Ongelman vaativuuden rajat

Onko algoritmiselle ongelmalle löydetty ratkaisualgoritmi riittävän hyvä?

Olisiko mahdollista löytää asymptoottisesti tehokkaampi ratkaisu, vai onko algoritmi **optimaalinen**?

Kysymyksiin vastaamiseksi meidän on arvioitava **laskentaongelman vaativuutta**:

Kuinka paljon aikaa (tai tilaa) ongelman P tapausten ratkaiseminen vaatii syötteen koon funktiona?

Löydetyn algoritmin A vaativuus muodostaa **ylärajan** ratkaistavan ongelman vaativuudelle:

Ongelma P voidaan ratkaista tarvitsematta ainakaan enemmän resursseja mitä algoritmi A vaatii.

Toisaalta saatamme pystyä todistamaan ongelman vaativuuden **alarajoja**:

Kaikki (tietyntyyppiset, esim. kiinnitettyihin perusoperaatioihin perustuvat) ongelman P ratkaiset algoritmit vaativat *vähintään* tietyn määrän aikaa tai tilaa.

Ks. Harel kuva 6.6.

Useimmat tunnetut alarajatodistukset koskevat jotakin *rajoitettua laskentamallia* jossa sallitaan vain tietyt perusoperaatiot tietoalkioille tai kontrollivuolle.

Esim. Järjestetystä taulukosta etsinnän alaraja

Voimme osoittaa, että binäärihaku on optimaalinen menetelmä arvon etsimiseen järjestetystä taulukosta:

Rajoitumme algoritmeihin, jotka kohdistavat syötteisiin, etsittävään arvoon ja taulukon alkioihin, ainoastaan kahden operandin välisiä vertailuja $<$, $\leq$, $=$, $>$ ja $\geq$. Eli alkion rakennetta (esim. bittihahmoa) *ei* saa käyttää!

Osoitamme, että jokainen tällainen algoritmi A suorittaa pahimmassa tapauksessa vähintään $\log_2 n$ vertailua:

Oletetaan, että haettava arvo Y esiintyy kasvavassa järjestyksessä olevassa taulukossa L[$1 \ldots n$].

Algoritmin A suorituksia yhdellä taulukolla L ja eri arvoilla Y voi kuvata **päätöspuulla**.

Ks. Harel kuva 6.7.

Päätöspuun haarautumasolmut vastaavat algoritmin tekemiä syötearvoihin kohdistuvia vertailuja, lehtisolmut algoritmin tuottamia tuloksia.

Vertailujen välissä algoritmi voi suorittaa jonon muita operaatioita.

Suoritukset haarautuvat kahteen vaihtoehtoon kunkin vertailun tuloksen perusteella (jos algoritmi ei tee turhia vertailuja), joten päätöspuu on binääripuu.

Yksittäinen suoritus = polku juuresta lehteen.

Päätöspuussa on oltava ainakin n lehteä: algoritmin on löydettävä Y jokaisesta taulukon $L[1 \ldots n]$ paikasta.

Binääripuussa, jonka korkeus on h, on enintään 2^h lehteä (helppo induktio)
$\rightsquigarrow$ n-lehtisen binääripuun korkeus on vähintään $\log_2 n$.

$\rightsquigarrow$ Jokin algoritmin A suoritus vaatii vähintään $\log_2 n$ vertailua.

$\rightsquigarrow$ Binäärihaku on asymptoottisesti optimaalinen algoritmi etsintään järjestetystä taulukosta.

Vastaavalla päätöspuuhun perustuvalla argumentoinnilla voidaan osoittaa, että arvojen vertailuun perustuva n-alkioisen listan tai taulukon lajittelu vaatii vähintään $\Theta(n \log n)$ vertailua.

$\rightsquigarrow$ Lomituslajittelu on eräs lukuisista asymptoottisesti optimaalisista lajittelumenetelmistä.

Muita esim. pika- (quick-) ja kekolajittelut (heapsort).

Monille ongelmille tunnetaan tehokkaita algoritmeja, joista osa on optimaalisia (vaativuus vastaa ongelman vaativuuden alarajaa).

Jatkossa näemme, että lukuisten ongelmien tarkka vaativuus on avoin: emme osaa ratkaista niitä tehokkaasti, vaikka tehokkaan ratkaisun olemassaolo on periaatteessa mahdollista.

5. Työläät ongelmat

(Harel luku 7)

"Eikä tätä asiaa saa hoidetuksi päivässä tai parissa"
[Esra 10:13]

• Riittääkö algoritmien tehokkuus kaikkien laskentaongelmien ratkaisemiseen?

Esim. Hanoin tornien ratkaisemisen työläys

Montako siirtoa Move(N,X,Y,Z)-proseduuri tulostaa?

Tarkastellaan suorituksen Move(N,X,Y,Z) kutsupuuta.

Puun jokaisessa solmussa tulostetaan yksi siirto. Paljonko solmuja on?

Lasketaan solmut kussakin syvyydessä.

(Solmun **syvyys** puussa on juuresta siihen johtavan polun pituus.)

Syvyydessä 0 on vain juuri Move(N,A,B,C).

Syvyydessä 1 on 2 solmua (Move(N-1,A,C,B) ja Move(N-1,C,B,A).

...

Lehtitasolla, syvyydessä $N-1$, on 2^{N-1} solmua.

$\rightsquigarrow$ yhteensä $1+2+\cdots+2^{N-1}=2^N-1$ solmua.

Moves-algoritmin aikavaativuus on **eksponentiaalinen!**

Jos esim. sekunnissa voidaan suorittaa miljoona Moves-kutsua, 64 kiekon ongelman siirtojen laskenta vie *yli puoli miljoonaa vuotta*!

Algoritmin eksponentiaalisuus on väistämätön, sillä sen tulosteen pituus on eksponentiaalinen.

Ongelman ratkaisu voi vaatia eksponentiaalisen ajan, vaikka tuloste olisikin lyhyt, esim. ”kyllä” /”ei”.

(Ns. **päätösongelmat**.)

Esim. ”Apinapalapeli”

Järjestettävä $n = m \times m$ neliönmuotoista korttia m-sivuiseksi neliöksi s.e. korttien värilliset kuviot osuvat kohdakkain.

Ks. Harel kuva 7.1.

Oletetaan, että kortteja ei saa pyörittää (niissä on vaikka taustana maisemakuva, joka kiinnittää niiden ylös-alas-suunnan).

Apinapalapelin päätösongelma: Voidaanko kortit järjestää halutulla tavalla?

Suoraviivainen ratkaisu:

Generoi kaikki mahdolliset järjestykset (esim. riveittäin alkaen alueen vasemmasta yläkulmasta). Ilmoita ”kyllä”, jos löytyy järjestys, jossa kaikkien vierekkäisten korttien reunat sopivat toisiinsa. Muuten ilmoita ”ei”.

Pahimmassa tapauksessa käytävä kaikki $n!$ järjestystä läpi.

Esim. 5×5-palapelin tapauksessa järjestyksiä on $25! = 1*2* \cdots * 25 \approx 15 * 10^{24}$. Jos tarkastetaan sekunnissa miljoona järjestystä, aikaa menee yli $491 * 10^{12}$ vuotta!

Apinapeliongelmaan (ja satoihin samankaltaisiin) ei tunneta pahimmassa tapauksessa oleellisesti tehokkaammin toimivaa ratkaisua.

Polynominen vs. ylipolynominen aika

Algoritmisen tehokkuuden vedenjakajana pidetään sitä, onko algoritminen ongelma

• ratkaistavissa **polynomisella** algoritmilla, ajassa $O(n^k)$ jollain kiinteällä k, vai

• vaatiiko sen ratkaiseminen (pahimmassa tapauksessa) **eksponentiaalisen** määrän (c^n, $c > 1$, tai enemmän) aikaa tai tilaa.

Ongelmia, joille on olemassa *polynomisessa* ajassa toimiva ratkaisualgoritmi, sanotaan **käytännössä ratkeaviksi** (tractable).

Ongelmia, joille on olemassa vain *eksponentiaalisen* ajan vaativia ratkaisualgoritmeja, sanotaan **työläiksi** (intractable).

Polynomisuuteen/ylipolynomisuuteen perustuva jako erottaa käytännössä käyttökelpoiset ja käyttökelvottomat algoritmit selvästi, vaikka teoriassa polynomisen algoritmin vaativuus voisi olla esim. $O(n^{1000})$.

Ks. Harel kuva 7.4.

Tarkastelujen perusteella algoritmiset ongelmat voidaan siis jakaa kahteen luokkaan: työläisiin ja käytännössä ratkeaviin.

Ks. Harel kuva 7.6.

Väliin jää **rajavyöhyke** jonka ongelmille *tunnetaan* vain eksponentiaalinen algoritmi vaikka polynominenkin *voi* olla olemassa (ja odottaa löytäjäänsä).

NP-täydelliset ongelmat

Kukaan ei ole osannut todistaa ylipolynomista alarajaa apinapalapelin vaativuudelle.

Toisaalta yhtään polynomista ratkaisualgoritmia ei tunneta.

Muita vastaavia **NP-täydellisiä** ongelmia tunnetaan satoja, useilta eri aloilta (mm. verkkoteoria, suunnittelu ja skedulointi, pelit, logiikka, ohjelmien optimointi).

Useat NP-täydelliset ongelmat ovat käytännössä esiintyviä keskeisiä ongelmia.

Tutustutaan muutamiin tyypillisiin NP-täydellisiin ongelmiin.

Kauppamatkustajan ongelma
(TSP, travelling salesman problem)

Annettuna painotettu verkko
(”kaupungit ja niiden väliset etäisyydet”).

Mikä on kokonaispituudeltaan lyhin reitti,
joka käy kertaalleen kussakin kaupungissa
(eli lyhin ns. **Hamiltonin kehä**)?

Vastaava päätösongelma: annettuna verkon
lisäksi maksimipituus K, onko verkossa
kauppamatkustajan reittiä pituudeltaan $\leq K$?

Harel kuva 7.9.

Huom: Kauppamatkustajan ongelma ei ole
”leikkiongelma” – tärkeä mm.
piirisuunnittelussa

Kauppamatkustajan päätösongelma voidaan ratkaista suoraviivaisesti kuten apinapalapeli: Generoi kaikki $N!$ kaupunkien järjestystä ja tarkasta kustakin, muodostaako se kehän, jonka pituus $\leq K$.

Kuten apinapalapelin tapauksessa, pahimmassa tapauksessa toivotonta esim. kun $N \geq 25$.

Kauppamatkustajan ongelmalle läheistä sukua on toinen NP-täydellinen ongelma, **Hamiltonin polku** -ongelma:

Onko annetussa verkossa polkua, joka kulkee täsmälleen kerran kunkin solmun kautta?

Ks. Harel kuva 7.10.

Pinnallisesti samankaltaisen ongelmien vaativuus voi kuitenkin erota huomattavasti:

Eulerin polku -ongelmassa kysytään, onko annetussa verkossa polkua, joka kulkee täsmälleen kerran jokaista verkon *kaarta* pitkin.

Ks. Harel kuva 7.11.

Voisi vaikuttaa, että tämänkin ongelman ratkaisemiseksi pitäisi pahimmillaan tarkastella kaikkia mahdollisia polkuja.

Kuitenkin voidaan osoittaa, että verkossa G on Eulerin polku (Euler 1736) täsmälleen silloin, kun
(1) G on yhtenäinen ja
(2) jokaisesta solmusta (mahdollisesti lukuunottamatta polun alkua ja loppua) lähtee parillinen määrä kaaria.

$\rightsquigarrow$ Eulerin polku -ongelma on ratkaistavissa tehokkaasti (ajassa $O(|E|)$, missä $|E|$ on verkon kaarien lukumäärä).

Lauselogiikan toteutuvuusongelma

Kenties keskeisin NP-täydellinen ongelma liittyy yksinkertaisten loogisten kaavojen toteutuvuuteen.

NP-täydellisyyden käsite määriteltiin vuonna 1971 juuri tämän ongelman kautta Stephen Cookin artikkelissa ”The Complexity of Theorem Proving Procedures” joka jatkoi loogikko Kurt Gödelin (välillä unohdettua) työtä.

Samalla nähdään yksinkertainen esimerkki kalvoilla 64–66 mainitusta loogisesta kielestä muoto- ja merkitysoppeineen.

Lauselogiikan formaalikieli koostuu *muuttujasymboleista* $\mathcal{X} = \{X_i : i \in \mathbb{N}\}$ ja induktiivisista *kaavanmuodostussäännöistä:*

Perustapaus: Jokainen muuttujasymboli on kaava.

Konjunktio: Jos A ja B ovat kaavoja, niin myös $(A \wedge B)$ on kaava.

Disjunktio: Jos A ja B ovat kaavoja, niin myös $(A \vee B)$ on kaava.

Negaatio: Jos A on kaava, niin myös $\neg A$ on kaava.

Implikaatio: Jos A ja B ovat kaavoja, niin myös $(A \to B)$ on kaava.

Merkitys annetaan
totuusarvojakeluilla $f\colon \mathcal{X} \mapsto \{\textbf{tosi}, \textbf{epätosi}\}$
jotka yleistyvät induktiolla:

- $f(A \wedge B)$ on tosi jos ja vain jos sekä $f(A)$ että $f(B)$ ovat **tosi**a.

- $f(A \vee B)$ on **tosi** jos ja vain jos edes toinen arvoista $f(A)$ tai $f(B)$ on **tosi**.

- $f(\neg A)$ on **tosi** jos ja vain jos $f(A)$ on **epätosi**.

- $f(A \rightarrow B)$ on **epätosi** jos ja vain jos $f(A)$ on **tosi** vaikka $f(B)$ on **epätosi**.

  (Ns. *materiaalinen* implikaatio, ei aina sama kuin semanttinen syy-seuraus-suhde.)

Toteutuvuusongelmassa (satisfiability, SAT) kysytään, onko annettu lauselogiikan kaava toteutuva, s.o, tuleeko se todeksi jollain totuusarvojakelulla.

Esim.

$$\neg(E \to F) \wedge (F \vee (D \to \neg E))$$

on toteutuva sijoituksella
$\{E :=$ **tosi**$, F :=$ **epätosi**$, D :=$ **epätosi**$\}$.
Toisaalta

$$\neg((D \wedge E) \to F) \wedge (F \vee (D \to \neg E))$$

ei ole toteutuva. (Miksi?)

Jos kaavassa on n muuttujaa, sen toteutuvuus on suoraviivaista tarkastaa tutkimalla mahdolliset 2^n totuusarvosijoitusta.

Toisaalta pahimmassa tapauksessa oleellisesti tehokkaampaa menetelmää ei tunneta.

Useat NP-täydelliset ongelmat liittyvät skedulointiin tai sovittamiseen:

Esim. lukujärjestyksen laatiminen
$\approx$ ns. kolmiulotteinen paritus.

NP-täydellisten ongelmien tyypillisiä piirteitä:

- Osaongelmien ratkaisemiseksi tehtävä valintoja, jotka estävät muita jatkovalintoja
  $\rightsquigarrow$ ahne lähestymistapa ei toimi.
- Osaongelmia on ylipolynominen määrä, esim. kaikki järjestykset ($n!$) tai kaikki osajoukot (2^n)
  $\rightsquigarrow$ dynaaminen ohjelmointi (osaongelmien ratkaisujen tallettaminen) ei toimi.
- Ei tunneta keinoja karsia osaongelmia etukäteen tutkimatta niitä.

NP-täydellisen ongelman ratkeavilla (”kyllä”-) tapauksilla on kuitenkin yksinkertainen (polynomisen tai usein lineaarisen kokoinen) **todiste**, josta on helppo vakuuttua ogelman tapauksen ratkeavuudesta.

Esimerkkejä todisteista: apinakorttien järjestys, polku, totuusarvosijoitus.

Todisteen tarkastamisen helppous tarkoittaa, että se on tehtävissä **polynomisessa ajassa**.

A onkin NP-ongelma jos ja vain jos on polynomi p ja polynominen algoritmi B joilla kysymys ”onko tapauksella x ongelman A vastaus **kyllä**” voidaan kääntää kysymykseksi ”onko olemassa jokin y kokoa p(x:n koko) jolla $B(x, y) =$ **kyllä**”.

Luokka NP

NP-täydelliset ongelmat ovat tehokkaasti (s.o. polynomisessa ajassa) ratkaistavissa **epädeterministisillä algoritmeilla**.

Epädeterministisillä algoritmeilla on käytössään (hypoteettinen) perusoperaatio **choose** A **or** B, joka valitsee käskyistä A ja B suoritettavaksi sen "oikean" joka aikanaan johtaa lopputulokseen **kyllä** (jos mahdollista).

Sen avulla algoritmi voi tehdä suorituksen onnistumisen kannalta parhaita mahdollisia ratkaisuja

mutta

se tarvitsee "kokonaisnäkemystä" laskennan tulevaisuudesta joten se ei enää olekaan paikallinen vain algoritmin nykytilan määräämä konekäsky!

Eräitä tapoja ymmärtää choose:

Onnenlantti jota heitettäessä saadaan aina oikea vastaus (jos sellainen on).

(Oikean valinnan n vaihtoehdon joukosta voi tehdä heittämällä lanttia $\log n$ kertaa.)

Jakaudutaan laskennassa A- ja B-haaroihin, jotka etenevät toisistaan riippumatta ('eri prosesseina") ja riittää, että jokin haara huutaa "kyllä!"

(Saadaan *epädeterministisen laskentapuun* käsite.)

Etsitään rekursiivisesti vaihtoehdot A ja B mutta suoritusajassa laskutetaan vain siitä vaihtoehdosta joka vastaa "kyllä".

(Vertaa todiste-ajatus.)

NP-täydelliset ongelmat voidaan ratkaista epädeterministisillä algoritmeilla seuraavalla periaatteella:

1. Generoi epädeterministisesti todiste ongelman tapauksen ratkeamisesta;

2. Tarkista, osoittaako todiste tapauksen ratkeavan; jos kyllä, palauta "yes", muuten palauta "no".

Todisteen pituus polynominen ja se voidaan tarkastaa polynomisessa ajassa $\leadsto$ kumpikin askel voidaan suorittaa polynomisessa ajassa.

(Polynomi q polynomista $p(n)$, $q(p(n))$, on polynomi. Esim
$5(3n^3 + 15n)^5 + 20(3n^3 + 15n)^2)$

Esim. toteutuvuusongelma voidaan y.o. periaatteella ratkaista epädeterministisesti polynomisessa (itse asiassa jopa lineaarisessa ajassa):

1. Generoi epädeterministisesti totuusarvosijoitus annetun kaavan F muuttujille.

2. Sovella sijoitusta kaavan F muuttujiin ja tarkista onko tulos tosi.
Jos on, tulosta "yes", muuten tulosta "no".

Epädeterministisellä polynomisessa ajassa toimivalla algoritmilla ratkaistavat ongelmat muodostavat **ongelmaluokan NP**.

Merkitsemme esim. SAT $\in$ NP.

NP-täydellisten ongelmien ”täydellisyys”

Luokassa NP on (luultavasti)* muitakin ogelmia kuin NP-täydelliset ongelmat.

NP-täydelliset ongelmat ovat sikäli erittäin mielenkiintoisia, että

1. ne ovat työläydeltään ekvivalentteja: joko kaikki työläitä tai kaikki polynomisessa ajassa ratkeavia

2. ne ovat työläimpiä luokan NP-ongelmista (”täydellisiä”): jos *yksikin* NP-täydellinen ongelma on ratkaistavissa deterministisellä (s.o. ”normaalilla”) polynomisessa ajassa toimivalla algoritmilla, niin *kaikki* luokan NP ongelmat ovat käytännössä ratkeavia.

*Vaativuusteoriassa on lukuisia konjektuureja, joita ei ole pystytty todistamaan. Asioita tarkastellaan perusteellisemmin erityisesti kurssilla *Laskennan vaativuusteoria*.

Edellisiin ominaisuuksiin liittyy vaativuusteorian tunnetuin ja keskeisin avoin ongelma: Päteekö $\mathbf{P} = \mathbf{NP}$, missä **P** on polynomisessa ajassa deterministisesti ratkeavien ongelmien luokka. Emme tiedä, onko maaginen epädeterminismi välttämätöntä luokan NP ongelmien tehokkaaseen ratkaisemiseen. Yleisesti uskotaan, että P$\neq$NP.

Kuinka *kaikkia* luokan NP ongelmia koskevia väitteitä (2. yllä) voidaan todistaa?

Cook osoitti v. 1971 SAT-ongelman NP-täydellisyyden: Jos SAT osataan ratkaista deterministisesti polynomisessa ajassa, niin jokainen luokan NP ongelma osataan ratkaista deterministisesti polynomisessa ajassa.

Sen jälkeen muita ongelmia on osoitettu NP-täydellisiksi käyttäen **polynomista palautusta**.

Ongelmien palauttaminen toisiin(sa)

Miten kahden ongelman A ja B vaikeutta voidaan vertailla?

Jos molempien tarkka vaativuus tunnetaan, niin helppoa.

Mutta entä jos ei tunneta (kuten juuri NP-ongelmille)?

Yleinen periaate: *palautetaan* ongelman A ratkaiseminen toisen ongelman B ratkaisemiseen, ja päätellään siitä ettei A ole ainakaan vaikeampi kuin B.

Toisin sanoen, *oletetaan* että B osattaisiin ratkaista, ja osoitetaan että silloin myös A osattaisiin ratkaista "pienellä lisätyöllä".

Lisätyötä sallitaan *vähemmän* kuin ongelmien oletettu vaikeustaso.

Esimerkiksi ongelmat

$A \equiv$ "Onko elämällä tarkoitus?" ja
$B \equiv$ "Onko Jumala olemassa?"

ovat sellaisia, joiden ratkaisemisen vaikeutta emme tunne.

Mutta ongelma A voidaan palauttaa ongelmaksi B järkeilemällä "Jos Jumala olisi olemassa, niin Hän ei olisi luonut elämää vain huvin vuoksi, joten silloin elämällä olisi tarkoitus."

Siis A ei ole ainakaan vaikeampi kuin B.

Toisaalta voisi myös järkeillä että "jos elämällä olisi tarkoitus, niin jonkin on pitänyt se tarkoitus asettaa".

Silloin ongelma B vuorostaan palautuisi ongelmaksi A, eli ne olisivat *yhtä vaikeita*.

(Lisävaivaa ei käytetty liikaa, koska järkeilyn suoritti ihminen. . .)

Polynomiset palautukset

Algoritmisen ongelman A palautuksella ongelmaan B tarkoitetaan konstruktioa, joka osoittaa, että A on ratkaistavissa käyttämällä apuna B:n ratkaisevaa algoritmia, mikäli sellainen on olemassa.

Polynominen palautus päätösongelmasta A päätösongelmaan B on polynomisessa ajassa laskettava muunnos $f()$, jolla syöte x on ongelman A "kyllä"-tapaus joss $f(x)$ on ongelman B "kyllä"-tapaus.

Polynominen palautus ongelmasta A ongelmaan B tarkoittaa, että A on ratkaistavissa polynomisessa ajassa *mikäli* B on.

Tarkastellaan esimerkiksi Hamiltonin polku -ongelman palauttamista kauppamatkustajan ongelmaan.

Olkoon verkko G käsiteltävä Hamiltonin polku -ongelman tapaus; ratkaistavana siis, onko verkossa jokaisen solmun kautta täsmälleen kerran kulkevaa polkua.

Muutetaan G painotetuksi verkoksi G', jossa on kaari kaikkien verkon solmuparien välillä; jos kaari kuuluu alkuperäiseen verkkoon G, sen paino verkossa G' on 1, muuten sen paino on 2.

Ks. Harel kuva 7.12.

Muunnos voidaan (”selvästi”) tehdä polynomisessa ajassa.

Muunnos on palautus ongelmien välillä: Olkoon n solmujen lkm verkossa G. Verkossa G on Hamiltonin polku joss verkossa G' on kauppamatkustajan reitti, jonka pituus on enintään $n + 1$.

Mikäli TSP-ongelmalla olisi tai sille löydetään polynomisessa ajassa toimiva algoritmi, palautus ja k.o. algoritmi yhdessä muodostavat Hamiltonin polku -ongelman polynomisessa ajassa ratkaisevan algoritmin.

Ks. Harel kuva 7.13.

Monet ongelmien väliset palautukset ovat monimutkaisempia kuin edellinen, mutta niiden periaate on sama.

Yhteenvetona: Tekniseltä kannalta ongelma A on NP-täydellinen, jos

1. A kuuluu luokkaan NP eli on ratkaistavissa epädeterministisellä polynomisessa ajassa toimivalla algoritmilla, ja

2. jokaisen luokan NP ongelman ratkaiseminen voidaan palauttaa polynomisessa ajassa ongelman A ratkaisemiseen.

Jälkimmäisestä ominaisuudesta sanotaan, että A on **NP-kova** (NP-hard).

NP-kovuus osoitetaan yleensä palautuksella jostain aiemmin NP-täydelliseksi (eli -kovaksi) tiedetystä ongelmasta.

NP-täydellisten ongelmien approksimointi

Useat NP-täydelliset päätösongelmat (esim. TSP) ovat kyllä/ei-versioita **kombinatorisista optimointiongelmista**.

Optimointiongelmat eivät ole niitä vastaavia päätösongelmia helpompia, joten niitäkin kutsutaan NP-täydellisiksi.

NP-täydellisten optimointiongelmien tarkkoja ratkaisuja pyritään **approksimoimaan**.

Esim. Geometrinen TSP.

Piirisuunnittelussa esiintyvä ongelma: Annettuna n pistettä tasossa, mikä on lyhin kunkin pisteen kautta kertaalleen kulkeva kehä? (Jokaisen pisteparin välillä mahdollinen yhteys, jonka pituus normaali geometrinen etäisyys.)

Ongelma voidaan osoittaa NP-täydelliseksi.

Polynomisessa ajassa voidaan (ns. Christofidesin algoritmilla) löytää kehä, joka on enintään 50% optimaalista pidempi

Enintään 2 $\times$ optimaalisen pituinen kehä C voidaan löytää seuraavalla periaatteella:

1. Muodosta annetun verkon pienin virittävä puu T.

2. Muodosta polku P kulkemalla kukin puun T kaari kahdesti.

3. Muuta polku P Hamiltonin kehäksi C oikaisemalla jo vierailtujen solmujen ohi.

Approksimoidun kehän pituus voidaan arvioida seuraavasti:

Jos C^* on lyhin mahdollinen Hamiltonin kehä, virittävän puun T kokonaispituus $L(T)$ on enintään sen kokonaispituus $L(C^*)$: T on halvin tapa yhdistää kaikki solmut puuksi, kukin ($C^* -$ yksi kaari) taas jokin tapa yhdistää ne poluksi.

Polun P pituus on $2L(T)$, ja koska polun P oikaiseminen kehäksi C ei voi pidentää polkua,

$$L(C) \leq L(P) = 2L(T) \leq 2L(C^*)$$

Tällä periaatteella löydetyn Hamiltonin kehän C pituus on siis enintään kaksi kertaa lyhimmän Hamiltonin kehän C^* pituus.

Todistettavasti työläät ongelmat

NP-täydellisten ongelmien täsmällinen vaativuus on avoin ongelma, vaikka niiden uskotaan melko yleisesti olevan aidosti työläitä.

Muita ongelmia on pystytty *todistamaan* työläiksi osoittamalla niillä olevan eksponentiaalisia aikavaativuuden alarajoja.

Tällaisia esimerkiksi yleistetyt (N$\times$N-laudalla pelattavat) lautapelit kuten shakki, tammi ja Go.

Myös ohjelmien verifiointiin liittyvä dynaaminen lauselogiikka (PDL, propositional dynamic logic).

Ongelmien vaativuusluokat

Ongelmat, jotka ovat ratkaistavissa deterministisesti tai epädeterministesti polynomisessa ajassa sekä NP-täydelliset ongelmat muodostavat vastaavat **ongelmaluokat** P (joskus PTIME), NP (joskus NPTIME) ja NPC.

Ongelmaluokkien välisten suhteiden selvittäminen muodostaa aktiivisen **laskennan vaativuusteorian** tutkimuskentän.

Tarkastelemalla logaritmisella (LOG), polynomisella (P) tai eksponentiaalisella (EXP) määrällä aika- (TIME) tai tilaresursseja (SPACE) ratkaistavissa olevia ongelmia saadaan erilaisia ongelmien vaativuusluokkia. Epädeterminismin salliminen (N) saattaa lisäksi muuttaa annetuilla resursseilla ratkaistavissa olevien ongelmien joukkoa.

Vaativuusluokkien yleisiä suhteita tiedetään, mutta monet kysymykset ovat avoimia.

Ks. Harel kuva 7.15.

Esimerkiksi on varsin selvää, että sallimalla epädeterminismi polynomisessa ajassa ratkeavien ongelmien joukko ei ainakaan suppene, ja että polynomisessa ajassa ratkaistavat ongelmat eivät voi vaatia ylipolynomista muistitilaa. Vaativuusluokkien suhteina ilmaistuna

$$\text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE}.$$

Kuitenkaan ei tiedetä, onko jompikumpi tai kumpikaan y.o. sisältyvyyksistä aito.

Kalvolla 126 esiteltiin NP todisteiden avulla: $A \in$ NP jos ja vain jos on $B \in$ P ja polynomi p siten, että

$$x \in A \Leftrightarrow \underline{\exists \text{ bittijono } y.\, |y| \leq p(|x|)} \wedge \langle x, y \rangle \in B.$$

Negaatio eli komplementti synnyttää luokan co-NP: $\bar{A} \in$ co-NP jos ja vain jos on $B \in$ P ja polynomi p siten, että

$$\begin{aligned} x \in \bar{A} &\Leftrightarrow \neg\exists y.\, |y| \leq p(|x|) \wedge \langle x, y \rangle \in B \\ &\Leftrightarrow \forall y.\, |y| \leq p(|x|) \rightarrow \underbrace{\langle x, y \rangle \notin B}_{\bar{B} \in \mathsf{P}} \end{aligned}$$

Operaatio <u>"on polynominen todiste"</u> ja komplementointi synnyttävät *polynomisen hierarkian* luokkien P ja PSPACE välimaastoon.

Esimerkiksi

$$\exists z.\, |z| \leq q(|x|) \wedge \forall y.\, |y| \leq p(|x|) \rightarrow \langle x, y, z \rangle \notin B$$

antaa ongelman joka on tämän hierarkian *toisen tason positiivisella puolella* Σ_2^p:

- Taso 0 on $\mathsf{P} = \Sigma_0^p = \Pi_0^p$.

- Tason $n+1$ positiivinen puoli Σ_{n+1}^p saadaan edellisen tason n negatiivisesta puolesta Π_n^p lisäämällä uusi polynominen todiste. Siis $\mathsf{NP} = \Sigma_1^p$.

- Tason $n+1$ negatiivinen puoli Π_{n+1}^p saadaan saman tason $n+1$ positiivisesta puolesta Σ_{n+1}^p komplementoimalla. Siis co-$\mathsf{NP} = \Pi_1^p$

Kiinnostava esimerkiksi peleissä: ”minulla on siirto, jonka jälkeen kaikilla sinun siirroillasi minä olen voittanut”.

Vaativuusluokkien määrittelyt ja niitä koskevat tulokset (esim. Cookin teoreema) perustuvat algoritmien mahdollisimman yksinkertaisiin perusmalleihin, erityisesti ns. Turingin koneisiin. Tutustumme näihin myöhemmin.

NP-täydellisiä ongelmia ei siis *osata* ratkaista *tehokkaasti*, ja

todistettavasti vaativia ongelmia ei *pystytä* ratkaisemaan *tehokkaasti*.

Seuraavaksi kohtaamme hyvin määriteltyjä algoritmisia ongelmia, joita ei *pystytä lainkaan* ratkaisemaan, vaikka resursseja olisi käytössa mielivaltaisesti.