Nikola Bebić

Traženje duplikata u kodu analizom apstraktnih sintaksnih stabala

Predložen je algoritam za pronalaženje duplikata u programskom kodu zasnovan na analizi usmerenog acikličnog grafa (DAG) dobijenog iz apstraktnog sintaksnog stabla početnog koda i detekciji strukturalno sličnih podgrafova. Prikazan je metod predstavljanja takvih duplikata preko šablona u grafu, kao i efikasan metod pronalaženja sličnih pografova. Predloženi algoritam je eksperimentalno evaluiran na tri Java softvera otvorenog koda. Dobijeni rezultati uporedivi su sa onima koje daje alat CCFinder koji primenjuje algoritam za detekciju klonova zasnova na leksičkoj analizi koda.

Uvod

Duplikati (neki izvori ih nazivaju i klonovima) u programskom kodu, odnosno fragmenti koda koji su slični po nekim osobinama, u literaturi se navode kao fenomen koji negativno utiče na kvalitet softvera, jer otežavaju kasnije održavanje koda. Postojeći radovi na ovu temu pokazuju da količina duplikata u produkcijskom kodu nije zanemarljiva, i predstavlja i do 25-30% ukupnog koda (Bari i Ahmad 2011).

Često se delovi koda kopiraju kako bi se postojeća implementacija nekog algoritma ponovo iskoristila bez izdvajanja u posebnu proceduru (refactoring). Time se kasnija izmena tih delova otežava i postaje podložnija greškama.

Do sada je analizirano i predloženo nekoliko metoda za detekciju duplikata, od kojih je svaki primenljiv uglavnom na određenu vrstu duplikata. Uspešnost svakog od metoda analizira se iz dva ugla:

– *Preciznost*, odnosno udeo ispravno detektovanih duplikata naspram svih detektovanih duplikata. To su oni detektovani delovi koda koji su prema određenom merilu nedovoljno slični ili suviše mali da bi se smatrali duplikatima.

– *Obuhvatnost*, odnosno udeo ispravno detektovanih duplikata naspram svih duplikata koji postoje u kodu.

Oba ova aspekta se moraju sagledati kako bi se utvrdila tačnost nekog metoda detekcije, jer povećanje jednog obično dovodi do smanjenja drugog. Ocena tačnosti mora biti empirijska, ručnom proverom rezultata detekcije, ili se više metoda mora međusobno upoređivati na realnim primerima projekata, jer do sada nije pronađen ni jedan metod detekcije koji bi se mogao koristiti kao bazični reper pri evaluaciji novih metoda.

Metode za detekciju mogu se podeliti u nekoliko kategorija, prema nivou apstrakcije na kom analiziraju dati kôd (Roy *et al*. 2009):

– *Tekstualne,* koje analiziraju linije ulaznih fajlova sa minimalnom tekstualnom transformacijom (brisanjem nepotrebnih razmaka, itd.).

– *Leksičke*, koje prvo razdvoje ulazni kôd na niz reči (tokena, leksema) na način sličan kompajleru, a zatim taj niz analiziraju. Ovakve metode mogu da detektuju promene u nazivima promenljivih, i slične *near miss* duplikate.

– *Sintaksni analizatori*, koji se zasnivaju na analizi sintaksnog stabla. Ovakvi detektori su često usko vezani za jezik nad kojim funkcionišu, pošto moraju u sebi sadržati parser za taj jezik. Oni mogu osim *near miss* duplikata pronaći i fragmente koji se razlikuju po većim celinama (Baxter *et al*. 1998).

– *Semantički analizatori*, koji idu korak dalje od sintaksne analize, te tako mogu različitim transformacijama pronaći funkcionalno ekvivalentne delove koda koji ne moraju imati istu stru-

Nikola Bebić (1999), Sremski Karlovci, Stražilovska 18, učenik 4. razreda Gimnazije „Jovan Jovanović Zmaj" u Novom Sadu

MENTOR: Miloš Savić, Prirodno-matematički fakultet Univeriteta u Novom Sadu

kturu. Oni primenjuju statičke analize raznih aspekata programa, kao što su dijagrami toka (flow diagrams), grafovi zavisnosti (program dependence graph, PDG) itd. (Krinke 2001).

– *Vizualni analizatori*, koji odnose u kodu prikazuju grafički i oslanjaju se na ljudsku sposobnost prepoznavanja struktura koje odgovaraju ponovljenom kodu (Reiger i Ducasse 1998).

U ovom radu predložena je modifikacija sintaksnog analizatora, koja spajanjem strukturalno identičnih podstabala sintaksno stablo pretvara u usmereni aciklični graf (DAG). Daljim operacijama nad takvim grafom izdvajaju se slične celine u analiziranom kodu.

Poreklo duplikata u kodu

Duplikati u kodu nastaju iz nekoliko razloga, od kojih svaki ima svoje karakteristike u rezultujućem kodu. Najčešće nastaju „kopiranjem i lepljenjem" (copy-pasting), ili prepisivanjem (a zatim eventualnim ispravljanjem na nekoliko mesta kako bi se prilagodio kontekstu) delova postojećeg koda na mesto gde se želi ponovo upotrebiti.

Takvi duplikati se često razlikuju samo po razmacima ili komentarima i lako se mogu detektovati tekstualnim metodama detekcije. Ukoliko je bilo ispravki, onda razlike postaju veće i potrebno je upotrebiti leksičke metode detekcije.

Veći problem za detekciju predstavljaju duplikati koji se razlikuju za više od nekoliko promenljivih ili konstanti, često po čitavim izrazima, dodatim ili izmenjenim linijama koda. Oni mogu nastati kao i prethodni, ukoliko su uvedene promene značajne, ali češće nastaju kao rezultat prepravke jednog od duplikata (npr. pri ispravljanju greške), dok je drugi ostao nepromenjen.

Drugi način nastanka ovakvih duplikata jeste često korišćenje i ručno implementiranje nekih elementarnih algoritama (traženje minimuma, binarna pretraga itd.), umesto korišćenja odgovarajućih apstraktnih tipova podataka, uzrokovano nedostatkom vremena ili znanja programera. Ukoliko se takvi algoritmi implementiraju svaki put iznova, moze doći do velikih razlika između ponavljanja, kao i do grešaka u individualnim instancama. Za njihovu detekciju potrebna je sintaksna ili semantička analiza.

Algoritam za detekciju duplikata

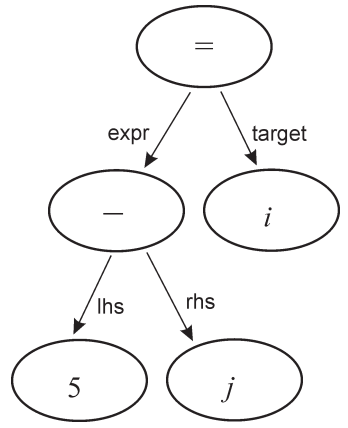
Glavni zadatak pri traženju duplikata jeste naći fragmente koda koji obavljaju istu funkciju, odnosno daju isti rezultat za iste ulazne parametre. Algoritam predložen u ovom radu to postiže transformacijama i analizom apstraktnog sintaksnog stabla, te spada u grupu sintaksnih detektora duplikata.

Algoritam se sastoji iz tri dela:

– Parsiranje koda i njegovo pretvaranje u AST
– Transformacija AST-a kako bi bio pogodniji za dalju obradu
– Traženje duplikata

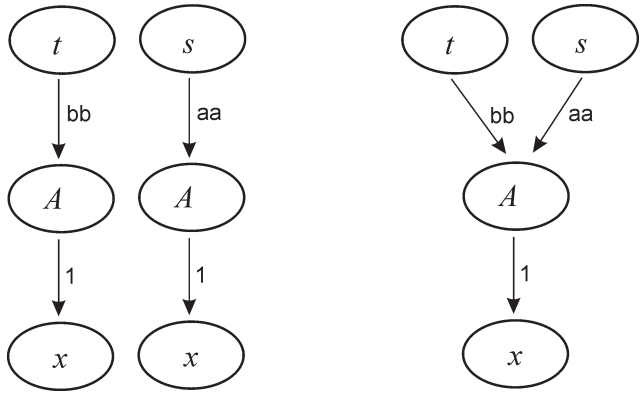
Samo parsiranje zavisi od programskog jezika. U ovom radu izabran je programski jezik Java, zbog sličnih istraživanja koja su rađena nad istim jezikom, kako bi se rezultati mogli uporediti. Nakon parsiranja se dobija AST, koji veoma blisko oslikava ulazni program, ali je lakši za algoritamsku obradu od tekstualnog oblika. U ovom slučaju, AST je predstavljen kao stablo sa obeleženim granama, gde oznake na granama predstavljaju ulogu deteta u operaciji predstavljenoj roditeljem. Na primer, izraz $i = 5 - j$ bi bio predstavljen kao na slici 1.

Ukoliko se analiza vrši nad više ulaznih fajlova, nakon stvaranja AST-a za svaki od njih, oni



Slika 1. Primer AST-a

Figure 1. Example of an AST



Slika 2. Primer transformacije spajanja identičnih čvorova

Figure 2. Example of transforming identical nodes

se dodatno spajaju i analiziraju kao jedan graf. Na ovaj način se mogu naći duplikati koji se pojavljuju u različitim fajlovima, i koji mogu ukazivati na dodatne probleme u kodu (Bruntnik *et al*. 2005).

Transformacija AST-a

Kako bi se olakšali kasniji koraci, napravljeni AST se mora dodatno transformisati. Prva transformacija koja je primenjena jeste pretvaranje stabla u usmereni aciklični graf – DAG, tako da se čvorovi istog tipa sa identičnim podstablima spoje u jedan. Na ovaj način se smanjuje veličina analiziranog grafa, i tako olakšava kasnije analiziranje. Na slici 2 prikazan je primer ovakvog procesa.

Naivna implementacija ovog algoritma, gde se proverava svaki sa svakim čvorom, i spajaju ako su jednaki, zahtevala bi $O(n^2)$ vremensku složenost, ukoliko se pretpostavi da je ukupan stepen svakog čvora u grafu $O(1)$ (što važi, jer je početni graf stablo).

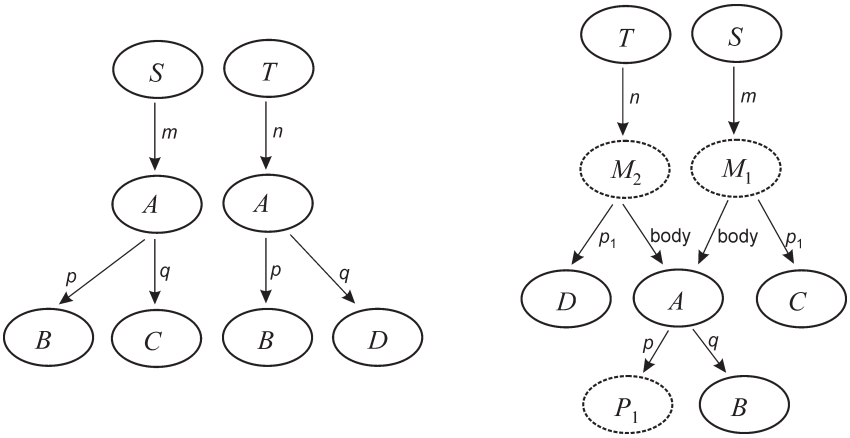
Kao optimizaciju ovog procesa možemo uvesti heširanje: najpre svakom čvoru dodelimo heš vrednost, zasnovanu na tipu čvora, heš vrednostima dece i oznakama ivica koje vode ka njima. Na taj način smo u heš vrednost uračunali tip čvora i „oblik" njegovog podstabla, pa će zbog toga čvorovi koje treba spojiti imati istu heš vrednost. Nakon toga možemo primeniti algoritam za uklanjanje identičnih čvorova iz stabla, dat na dnu ove strane.

Na kraju ovog koraka, ukoliko uzmemo samo komponentu do koje se može doći iz početnog korena, dobijamo usmereni aciklični graf koji predstavlja početno stablo sa „spojenim" identičnim čvorovima.

Ovom „pripremom" stabla za dalju obradu takođe smo uradili značajan korak u procesu traženja duplikata: našli smo sve identične duplikate. Svaki čvor sa ulaznim stepenom većim od 1 zajedno sa svojim podstablom predstavlja duplikat, ukoliko je dovoljno „velik" da se može smatrati duplikatom.

Algoritam za spajanje identičnih čvorova

```
1: procedure UNIFY (n)
2:     if n ∉ hashmap[HASH(n)] then
2:         return
4:     for all x : children of n do
5:         UNIFY(x)
6:     for all m : m ∈ hashmap[HASH(n)] and m ≠ n do
7:         if TYPE(m) = TYPE(n) and m and n have identical children then
8:             replace all occurences of m with n in graph
9:             remove m from hashmap[HASH(n)]
10:    remove n from hashmap[HASH(n)]
```



Slika 3. Primer upotrebe šablonskih i paramatarskih čvorova (označeni isprekidanom linijom)

Figure 3. Example of using pattern- and parameter-nodes (dashed)

Traženje duplikata

Poslednji deo algoritma jeste sâmo traženje duplikata, tj. pronalaženje sličnih delova koda (tzv. near miss), uključujući pritom i identične duplikate pronađene u prethodnom koraku. Ovde se algoritam može podeliti na dva dela:

– Traženje delova grafa koji predstavljaju slične delove koda, po nekoj prethodno utvrđenoj oceni „sličnosti". U ovom radu predlažemo sledeću formulu za računanje sličnosti:

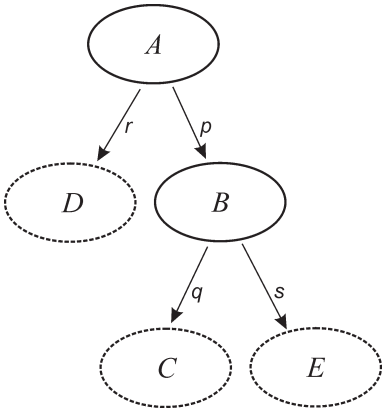
$$\eta = \frac{L_c + \rho I_c}{L_c + L_d + \rho (I_c + I_d)}$$

gde L_c, L_d predstavljaju broj zajedničkih i različitih listova, I_c i I_d broj različitih čvorova, a ρ koeficijent učestvovanja unutrašnjih čvorova, jedan od parametara algoritma. Napominjemo da se ovde posmatra broj ponavljanja određenog čvora u transformisanom grafu, a ne u početnom AST-u, odnosno kodu: na ovaj način razlike kao što su promena imena promenljive ili konstante u čitavom posmatranom delu koda nose manju težinu, pa se takvi duplikati smatraju „sličnijim".

– Pronalaženje zajedničkog podstabla nad grupama sličnih podgrafova pronađenih u prvom koraku. Ovde se u graf uvode dve dodatne vrste čvora: šablon i parametar. Na slici 3 može se videti njihova upotreba na primeru grafa. Šabloni predstavljaju parametrizovanu zamenu, gde jedna grana predstavlja „telo" smene, odnosno zajednički deo (body), dok ostale predstavljaju „vrednosti" parametara (p_1, …, p_n) koji se nalaze u zajedničkom delu. U prvom primeru, čvorovi obeleženi sa *A* i njihova podstabla su spojeni. Oni se razlikuju po detetu čvora *A* označenog granom *q*, u jednom slučaju obeleženom sa *C*, a u drugom sa *D*. Na mesto ovih razlika stavlja se poseban parametarski čvor (P_1), dok se kao odgovarajuće dete šablonskog čvora stavlja odgovarajuća smena (*C* i *D* na grafu desno).

Oba metoda (traženje identičnih i near miss duplikata) primenjuju se istovremeno, počevši od vrha: na taj način se izbegava pronalaženje suvišnih duplikata (npr. čvor *B* iz primera sa slike 3 je takođe duplikat, ali je „uklonjen" spajanjem većeg duplikata *A*).

Implementacija traženja sličnih duplikata poređenjem svakog sa svakim čvorom, zajedno sa njihovim podgrafovima, imala bi očekivano vreme izvršavanja $O(n^3)$. Kao i kod traženja identičnih duplikata, i ovde se može primeniti heširanje, ali ovaj put ne uzimajući u obzir čitav podgraf nekog čvora, već samo njegove potomke do određenog nivoa. Primer upotrebe ovakvog heširanja dat je na slici 4, računajući sve potomke osim listova. Ovakvim heširanjem gubi se uspešnost (npr. metoda sa slike 4 neće moći prepoznati duplikate koji se razlikuju po izrazima većim od jednog člana), ali se značajno smanjuje prostor pretrage.



Slika 4. Primer upotrebe metode delimičnog heširanja: čvorovi označeni isprekidanom linijom ne učestvuju u heš vrednosti čvora *A*

Figure 4. Example of using partial hashing method: dashed nodes don't contribute to the hash value of node *A*

Tabela 1. Kvantitativni rezultati

Ime projekta	Rezultati	
	Pokrivenost	Broj grupa duplikata
NEKOHTML	11.93%	23
WEKA.CLUSTERERS	14.07%	31
APACHE MAVEN	10.18%	98

Poređenja radi, u tabeli 2 mogu se videti isti rezultati dobijeni upotrebom alata CCFinder na istim izvornim fajlovima.

Tabela 2. Rezultati dobijeni upotrebom alata CCFinder

Ime projekta	Pokrivenost
NEKOHTML	9.33%
WEKA.CLUSTERERS	15.83%
APACHE MAVEN	19.39%

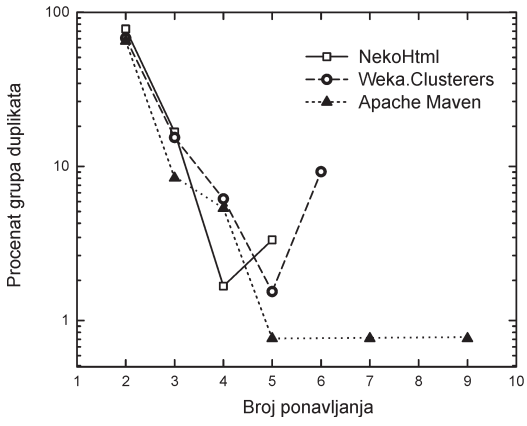
Evaluacija algoritma

Algoritam za detekciju duplikata testiran je na izvornom kodu sledećih projekata otvorenog koda:

- NekoHtml 1.9.21, 7000 linija koda
- Weka 3.9.2, paket clusterers, 8000 linija koda
- Apache Maven 1.10.0, 35000 linija koda

Osim predloženim algoritmom, koji u obzir uzima sintaksnu strukturu koda, detekcija je rađena i alatom CCFinder (Kamiya *et al*. 2002) koji spada u grupu leksičkih analizatora, odnosno ulazne fajlove analizira vodeći se isključivo leksičkom analizom koda, dok ovde predloženi algoritam u obzir uzima sintaksnu strukturu koda.

Kao rezultat posmatrana je pokrivenost koda duplikatima, odnosno procenat ulaza koji predstavlja dupliciran kod, kao i broj grupa duplikata. Ti rezultati mogu se videti u tabeli 1. Takođe na slici 5 može se videti raspodela veličina pomenutih grupa, odnosno broja ponavljanja sličnih komada koda.



Slika 5. Procenat grupa duplikata po broju ponavljanja

Figure 5. Percentage of clone groups by repetition

Zaključak

Pronalaženje duplikata u kodu je otvoren problem u računarstvu. Ovde je prikazan jedan metod za pronalaženje identičnih i sličnih delova koda, zasnovan na usmerenim acikličnim grafovima konstruisanim od apstraktnog sintaksnog stabla početnog programa. Taj metod upoređen je na nekoliko primera projekata sa alatom CCFinder, i dobijeni su slični rezultati.

Dalji rad bi se mogao zasnivati na korišćenju dobijenog stabla sa označenim duplikatima u automatizovanom uklanjanju duplikata iz koda.

Literatura

Bari M. A., Ahmad S. 2011. Code cloning: the analysis, detection and removal. *International journal of computer applications*, **20** (7): 34.

Baxter I. D., Yahin A., Moura L., Sant'Anna M. Bier L. 1998. Clone Detection Using Abstract Syntax Tree. U *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. Washington DC: IEEE Computer Society, str. 368–377.

Bruntnik M., Van Deursen A., Van Engelen R., Tourwe, T. 2005. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering*, **31** (10): 804.

Kamiya T., Kosumoto S., Katsuro I. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, **28** (7): 654.

Krinke J. 2001. Identifying similar code with program dependence graphs. U *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE '01)*. Washington DC: IEEE Computer Society, str. 301–309.

Rieger M., Ducasse S. 1998. Visual detection of duplicated code. U *ECOOP '98: Workshop ion on Object-Oriented Technology* (ur. S. Demeyer i J. Bosch). Springer, str. 75–76.

Roy C. K., Cordy J. R., Koschke R. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, **74** (7): 407.

Nikola Bebić

Finding Code Duplicates by Analyzing Abstract Syntax Trees

This paper explains the problem of duplicate code in software, how duplicates appear and the existing solutions for their detection. Further, it proposes an algorithm for their detection based on manipulation of the directed acyclic graph (DAG) derived from the abstract syntax tree of the starting code, and detection of structuraly simmilar subgraphs. It also presents a method for representing duplicates as patterns in the graph, as well as an efficient method for finding similar subgraphs. Finally, the algorithm is evaluated on three Java open source software codebases. Results were comparable to those of the CCFinder tool, a clone detection tool based on lexical code analisys.