T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Raivo Laanemets

DPLL protseduur lahendite loendamiseks

Bakalaureusetöö (4 AP)

Juhendaja: Tõnu Tamme, MSc

Autor: ..... "....." jaanuar 2008

Juhendaja: ..... "....." jaanuar 2008

TARTU 2008

Sisukord

Sissejuhatus

Lausearvutusvalemi lahendite loendamise probleem on tihedalt seotud kehtestatavuse probleemiga ja omab seetõttu suurt teoreetilist ning praktilist tähtsust. Mitmed olulised probleemid on teisendatavad nendele probleemidele. Seetõttu on tähtis nii loendamise kui kehtestatavuse algoritmide efektiivsus, st. töökiirus. Selles vallas on viimasel ajal toimunud palju uuringuid ning erinevate lahendusalgoritmide kohta on kirjutatud kümneid artikleid.

Valemi kehtestatavus kui otsustusprobleem on *NP-täielik* [Co71], mis paigutab ta samasse keerukuslassi kümnete teiste oluliste probleemidega.

Lahendite loendamise probleem on kehtestatavusele vastav loendamisprobleem. Tema kuulub keerukusklassi *#P-täielik* [Ro96]. Samuti on teada, et lahendite arvu isegi ligikaudne leidmine väikese garanteeritud veaga on arvutuslikult raske ülesanne. Klassis *#P-täielik* asuvad ka mitmed tehisintellekti valdkonnast tuntud ülesanded, näiteks tuletus Bayes'i võrgus ja teised tuletusprobleemid mittetäieliku teadmusega (tõenäosuslikes) süsteemides.

Kehtestatavuse jaoks on teada palju algoritme. Üheks neist on *DPLL (Davis-Putnam-Logemann-Loveland)* protseduur [Dav60, Dav62], mille edasiarendatud variandid on kõrge efektiivsusega lahendusprogrammides kasutusel ka tänapäeval, rohkem kui 40 aastat pärast originaalalgoritmi publitseerimist.

Hiljuti selgus, et *DPLL* algoritmi saab küllaltki edukalt täiendada ka lahendite loendamise jaoks [Bir99].

Töö ülesehitus ja eesmärgid

Käesolevas töös vaadeldakse originaalsest *DPLL* algoritmist saadud lausearvutusvalemite lahendite loendamisalgoritmi.

Töö alguses esitatakse probleemi formaalne kirjeldus ja töös läbivalt kasutatavate mõistete definitsioonid.

Esimeses osas esitatakse algoritmide kirjeldus ning pseudokood, selgitatakse lahti kasutatud ideed ja antakse põhjendus, miks ühe või teise põhimõtte tarvitamine on lubatav ja mõistlik.

Teises osas antakse lihtne realisatsioon programmerimiskeeles Prolog. Põhiliselt sisaldab see peatükk kasutatud andmestruktuuride ja protseduuride kirjeldusi ning näiteid nende kasutamisest.

Tööd jääb lõpetama lisaosana kaasa pandud programmi dokumentatsioon, s.o. koostatud protseduuride sisend/väljundargumentide kirjeldused ning lühike ülevaade sellest, mida konktreetne protseduur teeb. Programmi masinloetava lähetkoodi võib leida kaasapandud optiliselt andmekandjalt.

Definitsioonid

Muutujaks nimetame suvalist sümbolit x fikseeritud sümbolite hulgast. *Literaaliks* l loeme muutujat x ($l \equiv x$) või tema eitust $-x$ ($l \equiv -x$). *Klausliks* C loeme literaalide disjunktsiooni $C = l_1 \vee \cdots \vee l_n$. *Konjuktiivsel normaalkujul valemiks* (KNK) F loeme klauslite konjuktsiooni $F = C_1 \wedge \cdots \wedge C_k$. Muutuja x *väärtustuseks* nimetame funktsiooni v, $v(x) \in \{0, 1\}$. Literaal l on tõene, kui $l \equiv x$ ja $v(x) = 1$ või $l \equiv \neg x$ ja $v(x) = 0$. Literaali l on väär, kui $l \equiv x$ ja $v(x) = 0$ või $l \equiv -x$ ja $v(x) = 1$. Literaali $l \equiv x$ *komplementaariks* on literaal $\bar{l} \equiv \neg x$ ja literaali $l' \equiv \neg x$ komplementaariks on $\bar{l'} \equiv x$. Klauslit $C = l_1 \vee \cdots \vee l_n$ nimetame väärtustusel v tõeseks ($v(C) = 1$), kui C sisaldab vähemalt ühte tõest literaali, vastasel korral vääraks ($v(C) = 0$). Valemit $F = C_1 \wedge \cdots \wedge C_k$ loeme väärtustusel v tõeseks ($v(F) = 1$), kui ükski klauslitest $C_1, \ldots, C_k$ pole väär, vastasel korral vääraks ($v(F) = 0$).

n-muutuja Boole'i funktsiooniks f nimetame funktsiooni signatuuriga $f : \{0, 1\}^n \to \{0, 1\}$. On selge, et suvalist Boole'i funktsiooni saab esitada mingi talle vastava lausearvutuse valemiga.

Kehtestatavusprobleem

Olgu antud n-muutuja Boole'i funktsioon $f(x_1, \ldots, x_n)$. Kehtestatavusprobleem küsib, kas leidub[1] selline muutujate $x_1, \ldots, x_n$ väärtustus, kus $f(x_1, \ldots, x_n) = 1$.

Loendamisprobleem

Olgu antud n-muutuja Boole'i funktsioon $f(x_1, \ldots, x_n)$. Lahendite loendamise probleem on loendamisprobleem, mitu erinevat $x_1, \ldots, x_n$ väärtustust leidub, kus $f(x_1, \ldots, x_n) = 1$.

[1] Praktilistes rakendustes omavad sageli tähtsust ka tegelikud muutujate väärtused.

Peatükk 1

DPLL algoritm

DP (Davis-Putnam) algoritm [Dav60] oli üks esimesi süstemaatilisi algoritme esimest järku predikaatarvutuse valemitega kirja pandud teoreemide tõestamiseks arvuti abil. Olgugi, et algoritm on küllaltki ebaefektiivne võrreldes paar aastat hiljem (1965) avaldatud Robinsoni algoritmiga, on tarvitatud sellest algoritmist pärit meetodeid hiljem kehtestatavuse algoritmide arendamiseks. *DP* kasutas loogilise tõesuse kontrollimise asemel valemi eituse mittekehtestatavuse väljaselgitamist, mis on esimesega samaväärne probleem.

Töö piiratud maht võimaldab anda ainult väga põgusa ülevaate originaalalgoritmist. Algoritm koosnes neljast osast:

1. Esialgse valemi F eituse $\neg F$ leidmine.
2. Valemi $\neg F$ kirjutamine prenex-kujule[2] G_p (prefikskuju leidmine).
3. G_p eksistentsikvantorite eemaldamine vastavate muutujate funktsionaalsümbolitega asendamise teel (skolemiseerimine[2]). Saadakse valem G_s. Tähtis on teada, et G_p ja G_s ei ole loogiliselt ekvivalentsed, kuid on mittekehtestatavuse suhtes ekvivalentsed, st. G_p on mittekehtestatav parajasti siis, kui G_s on mittekehtestatav.
4. Valemist G_s muutuja- ja kvantorivabade lausearvutusvalemite $G_{s,1}, G_{s,2}, \ldots$ genereerimine kuni nendest n esimese valemi konjuktsioon $G_{s,1} \wedge \cdots \wedge G_{s,n}$ on mittekehtestatav.

Algoritm töötas nii kaua, kuni sammus (4) oli genereeritud piisaval hulgal kvantori- ja muutujavabu valemeid. On ära näidatud, et kui esialgne valem F on loogiliselt tõene, siis leidub lõplik kogus valemeid $G_{s,1}, G_{s,2}, \ldots$ ja programm lõpetab oma töö, vastasel

[2]Prenex-kuju/skolemiseerimine jt. valdkonnaga seotud mõisted seletatakse lahti allikas [Schö94].

korral jääbki ta valemeid genereerima. Kuna predikaatloogika on poollahenduv, siis see ongi parim tulemus, mille me võime saada.

Reeglid sammus (4) valemite konjuktsiooni $W = G_{s,1} \wedge \cdots \wedge G_{s,n}$ kehtestatavuse kindlakstegemiseks olid järgmised:

1. *Ühikklauslite elimineerimine*

   (a) Kui valem W sisaldab ühikklauslit C_i literaaliga l ja samuti ühikklauslit literaaliga $\bar{l}$, siis sisaldab W vastuolu ning on mittekehtestatav.

   (b) Kui (a) ei ole rakendatav ja valem W sisaldab ühikklauslit C_i literaaliga l, siis kustutada valemist W kõik klauslid, mis sisaldavad literaali l ja igast klauslist, mis sisaldab l komplementaari $\bar{l}$, kustutada $\bar{l}$. Saadav valem W' on mittekehtestatav parajasti siis, kui seda on W.

2. *Puhta literaali elimineerimine*
   Kui muutuja x esineb valemi W klauslites ainult positiivse või negatiivse literaalina, siis võib kõik muutujat x sisaldavad klauslid eemaldada, saades valemi W', mis on mittekehtestatav parajasti siis, kui seda on W.

3. *Muutujate elimineerimine*
   Kui valem W on kujul $W = (A \vee l) \wedge (B \vee \bar{l}) \wedge R$, kus A, B ja R literaali l ega tema komplementaari $\bar{l}$ ei sisalda, siis võib valemi W ümber kirjutada kujule $W' = (A \vee B\vee) \wedge R$. Valem W on mittekehtestatav parajasti siis, kui W' on mittekehtestatav.

Hiljem publitseeritud artiklis [Dav62] asendatakse muutujate elimineerimisreegel teistsugusel kujul oleva nn. *jaotamisreegliga*:

Kui valem W on kujul $W = (A \vee l) \wedge (B \vee \bar{l}) \wedge R$, kus A, B ja R literaali l ega tema komplementaari $\bar{l}$ ei sisalda, siis võib valemi W jaotada valemiteks $W_1 = A \wedge R$ ja $W_2 = B \wedge R$ ning esialgne valem on mittekehtestatav parajasti siis kui mõlemad W_1 ja W_2 on mittekehtestatavad.

Koos selle reegliga moodustab *DP* algoritm algoritmi *DPLL*.

Hetkel levinud *DPLL*-tüüpi kehtestatavuse algoritmid kasutavad originaalsest algoritmist erinevat lähenemist. Eelkõige on algoritmid realiseeritud tagurdusalgoritmidena, mis töötavad muutujate osalise väärtustusega, samm-sammult valides muutujatele väärtused, lihtsustades valemit pärast iga väärtustamist, kuni saadakse selline lõpptulemus, mille korral valemit võib lugeda seni leitud väärtustuse piires tõeseks.

1.1 Boole-Shannoni dekompositsioon

Boole-Shannoni dekompositsioon (lahutus) vastab *DPLL* algoritmis jaotamisreeglile. Sisuliselt on siin tegemist universaalse vahendiga n-muutuja ($n > 0$) Boole'i funktsiooni avaldamiseks kahe, või erijuhul ühe, $n-1$ muutuja Boole'i funktsiooni kaudu.

Olgu antud antud n-muutuja Boole'i funktsioon $f(x_1, \ldots, x_n)$. Sellest funktsioonist võime saada kaks $n-1$-muutja funktsiooni, asendades muutuja x_i väärtusega 1 esimeses ja 0 teises. Nendeks funktsioonideks on:

$$f_{\bar{x_1}}(x_1, \ldots, x_{i-1}, x_{i+1} \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1} \ldots, x_n)$$

ja

$$f_{x_1}(x_1, \ldots, x_{i-1}, x_{i+1} \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1} \ldots, x_n).$$

Sealjuures kehtib ka samaväärsus

$$f = f_{\bar{x_1}} \vee f_{x_1},$$

mille korrektsuse tõestuse võib leida allikast [Tom07]. Põhimõtteliselt on võimalik kehtestatavuse algoritm koostada ainult dekompositsiooni kasutades, sest rakendades seda rekursiivselt järjest igale alamvalemile, jõuame ükskord olukorrani, kus kõik muutujad on asendatud väärtustega 1 või 0, ning saame valemi väärtuse välja arvutada. Paraku oleks see väga ebaefektiivne algoritm, võrreldav tõeväärtustabeli koostamisega.

Vaatleme dekompositsiooni rakendust KNK valemile $F = (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$ muutuja y järgi. Saame valemid:

$$F_y = (x \vee 0 \vee z) \wedge (\neg x \vee 1 \vee \neg z) \wedge (x \vee 0 \vee \neg z) \wedge (\neg x \vee 0 \vee \neg z)$$

ja

$$F_{\bar{y}} = (x \vee 1 \vee z) \wedge (\neg x \vee 0 \vee \neg z) \wedge (x \vee 1 \vee \neg z) \wedge (\neg x \vee 1 \vee \neg z).$$

Oh lihtne märgata, et saadud valemeid oleks võimalik oluliselt lihtsustada. Üldiselt saame lihtsustamise põhimõtted kirja panna lihtsa eeskirjana, mis lihtsustab dekompositsiooni $F \Leftrightarrow F_x \vee F_{\bar{x}}$ valemid F_x ja $F_{\bar{x}}$:

1. Valemist F_x eemalda kõik klauslid, kus muutuja x esineb positiivselt ja kõikidest F_x klauslitest, kus x esineb negatiivselt, eemalda x.

2. Valemist $F_{\bar{x}}$ eemalda kõik klauslid, kus muutuja x esineb negatiivselt ja kõikidest $F_{\bar{x}}$ klauslitest, kus x esineb positiivselt, eemalda x.

See algoritm võimaldab lihtsustada F_x ja $F_{\bar{x}}$ n.ö. vahepealse väljarvutamiseta, kuigi asendades x väärtuse valemisse sisse, saame valemi lihtsustada ka tuntud lausearvutuse samaväärsusi $x \vee 0 \equiv x$ ja $x \vee 1 \equiv 1$ kasutades.

Võttes eespool toodud näite uuesti ette, rakendame lihtsustusi leitud valemitele F_y ja $F_{\bar{y}}$, saades semantiliselt samaväärsed, kuigi süntaktiliselt märksa lihtsamad KNK valemid:

$$F_y = (x \vee z) \wedge (x \vee \neg z) \wedge (\neg x \vee \neg z)$$

ja

$$F_{\bar{y}} = (\neg x \vee \neg z).$$

Dekompositsiooni ei saa rakendada kuitahes palju arv kordi, sest valemi muutujate arv väheneb iga korraga ühe võrra (kuigi lihtsustuste tõttu võivad kaduda valemist ka need muutujad, mille järgi ei ole lahutust rakendatud). On võimalik kahe erineva olukorra tekkimine:

(*a) Valemist eemaldatakse kõik klauslid - sellisel juhul ei saa valem enam vääraks muutuda, sest ei ole võimalik, et mõni valemi klausel oleks väär. Sellisel juhul on valem tõene.

(*b) Valemisse juhtub sisse jääma klausel, millest on eemaldatud kõik muutujad - selline valem on väär, kuivõrd temas leidub väär klausel.

Rakendades dekompositsiooni näites saadud valemile $F'_{\bar{y}}$ edasi muutuja x järgi, saame omakorda valemid $F_{\bar{y},x} = \neg z$ ja $F_{\bar{y},\bar{x}}$, millest viimane ei sisalda ühtegi klauslit ja on seetõttu tõene ning seega oleme leidnud kehtestava väärtustuse $\{y = 0, x = 0\}$.

Pannes kokku nii väärtustamise kui valemite lihtsustamise, saab koostada järgmise algoritmi (1):

Algoritm 1. SAT(F)

Sisend: konjuktiivsel normaalkujul valem F.
Väljund: tulemus, kas valem F on kehtestatav.

1: **if** F ei sisalda ühtegi klauslit **then**
2: **return true**
3: **else if** F sisaldab tühja klauslit **then**
4: **return false**
5: **else**
6: Vali valemist F muutuja x.
7: Rakenda valemile F dekompositsiooni x järgi, saades valemid F_x ja $F_{\bar{x}}$.
8: Lihtsusta valemeid F_x ja $F_{\bar{x}}$.
9: **return** SAT(F_x)$\vee$SAT($F_{\bar{x}}$).
10: **end if**

Esitatud rekursiivsest pseudokoodist on näha, et ridadel (1-2) kontrollitakse eespool antud lõpptingimuse (*a) täidetust, kus valemi F kõik klauslid on tõeseks muutunud ja lihtsustamiste tulemusena ei ole valemisse F enam ühtegi klauslit jäänud. Sellisel juhul tagastab protseduur `true`, st. etteantud valem on kehtestatav.

Ridadel (3-4) kontrollitakse seevastu lõpptingimuse (*b) täidetust, st. inspekteeritavasse valemisse on sattunud väär klausel. Sellisel juhul ei ole enam mõtet midagi edasi arvutada, sest valemit ei õnnestu ühelgi viisil tõeseks muuta. Protseduur tagastab `false`, st. etteantud valem F on mittekehtestatav.

Kui protseduuri kutsungist ei väljutud ridade (1-4) kaudu, siis rakendadakse valemile Boole-Snannoni dekompositsiooni (6-7), vahetult mille järel toimub lihtsustamine (8). See annab kaks valemit F_x ja $F_{\bar{x}}$, milles on vähem muutujaid, kui esialgses valemis F. Real (9) rakendatakse mõlemale alamvalemile algoritmi SAT rekursiivselt.

Nagu eespool mitu korda mainitud, realiseeritakse sellised algoritmid tagurdusalgoritmidega. Tagurduse valikupunktiks on siin rida (9). Siin on näha, et valik tuleb teha kahe valemi F_x ja $F_{\bar{x}}$ vahel, millele algoritmi rekursiivselt edasi rakendada. Kui me oskaks kohe valida õige valemi, oleks sellise algoritmi keskmine keerukus lineaarne muutujate arvu suhtes. Paraku me ei oska õiget valemit valida ja selletõttu tuleb halvimal juhul mõlemad valemid läbi proovida, aga see muudab algoritmi tööaja juba eksponentsiaalseks. On näidatud, et leidub isegi terve valemite klass, mille jaoks see algoritm töötab alati eksponentsiaalses ajas [Tom07].

1.1.1 Ühikklauslite elimineerimine

Boole-Shannoni dekompositsioonil on erikuju. Olgu F konjuktiivsel normaalkujul olev valem. Kui F sisaldab sellist klauslit C, mis koosneb ainult ühest literaalist, st. $C = l$ ja $l \equiv x$, siis võib sooritada F lahutuse (F_x, $F_{\bar{x}}$) muutuja x järgi ja ignoreerida valemit $F_{\bar{x}}$, sest see on samaselt väär. Seda põhjendab asjaolu, et väärtustusel $v(x) = 0$, $C = l = 0$. See aga tähendab omakorda, et valem $F_{\bar{x}}$ sisaldab väära klauslit. Täpselt sarnane olukord kehtib ka juhul kui $l \equiv \neg x$, aga siis võime vaatluse alt välja jätta valemi F_x.

Ühikklauslite elimineerimine kui Boole-Shannoni lahutuse erikuju vastab originaalses *DPLL* algoritmis ühikklauslite elimineerimisreegli juhule (b). Selle reegli saame lisada otse algoritmi (1), saades juba märksa efektiivsema, kuigi siiski veel eksponentsiaalses ajas töötava, algoritmi:

Algoritm 2. DPLLSAT(F)

Sisend: konjuktiivsel normaalkujul valem F.
Väljund: tulemus, kas valem F on kehtestatav.

1: **if** F ei sisalda ühtegi klauslit **then**
2: **return true**
3: **else if** F sisaldab tühja klauslit **then**
4: **return false**
5: **else if** F sisaldab ühikklauslit $C_i = l$ **then**
6: Rakenda valemile F dekompositsiooni x järgi, saades valemid F_x ja $F_{\bar{x}}$.
7: **if** $l \equiv x$ **then**
8: Lihtsusta valemit F_x.
9: **return** DPLLSAT(F_x).
10: **else**
11: Lihtsusta valemit $F_{\bar{x}}$.
12: **return** DPLLSAT($F_{\bar{x}}$)
13: **end if**
14: **else**
15: Vali valemist F muutuja x.
16: Rakenda valemile F dekompositsiooni x järgi, saades valemid F_x ja $F_{\bar{x}}$.
17: Lihtsusta valemeid F_x ja $F_{\bar{x}}$.
18: **return** DPLLSAT(F_x)$\vee$DPLLSAT($F_{\bar{x}}$).
19: **end if**

1.1.2 Puhta literaali elimineerimine

Puhta literaali elimineerimine on lisaks ühikklausli elimineerimisele veel üks täiendav heuristiline reegel kehtestatavuse leidmiseks. Olgugi et see esines originaalses DPLL algoritmis, on temast hilisemates variantides loobutud eelkõige selle tõttu, et praktikas on osutunud tema kasutamine ebaefektiivse realisatsiooni tõttu vähetõhusaks.

Puhta literaali l elimineerimine kujub endast valemi $F = C_1 \wedge \cdots \wedge C_k$ klauslitest nende eemaldamist, kus l esineb. Sealjuures ei tohi üheski klauslis $C_1 \wedge \cdots \wedge C_k$ esineda l komplementaari $\bar{l}$. Reegli õigustuse võib leida allikast [Tom07]. Puhta literaali reeglile vastab Boole-Shannoni dekompositsiooni rakendamine valemile F literaalile l vastava muutuja x järgi. Siis ütleb reegel, et me võime ignoreerida valemit $F_{\bar{x}}$, kui $l \equiv x$ ja valemit F_x juhul $l \equiv -x$.

Paraku on puhta literaali eemaldamisel mittekasulik omadus valemi lahendite arvu suhtes. Nimelt kaotame selle kasutamisel osa lahenditest, sest võib juhtuda, et valem võib olla tõene ka juhul, kui puhas literaal l on väär, st. esialgne valem ja pärast teisendust saadud valem ei ole loogiliselt samaväärsed. Näitena sobib valem $F = x \vee y$, millest saaksime rangelt puhta literaali elimineerimist tarvitades lahendi $x = 1, y = 1\}$, aga sealjuures ignoreeriksime täielikult lahendeid $\{x = 0, y = 1\}$ ja $\{x = 1, y = 0\}$.

See halb omadus on põhjuseks, miks puhta literaali elimineerimise reegel on käesolevas töös vaadeldud *DPLL* algoritmi variandist välja jäetud. Teiseks põhjuseks on valemist puhta literaali otsimise arvutuslik keerukus valemi andmestruktuurist, mis on esitatud töö teises pooles, sest andmestruktuur on optimeeritud ühikliteraali leidmiseks ning valemi kiireks lihtsustamiseks vahetult pärast muutuja valikut.

1.2 Lahendite loendamine

Seni oleme töös vaadelnud ainult kehtestatavuse algoritmi. Selgub, et lahendite loendamiseks tuleb seda (2) ainult vähe muuta. Loendamine kasutab ära asjaolu, et ülalkirjeldatud algoritm üritab leida kehtestatavat väärtustust üksikute muutujate väärtustamise kaupa, kusjuures alati lõpetab protseduur ühes kahest olukorrast (*a, *b), mida on töös eespool juba kirjeldatud.

On ilmselt selge, et valemi lahendite arv (kujutades ette protseduuri SAT või DPLLSAT rakendamist valemile F ja vahetult olukorra (*a) või (*b) esinemist) on neil juhtudel järgmine:

(*a): Valemis F oli esialgu n muutujat, nendest on ära väärtustatud k tükki. See tähendab, et ülejäänud $t = n - k$ muutujale võib anda suvalise väärtustuse, ilma et peaks

kartma, et valem F vääraks muutuks. t muutujale saab kokku anda 2^t erinevat väärtustust, seega on F lahendite arv 2^{n-k}.

(*b): Kuna valem F on väär, siis on tema lahendite arv 0.

Vastavalt nendele juhtudele on vaja modifitseerida algoritmi SAT või DPLLSAT ainult niipalju, et anda sisendparameetrina kaasa valemi F esialgne lahendite arv n ja seni väärtustatud muutjate arv k. Protseduuri rekursiivses sammus (SAT, rida 9; DPLLSAT, rida 18) tuleb loomulikult igal sammul väärtustatud muutujate arvule juurde liita 1 . Saadav algoritm on esitatud järgnevalt:

Algoritm 3. DPLL_LOENDA(F, N, K)

Sisend: knk valem F, F muutjate arv N, väärtustatud muutujate arv K.
Väljund: valemi F lahendite arv S.

1: **if** F ei sisalda ühtegi klauslit **then**
2: **return** 2^{N-K}
3: **else if** F sisaldab tühja klauslit **then**
4: **return** 0
5: **else if** F sisaldab ühikklauslit $C_i = l$ **then**
6: Rakenda valemile F dekompositsiooni x järgi, saades valemid F_x ja $F_{\bar{x}}$.
7: **if** $l \equiv x$ **then**
8: Lihtsusta valemit F_x.
9: **return** DPLL_LOENDA(F_x).
10: **else**
11: Lihtsusta valemit $F_{\bar{x}}$.
12: **return** DPLL_LOENDA($F_{\bar{x}}$)
13: **end if**
14: **else**
15: Vali valemist F muutuja x.
16: Rakenda valemile F dekompositsiooni x järgi, saades valemid F_x ja $F_{\bar{x}}$.
17: Lihtsusta valemeid F_x ja $F_{\bar{x}}$.
18: **return** DPLL_LOENDA(F_x)+DPLL_LOENDA($F_{\bar{x}}$).
19: **end if**

Sarnaselt algoritmile DPLLSAT, saab antud protseduuri kirja panna tagurdusalgoritmina, mida on tehtud ka töö teises pooles realiseeritud programmis. Selleks piisab võtta

kasutusele globaalne muutuja, mille väärtust suurendada iga kord 2^{n-k} võrra (n - muutujate koguarv, k - väärtustatud muutujate arv), kui algoritm leiab kehtestava väärtustuse. Seejärel tuleb lasta algoritmil tagurdada kuni viimase rekursioonitasemeni, milles pole läbi vaadatud mõlemat valemit. (rida 18 algoritmis DPLL_LOENDA) ning pärast protseduuri rakendamise lõppu lugeda selle muutuja väärtus, mis annabki lahendite arvu.

Peatükk 2

Loendamisalgoritmi realisatsioon

Selles töö osas anname eespool kirjeldatud algoritmi realisatsiooni keeles Prolog. Prolog sai valitud tema mitmete heade omaduste tõttu, mida saab edukalt ära kasutada sellist tüüpi ülesannete lahendamisel. Prolog baseerub ise tagurdusalgoritmil ja see vähendab algoritmi kirjapanekul tööd oluliselt, sest programmeerija ise ei pea tegelema selle keerulise osaga, mida on tagurdamisel muutujate väärtuste efektiivne taastamine. Lisaks on Prolog tuntud kui sümbolmanipuleerimise keel, peale selle ei ole vaja Prologis programmeerijal muretseda korrektse mäluhalduse pärast, sest Prolog sisaldab automaatset mälukoristust. Need omadused teevad Prologist keele, mis on vägagi sobiv keeruliste algoritmide realiseerimiseks või prototüüpimiseks. Samuti on oluline mainida, et Prolog on populaarne keel tehisintellekti valdkonda kuuluvate ülesannete lahendamiseks. See muudab parajasti lihtsaks antud algoritmi integreerimise ja laiendamise teiste tehisintellekti probleemide jaoks.

Käesoleva töö praktilise osa täielikuks mõistmiseks on tarvis osata lugeda Prologi programme. Väga heaks sissejuhatavaks materjaliks (millest ka täiesti piisab töö mõistmiseks), on allikas [Tam03].

Realisatsiooni kirjeldust alustame programmi üldise struktuuri esitamisega, kusjuures ei ole isegi olulised kasutatavad andmestruktuurid, mille kirjelduse anname alles mõnevõrra hiljem.

Selles töös esitatud viis *DPLL* kehtestatavus- ja loendamisalgoritmi jaoks ei ole loomulikult ainuõige võimalus programmi realiseerimiseks. Samas on see lähenemine töö autori arvates sobiv Prolog-tüüpi keeles algoritmi implementeerimiseks. Imperatiivses programmeerimiskeeles, näiteks C keeles, on märksa levinum kasutada lähenemist, mille pseudokood on toodud artiklis [Si96].

2.1 Realisatsiooni ülesehitus

Algoritmi realiseerimisel sai koostatud lihtne tagurdusel põhinev programm. Kasutatud andmestruktuurid seletatakse põhjalikumalt lahti selleks pühendatud töö sektsioonis pärast algoritmi kirjeldust.

2.1.1 Muutujate kodeerimine

Programmis esitame lahendatava valemi muutujad täisarvudena 1,2,.... See on levinud viis, kuna võimaldab mugavat indekseerimist lineaarsetes andmestruktuurides, näiteks massiivides. Samuti saame kompaktselt ja elegantselt esitada literaale. Muutuja positiivse esinemise tähistamiseks tuleb kasutada lihtsalt muutujale vastavat täisarvu, negatiivse esinemise tarvis aga vastandarvu.

See kodeerimismeetod pakub triviaalsed lahendused operatsioonidele nagu literaali komplementaari leidmine jt. Olgugi et Prolog kui predikaatarvutust põhjana kasutav keel otseselt massiive andmestruktuuridena ei paku, leidub alternatiivne lähenemine termide inspekteerimise vahendite näol.

2.1.2 Põhiprotseduur

Eespool esitatud algoritm DPLLSAT on kirja pandud kahe protseduuri järjest rakendamisel saadava rekursiivse programmi abil:

```
1: dpll(L, F, S, Ct, D, Ys):-
2:          kontrolli(0, L, F, S, Ct, Ys),
3:          samm(0, F, S, Ct, D).
```

Selles protseduuris on muutujate kirjeldused järgmised (kõik muutujad on sisendparameetrid):

`L` - viimati tõeseks valitud literaal;

`F` - lahendatav valem;

`S` - hetkel kehtiv muutujate ja klauslite väärtustus;

`Ct` - hetkel tõeste klauslite arv;

`D` - seni sooritatud lahendussammude arv, võrdne seni väärtustatud muutujate arvuga;

`Ys` - ühikliteraalide vahepinu (sisaldab literaale, mis on programmi töö käigus ühikliteraalideks muutunud ja tuleks deskompositsiooni rakendades esmajärjekorras ära kasutada);

`O` - protseduuri `kontrolli/6` rakendamisel saadav väärtus, mille põhjal tehakse järgmine lahendussamm.

Protseduur `kontrolli/6` sisuliselt vaatab järgi, kas hetkel kehtiva väärtustuse piires on valem `F` juba tõeseks muutunud või mitte. Põhimõtteliselt on võimalikud kolm olukorda:

1. Valem `F` on tõeseks muutunud, st. kõik tema klauslid sisaldavad vähemalt ühte tõest literaali. Sellisel juhul on leitud kehtestav väärtustus ja programm lõpetab oma töö ära. `kontrolli/6` väljund on siis `O = sat`.

2. Valemis `F` leiduvad ühikklauslid ning sooritada tuleb dekompositsioon nende järgi. Sellele olukorrale vastab `kontrolli/6` väljund `O = yhik(Ls)`, kus `Ls` on list literaalidest, mis ühikklauslitena esinevad. `kontrolli/6` on koostatud niimoodi, et see list ei saa kunagi tühi olla. Kui protseduuri `dpll/6` kutsudes sisaldas loend `Ys` mõnd elementi, siis täitub alati `kontrolli/6` praegune tulemus ja `Ls` sisaldab vähemalt neid elemente, mida sisaldab `Ys`.

3. Valem `F` ei ole veel tõene, aga temas puuduvad hetkel ühikklauslid, mistõttu tuleb valida suvaline muutuja dekompositsiooni rakendamiseks. Siin on erijuhuks olukord, kus kõik muutujad on juba varem ära valitud ja antud sammus ei ole see enam võimalik. Programm tagurdab automaatselt viimase valikupunktini, milleks on eelmine rekursioonisamm, kus valiti küll muutuja, aga pole veel läbi proovitud tema mõlemat väärtustust. Kui selliseid valikupunkte ei ole, siis lõpetab programm töö negatiivse tulemusega. Olukorrale, kus järgmises sammus tuleks teha muutuja valik, vastab `kontrolli/6` väljund `O = vali`.

Järgnevalt esitame protseduuri `samm/5` ülalkirjeldatud juhtudele. On selge, et ei ole tähtis definitsioonide järjekorral, sest nad on üksteist välistavad `samm/5` esimese sisendargumendi järgi (mis samal ajal on ka `kontrolli/6` väljundiks).

Juht 1:

Kuivõrd on leitud kehtestav väärtustus, siis ei pea enam midagi edasi arvutama ja samal ajal ei ole tarvis hoolida muutujate `F, S, Ct, D` väärtustest, mida võime ignoreerida. Seega sobib järgmine definitsioon:

```
1: samm(sat, _, _, _, _).
```

Juht 2:

Valemis esinesid ühikklauslid, võtame esimese vastava literaali tõeseks ja rakendame väärtustusprotseduuri `propageeri/5`, mille väljundina saame tekkinud väärade klauslite arvu `Cf` ja tõeste klauslite arvu `Ct1`. Loomulikult on tarvis kontrollida ega vääri klausleid ei saadud, mida tehakse real (3). Kui saadi, lõpetatakse selles harus arvutamine ära ja tagurdatakse, muidu arvutatakse tõeste klauslite koguarv `Ct2` ja rekursioonisügavus `D1` ning sooritatakse päring `dpll/6`.

```
1: samm(yhik([Y|Ys]), F, S, Ct, D):-
2:          propageeri(Y, F, S, Cf, Ct1),
3:          (Cf > 0 ->
4:                  fail
5:                  ;
6:                  Ct2 is Ct + Ct1,
7:                  D1 is D + 1,
8:                  dpll(Y, F, S, Ct2, D1, Ys)
9:          ).
```

Juht 3:

Viimases võimalikus olukorras toimetetakse sarnaselt eelmise juhuga. Erinev on vaid see, et siin tuleb muutuja valida eraldi päringuga `vali_muutuja(F, S, L)`. See protseduur teostab valiku valemi `F` ja hetkel kehtiva väärtustuse `S` põhjal. Tema väljundiks on literaal `L`, mis tähistab, kas vastav muutuja tuleb valida tõeseks või vääraks. Ülejäänud osas toimitakse samamoodi kui juhul (2), ainult et ühikliteraalide vahepinu võetakse võrdseks tühja listiga.

```
1: samm(vali, F, S, Ct, D):-
2:          vali_muutuja(F, S, L),
3:          propageeri(L, F, S, Cf, Ct1),
4:          (Cf > 0 ->
5:                  fail
6:                  ;
7:                  Ct2 is Ct + Ct1,
8:                  D1 is D + 1,
9:                  dpll(L, F, S, Ct2, D1, [])
10:         ).
```

Kehtestatavusprogramm koos silumismooduli kutsetega, mis selguse huvides jäeti välja siin esitatud kirjeldusest, on realiseeritud programmimoodulis `dpll` ja asub failis *dpll.pl*.

2.2 Lahendite loendamine

Lahendite loendamise jaoks tuleb programmi täiustada tõeste väärtustuste loenduriga (lisa-sisendparameeter predikaatidele `dpll` ja `samm`) ja asendada sammu sooritamise protseduuri `samm` definitsioon ülalkirjeldatud juhule (1) järgmisega:

```
1: samm(sat, F, _, _, D, Lo):-
2:          muutujate_arv(F, N),
3:          nb_getval(Lo, C),
4:          C1 is C + integer(2^(N-D)),
5:          nb_setval(Lo, C1),
6:          fail.
```

Protseduuris tähistab `Lo` loendurit, millest küsitakse kõigepealt seni saadud lahendite arv `C` (rida 3), millele liidetakse juurde hetkel leitud lahendite arv (rida 4). Saadud tulemus `C1` salvestatakse tagasi loendurisse `Lo` (rida 5). Protseduuri lõpus teostatakse pseudopäring `fail` (rida 6), mis programselt kutsub esile tagurdamise. Loenduri kui globaalse muutuja kasutamiseks tarvitatakse ekstraloogilisi predikaate `nb_getval/2` ning `nb_setval/2`, mille tulemus ei sõltu üldisest Prologi virtuaalmasina käitumisest tagurdusoperatsiooni käigus.

2.3 Põhiandmestruktuurid

Eespool esitatud programmi selgitusest on välja jäetud valemi andmestruktuuri kirjeldus (`F` struktuur protseduurides `dpll` ja `samm`), samuti hetkel lahendatava valemi lausearvutusmuutujate väärtuse talletamine (`S` struktuur). Programmi üldisel kirjeldamisel on nad ebaolulised, kuid siiski vajalikud kasutatud abipredikaatide `kontrolli/6`, `propageeri/5` ja `muutujate_arv/2` implementeerimisel.

2.3.1 Valem

Valemit käsitletakse programmis kui staatilist struktuuri. Eespool kirjeldatud lihtsustusteisendused on lahendatud klauslite vaheväärtuste hoidmisega ning efektiivse muutujate-klauslite indekseerimispõhimõttega.

Valemit säilitatakse termis kujul `valem(n,m,vc(C1,...,Cn),cv(V1,...,Vm))`, kus `n` on valemi muutujate arv, `m` on valemi klauslite arv ning termid `vc` ja `cv` on kasutusel järgmisel otstarbel:

- Termis `vc(C1,...,Cn)` hoitakse muutuja indeksi `i` järgi klausleid, kuhu antud muutuja kuulub. Iga `Ci` on omakorda term `c(Cip, Cin)`, kus `Cip` tähistab nende klauslite listi, kus muutuja indeksiga `i` positiivselt esineb ja `Cin` nende klauslite listi, kus see muutuja negatiivselt esineb.
- Termis `cv(V1,...,Vm)` seevastu hoitakse klausli indeksi `j` järgi muutujaid, mis sellesse klauslisse kuuluvad. Siin tuleb sammuti vahet teha positiivse ja negatiivse esinemise vahel. Selletõttu oleks sobiv `Vj` struktuur `v(Vjp, Vjn)`, kus list `Vjp`(`Vjn`) tähistab muutujate indekseid, mis esinevad klauslis indeksiga `j` positiivselt(negatiivselt).

Näide valemi teisendamisest sisekujule:

```
?- teisenda([[-1, 2, -3], [-2, -4], [4]], T).
T = valem(
    4,
    3,
    vc(c([], [1]), c([1], [2]), c([], [1]), c([3], [2])),
    cv(v([2], [1, 3]), v([], [2, 4]), v([4], []))
)
```

Selline andmestruktuur võimaldab leida konstantse ajaga kõik klauslid, mis sisaldavad etteantud literaali, samuti mingi kindla klausli (teades klausli indeksit) kõik literaalid, kuna termi argumendi inspekteerimine protseduuri `arg/3` abil on konstantse keerukusega [Swi].

2.3.2 Väärtustus

Programmis hetkel kehtivat väärtustust hoiame struktuuris
`olek(v(X1,...,Xn), c(Y1,...,Ym))`, kus `X1,...,Xn` on vastavate muutujate väärtused ja `Y1,...,Ym` klauslite väärtused. Tõest väärtust tähistame täisarvuga 1 ja väära täisarvuga 0. Kui muutujal väärtust ei ole, siis on vastav koht termis lihtsalt tühi. Kuna korraga saab ühel kohal asuda ainult üks väärtus, siis garanteerib selline andmestruktuur muutujate oleku ühesuse. Klauslite väärtustamist kasutatakse ära programmi efektiivsuse tõstmiseks, sest siis ei ole tarvis pidevalt uuesti välja arvutada klauslite väärtuseid.

Näide tühja väärtustuse koostamisest valemi jaoks, milles on 4 muutujat ja 3 klauslit:

```
?- tyhi_olek(4, 3, S).
S = olek(v(_G256, _G259, _G262, _G265), v(_G276, _G279, _G282))
```

Kasutades algoritmis ära teadmist viimati väärtustatud muutuja kohta (`L` - viimati tõeseks valitud literaal protseduurides `kontrolli/6` ja `propageeri/5`), on selge, et saab praktiliselt konstantses ajas teha järgmisi operatsioone:

1. Ühikklauslite leidmine - vaadata läbi ainult need klauslid, mis sisaldasid literaali `L` komplementaari `-L`, sest ainult sellistes klauslites vähenes esinevate muutujate arv.
2. Tõeste klauslite leidmine - tõeseks märkida kõik klauslid, milles esines `L`.
3. Väärade klauslite leidmine - vääraks saavad muutuda ainult klauslid, mis sisaldasid `L` komplementaari `-L`.

Peatükk 3

Programmi kasutamine

Programmi kasutamiseks on vajalik arvutisse paigaldatud Prologi interpretaator. Programm on küll kirjutatud Swi-Prologi realisatsioonile, kuid on lihtsalt porditav ka teistele populaarsetele Prologi implementatsioonidele.

3.1 Põhiprotseduurid

3.1.1 Lahendite loendamine

Loendamiseks saab kasutada protseduuri `dpll_loenda/2`, mis asub failis *dpll_loenda.pl*, ning mis võtab sisendargumendiks konjuktiivsel normaalkujul listiesituses valemi, kus muutujad on kodeeritud täisarvudena. Lahendite arv antakse väljundmuutujas, mis on protseduuri teiseks argumendiks.

Näide kasutamisest:

```
?- dpll_loenda([[-1, 2, -3], [-2, -4], [4]], C).
C = 3
```

3.1.2 Kehtestava väärtustuse leidmine

Valemi kehtestatavuse leidmise protseduur asub failis *dpll.pl*. Põhiprotseduuriks on `dpll/2`, mis võtab sisendiks valemi samal kujul nagu loendamiseks kasutatav protseduur. Väljundiks antakse muutujate väärtustuse termi muutujate osa (andmestruktuuri kirjeldus on esitatud eespool).

Kui mingi muutuja väärtustus pole vajalik kehtestatavuse jaoks, siis esineb termis sellel kohal Prologi muutuja, st. muutuja väärtust ei kitsendata.

Tagurdamisel genereeritakse automaatselt järgmine väärtustus. Kui valem ei olnud kehtestatav, siis protseduuri kutse lõpetab negatiivse tulemusega.

Näited päringutest:

```
?- dpll([[-1, 2, -3], [-2, -4], [4]], C).
C = v(1, 0, 0, 1) ;
C = v(0, 0, 1, 1) ;
C = v(0, 0, 0, 1) ;

?- dpll([[-1, 2, -3], [-2, -4]], C).
C = v(1, 1, _G778, 0)

?- dpll([[-1], [1]], C).
No
```

Teine võimalus on kasutada integratsioonimoodulit `dpllp/2` (failis *dpllp.pl*), mis võimaldab muutujatena tarvitada Prologi muutujaid (`dpllk/2`). Sellisel viisil saab siduda lahendaja teiste Prologi programmidega. Lisaks sisaldab moodul võimalust kasutada mittekonjuktiivsel normaalkujul olevaid valemeid (`dpllp/2`). Kasutatavad operaatorid on prioriteedi kasvamise järjekorras: `<=>,=>,v,&,-`.

Näited päringutest:

```
?- dpllk([[-A, B, -C], [-B, -D], [D]]).
A = 1,
B = 0,
C = 0,
D = 1

?- dpllp((-A v B v -C) & (B => -D) & D).

A = 1,
B = 0,
C = 0,
D = 1
```

3.2 cnf failide sisselugemine

Suuremate valemite (rohkem kui kümned muutujad) hoidmiseks on populaarsed `cnf` formaadis failid. `cnf` formaadis faile saab sisse lugeda päringuga `loe_cnf`, mille esimeseks argumendiks on loetava faili nimi ning teiseks argumendiks saadav valem.

Näide kasutamisest:

```
?- loe_cnf('../testid/ais/ais6.cnf', F), dpll_loenda(F, C).
F = [[1, 2, 3, 4, 5, 6], [-1, -2], [-1, -3],
  [-1, -4], [-1, -5], [-1, -6], [-2, -3], [-2|...], [...|...]|...],
C = 24
```

3.3 Silumisvahend

Programmi koostamisel oli oluliseks abivahendiks silumismoodul `silur`. Silumismoodulit toetavad kõik eespool esitatud lahendus- ja loendamisprotseduurid. Kui silumine on sisse lülitatud (seda saab teha päringuga `silur:sisse.`), siis kirjutab programm kasutajale väljundisse infot teostatavate tegevuste kohta lahendussammude kaupa. Vaikimisi on silumisvahend välja lülitatud.

Näide siluri väljundist:

```
?- dpll([[-1, 2, -3], [-2, -4], [4]], C).
0 - Valem: [[2, -1, -3], [-2, -4], [4]]
0 > Tõeseks valitud ühikliteraal 4
|:
1 - Valem: [[2, -1, -3], [-2]]
1 > Tõeseks valitud ühikliteraal -2
|:
2 - Valem: [[-1, -3]]
2 > Tõeseks valitud literaal 1
|:
3 - Valem: [[-3]]
3 > Tõeseks valitud ühikliteraal -3
|:
4 - Valem: []
C = v(1, 0, 0, 1)
Yes
```

Kokkuvõte

Käesolevas bakalaureusetöös uurisime lausearvutusvalemi kehtestatavate lahendite loendamise probleemi. Kuna antud probleem on tihedalt seotud kehtestatavusprobleemiga ja kuivõrd töös vaadeldud algoritm *DPLL* on aluseks nii loendamise kui kehtestatavuse lahendamiseks, oli mõistlik neid probleeme koos vaadelda.

Töös esimeses osas esitasime *DPLL* algoritmi üldisel kujul, nagu teda kasutatakse tänapäeval levinud kõrge efektiivsusega kehtestatavusalgoritmides. Natuke vaatlesime ka algoritmi originaalkuju ning mehaanilist teoreemide tõestamist - originaalülesannet, mille lahendamiseks *DPLL* algoritm mõeldud oli.

Töö teises osas andsime algoritmi realisatsiooni tehisintellekti programmeerimiskeeles Prolog, mis tänu sisseehitatud tagurdusalgoritmile ja hea sobivuse poolest sümbolarvutuse ülesannete jaoks, sobis suurepäraselt käesoleva probleemi lahendusalgoritmi implementeerimiseks.

Realiseeritud algoritm on täielik ning seda saab kasutada ka samal otstarbel, milleks originaalalgoritmi, s.o. tõestusprotseduurides. Implementatsioon on hõlpsasti integreeritav teiste Prologi programmidega, ning tänu ülevaatlikule koodile ning realisatsiooniga kaasa tulevale silumisvahendile, kasutatav ka hariduslikel eesmärkidel *DPLL*-tüüpi algoritmide tutvustamisel.

Paraku ei võimaldanud töö piiratud maht esitada *DPLL* algoritmi mitmeid edasiarendusi, mis tõstavad lahendusprotseduuri efektiivsust veelgi. Vaatluse alt jäi välja konfliktide analüüs ja muutuja valiku heuristika, mille abil saab algoritmi häälestada teatud valdkonnast (näiteks riistvara verifitseerimisega seotud ülesanded), kus valemid on teatud struktuuriga pärit valemite kiiremaks lahendamiseks, samuti mitmed võtted lahendite ligikaudseks loendamiseks. Selles valdkonnas on viimasel ajal toimunud eriti suur areng. Töös pole kajastatud ka olulisi lahendite loendamise rakendusi, näiteks kaalutud loendamist, mis vastab tuletusele Bayes'i võrgus, samuti Dedekindi arvude leidmist.

Using DPLL procedure to count models

Bachelor thesis (6 ECTS credits)

Raivo Laanemets

Abstract

In this thesis we study the problem of model counting of boolean formulas. The model counting problem is closely related to boolean satisfiability problem and it turns out that *DPLL*-like backtracking satisfiability algorithms are also suitable for the counting problem. We look at the ideas behind *DPLL* algorithm and implement it in Prolog programming language. It appears that Prolog is suitable for solving these kind of problems because of built-in backtracking and general suitability for symbol manipulations. The developed implementation can also be integrated with other Prolog programs.

Kirjandus

[Bir99] E.Birnbaum, E.L.Lozinskii, *The Good Old Davis-Putnam Procedure Helps Counting Models*, Journal of Artifical Intelligence Research 10, 1999, p. 457-477.

[Co71] S.Cook, *The complexity of theorem-proving procedures*, Proceedings of the 3rd ACM Symposium on Theory of Computing, p. 151-158, ACM Press May 1971.

[Dav60] M.Davis, H.Putnam, *A Computing Procedure for Quantification Theory*, J. Assoc. Comput. Mach., 7:201-215, 1960.

[Dav62] M.Davis, G.Logemann, D.Loveland, *A Machine Program for Theorem-Proving*, Communications of the ACM, 4:394-397, 1962.

[Ro96] D.Roth, *On the hardness of approximate reasoning*, Artifical Intelligence, 82(1/2):273-302, 1996.

[Schö94] U.Schöning, *Logic for Computer Scientists*, Progress in Computer Science and Applied Logic, Birkhäuser 1994.

[Si96] J.P.M.Silva, K.A.Sakallah, *GRASP - A New Search Algorithm for Satisfiability*, Proceedings of the International Conference on Computer-Aided Design, November 1996.

[Swi] J.Wielemaker, *SWI-Prolog 5.6.47 Reference Manual*, University of Amsterdam 2007.

[Tam03] T.Tamme, *Loogilise programmeerimise meetod*, Tartu, 2003.

[Tom07] M.Tombak, *Keerukusteooria*, Tartu 2007.

Lisa 1: programmi dokumentatsioon

Programmi moodulite dokumentatsiooni koostamiseks on kasutatud Prologi programmeerijate hulgas laialt levinud stiili, kus predikaatide argumentide ülesmärkimisel tähistatakse prefikssümboliga `'+'` sisendargumente, sümboliga `'-'` väljundargumente ning sümboliga `'?'` argumente, mis võivad olla nii sisendiks kui ka väljundiks. Samuti on üles märgitud ka argumentide tüübid. Selleks on lisatud argumendi nimele sobiv tüübi nimi kujul `:tüübinimi`. Tüübinimed on kasutusel ainult dokumentatsiooni lugemise lihtsustamiseks ning nad ei oma mingit formaalset tähendust. Ära on toodud ainult moodulitest välja eksporditud predikaatide dokumentatsioon.

1.4 Moodul "dpllp"

Moodul DPLL algoritmi kasutamiseks läbi Prolog muutujate.

- `dpllp(+`*F*`)` - Kehtestatavuse lahendaja, mille sisendiks on konjuktiivsel normaalkujul listiesituses antud valem. Valemi muutujad, mis on vajalikud valemi tõesuseks, väärtustatakse väärtusega 1 (tõene) või 0 (väär). Näide kasutamisest:

```
?- dpllp((A v B) => C).
C = 1 ;
A = 0,
B = 0,
C = 0
?-
```

1.5 Moodul "dpll"

DPLL kehtestatavusalgoritmi peamoodul.

- `dpll(+`*F*`:valem, -`*V*`)` - DPLL algoritmi realisatsioon. Valem *F* on konjuktiivsel normaalkujul valem, mis on antud listiesituses ja mille muutujad on kodeeritud täisarvudena. Väljund *V* on muutujate väärtustus, mis on antud andmestruktuuriga `olek`. Näide kasutamisest:

```
?- dpll([[1, 2, 3], [-1]], V1).
V1 = v(0, 1, _G657)
Yes
```

1.6 Moodul "abi"

Üldiste abimeetodite moodul.

- `algseadista_loendur(-`*Lo*`)` - Uue loenduri *Lo* algseadistamine väärtusele 0.
- `kl_vaartuseta_muutujad(+`*F*`:valem, ?`*Ci*`:int, +`*S*`:olek, -`*Ps*`:list, -`*Ns*`:list)` - Valemist *F* klausli *Ci* väärtustamata positiivsena esinevate muutujate Ps ja negatiivsena esinevate muutujate *Ns* leidmine.
- `m_filtreeri_vaartuseta(+`*Ms*`:list, +`*S*`:olek, -`*Ms1*`:list)` - Muutujate listist *Ms* väärtustamata muutujate väljafiltreerimine listi *Ms1*.
- `algvaartustus(+`*Fin*`:list, -`*F*`:valem, -`*S*`:olek, -`*V*`:term)` - DPLL algoritmides algandmestruktuuride koostamine. *Fin* - sisendvalem, *F* - valem sisekujul, *S* - tühi väärtustus, *V* - väärtustuse muutujate osa.
- `indekslist(+`*Xs*`:list, -`*Ps*`:list)` - Listi *Xs* konverteerimine listiks Ps, mille elementideks on paarid *I*-*E*, kus *I* on elemendi täisarvuline indeks ja *E* on element ise. Indeksid algavad väärtusega 1 ja väljendavad elemendi esinemispositsiooni listi algusosast alates.

1.7 Moodul "vaartustus"

Väärtustuse lisamiseks/kontrollimiseks kasutatvad protseduurid.

- `tyhi_vaartustus(+`*F*`:valem, -`*V*`:olek)` - Valemi *F* muutujate jaoks tühja väärtustuse koostamine.

- `muutuja_vaartuseta(+`*V*`:olek, +`*I*`:int)` - Kontrollimine, kas etteantud indeksiga muutuja on vaartustuses *V* väärtustatud hetkel või mitte.
- `klausel_vaartuseta(+`*V*`:olek, +`*I*`:int)` - Kontrollimine, kas etteantud indeksiga klausel on vaartustuses *V* väärtustatud hetkel või mitte.
- `vaartusta_muutuja(+`*V*`:olek, +`*I*`:int, +`*Val*`)` - Etteantud indeksiga *I* muutuja väärtustamine väärtusega *Val* väärtustuses *V*.
- `vaartusta_klausel(+`*V*`:olek, +`*I*`:int, +`*Val*`)` - Klausli indeksiga *I* väärtustamine väärtusega *Val* väärtustuses *V*.

1.8 Moodul "valem"

Valemiga tegevuste sooritamise moodul.

- `klausli_muutujad(+`*F*`:valem, +`*I*`:int, -`*Ps*`:list, -`*Ns*`:list)` - Leia valemist *F* klauslis indeksiga *I* positiivse literaalina esinevate muutujate indeksid Ps ja negatiivsena esinevate muutujate indeksid *Ns*.
- `klausli_pos_muutujad(+`*F*`:valem, +`*I*`:int, -`*Ps*`:list)` - Leia valemist *F* klauslis indeksiga *I* positiivse literaalina esinevate muutujate indeksid *Ps*.
- `klausli_neg_muutujad(+`*F*`:valem, +`*I*`:int, -`*Ps*`:list)` - Leia valemist *F* klauslis indeksiga *I* negatiivse literaalina esinevate muutujate indeksid Ns.
- `muutuja_klauslid(+`*F*`:valem, +`*I*`:int, -`*Ps*`:list, -`*Ns*`:list)` - Leia valemist *F* klauslite indeksid Ps, kus muutuja indeksiga *I* esineb positiivselt ja klauslite indeksid Ns, kus muutuja indeksiga *I* esineb negatiivselt.
- `pos_muutuja_klauslid(+`*F*`:valem, +`*I*`:int, -`*Ps*`:list)` - Leia valemist *F* klauslite indeksid Ps, kus muutuja indeksiga *I* esineb positiivselt.
- `neg_muutuja_klauslid(+`*F*`:valem, +`*I*`:int, -`*Ps*`:list)` - Leia valemist *F* klauslite indeksid Ns, kus muutuja indeksiga *I* esineb negatiivselt.
- `teisenda(+`*F1*`:valem, -`*F2*`:valem)` - Konjuktiivsel normaalkujul listiesituses valemi *F1* teisendamine sisekujule *F2*. Näide kasutamisest:

```
?- teisenda([[-1, 2, -3], [-2, -4], [4]], T).
T = valem(
```

```
4,
3,
vc(c([], [1]), c([1], [2]), c([], [1]), c([3], [2])),
cv(v([2], [1, 3]), v([], [2, 4]), v([4], []))
)
```

1.9 Moodul "kontrolli"

Valemi kontrollimine vahetult pärast muutuja väärtustamist.

- `kontrolli(-`O`:otsus, +`L`:int, +`F`:valem, +`V`:olek, +`Ct`:int, +`Ys`:list)` - Kontrolli hetkel väärtustust viimati tehtud lahendussammu järgi. O - väljund, L - viimati tõeseks valitud literaal, F - valem, mida lahendatakse, V - hetkel kehtiv väärtustus, Ct - seni tõeseks saadud klauslite arv, Ys - ühikliteraalide pinu. Väljund O võib olla üks järgmistest: `sat` - valem on tõene; `yhik(Ls)` - võta järgmistes sammudes tõeseks ühikliteraalid `Ls`; `vali` - vali järgmises sammus suvaline muutuja.

1.10 Moodul "propageeri"

Protseduur muutuja väärtustuse valimise propageerimiseks.

- `propageeri(+`L`:int, +`F`:valem, +`V`:olek, -`Cf`:int, -`Ct`:int)` - Etteantud tõese literaali L väärtuse propageerimine (valemi F vastava muutuja/klauslite väärtustamine). Väljundparameetrid: Cf - saadud väärade klauslite arv; Ct - saadud tõeste klauslite arv.

1.11 Moodul "silur"

Programmi silumist abistavad protseduurid.

- `sisse` - Siluri väljundi näitamise lubamine. Kui siluri väljundi näitamine on lubatud, kirjutatakse standardväljundisse infot algoritmi sammude kohta.
- `valja` - Siluri väljundi näitamise keelamine. Kui siluri väljundi näitamine on lubatud, kirjutatakse standardväljundisse infot algoritmi sammude kohta.

1.12 Moodul "knk"

Prologi muutujatega antud valemi teisendamine listiesituses konjuktiivsele normaalkujule.

- `knk(+`F`:valem, -`G`:valem)` - Lausearvutusvalemi F teisendamine konjuktiivsele normaalkujule G. Protseduur aksepteerib loogiliste operaatoritena `-` (eitus), `v` (disjunktsioon) `&` (konjuktsioon), `=>` (implikatsioon), `<=>` (ekvivalents). Operaatorite prioriteedid kasvavalt: `-`, `&`, `v`, `=>`, `<=>`. Protseduur ei eemalda valemist tautoloogiaid. Näide kasutamisest:

  ```
  ?- knk((A v B) => C, W).
  W = (-A v C)& (-B v C)
  ```

- `konj_list(+`F`:valem, -`L`:list)` - Valemi F teisendamine konjuktiivsele normaalkujule. Protseduur aksepteerib neidsamu operaatoreid, mida `knk/2`.