eri tavoin. Tilannetta on sopivasti yksinkertaistettava. Vakiintunut tapa on arvioida resurssivaatimuksia algoritmin syötteen pituuden n funktiona. Kaikki saman n pituiset syötteet arvioidaan siis yhdessä. Yleensä tarkastellaan, miten resurssivaatimukset pahimmassa tapauksessa kasvavat, kun syötteen pituus n kasvaa rajatta.

Tällaisen *pahimman tapauksen asymptoottisen analyysin* ratkaisevana etuna on analyysituloksen riippumattomuus algoritmia suorittavan tietokoneen nopeudesta ja toisarvoisista yksityiskohdista. Myös analyysilaskut tulevat suhteellisen yksinkertaisiksi.

Haittana voi olla epätarkkuus: jos pahimman tapauksen antava syöte on käytännössä harvinainen ja keskimäärin algoritmi käyttäytyy huomattavasti pahinta tapausta tehokkaammin, analyysillä on saatu liian huono kuva. Asymptoottisen arvion ilmaisema käyttäytyminen voi myös tulla esiin vasta niin suurilla arvoilla n, että ne eivät tule käytännössä kyseeseen. Silloin algoritmien vertailu vain asymptoottisten arvioiden perusteella voi johtaa harhaan.

Palataan kertolaskun peruskoulualgoritmiin. Kun sillä kerrotaan kaksi n-numeroista lukua, ei ole vaikea osoittaa, että laskenta-ajan tarve kasvaa pituuden n kasvaessa kuten n^2. Silloin algoritmin asymptoottista aikavaatimusta merkitään $O(n^2)$. Tällä kertaa analyysin tulos on varsin tarkka, koska pahimmalla ja parhaalla tapauksella ei ole suurta eroa.

Tunnetaan myös hajoita ja hallitse -tyyppinen kertolaskualgoritmi, jonka aikavaatimus on $O(n^{1.59})$, siis edellistä pienempi. Asymptoottisesta paremmuudestaan huolimatta tämä algoritmi on siinä määrin monimutkainen, että eräässä kokeessa se saatiin toimimaan peruskoulualgoritmia nopeammin vasta kun n oli yli 600! Syynä on monimutkaisuuden aiheuttama yleisrasite, joka dominoi kun n on pieni. Algoritmilla näyttäisi kuitenkin olevan käyttöä yli 600-numeroisten lukujen kertolaskua tarvitsevissa erikoissovelluksissa.

Ei ole aivan harvinaista, että asymptoottiset parannukset saadaan aikaan algoritmeilla, joiden käytännön arvo tällä hetkellä on rajoittunut. Kuitenkin niillä on pysyvää merkitystä. Onhan mahdollista, että tulevaisuuden tietokoneissa algoritmi voidaan toteuttaa niin, että se pääsee oikeuksiinsa jo kun n on pieni. Toisaalta uusissa sovelluksissa yhä suuremmat n voivat olla tarpeen.

5 Optimaaliset algoritmit

Algoritmitutkimuksen eräs luonnollinen tavoite on selvittää, onko ongelman annettu ratkaisualgoritmi jossakin mielessä paras mahdollinen.

Tarkastellaan esimerkkinä etsintäongelmaa. On annettu pakka nimikortteja. Pakasta on löydettävä tietty kortti, esimerkiksi Hannes Hemulin kortti. Etsintä voidaan hoitaa ilmeisellä tavalla peräkkäisetsintänä seuraavasti: Tutkitaan kortit yksitellen siinä järjestyksesä kuin ne ovat pakassa. Kun Hannes Hemuli löytyy tai kaikki kortit on tuloksetta katsottu, tehtävä on suoritettu. Aikaa menee pahimmillaan $O(n)$, jos n on korttien lukumäärä. Kaikki n korttia tulevat nimittäin katsotuiksi ainakin siinä tapauksessa, että Hannes Hemulin korttia ei ole lainkaan pakassa.

Toisaalta tehtävästä ei voi pahimmassa tapauksessa selvitä nopeamminkaan kuin $O(n)$ askeleessa. Jos yksikin kortti jätetään katsomatta, juuri se voi olla etsitty Hannes Hemuli. Peräkkäisetsintä on tässä mielessä optimaalinen.

Näin hidas etsintämenetelmä on tarpeen kuitenkin vain silloin kun korttipakka on sekaisin. Jos kortit ovat järjestettynä aakkosjärjestykseen, huomattavasti nopeampi algoritmi tulee mahdolliseksi. Silloin soveltuu binääriseksi etsinnäksi sanottu menetelmä.

Binäärisessä etsinnässä katsotaan ensin pakan puolivälissä oleva kortti. Jos se on Hannes Hemulin, etsintä on valmis. Muuten jos Hannes Hemuli on aakkosjärjestyksessä ennen kortissa olevaa nimeä, jatketaan etsintää pakan alkupuoliskosta käyttäen samaa puolittamisperiaatetta, muuten jatketaan pakan loppupuoliskosta. Jokainen kortin tutkiminen puolittaa sen pakan osan, jossa Hannes Hemulin kortti voi olla. Vertailuja tarvitaan näin ollen korkeintaan korttien määrän logaritmin verran eli $O(\log n)$ kappaletta. Tämän ylärajan rinnalle voidaan asettaa alarajatulos: mikä tahansa algoritmi, joka etsii lajitellusta pakasta, joutuu tekemään pahimmassa tapauksessa ainakin $O(\log n)$ vertailua. Binäärinen etsintä on siis optimaalinen.

Näiden lupaavien esimerkkien jälkeen on huomautettava, että yleisesti asiain tila ei ole yhtä valoisa. Ei-triviaalien alarajojen todistaminen mielenkiintoisten ongelmien algoritmisen ratkaisun resurssivaatimuksille on osoittautunut usein erittäin vaikeaksi, ja monien tärkeiden ongelmien ratkaisualgoritmien optimaalisuutta ei ole pystytty lainkaan osoittamaan.

6 Ratkeavuus periaatteessa ja käytännössä

Algoritminen ratkeamattomuus on algoritmikäsitteen formalisoinnin yhteydessä paljastuneita ilmiöitä. Tunnetaan runsaasti ongelmia, joista voidaan osoittaa, että niiden ratkaiseminen algoritmilla on mahdotonta. Pysähtyvyysongelma on näistä tunnetuimpia. Tehtävänä on selvittää pysähtyykö annetun algoritmin tai tietokoneohjelman suoritus aina vai onko mahdollista että suoritus jää silmukkaan jatkuen loputtomasti. Ongelman ratkaisevaa yleistä algoritmia ei voi olla. Sivuhuomautuksena todettakoon, tämä tulos asettaa vahvoja periaatteellisia rajoituksia mahdollisuuksille kehittää automaattisia ohjelmoinnin apuvälineitä.

Käytännön tarpeisiin ei kuitenkaan pelkkä algoritmin olemassaolo riitä. Algoritmin on oltava myös käyttökelpoisen tehokas. Esimerkiksi sopii seuraava helposti syntyvä "kombinatorisen räjähdyksen" sisältävä tilanne. Tarkastellaan algoritmia, jonka suoritusaika kasvaa kuten 2^n, missä n on jälleen syötteen pituus. Voidaan yksinkertaisesti ajatella, että n merkkiä pitkän syötteen käsittely vaatii 2^n mikrosekuntia. Tämä algoritmi ehtii silloin yhden vuorokauden aikana käsitellä syötteen, jonka pituus on vaatimattomat 32. Entä jos aikaa on käytettävissä runsaasti? Siinä ajassa, joka on kulunut maailmankaikkeuden arvioidusta alkuräjähdyksestä, algoritmi olisi ehtinyt käsitellä syötteen, jonka pituus on 70!

Esimerkin sanoma on selvä. Tällainen eksponentiaalinen algoritmi on käyttökelvoton lukuunottamatta aivan lyhyitä syötteitä. Voidaan tosin ajatella että ongelma ratkeaisi tulevaisuuden hypernopeilla ultrarinnakkaisilla superkomputaattoreilla. Näin ei käy: jos esimerkkialgoritmimme suoritettaisiin koneella, joka olisi 1 000 000 kertaa nopeampi, kasvaisivat edellä annetut pituudet 32 ja 70 vain 20:llä, ja jos tällaisia koneita olisi 1 000 000 kappaletta rinnakkaisesti suorittamassa algoritmiamme, kasvaisivat pituudet parhaimmillaankin jälleen vain 20:llä. Koneiden nopeuttaminen ja niiden kytkeminen rinnakkain eivät auta oleellisesti.

Käytännössä ratkeavina pidetäänkin yleensä vain ongelmia, joilla on polynomisessa ajassa toimiva algoritmi. Tämä tarkoittaa, että algoritmin vaatima toiminta-aika on korkeintaan jokin syötteen pituuden polynomi. Näin lausuttu yleinen periaate ei tosin ole ilman poikkeuksia. Jos syötteen koko on pieni, voi eksponentiaalinenkin algoritmi riittää. Toisaalta on sovelluksia, joissa ajankäytön polynomisuus ei saa olla liian korkea-asteista. Reaaliaikaiset sovellukset voivat vaatia ajassa $O(n)$ toimivaa algoritmia, $O(n^2)$

saattaa olla jo liian hidas.

7 NP-täydellisyys

Algoritmitutkimuksen ja kompleksisuusteorian kuuluisin avoin kysymys liittyy NP-täydellisyyteen. Kysymys esitetään yleensä muodossa: Onko $P = NP$? Tässä P tarkoittaa käytännössä ratkeavien ongelmien luokkaa, siis ongelmia, joilla on polynominen ratkaisualgoritmi. Luokka NP sisältää muiden ohella myös kaikki NP-täydelliset ongelmat. Kysymys on siitä, voidaanko NP-täydelliset ongelmat ratkaista polynomisessa ajassa. Pian kaksi vuosikymmentä jatkuneesta intensiivisestä tutkimuksesta huolimatta kysymys onko $P = NP$ pysyy sitkeästi avoimena.

NP-täydellisiä ongelmia tunnetaan runsaasti. Useat ovat tärkeitä monissa sovelluksissa, esimerkiksi n.s. kauppamatkustajan ongelma on tällainen. Tarkastellaan esimerkkinä osasummaongelmaa.

Osasummaongelmassa on annettu mielivaltainen joukko kokonaislukuja, vaikkapa luvut

$$15, 27, 16, 8, 24, 12, 19, 3, 31, 45, 33,$$

ja lisäksi erityinen luku, esimerkiksi 50. Tehtävänä on selvittää, sisältääkö lukujoukko osajoukon, jonka lukujen summa on yhtä suuri kuin erityinen luku. Esimerkkitapauksessa olisi päätettävä, voidaanko annetusta joukosta poimia luvut, joiden summa on 50.

Osasummaongelmalle on helppo keksiä ratkaisualgoritmi. Poimitaan systemaattisesti vuoron perään kaikki lukujoukon osajoukot ja testataan, onko osajoukon summa yhtä suuri kuin erityinen luku. Esimerkkitapauksessa löytyisivät näin jossakin vaiheessa luvut 16, 3 ja 31, jotka antavat myönteisen ratkaisun, koska $16 + 3 + 31 = 50$. Algoritmi on kuitenkin eksponentiaalinen: jos lukuja on n kappaletta, erilaisia osajoukkoja eli ratkaisuehdokkaita on 2^n, joten aikaa menee pahimmillaan ainakin $O(2^n)$. Oleellisesti tehokkaampaa algoritmia ei tunneta, mutta liioin ei ole kyetty osoittamaan etteikö sellaista voisi olla.

Osasummaongelma on NP-täydellinen. Sillä on seuraava kaikille NP-täydellisille ongelmille luonteenomainen piirre: ratkaisun löytäminen on vaikeaa, mutta annetun ratkaisuehdokkaan tarkistaminen on helppoa. Esimerkkitapauksessa luvut 16, 3 ja 31 muodostavat erään ratkaisun. Vaikeus on siinä, miten juuri nämä suotuisat luvut löydetään nopeasti. Ainoa mahdollisuus näyttää olevan aikaavievä kaikkien eri vaihtoehtojen läpikäynti. Annetun ratkaisuehdokkaan tarkistus sensijaan sujuu nopeasti. Jos joku väittää, että ongelmalla on myönteinen ratkaisu ja antaa väitteensä tueksi ratkaisuehdokkaan, ehdokkaan luvut vain lasketaan yhteen ja katsotaan tuliko 50.

Toinen NP-täydellisten ongelmien oleellinen ominaisuus on, että ne ovat tavallaan yhtä vaikeita. Kaikki NP-täydelliset ongelmat nimittäin palautuvat toisiinsa. Jos yhdellekin tunnettaisiin tehokas algoritmi, sen avulla voitaisiin ratkaista tehokkasti myös kaikki muut. Algoritmin avulla voitaisiin myös ratkaista kaikki NP-täydellisiä helpommat ongelmat. NP-täydelliset ongelmat ovat tässä mielessä täydellisen vaikeita ongelmia luokassaan.

Yleensä arvellaan, että luokka NP ei sisälly luokkaan P eli yhtäpitävästi, että osasummaongelmalla tai millään muulla NP-täydellisellä ongelmalla ei ole polynomisessa ajassa toimivaa algoritmia. Jos tämä arvelu osoittautuu oikeaksi, ei asian lopullisella todistuksella välttämättä ole suurta vaikutusta, vastaisihan tulos odotuksia. Jos toisaalta osoittautuisi että $P = NP$, laajentuisi käytännössä ratkeavien ongelmien joukko merkittävästi, millä voisi olla selviä taloudellisiakin seurauksia.

8 Vaikeiden ongelmien käsittely

Miten tullaan käytännössä toimeen laskennallisesti vaikeiden ongelmien kanssa? Näitä ongelmia, joille ei tunneta polynomista algoritmia, joudutaan jatkuvasti ratkaisemaan eri yhteyksissä, vieläpä varsin hyvällä menestyksellä.

Kuten edellä jo mainittiin, eksponentiaalinenkin algoritmi voi olla käyttökelpoinen, jos laskenta-ajan kombinatoriseen räjähdykseen johtavat syötteet ovat algoritmin käyttöympäristössä harvinaisia ja tavallisesti algoritmi on nopea. Tyyppiesimerkki tällaisesta algoritmista on lineaarisen ohjelmoinnin tunnettu Simplex-algoritmi. Samalle ongelmalle löydettiin jo yli 10 vuotta sitten huomattavan kohun saattelemana myös polynominen menetelmä, joka kuitenkaan ei ole pystynyt syrjäyttämään Simplex-algoritmia.

Toinen laajassa käytössä oleva tapa ratkoa laskennallisesti vaikeita ongelmia on tyytyä tarkan ratkaisun sijasta likiarvoon. Ongelma on usein optimointitehtävä: on löydettävä jonkin suureen maksimi tai minimi. Käytännössä saattaa hyvin riittää, että löydetään optimin likiarvo. Likiarvo saattaa löytyä nopealla algoritmilla. Tosin on osoittautunut, että monessa tapauksessa hyvää likiarvoa ei ole sen helpompi löytää kuin tarkkaa ratkaisuakaan. Siksi on ymmärrettävää, että optimointiongelmien ratkaisemisessa on vallalla varsin anarkistinen tila: likimääräisiä ratkaisuja etsitään algoritmeilla, jotka perustuvat erilaisiin huonosti ymmärrettyihin peukalosääntöihin. Ne ovat nopeita ja näyttävät toimivan luotettavan tarkasti, mutta kunnollista laatutakuuta ratkaisun tarkkuudelle ei yleensä tunneta.

Toisaalta on huomattava, että laskennallisesti vaikeista ongelmista ei ole pelkästään harmia. Niillä on hyötykäyttöä salakirjoituksessa ja tiedonsuojausmenetelmissä, jotka perustuvat yksisuuntaisiksi funktioiksi kutsuttujen laskentaongelmien käyttöön. Funktion arvon laskenta on vaikeaa, mutta sen käänteisfunktion laskenta on helppoa.

9 Algoritmikäsitteen laajennuksia

Laskennallisen kompleksisuuden voittamiseksi on myös etsitty uudentyyppisiä, rinnakkaisuutta ja satunnaisuutta sisältäviä algoritmeja.

Tavallinen algoritmi on peräkkäisalgoritmi, jonka suorittajana on yksi suoritin eli prosessori. Yhteen suorittimeen perustuvassa tietokonearkkitehtuurissa suorittimen ja muistin välinen tietoliikenne muodostaa kokonaisnopeutta rajoittavan ahtaumakohdan, “von Neumanin pullonkaulan”.

Pullonkaula avartuu, kun otetaan käyttöön useita rinnakkaisia suorittimia. Algoritmien kehittäminen tällaiselle yhteistä muistia käyttävälle moniprosessorikoneelle on osoittautunut haastavaksi. Yhteisen algoritmin jako rinnakkaisille suorittimille ja muistinkäytön organisointi aiheuttavat yleisrasitetta, johon rinnakkaisuudella saavutettu lisänopeus voi osittain haaskaantua. Yksinkertaisiin vektorointi- ja liukuhihnaperiaatteisiin perustuvaa rinnakkaisuutta käytetään menestyksellisesti supertietokoneissa. Teorian kehittelyn puolella tutkimus ei ole vielä päässyt yksimielisyyteen rinnakkaislaskennan formaalista mallista. Yhtenäisen mallin puuttuminen hidastaa algoritmiteoreettista kehitystä.

Jos eri prosessoreilla ei ole yhteistä muistia ja kukin prosessori toimii itsenäisesti omaan tahtiinsa, joudutaan ratkaisemaan epäsynkronisten moniprosessori- ja tietokoneverkkojen ja hajautettujen algoritmien ongelmia. Perustehtävänä on järjestää eri suorittimien välille niiden yhteistyön mahdollistava kommunikointi. Tällöin syntyy uudentyyppisiä algoritmisia ongelmia, esimerkiksi pitää varautua verkossa olevien suorittimien tilapäiseen epäkuntoisuuteen. Paitsi tehokkuuden, jopa algoritmien oikeellisuuden ja luotettavuuden saavuttaminen on vaikeaa.

Suoraan elektronisina piireinä toteutettavat algoritmit sisältävät runsaasti alkeellisia prosessoreja käyttävää synkronista rinnakkaisuutta. Integroitujen piirien kehittyessä yhä useampia algoritmeja tullaan toteuttamaan tällä tasolla. Kytkentäpiirien algoritmiteoria liittyy luontevasti peräkkäisalgoritmien teoriaan.

Satunnaisuutta sisältävät algoritmit käyttävät satunnaislukuja. Havainnollisesti voidaan ajatella, että algoritmit perustavat jotkin toimintapäätöksensä lantinheittoon! Perinteinen algoritmi on deterministinen: samoilla syötteillä algoritmi tekee aina täsmälleen samat laskenta-askeleet ja tuottaa samat tulokset. Satunnaisalgoritmin toimintaan vaikuttaa se, mitä lukuja satunnaislukulähde on sattunut algoritmin toimiessa antamaan. Jos tällainen algoritmi toistetaan samoilla syötteillä, on mahdollista että algoritmi suorittaa eri kerroilla eri askeleet ja saa myös eri lopputulokset. Satunnaislukujen käyttö tehokkaiden algoritmien kehittämisessä perustuu siihen, että niiden avulla voidaan organisoida onnekkaita arvauksia, jotka johtavat algoritmin nopealle suoritusreitille.

Tarkastellaan esimerkkinä erästä reititysalgoritmia. Prosessoriverkon prosessorit lähettävät viestejä toisilleen verkkoa pitkin. Lähettäjäprosessori liittää viestiin osoitteeksi vastaanottajaprosessorin nimen, ja verkko huolehtii viestin viemisestä perille. Verkon ruuhkien tasaamiseksi sopii seuraava satunnaisalgoritmi: kun viesti lähtee, sille arvotaan satunnaislukujen avulla väliaikainen osoite, joka voi olla mikä tahansa verkon prosessoreista. Sen jälkeen viesti kulkee ensin väliaikaiseen osoitteeseen ja vasta sieltä varsinaiseen osoitteeseensa. Tämä yksinkertainen menettely purkaa ruuhkia ja tasaa kuormitusta tehokkaasti.

Erään luokituksen mukaan satunnaisalgoritmit jakautuvat Sherwood-, Monte Carlo ja Las Vegas -algoritmeihin. Sherwood-algoritmin lähtökohtana on tavallinen deterministinen algoritmi. Ideana on satunnaislukujen avulla tasoittaa lähtöalgoritmin pahinta tapausta. Tuloksena on algoritmi, joka millä tahansa syötteellä vaatii suurella todennäköisyydellä sen ajan minkä alkuperäinen algoritmi vaati keskimäärin.

Monte Carlo -algoritmissa sallitaan virheellinen tulos. Algoritmit ovat aina nopeita mutta tulos voi olla väärin. Virheen todennäköisyys voidaan kuitenkin tehdä mielivaltaisen pieneksi. Las Vegas -algoritmit antavat aina oikean tuloksen, mutta ovat vain todennäköisesti nopeita. Pienellä todennäköisyydellä laskenta voi olla pitkä.

Satunnaisalgoritmien tutkimus jatkuu vilkkaana. Alaa rikastuttavat monet yhteydet eri tahoille kuten lukuteoriaan, kryptologiaan ja tiedonsuojaukseen sekä hajautettuihin algoritmeihin. Merkittäviin sovelluksiin johtavia tuloksia voi tältä suunnalta myös odottaa.

10 Lopuksi

Algoritmitutkimuksen arvomaailma muotoutui ainakin tavallisten peräkkäisalgoritmien osalta 1970-luvulla. Monien uudentyyppisten menetelmien, kuten rinnakkais- ja satunnaisalgoritmien, tutkimus kuplii kuitenkin edelleen varsin vapaasti, ja työtä on runsaasti jäljellä. Algoritmitutkimuksen asema on erityisen vahva amerikkalaisessa tietojenkäsittelytieteessä, mutta myös Suomessa on ylletty kansainvälisen tason tuloksiin.

Algoritmianalyysi ja kompleksisuusteoria vaatii täsmällistä matemaattista tutkimusotetta. Monesti tosin riittää varsin rutiininomainen matematiikan käyttö. Kuitenkin uusin algoritmisten ideoiden kehitys ja monet sitkeästi avoimina säilyvät vanhat ongelmat viittaavat siihen, että algoritmitutkija tarvitsee yhä laajempia matematiikan tietoja.

Algoritmitutkimuksen viehättävimpiä piirteitä on lyhyt ja mutkaton yhteys sovelluksiin. Yhteyden vaaliminen, mikä käytännössä tarkoittaa algoritmien kehittämistä ja analysointia konkreettisen ohjelmiston rakentamisen yhteydessä, on alan elinvoimaisuuden perusedellytys.

Kirjallisuutta: D. Harel: Algorithmics - The Spirit of Computing. Addison-Wesley 1987.