

2023_Analysis_DP-algorithm

U ovom projektu analiziran je DP-algoritam za ispitivanje zadovoljivosti formule iskazne logike. Projekat koji je analiziran se nalazi na linku: <https://github.com/irena98/DP-algorithm>

Alati koji su korisni u testiranju projekta su sledeci:

- Unit testovi
- Clang-tidy
- Valgrind-memcheck
- lcov
- Valgrind-callgrind

Unit testovi:

Napravljen je poseban projekat u okruženju QtCreator za testiranje pocetnog projekta. Većina testova je napisana tako da pokriva tačan i netačan slučaj, ti testovi imaju isto ime samo je dotato `_f` u imenu testa za netačan slučaj.

Napisani su sledeći testovi:

- `void test_wrong_input_dnf();`
 - koji prosledjuje formulu u disjunktivnoj normalnoj formi cnf resavaču i to nikako nije obradjeno, ta informacija se u originalnom projektu ignoriše.
- `void test_wrong_input_cnf()`
 - koji šalje formulu u cnf, ali u neispravnom obliku, naime dimacs format prihvata broj klauza i broj promenljivih, ako imam 2 promenljive ne bi trebalo u formatu da imam npr promenljivu 8. Ovo takođe nije provereno.
- `void test_wrong_input_cnf2()`
 - ukoliko se posalje neispravan dimacs format program ulazi u beskonacnu petlju.
- `void test_is_literal_pure();`
 - Pravi formulu, a onda za tu formulu u kojoj je promenljiva 1 cist literal, a onda proverava da li funkcija potvrđuje da je to cist literal.
- `void test_is_literal_pure_f();`
 - Pravi formulu, a onda za tu formulu u kojoj promenljiva 1 nije cist literal, a onda proverava da li funkcija to potvrđuje.
- `void test_find_pure_literal();`
 - pravi formulu koja ima jedan cisti literal (promenljiva koja se uvek nalazi sa negacijom ili se nalazi uvek bez negacije), tu formulu salje funkciji koja bi trebalo da pronadje taj cisti literal
- `void test_find_pure_literal_f();`
 - pravi formulu koja ne sadrži ciste literale (sve promenljive koje su u formuli imaju pojavljivanje sa i bez negacije), tu formulu salje funkciji koja treba da ustanovi da ne

postoji cist literal.

- `void test_empty_clause();`
 - proverava da li potvrđuje da data formula sadrži praznu klauzu
- `void test_empty_clause_f();`
 - proverava da li potvrđuje da data formula ne sadrži praznu klauzu
- `void test_find_literal();` i `void test_find_literal_f();`
 - proveravaju da li funkcija `find literal` vraca literal koji nije cist
- `void test_unit_clause ();` i `void test_unit_clause_f();`
 - proveravaju da li postoji literal koji sam čini celu klauzu
- `void test_tautolgy_clause();` i `void test_tautolgy_clause_f();`
 - proveravaju da li je klauza tautologicna
- `void test_delete_duplicates();`
 - proverava da li se iz klauze brisu duplikati

Testovi su koristili makroe iz QtTest zaglavlja. Kako se ispravnost ulaza u program u originalnom projektu ne proverava, prilikom izvršavanja tih testova su korišćeni makroi koji nam to i naglašavaju. Za prva dva je korišćen `QEXPECT_FAIL` koji oznacava da očekujemo da taj test padne i u izveštaju ne oznacava kao `FAIL`, već kao `XFAIL` koji se ne računa u testove koji nisu prošli. Za treći test je korišćen `QSKIP` jer on izaziva beskonačnu petlju, pa je taj test preskočen. Primetimo da sam kod nije bas namenjen tesitranju, na primer funkcija koja proverava da li imamo duplikate u klauzama se ne može pozvati bez objekta tipa `formula`, iako nam `formula` nije potrebna za tu funkciju. Jedno od rešenja ovog problema je da ta i njoj slične funkcije budu `static`.

Clang-tidy

Clang-tidy je integrisan u razvojno okruženje Qt Creator, i odatle će biti pokrenut. Baziran je na Clangu i prvobitno namenjen za programski jezik C++. Ima svoje proveravače, ali moguće je i pravljenje naših proveravaca, kao i pokretanje Clang Static Analyzer-a. Kako po podrazumevanim podešavanjima nije prijavio skoro nista, uključio sam neke dodatne chekere. To je moguće uraditi tako što idemo na `projectc` → `clang tools` i promenimo `diagnostic configuration` jednostavnim čekiranjem proveravača koje želimo pokrenuti.

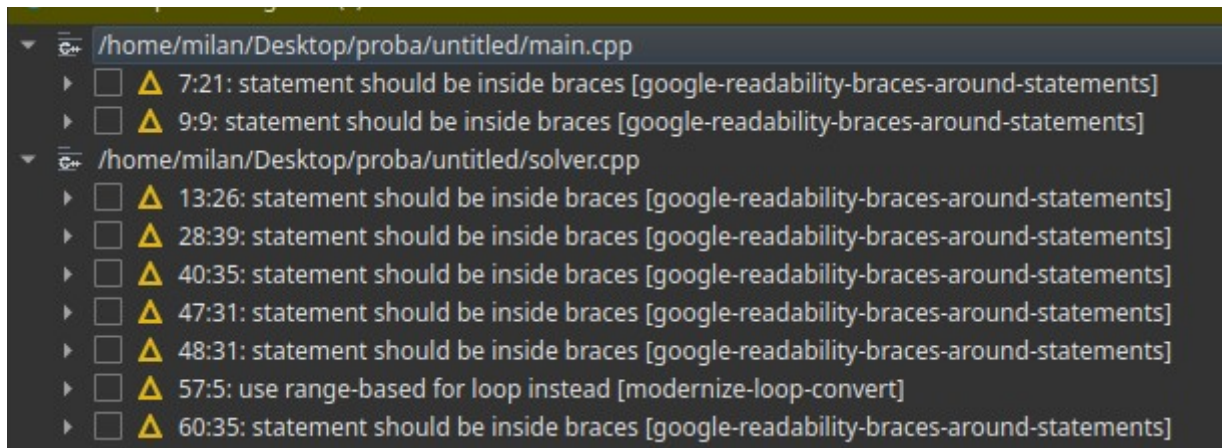
Dodatni proveravači koje sam uključio su iz par grupa:

- citljivost: `delete-null-pointer`, `duplicate-include`, `else-after-return`, `implicit-bool-conversation`, `magic-numbers`
- uključena je cela grupa za performanse
- modernizacija: `concat-nested-namespaces`, `loop-convert`, `macro-to-enum`
- i cela google grupa proveravača

Podrazumevano se pokrene i Clang Static Analyzer, tako da je to ostavljeno da se pokrene.

Sa ovakvim podešavanjem prijavljeno je dosta promena, ali većina istih, odnosno na više mesta u projektu kada imamo jednu naredbu unutar `if` bloka ili unutar tela petlje, ta naredba nije ogradjena zagrada, tako da je to prijavio na više mesta. Takodje prijavio je i da imamo jednu petlju koja ide po indeksima, `for(int i=0;i<formula.size();i++)` i predložio promenu u `for(auto &i:formula)`.

Neke od rezultata možemo videti na sledećoj slici:



Valgrind- memcheck

Memcheck detektuje greške u radu sa memorijom. Može detektovati brojne greške u radu sa memorijom koje su specifične za programske jezike C i C++ kao što su pristupanje nedozvoljenoj memoriji, curenje memorije, duplo oslobađanje memorije, korišćenje neinicijalizovanih promenljivih i slično. Napravljen je shell skript za prevodjenje programa i pokretanje memcheck alata. Sa sledećim sadržajem:

```
-g++ -g -O0 -Wall main.cpp solver.cpp  
  
-valgrind --show-leak-kinds=all --leak-check=full  
--log-file="memcheck.txt" ./a.out <primer.txt
```

Prva naredba prevodi kod u debug režimu i isključuje sve optimizacije. Druga pokreće memcheck nad izvršnim fajlom dobijenim iz prethodne naredbe sa opcijama:

--leak-check ukljućuje detaljnu pretragu za curenjem memorije
--show-leak-kinds, ova opcija oznaćava koju vrstu curenja memorije da prikaze, ona moće dobiti vrednosti iz skupa {definite, indirect, possible, reachable} i mogu biti razdvojeni zarezima ako saljemo više vrednosti. Skracenica all oznaćava sve ove vrednosti odvojene razmakom.

-- log-file nam omogućava da zadamo fajl koji se koristi za rezultate poziva alata

Rezultate mozemo videti na sledecoj slici:

```
==9212== Memcheck, a memory error detector
==9212== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==9212== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==9212== Command: ./a.out
==9212== Parent PID: 9204
==9212==
==9212==
==9212== HEAP SUMMARY:
==9212==    in use at exit: 0 bytes in 0 blocks
==9212== total heap usage: 20 allocs, 20 frees, 79,280 bytes allocated
==9212==
==9212== All heap blocks were freed -- no leaks are possible
==9212==
==9212== For lists of detected and suppressed errors, rerun with: -s
==9212== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Kao sto vidimo alat memcheck nije detektovao nikakav problem sa memorijom. Odnosno, sva alocirana memorija je i oslobodjena.

Lcov

Projekat je koriscen iz okruzenja Qt Creator, gde je modifikovano prevodjenje projekta dodavanjem neophodnih opcija za prevodjenje programa. Naime u delu za prevodjenje je dodato da se program prevodu sa opcijama -g -Wall -fprofile-arcs -ftest-coverage -O0.

Pokretanjem programa koji je generisan na ovakav nacin generise fajlove koje zatim lcov koristi za generisanje izvestaja pokrivenosti koda.

Alat lcov je pokrenut naredbom `lcov --capture --directory . --directory ../untitled --output-file pokrivenost.info --no-external` koja generise fajl pokrivenost.info. Opcija `--no-external` nam omogucava da obavestimo alat da nam nije potrebna pokrivenost koda za bilo koje fajlove van fajlova koji su zadati posle opcija `--directory`. Zatim se naredbom `genhtml -o folder pokrivenost.info` dobija citljiviji prikaz informacija iz fajla pokrivenost.info, odnosno generise folder ciji je sadrzaj html stranica sa rezultatima. Deo rezultata je prikazan na narednoj slici:

LCOV - code coverage report

| Current view: top level | | Hit | | Total | Coverage |
|---|--|------------------------|--------|-------------|---------------|
| Test: pokrivenost.info | | Lines: | 102 | 199 | 51.3 % |
| Date: 2023-09-02 17:38:20 | | Functions: | 10 | 19 | 52.6 % |
| Directory | | Line Coverage ↕ | | Functions ↕ | |
| build-novo_testiranje-Unnamed-Debug | | <div><div></div></div> | 57.1 % | 24 / 42 | 50.0 % 2 / 4 |
| untitled | | <div><div></div></div> | 49.7 % | 78 / 157 | 53.3 % 8 / 15 |

Generated by: [LCOV version 1.16](#)

Iako postignuta pokrivenost koda nije za pohvalu, dosta testova je napravljeno, i testirane su manje celine. Da bi se postigla veca pokrivenost potrebno je jednostavno dodati testove za ostale funkcije. Pritom broj testova u tom slucaju bi postao veliki. Na sledecoj slici je prikazan drugaciji prikaz pokrivenosti koji je takodje generisan istim alatom.

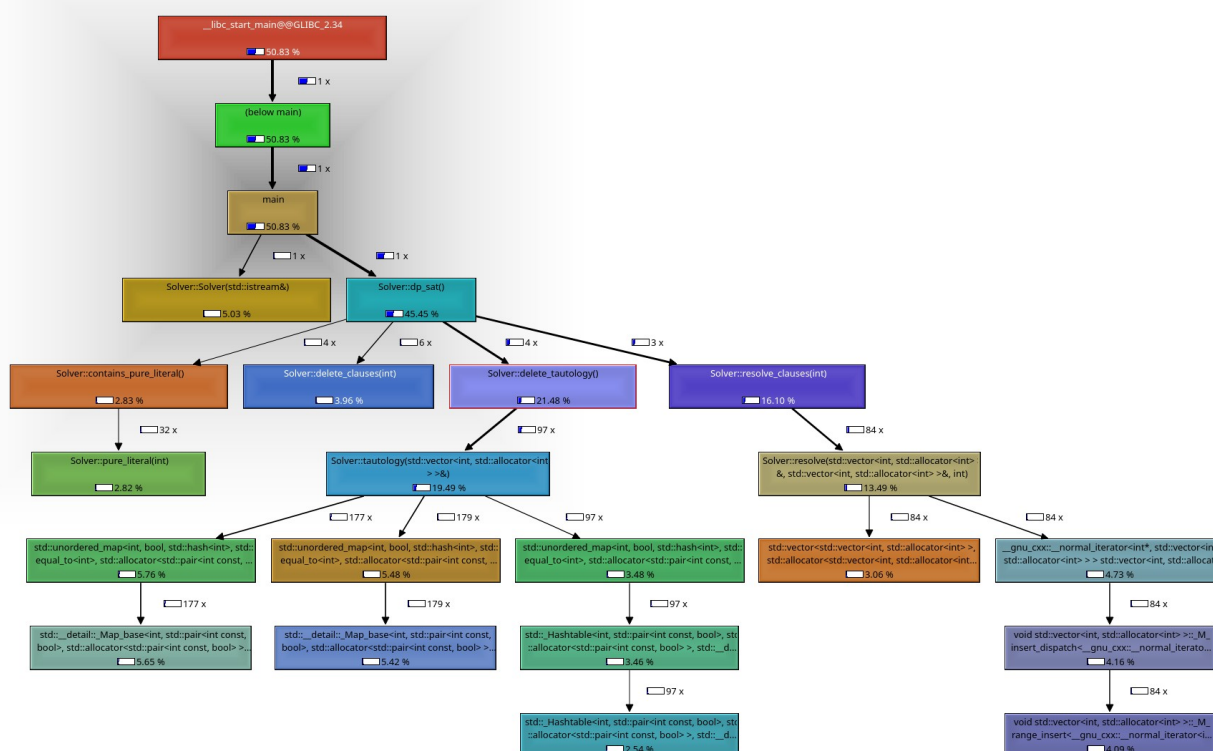
```
32         :      }
33         :
34     27 :      formula.push_back(clause);
35     27 :      }
36     12 : }
37     :
38     0 : void Solver::print_formula() {
39     0 :     for(auto &clause: formula) {
40     0 :         for(auto &literal: clause)
41     0 :             std::cout << literal << " ";
42     0 :             std::cout << std::endl;
43     :     }
44     0 : }
45     :
46     2 : Literal Solver::unit_clause() {
47     6 :     for(auto &clause: formula)
48     5 :         if(clause.size() == 1)
49     1 :             return clause.at(0);
50     :
51     1 :     return Nullliteral;
52     : }
53     :
54     0 : void Solver::unit_propagation(Literal literal) {
55     0 :     delete_clauses(literal);
56     :
57     0 :     for(int i = 0; i < formula.size(); i++) {
58     0 :         int n = formula[i].size();
59     :
60     0 :         for(int j = 0; j < n; j++)
61     0 :             if(formula[i][j] == -literal) {
62     0 :                 formula[i][j] = formula[i].back();
63     0 :                 formula[i].pop_back();
64     0 :                 n--;
65     0 :                 j--;
66     :     }
```

Callgrind

Callgrind je alat koji nam može pokazati koje sve funkcije smo pozivali, koliko smo se zadržali u tim funkcijama, kao i koja tačno funkcija je pozvala bilo koju od tih funkcija i sve to zabeležiti u jednom grafu.

Alat callgrind je pozvan iz terminala naredbom: `valgrind --tool=callgrind ./untitled <primer.txt`, ova naredba generise fajl sa informacijama o pozivima funkcija. Da bismo lakse videli informacije koje su zapisane u fajlu koristimo program KCachegrind.

Na sledecoj slici mozemo videti graf sa istorijom poziva funkcija:



Na ovom grafu su prikazane razne informacije, svaka grana ima broj, taj broj predstavlja koliko puta je ta funkcija pozvana u tom cvoru. Takođe, svaki cvor ima procenat u sebi, ti procenti predstavljaju koliko vremena ukupnog izvršavanja je provedeno u toj funkciji zajedno sa svim funkcijama koje su pozvane iz te funkcije. Iz ovog grafa možemo videti da je najviše vremena provedeno u funkciji `delete_tautology`, koja briše tautologicne klauze iz originalnog skupa. Odatle možemo zaključiti da je ta funkcija pravi izbor za optimizaciju.

Na prethodnom grafu nisu prikazana bas sve funkcije koje je nas program pozvao. Naime mozemo ici u detalje sa ovim grafom, ako se odlucimo za deo sa funkcijom `delete_tautology`, mozemo videti neke funkcije koje nisu prikazane na prethodnom grafu.