T−79.194 Tietojenkäsittelyteorian seminaari

Epäsystemaattinen haku ja no−good−oppiminen

Henna Ruusu

8.5.2003

Tiivistelmä. Richards ja Richards esittelevät artikkelissaan [1] lauselogiikan toteutuvuustarkistusalgoritmin nimeltä learn−SAT. Algoritmi perustuu epäsystemaattiseen hakuun, mutta on täydellinen no−good−oppimisen ansiosta. Lisäksi algoritmi soveltaa eteenpäin tarkistusta ja binäärijärjestystä. Sekä tilan että ajan osalta algoritmi on pahimmassa tapauksessa erittäin raskas. Se kuitenkin pärjää Richardsin ja Richardsin tekemissä empiirisissä kokeissa varsin hyvin verrattuna silloisiin parhaisiin toteutuvuustarkistimiin.

Johdanto ja käsitteitä

Kun systemaattinen haku käy systemaattisesti läpi kaikki mahdollisuudet jossakin järjestyksessä, epäsystemaattinen haku tyypillisesti käyttää paikallista informaatiota ja satunnaisuutta haun ohjaamiseen. Epäsystemaattiseen hakuun perustuvat algoritmit ovat yleensä tehokkaita ratkaisemaan suuria ongelmia, joilla on useita ratkaisuja. Vastaavasti ne pärjäävät heikosti, kun ratkaisuja on vain yksi tai muutamia. Usein ne ovat epätäydellisiä eivätkä siis selviä tapauksista, joissa ratkaisuja ei ole lainkaan. Algoritmi on täydellinen, jos se pysähtyy äärellisessä ajassa ja löytää aina ratkaisun, mikäli sellainen on.

Richardsin ja Richardsin [1] tarkoituksena on tutkia, voisiko epäsystemaattiseen hakuun perustuva täydellinen algoritmi olla kilpailukykyinen sellaisten SAT−ongelmien ratkaisemisessa, joilla on vain vähän tai ei lainkaan ratkaisuja.

Learn−SAT on pohjimmiltaan yleisen rajoiteongelman (constraint satisfaction problem CSP) ratkaisualgoritmi.

CSP–ongelma

Määritellään CSP–ongelma seuraavasti:

Olkoon muuttujien joukko V. Kullakin muuttujalla $v \in V$ on äärellinen mahdollisten arvojen arvojoukko D_v.

Arvojakelu antaa V:n osajoukolle muuttujia tietyt arvot näiden muuttujien arvojoukoista.

Rajoitteiden joukko on C. Kukin rajoite $c \in C$ on sisältää tietyt muuttujat v_1, v_2, ..., v_k . Rajoite on sellainen arvojakelu näille muuttujille, joka on kielletty.[1]

Arvojakelu P **rikkoo** rajoitetta c, jos $c \subseteq P$.

Arvojakelu on **täydellinen**, jos se antaa arvon kaikille muuttujille.

Arvojakelu on **konsistentti**, jos se ei riko yhtään rajoitetta.

Ongelman **ratkaisu** on täydellinen ja konsistentti arvojakelu.

SAT

Lauselogiikan toteutuvuusongelmassa (SAT) etsitään jonkin lauselogiikan lauseen toteuttavaa arvojakelua. SAT on CSP–ongelman erikoistapaus. Kun SAT–ongelma on konjunktiivisessa normaalimuodossa, ongelman rajoitteet saadaan siitä yksinkertaisesti: Klausuulin literaalit eivät saa yhtä aikaa olla epätosia. Klausuuli (x, y, z) tuottaisi siis rajoitteen {(x, False) (y, False), (z, False)}.

Algoritmin kuvaus

Algoritmi jakautuu neljään osaan:

1. Korjaus ja uudelleenkäynnistys (RestartRepair)

2. Sulauttamisoppiminen (learning–by–merging, LbM)

3. Eteenpäin tarkistus (forward checking)

4. Heuristinen binäärijärjestys (heuristic binary ordering).

Heuristista binäärijärjestystä lukuunottamatta komponentit eivät sisällä erityisesti SAT–ongelmaan liittyviä toimintoja. Jos se jätetään lisäämättä, algoritmi soveltuu yleisen rajoiteongelman ratkaisemiseen. Richards ja Richards ehdottavatkin, että sen sijasta voisi lisätä jonkin muun ongelmatyyppikohtaisen komponentin [1, xx].

---

1 Artikkelissa [1] rajoite on määritelty sen sisältämien muuttujien arvoalueiden karteesisen tulon $D_1 x ... x D_k$ osajoukkona, mutta tällainen määrittely ei ole konsistenttia sen kanssa, miten termiä rajoite käytetään muualla samassa artikkelissa.

RestartRepair

Ongelman ratkaisun hakeminen alkaa antamalla kaikille muuttujille jokin arvo. Näin saadaan täydellinen arvojakelu, joka on yleensä epäkonsistentti. Jokaisella muuttujalla on tästä lähtien koko ajan jokin arvo. Muuttujat arvoineen jakaantuvat kuitenkin kahteen joukkoon.VarsDone sisältää muuttujat, joille on jo kiinnitetty arvo. Sen konsistentti arvojakelu näille muuttujille. VarsLeft sisältää loput muuttujat. Aluksi kaikki muuttujat kuuluvat joukkoon VarsLeft. Konsistenttia arvojakelua VarsDone laajennetaan vähitellen.

Määrittelen vielä konfliktimuuttujan:

Konfliktimuuttuja on muuttuja, jonka nykyinen arvo rikkoo jotakin rajoitetta (mahdollisesti yhdessä muiden muuttujien nykyisten arvojen kanssa).

RestartRepair voidaan kuvata seuraavasti:

0. VarsDone asetetaan tyhjäksi ja VarsLeftiksi asetetaan satunnainen arvojakelu.

Repair:

1. Valitaan VarsLeftistä satunnaisesti konfliktimuuttuja. Mikäli konfliktimuuttujaa ei löydy, nykyinen arvojakelu on ongelman ratkaisu.

2. Valitulle muuttujalle yritetään löytää arvo, joka on yhdessä VarsDonen kanssa muodostaa konsistentin arvojakelun.

3. Mikäli tällainen arvo löytyy, muuttuja arvoineen lisätään VarsDoneen (ja poistetaan VarsLeftistä).

4. Toistetaan 1−3 niin kauan kuin valituille konfliktimuuttujille löytyy VarsDonen kanssa konsistentti arvo.

Restart:

5. Kun konfliktimuuttujalle ei löydy VarsDonen kanssa konsistenttia arvoa (umpikuja), haku alkaa alusta. VarsLeftiksi asetetaan muuttujien nykyinen arvojakelu (VardLeft+VarsDone) ja VarsDone asetetaan tyhjäksi.

RestartRepair−algoritmi on epätäydellinen. Se ei aina löydä ratkaisua, vaikka sellainen olisi olemassa, eikä pysähdy lainkaan, jos ratkaisua ei ole. Kun siihen lisätään learning−by−merging, algoritmista tulee täydellinen.

Sulauttamisoppiminen (LbM)

Kun RestartRepair päätyy umpikujaan, valittua konfliktimuuttujaa ei voida lisätä VarsDoneen millään sen arvojoukon arvolla. VarsDone ei siis voi missään tapauksessa olla osa ongelman ratkaisua. Jotta algoritmi ei myöhemmin yritä rakentaa samanlaista VarsDonea, lisätään prosessiin no−good−oppiminen.

Määritellään no−good seuraavasti:

No−good on muuttujajoukon arvojakelu, joka rikkoo rajoitteita tai jota ei voida laajentaa ongelman ratkaisuksi. Arvojakelu P rikkoo no−goodia ng, jos ng $\subseteq$ P. Jos arvojakelu rikkoo no−goodia, se on itsekin no−good. Kaikki arvojakelut rikkovat tyhjää no−goodia $\emptyset$.

No−good on siis ongelman ratkaisemisen kannalta aivan vastaavanlainen rajoite kuin ongelman alkuperäiset rajoitteetkin. Rajoitteiden joukkoon lisättyjä no−goodeja ei myöskään poisteta myöhemmin missään vaiheessa ja niistä voidaan myös käyttää nimitystä rajoite. Määritelmän mukaan kaikki alkuperäiset ongelman rajoitteet ovat myös no−goodeja.

Koska umpikujatilanteessa silloinen VarsDone on ratkaisun saavuttamisen kannalta mahdoton arvojakelu, se voidaan lisätä sellaisenaan rajoitteiden joukkoon. Näin jää kuitenkin epäselväksi, mitkä muuttujien arvot varsinaisesti rikkoivat rajoitteita ja uusi no−good voi olla tarpeettoman suuri kooltaan. Mitä pienemmän no−goodin löydämme, sitä enemmän se rajaa mahdollisia muuttujien arvoja pois.

ESIMERKKI: no−good {(x,1), (y,1), (z,0), (v,0)} kertoo meille vain, että muuttujat x, y, z ja v eivät voi saada tätä nimenomaista arvojakelua. Lyhyempi no−good {(x,1)} kertoisi meille suoraan, että muuttuja x ei koskaan voi saada arvoa 1.

LbM−proseduuri voidaan kuvata seuraavasti:

> Olkoon P konsistentti osittainen arvojakelu, jota ei voida konsistentisti laajentaa muuttujalle v. Silloin kullekin v:n arvojoukon arvolle x_i ($1 \leqslant i \leqslant m$) on olemassa no−good ng_i, jonka perusteella v:lle ei voida antaa kyseistä arvoa. Muodostetaan uusi no−good ng yhdistämällä no−goodit ng_1, ng_2, ..., ng_m seuraavasti: ng = $\bigcup(ng_i - (v, x_i))$. P selvästikin rikkoo tätä uutta no−goodia.
>
> Tässä ng_i :ksi voidaan valita ensimmäinen rajoite, jonka huomataan estävän antamasta v:lle kyseistä arvoa. Se löydetään, kun v:lle yritetään antaa arvo v_i ja tarkistetaan, rikkooko tämä jotakin rajoitetta.

ESIMERKKI:

Olkoon ongelmalla kaksi rajoitetta: ng_1 = {(s,1), (v,0)} ja ng_2 = {(t, 2), (v, 1)}. Olkoon VarsDone {(s, 1), (t, 2), (u, 3)}. Yritämme laajentaa sitä muuttujaan v, jonka arvoalue on {0,1}.

Rajoite ng_1 estää antamasta v:lle arvoa 0 ja rajoite ng_2 estää antamasta sille arvoa 1.

Sulauttamisoppiminen tuottaa uuden no−goodin seuraavasti:

$ng = \bigcup_i (ng_i - (v, x_i)) = \{(s, 1)\} \cup \{(t, 2)\} = \{(s, 1), (t, 2)\}$

Näin saimme lyhyemmän no−goodin kuin mitä VarsDone {(s, 1), (t, 2), (u, 3)} sellaisenaan olisi ollut.

RestartRepair{LbM}

Kun RestartRepair−algoritmiin sisällytetään sulauttamisoppiminen, saadaan seuraavan pseudokoodin mukainen algoritmi RestartRepair{LbM}.

```
VarsDone ← konsistentti osittainen arvojakelu (aluksi tyhjä)
VarsLeft ← arvojakelu muuttujille, jotka eivät ole VarsDonessa
Status ← Unknown
while Status = Unknown do
    if valitaan satunnaisesti konfliktimuuttuja VarsLeftistä (jos voidaan valita)
    then Consistent ← False
    ng ← Ø
    while Consistent = False do
        val'∈ D_v
        Consistent ← check(VarsDone ∪ {(v, val')})
        if Consistent = False then Dv ← Dv −{val'}
        ng ← ng ∪ VarsDone ↓_c (c rikottu rajoite)
        else vl ← val'
        endif
    if D_v = Ø then
        restart(VarsDone, VarsLeft)
        record(ng)
        if ng = Ø then Status ← Insoluble
        endif
    else VarsDone ← VarsDone ∪ {(v, vl)}
        VarsLeft VarsLeft − {(v, val)}
    endif
    else return VarsDone ∪ VarsLeft on ratkaisu
    Status ← Solved
endif
```

Tässä funktio **check** tarkistaa, onko parametrina oleva arvojakelu konsistentti. Funktio **restart** aloittaa algoritmin ajamisen alusta asettaen VarsLeftiksi yhdisteen VarsDone ∪ VarsLeft. Funktio **record** lisää parametrin rajoitteiden joukkoon. Merkintä VarsDone $\downarrow_c$ tarkoittaa, että VarsDonesta otetaan vain c:n muuttujat.

Oikeellisuus ja täydellisyys

RestartRepair{LbM} on täydellinen ja oikeellinen menetelmä. Siihen lisätään vielä eteenpäin tarkistus ja binäärijärjestys, jotta saadaan learn−SAT. Mutta nämä lisäykset eivät vaikuta menetelmän täydellisyyteen ja oikeellisuuteen, joten seuraavat todistukset pätevät myös learn−SAT:lle.

Oikeellisuus

Menetelmä on oikeellinen, jos se ei koskaan johda väärään tulokseen. Oletamme, että algoritmin käytössä on oikeellinen tarkistusmenetelmä sille, onko joku arvojakelu konsistentti rajoitteiden kanssa. Näin on selvää, että jos rajoitteet toteuttava arvojakelu löytyy, se on ongelman oikea ratkaisu. Lisäksi täytyy osoittaa, että kun algoritmi tuottaa tuloksen *Insoluble*, toteuttavaa arvojakelua ei todella ole olemassa.

Milloin algoritmi sitten voisi mahdollisesti tuottaa väärän tuloksen? Algoritmi voisi hukata oikean ratkaisun, jos se tuottaisi jossain kohdassa väärän no−goodin, joka rajoittaisi ratkaisuavaruudesta pois liikaa arvojakeluja.

VÄITE: RestartRepair{LbM} on oikeellinen.

Todistus: Oletetaan, että RestartRepair{LbM} on päätynyt umpikujaan. Olkoon v muuttuja, jolle ei voida konsistentisti asettaa arvoa. Olkoot ng_1, ng_2, ... ng_m joukko no−goodeja, jotka rajaavat koko v:n arvoalueen $\{x_1, x_2, \ldots, x_m\}$ pois. Olkoon ng näistä sulauttamalla yhdistetty no−good.

Oletetaan nyt, että tämä ng ei olisi oikea no−good. Silloin olisi olemassa CSP:n ratkaisu S siten että $ng \subseteq S$. Olkoon x_i muuttujan v arvo S:ssä. Silloin myöskin $ng \cup \{(v, x_i)\} \subseteq S$. Kuitenkin tiedämme, että $ng_i \subseteq ng \cup \{(v, x_i)\}$. Tämän mukaan S rikkoisi ng_i:tä eikä siten voi olla ratkaisu, ristiriita. Siis ng:n täytyy olla oikea no−good. □

Täydellisyys

VÄITE: RestartRepair{LbM} on täydellinen.

Todistus: Edellä on osoitettu, että haun aikana tuotetut no−goodit eivät eliminoi ongelman ratkaisuja pois. Täytyy vielä osoittaa, että haku päättyy aina äärellisessä ajassa oikean ratkaisun löytymiseen tai tyhjään no−goodiin (joka tarkoittaa, että mikään arvojakelu ei voi olla ratkaisu). Kun muuttujia on äärellinen määrä, algoritmi selvästikin päätyy äärellisessä määrässä askelia joko ongelman ratkaisuun tai umpikujaan, jossa VarsDonea ei voida enää laajentaa konsistentisti. Kun päädytään umpikujaan, lisätään rajoitteiden joukkoon uusi no−good. Tämä no−good eliminoi hakuavaruudesta pois vähintäänkin silloisen arvojakelun kaikille muuttujille, eli VarsDonen ja VarsLeftin unionin. Koska no−goodeja ei koskaan poisteta, niiden pois rajaamien arvojakelujen määrä kasvaa joka kerta, kun päädytään umpikujaan. Lopulta koko hakuavaruuden täytyy rajautua pois, jos ratkaisua ongelmalle ei ole olemassa. □

Sitä, miten algoritmi päätyy tyhjään no−goodiin, valaissee esimerkki parhaiten.

ESIMERKKI:

Olkoon rajoitteiden joukossa seuraavat neljä rajoitetta: {(x, 0), (y,0)}, {(x, 1), (y,1)}, {(x, 0), (y,1)} ja {(x, 1), (y,0)}. Muuttujien x ja y molempien arvo on aluksi 0 ja arvoalue {0, 1}.

Valitaan konfliktimuuttujaksi x ja lisätään (x, 0) VarsDoneen. Valitaan konfliktimuuttujaksi y. Rajoite {(x, 0), (y,0)} estää antamasta y:lle arvoa 0 ja rajoite {(x, 0), (y,1)} estää antamasta sille arvoa 1.

Saamme uuden no−goodin {(x, 0)} ∪ {(t, 0)} = {(x, 0)} ja aloitetaan uudestaan tyhjästä VarsDonesta.

Oletetaan, että seuraallakin kerralla valitaan konfliktimuuttujaksi x. Lisätään (x, 1) VarsDoneen ja konfliktimuuttujaksi valitaan y. Nyt {(x, 1), (y,1)} ja {(x, 1), (y,0)} tuottavat uuden no−goodin {(x, 1)}.

Kun seuraavalla uudelleenkäynnistyksellä valitaan jälleen konfliktimuuttujaksi x, {(x, 0)} ja {(x, 1)} estävät antamasta silel mitään arvoa. Näistä muodostuu uusi no−good { }.

Eteenpäin tarkistaminen

RestartRepair{LbM}:ään voidaan lisätä hakua eteenpäin tarkastelevia tekniikoita. Ideana on, että ennen kuin VarsDoneen lisätään muuttuja arvoineen, voidaan tutkia, johtaako se kenties tilanteeseen, jossa jollekin toiselle muuttujalle ei voida antaa mitään arvoa konsistentisti. Kullekin muuttujalle VarsLeftissä pidetään kirjaa siitä, mitkä arvot sen arvoalueessa vielä ovat mahdollisia lisätä VarsDoneen. VarsDonen kanssa epäkonsistentit arvot rajataan pois.

Kun jokin muuttuja lisätään VarsDoneen, sen aiheuttamat rajoitukset VarsLeftin muuttujien arvoalueisiin päivitetään. Kun muuttuja valitaan korjattavaksi eli yritetään lisätä se VarsDoneen, sen mahdollinen arvoalue on jo saattanut kaventua aiempien VarsDoneen tehtyjen lisäysten yhteydessä. VarsDonen muuttujien arvot katsotaan kiinnitetyiksi, joten niistä ei enää tarvitse pitää kirjaa.

Jos muuttujan v arvoalueesta on eliminoitu arvo a, kuuluu (v, a) johonkin no−goodiin, jonka muiden muuttujien arvot ovat VarsDonessa. Näitä muita muuttujien arvoja kutsutaan (v, a):n **eliminointisyyksi.** Kullekin muuttujalle VarsLeftissä pidetään kirjaa sen eliminoitujen arvojen eliminointisyistä. Jos yhden muuttujan arvoja on eliminoitu useita, niiden eliminointisyitä ei talleteta erikseen vaan ainoastaan niiden unioni Ev. Se kertoo syyn kaikkien tähän mennessä eliminoitujen arvojen eliminoimiselle. **Arvoalueen häviämiseksi** (domain wipe−out) kutsutaan tilannetta, jossa kaikki mahdolliset arvot jollekin muuttujalle on eliminoitu.

Aluksi kaikkien VarsLeftin muuttujien eliminointisyiden joukko on tyhjä. Algoritmi valitsee konfliktimuuttujan v ja pyrkii antamaan sille sellaisen arvon, joka ei aiheuta millekään VarsLeftin muuttujalle koko arvoalueen häviämistä. Jos tällainen arvo voidaan muuttujalle antaa v:lle, se lisätään VarsDoneen. Sen aiheuttamat rajoitukset VarsLeftin muuttujien arvoalueisiin päivitetään ja eliminointisyyt lisätään niiden muuttujien kohdalle, joiden arvoalue kapenee. Jos v:lle ei voida antaa mitään arvoa niin että jonkin toisen muuttujan w_i arvoalue ei häviäisi, on saavutettu umpikuja.

Uusi no−good saadaan silloin yhdistämällä v:n jo eliminoitujen arvojen eliminointisyyt ja w_i:n arvojen eliminointisyyt:

$ng = \bigcup_i Ew_i \cup Ev.$

Jos muutujalle v:lle annetusta arvosta seuraa, että muuttujan w arvoalue kutistuu yhteen mahdolliseen arvoon, w:n oletetaan saavan tämän arvon ja tämän aiheuttamat muutokset päivitetään jälleen muuttujien arvoalueisiin.

Binäärinen järjestys ja learn−SAT

Tähän asti käsitellyt algoritmit ovat soveltuneet yleisen CSP:n ratkaisemiseen. Lisätään mukaan vielä SAT−spesifinen komponentti ja saamme valmiiksi learn−SAT:n määrittelyn.

Binäärinen järjestys (binary ordering) on heuristiikka, jolla SAT−ongelman ratkaisun hakua ohjataan löytämään mahdollisimman pian umpikuja. Tarkoituksena on muodostaa no−good, joka rajaa mahdollisimman paljon hakuavaruutta. SAT−ongelman oletetaan olevan konjunktiivisessa normaalimuodossa.

Binäärinen järjestys voidaan kuvata seuraavasti:

(1) Valitaan mahdollisista konfliktimuuttujista se, joka on mukana suurimmassa lukumäärässä binääriklausuuleja.

(2) Tälle muuttujalle annetaan VarsDonen kanssa konsistenteista arvoista se, josta seuraa suurin määrä binääriklausuuleja.

Artikkelissa [1] ei tarkemmin selitetty, miten binäärinen järjestys tässä yhteydessä toimisi. Klausuuleja vastaavat tietenkin rajoitteet. Ilmeisesti täytyy jollakin uudella tekniikalla pitää kirjaa siitä, miten muuttujien arvojen kiinnittäminen vaikuttaisi klausuuleihin. Suoraan näitä muutoksia ei voida tehdä rajoitteisiin, koska rajoitteet ovat pysyviä kierroksesta toiseen. En määrittele algoritmin toimintaa tässä tämän tarkemmin, koska Richards ja Richardskaan eivät niin tee.learn−SAT pseudokoodilla:

learn−SAT

Valmis learn−SAT pseudokoodilla näyttää seuraavalta:

```
VarsDone ← konsistentti osittainen arvojakelu (aluksi tyhjä)
VarsLeft ←  arvojakelu muuttujille, jotka eivät ole VarsDonessa
foreach variable v, E_v (eliminointisyyt)
Status  ←  Unknown
while Status = Unknown do
    if valitaan satunnaisesti VarsLeftistä konfliktimuuttuja, joka on mukana mahd.
              suuressa määrässä binääriklausuuleja
     then Consistent ← False
```

```
ng ← Ev
while Consistent = False do
    val' ∈ Dv, v:n jäljellä oleva arvoalue
    Consistent ← forward−check(VarsDone ∪ {(v, val')},
        VarsLeft − {(v, val)})
    if Consistent = False then Dv ← Dv −{val'}
        ng ← ng ∪ VarsDone ↓EW (w:n arvoalue hävinnyt)
        undo−forward−check(VarsDone ∪ {(v, val')},
                    VarsLeft − {(v, val)})
    else nb ← no_of_bin_clauses((VarsDone ∪ {(v, val')}
                    ∪ (VarsLeft − {(v, val)}))
        ValList ← ValList ∪ {(nb, val')}
    endif
    if ValList = Ø then
        restart(VarsDone, VarsLeft)
        record(ng)
        if ng = Ø then Status ← Insoluble
        endif
    else select (nb, vl) ∈ ValList, jolla pienin nb
        VarsDone ← VarsDone ∪ {(v, vl)}
        VarsLeft VarsLeft − {(v, val)}
    endif
    else return VarsDone ∪ VarsLeft on ratkaisu
        Status ← Solved
    endif
```

Funktio **forward−check** tutkii voidaanko (v, val') lisätä VarsDoneen ilman että jonkun VarsLeftin muuttujan arvoalue häviää. Samalla se päivittää muiden muuttujien arvoalueisiin rajoitukset. Funktio **undo−forward−check** taas palauttaa niiden arvoalueet ennalleen.

Laskennallinen vaativuus

Richards ja Richards ovat tutkineet RestartRepair{LbM}:n laskennallista vaativuutta. Sen tutkiminen on riittävää learn−SAT:nkin kompleksisuuden tarkastelemiseksi. He ovat tarkastelleet vierailtujen hakusolmujen määrää, rajoitteiden toteutumistarkistusten määrää ja tilan tarvetta.

Pahimmillaan haku on erittäin raskas. Se vierailee $O(n(d + 1)^n)$ solmussa. Rajoitteiden toteutuminen tarkistetaan $O(n(d + 1)^{2n})$ kertaa. No−goodit vievät tilaa $O(n(d + 1)^n)$.

Aikaa sekä tilaa kuluu siis pahimmillaan eksponentiaalisesti.

Hieman artikkelin empiirisistä tuloksista

1. **Yksiratkaisuiset SAT−ongelmat**
   learn−SAT ihan hyvä verrattuna Tableau{CJB+3rdLrn}−algoritmiin, mutta tiheydellä 3,4 pärjää jostain syystä huomattavan huonosti.

2. **Strukturoidut ratkeamattomat ongelmat**

Verrattuna relsat−algoritmiin learn−SAT pärjää kilpailukykyisesti. Tiettyä ongelman koosta riippuvaa rajaa pienemmillä klausuulitiheyksillä learn−SAT on aina parempi.

3. **Literaalisäännölliset ongelmat**
   learn−SAT häviää epäsystemaattiselle weight−algoritmille, mutta voittaa selvästi relsatin.

4. **Ratkeamattomat ongelmat DIMACS−benchmarkeista**
   learn−SAT päihittää relsatin kaikissa ongelmissa paitsi kyyhkyslakkaongelmissa.

Artikkelin johtopäätöksiä ja omaa pohdintaa

Artikkelissa esitettiin kaksi kysymystä jatkotutkimukselle:

1. Pärjääkö learn−SAT hyvin vain pienillä klausuuleilla?

2. Voisiko vanhoja no−goodeja tehokkaasti poistaa, kun uusi peittää ne?

Artikkelissa on mielestäni mielenkiintoista yritys yhdistää erilaisia tekniikoita ja tutkia, miten ne yhdessä vaikuttavat. Siinä on laajasti vertailuja, joilla pyritään hahmottamaan, millaisiin ongelmiin learn−SAT puree hyvin. Koska learn−SAT:ssa yhdistetään monia lähestymistapoja, vaatisi lisätutkimuksia selvittää, miksi se pärjää hyvin jossain ja huonosti jossain. Vaikka learn−SAT ei nykyisiin SAT−solvereihin verrattuna enää ole kilpailukykyinen, on hyödyllistä hahmottaa tarkemmin, mistä suorituskyky muodostuu. Joskus onnistutaan ehkä sen tiedon pohjalta löytämään jotain uutta yhdistelmää, joka toimii tehokkaammin.

Viitteet

[1] E. Thomas Richards, Barry Richards, Nonsystematic Search and No−Good Learning, Journal of Automated Reasoning 24 (2002) 4, 483−533.