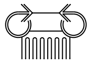
 Tietojenkäsittelytiede **22** Joulukuu 2004 sivut 47–63
Toimittaja: Antti Valmari

© kirjoittaja(t)

Lauselogiikan toteutuvuustarkastus: käytännönläheistä teoriaa

Matti Järvisalo
Teknillinen korkeakoulu
Tietojenkäsittelyteorian laboratorio
matti.jarvisalo@tkk.fi

Tiivistelmä

Tämä artikkeli käsittelee loogisten lauseiden toteutuvuutta laskennallisena ongelmana. Tavoitteena on esittää tutkimusalueena teorian ja käytännön välimaastoon sijoittuvan toteutuvuustarkastusmenetelmien kehittämisen ja tehokkuusvertailun perusideoita. Lisäksi pyritään kartoittamaan toteutuvuustarkastusmenetelmien nykyisiä käyttökohteita sekä tutkimusalueen haasteita.

1 Johdanto

Laskennan vaativuusteoriassa tavoitteena on selvittää, mitkä ongelmat ovat ratkaistavissa algoritmisesti ja kuinka tehokkaasti. Ehkä olennaisin tutkimusalueen kysymyksistä on, voidaanko tietyille, ns. ***NP**-täydellisille*, ongelmille [28] löytää tehokas eli *polynomiaikainen* algoritmi.

Kauppamatkustajan ongelma, jossa tavoitteena on löytää annettua rajaa lyhyempi n:n kaupungin kautta kulkeva kierros, lienee yleisesti tunnetuin **NP**-täydellinen ongelma. Tietojenkäsittelytieteen näkökulmasta *lauselogiikan toteutuvuusongelma* (propositional satisfiability problem, SAT) [28, sivu 77] on kuitenkin olennaisempi monista syistä. Yksinkertaisesta esitysmuodosta johtuen lauselogiikan toteutuvuusongelma on tärkeässä asemassa usein haluttaessa todistaa jokin ongelma laskennan näkökulmasta haastavaksi. Stephen Cookin 1971 SAT:lle esittämä **NP**-täydellisyystodistus [12] oli ensimmäinen laatuaan, luoden toteutuvuusongelmasta yhden laskennan vaativuusteorian kulmakivistä. Nyt, yli neljännesvuosisata myöhemmin, "$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$", eli "*onko **NP**-täydellisille ongelmille (kuten SAT) olemassa yleistä polynomiaikaista ratkaisualgoritmia*", on yhä yksi tietojenkäsittelytieteen alan tärkeimmistä avoimista ongelmista, lukuisista ratkaisuyrityksistä huolimatta. Ongelman olennaisuutta on omiaan luonnehtimaan se, että "$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$" on Clay Mathematics Instituten vuonna 2000 nimeämän seitsemän avoimen klassisen matemaattisen ongelman joukossa [2]. Ratkaisun esittäjälle on luvassa miljoonan dollarin palkinto. Yleisesti uskotaan, että polynomiaikaisen algoritmin löytäminen **NP**-täydellisille ongelmille on hyvin epätodennäköistä.

Lauselogiikalla, joka on tietojenkäsittelytieteen alan opintojen loogisten perusteiden olennainen osa, voidaan ilmaista yksinkertaisia ehtoja. Tällaisesta esimerk-

OHJE KIRJAPAINOLLE: B5-arkin vasen- ja yläreuna kohdistetaan A4-arkin vasempaan ja yläreunaan. Näin pitäisi marginaaliksi tulla taitteen puolella noin 33 mm ja muualla noin 22 mm.

kinä olkoon "täsmälleen toinen, joko Anssi tai Bettiina, voi osallistua juhliin". Ehto voidaan kirjoittaa toisin "Anssi osallistuu juhliin TAI Bettiina osallistuu juhliin, JA Anssi EI osallistu juhliin TAI Bettiina EI osallistu juhliin". Ottamalla käyttöön loogiset muuttujat a = "Anssi osallistuu juhliin" ja b = "Bettiina osallistuu juhliin", voidaan yllä oleva ehto kirjoittaa muotoon

$$(a \text{ TAI } b) \text{ JA } ((\text{ EI } a) \text{ TAI } (\text{ EI } b)),$$

eli lauselogiikan merkintöjä käyttäen

$$(a \vee b) \wedge (\neg a \vee \neg b). \tag{1}$$

Yksi ratkaisu ongelmaan on, että Anssi osallistuu juhliin ja Bettiina ei. Tätä vastaava lauselooginen ratkaisu eli *lauseen toteuttava totuusjakelu* on antaa arvo *tosi* muuttujalle a ja arvo *epätosi* muuttujalla b.

Esimerkkiin peilaten lauselogiikan toteutuvuusongelman — "*Onko annetulle lauselogiikan lauseelle olemassa sen toteuttavaa totuusjakelua?*" — **NP**-täydellisyys yleisessä tapauksessa on hämmästyttävää. Käytännössä kiinnostavat loogiset lauseet ovat kuitenkin usein huomattavan monimutkaisia, ja niille on näin ollen laskennallisesti haastavaa löytää toteuttava totuusjakelu tai todeta, että toteuttavaa jakelua ei ole olemassa.

Ongelman oletettavasta vaikeudesta huolimatta lauselogiikan toteutuvuuden tarkastusmenetelmät ja niiden toteutukset, ns. *toteutuvuustarkastimet*, ovat kehittyneet huomattavasti erityisesti viimeisimpien 10–15 vuoden aikana. Sekä täydellisiä systemaattisia että satunnaistettuja paikalliseen hakuun perustuvia toteutuvuustarkastimia on käytetty onnistuneesti monien erilaisten ongelmien tehokkaaseen ratkaisemiseen; katso esimerkiksi [17, 39] katsauksina erilaisiin toteutuvuustarkastustekniikoihin. Muun muassa suunnittelu-, testaus- sekä mallintarkastus- ja muita verifiointiongelmia [24, 25, 10, 9] voidaan ratkaista toteutuvuusongelmatapauksina, mistä johtuen tehokkaille toteutuvuustarkastimille on syntynyt suuri kysyntä. Esimerkiksi prosessorivalmistaja Intel on panostanut erityisesti verifiointiyksikköönsä Pentium-prosessorin laskuyksiköstä vuonna 1994 löytyneen, huomattavan kalliiksi koituneen vian löytämisen jälkeen.

Näin alunperin yksinomaan teoreettisesta näkökulmasta mielenkiintoisesta ongelmasta on kehittynyt ratkaisumenetelmien kehittymisen myötä kiinnostava myös käytännön näkökulmasta. Toteutuvuustarkastukseen keskittyvä *International Conference on Theory and Applications of Satisfiability Testing* järjestetään vuonna 2005 jo kahdeksatta kertaa [1]. Osana konferenssia on vuosittain kilpailu tehokkaimmasta toteutuvuustarkastimesta. Tyypillistä on, että edeltävän vuoden voittaja sijoittuu sellaisenaan seuraavan vuoden kilpailijoihin verrattuna vasta keskikastiin, mikä kuvaa hyvin menetelmien kiivasta kehitystä.

Teorian ja käytännön välimaastoon asettuva, toisaalta täsmällisiin matemaattisiin todistuksiin, toisaalta teollisuudesta kumpuavien ongelmatapausten ratkaisuajan vertailuun perustuva toteutuvuustarkastusmenetelmien tehokkuuden arviointi on saanut yhä enemmän huomiota osakseen viime vuosina. Yhtenä syynä tähän lienee, että teoreettisten tulosten ja käytännön välinen kuilu on melko kapea; usein teoreettisia tuloksia voidaan soveltaa helposti uusia tarkastusmenetelmiä kehitettäessä. Tämä pätee myös yleisemmin ns. *laskennallisen logiikan* [32] tutkimusalueella, jonka yhtä osaa toteutuvuustarkastustutkimus edustaa.[1]

[1] Ainoa laskennallisen logiikan tutkimusryhmä Suomessa toimii osana Tietojenkäsittelyteorian labo-

On hämmästyttävää, että nykyiset käytännössä tehokkaimmat täydelliset toteutuvuustarkastimet tyypillisesti perustuvat jo 1960-luvulla esiteltyyn *Davis-Putnam-Logemann-Loveland-menetelmään* (DPLL) [16], joka on pohjimmiltaan varsin yksinkertainen hakumenetelmä. Yksi hypoteesi on, että DPLL-pohjaiset tarkastusmenetelmät alkavat olla varsin huippuunsa viritettyjä. Eräitä tämän päivän tutkimussuuntauksia ovatkin

(i) sallivampiin, ongelmaspesifisempiin kuvausmuotoihin perustuvien toteutuvuustarkastusmenetelmien kehittäminen,

(ii) toteutuvuustarkastusmenetelmien laajentaminen tehokkaiksi rajoitetuiksi lineaarirajoiteratkaisimiksi ja

(iii) tehokkaimpien toteutuvuustarkastusmenetelmien valjastaminen uusien, esimerkiksi bioinformatiikan alalta kumpuavien, ongelmien ratkaisemiseen.

Huomattavaa on, että nämä kolme kehityssuuntaa ovat läheisessä yhteydessä toisiinsa.

Tässä artikkelissa pyritään esittelemään lauselogiikan toteutuvuusongelmaa (kappale 2), sen ratkaisemisessa yleensä käytettäviä, käytännössä tehokkaita algoritmisia ideoita (kappale 4) ja ratkaisumenetelmien sovellutuskohteita (kappale 3) ja arviointimenetelmiä (kappale 5), sekä kartoittamaan hieman, mitä haasteita toteutuvuustarkastus nyt ja tulevaisuudessa tarjoaa (kappale 6). Erityisesti keskitytään täydellisiin tarkastusmenetelmiin ja käytännöstä kumpuaviin ongelmatapauksiin.

2 Toteutuvuusongelman määrittely

Lauselogiikan lauseet koostuvat Boolen muuttujista (arvoalueena $\{tosi, epätosi\}$) ja niitä yhdistävistä *konnektiiveista*

- $\neg$ (looginen ei, *negaatio*),
- $\vee$ (looginen tai, *disjunktio*) ja
- $\wedge$ (looginen ja, *konjunktio*).

Lisäksi tyypillisesti määritellään *implikaatio*

$$a \rightarrow b \equiv \neg a \vee b$$

ja *ekvivalenssi*

$$a \leftrightarrow b \equiv (\neg a \vee b) \wedge (a \vee \neg b).$$

Esimerkkinä lauselogiikan lauseesta toimii johdannon ehto toisensa poissulkevasta Anssin ja Bettiinan juhliin osallistumisesta. Esimerkki voidaan ekvivalenssin avulla kirjoittaa muodossa $\neg a \leftrightarrow b$.

Literaali on looginen muuttuja x tai sen negaatio, $\neg x$. Muotoa $l_1 \vee \cdots \vee l_n$ olevaa lausetta sanotaan *klausuuliksi*, missä jokainen l_i on literaali. Jokainen lauselogiikan lause voidaan ilmaista ekvivalenttina ns. *konjunktiivisessa normaalimuodossa* (KNM) olevana lauseena. Lause on KNM:ssa, jos se on muotoa $C_1 \wedge \cdots \wedge C_n$, missä jokainen C_i on klausuuli. Esimerkiksi (1) on KNM:ssa.

Kuten yleisesti on tapana, tässä artikkelissa käytetään tarvittaessa lyhennysmerkintöjä

$$\bigvee_{i=1}^{k} P_i = P_1 \vee \cdots \vee P_k$$

ja

$$\bigwedge_{i=1}^{k} P_i = P_1 \wedge \cdots \wedge P_k.$$

ratoriota Teknillisen korkeakoulun tietotekniikan osastolla, katso `http://www.tcs.hut.fi/Research/Logic/`.

Lauseen *totuusjakelu* asettaa kullekin lauseen muuttujalle totuusarvon *tosi* tai *epätosi*. Näin ollen totuusjakeluiden määrä on 2^n, kun n on lauseessa olevien muuttujien määrä. Totuusjakelu *toteuttaa* lauseen, jos lause *toteutuu* totuusjakelun määräämillä muuttujien arvoilla; esimerkiksi $\neg x$ toteutuu, kun muuttujalle x asetetaan arvo *epätosi*, ja mielivaltainen klausuuli toteutuu, jos jokin klausuulin literaali toteutuu. Tällaista totuusjakelua sanotaan lauseen *toteuttavaksi totuusjakeluksi* tai *malliksi*. Lause, jolle on olemassa malli, on *toteutuva*, ja *toteutumaton*, jos mallia ei ole olemassa. Esimerkiksi (1) on toteutuva; sen malleja ovat

$$\{a \leftarrow tosi,\ b \leftarrow epätosi\}$$

ja

$$\{a \leftarrow epätosi,\ b \leftarrow tosi\}.$$

Lauselogiikan toteutuvuusongelmassa (SAT) kysytään, onko annettu lauselogiikan lause toteutuva. *Toteutuvuusongelmatapaukseksi* kutsutaan lauselogiikan lausetta. Toteutuvuusongelma on **NP**-täydellinen, vaikka rajoituttaisiin KNM-muotoisiin lauseisiin, ja jopa vaikka jokaisessa klausuulissa olisi tasan kolme literaalia [28, Prop. 9.2].

3 Ongelmat toteutuvuusongelmina

Toteutuvuustarkastusmenetelmien kysynnän kasvun syinä voidaan mainita tarkastusmenetelmien huiman kehityksen lisäksi menetelmien laajat käyttömahdollisuudet. Edellämainitut suunnittelu-, testaus- ja verifiointiongelmat ovat vain esimerkkejä ongelmatyypeistä, joita voidaan ratkaista toteutuvuusongelmatapauksina tehokkaasti. Menetelmien tehokkuutta kuvaa se, että jopa satojatuhansia muuttujia ja klausuuleja sisältäviä mallintarkastusongelmia on kyetty ratkaisemaan tavallisilla tietokoneilla.

Yleisesti ideana on kuvata alkuperäinen ongelma lauselogiikan lauseena siten, että lause toteutuu jos ja vain jos alkuperäiselle ongelmalle on olemassa ratkaisu, ja lisäksi lauseen toteuttavasta totuusjakelusta voidaan lukea suoraan ratkaisu alkuperäiselle ongelmalle. Kuvaustapa vaikuttaa olennaisesti menetelmän tehokkuuteen. Tämä pätee myös yleisemmin, ja tästä johtuen kompaktien, käytännössä hyvin toimivien kuvausideoiden kehittäminen on vireä tutkimusalue.

Tässä kappaleessa keskitytään hieman tarkemmin tarkastelemaan, miten suunnitteluongelmia voidaan ratkaista ja ns. *rajoitettua mallintarkastusta* (*bounded model checking, BMC*) voidaan tehdä käyttäen toteutuvuustarkastusmenetelmiä.

3.1 Rajoitettu mallintarkastus

Digitaalijärjestelmiä suunniteltaessa laaditaan järjestelmästä *järjestelmäkuvaus*, joka koostuu järjestelmän toiminnallisuuden yksityiskohtaisesta määrittelystä. *Verifioinnin* tavoitteena on tarkastaa, toteuttaako annettu järjestelmäkuvaus tietyn ominaisuuden — esimerkiksi “*järjestelmä ei voi päätyä lukkiumatilaan*” eli järjestelmää ei ole mahdollista ajaa millään syötejonolla tilaan, jossa se ei kykene tekemään mitään. Ongelma voidaan periaatteessa ratkaista simuloimalla järjestelmän kaikkia mahdollisia suorituksia. Nykyisissä järjestelmissä mahdollisten suoritusten määrä on kuitenkin huikea, mistä johtuen ongelmaksi voi muodostua *tilaavaruusräjähdys* [37].

Mallintarkastus [11] on kirjoittajasta riippuen synonyymi sanalle verifiointi, tai verifioinnin alalaji, jossa verifioitava ominaisuus ilmaistaan loogisella kaavalla. *Symbolisessa mallintarkastuksessa*

tila-avaruusräjähdys pyritään välttämään luopumalla järjestelmän tilojen eksplisiittisestä esittämisestä. Perinteiset symboliset menetelmät perustuvat ns. *binäärisiin päätöskaavioihin* (*binary decision diagram*, BDD) [4]. Noin viiden viime vuoden aikana on kuitenkin tutkittu toteutuvuustarkastukseen perustuvaa symbolista mallintarkastusta.

Rajoitettu mallintarkastus [10] on toteutuvuustarkastusmenetelmiin pohjautuva symbolinen mallintarkastusmenetelmä. Rajoitetussa mallintarkastuksessa tavoitteena on tarkastaa, toteuttavatko järjestelmän kaikki enintään k:n mittaiset suoritukset tietyn ominaisuuden. Rajoitettua mallintarkastusta voidaan suorittaa toteutuvuustarkastusmenetelmiin pohjautuen. Tällöin ideana on kuvata halutun ominaisuuden negaatio sekä järjestelmän k:n askeleen tilasiirtymät lauselogiikan lauseena ja tarkastaa, onko tälle lauseelle mallia toteutuvuustarkastinta käyttäen. Tavoitteena on löytää iteratiivisesti k:n arvoa kasvattaen vastaesimerkki, joka osoittaa järjestelmän kuvauksen virheellisyyden. Kuvaus muodostetaan niin, että mahdollinen vastaesimerkki voidaan lukea suoraan löydetystä mallista. Tästä johtuen menetelmä soveltuu erityisesti järjestelmien *virheiden etsintään* (*debugging*).

Seuraavassa käydään esimerkin kautta läpi yksi mahdollinen tapa kuvata rajoitettu mallintarkastusongelma lauselogiikalla. Oletetaan, että halutaan tarkastaa, että järjestelmä ei voi millään k:n askeleen suorituksella käydä tilassa, jossa ominaisuus P pätee. Tätä voidaan tarkastella tutkimalla seuraavan lauselogiikan lauseen toteutuvuutta:

$$I(v_0) \wedge \bigwedge_{i=0}^{k-1} R(v_i, v_{i+1}) \wedge (\bigvee_{i=0}^{k} P(v_i)),$$

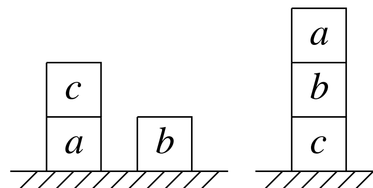
missä

- v_i on järjestelmätilamuuttujien vektori i:n tilasiirtymän jälkeen,
- lause $I(v_0)$ luonnehtii järjestelmän alkutilat,
- R järjestelmän siirtymärelaation, ja
- $P(v_i)$ kuvaa ominaisuuden P voimassaolemista v_i:n määrittelemässä tilassa.

Jos tälle lauseelle on olemassa malli, jokin $P(v_i)$ on *tosi*. Mahdollinen malli toimii siis suoraan vastaesimerkkinä, joka kertoo, mikä $P(v_i)$ on *tosi*, eli millä ajanhetkellä ominaisuus P on voimassa, sekä operaatiojonon, jolla ko. tilaan päästiin.

3.2 Suunnittelu

Monet tekoälyongelmat, kuten aikataulutus (*scheduling*), ovat suunnitteluongelmia. Seuraavassa esitellään suunnitteluongelma klassisen esimerkin kautta. Kyseessä on *palikkamaailma* (blocks world) ns. *Sussmanin anomalian* tapauksessa. Siinä tarkastellaan kolmea palikkaa a, b ja c. Alkutilassa palikat a ja b ovat pöydällä, ja c on a:n päällä. Tavoitteena on siirtää palikat päällekkäin järjestykseen, jossa c on pöydällä, b on c:n päällä ja a b:n päällä. Alkutila ja tavoitetila on esitetty kuvassa 1.



Kuva 1: Sussmanin anomalian alkutila (vasemmalla) ja tavoitetila (oikealla).

Vain palikkaa, jonka päällä ei ole toista palikkaa, voidaan siirtää. Palikka voi olla vain pöydällä tai toisen palikan päällä.

Mallin yksinkertaistamiseksi pöytä ajatellaan palikaksi d, jota ei voi siirtää, mutta jonka päälle ja päältä voi siirtää.

Suunnitteluongelmassa tila voidaan nähdä totuusarvojakeluna joukolle tilamuuttujia. Palikkamaailmaesimerkissä valitaan tilamuuttujiksi kahdentyyppisiä ehtoja. Kullekin palikalle P otetaan ehto, joka kertoo, onko palikan P päällinen tyhjä. Lisäksi kullekin parille palikoita P_1, P_2 otetaan ehto, joka kertoo, onko palikka P_1 palikan P_2 päällä. Käytämme seuraavassa näistä tilamuuttujista merkintöjä $clear(P)$ ja $on(P_1, P_2)$, missä P, P_1 ja P_2 ovat joukosta $\{a, b, c, d\}$. Siis esimerkiksi tilamuuttuja $on(a, c)$ saa arvon tosi tilassa, jossa palikka a on palikan c päällä.

Palikkamaailmassa mahdolliset toiminnot ovat muotoa $\texttt{MOVE}(P_1, P_2, P_3)$ (siirrä palikka P_1 palikan P_2 päältä palikan P_3 päälle), missä P_1, P_2 ja P_3 ovat joukosta $\{a, b, c, d\}$.

Toiminnoille on olemassa rajoittavia ehtoja, jotka kuvataan *esiehtojen* ja *jälkiehtojen* avulla; nykyinen tila voidaan muuntaa jälkiehtojen määrittelemäksi tilaksi vain jos esiehdot ovat voimassa nykyisessä tilassa. Jos esi- tai jälkiehdot eivät koske tiettyä tilamuuttujaa, pysyy tämän totuusarvo ennallaan. Toimintoon MOVE liittyvät esi- ja jälkiehdot ovat seuraavat.

- Esiehdot:
  $\texttt{clear}(P_1)$, $\texttt{clear}(P_3)$, $\texttt{on}(P_1, P_2)$.

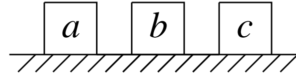
- Jälkiehdot:
  $\texttt{clear}(P_1)$, $\texttt{clear}(P_2)$, $\texttt{on}(P_1, P_3)$, $\neg\texttt{clear}(P_3)$, $\neg\texttt{on}(P_1, P_2)$.

Suunnitteluongelmassa on siis annettu alkutila, mahdolliset toiminnot ja ehdot hyväksyttävälle lopputilalle. Ongelman ratkaisu eli *suunnitelma* on jono toimintoja, jotka peräkkäin suoritettuna muuntavat alkutilan sellaiseksi tilaksi, jossa lopputilaehdot ovat voimassa.

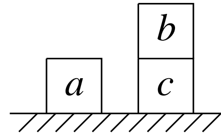
Palikkaesimerkissä (lyhyimmäksi) suunnitelmaksi saadaan seuraava jono toimintoja:

$$(\texttt{MOVE}(c, a, d), \texttt{MOVE}(b, d, c), \texttt{MOVE}(a, d, b)).$$

Toiminnon $\texttt{MOVE}(c, a, d)$ jälkeinen tila on esitetty kuvassa 2 ja toiminnon $\texttt{MOVE}(b, d, c)$ jälkeinen tila kuvassa 3.



Kuva 2: Tila toiminnon $\texttt{MOVE}(c, a, d)$ suorittamisen jälkeen.



Kuva 3: Tila toiminnon $\texttt{MOVE}(b, d, c)$ suorittamisen jälkeen.

Edellä annettu suunniteluongelman määrittely lähtee siitä, että ratkaisuna saatavassa suunnitelmassa kussakin tilassa yksi toiminto aiheuttaa tilan muuttumisen toiseksi. Seuraavassa esitettävää suunnitteluongelman lauselogiikan toteutuvuusongelmakuvausta varten otetaan käyttöön $k+1$ ajanhetkeä $t = 0, 1, \ldots, k$ siten, että ajanhetki 0 tarkoittaa järjestelmän alkutilaa ja yleisesti ajanhetkellä t tarkoitetaan järjestelmän tilaa t:n toiminnon jälkeen. Kuvauksessa vaaditaan siis, että jokaisena ajanhetkenä t suoritetaan täsmälleen yksi toiminto. Jokaista tilamuuttujaa kohden otetaan muuttuja jokaiselle $t \in \{0, \ldots, k\}$ ja jokaista toimintoa kohden muuttuja jokaiselle $t \in \{0, \ldots, k-1\}$; palikkamaailmatapauksessa siis $\texttt{on}(P_1, P_2, t)$ ja $\texttt{clear}(P, t)$, missä $t = 0, \ldots, k$, ja $\texttt{MOVE}(P_1, P_2, P_3, t)$, missä $t = 0, \ldots, k-1$.

Palikkamaailmatapauksessa yksi mahdollinen lauselooginen kuvaus on

muotoa

$$A \wedge \bigwedge_{t=0}^{k-1} (Y_t \wedge V_t \wedge K_t) \wedge L,$$

missä

- A kuvaa alkutilan,
- L tavoitetilan,
- Y_t vaatimuksen, että ajanhetkellä t suoritetaan täsmälleen yksi toiminto,
- V_t ajanhetkellä t suoritettavan toiminnon esi- ja jälkiehdot, ja lisäksi
- K_t ns. *kehysaksioomat*, jotka huolehtivat siitä, että tila säilyy ajanhetkestä t ajanhetkeen $t+1$ ennallaan niiden tilamuuttujien osalta, joihin ei kohdistu toimintoa hetkellä t.

Seuraavassa pyritään antamaan idea siitä, miten yllämainitut kuvauksen osat muodostetaan.

Alkutilan ($t = 0$) kuvaus A on muotoa

$$\texttt{on}(c,a,0) \wedge \texttt{on}(a,d,0) \wedge \texttt{on}(b,d,0) \wedge \texttt{clear}(c,0) \wedge \texttt{clear}(b,0).$$

Tavoitetilan ($t = k$) kuvaus L on muotoa

$$\texttt{on}(a,b,k) \wedge \texttt{on}(b,c,k) \wedge \texttt{on}(c,d,k).$$

Vaatimuksen "jokaisena ajanhetkenä suoritetaan täsmälleen yksi toiminto" kuvaus Y_t on muotoa $Y_t^1 \wedge Y_t^2$, missä Y_t^1 sallii, että enintään yksi toiminto suoritetaan ja Y_t^2 että vähintään yksi toiminto suoritetaan ajanhetkellä t. Esimerkiksi Y_t^1:n osana on palikan a siirtämisen ajanhetkellä t palikan b päältä *sekä* palikan c päälle *että* pöydälle (d) kieltävä lause

$$\neg\texttt{MOVE}(a,b,c,t) \vee \neg\texttt{MOVE}(a,b,d,t).$$

Jatkoa varten tarvitaan läpikäytävien muuttujien x, y ja z arvoyhdistelmiä rajoittava kaava:

$$x,y,z \in \{a,b,c,d\} \ \wedge \ x \neq y \ \wedge \ x \neq z \ \wedge \ y \neq z \ \wedge \ x \neq d \qquad (2)$$

Lause Y_t^2 on muotoa

$$\bigvee_{(2)} \texttt{MOVE}(x,y,z,t).$$

Lisäksi pitää kuvata toiminnon MOVE vaikutukset tilaan jokaisella ajanhetkellä t, eli esi- ja jälkiehdot kyseiselle toiminnolle:

$$V_t = \bigwedge_{(2)} (\texttt{MOVE}(x,y,z,t) \ \rightarrow (\texttt{clear}(x,t) \wedge \texttt{on}(x,y,t) \wedge \texttt{clear}(z,t) \ \wedge \texttt{on}(x,z,t+1) \wedge \texttt{clear}(y,t+1) \ \wedge \texttt{clear}(x,t+1) \wedge \ \neg\texttt{clear}(z,t+1) \wedge \neg\texttt{on}(x,y,t+1))).$$

Ylläolevalla lauseella vaaditaan, että joko toimintoa $\texttt{MOVE}(x,y,z,t)$ ei suoriteta tai esi- ja jälkiehtojen on oltava voimassa.

Lisäksi jokaista toiminto/tilamuuttujaparia kohden on lisättävä jokaiselle ajanhetkelle edellämainittu kehysaksiooma. Määritellään esityksen avuksi rajoitus

$$x',y',z' \in \{a,b,c,d\} \ \wedge \ (x' \neq x \ \vee \ (y' \neq y \ \wedge \ y' \neq z)). \qquad (3)$$

Nyt esimerkiksi toiminnolle MOVE ja on-tyyppiselle tilamuuttujalle kirjoitetaan kehysaksiooma $K_t^{\texttt{MOVE},\texttt{on}}$, joka on muotoa

$$\bigwedge_{(2)} (\texttt{MOVE}(x,y,z,t) \rightarrow \bigwedge_{(3)} (\texttt{on}(x',y',t) \leftrightarrow \texttt{on}(x',y',t+1))).$$

Merkittäessä kaikkien ajanhetkeä t koskevien kehysaksioomien voimassaoloa vaativaa lausetta K_t:llä — tässä tapauksessa

siis

$$K_t^{\texttt{MOVE,on}} \wedge K_t^{\texttt{MOVE,clear}}$$

— on kaikki kuvauksen osat saatu määriteltyä.

4 Toteutuvuustarkastusmenetelmistä

Toteutuvuusongelman **NP**-täydellisyydestä huolimatta suuri joukko erityyppisiä lauselogiikan toteutuvuustarkastukseen tarkoitettuja menetelmiä on esitetty erityisesti 1980-luvulta tähän päivään (katso [17]). Lähestymistapoja perustuen esimerkiksi *neuroverkkoihin* tai (globaaliin/lokaaliin, lineaariseen/epälineaariseen) *optimointiin* on esitetty. Erityisesti viimeksi kuluneiden noin 10–15 vuoden aikana toteutuvuustarkastusmenetelmien kehitys on ollut huomattavaa.

Menetelmät voidaan luokitella niiden *systemaattisuuden* mukaan. Systemaattinen menetelmä on tyypillisesti täydellinen; jos lause on toteutuva, menetelmä löytää sille mallin, ja erityisesti jos lause on toteutumaton, menetelmä pystyy toteamaan tämän. Ei-systemaattiset menetelmät sisältävät *satunnaisen komponentin* ja perustuvat tyypillisesti *paikalliseen hakuun* (katso esimerkiksi [34, 35, 20]). Ei-systemaattisia menetelmiä on käytetty erityisesti *satunnaisten* toteutuvuusongelmatapausten ratkaisemiseen. Tässä artikkelissa keskitymme systemaattisiin menetelmiin.

On hämmästyttävää, että nykyiset käytännössä tehokkaimmat systemaattiset toteutuvuustarkastimet tyypillisesti perustuvat jo 1960-luvulla esiteltyyn *Davis-Putnam-Logemann-Loveland-menetelmään* (DPLL) [16], joka on pohjimmiltaan varsin yksinkertainen syvyyssuuntainen hakumenetelmä. Seuraavassa käsitellään DPLL-menetelmän perusideaa ja sen tehokkuutta huomattavasti parantavaa, nykyisissä tarkastimissa käytettävää *oppimista*.

4.1 DPLL

DPLL-menetelmä olettaa syötteenä annettavan toteutuvuustarkastustapauksen olevan KNM:ssa. Menetelmän esittäminen selkeytyy, kun KNM:ssa oleva lause nähdään joukkona klausuuleja ja klausuulit edelleen joukkoina literaaleja. Näin esimerkiksi lause $(a \vee b) \wedge (\neg a \vee \neg b)$ voidaan nähdä joukkona $\{\{a,b\},\{\neg a,\neg b\}\}$.

Algoritmi $\mathcal{A}$ kuvaa DPLL-menetelmän perusperiaatteen. Algoritmi valitsee klausuulijoukon muuttujista yhden (x), ns. *haarautumismuuttujan*, käytetyn heuristiikan mukaan (HeuristiikkaA), ja edelleen tälle muuttujalle arvon v (*tosi* tai *epätosi*) toisen heuristiikan (HeuristiikkaB) mukaan. Merkintä $\mathcal{C}[x \leftarrow v]$ tarkoittaa klausuulijoukkoa, joka on saatu klausuulijoukosta $\mathcal{C}$ muuttamalla sitä seuraavilla kahdella tavalla.

(i) On poistettu jokainen sellainen klausuuli, jossa on muuttujan x ilmentymä (literaali), joka toteutuu totuusarvojakelulla $\{x \leftarrow v\}$.

(ii) On poistettu jokainen muuttujan x ilmentymä, joka ei toteudu totuusarvojakelulla $\{x \leftarrow v\}$.

Jos $\mathcal{A}(\mathcal{C}[x \leftarrow v])$ palauttaa arvon "*toteutumaton*", haarasta, jossa x saa arvon v, ei löydetty mallia. Tällöin kokeillaan sijoittaa muuttujaan x arvon v vasta-arvo $\neg v$. Jos kummastakaan haarasta ei löydetä mallia, on klausuulijoukon $\mathcal{C}$ oltava toteutumaton.

Algoritmi $\mathcal{A}$
Syöte: Klausuulijoukko $\mathcal{C}$
Palauttaa: “toteutuva” tai “toteutumaton”

1. $\mathcal{C} \leftarrow$ Yksinkertaista($\mathcal{C}$)
2. **If** $\mathcal{C} = \emptyset$ **return** *toteutuva*
3. **If** $\emptyset \in \mathcal{C}$ **return** *toteutumaton*
4. muuttuja $x \leftarrow$ HeuristiikkaA($\mathcal{C}$)
5. arvo $v \leftarrow$ HeuristiikkaB($\mathcal{C}, x$)
6. **If** $\mathcal{A}(\mathcal{C}[x \leftarrow v]) =$ *toteutuva* **return** *toteutuva*
7. **If** $\mathcal{A}(\mathcal{C}[x \leftarrow \neg v]) =$ *toteutuva* **return** *toteutuva*
8. **return** *toteutumaton*

Jos kaikki klausuulit poistetaan klausuulijoukosta, kohdan (i) perusteella jokainen klausuuli toteutuu, ja näin ollen klausuulijoukko on toteutuva (rivi 2). Toisaalta, jos jonkin klausuulijoukon klausuulin kaikki literaalit on poistettu haun aikana, kohdan (ii) perusteella kyseinen klausuuli ei toteudu tehdyillä totuusarvovalinnoilla (rivi 3).

Lausejoukkoa voidaan *yksinkertaistaa* aina ennen uuden haarautumismuuttujan valintaa. Esimerkkinä käytettävistä yksinkertaistamissäännöistä voidaan mainita ns. *monotonisen literaalin sääntö*: jos jonkin muuttujan x kaikki jäljellä olevat ilmentymät ovat samaa polariteettia (eli x literaalit ovat kaikki muotoa x tai kaikki muotoa $\neg x$), korvataan $\mathcal{C}$ joukolla $\mathcal{C}[x \leftarrow v]$, missä v on se totuusarvo, jolla x:n literaalit toteutuvat.

Olennaisin DPLL-algoritmien yksinkertaistamismenetelmä on kuitenkin ns. *yksikkövyörytys* (unit propagation). Oletetaan, että klausuulijoukko $\mathcal{C}$ sisältää klausuulin, jossa on ainoastaan yksi literaali l (ns. *yksikköklausuuli*). Olkoon l muuttujan x ilmentymä. *Yksikkövyörytyssäännön* mukaan lausejoukko $\mathcal{C}$ voidaan tällöin korvata joukolla $\mathcal{C}[x \leftarrow v]$, missä v on se totuusarvo, jolla l toteutuu. Yksikkövyörytyksessä yksikkövyörytyssääntöä sovelletaan, kunnes sen hetkisessä klausuulijoukossa ei enää ole yksikköklausuuleja.

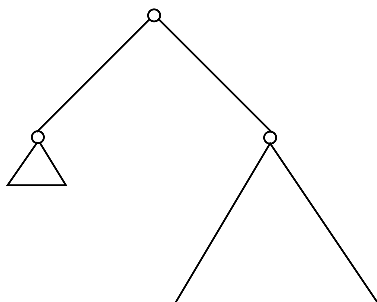
Yksikkövyörytys ei teoriassa tehosta DPLL-algoritmia; sama vaikutus saadaan aikaan pakottamalla heuristiikat valitsemaan mahdollisten yksikköklausuulien sanelemia muuttuja/totuusarvo-pareja. Yksikkövyörytys voidaan kuitenkin käytännössä toteuttaa erittäin tehokkaasti. Lisäksi monotonisen literaalin sääntöä ei pystytä suoraan simuloimaan yksikkövyörytyksen tapaan heuristisilla valinnoilla; vaikkakin monotoninen literaali ja sille sopiva arvo voidaan tunnistaa sopivilla heuristiikoilla, ilman monotonisen literaalin säännön käyttöä voidaan joutua tilanteeseen, jossa algoritmin $\mathcal{A}$ rivin 7 rekursiivinen kutsu suoritetaan turhaan.

4.2 Heuristiikoista ja oppimisesta

Haarautumismuuttujien ja niiden arvojen valintaan vaikuttavilla heuristiikoilla on käytännössä erittäin suuri vaikutus DPLL-menetelmään pohjautuvan toteutuvuustarkastimen tehokkuuteen. Monenlaisia heuristiikkoja on esitetty, katso esimerkiksi [27, 26, 6, 21, 19].

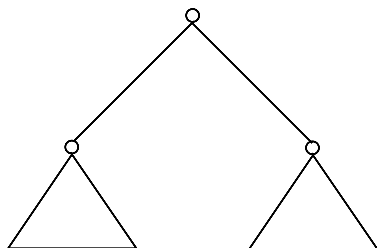
Intuitiivisesti ajatellen ahne heuristiikka pyrkii tekemään valintoja, jotka johtavat nopeasti mallin löytymiseen tai ristiriitaan. Kuvan 4 hakupuu on syntynyt tällaisen valinnan (vasen haara) kautta. Kuvassa vasen haara edustaa jäljellejäävää hakua, kun valittuun muuttujaan sijoitetaan arvo *epätosi*, oikea haara taas sijoitusta *tosi*. Mahdollinen ongelma on silminnähtävä. Vaikka ahne valinta on joh-

tanut tavoitteena olleeseen pieneen alihakupuuhun (vasen alipuu), ei valinta ole ottanut huomioon jäljellejäävää hakua (oikea alipuu) siinä tapauksessa, että ratkaisua ei löydy vasemmasta haarasta. Oikean alipuun koko saattaa tässä muodostua pullonkaulaksi.



Kuva 4: Epätasapainoinen hakupuu.

Kuvassa 5 esitetty hakupuu on tasapainoinen; on päädytty suurinpiirtein samankokoiseen hakuun kummassakin alipuussa. Kumman tahansa totuusarvon sijoittaminen valitulle muuttujalle johtaa siis kohtuullisen kokoiseen hakuun. Verrattaessa kuvien 4 ja 5 alihakupuita huomataan, että kuvan 5 alihakupuiden yhteispinta-ala on pienempi (eli haku lyhyempi) kuin kuvan 4. Tähän vedoten voidaan argumentoida, että hyvä heuristinen idea olisi pyrkiä valitsemaan sellaisia muuttujia, joilla haarauduttaessa hakupuusta tulee tasapainoinen ja molemmat alihakupuut ovat kohtuullisen pieniä.



Kuva 5: Tasapainoinen hakupuu.

Tehokas hakuavaruuden karsinta osana syvyyssuuntaista hakua on erittäin olennaista toteutuvuustarkastuksessa, sillä muutoin haun tehokkuus ei missään nimessä riittäisi suurten teollisuusongelmatapausten ratkaisemiseen. Nykyiset tehokkaimmat DPLL-pohjaiset toteutuvuustarkastimet käyttävät hyväksi ns. *ristiriitoihin perustuvaa oppimista* (*conflict driven learning*) [38]. Intuitiivisesti oppimisen integroiminen osaksi toteutuvuustarkastinta johtaa hakuavaruuden huomattavaan karsimiseen verrattuna perus-DPLL-algoritmiin.

Oppimisessa muodostetaan hakua ohjaavia ns. *konfliktiklausuuleja*, joilla eksplisiittisesti kielletään totuusarvoyhdistelmiä, joiden on haun aikana todettu johtavan totuusarvojakeluihin, jotka eivät toteuta klausuulijoukkoa. Yksinkertainen esimerkki tästä on, että valintojen johtaessa ristiriitaan voidaan käsiteltävään klausuulijoukkoon lisätä tehtyjen valintojen vastakohtaa esittäviä klausuuleja. Esimerkiksi jos valintojen $x \leftarrow$ *tosi*, $y \leftarrow$ *epätosi* jälkeen voidaan yksinkertaistamalla johtaa tyhjä klausuuli, pätee yleisesti, että klausuulijoukkoa ei toteuta mikään totuusarvojakelu, jossa muuttujaan x sijoitetaan arvo *tosi* ja muuttujaan y arvo *epätosi*. Tästä voidaan *oppia* klausuuli $\neg x \vee y$, joka ilmaisee, että molempia valinnoista $x \leftarrow$ *tosi*, $y \leftarrow$ *epätosi* ei voida tehdä. Opittu klausuuli voidaan nyt lisätä osaksi alkuperäistä klausuulijoukkoa.

Oppimisen haasteena on löytää *lyhyitä* (ja näin ollen hakua tehokkaasti rajaavia) konfliktiklausuuleja. Tässä mielessä yllä kuvattu yksinkertainen oppimisidea ei ole suoraan käyttökelpoinen. Käytännössä tehokkaissa menetelmissä oppiminen on yhdistetty ns. *epäkronologiseen peruuttamiseen* (nonchronological backtracking), jossa valintoja peruutetaan (backtracking) monta kerrallaan. On osoitettu, että konflikteihin perustuva oppiminen tehostaa — sekä käytännössä että teoreetti-

sesti — toteutuvuustarkastusmenetelmää huomattavasti.

Useissa tarkastimissa valintaheuristiikka ottaa huomioon opitut klausuulit. Seuraava, käytännössä toimiva, ns. VSIDS-heuristiikka (*variable state independent decaying sum*) on osana erityisesti rakenteisten ongelmien tapauksessa yhtä parhaista toteutuvuustarkastimista nimeltä `zChaff` [27]. VSIDS ylläpitää jokaiselle *literaalille* l *pistearvoa* $s(l)$ ja *laskuria* $c(l)$. Aluksi kaikille literaaleille l asetetaan $c(l) = 0$. Ennen haun aloittamista $s(l)$:n arvoksi asetetaan literaalin esiintymiskertojen määrä syötteenä saatavassa lausejoukossa. Opittaessa konfliktiklausuuli C korotetaan jokaisen C:n sisältämän literaalin arvoa $c(l)$ yhdellä. Lisäksi määräajoin tehdään jokaisen literaalin l osalta päivitys

$$s(l) := c(l) + \frac{s(l)}{2};\ c(l) := 0.$$

Muuttujaa valittaessa VSIDS-heuristiikka valitsee literaalin l, jolla arvo $s(l)$ on suurin, ja asettaa literaalia vastaavalle muuttujalle arvon, joka toteuttaa literaalin.

Intuitiivinen oletus VSIDS-heuristiikan takana on, että *uusimmat* opitut klausuulit ovat olennaisimpia. Tämä pyritään ottamaan huomioon sillä, että literaalien pistearvoa jaetaan määräajoin vakiolla, ja pistearvo kasvaa ainoastaan opittaessa konfliktiklausuuli, jossa kyseinen literaali on.

5 Tarkastusmenetelmien arvioinnista

Eri menetelmien tehokkuusvertailu on olennaista tehokkaita toteutuvuustarkastusmenetelmiä kehitettäessä. Vertailua voi tehdä joko *empiirisesti* tai *analyyttisesti*. Näistä jälkimmäinen, matemaattista pohjaa hyödyntävä keino voidaan edelleen jakaa *pahimman tapauksen analyysiin* ja *todistuskompleksisuustarkasteluihin*. Huomattavaa on, että vertailumenetelmät eivät suinkaan ole toisensa poissulkevia, vaan toisiaan tukevia, ja jokaisella niistä on omat hyvät ja huonot puolensa.

5.1 Empiirinen arviointi

Tarkastusmenetelmien empiirisen arvioinnin osalta toteutuvuustarkastusyhteisö on huomattavan organisoitunut. Tunnetusti vaikeita, yleisesti käytössä olevia ns. *benchmark-tapauksia* on koottu yhteen[2]. Tarve laajalle valikoimalle erityyppisiä (satunnaisia, teollisuudesta kumpuavia, jne.) ongelmia on selvä; yleisesti tehokas toteutuvuustarkastin on huomattavasti hankalampi toteuttaa kuin vain tietynlaisia tapauksia tehokkaasti ratkaiseva tarkastin.

Ongelma empiirisessä arvioinnissa on benchmark-tapausten kattavuudessa ja objektiivisuuden säilyttämisessä eri menetelmiä vertailtaessa. Empiiriset kokeet ovat kuitenkin erittäin tärkeä osa käytännön sovellutuksiin tähtäävää toteutuvuustarkastustutkimusta.

5.2 Pahimman tapauksen analyysi

Mielenkiinto toteutuvuustarkastusmenetelmien pahimman tapauksen analyysia kohtaan on kasvanut erityisesti viime vuosien aikana, katso esimerkiksi [15]. Pahimman tapauksen analyysi on kuitenkin useissa tapauksissa varsin haastavaa. Tällä hetkellä parhaimman tunnetun täydellisen menetelmän pahimman tapauksen laskenta-aika on luokkaa $O(1{,}3^n)$, missä n

[2] Katso `http://www.satlib.org/`.

on tapauksen muuttujien määrä [15]. Eksponentin kantaa on saatu tasaisesti pienennettyä. Tällä rintamalla tutkimustyö on aktiivista ja erityisesti teoreettisesta näkökulmasta hyvin mielenkiintoista.

Analyysin luonteenomaisen haastavuuden lisäksi ongelmana on, että tunnetusti pahimman tapauksen käyttäytymiseltään parhaimmat menetelmät ovat hankalasti toteutettavissa. Usein hyvin optimoitu DPLL-pohjainen menetelmä onkin empiirisesti tällaista menetelmää tehokkaampi käytännössä. Pahimman tapauksen käytös onkin usein varsin karkea vertailuparametri käytännössä tehokkaille menetelmille.

5.3 Todistuskompleksisuus

Lauselogiikan lauseelle voidaan määrittää ns. *todistuskompleksisuus* [8] jonkin tarkastusmenetelmän suhteen. Intuitiivisesti todistuskompleksisuus on mitta pienimmälle askelmäärälle, jossa menetelmällä on mahdollista osoittaa lause toteutuvaksi tai toteutumattomaksi.

Todistuskompleksisuuskäsite mahdollistaa eri tarkastusmenetelmien teoreettisen vertailun. Todistuskompleksisuuteen pohjautuen voidaan sanoa, että menetelmä *M polynomisesti simuloi* [13] menetelmää M', jos on olemassa polynomi $p(n)$ siten, että jokaiselle lauseelle $\mathcal{C}$ pätee, että jos $\mathcal{C}$ on mahdollista osoittaa toteutuvaksi (toteutumattomaksi) menetelmässä M' askelmäärällä n, niin $\mathcal{C}$ on mahdollista osoittaa toteutuvaksi (toteutumattomaksi) menetelmässä M askelmäärällä $p(n)$.

Erityisesti voidaan eräissä tapauksissa osoittaa, että jollakin menetelmällä *M ei* voida polynomisesti simuloida toista menetelmää M'. Tästä seuraa, että on tapauksia, joilla menetelmä M' on huomattavasti tehokkaampi kuin M.

Todistuskompleksisuustarkastelu on arviointimenetelmistä kenties matemaattisin. Esimerkkinä klassisista tuloksista mainittakoon Hakenin todistus [18], josta suoraan seuraa, että DPLL-menetelmä on pohjimmiltaan eksponentiaaliaikainen. On myös osoitettu [7], että DPLL-menetelmä ilman oppimista ei kykene polynomisesti simuloimaan oppimisella varustettua DPLL-menetelmää.

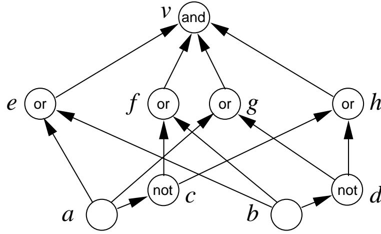
6 Yleistyksistä

Tässä kappaleessa keskitytään toteutuvuustarkastusmenetelmien yleistyksiin, jotka mahdollisesti lyövät itsensä läpi tulevaisuudessa. Eräs tutkimussuunta liittyy toteutuvuusongelman esitysmuotoon; kömpelön konjunktiivisen normaalimuodon käyttämisen sijaan voidaan käyttää muun muassa ns. *Boolen piirejä*, jotka esitellään seuraavassa. Toisena suuntauksena mainitaan *0–1-kokonaislukuoptimointi-* ja *mallienlaskentaongelmien* tehokkaat toteutuvuustarkastukseen pohjautuvat ratkaisumenetelmät.

6.1 Esitysmuodosta

Konjunktiivinen normaalimuoto on varsin kömpelö, vaivalloisesti käytettävissä oleva esitysmuoto monissa tapauksissa. Esimerkiksi mallintarkastusongelman toteutuvuusongelmakuvaus tehdään usein välimuotoisen esityksen kautta. Lisäksi KNM:oon käännettäessä joudutaan usein lisäämään ylimääräisiä muuttujia käännöksen koon hallittavuuden takia; ilman ylimääräisiä muuttujia käännöksen koko kasvaa pahimmassa tapauksessa eksponentiaalisesti. Ylimääräisten muuttujien esittely taas huonontaa DPLL-pohjaisten toteutuvuustarkastimien pahimman tapauksen käyttäytymistä johtuen käyttäytymisen eksponentiaalisesta riippuvuudesta muuttujien määrästä.

Monissa tapauksissa tiivis, luonnollinen ja alkuperäisen ongelman rakenteen säilyttävä kuvaus saadaan käyttämällä *Boolen piirejä*. Boolen piirit ovat suunnattuja, asyklisiä verkkoja, joiden solmuja kutsutaan *porteiksi*. Portit voidaan jakaa kolmeen joukkoon: *tulosportteihin*, *väliportteihin* ja *syöteportteihin*. Jokaiseen tulos- ja väliporttiin liittyy Boolen funktio. Esimerkki Boolen piiristä on kuvassa 6. Kuvan piirissä v on tulosportti, c, d, e, f, g, h väliportteja ja a, b syöteportteja.



Kuva 6: Esimerkki Boolen piiristä.

Piirin kaaret kuvaavat porttien funktionaalista riippuvuutta toisistaan. Esimerkiksi kuvan 6 piirissä porttien b, c, f välinen riippuvuus on muotoa $f = \mathsf{or}(b,c)$, ja edelleen porttien a, c välillä on riippuvuus $c = \mathsf{not}(a)$. Boolen piirien semantiikka on ilmeinen: kiinnitettäessä totuusarvot syöteportteihin määrittyvät muiden porttien totuusarvot yksikäsitteisesti.

Rajoitetussa Boolen piirissä sallitaan piirin porttien totuusarvojen rajoittaminen. Esimerkiksi kuvan 6 piiriin voitaisiin lisätä rajoite "portti v on *tosi*". *Boolen piirien toteutuvuusongelmassa* kysytään, onko annetun rajoitetun Boolen piirin syöteporteille olemassa sellaista totuusarvojakelua, joka toteuttaa piirin rajoitteet. Jos tällainen totuusjakelu on olemassa, sanotaan kyseistä piiriä *toteutuvaksi*, ja muulloin *toteutumattomaksi*.

Kuten lauselogiikan tapauksessa, myös Boolen piirien toteutuvuusongelma on **NP**-täydellinen. Esimerkiksi kuvan 6 piirille ei ole sen toteuttavaa totuusjakelua rajoitteella "portti v on *tosi*". Toisaalta rajoitteilla "portti e on *epätosi*" ja "portti g on *tosi*" totuusjakelu

$$\{a \leftarrow \textit{epätosi},\ b \leftarrow \textit{epätosi}\}$$

toteuttaa piirin.

Erityisen mielenkiintoista on, että DPLL-menetelmälle on määriteltävissä vastine Boolen piireille, katso esimerkiksi [22]. Lisäksi useat nykyisten tehokkaimpien DPLL-pohjaisten tarkastimien käyttämät tekniikat ovat hyödynnettävissä myös Boolen piirejä käsittelevässä vastineessa [36]. Boolen piirien etu KNM-esitykseen nähden on se, että piirien säilyttämää alkuperäistä rakennetta voidaan käyttää suoraan syötteestä lukien hyödyksi haun aikana. Rakenteen säilyttävien KNM-käännöksien aikaansaaminen on haastavaa; katso esimerkiksi [30]. Boolen piireillä operoiviin DPLL:ään pohjautuviin menetelmiin keskittyvä tutkimus onkin tällä hetkellä suuren mielenkiinnon kohteena. Lisäksi Boolen piirien rakenteellisuus antaa oivallisen pohjan muun muassa todistuskompleksisuustarkasteluille [23].

6.2 Lineaarirajoitteista ja mallienlaskennasta

Lineaariset 0–1-rajoitteet ovat muotoa

$$\sum_i a_i x_i \sim b,$$

missä a_i, b ovat kokonaislukuja, $x_i \in \{0,1\}$ ja "$\sim$" $\in$ { "$=$", "$<$", "$\leq$", "$>$", "$\geq$" }. Huomattavaa on, että mielivaltainen klausuuli

$$x_0 \vee x_1 \vee \cdots \vee x_i \vee \neg x_{i+1} \vee \cdots \vee \neg x_n$$

voidaan esittää lineaarirajoitteena

$$x_0 + \ldots + x_i + (1 - x_{i+1}) + \ldots + (1 - x_n) \geq 1.$$

Yleisesti lineaariset 0–1-rajoitteet voidaan kuvata polynomisen kokoisena (mutta suurena) klausuulijoukkona. Osa rajoitteista voidaan tosin usein kuvata tiiviisti ja luonnollisesti klausuuleina. Tehokkaaksi todettuja toteutuvuustarkastustekniikoita onkin pyritty laajentamaan hyväksymään myös lineaarisia 0–1-rajoitteita loogisten rajoitteiden lisäksi. Ajatuksena on hyväksikäyttää tehokkaita toteutuvuustarkastusmenetelmiä klausuulimuotoisiin rajoitteisiin ja samalla hyväksyä myös tiiviimpien lineaaristen 0–1-rajoitteiden käyttö osana ongelmakuvausta. On alustavaa näyttöä siitä, että tämä lähestymistapa on kilpailukykyinen tietyillä sovellutusalueilla jopa kaupallisten huippuunsaviritettyjen — ja huippukalliiden — lineaariepäyhtälöryhmien ratkaisimien[3] kanssa [3]. Tämänkaltaisia menetelmiä olisi mahdollista soveltaa esimerkiksi bioinformatiikan alalta kumpuavien ongelmien, kuten **NP**-täydellisen *usean merkkijonon sovitusongelman* (*multiple sequence alignment*) [31], ratkaisemiseen.

Mallienlaskentaongelmassa kysymys on annetun toteutuvuusongelmatapauksen mallien *lukumäärän* määrittämisestä. Mallienlaskentaongelma on vaativuusteoreettisesti toteutuvuusongelmaa vaikeampi, ns. #**P**-täydellinen [28]. On ilmiselvää, että toteutuvuustarkastumenetelmiä voidaan käyttää suoraan hyväksi mallienlaskentaongelmatapausten ratkaisemisessa hakemalla yksinkertaisesti kaikki ko. tapauksen mallit. Viimeisimpien parin vuoden aikana on herännyt kiinnostus yhä tehokkaampien mallienlaskentaongelmaratkaisimien kehittämiseen pohjautuen toteutuvuustarkastusmenetelmiin. Joitakin lupaavia tuloksia on jo esitetty [33, 5]. Kiinnostus mallienlaskentaongelman ratkaisumenetelmiä kohtaan johtuu sen läheisestä suhteesta *probabilistiseen päättelyyn* [29].

[3]Esimerkiksi ILog CPLEX-ratkaisin.

7 Lopuksi

Lauselogiikan toteutuvuusongelma on yksi tutkituimmista **NP**-täydellisistä ongelmista. Tähän teorian ja käytännön välimaastoon asettuvaan ongelmaan liittyvä algoritmitutkimus on edistynyt huimasti erityisesti viimeksi kuluneiden noin 10–15 vuoden aikana. Toteutuvuustarkastimia on viime vuosina käytetty laajalti erilaisten käytännön ongelmien ratkaisemisessa. Tässä artikkelissa pyrittiin tuomaan esille lauselogiikan toteutuvuustarkastuksen perusperiaatteita sekä tutkimusongelmia, keskittyen systemaattisiin tarkastusmenetelmiin ja käytännöstä kumpuavien ongelmien ratkaisemiseen.

Toteutuvuustarkastukseen liittyvä tutkimusalue on — kenties yllättävänkin — laaja. Tämä artikkeli käsitteli ainoastaan pientä, epätasapainoisesti valittua osaa alueeseen liittyvistä tutkimusongelmista (ja sitäkin hyvin pintapuolisesti). Käsittelyn ulkopuolelle jääneistä aiheista mainittakoon satunnaisiin toteutuvuusongelmatapauksiin liittyvä *faasitransitioilmiö* [14], jolla on vahva kytkentä materiaalifysiikkaan, ja **PSPACE**-täydellisen *kvantifioidun toteutuvuusongelman* (QSAT) [28] ratkaiseminen, sekä stokastiset toteutuvuustarkastusmenetelmät.

Pintapuolisen käsittelyn johdosta lukijaa lämpimästi kehoitetaan tutustumaan hänessä kiinnostusta herättäneisiin toteutuvuustarkastuksen osa-alueisiin artikkelin viitteiden kautta.

Viitteet

[1] Eighth International Conference on Theory and Applications of Satisfiability Testing. http://www.satisfiability.org/SAT05/.

[2] Clay Mathematics Institute. Millennium Problems. http://www.claymath.org/millennium/.

[3] Fadi A. Aloul, Arathi Ramani, Igor L. Markov ja Karem A. Sakallah. Generic ILP versus 0-1 specialized ILP: An update. *Technical Report CSE-TR-461-02*, University of Michigan 2002.

[4] Henrik Reif Andersen. An introduction to binary decision diagrams, 1997. *Lecture notes for 49285 Advanced Algorithms E97*, Department of Information Technology, Technical University of Denmark.

[5] Fahiem Bacchus, Shannon Dalmao ja Toniann Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. *Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS'03)*, s. 340–351. IEEE 2003.

[6] Roberto J. Bayardo ja Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, s. 203–208. AAAI Press 1997.

[7] Paul Beame, Henry Kautz ja Ashish Sabharwal. Understanding the power of clause learning. *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, s. 1194–1201. Morgan Kaufmann 2003.

[8] Paul Beame ja Toniann Pitassi. Propositional proof complexity: past, present, and future. *Bulletin of the European Association for Theoretical Computer Science*, 65:66–89, 1998.

[9] Armin Biere ja Wolfgang Kunz. SAT and ATPG: Boolean engines for formal hardware verification. *Proceedings of 20th IEEE/ACM International Conference on Computer Aided Design (ICCAD'02)*, s. 782–785. IEEE Press 2002.

[10] Edmund Clarke, Armin Biere, Richard Raimi ja Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[11] Edmund Clarke, Orna Grumberg ja Doron A. Peled. *Model Checking*. The MIT Press 2000.

[12] Stephen A. Cook. The complexity of theorem proving procedures. *Proceedings of the 3rd Annual Symposium on Theory of Computing*, s. 151–158. ACM 1971.

[13] Stephen A. Cook ja Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 1979.

[14] James M. Crawford ja Larry D. Auton. Experimental results on the crossover point in random 3SAT. *Artificial Intelligence*, 81(1–2):31–57, 1996.

[15] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan ja Uwe Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[16] Martin Davis, George Logemann ja Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[17] Jun Gu, Paul W. Purdom, John Franco ja Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: a survey. *Satisfiability Problem: Theory and Applications*, *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science* 35, s. 19–152. American Mathematical Society, 1997.

[18] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2–3):297–308, 1985.

[19] John N. Hooker ja V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:177–186, 1995.

[20] Holger H. Hoos ja Thomas Stützle. Local Search Algorithms for SAT: An Empirical Evaluation. *Journal of Automated Reasoning*, 24:421–481, 2000.

[21] Jinbo Huang ja Adnan Darwiche. A structure-based variable ordering heuristic for SAT. *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, s. 1167–1172, Morgan Kaufmann 2003.

[22] Tommi A. Junttila ja Ilkka Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. *Computational Logic – CL 2000; First International Conference*, *Lecture Notes in Artificial Intelligence* 1861, s. 553–567. Springer 2000.

[23] Matti Järvisalo, Tommi A. Junttila ja Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for Boolean circuits. AI&M 15–2004, 8th International Symposium on Artificial Intelligence and Mathematics, 2004. Julkaisu saatavilla: `http://rutcor.rutgers.edu/%7Eamai/aimath04/`.

[24] Henry Kautz ja Bart Selman. Planning as satisfiability. *Proceedings of the 10th European Conference on Artificial Intelligence*, s. 359–363. Wiley 1992.

[25] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, 1992.

[26] Chu Min Li ja Anbulagan. Heuristics based on unit propagation for satisfiability problems. *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, s. 366–371. Morgan Kaufmann 1997.

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang ja Sharad Malik. Chaff: Engineering an efficient SAT solver. *Proceedings of the 38th Design Automation Conference*, s. 530–535. ACM 2001.

[28] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley 1994.

[29] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann 1988.

[30] David A. Plaisted ja Steven A. Greenbaum. A structure–preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[31] Steve Prestwich, Des Higgins ja Orla O'Sullivan. A SAT-based approach to multiple sequence alignment. *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science* 2833, s. 940–944. Springer 2003.

[32] John Alan Robinson. Computational logic: Memories of the past and challenges for the future. *Computational Logic – CL 2000; First International Conference*, *Lecture Notes in Artificial Intelligence* 1861, s. 1–24. Springer 2000.

[33] Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz ja Toniann Pitassi. Combining component caching and clause learning for effective model counting. Seventh International Conference on Theory and Applications of Satisfiability Testing, 2004. Julkaisu saatavilla: `http://www.satisfiability.org/SAT04/`

[34] Bart Selman, Henry A. Kautz ja Bram Cohen. Local search strategies for satisfiability testing. *Proceedings of the Second DIMACS Challenge on Cliques, Coloring, and Satisfiability*, 1993.

[35] Bart Selman, Hector J. Levesque ja David Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, s. 440–446. AAAI Press 1992.

[36] Christian Thiffault, Fahiem Bacchus ja Toby Walsh. Solving non-clausal formulas with DPLL search. In *Proceedings of the 10th International Conference on*

Principles and Practice of Constraint Programming, *Lecture Notes in Computer Science* 3258, s. 663–678. Springer 2004.

[37] Antti Valmari. The State Explosion Problem. Lectures on Petri Nets I: Basic Models, *Lecture Notes in Computer Science* 1491, s. 429–528. Springer 1998.

[38] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz ja Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, s. 279–285. IEEE Computer Society 2001.

[39] Lintao Zhang ja Sharad Malik. The quest for efficient Boolean satisfiability solvers. *Automated Deduction – CADE-18*, *Lecture Notes in Computer Science* 2392, s. 295–313. Springer 2002.