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 korisceni u testiranju projekta su sledeci:

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

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 normalnj 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 potvrdjuje 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 potvrdjuje.

- 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 izvrsavanja tih testova su korisćeni makroi koji nam to i naglašavaju. Za prva dva je korisćen QEXPECT_FAIL koji oznacava da ocekujemo da taj test padne i u izvestaju ne oznacava kao FAIL, već kao XFAIL koji se ne računa u testove koji nisu prošli. Za treći test je koriscen 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 moze pozvati bez objekta tipa formula, iako nam formula nije potrebna za tu funkciju. Jedno od resenja ovog problema je da ta i njoj slicne funkcije budu static.

Clang-tidy

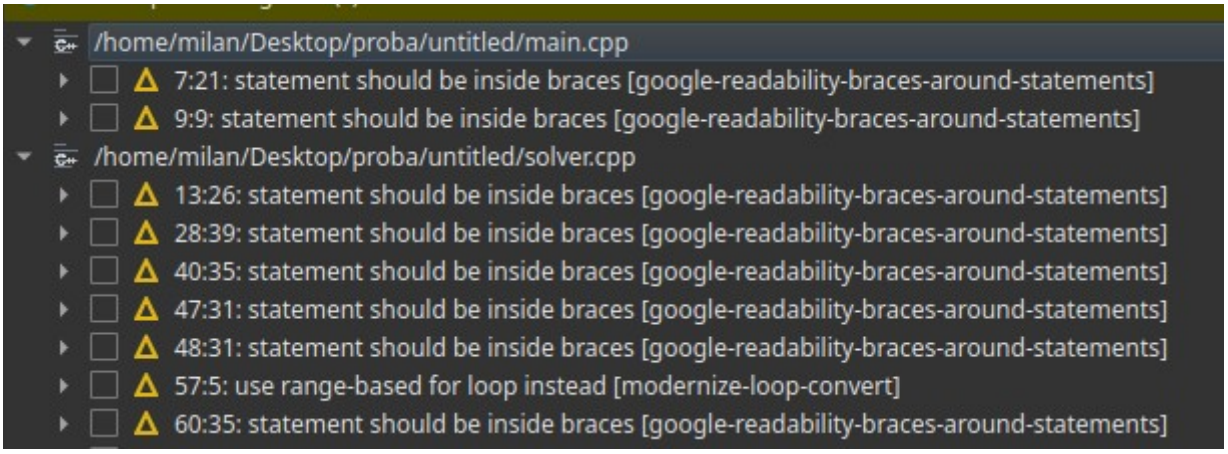
Clang-tidy je integrisan u razvojno okruzenje Qt Creator, i odatle će biti pokrenut. Baziran je na Clangu i prvobitno namenjen za programski jezik C++. Ima svoje proveravače, ali moguće je i pravljenje naših proveravaca, kao i pokretanje Clang Static Analyzer-a. Kako po podrazumevanim podesavanjima nije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

Dodatni proveravači koje sam ukljucio 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 pokrece 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 zagradama, tako da je to prijavio na vise mesta. Takodje prijavio je i da imamo jednu petlju koja ide po indeksima, for(int i=0;i<formula.size();i++) i predlozio promenu u for(auto &i:formula).

Neke od rezultata mozemo videti na sledećoj slici:



Valgrind- memcheck

Memcheck detektuje greške u radu sa memorijom. Može detektovati brojne greške u radu sa memorijom koje su specificne 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 debag režimu i iskljucuje sve optimizacije. Druga pokreće memcheck nad izvrsnim fajlom dobijenim iz prethodne naredbe sa opcijama:

`--leak-check` ukljucuje detaljnu pretragu za curenjem memorije

`--show-leak-kinds,` ova opcija oznacava koju vrstu curenja memorije da prikaze, ona moze dobiti vrednosti iz skupa {definite, indirect, possible, reachable} i mogu biti razdvojeni zarezima ako saljemo vise vrednosti. Skracenica all oznacava sve ove vrednosti odvojene razmakom.

`-- log-file` nam omogucava 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 neredbom 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

			Hit	Total	Coverage
Current view:	top level	**Lines:**	102	199	51.3 %
Test:	pokrivenost.info	**Functions:**	10	19	52.6 %
Date:	2023-09-02 17:38:20				

Directory	Line Coverage		Functions	
build-novo_testiranje-Unnamed-Debug	57.1 %	24 / 42	50.0 %	2 / 4
untitled	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            :             }
```