8. Rinnakkaisalgoritmit

(Harel, luvun 10 sivut 265-281.)

"Monet harhailevat, mutta tieto lisääntyy"
[Daniel 12:4]

"Kaksin on parempi kuin yksin, sillä kumpikin saa vaivoistaan hyvän palkan" [Saarnaaja 4:9]

- Kuinka prosessorien lisääminen vaikuttaa algoritmiseen laskentavoimaan?

Rinnakkaisuuden edut ja rajat

Rinnakkaisuudesta eli samanaikaisesta yhteistoiminnasta on selvästi etua.

Esim. talon rakennus 4 tunnissa 200 työntekijän voimin.

Algoritmisen toiminnan rinnakkaistaminen: Vrt

X:= 3; Y:= 4;

ja

X:= 3; Y:= X;

Ylempi rinnakkaistuu, jälkimmäinen ei.

Esim. kymmenen metriä pitkän ojan vs. kymmenen metriä syvän kuopan kaivaminen .

Esim. Onko (1. luennolla käsitelty) palkkalistan summaus **väistämättä peräkkäinen** (inherently sequential) tehtävä?

V: Ei; N luvun summaus voidaan tehdä $O(\log N)$ askeleessa käyttämällä $N/2$ prosessoria: Ensimmäisessä vaiheessa prosessorit summaavat yhtäaikaisesti lukupareja (ensimmäinen, toinen), (kolmas, neljäs), ..., ($N-1$:nen, N:s). Toisessa vaiheessa summataan $N/4$ paria edellisen vaiheen summia jne kunnes jäljellä kokonaissumma.

Harel kuva 10.1.

Vrt. N joukkueen turnaus, jossa ottelujen häviäjät karsiutuvat, voidaan järjestää $N/2$ kentällä $O(\log N)$ ottelupäivässä

Kiinteä ja laajeneva rinnakkaisuus

Keskitytään toistaiseksi ongelman rinnakkaisessa ratkaisemisessa tarvittavien prosessorien lukumäärään.*

Edellä saimme aikaiseksi kertaluokkanopeutuksen ($O(N) \rightsquigarrow O(\log N)$) käyttämällä syötteen koon mukaan lineaarisen määrän ($N/2$) prosessoreita.

ns. **laajeneva rinnakkaisuus** (expanding parallelism).

Jos käytettävissä kiinteä määrä prosessoreja, vain vakiokertoimella nopeuttaminen mahdollista.

*Tiedonsiirto myös tärkeä ongelma: miten syötteet jaetaan prosessoreille, ja miten prosessorit välittävät tuloksensa toisilleen?

Rinnakkaisalgoritmien tutkimuksessa tarkastellaan suoritusajan ja muistitilan lisäksi kolmatta suuretta ja sen tarvetta syötteen koon funktiona:

prosessorien määrää ("size complexity").

Esim. N kokonaislukua voidaan laskea yhteen ajassa $O(\sqrt{N})$ käyttämällä $\sqrt{N}$ prosessoria.

(HT?)

Rinnakkainen lomituslajittelu

Palautetaan mieliin rekursiivinen lomituslajittelualgoritmi:

MSort($a_1, \ldots, a_n$):
1. **if** n=1 **then return** a_1;
2. **else**
3. S1:= MSort($a_1, \ldots, a_{\lfloor n/2 \rfloor}$);
4. S2:= MSort($a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$);
5. **return** Merge(S1, S2);

Hajota-ja-hallitse-tyyppinen algoritmi rinnakkaistuu helposti, jos aliongelmien ratkaisujärjestyksellä ei ole väliä.

Merkitään toimenpiteiden A ja B rinnakkaista käynnistämistä ($A||B$). Lomituslajittelu voidaan toteuttaa rinnakkaisena:

ParMSort($a_1, \ldots, a_n$):
1. **if** n=1 **then return** a_1;
2. **else**
3. (S1:= ParMSort($a_1, \ldots, a_{\lfloor n/2 \rfloor}$) ||
4. S2:= ParMSort($a_{\lfloor n/2 \rfloor+1}, \ldots, a_n$))
5. **return** Merge(S1, S2);

Suoritusta ParMSort($a_1, \ldots, a_n$) kuvaa binäärinen kutsupuu, jossa kutakin solmua suorittaa oma prosessori.

Puussa lehtiä n ja siten tasoja $O(\log n)$.

Tehtävänsä (syvemmillä tasoilla) suorittaneita prosessoreja voidaan käyttää uudestaan $\rightsquigarrow n$ prosessoria riittää.

Rinnakkaisuus $\rightsquigarrow$ yhden tason suoritusaika = yhden k.o. tasolla olevan kutsun suoritusaika.

Suoritusaikaa dominoi listojen S1 ja S2 lomittaminen.

Oletetaan, että n on kahden potenssi, ja lasketaan lomittamisessa tehtävien vertailujen lukumäärät. Yhden mittaisten listojen lomittamiseksi tarvitaan yksi vertailu, kahden mittaisten lomittamiseksi enintään 3, kahdeksan mittaisille enintään 7 jne:

$$\begin{aligned}
& 1 + 3 + 7 + 15 + \cdots + (n-1) \\
= \; & (2^1 - 1) + (2^2 - 1) + \cdots + (2^{\log n} - 1) \\
= \; & \sum_{i=1}^{\log n} (2^i - 1) = 2 \sum_{i=0}^{\log n - 1} 2^i - \log n \\
= \; & 2(2^{\log n} - 1) - \log n = 2n - \log n - 2 < 2n.
\end{aligned}$$

Siis lajittelun voi tehdä ajassa $O(n)$ käyttäen $O(n)$ prosessoria.

Aiemmin näimme, että alkioiden vertailuun perustuvan lajittelun optimaalinen peräkkäistoteutus vaatii $\Theta(n \log n)$ vertailua.

Tulovaativuus ja työoptimaalisuus

Merkitään rinnakkaisalgoritmin tarvitsemaa prosessorien määrää $p(n)$ ja aikaa $t(n)$.

Mikä on hyvä/tehokas rinnakkaisalgoritmi? Nopein? Vähän prosessoreja tarvitseva?

Rinnakkaisalgoritmin toimintaa voidaan simuloida yhdellä prosessorilla

- suorittamalla rinnakkaiset osuudet jossain peräkkäisjärjestyksessä, ja siten
- ajassa $O(p(n) \times t(n))$

Yleensä tavoitellaan mahdollisimman nopeaa rinnakkaisalgoritmia, joka on **työoptimaalinen**:

Sen **tulovaativuus** $p(n) \times t(n)$ on samaa kertaluokkaa kuin tehokkaimman peräkkäisalgoritmin aikavaativuus.

Esim. rinnakkaisella lomituslajittelulla $p(n) = O(n)$ ja $t(n) = O(n)$, joten se ei ole työoptimaalinen:
$p(n) \times t(n) = O(n^2) \neq O(n \log n)$.

Ks. Harel kuva 10.2.

Kiinteät laskentaverkot

Rinnakkaisalgoritmien suunnittelussa käytetyt prosessorimallit voivat olla vaikeita toteuttaa käytännössä.

Jos jokainen prosessori voi käsitellä jokaista muistipaikkaa, kuinka kytkennät toteutetaan? Kuinka muistipaikan samanaikaisen lukemisen tai kirjoittamisen konfliktit ratkaistaan?

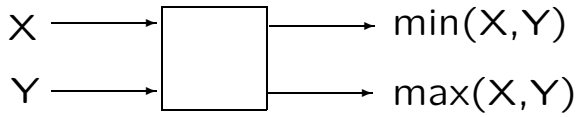
Käytännössä suoraviivaisesti toteutettavia ovat yksinkertaisista laskentaelementeistä koostuvat **laskentaverkot** (networks). Kukin laskentaelementti/prosessori kytketty kiinteästi muutamiin naapuriprosessoreihin.

Esim. Normaalin prosessorin perustoiminnot toteutettu yksinkertaisista AND-, OR- ja NOT-**porteista** (gate) koostuvina logiikkapiireinä.

Lajitteluverkot

Lajitteluverkot laskentaverkkoja, joilla voidaan järjestää n syötelukua kasvavaan järjestykseen.

Ne koostuvat yksinkertaisista vertailuprosessoreista (comparator):



Ns. **odd-even-lajitteluverkko** toteuttaa lomituslajittelun idean kiinteästi kytketyillä vertailuprosessoreilla.

Harel kuva 10.4.

Verkon alkuosa: kaksi rinnakkaista aliverkkoa $n/2$ alkion lajittelemiseksi.

Verkon loppuosa: aliverkko järjestettyjen $n/2$-alkioisten jonojen lomittamiseksi.

Lajitteluverkon suoritusaika $t(n)$ = verkon *syvyys*: monenko prosessorin läpi lukujen on pahimmillaan kuljettava.

Voidaan osoittaa: odd-even-lajitteluverkolla $t(n) = O((\log n)^2)$ ja $p(n) = O(n(\log n)^2)$.

$\rightsquigarrow$ Odd-even-lajittelu ei ole työoptimaalinen.

On olemassa myös työoptimaalinen lajitteluverkko, jolla $p(n) = O(n)$ ja $t(n) = O(\log n)$.

Se on kuitenkin monimutkainen ja käytännössä tehoton (vakiokertoimet suuria).

Systoliset laskentaverkot

Numeerisessa tieteellisessä laskennassa käsitellään tyypillisesti kookkaita lukuvektoreita ja -matriiseja.

laskutoimitukset usein tehokkaita ns. **systolisilla laskentaverkoilla**, joissa laskenta etenee jaksollisesti synkronoituina vaiheina (vrt. veren pumppaus elimistössä).

Esim. matriisin ja vektorin kertolasku.

Kurssilla N oppilasta, kukin osallistunut pisteytettyihin osasuorituksiin, joita M kpl $\rightarrow$ $N \times M$-matriisi.

Oppilaan arvosana määräytyy osasuoritusten painokertoimilla $(w_1, \ldots, w_M)$ painotettuna summana.

Peräkkäisratkaisu:

for opp:= 1 **to** N **do**
    Arv:= 0.0;
    for suor:= 1 **to** M **do**
        Arv:= Arv + Suor[opp][suor] * w[suor];
    output "Oppilaan", opp, "arvosana:", Arv;

Aikavaativuus selvästi $O(NM)$.

Ratkaistavissa ajassa $O(N + M)$ systolisella laskentaverkolla, jossa M liukuhihnaksi kytkettyä prosessoria.

Harel kuvat 10.5 ja 10.6.

Rinnakkaisuuden voima?

Käyttämällä tapauksen koon mukaan kasvava määrä prosessoreita ogelmille saadaan siis kertaluokaltaan nopeampia ratkaisualgoritmeja.

Auttaako rinnakkaisuus ratkaisemaan ongelmia, jotka ovat peräkkäisalgoritmeille (i) ratkeamattomia tai (ii) työläitä ongelmia?

Vastaus kysymykseen (i) on ”ei”:

Ongelman ratkaisua rinnakkaisalgoritmilla voi simuloida yhdellä prosessorilla, joka tekee jossain järjestyksessä kaikkien prosessorien työt.

Siis Church-Turingin teesi pätee myös rinnakkaisalgoritmeihin.

Vastaus kysymykseen (ii) on ”emme tiedä”:

Todistettavasti työläille ongelmille ei tunneta polynomisessa ajassa toimivia rinnakkaisalgoritmeja.

Toisaalta (työläiksi *epäillyt*) NP-täydelliset ongelmat ovat rinnakkaisalgoritmeilla polynomisessa ajassa ratkeavia:

Luokan NP päätösongelma A voidaan ratkaista epädeterministisellä algoritmilla Q_A jonkin polynomin $p(n)$ rajoittamassa määrässä askelia, tekemällä valintatilanteissa ”maagisia oikeita arvauksia”.

Rinnakkaistoteutus voi simuloida y.o. suoritusta deterministisesti allokoimalla valintatilanteissa uuden prosessorin jatkamaan kunkin vaihtoehtoisen valinnan mukaista laskentaa.

Jonkun prosessorin suoritus päättyy lopputilaan "kyllä" joss algoritmi Q_A hyväksyy syötteen. Toisaalta jos mikään prosessori ei päädy hyväksyvään lopputilaan $p(n)$ askeleen sisällä, tiedetään, että algoritmi Q_A hylkäisi syötteen.

Huom: ylläoleva NP-laskennan simulointi voi vaatia *eksponentiaalisen* määrän prosessoreita!

Tämä on kalvon 128 "jakaudutaan"-idea: NP on polynomiaikaista mutta muuten rajoittamatonta rinnakkaisuutta jossa eri prosessit kommunikoivat vain siten, että ensimmäinen "kyllä"-prosessi tappaa kaikki muut, ja vastaukseen "ei" päädytään vasta jos kaikki prosessit kuolevat itsekseen.

Rinnakkaislaskennan vaativuusluokat

Edellisen perusteella rinnakkaisalgoritmeilla polynomisessa ajassa ratkeavien ongelmien luokka **Parallel-P** (tai Parallel-PTIME) tuntuu epäkäytännöllisen laajalta: mahdollisesti eksponentiaalisesti kasvava prosessorien lukumäärä ei ole realistinen.

Käytännöllisesti mielekkääksi rinnakkaisvaativuuden luokaksi nimeltä **NC** (Nick's Class*) on esitetty "erittäin nopeasti" polynomisella määrällä prosessoreja ratkaistavissa olevien ongelmien luokkaa.

"Erittäin nopeasti": **polylogaritmisessa** ajassa $O((\log n)^k)$ jollain kiinteällä k.

Edes syötettä ei ole aikaa lukea annetussa järjestyksessä...

*Nicholas Pippengerin mukaan.

Monet käytännölliset ongelmat (esim. tarkasteltu palkkasummaus ja lajittelu) kuuluvat luokkaan NC.

Voidaan osoittaa, että luokan NC ongelmat ovat ratkaistavissa deterministisillä peräkkäisalgoritmeilla polynomisessa ajassa.

Siten tiedetään seuraavat ogelmaluokkien suhteet:

$$\mathrm{NC} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}$$

Voidaan lisäksi osoittaa, että
Parallel-P = PSPACE.

Edellisten vaativuusluokkien sisältyvyyksien uskotaan (muttei tiedetä) olevan aitoja

Tämä tarkoittaisi (järjestyksessä vasemmalta oikealle), että

• joitain käytännössä ratkeavia ongelmia ($\in$ P) ei voida ratkaista erittäin nopeasti käytännöllisellä määrällä rinnakkaisprosessoreja
(esim. suurimman yhteisen tekijän laskenta?)

• joitain epädeterminismin avulla tehokkaasti ratkaistavissa olevia ongelmia ei voida ratkaista deterministisesti käytännöllisessa ajassa

• joitain rajoittamattomalla määrällä rinnakkaisuutta ratkaistavissa olevia ongelmia ei voida ratkaista peräkkäisalgoritmeilla käytännöllisessä ajassa edes epädeterministisesti.

NP-täydellisyys: ”Uskomme että epädeterministinen polynominen aika on aidosti determinististä vahvempaa. Silloin ainakin nämä ongelmat olisivat työläitä.”

P-täydellisyys: ”Uskomme että luokassa P on 'luonteeltaan sarjallisia' ongelmia jotka eivät rinnakkaistu merkittävästi. Näitä lienevät ainakin sen 'vaikeimmat' ongelmat.”

Luokan P sisällä *polynomiset* palautukset ovat liian karkea väline ongelmien luokitteluun (HT); niiden tilalla käytetään rajoittuneempia kuten *logaritmitilaisia* palautuksia.

Looginen esimerkki P-täydellisestä ongelmasta on **samastus (unification)** (C. Dwork, P.C. Kanellakis ja J.C. Mitchell: On the sequential nature of unification, *Journal of Logic Programming* 1 (1984), sivut 35–50):

Annettu kaksi lauseketta ϕ ja ψ jotka sisältävät (yhteisiä) muuttujasymboleita. Voidaanko näille muuttujasymboleille valita sellaiset arvot, että lausekkeista tulee samat?

Jos esimerkiksi $\phi = f(x, x)$ ja $\psi = f(g(y), x)$, niin sijoitus $x = g(y)$ tuottaisi yhteisen muodon $f(g(y), g(y))$.

Jos taas $\phi = f(h(x), x)$, niin yhteistä muotoa ei ole koska $h \neq g$.

Tuntuu luontevalta, että "patologisilla" ϕ ja ψ tarvittavat muuttujasijoitusten valinnat vaikuttavat toisiinsa niin, ettei niitä voi tehdä toisistaan riippumatta eli rinnakkain.

9. Satunnaistetut algoritmit

(Harel, luvun 11 sivut 309-310, 313-319 ja 322-331.)

"Heitetään arpaa, että saamme tietää" [Joona 1:7]

- Hyödymmekö jotain sallimalla algoritmien tehdä satunnaisia valintoja?

Alkulukujen tunnistus ja generointi

Alkuluvuilla on keskeinen asema paitsi perinteisessä lukuteoriassa myös moderneissa salausmenetelmissä. (Ks. myöh.)

Positiivinen kokonaisluku n on **alkuluku** (prime), jos se on jaollinen positiivisista luvuista ainoastaan ykkösellä ja itsellään.

Muuten n on **yhdistetty luku** (eli koosteinen luku; composite number).

Ykköstä ei yleensä pidetä alkulukuna, joten alkulukuja ovat

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots$$

Alkulukuja on kaikkien lukujen joukossa äärettömästi, epätasaisesti jakautuneina.

Lukua n pienempien alkulukujen lukumäärää merkitään $\pi(n)$; Tiedetään $\pi(n) \approx n/\ln n$.

Lukuteoreettisissa salausmenetelmissä tarvitaan suuria, esim. 150-numeroisia alkulukuja.

Kuinka näitä voidaan tuottaa?

Alkulukujen lukumäärästä $\pi(n)$ voidaan arvioida, että satunnaisesti valittu luku n on alkuluku todennäköisyydella $1/\ln n$.
$\rightsquigarrow$ 150-numeroisen alkuluvun löytämiseksi riittää testata keskimäärin.

$$\ln 10^{150} = 150 \ln 10 \approx 150 \times 2,3 = 345$$

satunnaista 150-numeroista lukua.*

Alkulukujen *generointi* siis palautuu alkulukujen *tunnistamiseen*.

Kuinka ratkaistaan, onko n alkuluku?

*Puolet vähemmän kokeilemalla vain parittomia.

Suoraviivainen menetelmä: Kokeillaan, onko n jaollinen millään luvuista $2, 3, \ldots, n-1$. Jos ei, n on alkuluku, muuten yhdistetty.

Ongelma: Eksponentiaalista luvun n *"tekstiesityksen" pituuden suhteen*; toimii, jos n pieni, mutta mahdotonta, jos $n \approx 10^{150}$.

Lievää parannusta: Testataan jakajina kakkosen lisäksi vain alkulukuja $\leq \sqrt{n}$

– näitäkin eksponentiaalisen paljon. (Esim. 150-numeroisella luvulla **paljon** enemmän kuin mikrosekunteja alkuräjähdyksestä!)

Alkulukujen tunnistamiseen ei tunneta todistetusti oikein toimivaa tehokasta (eli polynomisessa ajassa toimivaa) determinististä algoritmia.

Toisaalta ongelma on suhteellisen helposti ratkaistavissa *satunnaistetulla* algoritmilla.

Satunnaistettu alkulukujen testaus

Ensimmäisiä vaikeitten laskentaongelmien satunnaistettuja ratkaisumenetelmiä (Solovay & Strassen, 1977).

algoritmityyppi	pysähtyy nopeasti	vastaa oikein
Monte Carlo	aina	todennäköisesti
Las Vegas	todennäköisesti	aina

Menetelmän periaate; sivuutamme lukuteoreettiset yksityiskohdat.

Perustuu sen osoittamiseen, että testattava luku N *suurella todennäköisyydellä* ei ole yhdistetty (eli on alkuluku).

Koosteisuuden **todiste**: luku $K \in \{1, \ldots, N-1\}$, joka osoittaa, että N on yhdistetty luku.

Jos

$$\text{syt}(K, N)^* > 1, \tag{1}$$

N on selvästi yhdistetty luku. Syt voidaan laskea tehokkaasti Eukleideen algoritmilla.

Muuten tutkitaan, päteekö

$$\mathit{Js}(K, N) \not\equiv K^{(N-1)/2} (\text{mod } N), \tag{2}$$

missä $\mathit{Js}(K, N) \in \{-1, 0, 1\}$ on ns. Jacobin symboli. Sekä Jacobin symbolien että modulaaristen potenssien laskenta voidaan tehdä tehokkaasti, $O(\log N)$ aritmeettisella operaatiolla.

Voidaan osoittaa, että jos (2) pätee, N ei ole alkuluku. Toisaalta, jos N on yhdistetty luku, ainakin puolella mahdollisista $K \in \{1, \ldots, N-1\}$ epäyhtälö (2) pätee.

*Eli gcd(K, N); lukujen suurin yhteinen tekijä.

Sen todennäköisyys, että N on yhdistetty luku, jolle yksi K ei toteuta testejä (1) ja (2), on siis $< 1/2$. Kokeilemalla kaksi satunnaista K saadaan tn $< 1/4$, ... kun on kokeiltu k lukua, N on yhdistetty todennäköisyydellä $< 1/2^k$, eli N on alkuluku todennäköisyydellä $> 1 - (1/2)^k$.

Harel, kuva 11.1.

50–200 todistajaa läpäissyt luku N on erittäin suurella todennäköisyydellä alkuluku.

Alkulukuja ja yhdistettyjä lukuja voidaan siis tunnistaa tehokkaasti satunnaistetulla algoritmilla.

Toisaalta yhdistetyn luvun tekijöihin jakoa ei osata tehdä tehokkaasti edes satunnaisuutta hyväksikäyttäen, vaikka ongelmaa on tutkittu matematiikassa noin kolmesataa vuotta!

Satunnaistetut vaativuusluokat

Satunnaisalgoritmien formaali malli: **probabilistinen Turingin kone**, joka voi tehdä satunnaisia valintoja (”heittää kolikkoa”).

Vaativuusluokka **RP** (randomized polynomial time): Ne päätösongelmat, jotka voidaan ratkaista probabilistisellä Turingin koneella, joka
1) sanoo varmasti ”ei”, jos syöte on ongelman ”ei”-tapaus, ja
2) sanoo todennäköisyydellä $> 1/2$ ”kyllä”, jos syöte on ongelman ”kyllä”-tapaus.

$\rightsquigarrow$ Virheellisen vastauksen todennäköisyys saatavissa mielivaltaisen pieneksi iteroimalla suorituksia.

Esim. Sen testaaminen, onko syöteluku yhdistetty, on luokan RP ongelma.

Probabilistinen Turingin kone on deterministisen yleistys. Toisaalta epädeterminististen Turingin koneiden ”maagisesti” oikeaan johtava ”kolikko” on voimakkaampi kuin ”normaali” jollain todennäköisyydellä oikeaan johtava.

Siten tiedämme ongelmaluokkien suhteista:

$$\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}.$$

Taaskaan ei tiedetä, ovatko sisältyvyydet aitoja.

Huom. Koska epädeterminismi ei laajentanut algoritmisesti ratkeavien ongelmien joukkoa, sama pätee sitä heikompaan satunnaisuuteen, eli Church-Turing-teesi pätee myös satunnaisalgoritmeihin.

Julkisen avaimen salaus

Alkulukujen generoinnin helppoudella ja lukujen tekijöinnin (arvellulla) työläydellä on tärkeitä sovelluksia verkossa tapahtuvassa tiedonsiirrossa.

Tarvitaan menetelmiä, joilla verkossa kulkeva tietoliikenne voidaan **salata** asiattomilta.

Lisäksi tarvitaan keinoja **digitaalisten allekirjoitusten** toteuttamiseen, joilla voidaan varmistaa viestin lähettäjä ja viestin muuttumattomuus.

Menetelmillä ilmeisiä sovelluksia sähköisen rahan ja kaupankäynnin toteuttamisessa.

Perinteiset menetelmät ovat **symmetrisiä**: Viestin salaus ja salauksen purku tapahtuvat (oleellisesti) samalla **avaimella**.

Ongelmana salassa pidettävän avaimen turvattu välittäminen kumppaneiden kesken.

Julkisen avaimen salausmenetelmät ovat **epäsymmetrisiä**.

Kullakin on **kaksi** avainta: **julkinen salausavain** *Encr* ja **henkilökohtainen purkuavain** *Decr*.

Viestejä käsitellään jonoina kokonaislukuja M (katkomalla ja tulkitsemalla niiden bittejä sopivasti).

Henkilön A purkuavain (tai -funktio) $Decr_A$ avaa hänen salausavaimellaan (-funktiollaan) $Encr_A$ salatun viestin M:

$$Decr_A(Encr_A(M)) = M$$

Funktioiden oltava tehokkaasti laskettavia.

Lisäksi salainen purkufunktio $Decr_A$ ei saa olla helposti (eli polynomisessa ajassa) laskettavissa julkisesta salausfunktiosta $Encr_A$.

Perinteinen menetelmä on kuin yhteinen lipas, jonka avaimesta on kopio kummallakin kumppanilla – mutta toivottavasti ei kenelläkään muulla. . .

Lähettäjä laittaa viestinsä lippaaseen, lukitsee sen ja lähettää postipakettina vastaanottajalle.

Julkisen avaimen menetelmässä vastaanottaja antaakin lähettäjälle *avoimen* lippaan, mutta pitääkin itsellään sen *ainoan* avaimen!

Avoin lipas on vastaanottajan *Encr*, sen avain taas vastaava *Decr*.

Jos salaus- ja purkufunktiot ovat bijektiivisiä, alkuperäinen viesti voidaan ensin *salata purkufunktiolla* $Decr_A$ ja sitten *purkaa salausfunktiolla*:

$$Encr_A(Decr_A(M)) = M.$$

Koska vain A tuntee purkufunktionsa $Decr_A$, hänen julkisellä salausfunktiollaan $Encr_A$ aukeavat viestit ovat välttämättä A:n lähettämiä
$\rightarrow$ digitaalinen allekirjoitus.

Esim. Harel kuva 11.4.

RSA-menetelmä

Tunnetuin julkisen avaimen salausmenetelmä.

Rivest, Shamir & Adleman, 1978.

Alice (A) valitsee kaksi satunnaista (esim. 150-numeroista) alkulukua P ja Q, kuten edellä käsiteltiin.

Lasketaan $N = P \times Q$ ja $R = (P-1) \times (Q-1)$.

Seuraavaksi A valitsee suuren (esim. 300-numeroisen) luvun K, jolla syt$(K, R) = 1$. Esim. molempia P ja Q suurempi alkuluku kelpaa.

Lopulta A laskee luvun G, jolla

$$G \times K = 1 (\text{mod } R).$$

Voidaan osoittaa, että tällainen G on olemassa ja yksikäsitteinen.

Julkinen avain on pari (G, N),
salainen avain on pari (K, N).

P ja Q voidaan hävittää, mutta ne on pidettävä salassa. (Miksi?)

Kuinka lähetetään viesti A:lle, jonka julkinen avain (G_A, N_A) tunnetaan?

Käsitellään viestiä M numeroina väliltä $0 \ldots N_A - 1$.

Salatun viestin $E_A(M)$ tuottaminen:

$$E_A(M) = M^{G_A} (\text{mod } N_A).$$

A purkaa vastaanottamansa salatun viestin:

$$D_A(E_A(M)) = E_A(M)^{K_A} (\text{mod } N_A).$$

Voidaan osoittaa, että tämä on sama kuin alkuperäinen viesti M.

Helpompi nähdä, että

$$\begin{aligned} D_A(E_A(M)) &= E_A(M)^{K_A} (\text{mod } N_A) \\ &= (M^{G_A} (\text{mod } N_A))^{K_A} (\text{mod } N_A) \\ &= M^{G_A \times K_A} (\text{mod } N_A) \\ &= E_A(D_A(M)) \end{aligned}$$

$\rightarrow$ menetelmä sopii digitaaliseen allekirjoittamiseen.

Menetelmän soveltamisessa tarvitut laskutoimitukset ovat polynomisessa ajassa ja kohtalaisen tehokkaasti laskettavia. (Sivuutetaan.)

RSA-menetelmää pidetään turvallisena: kaikki sen murtamiseen kehitetyt mahdollisuudet ovat vähintään yhtä työläitä kuin lukujen tekijöihin jako, johon ei ole löydetty polynomista ratkaisumenetelmää.

10. Algoritminen älykkyys

(Pohjana Harel, luku 12. Lisäksi materiaalia kirjasta S.J. Russell & P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall 1995.)

"What a piece of work is a man! how noble in reason! how infinite in faculty! in form and moving, how express and admirable! in action, how like an angel! in apprehension, how like a god! the beauty of the world! the paragon of animals!

And yet, to me, what is this quintessence of dust?"

[William Shakespeare: Hamlet (2. näytös, 2. kohtaus)]

Mitä on **teko-** tai **keinoäly**?

Voiko mekaaninen laskenta tuottaa älykkyyttä?

Lisää kurssilla *Tekoäly*.

”Spesisimin” synti

”Kysymys ’osaako tietokone ajatella?’ on kuin kysymys ’osaako sukellusvene uida?’.”

Olisi epäreilua *määritellä* ”älykkyys” tyyliin ”se erityinen kyky jolla ihmiset ratkovat menestyksekkäästi vastaantulevia ongelmia”.

Entä (omissa puuhissaan varsin menestyksekkäät) eläimet? Tai Maapalloa valloittamaan laskeutuvat avaruusoliot?

Älykkyyden määritelmä ei siis saa perustua biologiaan, vaan yleiseen kykyyn *käyttää informaatiota ongelmien ratkaisussa* tms.

Mutta sitähän tietokoneetkin tekevät! Voisivatko nekin siis olla älykkäitä?

Turingin testi

”Jos se näyttää leijonalta, kuulostaa leijonalta ja tuntuu leijonalta, niin se *on* leijona” – tai minun on ainakin syytä toimia ikään kuin se olisi leijona. . .

Alan M. Turing (taas!) sovelsi tätä periaatetta ehdottaessaan vuonna 1950 nimeään kantavaa määritelmää sille, milloin tietokone olisi älykäs.

Taustalla on silloin psykologiassa vallinnut *behavioristinen* ajattelu: se on älykäs jos se *käyttäytyy* älykkäästi.

Tentaattori A olkoon ihminen. Ihminen lienee kyvykäs arvioimaan muiden älykkyyttä...

Valitaan tietokonekandidaatille ihmisopponentti, ja annetaan niille nimet B ja C tentaattorin A tietämättä kumpi on kumpi.

Liitetään B ja C tentaattoriin A pelkkää tekstiä välittävillä tietoliikennelinjoilla. Näin sokaistaan tentaattorin A biologiset erityistaidot lajitoveriensa tunnistamiseen kuuloaistilla, näköaistilla,...

Harel, kuva 12.1.

A tekee tenttijöille B ja C kirjallisia kysymyksiä, ja yrittää vastausten perusteella päätellä kumpi on kumpi.

Tietokone on tunnustettava älykkääksi jos A ei tähän pysty. (Tai jos toistettaessa testiä eri tentaattoreilla A oikean vastauksen todennäköisyys lähestyy lantinheittoa $\frac{1}{2}$.)

Kiinalainen huone

Filosofi John Searle on kääntänyt tilanteen toisin päin:

Istun ääni- ym. eristetyssä huoneessa jossa on kaksi luukkua, ”syöte” ja ’tulos”.

Huoneessa on myös kirja, jonka jokaisella sivulla on yksi kiinalainen kirjoitusmerkki (enkä minä osaa kiinaa).

Aina kun syöteluukkuun ilmestyy kirjoitusmerkki, etsin sitä kirjani aukeamien vasemman puoleisilta sivuilta. Kun löydän, laitan tulosluukkuun oikean puoleisella sivulla olevan merkin.

Huone kuulemma keskustelee ulkomaailman kanssa älykkäästi kiinaksi – mutta missä huoneen älykkyys on?

Huone on korkeintaan yksinkertainen Turingin kone! Missä sen älykkyys sitten on?

Neuroottista älykkyyttä

Searle siis kieltää Turingin perusidean että älykkyys voitaisiin määritellä ”älykkään näköisenä” *makro*tason syöte-tulos-vastaavuutena. Entä *mikro*tasolla?

Jos hermosoluverkon kytkentöjen voimakkuudet eivät muutu – eli aivot eivät muista – voidaan verkon kokonaistoiminta (= yksittäisten hermosolujen syöte-tulos-operaatioiden yhdistelmä) kuvata loogisena operaationa (McCulloch & Pitts 1943) tai jopa äärellisenä automaattina. . .

. . . ja Turingin kone taas voidaan nähdä äärellisenä automaattina jolla *on* muisti!

Eli muistimme malli ei saisi toisaalta olla ”Turing-tyyppinen” eikä toisaalta liian sidoksissa biologiamme yksityiskohtiin.

Pelit

Kysymys *täyden* älykkyyden *periaatteellisesta* ohjelmoitavuudesta on enemmänkin filosofien, kognitiotieteilijöiden, psykologien yms. alaa.

Tietojenkäsittelijä taas yrittää kehittää laskennallisia mentelmiä jotka antavat *käytännössä* ”älykkäitä” vastauksia *rajoitetuilla* sovellusalueilla.

Yksi tälläinen alue ovat **pelit**: maailmasta tarvittava *tieto mahtuu laudalle* ja mahdollisia *siirtoja on rajallisesti*, ja silti menestyksekästä pelaamista pidetään älykkyyttä vaativana.

Pelejä kuvaillaan *pelipuilla* joiden solmut ovat pelitilanteita ja solmusta lähtevät kaaret mahdollisia siirtoja.

Harel, kuva 12.2.

Lehdet ovat lopputilanteita joissa ratkeaa voitto ja häviö.

3×3-jätkänshakin pelipuun juuressa on tyhjä ruudukko, ja sillä on 9 lasta, eli eri mahdollisuudet laittaa ensimmäinen ristimerkki. (R&N, kuva 5.1.)

Kussakin näin syntyneessä ruudukossa on 8 mahdollisuutta laittaa ensimmäinen ympyrämerkki. Puuhun saadaan korkeintaan $9 \cdot 8 \cdot 7 \cdot \cdots \cdot 1 = 362\,880$ solmua. Käsiteltävissä helposti tietokoneella.

Tavallisen shakin haarautumisaste (=lasten lukumäärä) on noin 35 ja peli voi edetä 40-50 siirtoparia. Saadaan suunnilleen kokoa 35^{100} oleva puu – suurempi kuin tunnetun maailmankaikkeuden protonien määrä tai mikrosekuntien määrä sitten alkuräjähdyksen!

Vaikka *periaatteessa* käsiteltävissä, käytännössä puusta karsitaan (= lapsia ei tuoteta eikä tutkita) mielenkiinnottomia solmuja.

Heuristiikat

(Kreikan ”heurisko”, ”keksiä”; vrt.
”heureka”, ”olen keksinyt”.)

Jos meillä olisi *varmaa* tietoa siitä mikä lapsisolmuista on koko loppupelin kannalta paras, niin emme tarvitsisi koko puuta: laskisimme nykyisen solmun lapset, valitsisimme niistä parhaan, unohtaisimme muut, ja jatkaisimme peliä sillä siirrolla. Saisimme klassisen algoritmin!

Vertaa operaatio **choose**, joka tietää. Tämä on kalvon 128 etsintäajatus.

Vaan emme tiedä, vaan joudumme tyytymään *arviointifunktioon* joka kertoo annetusta solmusta
• pitäisikö sen lapset tuottaa ja tutkia
• ja jos ei niin kuinka hyvä se kannaltamme on.

Shakissa voisi heuristinen arviointifunktio olla vaikkapa:

”Jos nykytilanteessa kukaan ei uhkaa ketään, niin tuota sen lapset tutkittaviksi – tilanne on epäselvä ja vaatii lisätietoa.

Muuten laske kummalle jäisi materiaalietu, jos kaikki uhkaukset pelattaisiin nyt heti tyyliin ’minä syön tämän – sinä syöt tuon – minä sen – . . . ’.”

Esimerkki *ahneesta* heuristiikasta: ajatellaan että kannattaa syödä heti kun vain pystyy eikä mietiä millainen tulevaisuus sen jälkeen on.

Altistaa tietenkin *uhrauksille* jossa tarjoamalla syötävää houkutellaan pelaaja lyhytnäköisen hyödyn toivossa lopulta tappioon johtavalle tielle.

Horisonttiongelma: kuinka kauaksi tulevaisuuteen pitää kurkistaa?
(R&N, kuva 5.5.)

Hyvässä arviointifunktiossa onkin aina sekä **ahne** että **konservatiivinen** komponentti sopivassa suhteessa.

Jos ahneeseen luotetaan liikaa, niin hävitään hurja/kaistapäisyyden vuoksi, koska puusta tutkitaan liian *vähän* vaihtoehtoja.

Jos konservatiiviseen, niin hävitään aikapulan vuoksi, koska tutkitaan liian *paljon*.

Arviointifunktio siis leikkaa pelipuun poikki (paljon) ennen kuin päästään lehtiin, ja liittää kuhunkin katkaisukohtaan luvun, joka kertoo (toivottavasti) kuinka *lupaavaa* olisi kulkea ensin siihen, kun lopullisena tavoitteena on päätyä voittolehteen.

Samat periaatteet sopivat myös *yksinpeleihin* kuten pasianssiin ja **ongelmanratkaisuun**.

Ohjattu haku ongelmanratkaisussa

Haetaan esimerkiksi lyhintä reittiä Aradista Bukarestiin (R&N, kuva 4.2.) – joskin tähän tehtäväänhän on "oikeakin" algoritmi. . .

Hakualgoritmi A^* (R&N, luku 4.2.) antaa kullekin vielä tutkimattomalle solmulle lupaavuusluvuksi "*vaadittu työ siirryttäessä alusta solmuun + heuristinen arvio jäljellä olevasta työstä jatkettaessa solmusta loppuun*", ja tutkii aina lupaavimman solmun.

Vaadittu työ = löydetty reitti teitä pitkin Aradista tähän kaupunkiin.
Arvio = etäisyys linnuntietä tästä kaupungista Bukarestiin.
(R&N, kuva 4.4.)

Kun arvio ei ole pessimistinen, löydetään paras reitti.

Jos arvio on liian optimistinen (esim. 0), etsintään kuluu liikaa aikaa.

Minimax-periaate

Miten katkaisuluvut nostetaan sisäsolmuihin?

Omalla vuorollani teen minulle lupaavimman siirron. Valitsen siis *maksimin* nykyisen (juuri)solmuni lasten luvuista.

Koska vastustajakin haluaa voittaa, niin hän valitsee omalla vuorollaan hänelle itselleen lupaavimman eli minulle vähiten lupaavan siirron. Hän siis valitsee *minimin*.

Harel, kuva 12.3.

Vertaa kalvojen 145-146 polynomiseen hierarkiaan: ”Minullapa onkin sellainen (siirto ja) lupaavuusluku, ettei mikään sinun seuraava (siirtosi ja) lukusi pysty sitä pienentämään!”

Minimax-periaate sallii puun pienentämisen edelleen (esim.) ns. α-β-karsinnalla.

Harel, kuva 12.4.