Laskennallisesta vaativuudesta

Oletamme jatkossa, että laskennan mallina ovat *Turing-koneet*.

Turing-kone *hyväksyy* merkkijonojen joukon O, jos Turing-kone pysähtyy syötteellä i hyväksyvään tilaan jos ja vain jos $i \in O$.

Esitämme ongelmat merkkijonoina ja käytämme Turing-koneita näiden merkkijonojen hyväksymiseen (ts. ongelman ratkaisemiseen).

Kaksi keskeistä ongelmien luokkka ovat

- **P**: ongelmat, jotka voidaan ratkaista polynomisessa ajassa *deterministisellä* Turing-koneella.
- **NP**: ongelmat, jotka voidaan ratkaista polynomisessa ajassa *epädeterministisellä* Turing-konella.

Luokka **P** on luokan **NP** aliluokka (ja mitä ilmeisimmin aito).

Esimerkki. Lauselogiikan toteututuvuusongelmassa on tarkoituksena selvittää onko annetulla lauseella $\phi \in \mathcal{L}$ malli.

Ongelma SAT on siis toteutuvia lauseiden ϕ joukko (lauseet merkkijonoesityksinä).

Väite. SAT kuuluu luokkaan NP.

Todistuksen idea: voidaan konstruoida epädeterministinen Turing-kone M, joka valitsee epädeterministisesti valuaation $\mathcal{V}$ ja tarkistaa, että $\mathcal{V}(\phi) = T$.

Tarvittava laskenta pystytään suorittamaan polynomisessa ajassa lauseen ϕ pituuden suhteen.

Määritelmä. Ongelma O on **C**-kova, jos kaikki luokan **C** ongelmat voidaan redusoida O:ksi polynomisessa ajassa.

Ongelman O_1 *redusoituvuus* ongelmaksi O_2 edellyttää, että löytyy i:n pituuden suhteen polynomisessa ajassa laskettava funktio $f : O_1 \to O_2$ siten, että $i \in O_1$ jos ja vain jos $f(i) \in O_2$.

Määritelmä. Ongelma O on **C**-täydellinen, jos O kuuluu luokkaan **C** ja O on **C**-kova.

$\Rightarrow$ **C**-täydelliset ongelmat ovat vaativimpia luokan **C** ongelmia.

Väite. SAT on NP-kova (todistus sivutetaan).

$\Rightarrow$ Pahimmassa tapauksessa ko. ongelman ratkaisu vaatii eksponentiaalisen ajan suhteessa lausejoukon pituuteen.

Tehokas algoritmi: Davis-Putnam [1960]

Logiikan sovelluskohteita tietojenkäsittelyssä

Voidaan nähdä karkea kahtiajako:

- **Päättelykomponentti järjestelmän osana**
  Toteutetaan loogista päättelyä jollain tasolla
  Logiikkaohjelmointi, rajoiteohjelmointi, sääntöpohjainen päättely
- **Järjestelmän ominaisuuksien analysointi**
  Ohjelmien oikeellisuustarkastelut
  Tarkistetaan, että järjestelmä toteuttaa annetun spesifikaation

Esimerkki: Sääntöpohjainen päättely

Monissa järjestelmissä (esim. logiikkaohjelmointi, deduktiiviset tietokannat, asiantuntijajärjestelmät) voidaan käyttää muotoa $A \leftarrow A_1, \ldots, A_n$ olevia sääntöjä.

- Ideana on, että säännön **seuraus** A voidaan päätellä, kunhan ensin säännön **ehdot** $A_1, \ldots, A_n$ on saatu pääteltyä.
- Tämä sääntö voidaan esittää lauselogiikan implikaationa $A_1 \wedge \ldots \wedge A_n \rightarrow A$, joka on normaalimuodossaan $\neg A_1 \vee \ldots \vee \neg A_n \vee A$ Horn-klausuuli.

Rajoitutaan jatkossa sääntöihin, joissa esintyy ainoastaan atomilauseita (ts. lauselogiikan tapaukseen).

Väite. Sääntöjoukon S säännöillä voidaan päätellä atominen lause A, jos ja vain jos A on looginen seuraus S:n sääntöjä $A \leftarrow A_1, \ldots, A_n$ vastaavista implikaatioista $A_1 \wedge \ldots \wedge A_n \rightarrow A$.

Merkitään jatkossa säännöillä S pääteltävien atomisten lauseiden joukkoa merkinnällä $\mathrm{Cn}(S)$.

Ratkaistava ongelma: voidaanko säänntöjoukon S säännöillä päätellä annettu atominen lause A? Entä jos sääntöjoukkoon S lisätään aluksi joukko atomisia lauseita F (lähtötiedot)?

Tehokas algoritmi näihin ongelmiin: Dowling-Gallier [1984]

Tietorakenteet

Otetaan käyttöön seuraavat taulukot.

- queue: jono atomeista A, jotka ovat seurauksena jossain säännössä ilman ehtoja (eli $n = 0$).
- val[A]: kuuluuko atomi A joukkoon $\mathrm{Cn}(S)$?
  Aluksi true, jos A on jonossa queue, ja false muutoin.
- rules[A]: lista säännöistä, joissa atomi A esiintyy ehtona.
- counter[s]: säännön s niiden ehtojen A määrä, joille val[A] = false.
- head[s]: säännön s seuraus

Näiden laskennan aikavaativuus on luokkaa $\mathcal{O}(n \times \log n)$.

Lineaarinen algoritmi

```
function cn:  while not empty(queue) do
                 A := get(queue);  rlist := rules[A] ;
                 while not empty(rlist) do
                    rule := first(rlist); rlist := rest(rlist);
                    counter[rule] := counter[rule] - 1;
                    if counter[rule] = 0 then
                       A1:= head[rule];
                       if val[A1] = false then
                          val[A1] := true;
                          put(A1, queue)
                       endif
                    endif
                 endwhile
              endwhile
```

Esimerkki. $S =$
$\{1 : P_0$
$2 : P_1 \leftarrow P_0$
$3 : P_2 \leftarrow P_1, P_0$
$4 : P_3 \leftarrow P_1$
$5 : P_4 \leftarrow P_1, P_4\}$

Lopputila :

val[0]	true
val[1]	true
val[2]	true
val[3]	true
val[4]	false

val[0] := true
queue := $\langle 0 \rangle$
A := 0
rules := [2,3]
  rule := 2
  counter[2] := 0
  val[1] := true
  queue := $\langle 1 \rangle$
  rule := 3
  counter[3] := 1
A := 1
rules := [3,4,5]
  rule := 3
  counter[3] := 0
  val[2] := true
  queue := $\langle 2 \rangle$
  rule := 4
  counter[4] := 0
  val[3] := true
  queue := $\langle 2, 3 \rangle$
  rule := 5
  counter[5] := 1
A := 2
rules := []
A := 3
rules := []

Esimerkki: Induktioperiaate

Määritelmä. (Induktioperiaate luonnollisille luvuille)
Jos $P(0)$ ja $P(n) \rightarrow P(n+1)$ kaikille luonnollisille luvuille n,
niin $P(n)$ kaikille luonnollisille luvuille n.

Monelle induktioperiaate on tuttu matematiikasta:

Esimerkki: Todistetaan, että $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$.

Perustapaus: $2^0 = 1$ ja $2^1 - 1 = 2 - 1 = 1$.

Induktiohypoteesi: $2^0 + 2^1 + \ldots + 2^{n-1} = 2^n - 1$

Induktioaskel:
$$\begin{aligned} 2^0 + 2^1 + \ldots + 2^n &= (2^0 + 2^1 + \ldots + 2^{n-1}) + 2^n \\ &= (2^n - 1) + 2^n \\ &= 2 \times 2^n - 1 = 2^{n+1} - 1 \end{aligned}$$

Tietojenkäsittelyssä induktiota voidaan käyttää mm. ohjelmien oikeellisuuden toteamisessa. Tarkastellaan seuraavaa esimerkkiä:

```
int sum(int a[], int n)  /* taulukko ja taulukon koko */
{
  int s=0;               /* taulukon lukujen summa */
  int i=0;               /* indeksi */
  while(i<n) { s = s+a[i]; i = i+1; }   /* summaus */
  return s;
}
```

Todistetaan induktiolla silmukan suorituskertojen (k) suhteen, että:

$$s_k = \sum_{j=0}^{j<i_k} a[j] \wedge i_k \leq n \text{ missä } s_k \text{ ja } i_k \text{ ovat s:n ja i:n arvot kerralla } k$$

Perustapaus ($k = 0$ eli silmukkaa ei ole suoritettu kertaakaan):

$$s_0 = 0 = \sum_{j=0}^{j<i_0} a[j] = \sum_{j=0}^{j<0} a[j] \text{ ja } i_0 = 0 \leq n.$$

Induktiohypoteesi (tilanne k:n suorituskerran jälkeen):

$$s_k = \sum_{j=0}^{j<i_k} a[j] \wedge \ i_k \leq n$$

Induktioaskel (huomaa, että $i_k < n$ silmukan ehdon perusteella):

$$s_{k+1} = s_k + a[i_k] = \sum_{j=0}^{j<i_k} a[j] + a[i_k] = \sum_{j=0}^{j<i_{k+1}} a[j] \text{ ja } i_{k+1} = i_k + 1 \leq n$$

Silmukan suoritus päättyy, kun $i_k < n$ tulee epätodeksi.
Koska tällöin $i_k \leq n$ on edelleen tosi, tiedämme, että $i_k = n$.

Tällöin muuttujan `s` arvona on

$$s_k = \sum_{j=0}^{j<n} a[j] \text{ mikä onkin taulukon } \texttt{a} \text{ lukujen summa.}$$

Toinen induktion käyttökohde:

Induktiota voidaan käyttää myös induktiivisesti määriteltyjä tietorakenteita käsittelevien ohjelmien analysoinnissa.

Esimerkkinä olkoon funktio `reverse`, joka kääntää listan jäsenet käänteiseen järjestykseen (`[1,2,3]` $\Rightarrow$ `[3,2,1]`).
Oikeellisuustarkastelussa osoitetaan induktiolla, että `reverse` käsittelee pituudeltaan n olevat listat oikein.

Haluatko tietää logiikasta lisää ???

Tietojenkäsittelyteorian laboratorion kurssitarjonnassa on mm.

- Tik-79.146 Logiikka tietotekniikassa: erityiskysymyksiä I (kl)
- Tik-79.154 Logiikka tietotekniikassa: erityiskysymyksiä II (sl)
- Tik-79.240 Laskennallisen vaativuuden erikoiskurssi (sl)

Näillä käsitellään lisää logiikkaan ja laskennalliseen vaativuuteen liittyviä aiheita.