6. Ratkeamattomat ongelmat

(Harel luku 8)

”*tämä työ on sinulle liian raskas*” [2. Moos. 18:18]

- On myös ongelmia, joita ei *lainkaan* pystytä ratkaisemaan algoritmisesti!

Algoritmista rakeamattomuutta käsitellään perusteellisemmin matematiikassa *Laskettavuuden teorian* tai Rekursioteorian nimikkeellä.

Suoraviivaisella laskenta-argumentilla voi nähdä, että kaikilla laskentaongelmilla ei ole ratkaisualgoritmia:

Tarkastellaan päätösongelmia.

Sekä syötteet että ohjelmat (algoritmit) voidaan pohjimmiltaan esittää binäärijonoina.

Algoritmeja/ohjelmia ja syötteitä on siten *numeroituvasti* ("yhtä paljon kuin luonnollisia lukuja").

Päätösongelmat ovat syötejonojoukkojen osajoukkoja

$$\{x \in \{0,1\}^n \mid x \text{ on ongelman "yes-tapaus}\}.$$

Erilaisia päätösongelmia on siten *ylinumeroituvasti* ("yhtä paljon kuin reaalilukuja").

Siis kaikilla päätösongelmilla ei voi olla ratkaisualgoritmia (vaikkei aika- tai tilaresursseja rajoitettaisi lainkaan).

$\leadsto$

Onko ihmisen ongelmanratkaisukyky rajallinen?

Onko ihminen algoritmi tai joukko algoritmeja?

Ainakin ihmisen aika- ja muistiresurssit ovat kovin rajalliset!

Sama päättely "graafisesti":

Laaditaan ääretön taulukko

	syöte 1	syöte 2	syöte 3	$\cdots$
ohjelma 1	$\pm$	$+$	$-$	$\cdots$
ohjelma 2	$-$	$\mp$	$+$	$\cdots$
ohjelma 3	$+$	$+$	$\mp$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Käännetään merkit sen *diagonaalilla* ympäri.

Löytyykö tämä käännetty diagonaali taulukon miltään riviltä r?

Ei, koska rivin r syöte r oli myös käännetty!

Algoritmisesti ratkeamattomia ongelmia siis on välttämättä olemassa.

Ratkeamattomat ongelmat eivät pelkästään teoreettisia kuriositeetteja, vaan osa myös käytännössä relevantteja.

Tutustutaan muutamiin ratkeamattomiin ongelmiin.

Tason kaakelointi

Laajennus apinapalapeliongelmasta.

Annettuna joukko (kiinteästi suunnattuja, s.o. ei pyöriteltäviä) kaakelilaattojen malleja.

Kaakeleissa kukin lävistäjien erottamista neljästä alueesta jonkin värinen (tai yhtä hyvin värillinen apinan puolikas, kuten aiemmin).

Harel kuvat 8.1 ja 8.2.

Voiko annetun tyyppisillä laatoilla peittää minkä tahansa äärellisen alueen siten, että toisiaan koskettavien reunojen väri on sama?

Kaakelointiongelma (monine variaatioineen) on algoritmisesti **ratkeamaton** (noncomputable, undecidable):

Sen ratkaisevaa algoritmia ei ole (*eikä voikaan olla olemassa*).

Palaamme kaakelointiongelmaan Turingin koneiden yhteydessä.

Kuvamme algoritmisten ongelmien kentästä on laajentunut, ks. Harel kuva 8.3.

Rajoittamattomuus ja ratkeamattomuus

Ratkeamattoman ongelman ratkaisemisyritykset edellyttävät rajoittamattoman monien tapausten tutkimista: jos tapauksia olisi rajoitettu määrä, ratkaisualgoritmi voisi tutkia ne äärellisessä ajassa.

Toisaalta rajoittamattomuudesta ei välttämättä seuraa ratkeamattomuus.

Esim. Dominopolkuongelma.

Annettuna joukko kaakelityyppejä ja kaksi tason pistettä V ja W, voidaanko muodostaa V:stä W:hen "dominopolku", jossa peräkkäisten laattojen toisiaan koskettavat reunat samanväriset?

Ks. Harel, kuva 8.4.

Jos käytettävissä äärellinen alue, ongelma on selvästi ratkeava. (Miksi?)

Voidaan osoittaa, että ongelma on ratkeamaton, jos käytettävissä on puolikas taso, mutta *ratkeava, jos "dominopolku" saa käyttää hyväkseen koko tasoa!*

Kontekstittomien kielten ekvivalenssi

Ohjelmointikielten syntaksi eli muotosäännöt määritellään usein **kontekstittomilla kieliopeilla** (tai jollain niiden variaatioilla)

Esim.

$$
\begin{array}{lll}
\text{Statement} & \rightarrow & \text{if (Expr) Statement} \\
 & | & \text{while (Expr) Statement} \\
 & & \ldots \\
\text{Expr} & \rightarrow & \text{(Expr)} \\
 & | & \text{Expr BinOp Expr} \\
 & & \ldots
\end{array}
$$

Annettuna kaksi kontekstitonta kielioppia, kuvaavatko ne saman kielen?

Ongelma on algoritmisesti ratkeamaton.

Pysähtymisongelma

Pysähtyykö annettu algoritmi/ohjelma annetulla syötteellä?

Keskeisin ratkeamaton ongelma – monien muiden ongelmien ratkeamattomuus palautuu pysähtymisongelman ratkeamattomuuteen.

Joskus terminoivuus on helppo nähdä:

Esim. Algoritmi A:

(1) **while** X≠1 **do** X:= X-2;

Pysähtyy jos ja vain joss syöte X on pariton positiivinen kokonaisluku.

Toisinaan erittäin vaikeaa:

Esim. Algoritmi B:

```
(1)      while X≠1 do
(1.1)        if X parillinen then X:= X/2;
(1.2)        else X:= 3*X+1;
         endwhile
```

Algoritmi B päättyy kaikilla positiivisilla kokonaisluvuilla X, *joilla sitä on kokeiltu*.

Kukaan ei ole pystynyt todistamaan sen päättymistä *kaikilla* pos. kokonaisluvuilla X.

Osoitetaan seuraavaksi, että yleinen pysähtymisongelma on algoritmisesti ratkeamaton.

Tyypilliseen tapaan jonkin asian mahdottomuus osoitetaan epäsuorasti, osoittamalla että asian mahdollisuudesta seuraa looginen ristiriita.

Tehdään vastaoletus:
On olemassa algoritmi Pysähtyy(R,X), joka

- saa syötteenään algoritmin R ja sen syötteen X
- palauttaa arvon **true**, jos R(X) pysähtyy, ja muuten arvon **false**.

Muodostetaan algoritmi S(W):

if Pysähtyy(W,W) **then while true do**
    { /* ei mitään, pysytään silmukassa! */ }
return false;

Päättyykö evaluointi S(S)?

S(S) pysähtyy täsmälleen, kun
Pysähtyy(S,S)=**false**, eli kun
S(S) *ei pysähdy!*

Ristiriita $\rightsquigarrow$ vastaoletus oli väärä.

Algoritmia Pysähtyy(R,X) ei voi olla
olemassa.

Pysähtymisongelman vaihtoehtoinen ratkeamattomuustodistus: Cantorin **diagonalisointiargumentti**, ks. kalvo 151 tai Harel sivut 210–211.

Ongelman Q NP-kovuus osoitetaan *polynomisella* palautuksella jostain toisesta NP-kovasta ongelmasta P. (Palautus osoittaa, että *jos* Q osattaisiin ratkaista polynomisessa ajassa, niin sama pätisi myös ongelmaan P.)

Vastaavasti ongelman Q ratkeamattomuus osoitetaan *rajoittamattomalla algoritmisella* palautuksella jostain ratkeamattomaksi tiedetystä ongelmasta P

(usein juuri pysähtymisongelmasta).

Tässä palautus perustuu oletukseen, että meillä olisi (hypoteettinen) ongelman Q ratkaiseva algoritmi A_Q, jota voisi käyttää ns. **oraakkelina** eli aliohjelmana, joka muodostaa osan ongelman P ratkaisevasta algoritmista.

(Polynominen palautushan oli käännös kielestä toiseen, ei aliohjelma jota voi kutsua useasti ja jonka jälkeen voi jatkaa edelleen pääohjelmassa.)

Ongelmalle P hahmotellaan ratkaisualgoritmi käyttäen algoritmia A_Q aliohjelmana.

Jos P on ratkeamaton, yllä hahmoteltu konstruktio osoittaa, että myöskään ongelmalla Q ei voi olla ratkaisualgoritmia A_Q.

Verifiointiongelman ratkamattomuus

Aiemmin väitimme, että algoritmien verifiointiongelma on ratkeamaton.

Perustellaan asia palautuksella pysähtymisongelmasta:

Vastaoletus: On olemassa algoritmi (oraakkeli) A, joka saa syötteenä ongelman spesifikaation P ja ohjelman R. Suoritus A(P,R) päättyy, ja sen paluuarvo on tosi joss R on spesifikaafion P mukainen täysin oikeellinen ohjelma.

Nyt pysähtymisongelman tapaus (R, X) voidaan ratkaista verifiointioraakkelin A avulla:

(1) C:= A(''Input=X; Output=anything'', R);
(2) **if** C = **true return** ''R(X) pysähtyy'';
(3) **else return** ''R(X) ei pysähdy'';

Rivillä (1) käytetty spesifikaatio ei vaadi osittain oikeellisuudelta mitään, joten oraakkeli A vain tarkistaa, pysähtyykö R syötteellä X.

Koska pysähtymisongelma on ratkeamaton, ylläoleva ''algoritmi'' on mahdoton.

$\leadsto$ verifiointiongelmalla ei ole ratkaisualgoritmia.

Osittain ratkeavat ongelmat

Pysähtymisongelma ei tunnu täysin ratkeamattomalta: Annettua algoritmia R voidaan simuloida annetulla syötteellä X ja katsoa, pysähtyykö se. Jos suoritus päättyy, vastaus on "kyllä".

Ongelmallista on tietää, kannattaako simulointia vielä jatkaa vai onko ohjelma R esim. ikuisessa silmukassa.

Pysähtymisogelman "kyllä"-tapauksilla on siis äärellinen ja äärellisessä ajassa tarkastettava **todiste** (certificate), nimittäin päättyvän suorituksen jäljitys.

Ongelmat, joiden "kyllä-tapauksilla on tällainen todiste, ovat **osittain ratkeavia** (partially decidable).*

*Rekursioteoriassa tällaisia ongelmia kutsutaan **rekursiivisesti lueteltaviksi** (recursively enumerable).

Osittain ratkeavat ongelmat voidaa osoittaa laskettavuuden kannalta ekvivalenteiksi: jokainen niistä on palautettavissa muihin osittain ratkeaviin ongelmiin.

Huom: Verifiointiongelma on "voimakkaammin ratkeamaton" ongelma kuin pysähtymisongelma:

Pysähtymisongelma voidaan palauttaa verifiointiongelmaan, mutta ei päinvastoin.

Verifiointiin sisältyy pysähtymisongelmaa vaikeampi terminointiongelma
(eli **totaalisuusongelma**):
Pysähtyykö R *kaikilla* syötteillään?

Ratkeamattomat ongelmat muodostavat vastaavan (mutta täsmällisemmin tunnetun) monitasoisen hierarkian kuin työläät ongelmat.

7. Laskennan perusmallit

(Harel luku 9)

"Puhun vielä sen jäsenistä, sen ihmeellisestä rakenteesta." [Job 41:4]

- Mikä on yksinkertaisin mutta riittävän voimakas malli algoritmien esittämiseen?

Algoritmien yksinkertaistettuja malleja tarkastellaan perusteellisemmin kursseilla *Ohjelmoinnin ja laskennan perusmallit* sekä *Laskennan teoria*.

Käsiteltävän datan malli?

Ohjelmien tietokonetoteutuksissa kaikki data (merkit, muuttujien arvot, tietorakenteet) on pohjimmallaan peräkkäiseen muistiin sijoitettuja bittien jonoja.

$\leadsto$ Riittää pystyä käsittelemään binäärijonoja.

Mukavuussyistä käytetään laajempia **aakkostoja** (esim. ASCII-merkkejä tai kymmenjärjestelmän numeroita 0,...,9).

Algoritmien käsittelemät arvot ja tietorakenteet voivat kasvaa.

$\leadsto$ käsitellään dataa *rajoittamattoman* pituisella **nauhalla** ("potentiaalinen", ei "aktuaalinen" äärettömyys).

Nauhalla voi kussakin positiossa olla yksi aakkoston merkki.

HUOM! Laskennan *tehokkuuteen* vaikuttaa montako "datamerkkiä" on käytössä tarvittavien "välimerkkien" lisäksi: unaarilukujärjestelmä ("tukkimiehen kirjanpito") on eksponentiaalisesti pidempää kuin binääri- ja muut.

Minimaalinen kotrollirakenne?

Tarkastelemissamme algoritmeissa (ja normaaleissa tietokoneohjelmissa) implisiittinen oletus, että algoritmia suorittava **prosessori** on kullakin hetkellä jossain *kohdassa* algoritmia.

Voidaan mallintaa **tiloina**: algoritmi on kussakin vaiheessa täsmälleen yhdessä tilassa.

Kontrollin eteneminen riippuu muuttujien ja tietorakenteiden arvoista: (**if** N=0 **then** ... **else**..., **while** $I < N$ **do** jne.)

Mallinnetaan kontrollin ohjaus minimaalisella tavalla: Algoritmeilla on käytössään yhtä nauhapositioa kerrallaan katsova **lukupää**.

Algoritmin seuraava tila määräytyy lukupään kohdalla olevan nauhamerkin perusteella, samoin se, siirtyykö lukupää seuraavaan vai edelliseen merkkiin.

Harel kuva 9.3.

Lisäksi algoritmit muuttavat tietorakenteitaan. Toteutetaan tämä sallimalla lukupään ennen siirtymistään kirjoittaa kohdalla olevan merkin tilalle uusi merkki (eli se on **luku- ja kirjoituspää**).

Kyseinen minimaalinen algoritmien esitysmalli on **Turingin kone** (Alan Turing, 1936).

Turingin kone

Hieman tarkemmin: **Turingin kone (TM)** koostuu

- äärellisestä joukosta **tiloja**,
- äärellisestä **aakkostosta**,*
- äärettömästä **nauhasta** ja
- nauhaa merkkipositio kerrallaan käsittelevästä **nauhapäästä**

+ **siirtymäfunktiosta**, joka tilan ja nauhapään kohdalla olevan merkin perusteella määrää 1) seuraavan tilan, 2) merkkipositioon kirjoitettavan uuden merkin ja 3) nauhapään siirron oikealle tai vasemmalle.

≈ "Turingin koneen ohjelma"

Siirtymäfunktio voidaan esittää tilasiirtymäkaaviona (Ks. Harel kuva 9.4).

*Käytetään merkkiä # esittämään puuttuvaa tai poistettua merkkiä

Tilat, joista ei siirtymiä eteenpäin, ovat **lopputiloja**.

Suorituksen alku: kone nimetyssä **alkutilassa** ja nauhapää syötteen ensimmäisen merkin kohdalla. Suoritus etenee siirtymäfunktion ohjaamana, kunnes saavuttaa lopputilan. Kone hyväksyy tai hylkää syötteen sen mukaan, onko lopputila YES vai NO.

Esim. Palindromien tunnistus (Harel sivut 229–230).

Turingin kone ei ole rajoittunut päätösongelmien ratkaisemiseen. Voidaan sopia, että suorituksen päätyttyä nauhalla (esim. !-merkkien välissä) on sen laskema tuloste.

Esim. Kymmenjärjestelmän lukujen yhteenlasku (Harel sivu 232).

Nauhalla syötteenä X'+'Y

Siirtymäfunktion periaate erittäin karkeasti:

Kirjoita X:n vasemmalle puolelle merkki '!'

Kunnes jompikumpi luku loppuu, toista:

Paikanna ja poista luvun Y viimeinen numero y (siirry sen arvoa vastaavaan tilaan *nro*y, $y \in \{0, \ldots, 9\}$).

Paikanna ja poista luvun X viimeinen numero x sekä siirry arvoa $x + y$ vastaavaan tilaan *sum0*, *sum0+muistinro*, ... tai *sum9+muistinro*.

Kirjoita summan numero syötettä lähinnä edeltävään tyhjään positioon.

Toista samaan tapaan, muistaen (tiloissa) onko muistinumero otettava huomioon.

Jos toinen luku loppuu, toisen jäljelle jääneet numerot siirrettävä yksitellen (tarpeen vaatiessa muistinumero lisäten) keskeneräisen tulosteen alkuun.

Lopuksi lisää '!'-merkki tulostetta edeltävään nauhapositioon.

Esimerkkisuoritus (tila nauhan sisällön vieressä):

```
... # # # # # 8 9 + 1 2 3 # # ...

... # # # # ! 8 9 + 1 2 3 # # ...

... # # # # ! 8 9 + 1 2 # # # ... (nro3)

... # # # # ! 8 # + 1 2 # # # ... (sum2+muistinro)

... # # # 2 ! 8 # + 1 2 # # # ...

... # # # 2 ! 8 # + 1 # # # # ... (nro3; 2+muistinro)

... # # # 2 ! # # + 1 # # # # ... (sum1+muistinro)

... # # 1 2 ! # # + 1 # # # # ...

... # # 1 2 ! # # + # # # # # ... (nro2; 1+muistinro)

... # 2 1 2 ! # # + # # # # # ... (luvut-loppu)

... ! 2 1 2 ! # # + # # # # # ... (YES)
```

Mahdollista, mutta ohjelmointina *erittäin hankalaa!*

Miksi siis tarkastellaan Turingin koneita (tai muita primitiivisiä malleja)?.

Yksinkertaisissa malleissa mahdollista keskittyä algoritmien oleellisiin piirteisiin.

Laskettavuuden ja formaalikielten teorian sekä vaativuusteorian keskeiset tulokset perustuvat Turingin koneiden tarkasteluun.

Esim. Cookin teoreema (SAT on NP-täydellinen).

Vaativuusluokkien (P, NP, EXP jne.) täsmällinen määrittely perustuu Turingin koneisiin:

aikavaativuus = montako tilasiirtymää suoritettava

tilavaativuus = montako nauhapositiota käytettävä.

Church-Turingin teesi

Onko primitiivinen Turingin kone sitten riittävän voimakas laskennan malli?

Algoritmien esittämiseen on esitetty lukuisia malleja: mm. Churchin lambda-kalkyyli, Postin produktiosysteemit, Kleenen rekursiiviset funktiot ja nykyaikaiset korkean tason ohjelmointikielet.

Voidaan osoittaa, että että kaikki algoritmien esitystavat ovat yhtä voimakkaita. (Tätä varten ohjelmointikieliä idealisoitava siten, että muuttujien arvoina saa olla rajoittamattoman suuria arvoja.)

Perustuu havaintoon, että missä tahansa mallissa esitetyn algoritmin toimintaa voidaan **simuloida** jotain toista mallia käyttäen.

$\leadsto$ **Church-Turingin teesi** (CT-teesi):
Intuitiivinen algoritminen laskettavuus = laskettavuus Turingin koneella.

"Intuitiivinen algoritminen laskettavuus" ei ole täsmällinen käsite, mutta täysin erilaisten algoritmisten mallien havaittu ekvivalenssi tukee CT-teesiä.

Huom: CT-teesin nojalla esimerkiksi korkean tason kielellä johtamamme pysähtymättömyyongelman ratkeamattomuustulos siirtyy kaikkiin muihin (riittävän voimakkaisiin) laskennan malleihin.

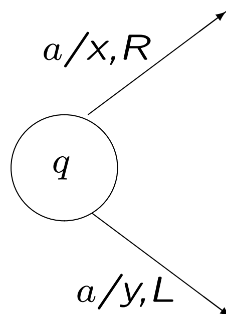
Laskettavuuden käsite on **robusti**, käytetystä mallista riippumaton.

Turingin koneen variaatiot

Turingin koneista on useita variaatioita, jotka voidaan nähdä laskentavoimaltaan yhteneviksi: ainoastaan toiseen suuntaan ääretön nauha, useampia kuin yksi nauha, kaksiulotteinen nauha, jne.

Harel, kuva 9.6.

Erityisen kiinnostava on **epädeterministinen Turingin kone** (NTM), jolla on yksikäsitteistä siirtymäfunktioita yleisempi **siirtymärelaatio**: Yhdestä tilasta q voi lähteä samalla nauhamerkillä a useampia siirtymiä:



Epädeterministisen Turingin koneen voi jälleen ajatella valintatilanteissa ”maagisesti” arvaavan YES-lopputilaan lopulta johtavat siirtymät, mikäli sellaiset on olemassa.

Epädeterminismi ei kuitenkaan laajenna periaatteellista laskettavuutta: Deterministisellä TM:llä voi (vaikkakin työläästi) *simuloida* epädeterministisen TM:n vaihtoehtoisia laskentoja ja tutkia, johtaako joku niistä YES-lopputilaan.

Jos kiinnitetään (aakkosto sekä) esitystapa Turingin koneiden (a) siirtymäfunktioille ja (b) konfiguraatioille (tilalle, pään paikalle ja käsiteltyjen nauhapaikkojen sisällöille), niin voidaan tehdä Turingin kone, joka tuottaa syötteistä (a) ja (b) joko (i) seuraavan konfiguraation tai (ii) tiedon ettei sellaista ole. Saadaan **universaalikone** tulkkaamaan muita(kin) Turingin koneita!

Mallien väliset simuloinnit

Perustellaan hieman korkean tason ohjelmointikielten ja Turingin koneiden laskentavoiman yhtenevyyttä.

Kohtalaisen helppo uskoa, että korkean tason ohjelmointikielellä voi simuloida Turingin koneiden toimintaa – opetuskäyttöön on ohjelmoitukin Turingin koneiden tulkkeja. (Täyttä simulointia varten täytyy tietysti olettaa idealisoitu ohjelmointikieli, jossa muistia on käytössä rajatta.)

Perustellaan hieman päinvastaista suuntaa, tietokoneohjelmien simulointia Turingin koneella.

Tietokoneohjelman suorituksen pohjana on konekielinen ohjelma, joko jotain prosessoria varten käännetty versio ohjelmasta tai sitä tulkitsevan ohjelman (esim. Java-virtuaalikoneen) koodi.

Konekieliohjelmat koostuvat yksinkertaisista konekäskyistä, jotka voivat käsitellä koneen muistipaikkojen sisältöä.

Laskennan ns. **RAM**-malli. (Tarkemmin ks. esim. Aho, Hopcroft & Ullman: Design and analysis of computer algorithms, luku 1. Addison-Wesley 1974.)

Yksityiskohdat ohittaen:

RAM-koneen muistipaikkojen sisältöjä voidaan ylläpitää TM:n nauhalla.

RAM-koneen konekieliohjelmaa vastaava siirtymäfunktio simuloi yksittäisten käskyjen suoritusta ja kontrollin kulkua ohjelmassa.

Muistipaikkojen sisällön paikantaminen ja muuttaminen nauhalla on hieman hankalaa, mutta ei liian työlästä: voidaan osoittaa, että simulointi vie enintään *polynomisen ajan* suhteessa RAM-koneen käyttämään aikaan.

Vastaava pätee (tietyin oletuksin) muidenkin algoritmisten mallien välisiin simulointeihin: Jos algoritmi toimii jossain laskennan perusmallissa polynomisessa ajassa, sitä voidaan simuloida myös muiden mallien mukaisilla algoritmeilla polynomisessa ajassa.

Siis myös ongelmien jako työläisiin (eksponentiaalisen ajan vaativiin) ja käytännössä (eli polynomisessa ajassa) ratkeaviin on robusti algoritmien esitystavan suhteen.

Huom: ”Alipolynomisissa” aikavaativuuksissa voi olla eroja. Korkean tason kielellä lineaarisessa ajassa suoritettavan algoritmin simulointi Turingin koneella voi vaatia esim. ajan $\Theta(n^2)$.

Cookin teoreema

Turingin koneiden pääasiallinen merkitys on ongelmien alarajatarkasteluissa (työläyden tai ratkeamattomuuden osoittamisessa).

Kuinka esimerkiksi voidaan sanoa, että jos lauselogiikan toteutuvuusongelma SAT voidaan ratkaista deterministisesti polynomisessa ajassa niin *mikä tahansa* epädeterministisesti polynomisessa ajassa ratkeava päätösongelma A voidaan ratkaista deterministisesti polynomisessa ajassa?

(Ts. SAT-ongelman NP-kovuus; Cook 1971)

Miten lausekalkyylin kaavan toteutuvuus liittyy sen ratkaisemiseen, onko syöte x ongelman A "kyllä"-tapaus?

Jos $A \in$ NP niin ongelma "$x \in A$" on ratkaistavissa jollain epädeterministisellä Turingin koneella M polynomisessa ajassa $p(|x|)$.

Turingin koneen yksinkertaisuuden takia on mahdollista (deterministisesti ja polynomisessa ajassa) muodostaa kaava F_x, joka kuvaa koneen M mahdollisia laskentoja syötteellä x; suoritus päättyy tilaan "kyllä" jos ja vain jos F_x on toteutuva.

Suoritusajan (ja -tilan) ylärajan $p(|x|)$ tunteminen "etukäteen" sallii *lause*logiikan käytön; vastaava "ei-äärellinen" tekniikka palauttaa pysähtymisongelman *1KL:n* toteutuvuuteen – ks. esim. G.S. Boolos & R.C. Jeffrey: Computability and Logic (3. painos), luku 10 (Cambridge University Press 1989).

Siis mikä tahansa NP-ongelma voidaan palauttaa polynomisesti ongelmaan SAT (eli SAT on NP-kova).

Kaakelointiongelman ratkeamattomuus

Tason kaakeloimisen ja Turingin koneen suorittamisen välillä on yllättävä yhteys, jonka avulla voidaan osoittaa kaakelointiongelman ratkeamattomuus palautuksella pysähtymisongelmasta.

Rajoitutaan kaakelointiongelman versioon, jossa kysytään, voiko koko äärettömän puolitason kaakeloida annetuilla laattatyypeillä siten, että kiinnitetty laatta t esiintyy alimmassa rivissä.

Palautus on seuraavanlainen:

Kukin kaakeloinnin vaakarivi saadaan sopivalla laattavalikoimalla vastaamaan yhtä Turingin koneen M suorituksen vaihetta.

Laattojen värejä käytetään koodaamaan koneen nauha-aakkoston merkkejä sekä tilojen ja merkkien pareja. (Molempia on äärellisesti!) (Tässä värit esitetään suoraan niiden koodaamina symboleina tai symbolipareina.)

Alin kaakelirivi saadaan esittämään laskennan alkutilannetta syötteellä $a_1, \ldots, a_n$ seuraavasti:

Laatta t: [laatta: ylhäällä q_0, a_1; vasemmalla 0; oikealla 1] kuvaa alkutilan q_0 ja tiedon, että nauhapää on ensimmäisen syötemerkin a_1 kohdalla.

Loppuja syötteen merkkejä $a_2, \ldots, a_n$ esittävät laatat saadaan pakotettua oikeille paikoilleen muodostamalla laattatyypit

[laatta: ylhäällä a_2; vasemmalla 1; oikealla 2], ..., [laatta: ylhäällä a_n; vasemmalla $n-1$; oikealla n] .

Syötettä edeltäviä ja seuraavia tyhjämerkkejä esitetään laatoilla

	#	
0		0

ja

	#	
n		n

Ks. Harel kuva 9.10.

Seuraavia kaakelirivejä (eli suorituksen tilanteita) varten suurin osa nauhamerkeistä säilyy ennallaan. Tämä saadaan aikaan laattatyypillä

	c	
	c	

kutakin aakkoston merkkiä c kohden.

Koneen tilan ja nauhapään position esitys saadaan siirrettyä seuraavaan kaakeliriviin laatoilla

	b	
		r
	q, a	

ja

	r, c	
r		
	c	

jokaiselle nauhamerkille c ja siirtymälle $q \xrightarrow{a/b,R} r$.

Nauhapään oikealle siirtävä siirtymä esitetään oleellisesti yllä olevan peilikuvana.

Muodostetuilla laattatyypeillä voidaan kaakeloida koko puolitaso täsmälleen silloin, kun Turingin koneen M suoritus syötteellä $a_1, \ldots, a_n$ ei pääty.

Konstruktio siis palauttaa kaakelointiongelman pysähtymisongelmaan (Harel, kuva 9.9) – tarkemmin sanoen sen komplementtiin.

$\leadsto$ Kaakelointiongelma on ratkeamaton.

(Jos nimittäin sekä ongelma itse että sen komplementti ovat osittain ratkeavia, niin se onkin kokonaan ratkeava – HT.)

Äärelliset automaatit

Turingin koneilla on myöskin laskentavoimaltaan rajoittuneempia versioita, jotka ovat silti hyödyllisiä .

Jos rajoitetaan Turingin koneen nauhapää etenemään vain oikealle päin, sen kirjoittamilla merkeillä ei ole laskennan etenemisen kannalta merkitystä. (Miksi?)

Saatu yksinkertainen malli, **äärellinen automaatti** siis vain käy syötemerkit kertaalleen läpi ja suorittaa niiden ohjaamana siirtymiä tilasta toiseen.

Yksinkertaisuudestaan huolimatta äärellisillä automaateilla on tietojenkäsittelyssä huomattava merkitys esim. merkkijonojen tunnistusmenetelmänä.

Esim. Desimaalilukujen tunnistaminen

Äärellisiä automaatteja voi käyttää myös reaktiivisten järjestelmien esittämiseen ja analysointiin; syötejonona tällöin järjestelmään vaikuttavat tapahtumat.

Ks. Harel kuva 9.14.

Turingin koneitten, äärellisten automaattien sekä laskentavoimaltaan näiden väliin sijoittuvien mallien tarkastelu luo perustan **automaattien ja formaalikielten** tutkimukselle.