Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Miha Stele

Razširitev sistema ALGator za določanje mej parametrov testnih primerov

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Sistem ALGator omogoča izvajanje algoritmov na podanih testnih primerih in primerjalno analizo indikatorjev izvajanja. V diplomskem delu zasnujte logiko opisa testnega primera s pomočjo enega ali več parametrov ter razširite sistem ALGator tako, da bo znal samostojno ustvarjati testne primere s podanimi vrednostmi parametrov. Sistemu dodajte tudi logiko avtomatskega izvajanja algoritmov na testnih primerih, ki jih dobimo s sistematičnim pregledovanjem naborov parametrov. Glavni cilj naloge naj bo izdelava postopka, s katerim sistem ALGator za podan problem določi meje za parametre testnih primerov, pri katerih se algoritmi še izvedejo v razumnem času.

Kazalo

Povzetek

Naslov: Razširitev sistema ALGator za določanje mej parametrov testnih primerov

Avtor: Miha Stele

ALGator je sistem za ocenjevanje kakovosti algoritmov in analizo rezultatov. Glavna naloga diplomskega dela je bila izdelava avtomatskega preiskovanja razumnih mej za poljubne parametre. Naloga je bila posredna, saj je bilo za dosego rezultata potrebno mnogo modulov implementirati. V grobem smo napravili štiri različne sklope, in sicer definirali smo strukture celotne rešitve, zbrali in razčlenili smo konfiguracije, izdelali smo logiko za generiranje in obvladovanje kombinacij parametrov ter logiko za globljo preiskavo parametrov. V prvi sklop spada tudi raziskovanje obstoječega sistema, kjer smo poskušali uporabiti že obstoječe metode in sestaviti čim manj novih razredov. Najbolj zahteven problem je bil pri glavni logiki, kjer je nastopalo obvladovanje kombinacij in logika za globljo preiskavo. Ta ni zahtevala pretirano obsežne kode, vendar pa je sistem zahteval zelo natančno logiko in med seboj odvisne komponente. Posledično je pri tem prihajalo do največ hroščev, ki pa jih je bilo tudi najtežje odpraviti.

Ključne besede: programska oprema, integracija, avtomatizacija.

Abstract

Title: Extension of ALGator system for defining parameter limits

Author: Miha Stele

ALGator is a system for evaluating quality of algorithms and for analysing its results. In this diploma work, the main goal was to develop an automatic search for the adequate limits of any parameters. The task could not be made directly since we had to build more modules to achieve the goal. In general, our implementation consisted of four parts: definition of the whole structure, gathering and parsing the configuration data, generating and handling the combinations and logic for deep searching of a parameter. The most complex part was the main logic, which did not contain too many lines of code, but I had to be very accurate. Consequently, the hardest bugs to fix occurred there.

Keywords: software, integration, automation.

Poglavje 1

Uvod

ALGator (logotip je prikazan na sliki 1.1) je sistem za izvajanje algoritmov na podanih testnih podatkih ter analizo rezultatov izvajanja. Sistem omogoča dodajanje in upravljanje poljubnega števila projektov. V okviru enega projekta je definiran problem, testne množice vhodnih podatkov ter način reševanja nalog tega problema. Projekt lahko vsebuje poljubno število algoritmov, ki naloge rešujejo na predpisan način. Sistem omogoča analizo izvajanja posameznega algoritma ter primerjavo med algoritmi istega projekta [1]. Govorili bomo o dveh implementacijah skozi delo, tj. programa in projekta. Program je zbirka razredov, zadolžena za izvajanje projekta. Projekt je skupek algoritmov, ki rešujejo isti problem.

Slika 1.1: Logotip sistema ALGator (vir slike: [1])

V diplomskem delu smo nadgradili sistem tako, da je sposoben izmeriti razumne meje oz. meje, v katerih se program še izvede v dovolj hitrem času. Zaradi zanimanja nad algoritmi in delom na nelastnih projektih smo se odločili, da je to izziv oziroma tematika za nas.

Večkrat bomo omenili besedo parameter. Ta predstavlja argument, ki je pomemben za delovanje algoritma. Dobi jih algoritem na vhod preko funkcije, v kateri se izvaja. Vsak ima svoj opis, kjer definiramo naslednje lastnosti:

1. opis parametra oz. njegov namen, ki je koristen za uporabnika,
2. zaloga vrednosti, definirana na dva načina:
   (a) neposredni vnos vrednosti (parameter tipa `enum`),
   (b) meje oz. želeno preiskovalno območje, npr. od vrednosti 100 do 100000 (parameter tipa `long`),
3. opcije, npr. način preskakovanja.

Strukture programa smo se lotili tako, da je program v zelo veliki meri nastavljiv. Usposobili smo nastavljanje parametrov tudi preko klicnih argumentov. Za preskakovanje vrednosti parametra smo omogočili dva načina, tj. z množenjem ali s prištevanjem konstante. Odprli smo možnosti raznih načinov urejanja parametrov. Urejanje parametrov nam določa, s katerim parametrom začnemo delati kombinacije.

Kombinacije se delajo s parametri tako, da se sprehodi čez vse diskretne vrednosti (parameter tipa `long` sestavi diskretno zalogo vrednosti s pomočjo opisa, parameter tipa `enum` pa je že podan v diskretni obliki). Pri parametru tipa `long` vemo, da je številska vrednost in da večje število pomeni večje breme za algoritem, zato smo zmožni te parametre preiskovati bolj natančno oz. podrobno z globokim preiskovanjem (glej 4.4.2). Pri tipu `enum` pa imamo že diskretno zalogo vrednosti in ni nujno, da je število, zato ne vemo, pri kateri vrednosti bo trajalo najdlje.

Parametre uredimo padajoče (implementacija se mora za zdaj obvezno najprej urediti po tipu tako, da so parametri tipa `long` prvi na vrsti pri tvorjenju kombinacij, nato pa še padajoče), kjer algoritem začne delati kombinacije najprej s prvim parametrom. Gre za optimizacijo, saj se tako sprehajamo čez večje nabore števil najprej in jih že takoj omejimo. Za lažjo predstavo pa si oglejte poglavje 4.3.1.

Poglavje 2

Orodja in ogrodja

Sistem Algator je v veliki meri že zasnovan, zato smo se odločili uporabljati orodja, ki so bila uporabljena že pri zasnovi. Za specifične opravke pa smo uporabili poljubna orodja.

2.1 Linux

2.1.1 Splošno o jedru Linux

Linux spada v družino odprtokodne programske opreme, zgrajene na osnovi jedra Linux. Uporablja se na osebnih računalnikih in strežnikih v obliki Linux distribucij, na raznih vgrajenih sistemih, kot so usmerjevalniki, brezžične dostopne točke, sprejemniki FTA, telefoni, pametne televizije, video snemalniki in pripomočki NAS (Network Attached Storage). Leta 1991 ga je Linus T. razvil za svoj osebni računalnik. Ni imel namena, da bi ga podprl za več platform, vendar je to sčasoma vseeno storil za veliko računalniških arhitektur [8].

2.1.2 Linux distribucije

Linux distribucija je operacijski sistem, ki za svoje delovanje uporablja jedro Linux. Obstaja veliko distribucij, kjer je lahko posamezna distribucija

specializirana za določen pomen. Primeri distribucij:

- Za splošno rabo: Ubuntu, Mint, Fedora, openSUSE, Elementary OS...
- Za strežniško rabo: Ubuntu, Fedora, CentOS, RHEL...
- Poudarek na zasebnosti: Tails
- Poudarek na testiranju varnosti: Kali

2.1.3 Linux Fedora

Fedora je distribucija, ki se verjetno najhitreje posodablja. Iz izkušenj trdimo, da pridejo posodobitve tedensko, nova različica sistema pa izide vsakega pol leta. Dobra lastnost s strani razvijalcev je, da distribucija v repozitoriju razpoložljive programske opreme ponuja samo odprtokodne aplikacije.

2.2 NetBeans

NetBeans je integrirano razvojno okolje (IDE) za programski jezik Java. NetBeans omogoča razvijanje aplikacij iz nabora modularnih komponent programske opreme, imenovanih modulov. NetBeans deluje na operacijskem sistemu Windows, MacOS, Linux in Solaris. Poleg podpore za Javo ima razširitve tudi za druge jezike, kot so PHP, C, C ++, HTML5 in Javascript [9].

2.3 Java

Java je objektno usmerjen programski jezik, ki ga je razvil Sun Microsystems leta 1995. Splošno o Javi:

- je posplošitev jezika C in C + +. Svojo obliko je dobila iz jezika C in OOP funkcije iz C++,

- programi so neodvisni od platforme, kar pomeni, da jih je mogoče zagnati v poljubnem operacijskem sistemu s poljubnim procesorjem, če je na tem sistemu na voljo tolmačenje Jave,

- ko prevedemo program, ga lahko zaženemo s poljubnega računalnika, ki podpira Javo,

- programski jezik za delovanje potrebuje JVM. To pa je komponenta, ki je sestavljena v programskem jeziku C in C++, zato je odvisna od platforme [6].

2.4 Javadoc

Javadoc je generator dokumentacije, ki ga je ustvaril Sun Microsystems za generiranje API dokumentacije v obliki HTML iz izvorne kode Java. Format "doc comments", ki ga uporablja Javadoc, je industrijski standard za dokumentiranje javanskih razredov. Nekateri IDE, kot IntelliJ IDEA, NetBeans in Eclipse, samodejno ustvarjajo Javadoc HTML. Mnogi urejevalniki datotek pomagajo uporabniku pri izdelavi vira Javadoc in uporabijo to informacijo kot notranje priporočilo za programerja. Javadoc nudi tudi API za ustvarjanje dockletov in oznak, ki uporabnikom omogočajo analizo strukture aplikacije Java. Tako lahko JDiff ustvari poročila o tem, kaj se je spremenilo med dvema različicama API-ja. Javadoc ne vpliva na zmogljivost Jave, saj so vsi komentarji odstranjeni v času urejanja. Pisanje komentarjev in Javadoc sta namenjena boljšemu razumevanju kode in s tem boljšemu vzdrževanju [7].

2.5 Tekstovni urejevalniki

Programska oprema NetBeans lahko deluje kot običajni tekstovni urejevalnik. Zaradi prevelikega števila različnih javanskih razredov in konfiguracij sem se

odločil uporabljati še tekstovne urejevalnike. Uporabljal sem Atom, Geany in sistemski tekstovni urejevalnik, ki ga ponuja operacijski sistem Fedora 28.

2.6 Git

Git je sistem za nadzor različic programske opreme, torej z njim lahko porazdelimo razvoj programske opreme za več razvijalcev na prijazen način. Primarno se uporablja za upravljanje izvorne kode, vendar je sposoben upravljati poljubne datoteke.

2.7 Github in Gitlab

V obeh primerih gre za spletno gostovanje za git repozitorije, ki obvladujejo verzioniranje in shranjevanje kode. Običajno ponujajo upravljanje dostopa in druge funkcionalnosti, tj. sledenje težavam (hroščem), prošnje za implementacijo funkcionalnosti, upravljanje opravil in zahtev, dokumentacijo za vse projekte in drugo. Projekti na omenjenih platformah so lahko dostopni preko ukazne vrstice, kjer so podprti vsi ukazi sistema git. V repozitorije je mogoče dostopati oz. so vidni preko spletnega brskalnika. Za ogled javnih repozitorijev se ni potrebno registrirati na Githubu, na Gitlabu pa je za ogled pogoj vsaj registracija. Dostop do Github funkcionalnosti je mogoč tudi preko namiznih in mobilnih aplikacij [3]. Github je verjetno bolj priljubljen, vendar brezplačno dovoli samo javne repozitorije. Javne komponente smo objavili na Github, zasebne komponente, kot so npr. nekateri projekti, pa na Gitlab.

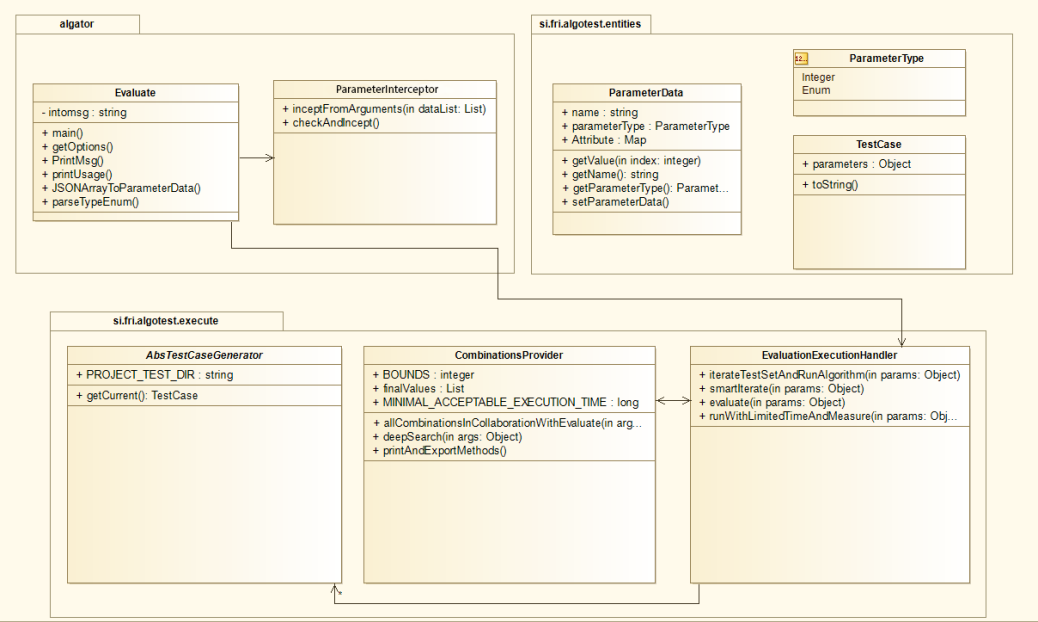
Poglavje 3

Prototip

Pri implementaciji smo morali dopolniti program in projekt. Program je kljub integraciji ostajal v veliki meri neodvisen od drugih funkcionalnosti, ki jih ponuja. Pri projektu pa podpiramo popolnoma enako strukturo kot prej, le da za izvajanje novih funkcionalnosti potrebujemo nekaj novih datotek. Med njima poteka interakcija tako, da program prevede projekt in ga uporabi v svojem izvajanju. Za poenostavitev postopka nam programski jezik celo omogoča nastaviti abstraktne razrede, ki nam omogočajo lažji pristop glede interakcije kot pa npr. v neobjektnih programskih jezikih, kot je C.

3.1 Struktura novih modulov v programu

Slika 3.1 prikazuje strukturo vseh razredov, ki nastopajo v programu pri iskanju mej. Pa začnimo pri glavnem razredu `Evaluate`. Ta je zadolžen za vse nastavitve v programu, ki so potrebne za pravilno delovanje. Sem spada razčlenjevanje opisa parametrov, pri čemer ga pretvori v seznam parametrov, opisan v poglavju 3.1.1. Razbere tudi klicne oz. vhodne argumente, pri čemer lahko nastavimo nekaj podrobnosti, kot npr. zagon specifičnega algoritma, nastavitev maksimalnega časa izvajanja ene kombinacije, opis parametrov (kar preko argumenta) in drugo. `Evaluate` najprej pogleda za opis parametrov v datoteki `./tests/[ime-projekta]-tcd.atrd`, nato pa razred



Slika 3.1: ALGatorjev razredni diagram novih metod (programa)

`ParameterInterceptor` pogleda še opis datoteke v argumentu, katere vrednosti nato prepiše v primeru, da so podane.

V programu uporabljamo tudi dve entiteti, tj. `ParameterData` in `TestCase`. Razred `ParameterData` predstavlja entiteto, v kateri se nahajajo vsi podatki o parametru. V njem obenem dobimo tudi podatek, za kateri tip parametra gre v obliki naštevalnega (angl. *enum*) razreda, imenovanega `ParameterType`. Entiteto TestCase so napravili drugi sodelujoči, uporabljam pa jo za komunikacijo s projektom, saj prenese parametre v zagon algoritma.

Jedro delovanja, kjer se izvajata avtomatsko testiranje in iskanje razumnih mej, je vidno na sliki 3.1 v paketu `si.fri.algotest.execute`. Sestavili smo dva razreda za obvladovanje generiranja kombinacij, iskanje razumnih mej in poganjanje algoritmov, tj. `EvaluationExecutionHandler` in `CombinationsProvider`. Nameravali smo ju združiti, vendar bi zaradi prevelike količine vrstic postalo zelo nepregledno. `CombinationsProvider` vsebuje logiko za generiranje kombinacij in za globlje preiskovanje parame-

trov, `EvaluationExecutionHandler` pa je postal zadolžen za izvajanje nastalih kombinacij. Razred `AbsTestCaseGenerator` pa poskrbi, da se podatki parametrov v obliki razpršilne tabele pretvorijo v entito `TestCase`.

3.1.1 Podatkovne strukture sistema

Seznam

V sekciji 3.1 smo zasledili uporabo seznama za shranjevanje parametrov. V Javi za uporabo seznama uporabljamo implementacijo `ArrayList`. Prednost te strukture je generičnost, zato lahko shranjuje tudi razrede tipa `ParameterData`. Lahko bi rešili problem tudi z uporabo tabele v Javi, vendar v tem primeru ni fleksibilna, kajti velikost deniramo samo enkrat. Razred `ParameterData` lahko štejemo tudi kot podatkovno strukturo, saj je njegov namen ohranjati lep zapis podatkov in optimalno obdelavo.

Razred `ParameterData`

Razred je entitetni, saj shranjuje podatke o parametrih v obliki atributov. Ohranja tri atribute, tj. ime, tip parametra in metapodatke. Sposoben je obvladovati tudi vse potrebne funkcije za obvladovanje podatkov, saj ponuja podatke preko metod v želeni obliki.

Razpršilna tabela

Glavna predstavitev parametrov se nahaja v razredu `CombinationsProvider` kot razpršilna tabela, kamor se shranjujejo pari `(ključ, vrednost)` v obliki `(ime parametra, trenutna vrednost parametra)`. Struktura ima vlogo shranjevanja vrednosti parametrov o trenutnem pogonu in prenosu teh algoritmu za obdelavo. Program generira kombinacije po vrsti tako, da za vsako kombinacijo hrani svoj unikatni indeks in s pomočjo tega indeksa sestavi ustrezno kombinacijo parametrov. Za podrobnejši opis o generiranju kombinacij si oglejte poglavje 4.3.2. Odražanje kombinacij glede na indeks

pa opisuje poglavje 4.4.1. Zamislimo si, da poganjamo `i`-to kombinacijo parametrov, hkrati pa hranimo tudi vrednosti parametrov s prejšnjega pogona oz. pogona `(i-1)`-te kombinacije. Glavni problem je, da v primeru prekinitve algoritma dobimo vrednost parametra, ki ni ustrezna in zato z globljim preiskovanjem (predstavljeno v podpoglavju 4.4.2) poiščemo ustrezno mejo med zadnjo in predzadnjo vrednostjo parametra.

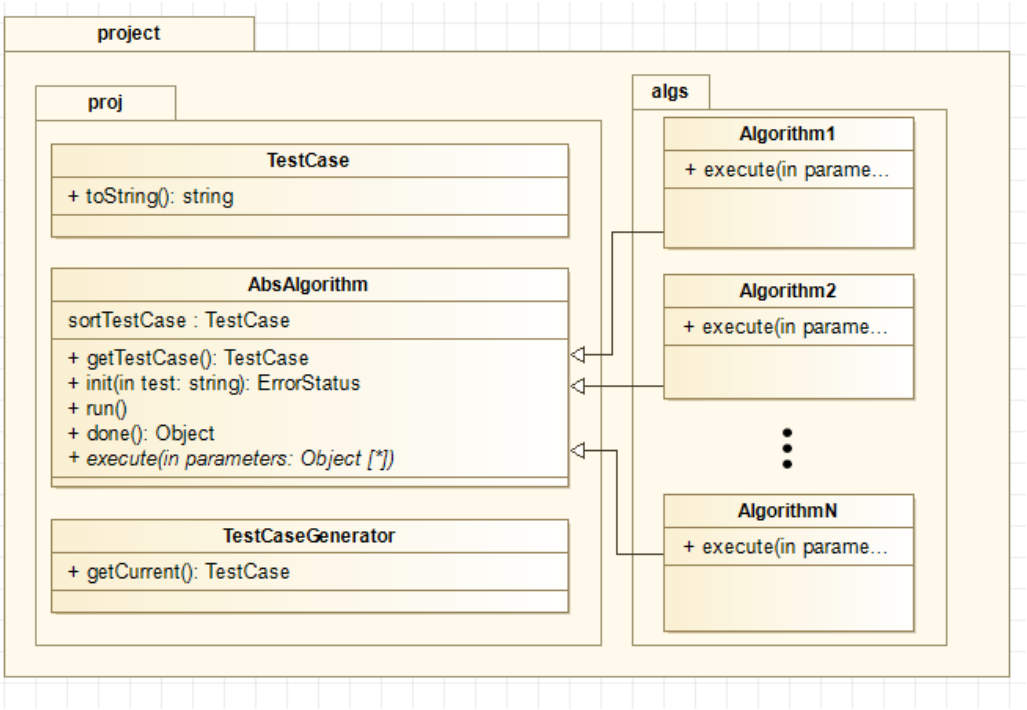
Atribut, imenovan metapodatki, izhajajoč iz razreda `ParameterData`, je razpršilna tabela. Namen tega razreda je opis pomembnih podatkov parametra za generiranje želenih kombinacij. Vsebovani podatki so tukaj lahko različni glede na tip parametra, kar naredi program bolj modularen. Zavedati pa se je treba, da to prinese tudi več možnih scenarijev, kjer delovanje programa ni pravilno.

3.2 Struktura novih modulov v projektu

Struktura projekta se ne razlikuje veliko od prejšnje strukture. Definirati moramo le dve novi datoteki, tj. razred `TestCaseGenerator` in datoteka, ki opisuje parametre. Datoteka se mora nahajati v direktoriju `[ime projekta]/tests` in biti definirana kot `[ime projekta]-tcd.atrd`, kjer `tcd` pomeni "test case description". Razred `TestCaseGenerator` smo naredili po vzorcu `TestSetIterator`ja. Struktura projekta je vidna na sliki 3.2.

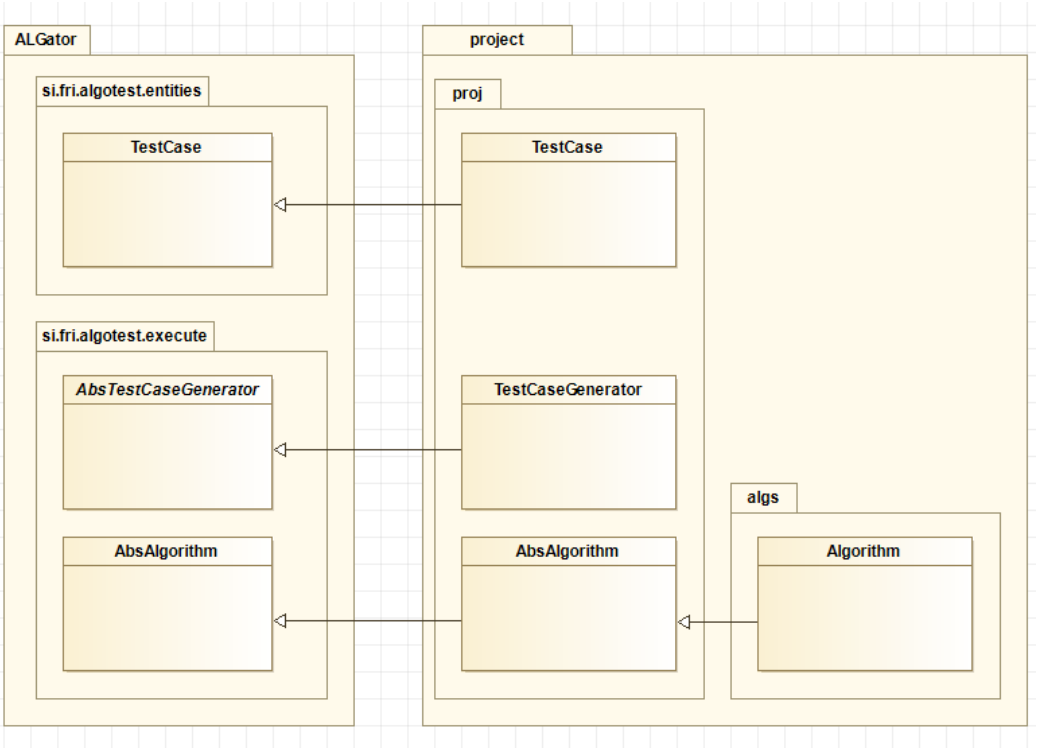
3.3 Povezava sistema in projekta

Sistem ALGator in projekt, ki se poda ALGatorju, sta strukturirana skladno, torej projekt vsebuje algoritem in še nekaj datotek za integracijo. Glavna povezava (v primeru, ko zaženemo razred `Evaluate`) sestoji iz razredov, razvidnih na sliki 3.3. Pri povezavi imata glavni vlogi `AbsTestGenerator` in `AbsAlgorithm`. `AbsTestGenerator` preko svoje metode prenese (glavno, omenjeno s poglavja 3.1.1) razpršilno tabelo s sistema v projekt. Projekt



Slika 3.2: Razredni diagram projekta

nima primarnega razreda, zato ALGator požene njegov algoritem preko razreda `AbsAlgorithm`. Ker je v projektu več algoritmov, definiramo tu še abstraktni razred, ki ga nato vsak algoritem podeduje. Glavna vloga razreda `TestCase` je prenos ustreznih podatkov o testnem primeru oz. vrednosti parametrov na vhod algoritma. `TestCaseGenerator` pa je zadolžen le za ustrezno preslikavo podatkov.



Slika 3.3: Povezava ALGatorja in projekta

Poglavje 4

Pristopi k reševanju

4.1 Povezovanje testnih primerov z novimi metodami sistema ALGator

Java je objektno usmerjen programski jezik, zato omogoča sistemu ALGator zgraditi abstraktne razrede, ki so temelj projekta. Projekt podeduje abstraktne razrede, ki jih potem avtor projekta razširi za pravilno interakcijo. ALGator se poveže s to implementacijo tako, da deklarira nove instance svojih abstraktnih razredov, ki se izpeljejo s pomočjo projekta, ki ga navedemo v ukazu. V vsakem projektu imamo štiri razrede, ki podedujejo razrede ALGatorja, tj. `TestCase`, `TestCaseGenerator`, `AbsAlgorithm` in `TestSetIterator`. `TestSetIterator` je neodvisen od diplomskega dela, zato bomo opis tega izpustili. Razred `TestCase` predstavlja le entiteto posameznega testnega primera, ki smo jo uporabili zaradi skladnosti z ostalim sistemom. `AbsAlgorithm` je še vedno abstraktni razred, ki predstavlja poenoteno obliko vseh implementacij algoritmov v projektu. `TestCaseGenerator` je na novo dodan modul, ki pa je zadolžen za pošiljanje zgeneriranih testnih primerov s sistema ALGator v entiteto `TestCase`, ki se kasneje poda v algoritem in izvede.

4.2 Prevajanje sistema

Sistem ALGator je napisan v celoti v programskem jeziku Java. Podpira pa projekte napisane v programskem jeziku Java ali C. Postopek prevajanja pa je enoten. Skrbnik sistema oz. uporabnik ALGatorja mora prevesti izvorno kodo sistema samo enkrat oz. jo lahko dobi tudi že prevedeno s spleta. Projektov v glavnem ni treba prevajati, lahko pa prevaja uporabljene knjižnice. Primarni razredi v sistemu delujejo tako, da prvič ali po presoji uporabnika kar v izvajanju programa prevedejo projekt in razred lahko potem dostopa do vsebine.

4.3 Reševanje problema kombinacij

Za avtomatizirano testiranje moramo generirati teste. Ena od možnih rešitev je generiranje vseh kombinacij. Najenostavnejša rešitev so ugnezdene zanke za vsak parameter, kjer se vsaka zanka sprehodi čez vse vrednosti. Generiranje vseh kombinacij za veliko število parametrov ni idealna rešitev, saj jih današnji procesorji niso sposobni izvesti v realnem času, saj je časovna zahtevnost $\mathcal{O}(n_1 * n_2 * ... * n_m)$, kjer je n_i obseg vrednosti i-tega parametra. Naša naloga je torej, da omejimo parametre tako, da se bodo algoritmi izvajali v smiselnih mejah parametrov.

4.3.1 Ugnezdene zanke

Implementacija je enostavna, sprehodimo se čez vse vrednosti parametrov, kjer za vsak parameter naredimo ugnezdeno zanko, ki se sprehodi čez vrednosti. Nastaneta dva problema, tj. problem poljubnega števila parametrov in omejitve. Napisati ugnezdene zanke je trivialno, če bi imeli vsi projekti enako število parametrov. V poenostavljenem primeru bi lahko rekli, da imajo vsi projekti tri parametre. Tako bi lahko generirali algoritme z algoritmom s slike 4.1. Rešitev je zelo nerodna, saj v praksi dobimo v projektih različno število parametrov. Za to implementacijo se nismo odločili, saj rešuje le

problem nespreminjajočega števila parametrov.

```
for i in parameter1:
    for j in parameter2:
        for k in parameter3:
            generate_test_case(i,j,k)
```

Slika 4.1: Psevdo koda vgnezdene for zanke

4.3.2 Zasnovana rešitev

Ena od možnih iterativnih rešitev je razvidna na sliki 4.2. Deluje preprosto tako, da se sprehodi čez vse kombinacije in odvisno od točno določene kombinacije postavi kazalce na pravo vrednost.

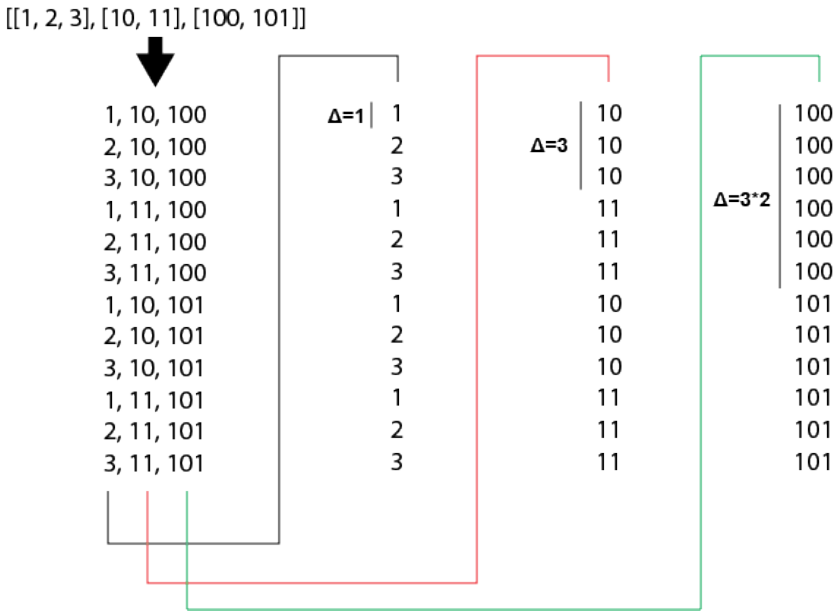
Najprej definiramo števec, ki predstavlja, katero kombinacijo bo generiral (drugače rečeno, predstavlja indeks kombinacije). Zamišljen algoritem predvidoma odpove le pri ničti kombinaciji, zato postavimo števec na 1 in prvo kombinacijo ročno vnesemo pred zanko. Definiramo tudi število vseh kombinacij in kazalec (na indeks v seznamu) za vsak posamezen parameter. Nato sledi zanka, v kateri iteriramo čez vse kombinacije. To enostavno naredimo s preverjanjem, ali je števec manjši od števila vseh kombinacij. Za vsak indeks kombinacije se sprehodimo po kazalcih parametrov in preverimo, ali je kazalec smiselno premakniti. Premik kazalca pomeni zamenjati vrednost, pri algoritmu pa premaknemo kazalec takrat, ko se generirajo vse kombinacije iz prejšnjih kazalcev. Kazalec prvega parametra premikamo konstantno, torej se sprememba zgodi za vsak indeks. Naslednji kazalec parametra se premakne, ko se prvi sprehodi čez vse vrednosti, to pa je ravno velikost nabora vrednosti v prvem parametru (slika 4.3).

Tako pridemo do dejstva, da je sprememba parametra odvisna od prejšnjih velikosti naborov vrednosti parametrov, zato jo lahko izračunamo po rekurzivni formuli:

$$a_n = \prod_{i=1}^{n-1} a_i$$

```
public static void allCombinations(vrednosti) {

        stevec = 1 #prvo kombinacijo naredimo izven zanke
        st_vseh_kombinacij = obseg(vrednosti)
        kazalci = [...] ##incializacija kazalcev za vsak parameter

        izpisiKombinacijo(vrednosti, kazalci);
        while (stevec < st_vseh_kombinacij) {
            for i in dolzina(kazalci) {
                int x = velikostPrejsnjih(vrednosti, i);
                if (stevec % x == 0) { #ne sme biti 0, saj 0 izjemoma pokvari algoritem
                    premakni_kazalec(kazalci,i)
                }

            }
            izpisiKombinacijo(values, pointers);
            stevec++;
        }

    }
```

Slika 4.2: Primer psevdokode zasnovane rešitve



Slika 4.3: Nastop spremembe vrednosti parametrov

Pri tem je a_n indeks spremembe ter indeks spremebe prvega parametra je $a_1 = 1$, saj on vedno spremeni vrednost (bodisi poviša vrednost bodisi se postavi na prvo stanje ob spremembi drugih parametrov).

4.4 Obvladovanje glavne zanke

V poglavju bomo naleteli na besedo uboj, s katero se bomo sklicevali na prekinitev izvajanja algoritma zaradi predolgega izvajanja.

4.4.1 Reševanje s kazalci

V programu smo pripravili primerne podatkovne strukture za generiranje kombinacij (predstavljeno na 3.1.1). Pripravil sem še seznam kazalcev za vsak parameter in indeks kombinacije. Pri vsakem indeksu so se parametri postavili na kombinacijo, pripadajočo unikatnemu indeksu. Zelo pomembno je, da se kazalci postavijo na vsakem indeksu, čeprav hočemo določeno kombinacijo preskočiti. Kombinacije se generirajo s pomočjo indeksa, kjer se kazalci skladno z njim povečujejo. S pomočjo kazalca se referenciramo na pravo vrednost določene kombinacije. Za preskoke odvečnih kombinacij **smo si morali zapomniti zadnji kazalec**, ki se je povečal pri določenem indeksu, saj v njem dobimo pomembne informacije. Zadnji spremenjeni parameter nam namreč pove, kateri parameter se je povečal v primeru, da gre za tip parametra `long`. Vsi prejšnji parametri, ki so se tudi spremenili, so se postavili le na prvo vrednost, ki jo kaže njihov kazalec. Za lažjo predstavo generiranja kombinacij si oglejte sliko 4.3, kjer se ob nastopu spremembe drugega parametra (z naborom vrednosti 10 in 11) pri spremembi postavi prvi parameter na minimalno vrednost oz. kazalec na prvo vrednost. Pri spremembi tretjega parametra (z naborom vrednosti 100 in 101) pa se postavita kazalca prejšnjih parametrov na prvo vrednost.

4.4.2 Globoko preiskovanje

Parameter tipa `long` se iz opisa pretvori v zalogo vrednosti svojih večkratnikov oz. seštevkov. Zaželeno je, da vsebuje vsaj dve vrednosti. Globoko preiskovanje se izvede, ko so pri vrednosti parametra na `i`-tem indeksu, kjer indeks kaže na vrednost iz zaloge vrednosti parametra, izvajanje algoritma ubije, medtem ko se vrednost pred tem še uspešno izvede, vendar se ni dovolj približala časovni omejitvi.

Rekurzivna razdelitev po delih

Prvi pristop implementacije globokega preiskovanja smo naredili že rekurzivno. Najprej smo razliko med ubito vrednostjo in vrednostjo pred ubojem razdelili na desetinke in čeznje izvedel iteracijo. Med izvajanjem teh vrednosti so pravila ostajala ista, torej, če se vrednost pred ubojem še vedno ni dovolj približala časovni meji, sta se vrednosti spet razdelili na desetinke.

Integracija bisekcije

Odločili smo se raje uporabiti bisekcijo namesto razdeljevanja. Pojavili so se problemi s skladnostjo sistema, kljub temu da je rešitev še vedno rekurzivna, saj jo je možno uporabljati na več načinov [15]. Rekurzija pri bisekciji ima dva vstopa v rekurzijo, torej si lahko premakne levo ali desno mejo na sredino. Previdni smo morali biti, da se je pravilen čas shranil in da ni izpisovalo odvečnih vrstic.

4.5 Težave

V delu je večkrat prihajalo do težav. Težave so se pojavljale že na začetku pri snovanju arhitekture, kjer je bilo veliko premisleka o posledicah uporabe raznih podatkovnih struktur. Kasneje smo našli napake tudi pri implementaciji. Pri teh smo potrebovali kar veliko premisleka, saj je bilo težko najti točno določen del, ki je povzročal nevšečnosti. Po posredovanju diplome

mentorju se je našlo še nekaj napak, ki so bile kritične. V poglavju bomo našteli le nekaj bistvenih napak.

4.5.1 Napačno obvladovanje glavne zanke

Glavna zanka nastopi pri obvladovanju generiranja in izvajanja kombinacij. Zanka je sestavljena iz veliko komponent, tj. območje generiranja, obvladovanje preskokov, logika za globlje preiskovanje itd. Pravilno strukturiranje zanke je bilo bistvenega pomena, saj vsaka napaka povzroči povsem neuporaben program. Morali smo pokriti veliko robnih pogojev. Že ničta kombinacija je v zanki povzročala težave, saj je ostanek pri deljenju z nič v matematiki nedefinirana operacija. Morali smo preverjati tudi za unikatne pojavitve posameznih parametrov, saj je lahko pri enakih vrednosti parametrov tipa `enum` obstajalo več različnih mej istega parametra tipa `long`. Preskakovanje neželenih kombinacij je v kar veliki meri zahtevalo premišljene pogojne stavke, primer pripadajoče težave je opisan v poglavju 4.5.3.

4.5.2 Integriranje novih funkcionalnosti

Nove oz. pozabljene funkcionalnosti so povzročile zastoj pri implementaciji, saj se je morala funkcionalnost skoraj vedno integrirati z obstoječo kodo. Primer nove funkcionalnosti se je zgodil pri omogočanju uporabniku, pri parametru tipa `long`, povečevanje parametra v obliki prištevanja vrednosti. V osnovi je podpiral le množenje prejšnje vrednosti, npr. večkratnik si določil 10 in tako se je vrednost povečevala za `10x`, `100x`, `1000x` in tako naprej. Še en znan primer funkcionalnosti je bila integracija bisekcije, opisan v poglavju 4.4.2.

4.5.3 Neželeni preskoki kombinacij

Za razumevanje problema se moramo zavedati nekaterih stvari. Kot že znano, sistem preiskuje vse kombinacije in jih filtrira, tj. če se pri določenem parametru program prekine, preskoči vse kombinacije z vsebovanim večjim številom.

Predstavljamo si lahko dva parametra `A` in `B`, kjer je `A` tipa `long` in ima zalogo vrednosti `[10, 100, 1000, 10000]`, `B` pa tipa `enum` in ima vrednosti `["primer1", "primer2"]`. Parameter `A` se povečuje in išče mejo od `10` do `10000` za vse kombinacije parametra `B` (saj išče meje za vse kombinacije `enum`ov), tj. za `"primer1"` in `"primer2"`. Predpostavimo, da se je program prekinil zaradi predolgega izvajanja pri vrednosti `1000`, zato sistem preskoči kombinacijo parametra A, ki vsebuje vrednost `10000`.
Ne smemo zamešati tudi tipov parametra `enum` in `long` s primitivnimi tipi programskih jezikov, npr. z Javo. Enum tukaj pomeni, da lahko algoritmu podamo parameter, ki bodisi ni celo število bodisi ga nočemo preiskovati globlje. Primer `enum`a si lahko predstavljamo v projektu urejanje, kjer algoritmom kot parameter povemo, ali gre za urejen vhod podatkov ali za naključno urejen ali za alternirajoče ipd. Pri generiranju vseh kombinacij hočemo, da se generirajo vse kombinacije parametrov tipa `enum` za vsak parameter tipa `long`. Pri parametrih tipa `long` pa vemo, da so številski in da manjša vrednost parametra pomeni manjšo obremenitev algoritma. Te parametre brez zahtevnih pogojev preiščemo globlje. Parametri v sistemu se uredijo po tipu, torej se `long` parametri prvi pojavijo na seznamu. Predpostavili smo, da imajo parametri tipa `enum` manjšo zalogo vrednosti in smo jih z vidika pristopa rešitve s for zankami postavili v zunanje zanke. Drugače rečeno, gre za optimizacijo. Dejanje je olajšalo implementacijo, hkrati pa imamo lahko shranjeno število parametrov tipa `long`, ki nam je v tem primeru koristno. V programu se je pojavljala kritična napaka, saj je logika za preskakovanje preskočila tudi vse kombinacije tipa `enum`. V primeru prekinitve, če je zadnji spremenjen parameter tipa `enum`, je preskočilo tudi vse njegove naslednje vrednosti.

Poglavje 5

Analiza

Za analizo posameznega projekta je treba razumeti tudi stikala, ki so navedena pri zagonu programa kot argumenti:

- `e` - vedno izvedi. V primeru podanega stikala se program izvede vedno, sicer pa le v primeru zastarelosti.

- `c` - vedno prevedi. V primeru podanega stikala se program prevede tudi v primeru, da je enkrat že bil preveden.

- `o` - optimiziraj s pomočjo urejanja. Stikalo je dodano na novo in podpira več načinov urejanja parametrov za optimizacijske potrebe. Trenutno podpira le osnovno urejanje po tipu parametra, ki je nujno za delovanje programa. Implementirali pa bomo lahko tudi še dodatno urejanje po velikosti zaloge vrednosti parametrov, ki bi predstavljalo primer optimizacije.

- `l` - omejitev izvajanja ene kombinacije. Novo stikalo, ki ga nastavimo v sekundah.

- `v` - izpisuj podatke med izvajanjem programa. Med izvajanjem uporabniku izpisuje zanimive informacije, količina izpisa pa je odvisna od nastavljenega nivoja.

- `a` - zagon posameznega algoritma. V primeru, da nas zanima le posamezen algoritem, lahko nastavimo to stikalo in mu podamo ime želenega algoritma.

Vse projekte bom pognal z argumenti `-e -c -l 1.6 [ImeProjekta]`.

5.1 Vzorec rezultata

Rezultat sistema se izpiše na koncu v obliki formata JSON. Število rezultatov je zmnožek velikosti parametrov tipa `enum` pomnoženo s številom parametrov tipa `long`.


```
[
   {
      "Name":"A",
      "Min":10,
      "Max":1000,
      "Multiplier":10
   },
   {
      "Name":"B",
      "Min":100,
      "Max":1000,
      "Delta":500
   },
   {
      "Name":"C",
      "Values":[
         "X",
         "O"
      ]
   },
   {
      "Name":"D",
      "Values":[
         "Y",
         "Z"
      ]
   }
]
```


Slika 5.1: Primer opisa parametrov

Primer iz slike 5.1 vsebuje osem vrednosti v rezultatih, saj ima dva parametra tipa `enum` (`C` in `D`) z dvema vrednostima (`C` ima `X` in `O`, `D` pa `Y` in `Z`) in dva različna parametra tipa `long` (`A` in `B`).

5.2 Urejanje

Številni računalniški znanstveniki menijo, da je urejanje najbolj temeljni problem študija algoritmov [14]. Urejanje predstavlja algoritem, ki postavlja elemente v določen vrstni red. Najpogosteje uporabljena sta številčni in leksikografski vrstni red. Izhod poljubnega algoritma za urejanje mora izpolnjevati dva pogoja:

- vsak element ni manjši od prejšnjega elementa glede na želeni vrstni red,
- izhod je permutacija (je preurejen in ohranja vse izvorne elemente) [11].

Pri urejanju poznamo več algoritmov, na primer `InsertionSort`, `BubbleSort`, `SelectionSort`, `HeapSort`, `MergeSort`, `ShellSort`, in `Quicksort`.

V projektu imamo definirana parametra `N` in `Group`. `N` nam pove, koliko elementov je treba urediti. `Group` nam pove način urejenosti podatkov, ki pridejo na vhod, in je tipa `enum`, zato vsebuje naslednje vrednosti:

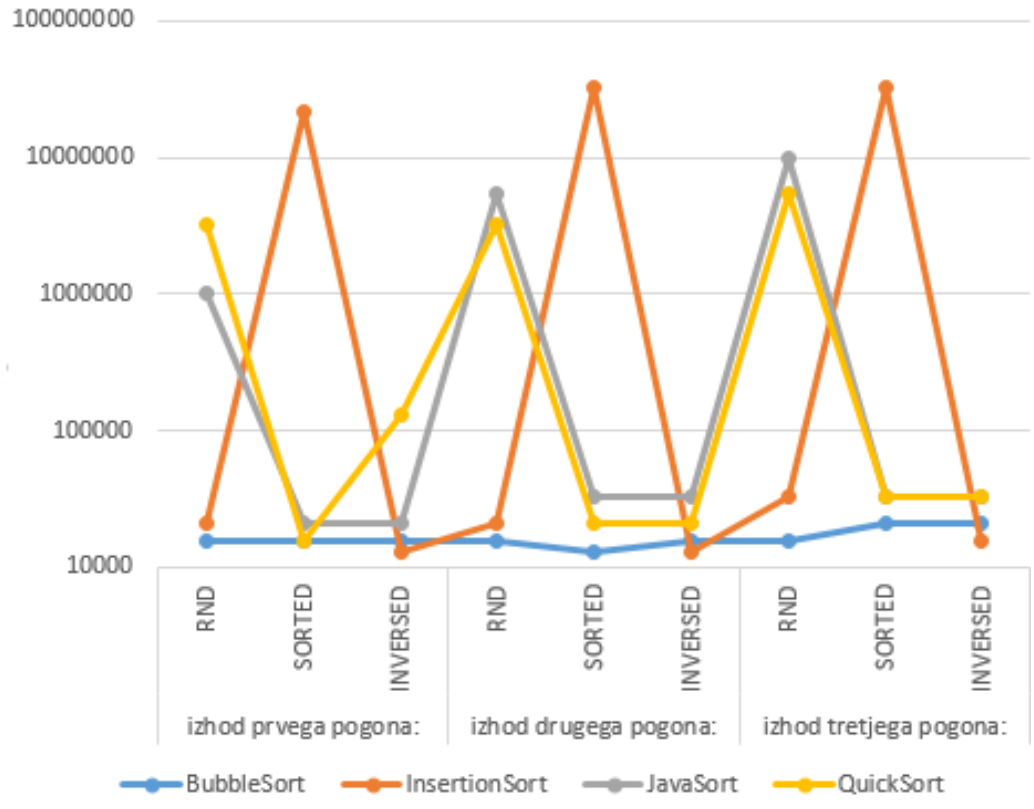
- RND - urejen naključno,
- SORTED - urejen naraščajoče,
- INVERSED - urejen padajoče.

5.2.1 Rezultati

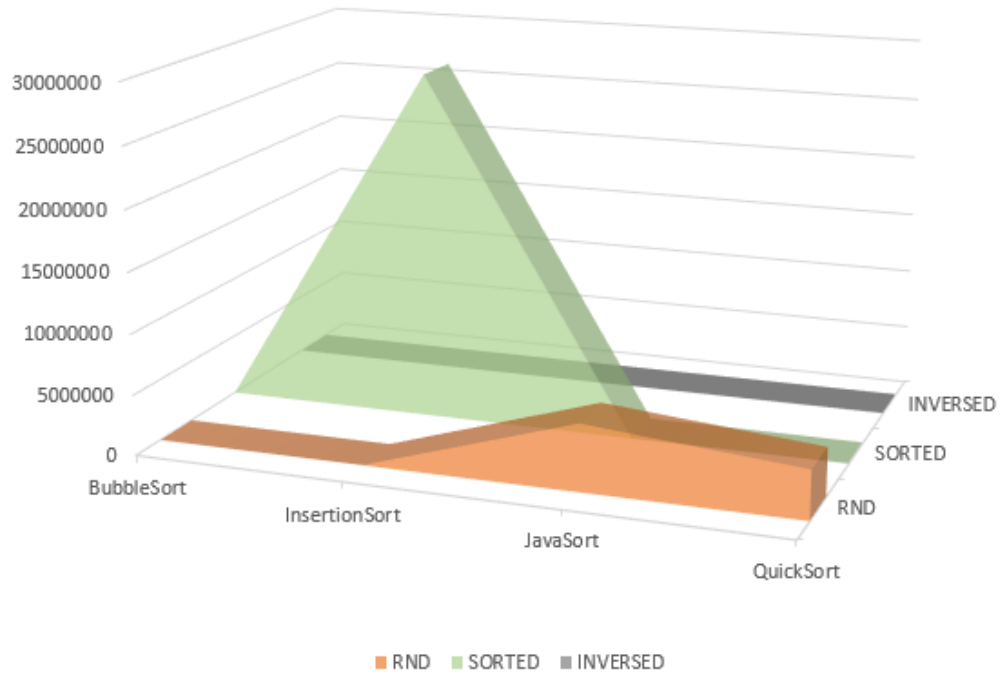
V projektu se nahajajo štirje algoritmi, to so `BubbleSort`, `InsertionSort`, `JavaSort` in `Quicksort`. Rezultati bodo prikazani glede na opis parametrov, ki je definiran na sliki 5.2. Primerjavo vseh rezultatov vidimo na sliki 5.3, povprečnih pa na sliki 5.4. Na hitro lahko razberemo, da je pri urejenem vhodu `InsertionSort` zdaleč najboljši algoritem. Pri naključnih podatkih se najbolj odreže `JavaSort`, takoj za tem pa že `Quicksort`. Pri padajoči urejenosti so vsi neučinkoviti in dosežejo podobne rezultate.

```
[
    {
        "Name":         "N",
        "Description": "Number of elements",
        "Min":          1000,
        "Max":          10000000000,
        "Multiplier":   10,
        "Type":         "long"
      },

      {
        "Name":         "Group",
        "Description": "Sortation of input",
        "Values":       ["RND","SORTED","INVERSED"],
        "Type":         "enum"
      }
]
```

Slika 5.2: Opis parametrov za urejanje



Slika 5.3: Primerjava vseh rezultatov urejanja



Slika 5.4: Primerjava povprečja rezultatov urejanja

BubbleSort

Urejanje z mehurčki ali `BubbleSort` temelji na primerjanju in zamenjavi parov sosednjih elementov, dokler niso vsi elementi urejeni [13]. Sprehodi se čez vse pare sosedov in to tolikokrat, kolikor ima elementov. Iz tega lahko razberemo, da ima algoritem časovno zahtevnost vedno $\mathcal{O}(n^2)$. Če na podlagi omenjene zahtevnosti primerjamo rezultate urejanja (glej sliko 5.5), lahko opazimo, da rezultati - ne glede na urejenost - konvergirajo k istemu oziroma zelo podobnemu številu. V rezultatih vidimo, da zaradi svoje časovne kompleksnosti doseže v vseh primerih najslabše rezultate. `BubbleSort` je bistveno počasnejši od `QuickSort`a v naključni urejenosti vhoda, pri čemer ima `QuickSort` povprečno časovno zahtevnost $\mathcal{O}(n \log n)$. Pri urejenih podatkih pa je dosegel podobne rezultate, saj tudi `QuickSort` takrat pokaže

svojo najslabšo časovno zahtevnost, tj. $\mathcal{O}(n^2)$.


```
izhod prvega pogona:
[{"Group":"RND","N":"15625"},{"Group":"SORTED","N":"15625"},{"Group":"INVERSED","N":"15625"}]

izhod drugega pogona:
[{"Group":"RND","N":"15625"},{"Group":"SORTED","N":"12812"},{"Group":"INVERSED","N":"15625"}]

izhod tretjega pogona:
[{"Group":"RND","N":"15625"},{"Group":"SORTED","N":"21250"},{"Group":"INVERSED","N":"20898"}]
```


Slika 5.5: Tri iteracije rezultatov urejanja z algoritmom BubbleSort

InsertionSort

Algoritem deluje tako, da si najprej zamislimo urejeni in neurejeni del tabele. Urejeni del je najprej prazen, nato pa se sprehodimo skozi neurejeno tabelo in vstavljamo elemente na ustrezna mesta v urejeni del. Algoritem tako nima veliko dela pri že urejeni tabeli (to je razvidno iz rezultatov na sliki 5.6).


```
izhod prvega pogona:
[{"Group":"RND","N":"21250"},{"Group":"SORTED","N":"21250000"},{"Group":"INVERSED","N":"12812"}]

izhod drugega pogona:
[{"Group":"RND","N":"21250"},{"Group":"SORTED","N":"32500000"},{"Group":"INVERSED","N":"12812"}]

izhod tretjega pogona:
[{"Group":"RND","N":"32148"},{"Group":"SORTED","N":"32500000"},{"Group":"INVERSED","N":"15625"}]
```


Slika 5.6: Tri iteracije rezultatov urejanja z InsertionSort

JavaSort

`JavaSort` predstavlja optimizirano različico algoritma `Quicksort`, zasnovanega v Javi [12]. Za podrobnejši opis si oglejte `QuickSort`. Iz rezultatov vidimo, da gre pri `JavaSort` res za optimizirano različico `QuickSort`a, saj ima skoraj povsod boljše rezultate.

```
izhod prvega pogona:
[{"Group":"RND","N":"1000000"},{"Group":"SORTED","N":"21250"},{"Group":"INVERSED","N":"21250"}]

izhod drugega pogona:
[{"Group":"RND","N":"5500000"},{"Group":"SORTED","N":"32500"},{"Group":"INVERSED","N":"32500"}]

izhod tretjega pogona:
[{"Group":"RND","N":"10000000"},{"Group":"SORTED","N":"32500"},{"Group":"INVERSED","N":"32500"}]
```


Slika 5.7: Tri iteracije rezultatov urejanja z JavaSort

Quicksort

Hitro urejanje ali `QuickSort` je učinkovit algoritem za urejanje podatkov v tabeli [4]. Je primer metode deli in vladaj, saj pri hitrem urejanju tabelo razdelimo na dve podtabeli, ki ju nato uredimo z enakim postopkom. Postopek je naslednji:

- Izberemo delilni element, ki ga imenujemo pivot,
- elemente v tabeli premečemo,
- uredimo podtabeli elementov pred in za pivotom [4].

Znano je, da algoritem, ko ima vhodne podatke urejene, deluje najpočasneje v primerih in to časovno zahtevnostjo $\mathcal{O}(n^2)$. To nam prikazujejo tudi rezultati izvedb, kjer doseže podobne rezultate kot `InsertionSort` in `Bubblesort`.


```
izhod prvega pogona:
[{"Group":"RND","N":"3250000"},{"Group":"SORTED","N":"15625"},{"Group":"INVERSED","N":"128125"}]

izhod drugega pogona:
[{"Group":"RND","N":"3250000"},{"Group":"SORTED","N":"21250"},{"Group":"INVERSED","N":"21250"}]

izhod tretjega pogona:
[{"Group":"RND","N":"5500000"},{"Group":"SORTED","N":"32500"},{"Group":"INVERSED","N":"32500"}]
```


Slika 5.8: Tri iteracije rezultatov urejanja s QuickSort

5.3 Razcep na prafaktorje

Algoritmi v projektu skušajo razbiti število na produkt dveh manjših števil. Razcep na prafaktorje pri velikih številih v računalništvu še vedno predstavlja problem. Za zdaj še ni odkritega učinkovitega algoritma, vendar pa še ni dokazano, da ne obstaja. Trenutno najhitrejši algoritmi razcepa na prafaktorje delujejo tako, da razcepijo z generiranjem manjših pomožnih števil z uporabo preprostih tehnik, kot je npr. deljenje s preskušanjem (angl. *Trial Division*) [16].

V projektu imamo definirana parametra `probSize` in `Group`. Parameter `probSize` nam pove, do katere zgornje meje naj generira naključna števila. Group imamo vrednosti za število, ki ga hočemo razcepiti, določene kot:

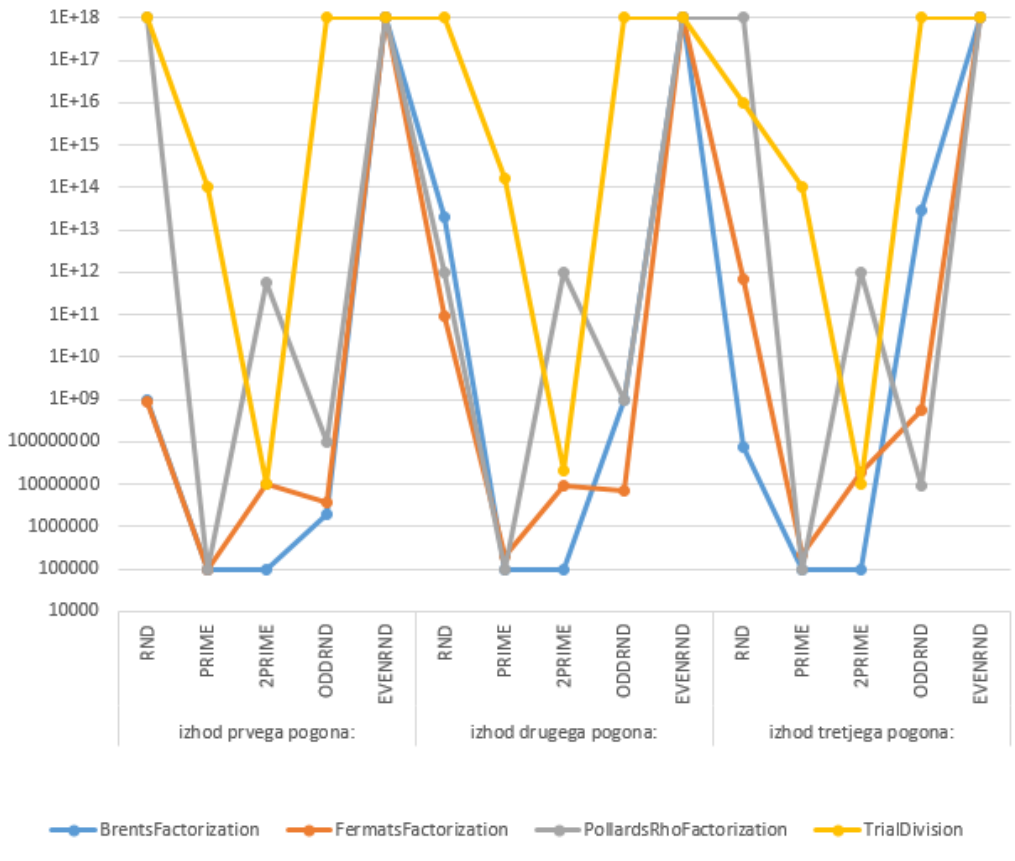
- RND - naključno število,
- PRIME - naključno praštevilo,
- 2PRIME - naključno število, sestavljeno iz zmnožka dveh praštevil,
- ODDRND - naključno liho število,
- EVENRND - naključno sodo število.

5.3.1 Rezultati

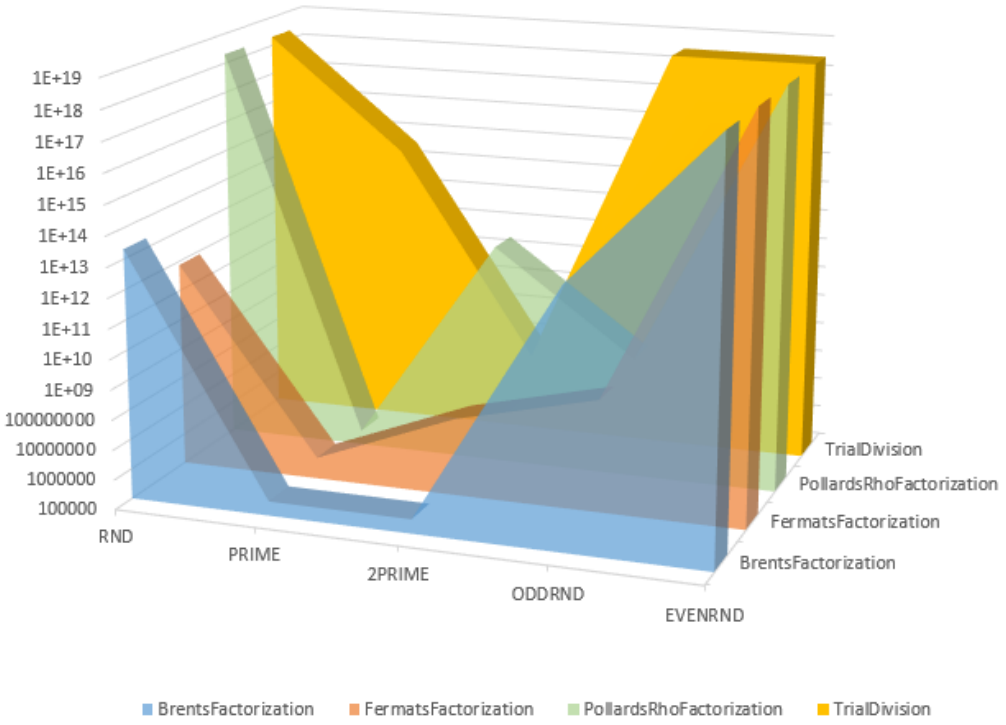
V projektu se nahajajo štirje algoritmi, to so `BrentsFactorization`, `FermatsFactorization`, `PollardsRhoFactorization` in `TrialDivision`. Rezultati bodo prikazani glede na opis parametrov, ki je definiran na sliki 5.9. Primerjavo vseh rezultatov vidimo na sliki 5.10, povprečnih pa na 5.11. Na prvi pogled zgleda `TrialDivision` prevladujoči s svojimi rezultati in zato je videti najučinkovitejši. Problem je v tem, da v analizi nismo pokrili dovolj velike zgornje meje. Na splošno vidimo, da je najpotratnejše razbijanje pri vseh algoritmih za praštevila. S tega lahko trdimo, da so algoritmi neučinkoviti za dokazovanje praštevila. `PollardsRhoFactorization` se izkaže za najboljši algoritem razbijanja števila, ki je zmnožek dveh praštevil.

```
[
      {
        "Name":        "Group",
        "Description": "type of input",
        "Values":      ["RND","PRIME", "2PRIME", "ODDRND", "EVENRND"],
        "Type":        "enum"
      },
      {
        "Name":        "ProbSize",
        "Description": "problem size",
        "Min":         100000,
        "Max":         1000000000000000000,
        "Multiplier":       10,
        "Type":        "long"
      }
]
```

Slika 5.9: Opis parametrov za razcep na prafaktorje



Slika 5.10: Primerjava vseh rezultatov urejanja



Slika 5.11: Primerjava povprečja rezultatov urejanja

TrialDivision

Po navadi, ko govorimo o algoritmu `TrialDivision`, govorimo o iskanju vseh prafaktorjev števila. Tokrat pa bomo za skladno primerjavo z drugimi algoritmi uporabljali tako, da uspešno razcepi število samo enkrat in se zaključi. Definirajmo iskano število n in števec i. Algoritem deluje tako, da povečuje i do velikosti $\sqrt{n}$ in preverja, ali je n deljiv z i. Algoritmu tako pripišemo časovno kompleksnost $\mathcal{O}(n^{\frac{1}{2}})$. V rezultatu na sliki 5.12 vidimo, da je algoritem skoraj pri vseh primerih zelo učinkovit, razen pri zmnožku dveh velikih praštevil. To je razumljivo, saj da prišteje do velikih praštevil, potrebuje več časa, kot pa npr. prištevanje do števila 2. Analiza nam prikaže rezultate le do razpona števil do 10^{18}, kar pomeni, da algoritem deluje dobro v tem območju. Za še večje razpone pa nimamo dovolj podatkov z analize,

kateri algoritmi so najboljši, vendar `TrialDivision` zagotovo ni učinkovit v primerjavi z drugimi hevrističnimi metodami.


```
izhod prvega pogona:
[{"Group":"RND","ProbSize":"1000000000000000000"}, {"Group":"PRIME","ProbSize":"100000000000000"},
 {"Group":"2PRIME","ProbSize":"10000000"}, {"Group":"ODDRND","ProbSize":"10000000000000000000"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]

izhod drugega pogona:
[{"Group":"RND","ProbSize":"1000000000000000000"}, {"Group":"PRIME","ProbSize":"156250000000000"},
 {"Group":"2PRIME","ProbSize":"21250000"}, {"Group":"ODDRND","ProbSize":"10000000000000000000"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]

izhod tretjega pogona:
[{"Group":"RND","ProbSize":"9964843750000000"}, {"Group":"PRIME","ProbSize":"100000000000000"},
 {"Group":"2PRIME","ProbSize":"10000000"}, {"Group":"ODDRND","ProbSize":"10000000000000000000"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]
```


Slika 5.12: Tri iteracije rezultatov razcepa na prafaktorje z uporabo algoritma TrialDivision

FermatsFactorization

S podanim številom n algoritem `FermatsFactorization` poišče cela števila x in y tako, da bo $n = x^2 - y^2$ [2]. Če zapišemo enačbo v razstavljeni obliki, dobimo razcepljeno število na faktorja $(x + y)$ in $(x - y)$ [2]. Naša implementacija `FermatsFactorizationa` je vidna na sliki 5.13. Tako iz re-

```
1  faktorja <- []
2  def fermat(n):
3      ce n mod 2 = 0:
4          faktorja.dodaj(2)
5          faktorja.dodaj(n / 2)
6
7      x <- zaokrozi_navzgor(sqrt(n))
8      y <- x * x - n
9      dokler ni_kvadratno_stevilo(y):
10         x <- x + 1
11         y <- x * x - n
12
13     s <- zaokrozi_navzgor(sqrt(y))
14     a <- x - s
15     b <- x + s
16     faktorja.dodaj(a)
17     faktorja.dodaj(b)
18
19     vrni faktorja
```

Slika 5.13: Psevdokoda algoritma FermatsFactorization

zultata kot iz izvorne kode razberemo, da pri sodih številih program deluje

zelo hitro. Zanimivo je, da program deluje dobro s števili, ki so zmnožek dveh praštevil. Pri tem je boljše rezultate pokazal le `PollardsRhoFactorization`. Z rezultati povprečnega delovanja, tj. s popolnoma naključnim številom, je program v delovanju primerljiv z drugi algoritmi. Algoritem v primeru, da gre za praštevilo, potrebuje n korakov [2]. V rezultatih na sliki 5.14 je to dobro razvidno, saj ima tam bistveno slabše rezultate. Ugotovili smo, da je `FermatsFactorization` slab algoritem za preverjanje, ali gre za praštevilo.


```
izhod prvega pogona:
[{"Group":"RND","ProbSize":"876953125"},{"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"10000000"}, {"Group":"ODDRND","ProbSize":"3636718"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]

izhod drugega pogona:
[{"Group":"RND","ProbSize":"95781250000"}, {"Group":"PRIME","ProbSize":"208984"},
 {"Group":"2PRIME","ProbSize":"8875000"}, {"Group":"ODDRND","ProbSize":"7187500"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]

izhod tretjega pogona:
[{"Group":"RND","ProbSize":"662500000000"}, {"Group":"PRIME","ProbSize":"212500"},
 {"Group":"2PRIME","ProbSize":"19843750"}, {"Group":"ODDRND","ProbSize":"553515625"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]
```


Slika 5.14: Tri iteracije rezultatov razcepa na prafaktorje z uporabo algoritma FermatsFactorization

PollardsRhoFactorization

`PollardsRhoFactorization` je algoritem tipa Monte Carlo. Algoritem Monte Carlo se izvaja s pomočjo naključnih števil, ki se preračunavajo z neko formulo za aproksimirano rešitev. Take vrste algoritem se konča, ko dobimo zadosti natančen rezultat. Delovanje algoritma `PollardsRhoFactorization` je vidno na sliki 5.15.

Funkcija $g(x)$ je v našem primeru $x^2 + 1 \mod n$. Algoritem se bo zaradi poskušanja iskanja dveh deljiteljev praštevila izvajal v nedogled [10]. V rezultatu na sliki 5.16 vidimo, da program nikoli ne doseže več, kot je minimalna vrednost v opisu. Program je s tako vrednostjo nakazal, da se je prekinil že pri prvem zagonu.

```
x <- 2; y <- 2; d <- 1
    dokler d = 1:
        x <- g(x)
        y <- g(g(y))
        d <- gcd(|x - y|, n)
    ce d = n:
        vrni opozorilo o neuspehu
    sicer:
        vrni d
```

Slika 5.15: Psevdokoda algoritma PollardsRhoFactorization


```
izhod prvega pogona:
[{"Group":"RND","ProbSize":"1000000000000000000"}, {"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"550000000000"}, {"Group":"ODDRND","ProbSize":"99648437"},
 {"Group":"EVENRND","ProbSize":"10000000000000000000"}]

izhod drugega pogona:
[{"Group":"RND","ProbSize":"996484375000"}, {"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"1000000000000"}, {"Group":"ODDRND","ProbSize":"996484375"},
 {"Group":"EVENRND","ProbSize":"1000000000000000000"}]

izhod tretjega pogona:
[{"Group":"RND","ProbSize":"1000000000000000000"}, {"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"1000000000000"}, {"Group":"ODDRND","ProbSize":"9648437"},
 {"Group":"EVENRND","ProbSize":"1000000000000000000"}]
```


Slika 5.16: Tri iteracije rezultatov razcepa na prafaktorje z uporabo algoritma PollardsRhoFactorization

BrentsFactorization

`Richard P. Brent` je razvil učinkovitejšo različico algoritma `PollardsRhoFactorization` [5]. Naša implementacija poteka po korakih na sliki 5.17.

```
1  def brent(n):
2      a <- 2, b <- 2, i <- 1, faktorja <- []
3      dokler 1 = 1: //pogoj, ki je ves cas resnicen
4          a <- (a * a + 1) mod n
5          s <- najvecji_skupni_deljitelj(n, a - b)
6
7          ce (s != 1 in s != n)):
8              faktorja.dodaj(s)
9              faktorja.dodaj(n / s)
10             vrni faktorja
11
12         ce veckratnik_stevila_dva(i):
13             b = a
14
15         i = i + 1
16
17
```

Slika 5.17: Psevdokoda algoritma BrentsFactorization

V rezultatih na sliki 5.18 opazimo, da se posledično tudi ta algoritem izvaja v nedogled pri praštevilih. Opazimo tudi, da se pri zmnožku dveh praštevil algoritem v vseh treh primerih prekine. Drugi rezultati pa nakazujejo, da algoritem doseže razcep manjših števil od `PollardsRhoFactorization`. Vzroki za take rezultate niso znani, kandidati pa so lahko slabša implementacija `BrentsFactorization`a v projektu, nepoštena (v smislu dodelovanja) izbira naključnih števil, ali pa sploh nismo našli pravih vhodnih podatkov, kjer je algoritem smiseln oz. koristen.

```
izhod prvega pogona:
[{"Group":"RND","ProbSize":"978906250"}, {"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"100000"}, {"Group":"ODDRND","ProbSize":"1914062"},
 {"Group":"EVENRND","ProbSize":"1000000000000000000"}]

izhod drugega pogona:
[{"Group":"RND","ProbSize":"20898437500000"}, {"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"100000"}, {"Group":"ODDRND","ProbSize":"933203125"},
 {"Group":"EVENRND","ProbSize":"1000000000000000000"}]

izhod tretjega pogona:
[{"Group":"RND","ProbSize":"77148437"}, {"Group":"PRIME","ProbSize":"100000"},
 {"Group":"2PRIME","ProbSize":"100000"}, {"Group":"ODDRND","ProbSize":"29335937500000"},
 {"Group":"EVENRND","ProbSize":"1000000000000000000"}]
```


Slika 5.18: Tri iteracije rezultatov razcepa na prafaktorje z uporabo algoritma BrentsFactorization

Poglavje 6

Sklep

6.1 Ugotovitve

Problem je bil na prvi pogled videti trivialen. Sčasoma se je izkazalo, da je problem kar kompleksen in za dobro rešitev potrebuje kar veliko časa. Izdelana rešitev je prestala veliko ročnega testiranja, zato si upam trditi, da v veliki meri deluje pravilno, vendar bi samo rešitev lahko izboljšali še na načine, ki so predstavljeni v podpoglavju 6.2.

6.2 Možne izboljšave

Izboljšave so možne z več vidikov. Program ni optimalen ne s prostorske niti s časovne zahtevnosti. Implementacija je bila odvisna od kratkoročnega časa, zato smo gledali le na enostavnost in čim večjo pravilnost programa.

6.2.1 Izboljšava prostorske zahtevnosti

Pri implementaciji sem uporabljal razrede skoraj za vsako predstavitev različnih delov. Nastajajo isti podatki drugačne oblike, kar predstavlja breme prostoru. Kot primer lahko vzamemo parametre, kjer jih imamo predstavljene kot entitete, nato pa te razbijemo še na razpršilne tabele posamičnih zalog vrednosti parametrov. Optimalno pa bi lahko uporabili le tabele, kjer se

hranijo le osnovni podatki. S teh bi lahko na račun časovne zahtevnosti generirali podatke, ko so ti potrebni.

6.2.2 Izboljšava časovne zahtevnosti

Za generiranje testnih primerov smo se odločili kar za generiranje vseh kombinacij. Zanima nas le maksimalna vrednost vsakega posameznega parametra, pri čemer so ostali parametri na minimalni vrednosti. Trdim, da je to možno rešiti optimalnejše, vendar je tudi ta implementacija zadovoljiva, saj običajno ni prevelikega nabora parametrov, zalogo vrednosti pa po navadi sami definiramo tako, da ne preišče preveliko nesmiselnih vrednosti.

6.3 Zaključek

V diplomskem delu sem se naučil nekaj pomembnih stvari, med drugim je to delo na projektu, ki ga nisem ustvaril sam. Začetki so bili težki, saj mi veliko stvari ni bilo jasnih. Sčasoma sem projekt vedno bolj usvojil in tako z lahkoto začel integrirati nove funkcionalnosti. Ker so te nove in neodvisne od ostalega sistema, so bile izdelane v ločenih razredih. Vseeno se mi zdi, da reševanje takšnih problemov pride zelo prav, saj se moraš v industriji večinoma prilagoditi že obstoječim sistemom.

Literatura

[1] Algator. https://github.com/ALGatorDevel/Algator/blob/master/doc/ALGator.docx. [Dostopano 10. 5. 2018].

[2] Fermat's factorization method. https://en.wikipedia.org/wiki/Fermat%27s_factorization_method. [Dostopano 7. 9. 2018].

[3] Github. https://en.wikipedia.org/wiki/GitHub. [Dostopano 5. 8. 2018].

[4] Hitro urejanje. http://wiki.fmf.uni-lj.si/wiki/Hitro_urejanje. [Dostopano 7. 9. 2018].

[5] An improved monte carlo factorization algorithm. https://maths-people.anu.edu.au/~brent/pub/pub051.html. [Dostopano 7. 9. 2018].

[6] Java introduction. https://www.w3schools.in/java-tutorial/intro/. [Dostopano 2. 8. 2018].

[7] Javadoc. https://en.wikipedia.org/wiki/Javadoc. [Dostopano 2. 8. 2018].

[8] Linux kernel. https://en.wikipedia.org/wiki/Linux_kernel. [Dostopano 2. 8. 2018].

[9] Netbeans. https://en.wikipedia.org/wiki/NetBeans. [Dostopano 14. 7. 2018].

[10] Pollard's rho algorithm for prime factorization. https://www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/. [Dostopano 7. 9. 2018].

[11] Sorting algorithm. https://en.wikipedia.org/wiki/Sorting_algorithm. [Dostopano 10. 9. 2018].

[12] Source for java.util.arrays. http://developer.classpath.org/doc/java/util/Arrays-source.html. [Dostopano 7. 9. 2018].

[13] Urejanje z mehurčki. http://wiki.fmf.uni-lj.si/wiki/Urejanje_z_mehur\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{c\global\mathchardef\accent@spacefactor\spacefactor}\accent20c\egroup\spacefactor\accent@spacefactorki. [Dostopano 10. 9. 2018].

[14] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[15] E. Roberts. *Thinking Recursively*. Wiley, 1986.

[16] S.S. Wagstaff. *The Joy of Factoring*. Student mathematical library. American Mathematical Society, 2013.