Dokumentacija implementacije Nelson-Oppen-ovog algoritma za kongruentno zatvorenje

Projektni rad u okviru kursa Automatsko rezonovanje
Matematički fakultet

Ana Stanković, Milica Đurić
anastankovic167@gmail.com, mdjuric55@gmail.com

19. jul 2017.

Sadržaj

1 Uvod

U okviru kursa Automatsko rezonovanje bavili smo se prvo problemom ispitivanja zadovoljivosti u iskaznoj logici, a zatim problemom ispitivanja zadovoljivosti u logici prvog reda. Simbol '=' se može u logici prvog reda interpretirati na razne načine, ali nama najvažnije interpretacije jesu one u kojima se ovaj simbol posmatra baš kao relacija jednakosti. Za sve modele koji zadovoljavaju prethodno pomenutu interpretaciju simbola '=' kažemo da su *normalni modeli.* Oni zadovoljavaju aksiome jednakosti: *refleksivnost, simetričnost, tranzitivnost, kao i kongruentnost.* Sintaksno-deduktivni sistem za jednakost opisuju Birkohofova pravila. Neka jednakost važi samo ukoliko ona može da se izvede iz početnih pretpostavki primenom Birkohofovih pravila. Ukoliko su sve jednakosti bazne, odnosno nemaju slobodnih promenljivih, onda nije potrebno koristiti pravilo instancijacije iz Birkohofovih pravila, tako da imamo fragment logike prvog reda koji je odlučiv.

Za ovakav pristup mogu se izgraditi procedure specijalizovane za rezonovanje u prisustvu jednakosti. Jedna od takvih jeste procedura odlučivanja Nelson-Oppen koja će biti detaljnije opisana u nastavku.

2 Nelson-Oppen algoritam za kongruentno zatvorenje

Ceo Nelson-Oppen algoritam radi na tome da dobijemo kongruentno zatvorenje. Počinjemo od prazne relacije i dodajemo jednu po jednu jednakost proširujući time relaciju kongruencije. Svaka od kongruencija se predstavlja klasama ekvivalencije.

Kako pronaći efikasne strukture podataka koje će brzo da otkriju da li su dva terma u relaciji? Možemo za svaku klasu ekvivalencije da čuvamo listu, a za svaki element da čuvamo indeks liste u kojoj se element nalazi. Međutim, kada je potrebno da se dve klase ekvivalencije spoje, onda je potrebno promeniti indeks liste za svaki od elemenata koji su promenili klasu ekvivalencije. Ovo zvuči, i jeste veoma sporo. Zato se u algoritmu koristi struktura podataka **union-find**, što znači da umesto liste koristimo obrnuto drvo (pokazivači idu od listova ka korenu). Da bi postupak bio što efikasniji, potrebno je da drvo bude što pliće. Ukoliko imamo dva terma za koja važi da imaju isti koren, onda to znači da ta dva terma jesu u istoj klasi ekvivalencije. Postupak objedinjavanja dve klase ekvivalencije je efikasan - samo treba preusmeriti pokazivač jednog korena na drugi, pri čemu treba voditi računa da stablo ostane što pliće.

Predstavimo sa $E = s1 = t1, ..., sn = tn$ skup svih baznih jednakosti za koje pravimo relaciju kongruencije.

Sa T možemo da označimo skup svih baznih termova iz E, njihovih podtermova i podtermova od podtermova, itd. (što znači da je skup zatvoren za podtermove), a skup T može da sadrži i još neke termove (termove za koje ćemo da ispitujemo da li se nalaze u istoj klasi ekvivalencije).

Kada smo pokupili sve termove, za ovaj algoritam je potrebno napraviti za svaki term t skup $use(t)$ koji sadrži sve termove iz T u kojima se term t nalazi kao podterm.

Neka funkcija $find(t)$ vraća koren drveta u kome se nalazi term t, odnosno kanonskog predstavnika klase ekvivalencije u kojoj se nalazi t.

Funkcija $union(s,t)$ spaja dva drveta, odnosno klase ekvivalencije, u kojima se nalaze termovi s i t. Pri spajanju voditi računa o dubini drveta, odnosno, preusmeravati pokazivač od plićeg ka dubljem drvetu kako ne bismo dobili dubinu koja je veća od dubljeg drveta.

Funkcija $cong(s,t)$ vraća $true$ ako ako s i t imaju iste funkcijske simbole, a za njihove odgovarajuće podtermove važi da pripadaju istoj klasi ekvivalencije. Što znači da je s oblika $f(s_1,...,s_n)$ i t oblika $f(t_1,...,t_n)$ pri čemu je $find(s_i) = find(t_i)$.

Algoritam se može predstaviti sledećim pseudokodom.

```
function cc(E,T)
begin
   // za svaki element proglasi da se nalazi u svojoj klasi ekv.
    foreach t from T
       find(t):=t
    //prolazimo kroz sve jednakosti i spajamo klase ekv.
    foreach s = t from E
       if(find(s) != find(t))
           merge(s, t)
end
```

```
function merge(s,k)
begin
   //pronadji termove u kojima se koriste termovi s i k i svi termovi
   //koji se nalaze u njihovim klasama ekv.
   S = U{use(u)| find(u)=find(s)}
   K = U{use(u)| find(u)=find(k)}
   union(s,k)
   foreach s' from S and k' from K
      if(find(s')!=find(k') and cong(s',k'))
         merge(s',k')
end
```

U sledećem odeljku će biti reči o našoj konkretnoj implementaciji ovog algoritma.

3 Detalji implementacije

Cela implementacija pomenutog algoritma obuhvata šest fajlova: *lexer.lpp, parser.ypp, unionFind.hpp, unionFind.cpp, fol.hpp, main.cpp*. Proći ćemo kroz svaki od njih i objasniti najvažnije funkcionalnosti koje oni pružaju.

3.1 fol.hpp

Ovo je zaglavlje potrebno za implementaciju same logike prvog reda.

BaseTerm je apstraktna klasa koja se koristi za predstavljanje termova. Jedini termovi koji su podržani u okviru naše implementacije jesu funkcijski termovi (jer radimo samo sa baznim jednakostima i baznim formulama) predstavljeni klasom *FunctionTerm*. Svaki funkcijski term se sastoji iz funkcijskom simbola i iz niza njegovih podtermova. Funkcija *equalTo* je funkcija koja pro-

verava da li su dva terma jednaka. Dva terma su jednaka ukoliko su im isti funkcijski simboli i ukoliko su im isti odgovarajući podtermovi.

BaseFormula je apstraktna klasa koja se koristi za predstavljanje formula. Jedine formule potrebne za našu implementaciju jesu atomi, i to jednakosti predstavljena klasom *Equality*. Svaka jednakost se sastoji iz predikastskog simbola '=' i iz dva podterma, jednog sa leve strane jednakosti i jednog sa desne strane jednakosti.

Koristimo pokazivače ($std :: shared\_ptr$ iz $< memory >$) na klasu $BaseTerm$ i $BaseFormula$

```
typedef shared_ptr<BaseTerm> Term;
typedef shared_ptr<BaseFormula> Formula;
```

Redefinisani su operatori << za termove i formule. Kako su nam u daljoj implementaciji potrebni skupovi termova i mapa čiji su ključevi termovi, bilo je potrebno redefinisati i operator $<$. Za dva terma s i t kažemo da je s manji od t ukoliko je funkcijski simbol leksikografski manji od funkcijskog simbola terma t. Ukoliko su funkcijski simboli isti, prvi term je manji od drugog ako ima manji broj podtermova. Ako je broj podtermova jednak, onda upoređujemo odgovarajuće podtermove termova s i t. Prvi term je manji ako naiđemo da mu je prvi podterm (koji nije jednak odgovarajućem podtermu drugog terma) manji od od odgovarajućeg podterma drugog terma. Inače, term s nije manji od terma t.

Ovde su definisani i skup termova $TermSet$ i mapa za predstavljanje use skupova $UseMap$.

```
//C je komparator koji koristi operator <
typedef set<Term, C> TermSet;
typedef map<Term, TermSet, C> UseMap;
```

3.2 lexer.lpp

U okviru ovog fajla se nalazi lekser potreban za čitanje sa ulaza. Funkcijski simboli jesu stringovi koji počinju malim slovom. Jedini predikatski simbol koji može da se pročita jeste '=' koje nam je potrebno za čitanje skupa jednakosti koji predstavljaju ulaz u algoritam. Ostali dozvoljeni simboli jesu: ( ) { } ; , i razmaci.

3.3 parser.ypp

U okviru ovog fajla nalazi se parser potreban za čitanje sa ulaza.

Ulaz je formata koji izgleda ovako '{' setFormulas '}' '{' formula '}' ';' , odnosno $\{jednakost_1, jednakost_2, ...jednakost_n\}\{jednakost_{n+1}\};$. Prva vitičasta zagrada jeste skup E opisan u odeljku 2. Druga vitičasta zagrada jeste jednakost za koju želimo da dokažemo da se može izvesti iz skupa E. Termovi iz ove vitičaste zagrade takođe ulaze u skup T. Pročitan ulaz se smešta u

```
//niz jednakosti potrebnih za algoritam
vector<Formula>* parsed_set_of_formulas;
//jednakost koja zeli da se dokaze da vazi
```

```
Formula parsed_formula;
```

3.4 unionFind.hpp

Zaglavlje potrebno za predstavljanje strukture podataka union-find. Svaki element ove strukture se predstavlja narednom strukturom

```
struct Node
{
   Term t; //vrednost cvora
   Node *parent; //roditelj cvora
   int rank; //dubina stabla za koje je on koren
   int position; //pozicija u vektoru elemenata ove strukture
};
```

Poziciju u vektoru je bilo potrebno čuvati zbog efikasnosti. Kad prolazimo kroz čvorove tražeći kanonskog predstavnika neke klase ekvivalencije, odnosno koren stabla, pronaći ćemo njemu odgovarajući Node. Da ne bismo prolazili opet kroz vektor i upoređivali koji element se poklapa sa pronađenim Node-om, bilo je efikasnije čuvati i poziciju u vektoru.

Klasa $UnionFind$ se sastoji iz vektora pokazivača na elemenate (odnosno na Node-ove) i $UseMap$-e koja nam je potrebna za implementaciju funkcija ove strukture podataka. Destruktor je potreban zbog vektora pokazivača koji je element ove klase. Konstruktor ove klase prima skup termova $TermSet$ i use mapu $UseMap$. Od svakog terma iz skupa se stvara poseban $Node$ i na početku, svaki term se nalazi u svojoj klasi ekvivalencije, odnosno, njegov pokazivač $parent$ pokazuje na njega samog. Dubina svakog stabla je na početku nula.

U nastavku ćemo objasniti dalje funkcionalnosti ove klase.

3.5 unionFind.cpp

Ovaj fajl predstavlja definiciju funkcija iz zaglavlja *unionFind.hpp*.

Funkcija **findPosition(Term s)** pronalazi poziciju u vektoru datog terma s. Ova funkcija će vratiti -1 ukoliko ne pronađe dati term u okviru vektora.

Funkcija **findRootOfTerm(Term t)** pronalazi poziciju u vektoru od kanonskog predstavnika klase ekvivalencije u okviru koje se nalazi term t. Krećemo se pokazivačima na roditelje sve dok ne naiđemo da neki $Node$ pokazuje na samog sebe, što označa da je on koren stabla. Vraća -1 ako term t ne postoji u vektoru.

Funkcija **unionOfSets(Term firstTerm, Term secondTerm)** spaja dve klase ekvivalencija. Funkcija zapravo pronalazi kanonske predstavnike odg. klasa ekvivalencije. Funkcija će preusmeriti pokazivač na roditelja korena plićeg stabla na pokazivač korena dubljeg stabla, a ukoliko su dubine jednake, onda će preusmeriti pokazivač roditelja od drugog na prvog. U tom slučaju je potrebno povećati dubinu stabla za jedan.

Funkcija **findAllRoots()** pronalazi sve kanonske predstavnike klasa ekvivalencije i vraća skup termova koji im odgovaraju.

Funkcija **findAllRootNodes()**je ista kao prethodna samo što vraća skup Node-ova koji im odgovaraju.

Funkcija **findTermsFromTheSameSet(Term t)** pronalazi sve termove koji se nalaze u okviru iste klase ekvivalencije kao term t i vraća njihov skup. Funkcija radi tako što prolazi kroz ceo vektor i proverava za svaki element da li se kanonski predstavnik njegove klase ekvivalencije poklapa sa kanonskim predstavnikom klase ekvivalencije u okviru koje se nalazi term t. Funkcija ne ubacuje u skup i sam term t.

3.6 main.cpp

U ovom fajlu se nalaze potrebne funkcije za implementaciju algoritma Nelson-Oppen. To su funkcije koje služe da pokupe termove za skup T iz ulaza, zatim funkcija koja stvara UseMap i funkcije *cong, merge, cc*.

Funkcija **getTerms(vector<Formula>* vf, Formula f, TermSet& allTerms)** kupi sve termove iz vf, a zatim iz f i ubacuje ih u skup termova $allTerms$ koji je zatvoren za podtermove. To radi pomoću funkcije koja kupi termove iz jedne formule tako što prvo pokupi termove iz terma sa leve strane jednakosti, a zatim iz terma sa desne strane jednakosti. Njoj pomaže funkcija koja kupi termove iz terma. Ta funkcija u skup $allTerms$ prvo ubaci term za koji je pozvana, a zatim poziva samu sebe za svaki svoj podterm. Izlaz iz rekurzije je kad naiđemo na simbol konstante, odnosno, kada naiđemo na term koji nema podtermove.

Funkcija **getUseMap(UseMap &um, TermSet &t)** je funkcija koja za svaki term iz t proverava u okviru kojih termova se on koristi (odnosno u okviru kojih termova je dati term podterm) i ubacuje u mapu um par: term i skup termova iz t u okviru kojih se ključ koristi.

Funkcija **cong(Term firstTerm, Term secondTerm, UnionFind &u)** odgovara opisanoj funkciji iz odeljka 2. Vraća $true$ ako su funkcijski simboli termova firstTerm i secondTerm isti i ukoliko se kanonski predstavnici odgovarajućih podtermova poklapaju. Inače vraća $false$. Sve se ovo radi u okviru odgovarajuće poslate UnionFind strukture u.

Funkcija **merge(Term s, Term t, UnionFind &u)** odgovara opisanoj funkciji iz odeljka 2. Pomoću funkcije **getSetsForMerge(Term s, UnionFind &u)** pronalaze se skupovi *P_s = U { use(s') — find(s') == find(s) }* i *P_t = U { use(t') — find(t') == find(t) }*. Ova pomoćna funkcija uzima iz use mape odgovarajuće union find strukture u skup termova u okviru kojih se koristi term s, a zatim za sve termove koji se nalaze u istoj klasi ekvivalencije kao i term s pronalazi skupove termove iz use mape u okviru kojih se oni koriste. Vraća odgovarajuću uniju termova, tj. skup. Vratimo se na glavnu funkciju merge. Kada je pronašla skupove, ova funkcija spaja klase ekvivalencije za termove s i t u okviru u, a zatim za svaki par termova iz pronađenih skupova proverava da li je potrebno spojiti njihove klase ekvivalencija. Potrebno je spojiti ih ako su im kanonski predstavnici klasa ekv. u kojima se nalaze različiti i ako funkcija *cong* pozvana za ta dva terma vraća *true*.

Funkcija **cc(vector<Formula>* E, TermSet T, UnionFind &u)** odgovara opisanoj funkciji iz odeljka 2. Prolazi kroz sve jednakosti iz vektora E i poziva *merge* za njima odgovarajuće termove samo ukoliko važi da se nalaze u različitim klasama ekvivalencije.

Na kraju dolazimo do glavnog cilja ove implementacije, a to je funkcija **checkEquality(vector<Formula>* E, Formula f)** koja proverava da li se termovi iz jednakosti f nalaze u okviru iste klase ekvivalencije i vraća $true$ ako to

i važi. Funkcija vadi termove iz E i f, kreira njima odgovarajuću UseMap, stvara UnionFind strukturu pomoću koje vodimo računa o klasama ekvivalencije, a zatim poziva funkciju *cc* koja će napraviti klase ekvivalencija. Kada funkcija *cc* završi svoj posao, glavna funkcija uzima levi i desni term iz jednakosti f, prolazi kroz sve klase ekvivalencije sve dok ne naiđe da su termovi iz jednakosti f u istim klasama ekvivalencije. Ako ne pronađe, vraća $false$, što znači da se iz datog skupa jednakosti ne može zaključiti jednakost f.

4 Neki od primera za testiranje

U nastavku ćemo dati neke od primera za testiranje.

1. **{f(a,b)=a,f(b,a)=b} {f(f(a,b),f(b,a))=a};** : treba da vrati klase ekvivalencija R = {a, f(a,b), f(f(a,b),f(b,a))} , {b, f(b,a)}
2. **{y=f(x),x=g(y)} {x=g(f(x))};** : treba da vrati klase ekvivalencija R: { y, f(x) } , {x, g(y), g(f(x)) }
3. **{x=f(x)} {x=f(f(f(x)))};** : treba da vrati klase ekvivalencija R: { x , f(x), f(f(x)), f(f(f(x))) }
4. **{y=f(x),x=f(y)} {y=f(f(x))};** : treba da vrati klase ekvivalencija R: {y, f(x)}, {x, f(y), f(f(x))}
5. **{f(x,y) = f(y,x),g(x,y) = f(f(x,y),f(y,x)),g(y,x) = f(f(y,x), f(x,y))} {g(x,y) = g(y,x)};** : treba da vrati klase ekvivalencija R = { x }, { y }, { f(x, y), f(y, x) }, { g(y, x), g(x, y), f(f(x,y),f(y,x)), f(f(y,x),f(x,y)) }
6. **{f(g(a)) = g(f(a)),g(g(a)) = f(a), f(f(a)) = g(a)} { g(f(g(g(a)))) = f(a)};** : treba da vrati klase ekvivalencija R = { a }, {f(g(a)), g(f(a))}, {g(a), f(f(a)), f(g(g(a)))}, { f(a), g(g(a)), g(f(g(g(a))))}

5 Zaključak

Dalje proširenje trenutne implementacije moglo bi da se tiče problema odlučivanja univerzalnog fragmenta EUF. Fragment EUF podrazumeva da ne postoji drugih predikatskih simbola osim simbola '=', a da postoji proizvoljan broj funkcijskih simbola koji se mogu slobodno interpretirati. Ispitivanje zadovoljivosti može da se svede na ispitivanje zadovoljivosti $s_1 = t_1 \wedge ... \wedge s_n = t_n \wedge s'_1 \neq t'_1 \wedge s'_n = t'_n$. Pomoću Nelson-Oppen algoritma se odredi kongruentno zatvorenje gde je $E = \{s_1 = t_1 ... s_n = t_n\}$, a termovi se dobijaju i iz jednakosti i iz nejednakosti. Za ovu konjukciju se može reći da je zadovoljiva ako za svaku nejednakost važi da njeni termovi ne pripadaju istoj klasi ekvivalencije.