Univerzitet u Nišu
Prirodno-matematički fakultet
Departman za računarske nauke

Muzafer H. Saračević

METODE ZA REŠAVANJE PROBLEMA TRIANGULACIJE POLIGONA I NJIHOVA IMPLEMENTACIJA

- Doktorska disertacija -

Mentor:

Prof. dr Predrag S. Stanimirović

Niš, 2013.

Imam posebnu čast i zadovoljstvo da se zahvalim mentoru dr Predragu Stanimiroviću, redovnom profesoru Prirodno-matematičkog fakulteta u Nišu, na nesebičnoj pomoći i brojnim korisnim savetima kojima je doprineo boljem kvalitetu ove disertacije.

Veliku zahvalnost dugujem kolegi Seadu Mašoviću na iskrenoj podršci, dobrim idejama i savetima. Zahvaljujem se svim profesorima PMF-a na korektnoj saradnji u toku doktorskih studija a posebno dr Predragu Krtolici i dr Milanu Tasiću na izradi zajedničkih naučnih radova. Zahvaljujem se dr Danijeli Milošević, vanrednom profesoru Fakulteta tehničkih nauka u Čačku Univerziteta u Kragujevcu, na nesebičnoj pomoći u toku master i doktorskih studija.
Zahvaljujem se dr Donaldu Tejloru, vanrednom profesoru Fakulteta za matematiku i statistiku Univerziteta u Sidneju, na stručnim savetima koji se odnose na programiranje i implementacije metoda. Iskrenu zahvalnost izražavam kolegama i profesorima Departmana za tehničke nauke Univerziteta u Novom Pazaru.

Posebno se zahvaljujem članovima moje porodice na podršci, razumevanju i ohrabrenjima, koji su bili glavni razlog mog istrajanja u radu.

ПРИРОДНО - МАТЕМАТИЧКИ ФАКУЛТЕТ НИШ

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	монографска
Тип записа, **ТЗ**:	текстуални / графички
Врста рада, **ВР**:	докторска дисертација
Аутор, **АУ**:	Музафер Х. Сарачевић
Ментор, **МН**:	Предраг С. Станимировић
Наслов рада, **НР**:	Методе за решавање проблема триангулације полигона и њихова имплементација
Језик публикације, **ЈП**:	српски
Језик извода, **ЈИ**:	енглески
Земља публиковања, **ЗП**:	Србија
Уже географско подручје, **УГП**:	Србија
Година, **ГО**:	2013
Издавач, **ИЗ**:	ауторски репринт
Место и адреса, **МА**:	Ниш, Вишеградска 33.
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	6 поглавља/ 115 страна/ 76 референци/ 19 табела/ 50 слика/ 2 прилога
Научна област, **НО**:	Рачунарске науке
Научна дисциплина, **НД**:	Рачунарска геометрија, Објектно - оријентисано програмирање и моделирање
Предметна одредница/Кључне речи, **ПО**:	Триангулација конвексних полигона, Јава програмирање, УМЛ моделирање
УДК	**004.42: 004.92**
Чува се, **ЧУ**:	
Важна напомена, **ВН**:	
Извод, **ИЗ**:	Дисертација садржи нове методе у циљу повећања брзине генерисања триангулација полигона. Прва метода се базира на поступку декомпозиције Каталанових бројева. Друга метода се базира на конструкцији триангулација на основу блокова са акцентом на могућност складиштења и рад са базама података у Java NetBeans окружењу. Дисертација изучава наведени проблем и са аспекта записивања и складиштења са циљем уштеде меморијског простора (метода за алфа-нумерички запис). Дата је веза између нотације триангулација са комбинаторним проблемима (*ballot* проблем и путеви у целобројној мрежи). Наведена је и објектно-оријентисана анализа и дизајн за проблем триангулације, са три аспекта: директни развој (*forward engineering*), повратна анализа (*reverse engineering*) и синхронизација (*round-trip engineering*).
Датум прихватања теме, **ДП**:	**Факултет: 14.11.2012, Универзитет: 19.11.2012**
Датум одбране, **ДО**:	
Чланови комисије, **КО**: Председник:	
Члан:	
Члан:	
Члан, ментор:	

PRIRODNO-MATEMATIČKI FAKULTET NIŠ

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	monograph
Type of record, **TR**:	textual / graphic
Contents code, **CC**:	doctoral dissertation
Author, **AU**:	Muzafer H. Saračević
Mentor, **MN**:	Predrag S. Stanimirović
Title, **TI**:	Methods for solving the polygon triangulation problem and their implementation
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	English
Country of publication, **CP**:	Serbia
Locality of publication, **LP**:	Serbia
Publication year, **PY**:	2013
Publisher, **PB**:	author's reprint
Publication place, **PP**:	Niš, Višegradska 33.
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	6 chapters/ 115 pages/ 76 references/ 19 tables/ 50 images/ 2 appendix
Scientific field, **SF**:	Computer Science
Scientific discipline, **SD**:	Computational Geometry, Object-oriented programming and modeling
Subject/Key words, **S/KW**:	Triangulation of the convex polygon, Java programming, UML modeling
UC	**004.42: 004.92**
Holding data, **HD**:	
Abstract, **AB**:	The dissertation includes new methods in order to increase the speed of generating polygon triangulation. The first method is based on the process of decomposing the Catalan numbers. The second method is based on the construction of triangulation on the basis of blocks with emphasis on the possibility of storing and working with databases in Java NetBeans environment. The dissertation will also study the above problem in terms of recording and storage in order to save storage space (the method of Alpha-numeric entry). The connection is given between notation triangulation with combinatorial problems (*ballot* and *lattice path*). The object-oriented analysis is also discussed as well as the design for the problem triangulation, with three aspects: forward, reverse and round-trip engineering.
Accepted by the Scientific Board on, **ASB**:	**Faculty: 14.11.2012, University: 19.11.2012**
Defended on, **DE**:	
Defended Board, **DB**: President:	
Member:	
Member:	
Member, Mentor:	

Sadržaj

Predgovor **vi**

1 Uvodna razmatranja **1**
1.1 Pojam Katalanovih brojeva 3
1.2 Pojam triangulacije poligona 4
1.3 Jedan algoritam za triangulaciju konveksnih poligona 6

2 Implementacija algoritama računarske geometrije u *Javi* **9**
2.1 Prednosti objektno-orijentisanog programiranja i modeliranja 9
2.2 *Java* biblioteke za GUI i računarsku geometriju 11
2.2.1 *AWT* i *Swing* paketi 11
2.2.2 Otvorena biblioteka *Java OpenGL* 12
2.2.3 Klase i paketi za triangulaciju poligona 12
2.3 Prednosti *Jave* u implementaciji algoritma za triangulaciju konveksnog poligona 14
2.3.1 Implementacija *Hurtado-Noy* algoritma 15
2.3.2 Komparativna analiza implementacija (*Java, C++, Python*) 19

3 Metode za generisanje triangulacija konveksnih poligona **22**
3.1 Metod dekompozicije Katalanovih brojeva 22
3.1.1 Težinsko stablo triangulacija i dekompozicija Katalanovog broja . . . 23
3.1.2 Triangulacija konveksnog poligona bazirana na dekompoziciji Katalanovog broja 27
3.1.3 Kompleksnost algoritma 32
3.1.4 Komparativna analiza i eksperimentalni rezultati 33
3.1.5 Detalji implementacije u *Javi* 35
3.2 Blok metod za generisanje triangulacija poligona 37
3.2.1 Uvodne napomene i osnovne postavke 37
3.2.2 Algoritam za blok metod 38
3.2.3 Komparativna analiza i eksperimentalni rezultati 42
3.2.4 Detalji implementacije i rad sa bazama podataka u *Javi* 44

4 Metode za notaciju i skladištenje triangulacija poligona **48**
4.1 Ballot notacija za triangulaciju poligona 48
4.1.1 Uvodne napomene i osnovne postavke 48
4.1.2 Ballot problem i triangulacija poligona 50
4.1.3 Primena steka za skladištenje triangulacija 54
4.1.4 Komparativna analiza i eksperimentalni rezultati 56
4.1.5 Detalji implementacije u *Javi* 59
4.2 Alfanumerička notacija 62
4.2.1 Uvodne napomene i osnovne postavke 62
4.2.2 Transformacija iz BP u AN zapis 63
4.2.3 Obrnuta transformacija iz AN u BP zapis 67
4.2.4 Eksperimentalni rezultati 68
4.2.5 Detalji implementacije u *Javi* 70

5 Primena objektno-orijentisane analize i dizajna na problem triangulacije poligona **73**
5.1 Direktna analiza za algoritam triangulacije poligona (*Forward engineering*) . 74
5.1.1 Modeliranje u *UML-u*: statični, dinamički i fizički model 75
5.1.2 Generisanje *Java* izvornog koda iz *UML* modela 81
5.2 Povratna analiza i sinhronizacija (*Reverse/Round-trip engineering*) 82
5.2.1 Dobijeni rezultati u unapređenju implementacija 84

6 Zaključna razmatranja **87**

Prilozi **91**
I) Podaci o preuzimanju izveštaja za klase i implementacije metoda 91
II) *Java* izvorni kôd 93
A) Klasa: *Triangulation* 93
B) Klasa: *GenerateTriangulations* 94
C) Klasa: *NoteTriangulation* 95
D) Klase: *Node, LeafNode, Point* 97
E) Klasa: *Database* 98
F) Klasa za kreiranje izlazne datoteke 99

Spisak algoritama, tabela i slika **100**

Literatura **103**

Biografija autora **110**

Predgovor

Tema doktorske disertacije pripada oblasti algoritama računarske geometrije u kombinaciji sa objektno-orijentisanim programiranjem i modeliranjem. Po *MSC2010* klasifikuje se u *68U05: (computer graphics; computational geometry)*, *65D18: (computer graphics, image analysis, and computational geometry)*, kao i *68N15: (programming languages)*.

U disertaciji su predstavljene nove metode za generisanje i notaciju triangulacija poligona. Sve metode su implementirane u programskom jeziku *Java*, koji je u poslednjih nekoliko godina najaktuelniji objektno-orijentisani programski jezik. Pored toga, u disertaciji je razmatran i novi pristup analizi i rešavanju problema triangulacije poligona primenom metodologije objektno-orijentisane analize i dizajna.

Disertacija je podeljena u šest poglavlja. Sledi kratak opis svakog poglavlja.

1. **Prvo poglavlje** sadrži osnovne podatke o metodološkom okviru istraživanja (problemu, predmetu, ciljevima i tehnikama istraživanja). Zatim, dat je osvrt na pojam Katalanovih brojeva i pojam triangulacije konveksnih poligona. Analiziran je jedan postojeći algoritam za konstrukciju triangulacija poligona (*Hurtado-Noy*) koji je poslužio za komparativnu analizu sa novim metodama koje predstavljaju rezultat ove disertacije.

2. **U drugom poglavlju** su navedene prednosti objektno-orijentisanog programiranja i modeliranja kroz primenjene jezike *Java* i *UML*. Izučavane su mogućnosti programskog jezika *Java* u implementaciji algoritama računarske geometrije, kroz primenu postojećih biblioteka i programskih interfejsa (*Java 2D, 3D, Open GL, GeometryInfo, Triangulator*). Analizirani su postojeći *Java* paketi koji su korišćeni za implementaciju novih metoda za triangulaciju poligona. Razvijene *Java* aplikacije za implementaciju metoda uvedenih u disertaciji karakteriše grafički korisnički interfejs (GUI), a to je postignuto korišćenjem *Java* klasa iz *AWT* i *Swing* paketa. Urađena je uporedna analiza implementacija u programskim jezicima *Java, Python i C++* za *Hurtado-Noy* algoritam, gde su analizirane mogućnosti navedenih jezika u implementaciji algoritama računarske geometrije.

   Objavljeni naučni radovi autora disertacije na kojima se bazira drugo poglavlje su [53, 54, 48].

3. **Treće poglavlje** sadrži nove metode za generisanje triangulacija konveksnog poligona. Prva tehnika generisanja triangulacija se bazira na dekompoziciji Katalanovog broja u formi suma termova $(2+i)$. Dobijeni izrazi dekompozicije predstavljaju ključ u postupku generisanja skupa triangulacija $\mathcal{T}_n$ na osnovu skupa $\mathcal{T}_{n-1}$. Na osnovu ove ideje je uspostavljeno težinsko stablo triangulacija.

   Druga tehnika generisanja triangulacija je nazvana *blok metoda*. Ova metoda se zasniva na odnosu broja triangulacija dva uzastopna poligona, koji je izražen kao $T_n = 2T_{n-1} + restR_n$. Generalna strategija primene ove metode je da se glavni problem razlaže na manje potprobleme koji su međusobno zavisni. U cilju izbegavanja ponavljanja generisanja prethodnih triangulacija iz skupa $\mathcal{T}_{n-1}$ korišćena je rekurzija sa memoizacijom. U ovoj metodi je stavljen akcenat na mogućnosti rada sa bazama podataka u *Java NetBeans* okruženju (*JDBC*). Za obe nove metode urađena je komparativna analiza sa *Hurtado-Noy* algoritmom, gde je ukazano na prednosti novih predloženih metoda. Za potrebe dobijanja eksperimentalnih rezultata za svaku od pomenutih metoda je isprogramirana posebna *Java* aplikacija.

   Objavljeni naučni radovi autora na kojima se bazira treće poglavlje su [65, 67].

4. **Četvrto poglavlje** obrađuje triangulacije sa aspekta zapisivanja (pridruživanja odgovarajuće notacije triangulacijama) i njihovog skladištenja. Date su nove metode sa ciljem uštede memorijskog prostora. Napravljena je veza između problema triangulacije poligona i kombinatornih problema kao što su *ballot* problem i problem puteva u mreži (*lattice path*).

   Prva tehnika zapisivanja je nazvana *ballot notacija*, i nju čine dva inverzna algoritma (iz *ballot* zapisa u grafički prikaz triangulacije i obrnuto). Navedena je mogućnost primene steka u obradi *ballot* zapisa, a sve u cilju dobijanja boljih rezultata kada su u pitanju dva važna resursa: vreme i memorija. Rezultati implementiranih metoda ukazuju na poboljšanje u konstrukciji i notaciji triangulacija u odnosu na postojeći algoritam.

   Druga metoda, predstavljena u ovom poglavlju, nazvana je *alfanumerička notacija (AN)*. Napravljena je komparativna analiza sa postojećom notacijom koja je bazirana na *uparenim zagradama (BP - balanced parentheses)*. Data su dva algoritma koja realizuju dve transformacije (iz BP zapisa u AN i obrnuto). Dobijeni eksperimentalni rezultati pokazuju znatnu uštedu memorije primenom AN metode u procesu skladištenja.

   Naučni radovi autora na kojima se bazira četvrto poglavlje su [49, 55].

5. **Peto poglavlje** se odnosi na objektno-orijentisanu analizu i dizajn algoritama za triangulaciju poligona. Problem triangulacije poligona je analiziran sa tri aspekta: pristupa direktnog razvoja (*forward engineering*), sa aspekta povratne analize (*reverse engineering*) i sinhronizacije oba pristupa (*round-trip engineering*). Prvi pristup je realizovan *UML* tehnikom kroz statičke, dinamičke i fizičke modele. Date su mogućnosti

generisanja *Java* izvornog koda na osnovu razvijenih *UML* modela. U drugom pristupu analize problema stavljen je akcenat na obrnuti (reverzni) inženjering koji se odnosi na tumačenje i vizuelizaciju izvornog koda primenom naprednih razvojnih okruženja (*Visual Paradigm for UML, NetBeansUML*). Treći pristup se odnosi na sinhronizaciju *Java* izvornog koda i *UML* modela.

Peto poglavlje se bazira na objavljenim naučni radovima [56, 66]. Pored navedenih radova koji se u potpunosti uključuju, neki delovi i rezultati iz radova [57, 69] se delimično navode u ovoj disertaciji.

6. U poslednjem poglavlju su data zaključna razmatranja i kratak pregled rezultata izloženih poglavlja, kao i budući pravci istraživanja.

Praktični deo ove disertacije se sastoji od osam aplikacija: šest u *Javi*, jedna u *Python*-u i jedna u *C++*-u. Razvijen je autorski set aplikacija za rešavanje problema triangulacije poligona preko novih metoda za generisanje, zapisivanje i skladištenje. Neki ključni delovi izvornih kodova ovih aplikacija su dati u prilogu disertacije, a kompletne aplikacije u vidu izvršnih programa se mogu preuzeti sa web adrese:

http://muzafers.uninp.edu.rs/triangulacija.html

Na kraju disertacije su navedeni spiskovi svih algoritama, tabela i slika.

Poglavlje 1

Uvodna razmatranja

Računarska geometrija je sastavni deo matematike i bavi se algoritamskim rešavanjem geometrijskih problema. Ova oblast se smatra i granom računarskih nauka. Nastala je u ranim sedamdesetim godinama prošlog veka kao rezultat rešavanja geometrijskih problema pomoću računara. Od samog početka, računarska geometrija je povezivala različite oblasti nauke i tehnike, kao što su teorija algoritama, kombinatorna i Euklidova geometrija, ali je uključivala i strukture podataka i optimizaciju. Danas, računarska geometrija ima veliku primenu u računarskoj grafici, vizuelizaciji, 3D modelovanju itd.

Geometrija zauzima važnu ulogu u oblasti računarske grafike. Može se reći da se računarska grafika razvila do ogromnog stepena zahvaljujući njoj. Računarska grafika se može smatrati graničnom oblašću računarstva i geometrije. Ona se bavi vernim prikazivanjem geometrijskih objekata, uzimajući u obzir ograničenja izlaznih uređaja računara. Nezamislivo je danas baviti se grafikom bez geometrije. U pozadini nekog prikaza (slike, animacije) kriju se složeni proračuni i teorije iz oblasti geometrije koje su već odavno potvrđene i razvijene, ali koje su sa aspekta primene u savremenim informacionim tehnologijama još na početku.

Jedan od važnijih problema računarske geometrije je triangulacija poligona. Primenjuje se u postupku dobijanja trodimenzionalnih prikaza objekata iz skupa tačaka. Jedan od problema izučavanja u ovom postupku je brzina pronalaženja triangulacija poligona u svim mogućim varijantama, jer se sa povećanjem broja temena poligona drastično povećava i broj različitih triangulacija. Pored toga, sa aspekta skladištenja, potrebno je obezbediti jedinstven sistem zapisivanja svih triangulacija koji omogućava uštedu memorijskog prostora.

Predmet istraživanja u ovoj disertaciji je analiza i testiranje metoda za rešavanje problema triangulacije konveksnog poligona, sa aspekta konstrukcije (generisanja u grafičkom obliku) i skladištenja (predstavljanja u obliku zapisa odnosno određene notacije).

Ova doktorska disertacija daje nove metode i tehnike u rešavanju problema triangulacije poligona, koje su implementirane primenom aktuelnih razvojnih okruženja i programskih jezika koji su danas najkorišćeniji u svetu. O aktuelnosti oblasti koja je obrađena u ovoj disertaciji govori veliki broj naučnih i stručnih radova koji su prezentovani na međunarodnim

i domaćim konferencijama i objavljeni u naučnim časopisima. U disertaciji su analizirani postojeći algoritmi iz oblasti računarske geometrije [9, 10, 23, 29, 30] ali su predstavljene i nove metode i algoritmi za generisanje triangulacija konveksnog poligona [65, 67] i nove metode za notaciju triangulacija poligona [49, 55].

Tema disertacije je multidisciplinarna jer dotiče oblasti računarske grafike, geometrije, programiranja i objektno-orijentisane analize i dizajna (OOAD) algoritama. Cilj je da se na osnovu teorijskih razmatranja novih metoda i njihove praktične primene ukaže na značajne mogućnosti unapređenja postojećeg stanja iz ove oblasti, primenom objektno-orijentisanog programiranja i modeliranja.

Napredna razvojna okruženja koja se primenjuju u implementacijama (*Java NetBeans IDE, VP for UML*) predstavljaju alate koji omogućavaju realizaciju metoda u cilju dobijanja njihovih eksperimentalnih rezultata. *Java* predstavlja kombinaciju najboljih elemenata uspešnih programskih jezika koji su joj prethodili, kombinovanih s novim konceptima u cilju postizanja efikasnijeg programiranja. Pored programiranja u *Javi*, problem triangulacije je u ovoj disertaciji povezan sa modernom tehnologijom objektno-orijentisanog softverskog inženjerstva, na nivou na kome se ona u svetu primenjuje u poslednjih nekoliko godina.

Na osnovu postavljenih ciljeva, definisani su sledeći zadaci istraživanja:

1. Najpre je potrebno utvrditi kakve su mogućnosti *Jave* (kao alata koji ćemo koristiti u implementaciji metoda) kada je u pitanju posedovanje gotovih paketa i klasa za rad sa računarskom geometrijom. Zatim, potrebno je utvrditi koje su prednosti ovog jezika u poređenju sa drugim objektno-orijentisanim programskim jezicima sa aspekta brzine generisanja i mogućnosti efikasnog skladištenja.

2. Napraviti komparativnu analizu datih novih metoda za generisanje sa postojećim tehnikama i algoritmima, i uporediti dobijene rezultate. Takođe, utvrditi procenat uštede memorije u primeni novih metoda za notaciju i skladištenje triangulacija u odnosu na druge postojeće tehnike zapisivanja.

3. Ispitati efekat primene objektno-orijentisane metodologije u analizi dobijenih implementacija i utvrditi prednosti i nedostatke pristupa sinhronizacije direktnog i obrnutog inženjeringa u cilju poboljšanja softverskih rešenja za date nove metode.

Da bi se realizovali ciljevi i zadaci istraživanja, korišćene su sledeće naučne metode i postupci:

- Metodom teorijske analize su ispitivana dosadašnja teorijska znanja o problemu triangulisanja poligona i analizirani su postojeći algoritmi i tehnike.

- Komparativnom metodom su upoređivani rezultati postojećih metoda i novih predloženih metoda rešavanja problema.

- Eksperimentalnom primenom smo dobili rezultate koji pokazuju efikasnost novih metoda (u brzini generisanja triangulacija ili uštedi memorijskog prostora pri skladištenju).

- Metoda generalizacije je primenjena u analizi određenog broja slučajeva a na osnovu matematičkih metoda gde se dolazi do uopštene tvrdnje koja važi za sve slučajeve. Metodom specijalizacije su predstavljeni određeni slučajevi u obliku konkretnog primera koji prati korake algoritma.

- Metode dedukcije i indukcije su korišćene u toku eksperimentalnog ispitivanja, gde se nakon dobijenih rezultata formirao zaključak o novim tehnikama i o mogućnostima njihove primene.

Disertacija se bazira na autorskim naučnim radovima [48, 49, 53, 54, 55, 56, 65, 66, 67] koji su objavljeni (ili su u proceduri) u međunarodnim naučnim časopisima. Rezultati disertacije se mogu svrstati u tri kategorije. Prvu kategoriju čine rezultati koji se odnose na nove metode koje se primenjuju u generisanju triangulacija konveksnog poligona. Drugu kategoriju čine rezultati koji se odnose na nove metode za notaciju triangulacija. Treću kategoriju čine rezultati koji se odnose na novu *OOAD* metodologiju, gde se primenom naprednih okruženja daje model za analizu i dizajn algoritama u računarskoj geometriji.

1.1 Pojam Katalanovih brojeva

Katalanovi brojevi (C_n) predstavljaju niz brojeva koji se prvenstveno koriste u geometriji, ali se ovaj niz pojavljuje i kao rešenje velikog broja kombinatornih problema. Prvi put ih je otkrio Leonard Ojler (*Leonhard Euler*, 1707-1783) tražeći opšte rešenje za broj različitih načina na koji se jedan mnogougao može podeliti na trouglove. Pritom je trebalo voditi računa da se ne koriste dijagonale mnogougla koje se međusobno seku.

Međutim, ovi brojevi su ipak dobili ime po belgijskom matematičaru Ežen Šarl Katalanu (*Eugene Charles Catalan*, 1814–1894) koji je otkrio vezu između ovih brojeva i problema korektnih nizova n parova zagrada.

Katalanovi brojevi se izračunavaju po sledećoj formuli [39]:

$$\begin{aligned} C_n &= \frac{(2n)!}{(n+1)!n!} \\ &= \frac{1}{n+1}\binom{2n}{n}, n \geq 0 \end{aligned} \tag{1.1}$$

gde važi da je n broj trouglova na koje se može podeliti dati poligon.

Često se koristi i sledeći alternativni izraz za definiciju Katalanovih brojeva:

$$\begin{aligned} \binom{2n}{n} - \binom{2n}{n+1} &= \frac{(2n)!}{(n!)^2} - \frac{(2n)!}{(n-1)!(n+1)!} \\ &= \frac{(2n)!}{n!(n+1)!} \\ &= C_n \end{aligned} \tag{1.2}$$

Katalanovi brojevi se još mogu izraziti i Segnerovom rekurzivnom formulom [40] kao i Bertrandovom *ballot* teoremom [22]. Tabela 1.1 sadrži Katalanove brojeve za vrednosti $n \in \{1, 2, ..., 30\}$, koji se izračunavaju po formuli (1.1) ili (1.2).

Tabela 1.1: Vrednosti prvih 30 Katalanovih brojeva

n	C_n	n	C_n	n	C_n
1	1	**11**	58,786	**21**	24,466,267,020
2	2	**12**	208,012	**22**	91,482,563,640
3	5	**13**	742,900	**23**	343,059,613,650
4	14	**14**	2,674,440	**24**	1,289,904,147,324
5	42	**15**	9,694,845	**25**	4,861,946,401,452
6	132	**16**	35,357,670	**26**	18,367,353,072,152
7	429	**17**	129,644,790	**27**	69,533,550,916,004
8	1,430	**18**	477,638,700	**28**	263,747,951,750,360
9	4,862	**19**	1,767,263,190	**29**	1,002,242,216,651,360
10	16,796	**20**	6,564,120,420	**30**	3,814,986,502,092,300

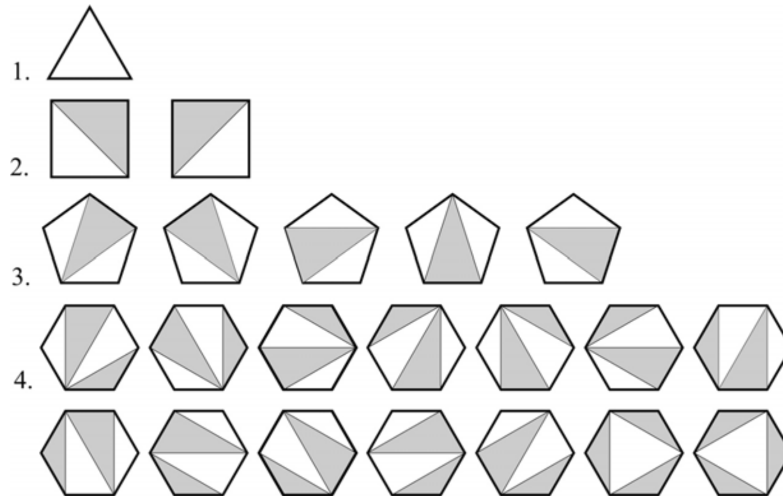
Katalanovi brojevi su veoma značajni u kombinatorici, pa se mnogi kombinatorni problemi zasnivaju na tom nizu (kao npr. *ballot* problem, problem puteva u mreži, problem uparenih zagrada itd.). U radovima [8, 17, 19, 28, 38, 63] su navedene konkretne primene ovih brojeva u rešavanju nekih kombinatornih problema i problema u računarskoj geometriji. Knjiga [68] sadrži skup zadataka koji opisuju preko 60 različitih interpretacija Katalanovih brojeva. Takođe, [39] daje osnovna svojstva, kao i konkretne primene i probleme koji se zasnivaju na ovim brojevima.

1.2 Pojam triangulacije poligona

Triangulacija poligona je istorijski veoma star problem koji je doveo do otkrića Katalanovih brojeva. Pri određivanju svih triangulacija poligona mora biti razmotren i oblik poligona. Ovo čini problem izračunavanja veoma teškim. Problem može biti smanjen tako što se ograničavamo na proračune vezane za konveksne poligone. Za konveksne poligone sve dijagonale su unutrašnje dijagonale. U ovom slučaju broj triangulacija konveksnog poligona je nezavisan od oblika i može biti jedinstveno okarakterisan brojem temena n.

Pod triangulacijom konveksnog poligona podrazumevamo razlaganje unutrašnjosti poligona na trouglove, međusobno nepresecajućim unutrašnjim dijagonalama. Kod ovog problema se zapravo razmatra broj triangulacija, gde je moguća maksimalna podela konveksnog poligona na $n - 2$ trougla. U radovima [23, 45, 46] su predstavljeni neki načini rešavanja ovog problema.

Triangulacija poligona sa n temena zahteva podelu na trouglove sa $n-3$ unutrašnjih dijagonala koje se ne ukrštaju (Slika 1.1).



Slika 1.1: Triangulacija konveksnog poligona za $n \in \{3, ..., 6\}$

Konveksni poligon sa n temena ćemo označiti sa P_n. On je opisan nizom temena v_1,v_2, ..., v_n. Unutrašnja dijagonala koja povezuje temena v_i i v_j je označena sa $\delta_{p,q}$. Takođe, i stranice poligona se smatraju dijagonalama pa tako $\delta_{i,i+1}$ predstavlja ivicu v_iv_{i+1}. Skup triangulacija poligona P_n je obeležen sa $\mathcal{T}_n$ a sa τ_n označićemo određenu triangulaciju iz skupa $\mathcal{T}_n$. Koristićemo oznaku $\deg(i)$ za stepen temena i u triangulaciji. Sa T_i ćemo označiti ukupan broj triangulacija u skupu $\mathcal{T}_n$.

Sada ćemo uspostaviti relaciju između triangulacija poligona i Katalanovih brojeva. Po Ojlerovoj postavci problema triangulacije poligona [39] važi sledeća jednakost:

$$T_n = C_{n-2}, n \geq 3 \tag{1.3}$$

gde je, u ovom slučaju, n broj temena poligona.

Na primer, broj triangulacija petougla (T_5) određuje vrednost Katalanovog broja $C_3 = 5$, za šestougao (T_6) važi $C_4 = 14$ itd. (videti Tabelu 1.1).

Dakle, vrednost Katalanovog broja C_{n-2} određuje broj triangulacija koji odgovara poligonu P_n. Na osnovu formule za dobijanje vrednosti Katalanovih brojeva (1.1) možemo definisati vrednost T_n, za $n \geq 3$:

$$\begin{aligned} T_n &= \frac{1}{n-1}\binom{2n-4}{n-2} \\ &= \frac{(2n-4)!}{(n-1)!(n-2)!} \end{aligned} \tag{1.4}$$

Sledi da vrednost T_{n+2} možemo izraziti kao:

$$\begin{aligned} T_{n+2} &= \frac{2}{n+1}\binom{2n-1}{n} \\ &= \frac{1}{n+1}\binom{2n}{n} = C_n \end{aligned} \tag{1.5}$$

što je ekvivalentno sa (1.3).

Triangulacija konveksnih poligona je aktuelan problem koji se javlja u dvodimenzionalnoj računarskoj geometriji. Tehnika triangulacije u računarskoj geometriji je najčešća metoda panelizacije. Najlakši način segmentacije i ravnanja dvostruko zakrivljene površi je putem mreže trouglova. Prednost trougla kao geometrijskog tela je što je površina između tri tačke uvek ravna. Prednosti ovakve primene su: mala odstupanja od izvornog oblika, dobra strukturalna svojstva i mogućnost oblaganja složenih slobodnih formi. Triangulacija poligona nalazi svoju primenu i u geoinformacionim sistemima, kao i u procesu digitalnog modeliranja terena [32].

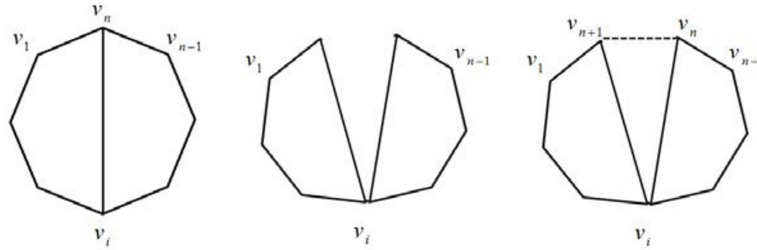
Triangulacija poligona ima mnogo primena u računarskoj grafici i koristi se u pretprocesnoj fazi broja netrivijalnih operacija prostog poligona. Ona omogućava da se iz skupa tačaka dobije trodimenzionalni prikaz objekata i omogućava mehanizam za tzv. *glačanje trodimenzionalnih figura*. Pa tako, triangulacija poligona ima široku primenu pri modelovanju 3D objekata. Ovaj postupak je veoma bitan za brzinu, kvalitet i rezoluciju prikaza objekata.

1.3 Jedan algoritam za triangulaciju konveksnih poligona

Triangulacija poligona spada u grupu poznatih problema koji su karakteristični za tehniku dinamičkog programiranja. Ovom tehnikom drastično možemo smanjiti složenost algoritma, sa glavnom idejom da izbegnemo izračunavanje iste stvari dva puta, pa se ubrzanje računanja postiže memorisanjem određenih međurezultata. Kasnije se ti isti rezultati ne moraju ponovo izračunavati, već se isti samo koriste za nova izračunavanja. Dakle, tehnika rešavanja jednog problema dinamičkim programiranjem se svodi na rekurzivno rešavanje tih nezavisnih potproblema.

Algoritam koji su dali autori Hurtado i Noj u [29] (u daljem tekstu *Hurtado-Noy* algoritam) se zasniva na generisanju triangulacija konveksnog poligona P_n na osnovu triangulacija poligona P_{n-1}. Njihova procedura se sastoji od tzv. "cepanja" dijagonala poligona (kako unutrašnjih dijagonala tako i spoljašnjih ivica poligona) sve do najvećeg označenog temena ($n-1$ za P_{n-1}). Ako se izvrši cepanje dijagonala $\delta_{i,n-1}, i \in \{1, 2, \ldots, n-2\}$ u rastućem redosledu od i, dobićemo uređene triangulacije za P_n.

Postupak "cepanja" dijagonale poligona je prikazan na Slici 1.2.



Slika 1.2: Prostupak "cepanja" dijagonale i konstrukcija potomka

Autori su definisali stablo triangulacija gde su sve triangulacije poligona P_n iz skupa $\mathcal{T}_n$ uređene na n-tom nivou stabla triangulacija. Svaka triangulacija na n-tom nivou ima "roditelja" u skupu $\mathcal{T}_{n-1}$ i dva ili više "potomaka" u skupu $\mathcal{T}_{n+1}$. Potomci istog roditelja se tretiraju kao "braća" i oni su uređeni u određenom redosledu. Na osnovu toga je definisano uređenje koje se odnosi na sve triangulacije u skupu $\mathcal{T}_n$.

Uzmimo triangulaciju $\tau_{n-1} \in \mathcal{T}_{n-1}$ zadovoljavajući $\deg(n-1) = l$ za τ_{n-1}^l. Pretpostavimo da su incidentne dijagonale sa temenom $n-1$ sortirane, počevši od $\delta_{1,n-1}$, po principu suprotno kretanju kazaljke na satu. Označimo k-tu dijagonalu u ovom redosledu sa $\delta_{i_k,n-1}$, gde k uzima vrednost iz skupa $k \in \{1, \ldots, l\}$. Sledi da je broj incidentnih dijagonala sa $n-1$ koje prethode $\delta_{i_k,n-1}$ jednak $k-1$.

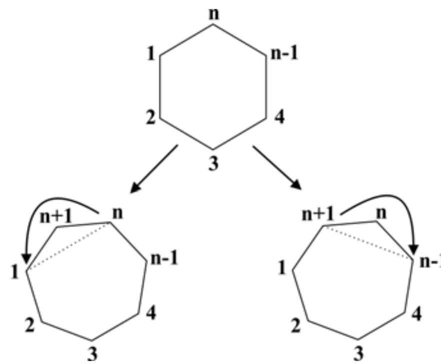
Potomci za τ_{n-1}^l se izvode tako što se "cepaju" incidentne dijagonale sa temenom $n-1$

$$\delta_{i_k,n-1},\ k = 1, \ldots, l,\ i_k \in \{1, \ldots, n-2\},\ 1 = i_1 < i_2 < \cdots < i_l = n-2.$$

Ako podelimo dijagonalu $\delta_{i_k,n-1}$ onda dobijamo potomka $S^{i_k}(\tau_{n-1}^l)$. Tada sledi da su potomci triangulacije τ_{n-1}^l

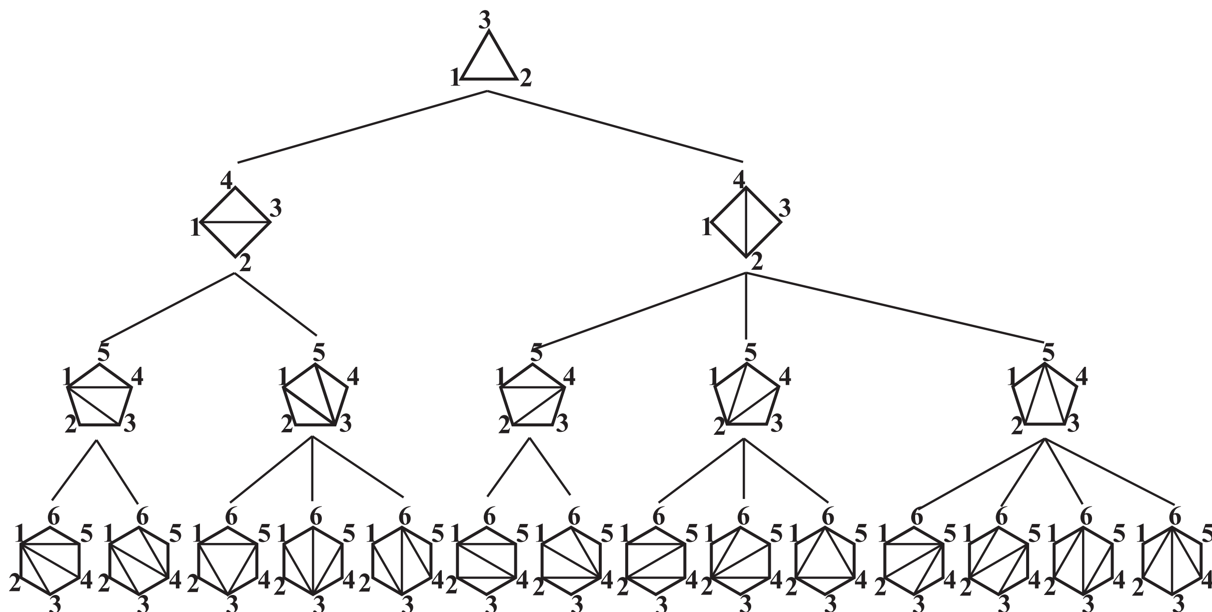
$$S^{i_1}(\tau_{n-1}^l), S^{i_2}(\tau_{n-1}^l), \ldots, S^{i_l}(\tau_{n-1}^l).$$

Cepanje dijagonale $\delta_{i_k,n-1}$ proizvodi potomka $S^{i_k}(\tau_{n-1}^l)$ sa $\deg(n) = 2 + k - 1$.



Slika 1.3: Primer generisanja dva potomka: $S^1(\tau_{n-1}^l)$ i $S^{n-1}(\tau_{n-1}^l)$

Na Slici 1.4 je prikazana hijerarhija na stablu triangulacija od $n = 3$ do $n = 6$.



Slika 1.4: *Hurtado-Noy* hijerarhija

Sada ćemo definisati algoritam koji realizuje opisanu proceduru iz rada [29], a naveden je u našem radu [67]. Algoritam 1.3.1 na ulazu zahteva prirodan broj n i skup triangulacija iz prethodnog nivoa $\mathcal{T}_{n-1}$.

Svaka triangulacija je opisana kao struktura koja sadrži $2n - 5$ parova temena koji predstavljaju dijagonale poligona P_{n-1} (gde se pod dijagonalama podrazumevaju unutrašnje dijagonale i spoljašnje ivice poligona).

Algoritam 1.3.1 *Hurtado-Noy* algoritam

Ulaz: Pozitivni ceo broj n i skup triangulacija $\mathcal{T}_{n-1}$ poligona P_{n-1}.

1: Proverava strukturu koja sadrži $2n - 5$ parova temena koje odgovaraju određenoj triangulaciji τ_{n-1}, gde se traže parovi $(i_k, n - 1)$, $i_k \in \{1, 2, \ldots, n-2\}$ (gornja granica za k je između 2 i $n - 2$), tj. dijagonale koje su incidentne sa temenom $n - 1$.

2: Za svako i_k generiše potomka $S^{i_k}(\tau_{n-1})$ tako što se vrši transformacija $\delta_{i_l,n-1} \rightarrow \delta_{i_l,n}$, $i_l < i_k$, $0 \leq l \leq n - 3$, i uvodi novi par temena $\delta_{i_k,n}$ i $\delta_{n-1,n}$.

3: Uzima sledeći i_k, ako postoji, i ide na Korak 2.

4: Nastavlja gore navedenu proceduru za sledeću triangulaciju poligona P_{n-1} (tj. strukturu sa $2n - 5$ parova temena), ako postoji. U suprotnom se završava.

U narednom poglavlju disertacije su predstavljene implementacije za *Hurtado-Noy algoritam* u tri programska jezika (*Java, C++ i Python)* i urađena je njihova komparativna analiza.

Poglavlje 2

Implementacija algoritama računarske geometrije u *Javi*

U ovom poglavlju su navedene prednosti objektno-orijentisanog programiranja i modeliranja kroz primenjene jezike: *Java* i *UML*. Ispitane su mogućnosti programskog jezika *Java* za implementaciju algoritama računarske geometrije, kroz primenu postojećih biblioteka i programskih interfejsa (*Java 2D, 3D, Open GL, GeometryInfo, Triangulator*).

Analizirani su postojeći *Java* paketi koji su korišćeni za implementaciju novih metoda za triangulaciju poligona. Data je uporedna analiza implementacija u programskim jezicima *Java, Python i C++* za *Hurtado-Noy* algoritam, gde su analizirane mogućnosti navedenih jezika u implementaciji algoritama računarske geometrije. Objavljeni naučni radovi autora na kojima se bazira ovo poglavlje su [53, 54, 48].

2.1 Prednosti objektno-orijentisanog programiranja i modeliranja

Objektno-orijentisano programiranje uključuje sposobnost jednostavnog korišćenja delova *izvornog koda* u različitim programima, štedeći vreme u analizi, dizajnu, razvoju i testiranju. Objektno-orijentisani koncepti pružaju mogućnost za kreiranje fleksibilnih i modularnih programa, kroz primenu glavnih svojstava i elemenata, kao što su: objekti, klase, nasleđivanje, polimorfizam itd.

Java je objektno-orijentisani programski jezik koji je razvila kompanija *Sun Microsystems* početkom devedesetih godina. Mnogi koncepti *Jave* su bazirani na jeziku *Oberon.* Izbačen je koncept modula i uveden koncept klasa iz objektno-orijentisane paradigme. Kao i većina objektno-orijentisanih jezika, *Java* uključuje biblioteke klasa koje obezbeđuju rad sa osnovnim tipovima podataka, ulazom i izlazom, osnovnim Internet protokolima, kontrolama za kreiranje korisničkog interfejsa itd.

Po statistikama portala *TIOBE Programming Community Index*[1] u kategoriji popularnosti objektno-orijentisanih programskih jezika u poslednjoj deceniji, prvo mesto zauzima programski jezik *Java* (Tabela 2.1).

Tabela 2.1: Statistika popularnosti programskih jezika (OOP jezici, 2007-2013)

Objektno-orijentisani programski jezici	**Pozicija** Jul-13	2012	2011	2007
Java	**1**	**1**	**1**	**1**
Objektni-C	2	2	5	-
C++	3	3	2	2
C#	5	4	3	5
(Visual) Basic	6	5	6	3
PHP	4	6	4	4
Python	7	7	7	6
Ruby	11	8	9	8
JavaScript	10	9	8	-
Delphi/Object Pascal	18	10	10	-

Po statistikama istog portala, u kategoriji svih tipova programskih jezika (objektno-orijentisani, proceduralni, funkcionalni i logički) *Java* se u poslednjih 15 godina opet nalazi na vrhu liste (Tabela 2.2).

Tabela 2.2: Statistika popularnosti programskih jezika (svi prog. jezici, 1998-2013)

Programski jezici	**Pozicija** Jul-13	2012	2011	2008	1998
C	1	1	2	2	1
Java	**2**	**2**	**1**	**1**	**3**
Objektni-C	3	3	6	42	-
C++	4	4	3	3	2
C#	6	5	4	7	-

Bitna prednost *Jave* se ogleda u nezavisnosti od platforme. To znači da se programi pisani u njoj lako prenose sa jednog računara ili uređaja na drugi, bez obzira na različito radno okruženje. Jedini preduslov je da je na uređaju na kome se program izvršava instaliran interpretator za *Javu*, nazvan *Java* virtuelna mašina (JVM - *Java Virtual Machine*). Još jedna prednost ovog programskog jezika je i jednostavnija sintaksa. *Java* sintaksa se oslanja na programske jezike *C* i *C++*, ali ona je daleko jednostavnija od njihove. U Javi nema pokazivača, nizovi su realni objekti, a upravljanje memorijom je automatsko.

[1] *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*

Bitna karakteristika ovog programskog jezika je i pogodnost za rad u mrežnom okruženju. Podržava konkurentno programiranje pomoću niti (*threads*), što znači da *Java* programi mogu izvršavati više zahteva istovremeno [35, 48].

Da bi implementacije metoda bile kompletne, primenjena je OOAD metodologija (objektno-orijentisana analiza i dizajn). Ova metodologija služi za specifikaciju, vizuelizaciju, konstrukciju i dokumentaciju razvoja sistema. Korišćen je *UML* (*Unified Modeling Language*), univerzalni jezik namenjen za objektno-orijentisano modeliranje. Prednost *UML*-a se ogleda u tome što on predstavlja eksplicitnu vezu između objektno-orijentisanih koncepata i izvršnog koda. *UML* standard se primenjuje kod OO pristupa i predviđa odgovarajuće poglede na sistem (opis sistema sa statičkog/strukturnog, dinamičkog i fizičkog aspekta). Ovaj jezik se koristi za analizu i vizuelizaciju algoritma triangulacije poligona u više dimenzija i nivoa detalja (navedeno u petom poglavlju disertacije). Ovakav način analize se može primeniti na mnoge algoritme i probleme [60, 61, 66].

2.2 *Java* biblioteke za GUI i računarsku geometriju

Algoritmi računarske geometrije kreću se od prostih ispitivanja kolinearnosti tačaka, njihove međusobne pozicije, pa sve do vrlo složenih pronalaženja najmanjeg konveksnog omotača skupa tačaka, generisanja Voronojevih dijagrama, triangulacija poligona itd. Neki od radova koji se bave algoritmima računarske grafike su [4, 62, 27]. Da bi ti algoritmi radili efikasno, potrebne su pogodne strukture podataka za reprezentaciju tačaka, segmenata, skupa tačaka i ostalih geometrijskih objekata.

2.2.1 *AWT* i *Swing* paketi

Aplikacije u *Javi* razvijene u okviru disertacije imaju grafički korisnički interfejs (GUI) gde su korišćeni paketi iz *AWT* i *Swing* biblioteke. Osnovni elementi koji su potrebni za kreiranje GUI-ja se nalaze u dva paketa: `java.awt` i `javax.swing`.

Java programski jezik u svojoj početnoj verziji je posedovao biblioteku komponenata za izgradnju grafičkog korisničkog interfejsa (GUI) zvanu *Abstract Window Toolkit (AWT)*. U pitanju je biblioteka koja se zasniva na korišćenju komponenata korisničkog interfejsa koje su dostupne na platformi na kojoj se program pokreće. To znači da je implementacija *AWT* komponenata različita za svaki operativni sistem (npr. u Windows distribuciji *Java* virtuelne mašine koriste *awt.dll* datoteku)[76].

Važan *Java* paket za rad sa geometrijskim oblicima je `geom` (`java.awt.geom.*`). On omogućava definisanje i izvođenje operacija nad objektima koji se odnose na rad sa dvodimenzionalnom geometrijom. Paket `geom` iz biblioteke *AWT* je korišćen u realizaciji klasa za implementaciju metoda za generisanje triangulacija (predstavljene u trećem poglavlju disertacije). Kao proširena verzija *AWT*-a, nastala je biblioteka komponenata za GUI nazvana *Swing* i to ime se zadržalo i kasnije. Paket `event` iz *SWING* biblioteke je korišćen u realizaciji regulisanja akcije koja pokreće proceduru generisanja triangulacija (`javax.swing.event.*`).

Većina klasa paketa `javax.swing` koje definišu GUI elemente, tzv. *Swing* komponente, obezbeđuju unapređene alternative za komponente definisane klasama iz `java.awt` paketa. *Swing* klase su fleksibilnije od odgovarajućih klasa iz paketa *AWT* pošto su u potpunosti implementirane u Javi [24].

Praktični deo ove disertacije se sastoji od osam aplikacija: šest u *Javi*, jedna u *Python*-u i jedna u *C++*-u. Razvijen je autorski set aplikacija za rešavanje problema triangulacija preko novih metoda za generisanje ali i zapisivanje i skladištenje istih. Za izradu svih *Java* aplikacija za realizaciju metoda predstavljenih u ovoj disertaciji korišćeno je okruženje *Java NetBeans IDE* (*Integrated Development Environment*). Ovo okruženje je pogodno, između ostalog, i za implementaciju algoritama računarske geometrije u formi *Java apleta* ili aplikacija [59]. U okviru *NetBeans* platforme nalaze se komponente za pravljenje GUI-ja. Te komponente su unapređene *Java Swing* komponente. Platforma nudi mehanizam za uključivanje korisničkih modula radi proširivanja funkcionalnosti celokupne platforme [31, 15].

2.2.2 Otvorena biblioteka *Java OpenGL*

Java otvorena biblioteka za grafiku (JOGL - *Java Open Graphics Library* ili *JOpenGL*) je biblioteka za rad sa geometrijskim oblicima i sadrži pakete i klase koji su važni za programiranje u računarskoj geometriji i grafici. *JOpenGL* je *Java* veza za *OpenGL 3D* grafički *API* (*Application programming interface*)[16] a podržava i integraciju sa *Swing* komponentama.

Java 2D sistem podržava dva nezavisna koordinatna prostora i to korisnički prostor za specifikaciju grafičkih primitiva i prostor za konkretni izlaz. Korisnički prostor se koristi za tumačenje koordinata svih primitiva koje do sistema dođu putem *Java* 2D API-ja [36].

Java 3D je *API* vrlo visokog nivoa, namenjen pisanju *Java applet*-a i aplikacija koje omogućavaju rad sa 3D interaktivnim sadržajem. *Java 3D* omogućava potpuno objektno-orijentisan pristup 3D grafici na visokom nivou. Bitno je naglasiti da *Java 3D* nije integrisani deo, već standardno proširenje *Java* platforme.

Paket `geometry` se odnosi na *Java* 3D API (`Package com.sun.j3d.utils.geometry`).

Implementacija *Java 3D* verzije 1.6 se sastoji od tri paketa: *OpenGL*, *DirectX* i *JOpenGL* [11, 12, 64]. *JOpenGL* zadržava fokus na 3D prikaz sa pomoćnim bibliotekama koje olakšavaju kreiranje geometrijskih primitiva.

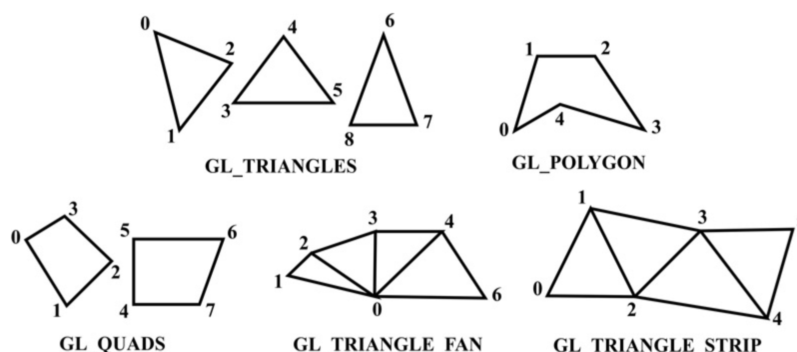
2.2.3 Klase i paketi za triangulaciju poligona

U ovoj sekciji je ispitano kakve su mogućnosti *Jave* kada je u pitanju posedovanje gotovih paketa i klasa za rad sa računarskom geometrijom, konkretnije u postupku triangulacije poligona. Dva paketa za ovaj algoritam računarske geometrije iz *Java* 3D su:

`com.sun.j3d.utils.geometry` (*GeometryInfo, Triangulator)* i
`org.j3d.geom` (*TriangulationUtilis, SeidelTriangulator).*

Klase `GeometryInfo` i `Triangulator` su iz paketa `geometry`. Klasa `GeometryInfo` sadrži operacije za rad sa nizovima temena za geometrijske objekte:

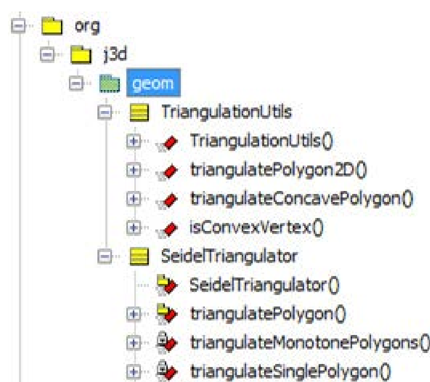
- TRIANGLE_ARRAY uzima skup od tri temena koji formira trougao,
- GL_TRIANGLES kreira niz trouglova koji formiraju triangulaciju poligona,
- POLYGON_ARRAY kreira niz temena za nekonveksni i konveksni poligon,
- GL_POLYGON kreira poligon od datog niza sa ulaza,
- QUAD_ARRAY uzima skup od četiri temena koji formira četvorougao,
- GL_QUADS kreira niz kvadrata koji formiraju kvadragulaciju poligona,
- TRIANGLE_FAN_ARRAY daje skup temena triangulacije u obliku lepeze,
- GL_TRIANGLE_FAN kreira triangulaciju od niza trouglova oko zajedničkog temena,
- TRIANGLE_STRIP_ARRAY daje skup temena triangulacije u obliku trake,
- GL_TRIANGLE_STRIP kreira triangulaciju od niza vezanih trouglova.



Slika 2.1: Neke operacije klase *GeometryInfo*

Bitno je naglasiti da klasa `GeometryInfo` ima mogućnost da iscrtava samo određen tip triangulacije (na primer, kreiranje triangulacije od niza vezanih trouglova – *STRIP* triangulacija ili kreiranje triangulacije od niza trouglova oko zajedničkog temena – *FAN* triangulacija). Zato se javila potreba za implementacijom novih tehnika i metoda koje omogućavaju efikasno generisanje svih mogućih triangulacija nekog konveksnog poligona.

Druga klasa iz paketa `Geometry` je `Triangulator`. Ovu klasu poziva `GeometryInfo` i nikad se ne koristi posebno. Klasa `TriangulationUtils` je iz paketa `org.j3d.geom` i namenjena je za rad sa poligonima sa malim brojem temena, jer koristi jednostavniji ali sporiji algoritam. Klasa `SeidelTriangulator` je iz istog paketa i predstavlja proširenje prethodne klase. Licencirana je na verziji *GNU LGPL v2.1.* Pošto klasa `TriangulationUtils` nije efikasna za poligone sa većim brojem temena, ona se koristi u kombinaciji sa klasom za implementaciju tzv. *Zajdelovog* algoritma (Raimund Seidel)[51].



Slika 2.2: Sadržaj paketa `Geom`: *TriangulationUtilis* i *SeidelTriangulator*

2.3 Prednosti *Jave* u implementaciji algoritma za triangulaciju konveksnog poligona

U prethodnoj sekciji smo predstavili mogućnosti programskog jezika *Java* u oblasti računarske geometrije, a u ovoj sekciji je ispitano na konkretnom primeru kakav je programski jezik *Java* u odnosu na svoju konkurenciju. Za potrebe komparativne analize odrađene su tri implementacije *Hurtado-Noy* algoritma: u *Javi, C++*-u i *Python*-u [53]. Razlog za biranje druga dva programska jezika je što su i oni, pored *Jave*, dva trenutno aktuelna objektno-orijentisana programska jezika (što se može videti iz Tabele 2.1).

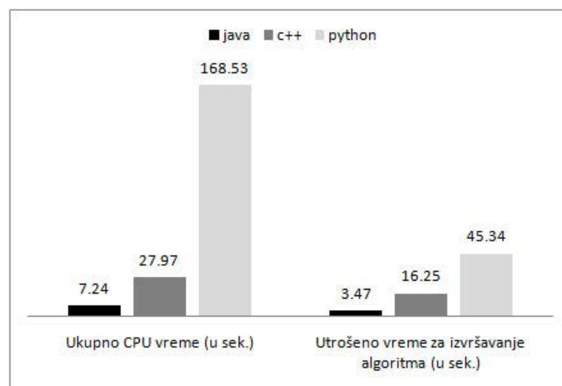
Pre komparativne analize implementacija u navedenim programskim jezicima, ispitano je da li je rađeno neko slično istraživanje za ova tri programska jezika, odnosno kakva su iskustva sa pomenutim programskim jezicima u implementaciji nekog srodnog problema u računarskoj geometriji.

Na portalu *The Computer Language Benchmarks*[2] postoje testiranja za mnoge programske jezike u implementaciji jednostavnijih problema u geometriji. Između ostalog, navedena je analiza rezultata za *Javu*, *Python* i *C++* za implementaciju algoritma za binarna stabla (*BinaryTrees algorithm*). Izabrana je analiza ovog algoritma zato što je na neki način srodan sa triangulacijama konveksnog poligona jer i on može interpretirati Katalanove brojeve [39].

Ispitivanje na ovom portalu je sprovedeno na procesoru *Intel Q6600 quad-core*. Kada je reč o CPU opterećenju, na najzahtevnijem slučaju za n=20, najviše opterećenje je zabeleženo kod *Python*-u (93% , 92% , 98%, 93%). Navedene vrednosti u zagradama se odnose na četiri jezgra procesora. U slučaju kod *C++*-a opterećenje je znatno manje (27%, 98%, 25%, 24%), kao i kod *Jave* (55%, 27%, 76%, 57%). Glavni razlog za ovakve rezultate je dinamičko alociranje memorije koje je brže kod *Jave* i *C++*-a, a i njihov I/O (*input-output*) je brži u odnosu na *Python* standardne biblioteke.

[2] *http://shootout.alioth.debian.org/*

Na grafikonu (Slika 2.3) su prikazani rezultati testiranja za tri navedena programska jezika. Uzete su prosečne vrednosti (u sekundama) za $n \in \{12, 16, 20\}$.



Slika 2.3: Merenje performansi za *BinaryTrees* algoritam u *Javi, Python*-u i *C++*-u

2.3.1 Implementacija *Hurtado-Noy* algoritma

Programska okruženja koja su korišćena za implementaciju *Hurtado-Noy* algoritma su: *Java* (NetBeans IDE 6.9.1), *Python* (Wing IDE Professional 4.0.4) i *C++* (NetBeans IDE C++ 7.0). Implementacija je odrađena kroz 6 klasa (sa istom strukturom u sva tri jezika). Važniji segmenti ovih klasa su navedeni u prilogu disertacije (Prilog II).

Tabela 2.3: Klase za implementaciju *Hurtado-Noy* algoritma

KLASE	**METODE**
Triangulation	`clear(), copyFrom()` `Draw(), DrawAll()`
GenerateTriangulations	`Vector<Node>Hurtado()` `main()`

Node	`leaves(), leftBranch()`
	`toString(), copy()`
	`getLeft(), setLeft()`
	`getRight(), setRight()`

LeafNode	`LeafNode(), copy()`
	`leaves(), leftBranch()`
	`toParen(), toString()`

Point	`Point(int x, int y)`

PostScriptWriter	`PostScriptWriter(BufferedWriter out)`
	`psHeader(), write(), Trailer()`
	`drawLine(), drawDot(), newPage()`

U Javi su korišćeni sledeći standardni paketi: `javax.swing, java.awt, awt.geom, awt.event, java.io, io.BufferedWriter` i `java.util.Vector`. U programskom jeziku *C++* su korišćeni `System, System::Collections::Generic`, dok u *Python*-u je korišćen `sys, math, time` i paket `OptionParser`.

1. Klasa `Triangulation` je zadužena za iscrtvanje triangulacija konveksnog poligona. Ova klasa sadrži dve ključne metode koje obezbeđuju grafičko prikazivanje triangulacija i njihov prikaz u formi izlazne datoteke. To su metode `Draw()` i `DrawAll()`.

   Metoda `Draw()` je zadužena za iscrtavanje pojedinačnih triangulacija konveksnih poligona. Metoda `DrawAll()` obezbeđuje izvršavanje metode `Draw()` onoliko puta koliko iznosi Katalanov broj za uneto n (izračunava se po formuli 1.4). Na kraju se dobija generisana datoteka koja povezuje sve triangulacije (detalji za obe metode su navedeni u Prilogu II-A).

2. Klasa `GenerateTriangulations` je glavna izvršna klasa i ona sadrži komponente za GUI aplikacije (Prilog II-B). Ova klasa ima dve metode: `main()` i `Hurtado()`.

3. Klase koje su zadužene za rad sa temenima (čvorovima) su: `Node` i njena klasa naslednica `LeafNode`, kao i klasa `Point` koja je zadužena za rad sa koordinatama temena poligona (Prilogu II-D). Klasa `PostScriptWriter` omogućava generisanje izlazne datoteke u *ps* formatu, sa grafičkim prikazom svih triangulacija za uneto n (Prilog II-F).

Metod `Hurtado()` generiše triangulacije konveksnog poligona u određenoj hijerarhiji. Na ulazu zahteva prirodan broj n i triangulacije iz prethodnog nivoa (Algoritma 1.3.1, opisan u uvodnim razmatranjima).

Sledi deo metode `Hurtado()` iz klase `GenerateTriangulations`:

```
public Vector<Node> Hurtado(int limit){

Vector<Vector> h = new Vector<Vector>();
    h.add(new Vector<LeafNode>());
    h.get(0).add(new LeafNode());

  for (int n=0; n < limit; n++){
     Vector<Node> level = new Vector();
    for (int q = 0; q < h.get(n).size(); q++) {
         Node t = (Node)h.get(n).get(q);
         Node s = new Node(new LeafNode(), t.copy());
         level.add(s);
      for (int k = 0; k<t.leftBranch(); k++){
          s = t.copy();
          Node r = s;
        for (int i=0; i<k; i++)
          s = s.getLeft();
          s.setLeft(new Node(new LeafNode(), s.getLeft()));
```

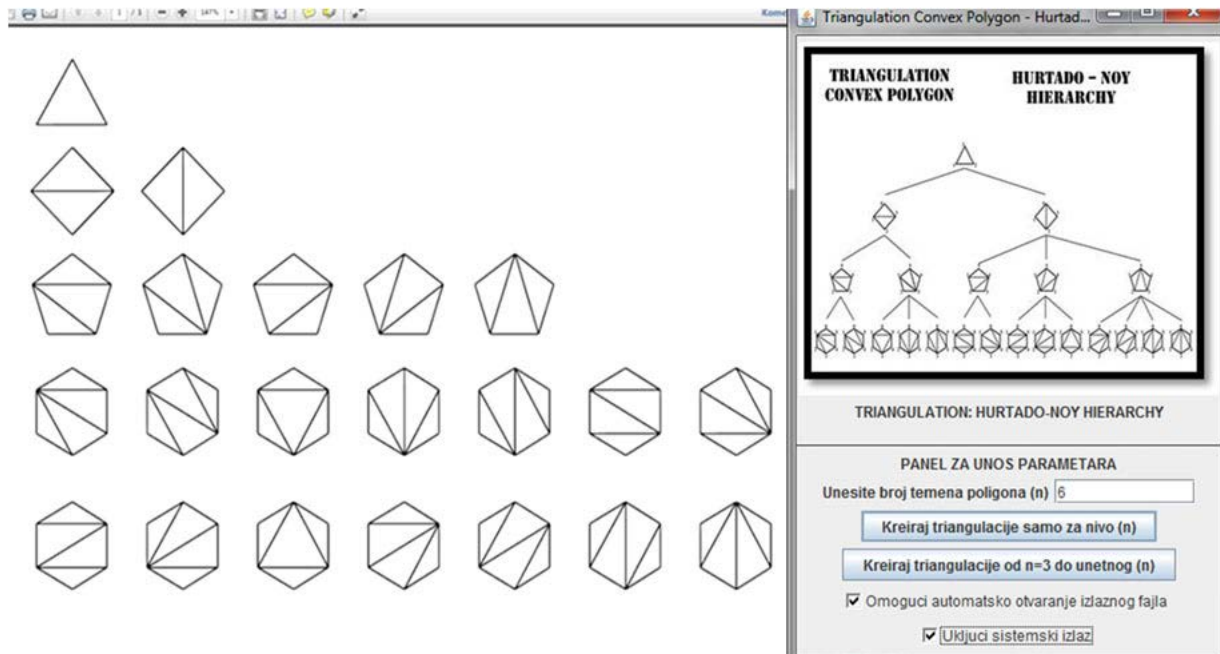
```
                level.add(r);
                }
            }
        h.add(level);
        }
        return h.get(limit);
    }
```

U izvršnoj metodi `main()` klase `GenerateTriangulations` kreira se primerak klase nad kojim se poziva metoda `Hurtado()`. Zatim se kreira jedan primerak klase `Triangulation` nad kojim se poziva metoda `DrawAll`. Ova metoda poziva metodu `Draw` C_{n-2} puta:

```
GenerateTriangulations app = new GenerateTriangulations();
Vector<Node> h = app.Hurtado(n-2);
Triangulation instanceTriangulation = new Triangulation(n,writer);
instanceTriangulation.DrawAll(hurt);
FileWriter fstream = new FileWriter(p+"-polygons.ps");
BufferedWriter out = new BufferedWriter(fstream);
PostScriptWriter writer = new PostScriptWriter(out);
```

Kao rezultat, daje se prikaz u *ps* formatu preko primerka klase `PostScriptWriter`. Da bi se triangulacije grafički prikazale, u klasi `PostScriptWriter` postoji metoda `drawLine` zadužena da u saradnji sa metodom `Draw()` prikaže sve generisane triangulacije (Prilog II-F). Kod programskog jezika *Python*, ovaj je postupak urađen preko metode `setOptionParser()`.

Na Slici 2.4 je dat GUI aplikacije i izlaz za $n \in \{3, 4, 5, 6\}$ po *Hurtado-Noy* hijerarhiji.



Slika 2.4: Generisanje triangulacija po *Hurtado-Noy* hijerarhiji za $n \in \{3, ..., 6\}$

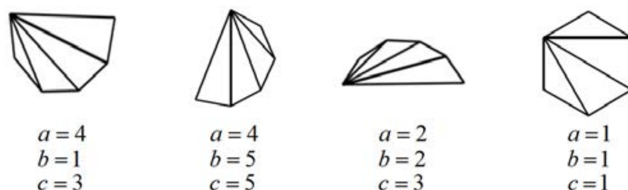
Aplikacija sadrži komponente koje omogućavaju manipulaciju nad generisanim triangulacijama. Na ovaj način, promenom standardnih parametara, možemo omogućiti triangulisanje i nepravilnih konveksnih poligona.

Sada ćemo navesti neke primere manipulacije u generisanju triangulacija, kroz odgovarajuće izmene parametara:

1. Linija kôda za iscrtavanje konveksnih poligona:

   ```
   ((a*k+c)+edges)*Math.PI/(b*edges)
   ```

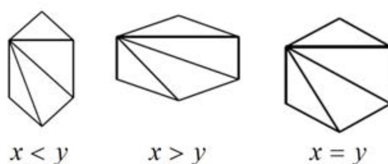
   gde se promenljive a i b koriste za formiranje oblika poligona, dok se promenljiva c koristi za rotaciju poligona. U slučaju da je $a \neq 1$ i $b \neq 1$ dobićemo nepravilne konveksne poligone:

2. Linija kôda za skaliranje poligona:

   ```
   this.sSine.add(x*Math.sin(d)); this.sCosine.add(y*Math.cos(d);)
   ```

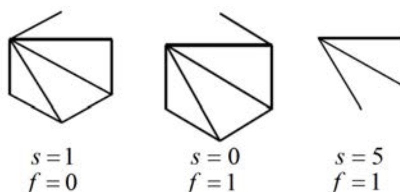
   U slučaju kad je $x \neq y$, generišu se sledeći oblici konveksnog poligona:

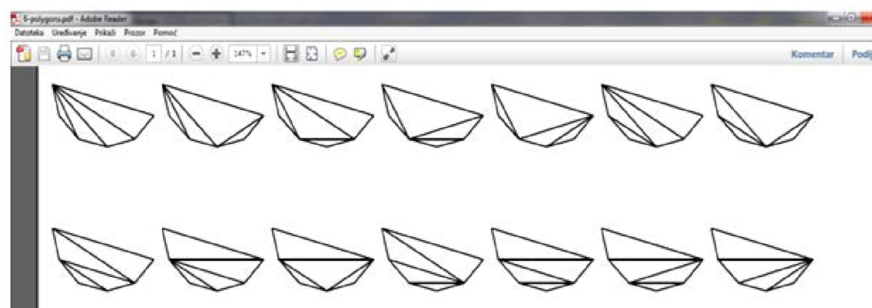
3. Metoda `Draw()` obezbeđuje eliminaciju spoljašnjih ivica poligona a da se pritom generišu samo unutrašnje dijagonale. To se postiže modifikacijom odgovarajućih vrednosti za s i f:

   ```
   for (int i = s; i<this.points.size()-f; i++)
   ```

   Neki primeri za slučajeve kada je $s \neq 0$ i $f \neq 0$:

Na ovaj način se omogućava rad Algoritma 1.3.1 i sa nepravilnim oblicima, a da se pritom dobije odgovarajuća *Hurtado-Noy* hijerarhija za skup triangulacija $\mathcal{T}_n$ (Slika 2.5).



Slika 2.5: *Hurtado-Noy* redosled triangulacija za nepravilan konveksni šestougao

Postoje još neki primeri intervencija od strane korisnika u trenutku testiranja. Postoji mogućnost izlistavanja željenog broja triangulacija, tako da metoda `DrawAll()` u tom slučaju neće imati onoliko ciklusa izvršavanja koliki je Katalanov broj C_{n-2} za n-tougao, već onoliko koliko smo uneli u polje za broj triangulacija. Na primer, ako je potrebno da za 12-ugao izlistamo nekoliko triangulacija od ukupno 16 796 kombinacija, unećemo odgovarajući parametar i metoda `DrawAll()` će izlistati toliko triangulacija.

2.3.2 Komparativna analiza implementacija (*Java, C++, Python*)

U ovoj sekciji je predstavljena komparativna analiza navedena tri programska jezika. Kroz testiranje tri aplikacije koje su kreirane za potrebe adekvatnog izbora programskog jezika, utvrdićemo koji je od navedenih jezika dobar izbor za implementaciju i nekih novih algoritama za triangulaciju poligona. Sve tri aplikacije za *Hurtado-Noy* algoritam se mogu preuzeti sa: *http://muzafers.uninp.edu.rs/triangulacija.html*.

Ispitano je vreme generisanja triangulacija za poligone sa većim brojem temena (gde je broj triangulacija veći od nekoliko desetina hiljada kombinacija), jer se u tom slučaju najbolje mogu ispitati mogućnosti navedenih programskih jezika. Za testiranje su uzete vrednosti $n \in \{10, 11, ..., 16\}$, gde je broj triangulacija redom 1.430, 4.862, ..., 2.674.440.

Testiranje je urađeno na računaru sledećih performansi: *CPU - Intel(R)Core(TM)2Duo CPU, T7700, 2.40 GHz, L2 Cache 4 MB (On-Die,ATC,Full-Speed), RAM Memory - 2 Gb, Graphic card - NVIDIA GeForce 8600M GS.* Za testiranje je korišćeno okruženje *NetBeans IDE - modul Profiler,"CPU Analyze Performance"* koji je sastavni deo *NetBeans* okruženja i koristi se za CPU testiranje kao i testiranje memorije [58]. Ovaj modul se pokreće u odeljku *Profile>Profile Main Project.*

Tabela 2.4 sadrži eksperimentalne rezultate testiranja (ukupno vreme za generisanje svih triangulacija i broj triangulacija u sekundi).

Tabela 2.4: Komparativna analiza za tri implementacije

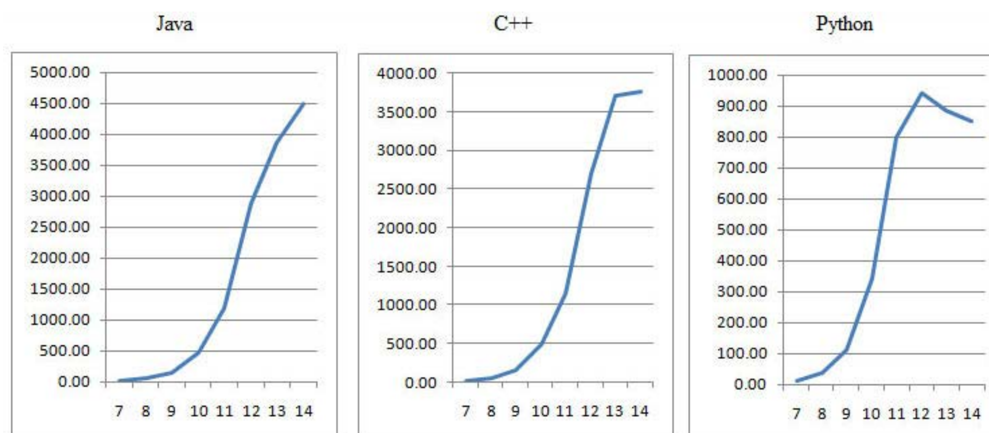
Kriterijum	***n***	***Java***	***Python***	***C++***
Vreme za generisanje triangulacija	10	2.96	4.19	2.95
	11	4.07	6.08	4.25
	12	5.81	17.83	6.23
	13	15.24	66.25	15.88
	14	46.34	244.52	55.28
	15	124.18	886.12	134.12
	16	328.16	3185.13	399.54
Broj triangulacija u sekundi	10	483.11	341.29	484.75
	11	1194.59	799.67	1144.00
	12	2890.88	942.01	2695.99
	13	3857.35	887.34	3701.89
	14	4488.82	850.70	3762.88
	15	5982.44	838.37	5539.07
	16	8149.80	839.66	6693.80

Eksperimentalni rezultati pokazuju sledeće:

(1) Pri testiranju programa u *Javi* i *C++*-u, sa povećanjem vrednosti za n može se uočiti povećanje generisanja triangulacija u sekundi, s tim što se kod *C++*-a za vrednosti $n \geq 13$ uočava blago opadanje.

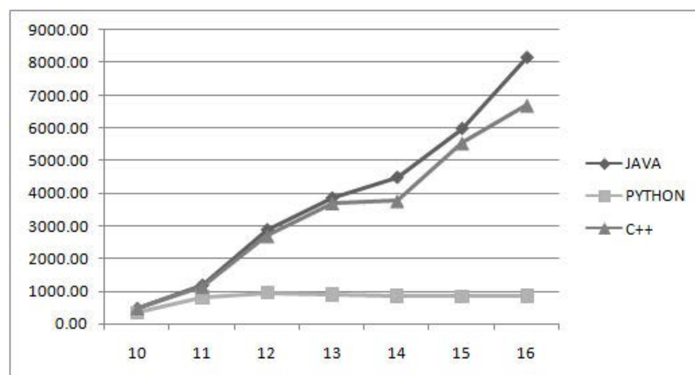
(2) Kod *Python*-a broj generisanih triangulacija u sekundi je znatno manji još od vrednosti za $n > 6$, ali se od $n > 12$ može videti da taj broj naglo opada.

Na grafikonu (Slika 2.6) su prikazani rezultati za broj triangulacija u sekundi (y koordinata) za poligone sa određenim brojem temena (x koordinata).



Slika 2.6: Broj generisanih triangulacija u sekundi za $n \in \{7, 8, \ldots, 14\}$

Grafikon (Slika 2.7) prikazuje odnos tri navedena programska jezika za broj trianulacija u sekundi, za $n \in \{10, \ldots, 16\}$. Glavni problem kod izvršavanja programa za veće vrednosti n je što sa povećanjem n u jednom momentu dolazi do opadanja broja generisanih triangulacija u sekundi. *Java* je u blagoj prednosti u odnosu na *C++*, dok je *Python* daleko iza njih.



Slika 2.7: Komparativna analiza: *Java, Python i C++*

Dobijeni rezultati pokazuju koji je od tri programska jezika najpogodniji za implementaciju algoritama iz oblasti računarske geometrije:

- Kod izvršavanja *Java* programa, *Java virtuelna mašina* rezerviše više radne memorije za svoje objekte a *Python* u trenutku formiranja primerka klase uzima određenu količinu memorije. Iz tog razloga je programskom jeziku *Python* potrebno više vremena za izvršavanje.

- Bitno je napomenuti da *C++* ima bolju produktivnost u količini memorije koju koristi, ali po ceni da se programer sam brine o upravljanju memorijom. *C++* i *Java* pružaju znatno veću brzinu izvršavanja od *Python*-a, kod kojih je interpreter uvek učitan u memoriju te se ne mora pozivati spoljašnji program [75, 44].

- Jedina prednost programskog jezika *Python* je da se odlikuje jednostavnom sintaksom i jasnoćom, pa su tako programi napisani u njemu za isti algoritam mnogo kraći.

Poglavlje 3

Metode za generisanje triangulacija konveksnih poligona

Ovo poglavlje sadrži analizu novih tehnika i metoda za generisanje triangulacija konveksnog poligona. Prva tehnika se bazira na *dekompoziciji Katalanovih brojeva* u formi suma termova $(2+i)$. Na osnovu ove ideje je uspostavljeno težinsko stablo triangulacija. Dobijeni izrazi dekompozicije se primenjuju u postupku generisanja triangulacija iz skupa $\mathcal{T}_{n-1}$ u skup $\mathcal{T}_n$.

Druga tehnika generisanja triangulacija je nazvana *blok metoda*. Generalna strategija koja je korišćena u ovoj metodi je da se glavni problem razlaže na manje potprobleme koji su međusobno zavisni. U ovoj metodi je stavljen akcenat na mogućnosti rada sa bazama podataka u *Java NetBeans* okruženju (*JDBC*).

Za potrebe dobijanja eksperimentalnih rezultata za svaku od pomenutih metoda je kreirana posebna *Java* aplikacija. Nakon toga, urađena je komparativna analiza predloženih metoda sa *Hurtado-Noy* algoritmom, gde je ukazano na prednosti novih metoda. Naučni radovi na kojima se bazira ovo poglavlje su [65, 67].

3.1 Metod dekompozicije Katalanovih brojeva

Algoritam 1.3.1 nam je poslužio kao inspiracija da razmotrimo specifičnosti dekompozicije Katalanovog broja, koje bi se mogle upotrebiti za brže generisanje triangulacija. Kao rezultat postupka dekompozicije dobićemo izraz u obliku suma termova $(2+i)$, gde je i iz opsega $0 \leq i \leq n-4$.

Izraz dekompozicije se može koristiti za generisanje skupa triangulacija $\mathcal{T}_n$ na osnovu skupa $\mathcal{T}_{n-1}$. Dobijeni izraz ukazuje koliko se triangulacija poligona P_{n-1} pojavljuje kao sastavni deo triangulacija poligona P_n. Izraz takođe ukazuje i na broj triangulacije poligona P_n koje u sebi ne sadrže triangulacije poligona P_{n-1}.

Treba napomenuti da su dobijene triangulacije poligona P_n istog redosleda kao i u hijerarhiji koja je opisana u [29]. Svaka triangulacija iz skupa $\mathcal{T}_n$ ima roditelja u $\mathcal{T}_{n-1}$ kao i određeni broj tačno definisanih potomaka u $\mathcal{T}_{n+1}$.

U nastavku ove sekcije biće ispunjena tri cilja:

1. Najpre, dodelićemo odgovarajuću težinu oblika $(2 + i)$ svakoj grani u stablu triangulacije (stablo iz hijerarhije koju su dali autori u radu [29]) i na taj način ćemo dobiti odgovarajuće ***težinsko stablo triangulacija***.
2. Definisaćemo ***dekompoziciju Katalanovog broja*** koja proizilazi iz dobijenog težinskog stabla triangulacija.
3. I na kraju, razvićemo ***algoritam za triangulaciju*** konveksnih poligona koji se bazira na dekompoziciji Katalanovog broja.

3.1.1 Težinsko stablo triangulacija i dekompozicija Katalanovog broja

Sada ćemo navesti postupak dodeljivanja težina u obliku term izraza $(2 + i)$ svakoj ivici na stablu triangulacija. Prilikom označavanja ivica na stablu triangulacija vodićemo se principom da svaka težina ivice predstavlja broj triangulacija potomka.

Broj potomaka za τ^l_{n-1} je u opsegu 2 i $n-2$. Prema tome, iz jedne određene triangulacije poligona P_{n-1} možemo izvesti $2+i$ triangulacija za P_n, gde i u opštem slučaju uzima vrednost iz skupa $i \in \{0, 1, \dots, n-4\}$. Obzirom da je broj potomaka veći ili jednak broju 2, iz toga proizilazi da se koriste težine $(2 + i)$, za $i \in \{0, 1, \dots, n-4\}$.

Sa $\left(\tau^l_{n-1}, S^{i_k}(\tau^l_{n-1})\right)$ označićemo ivicu na stablu triangulacija povezujući čvorove $\tau^l_{n-1} \in \mathcal{T}_{n-1}$ i $S^{i_k}(\tau^l_{n-1}) \in \mathcal{T}_n$ između nivoa $n-1$ i n, respektivno.

Triangulacija $S^{i_k}(\tau^l_{n-1})$ se dobija "cepanjem" dijagonale $\delta_{i_k,n-1}$. Pošto dijagonala $\delta_{i_k,n-1}$ ima $k-1$ dijagonala koje se nalaze levo u odnosu na nju i koje su incidentne sa temenom $n-1$, cepajući ovu dijagonalu dobijamo triangulaciju $S^{i_k}(\tau^l_{n-1})$ sa $2+k-1$ potomaka (tj. $2+k-1$ dijagonale koje su incidentne sa temenom n).

Dakle, dodeljujemo težine u obliku $(2+i)$ za odlazne ivice od τ^l_{n-1}, pa prema tome sledi

$$\left(\tau^l_{n-1}, S^{i_1}(\tau^l_{n-1})\right), \dots, \left(\tau^l_{n-1}, S^{i_l}(\tau^l_{n-1})\right). \tag{3.1}$$

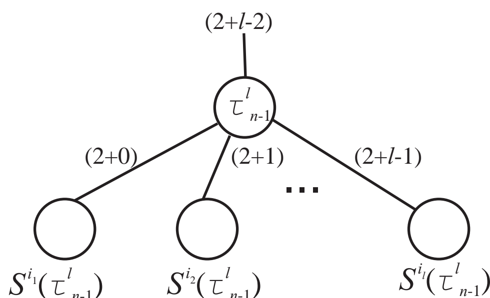
Težina $(2+k-1)$ ivice $\left(\tau^l_{n-1}, S^{i_k}(\tau^l_{n-1})\right)$ označava da triangulacija τ^l_{n-1} ima $k-1$ dijagonala koje su incidentne sa temenom $n-1$ levo od dijagonale $\delta_{i_k,n-1}$, i da $S^{i_k}(\tau^l_{n-1}) \in \mathcal{T}_n$ ima $2+k-1$ potomaka.

Ivice (3.1) imaju redom sledeće težine

$$(2+0), (2+1), \dots, (2+l-1).$$

Triangulacija τ^l_{n-1} ima l potomaka. Prema usvojenom principu dodeljivanja težina stablu triangulacija, dolazna ivica u čvor τ^l_{n-1} ima težinu $2 + l - 2$.

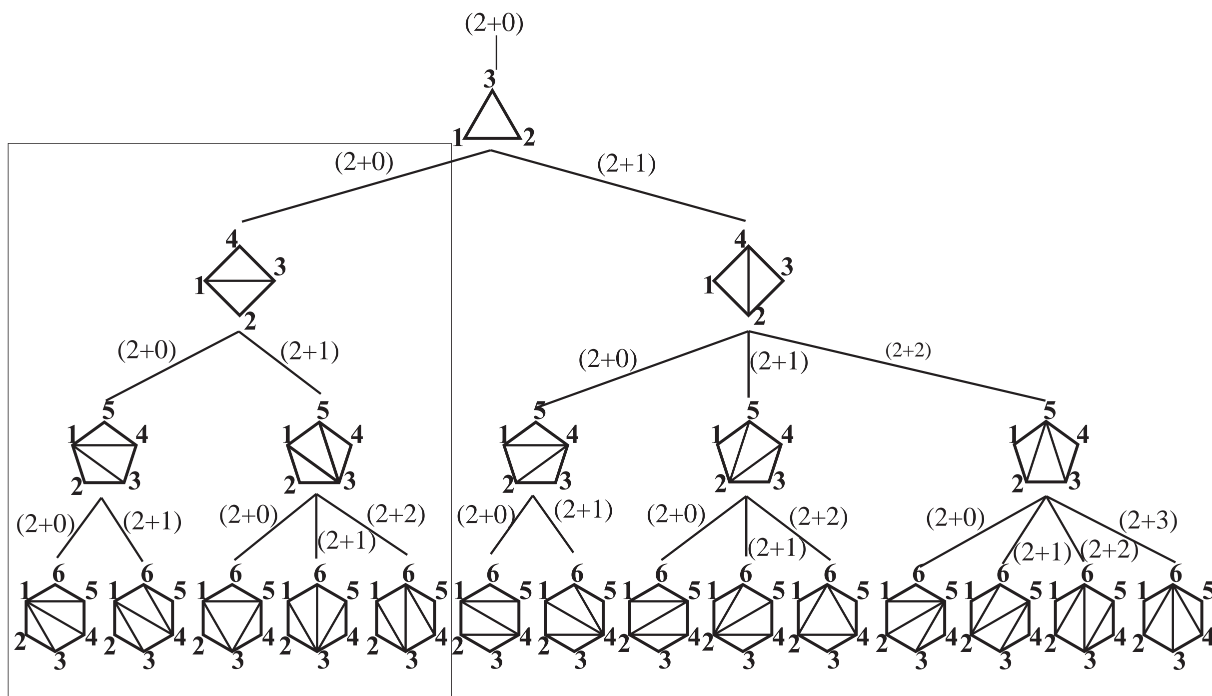
Ovaj postupak je prikazan na Slici 3.1.



Slika 3.1: Težine ivica koje povezuju "roditelja" i njegove "potomke"

U osnovi, pre samog korena stabla definisan je term $(2 + 0)$ koji ukazuje da trougao ima dva potomka (to su dve moguće triangulacije u narednom nivou, tj. u četvorouglu).

Na ovaj način dobijamo težinsko stablo triangulacija (Slika 3.2). Označeni deo stabla na Slici 3.2 je prikazan sa više detalja na Slici 3.3.



Slika 3.2: Težinsko stablo triangulacija za $n \in \{3, ..., 6\}$

Slika 3.3: Izabrani deo stabla sa više detalja

Neka $\mathcal{V}$ bude skup celobrojnih vektora čiji elementi imaju oblik $(2+i)$ za $i \geq 0$, opremljeni sa dve operacije Θ i $\oplus$.

1. Operacija Θ, zatvorena na skupu $\mathcal{V}$, je definisana na sledeći način

$$\Theta\,(2+i) = \langle (2+0), \ldots, (2+i+1) \rangle,\ i \geq 0$$
$$\Theta\,(v) = \langle \Theta\,(v_1)\,, \ldots, \Theta\,(v_m) \rangle, v \in \mathcal{V}$$

   gde $\langle \cdot \rangle$ označava uređenu m-torku, gde su svi ugnježdeni vektori poravnati (što znači da njihove komponente čine jedinstven vektor).

2. Operacija $\oplus$ je definisana kao suma elemenata vektora $\oplus(v) = \sum\limits_{k=1}^{m} v_i$.

Definicija 3.1.1. *Dekompozicija C_{n-1}, označena sa $\Delta(C_{n-1}) \in \mathcal{V}$, je definisana kao vektor svih težina ivica koje povezuju nivoe stabla $n-1$ i n.*

Prema tome, na osnovu postupka dodeljivanja težina imamo da je $C_{n-1} = \oplus(\Delta(C_{n-1}))$ i da su dimenzije vektora $\Delta(C_{n-1})$ jednake C_{n-2}.

Lema 3.1.1. *Operator $\Delta(C_{n-1})$ ispunjava sledeću rekurzivnu relaciju:*

$$\Delta(C_{n-1}) = \Theta(\Delta(C_{n-2})),\ n \geq 4.$$

Dokaz. Za svako l, dolazna ivica u čvor τ^l_{n-1} sa težinom $2+l-2$ proizvodi odlazne ivice (3.1) iz čvora τ^l_{n-1} sa težinama koje su uključene u vektor

$$\langle (2+0), (2+1), \ldots, (2+l-1) \rangle = \Theta\,(2+l-2) = \Lambda^{l}_{l_1=1}(2+l_1-1). \tag{3.2}$$

Kako je suma svih težina ivica na stablu između nivoa $n-2$ i $n-1$ jednaka C_{n-2}, i suma težina dodeljena ivicama koje povezuju nivoe $n-1$ i n jednaka C_{n-1}, jasno je da je iskaz dokazan. □

Primer 3.1.1. *Iz prethodnih ispitivanja dobijamo sledeće dekompozicije:*

$$\begin{aligned}
\Delta(C_2) &= \langle (2{+}0)\rangle \\
\Delta(C_3) &= \langle (2{+}0), (2{+}1)\rangle = \Theta(\Delta(C_2)) \\
\Delta(C_4) &= \langle (2{+}0), (2{+}1),\ (2{+}0), (2{+}1), (2{+}2)\rangle = \langle \Theta(2{+}0), \Theta(2{+}1)\rangle = \Theta(\Delta(C_3)) \\
\Delta(C_5) &= \langle (2{+}0), (2{+}1),\ (2{+}0), (2{+}1), (2{+}2), \\
&\qquad (2{+}0), (2{+}1),\ (2{+}0), (2{+}1), (2{+}2),\ (2{+}0), (2{+}1), (2{+}2), (2{+}3)\rangle \\
&= \langle \Theta(2{+}0), \Theta(2{+}1), \Theta(2{+}0), \Theta(2{+}1), \Theta(2{+}2)\rangle = \Theta(\Delta(C_4)).
\end{aligned} \tag{3.3}$$

Pored jednakosti $\Delta(C_n) = \Theta(\Delta(C_{n-1}))$, sledeća rekurentna relacija je zadovoljena.

Posledica 3.1.1. *Za svako $n \geq 4$, važi opšta rekurentna relacija:*

$$\Delta(C_n) = \Theta^k(\Delta(C_{n-k})),\ \ k \geq 1.$$

U opštem slučaju imamo sledeću teoremu.

Teorema 3.1.1. *Dekompozicija C_n je definisana nizom svih težina ivica povezujući nivoe n i $n+1$:*

$$\begin{aligned}
\Delta(C_n) &= \Theta^{n-2}(2+0) \\
&= \Theta^{n-3}(\Theta(2+1-1)) \\
&= \Lambda^{2}_{l_1=1}\Theta^{n-4}(\Theta(2+l_1-1)) \\
&= \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\cdots\Lambda^{l_{n-4}+1}_{l_{n-3}=1}\ \Lambda^{l_{n-3}+1}_{l_{n-2}=1}(2+l_{n-2}-1),\ \ n \geq 3.
\end{aligned}$$

Dokaz. Dovoljno je da se pokaže induktivni korak. Iz induktivne hipoteze i Leme 3.1.1 dobijamo

$$\begin{aligned}
\Delta(C_n) &= \Theta(\Delta(C_{n-1})) \\
&= \Theta\left(\Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\cdots\Lambda^{l_{n-5}+1}_{l_{n-4}=1}\ \Lambda^{l_{n-4}+1}_{l_{n-3}=1}(2+l_{n-3}-1)\right) \\
&= \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\cdots\Lambda^{l_{n-5}+1}_{l_{n-4}=1}\ \Lambda^{l_{n-4}+1}_{l_{n-3}=1}\Theta(2+l_{n-3}-1).
\end{aligned}$$

Dokaz se može dovršiti primenom (3.2). □

Primer 3.1.2. *Prema Teoremi 3.1.1, dekompozicije* (3.3) *mogu biti pojednostavljene:*

$$\begin{aligned}
\Delta(C_3) &= \Theta(2+0) = \Lambda^{2}_{l_1=1}(2+l_1-1) \\
\Delta(C_4) &= \Lambda^{2}_{l_1=1}\Theta(2+l_1-1) = \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}(2+l_2-1) \\
\Delta(C_5) &= \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\Theta(2+l_1-1) = \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\Lambda^{l_2+1}_{l_3=1}(2+l_3-1) \\
\Delta(C_6) &= \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\Lambda^{l_1+1}_{l_2=1}\Theta(2+l_2-1) = \Lambda^{2}_{l_1=1}\Lambda^{l_1+1}_{l_2=1}\Lambda^{l_2+1}_{l_3=1}\Lambda^{l_3+1}_{l_4=1}(2+l_4-1).
\end{aligned}$$

Dobijena dekompozicija je jedinstvena i može se primeniti za generisanje triangulacija.

Posledica 3.1.2. *Dekompozicija C_n određena sumiranjem svih težina ivica povezujući nivoe n i $n+1$ je definisana:*

$$C_n = \sum_{l_1=1}^{2} \sum_{l_2=1}^{l_1+1} \cdots \sum_{l_{n-3}=1}^{l_{n-4}+1} \sum_{l_{n-2}=1}^{l_{n-3}+1} (2 + l_{n-2} - 1), \; n \geq 3.$$

3.1.2 Triangulacija konveksnog poligona bazirana na dekompoziciji Katalanovog broja

U ovoj sekciji ćemo prikazati postupak za triangulaciju konveksnog poligona na osnovu odgovarajućeg izraza dekompozicije Katalanovog broja. Dekompozicija $\Delta(C_{n-2})$ nam služi kao smernica za generisanje skupa $\mathcal{T}_n$.

Na Slici 3.2 možemo videti da prve dve triangulacije $S^{i_1}(\tau^l_{n-1})$ i $S^{i_2}(\tau^l_{n-1})$ imaju skup unutrašnjih dijagonala dobijen dodavanjem jedne nove dijagonale u skup τ^l_{n-1}. Ovaj princip se koristi za generisanje skupa triangulacija $\mathcal{T}_n$. Na istoj slici se uočava veza između stabla triangulacija i pridruženih težina oblika $(2+i)$.

Izraz $(2+i)$ označava da su dve triangulacije izvedene tako što se kompletiraju odgovarajuće triangulacije iz skupa dijagonala za τ^l_{n-1}. Napomenućemo da se ove dve triangulacije uvek dobijaju "cepanjem" dijagonale $\delta_{1,n-1}$. Ostatak triangulacija i se formira "od nule" (u daljem tekstu ćemo ih tretirati kao nove triangulacije). U nastavku, ova opšta ideja će biti detaljnije objašnjena.

Algoritam 3.1.1 razlaže dati Katalanov broj u obliku sume termova $(2+i)$. Na ulazu očekuje n temena poligona, a na izlazu daje odgovarajući izraz koji kasnije koristimo za generisanje triangulacija za P_n.

Algoritam 3.1.1 Dekompozicija Katalanovog broja C_{n-2} u sumu termova $(2+i)$

Ulaz: n

1: Postavi kao tekući izraz `expr=(2+0)` (odgovara broju triangulacija četvorougla, $C_2 = 2$).

2: Ponovi $n-4$ puta korake 2.1–2.4.

2.1 Broji termove `(2+i)` u okviru tekućeg izraza i označi taj broj kao *count*.

2.2 Izračunaj sumu u svakom termu (tj. u svakom paru zagrada).
Označi sume sa $sum[s], s = 1, \ldots, count = C_{n-3} = T_{n-1}$.
Postavi `expr` kao prazan string `expr=""`.

2.3 Za svako $sum[s], s = 1, \ldots, count$ izvrši sledeću petlju
`for (i=0; i<sum[s]; i++)`{
Kreiraj term $(2+i)$.
Spoji term sa `expr`, navodjenjem ',' između njih (ako nije prvi term u izrazu). }

2.4 Postavi `expr` kao tekući izraz.

Napomenimo da je potrebno pronaći vrednost drugog sabirka u termu, tj. vrednosti parametra i u $(2+i)$, što čini implementaciju gore navedene procedure mnogo jednostavnijom.

Algoritam 3.1.2 daje niz na izlazu čiji su elementi sume sabiraka u termu $(2+i)$. Ako nam je potrebna vrednost za i, uzećemo odgovarajući element iz niza umanjen za 2.

Algoritam 3.1.2 Izračunavanje suma u termovima $(2+i)$

Ulaz: n

1: Postavi `count = 1` i `sum[0] = 2`.

2: Ponovi $n-4$ puta sledeće korake.

```
l=0;
  for (i=0; i < count; i++)
    for (j=2; j <= sum[i] + 1; j++)
      newsum[l++]=j;
count=l;
for (i=0; i<count; i++) sum[i]=newsum[i];
```

Primer 3.1.3. Pokazaćemo kako radi Algoritam 3.1.1 na primeru dekompozicije $C_4 = 14$ (broj triangulacija za šestougao, gde je na ulazu algoritma $n=6$).

- Korak 1 postavlja tekući izraz na `expr=(2+0)`.

- Korak 2 se sastoji od $n-4=2$ ponavljanja koraka 2.1–2.4.

Na osnovu ovih koraka, u prvom od dva prolaska dobijamo `count=1` i `sum[1]=2` što dovodi do generisanje izraza `expr=(2+0),(2+1)`. U drugom prolasku dobijamo vrednosti `count=2`, `sum[1]=2`, `sum[2]=3`.

Na osnovu ovih parametara se formira izraz `expr=(2+0),(2+1),(2+0),(2+1),(2+2)`.

U nastavku, napravićemo vezu između izraza dekompozicije i postupka dobijanja triangulacija konveksnog poligona. Potrebne vrednosti za dalji postupak su sadržane u izrazu *expr*.

Primer 3.1.4. Relaciju između $\mathcal{T}_{n-1}$ i $\mathcal{T}_n$ na osnovu izraza `expr` ćemo objasniti na primeru šestougla. U tom slučaju odgovarajući izraz dekompozicije je

$$\Delta(C_4) = expr = (2+0),(2+1),(2+0),(2+1),(2+2).$$

Triangulacija iz skupa $\mathcal{T}_5$ definisana parovima dijagonala $\delta_{1,3}$ i $\delta_{1,4}$ ima pridružen prvi term $(2+0)$. To znači da se ova triangulacija pojavljuje dva puta kao početni deo triangulacija u $\mathcal{T}_6$, koje su definisane dijagonalama $\delta_{1,3}$, $\delta_{1,4}$, $\delta_{1,5}$, i $\delta_{1,3}$, $\delta_{1,4}$, $\delta_{4,6}$.

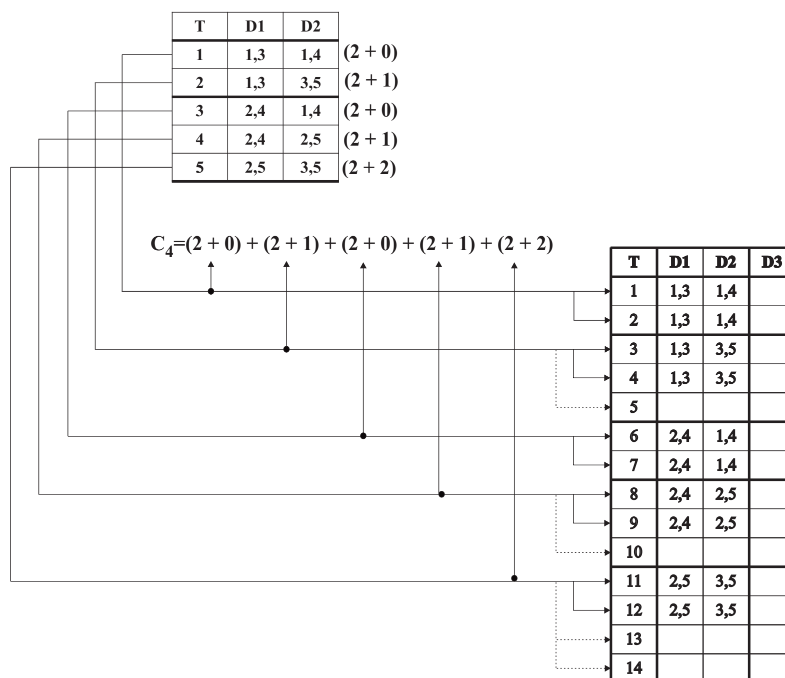
Drugi sabirak u termu $(2+0)$ je jednak nuli, što ukazuje da nema više potomaka koji sadrže ovu triangulaciju. Isti slučaj se javlja i kod triangulacije koja je definisana dijagonalama $\delta_{2,4}$ i $\delta_{1,4}$.

Sa druge strane, u skupu $\mathcal{T}_5$ imamo triangulaciju koja je definisana dijagonalama $\delta_{1,3}$ i $\delta_{3,5}$, i kojoj odgovara term $(2+1)$. Ovo ukazuje da pored dve triangulacije u $\mathcal{T}_6$ koje sadrže u sebi triangulacije iz $\mathcal{T}_5$ (to su: $\delta_{1,3}$, $\delta_{3,5}$, $\delta_{1,5}$ i $\delta_{1,3}$, $\delta_{3,5}$, $\delta_{3,6}$), sadrži i novu triangulaciju definisanu dijagonalama $\delta_{1,3}$, $\delta_{3,6}$, $\delta_{4,6}$.

Isti slučaj se javlja i kod triangulacije koja je definisana dijagonalama $\delta_{2,4}$ i $\delta_{2,5}$.

Na kraju imamo triangulaciju koja je definisana dijagonalama $\delta_{2,5}$ i $\delta_{3,5}$, i kojoj odgovara term $(2+2)$. Pored triangulacija $\delta_{2,5}$, $\delta_{3,5}$, $\delta_{1,5}$ i $\delta_{2,5}$, $\delta_{3,5}$, $\delta_{2,6}$ postoje i dve nove triangulacije: $\delta_{2,6}$, $\delta_{3,6}$, $\delta_{3,5}$, i $\delta_{2,6}$, $\delta_{3,6}$, $\delta_{4,6}$.

Početni deo skupa triangulacija $\mathcal{T}_6$ se može dobiti jednostavnim kopiranjem skupa triangulacija $\mathcal{T}_5$, kao što je ilustrovano na Slici 3.4. Oznaka T u tabeli koje se nalazi na slikama 3.4 i 3.5 označava redni broj triangulacija. Dijagonale koje formiraju odgovarajuću triangulaciju su označene sa $D1, D2, D3$. Nove triangulacije će biti smeštene u praznim redovima prikazane tabele.



Slika 3.4: Početni deo za $\mathcal{T}_6$ generisan kopiranjem triangulacija iz $\mathcal{T}_5$

U ovom trenutku, možemo zaključiti da se veliki deo tabele $\mathcal{T}_n$ može popuniti kopiranjem triangulacija iz $\mathcal{T}_{n-1}$. Razmotrimo sada postupak popunjavanja ostatka tabele za skup triangulacija $\mathcal{T}_n$.

Definicija 3.1.2. *Teme i je zatvoreno unutrašnjom dijagonalom $\delta_{k,l}$ u odnosu na teme j ako se dijagonale $\delta_{i,j}$ i $\delta_{k,l}$ seku.*

Proces kreiranja triangulacije τ_n na osnovu triangulacije τ_{n-1} u suštini znači uvođenje novog temena označenog sa n i nove unutrašnje dijagonale koja je moguće incidentna sa

temenom n. Skup unutrašnjih dijagonala za τ_{n-1} označićemo sa $\mathcal{I}_{n-1}$. Pošto se triangulacija τ_n sastoji od nepresecajućih unutrašnjih dijagonala, naš je cilj da eliminišemo sva temena iz P_n koja su zatvorena dijagonalama iz $\mathcal{I}_{n-1}$ u odnosu na teme n.

Kada eliminišemo sva temena koja su zatvorena unutrašnjim dijagonalama, dobijamo četvorougao sa dve moguće triangulacije i tako kompletiramo kopirane triangulacije τ_{n-1}. Na taj način dobijamo dve triangulacije poligona P_n koje sadrže triangulaciju τ_{n-1} kao početni deo.

Algoritam 3.1.3 Formiranje četvorougla

Ulaz: Ceo broj n i niz

$$\mathcal{T}_{n-1}[row_ind] = \{(i_{\alpha_1}, i_{\beta_1}), \ldots, (i_{\alpha_{n-4}}, i_{\beta_{n-4}})\}$$

od $n-4$ dijagonala (red u tabeli za $\mathcal{T}_{n-1}$).

1: Pronađi

$$i_{min} = \min\{\alpha_1, \ldots, \alpha_{n-4}\},\ i_{max} = \min\{\beta_1, \ldots, \beta_{n-4}\},$$

i generiši listu od četiri elementa $L_1 = \{i_1, i_2, i_3, i_4\}$,

gde je $L_1 = \{1, \ldots, i_{min}, i_{max}, \ldots, n\}$.

Ako L_1 sadrži samo tri elementa, onda koristi

$$L_1 = \{1, \ldots, i_{min}, \beta_j, i_{max}, \ldots, n\},$$

gde je β_j definisano kao

$$\beta_j = \alpha_{j+1},\ \alpha_p > \beta_j, p = j+2, \ldots, n-4. \tag{3.4}$$

Algoritam 3.1.4 Kompletiranje dva kopirana reda

Ulaz: Ceo broj n i indeks *row_ind* tabele koja sadrži $n-4$ kopiranih dijagonala iz jednog reda tabele za $\mathcal{T}_{n-1}$.

1: Poziva Algoritam 3.1.3 sa redom *row_ind* kao parametar.

2: Od preostala četiri temena u listi L_1 kreira dijagonalu δ_{i_1,i_3} i smešta je u poslednju kolonu reda *row_ind*; a zatim smešta dijagonalu δ_{i_2,i_4} u poslednju kolonu reda $row_ind + 1$.

Algoritam 3.1.5 Kompletiranje "novih" triangulacija

Ulaz: Ceo broj n i i (iz izraza $(2+i), i > 0$) i red iz tabele za $\mathcal{T}_{n-1}$ kojem odgovara term $(2+i)$.

1: `for (k = i-1; k >= 0; k--){`

Transformiše ivicu $(n-2-k, n-1-k)$ poligona P_{n-1} u trougao dodavanjem dijagonale $\delta_{n-2-k,n-k}$ u poslednjoj koloni odgovarajućeg reda u tabeli za $\mathcal{T}_n$. Zatim, kopira red iz tabele za $\mathcal{T}_{n-1}$ u prvih $n-4$ kolona odgovarajućeg reda, povećavajući svaku krajnju tačku dijagonala za 1 ako zadovoljavaju uslov $n-2-k$. }

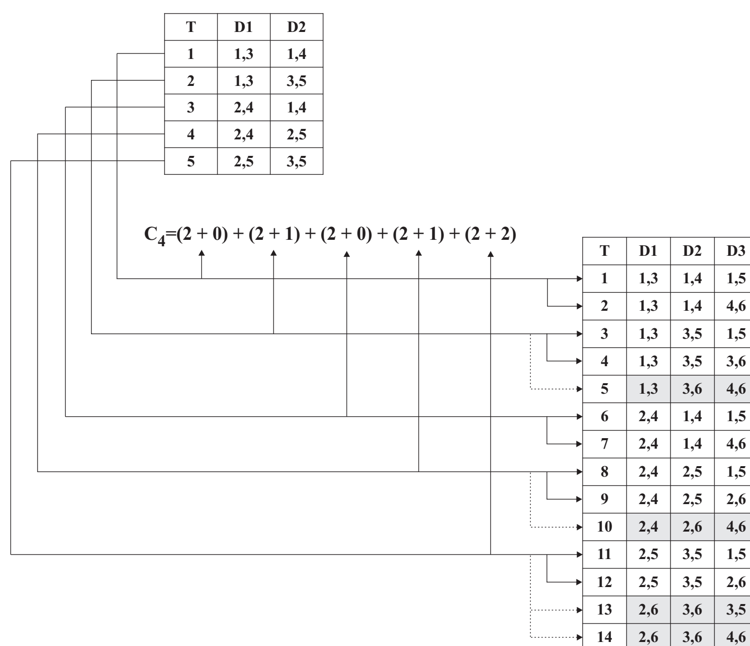
Primer 3.1.5. Objasnićemo način rada Algoritama 3.1.4 i 3.1.5 (tj. postupak dopune tabele sa Slike 3.4 i dobijanje tabele koja je prikazana na Slici 3.5).

Iz primera 3.1.3 dobijamo odgovarajući izraz `(2+0)+(2+1)+(2+0)+(2+1)+(2+2)`. Prvi term izraza je $(2+0)$ koji pokazuje da se kopira triangulacija $\delta_{1,3}$, $\delta_{1,4}$ dva puta i da ona nema više potomaka. Kako bi kompletirali dva kopirana reda potrebno je pozvati Algoritam 3.1.4 sa ulaznim parametrima `n=6` i `row_ind=1`. Ovaj algoritam poziva Algoritam 3.1.3 sa parametrom `row_ind=1`. U okviru ovog reda imamo dijagonale $\delta_{1,3}$ i $\delta_{1,4}$. Algoritam 3.1.3 pronalazi $i_{min} = 1$, $i_{max} = 4$ i generiše listu od četiri elementa $L_1 = \{1, 4, 5, 6\}$.

Korak 3 Algoritma 3.1.4 postavlja dijagonalu $\delta_{1,5}$ u poslednju kolonu reda `row_ind=1` i dijagonalu $\delta_{4,6}$ u poslednju kolonu reda `row_ind+1=2` u tabeli za $\mathcal{T}_6$. Za drugi red koji sadrži dijagonale $\delta_{1,3}$, $\delta_{3,5}$, Algoritam 3.1.3 pronalazi $i_{min} = 1$, $i_{max} = 5$, kao i postojanje parametra $\beta_j = 3$ koji zadovoljava (3.4). Generisana je lista $L_1 = \{1, 3, 5, 6\}$ i dva reda su kompletirana na sličan način.

Drugi sabirak u termu $(2 + 1)$ nam govori da pored dva kopirana reda iz tabele za $\mathcal{T}_5$ koji se dopunjuju na gore opisan način, imamo i dodatnu triangulaciju koja ne sadrži drugi red tabele za $\mathcal{T}_5$ kao početni deo. Te triangulacije su označene kao "nove" i one se generišu pomoću Algoritma 3.1.5. Parametri pomoću kojih se poziva su `n=1` i `i=1`, pa prema tome imamo ciklus od jednog prolaza. Smeštamo dijagonalu $\delta_{4,6}$ u zadnju kolonu. U tekući red tabele za $\mathcal{T}_6$ se kopiraju dijagonale $\delta_{1,3}$, $\delta_{3,5}$ iz drugog reda tabele za $\mathcal{T}_5$, tako što se uvećava svako teme za jedan koje je veće od četiri. Na kraju dobijamo $\delta_{1,3}$, $\delta_{3,6}$, $\delta_{4,6}$.

Obrada poslednja tri terma u izrazu može da se uradi na isti način. Kompletna tabela za skup triangulacija $\mathcal{T}_6$ je prikazana na Slici 3.5. Nove triangulacije odgovaraju sivo obojenim redovima prikazane tabele.



T	D1	D2
1	1,3	1,4
2	1,3	3,5
3	2,4	1,4
4	2,4	2,5
5	2,5	3,5

T	D1	D2	D3
1	1,3	1,4	1,5
2	1,3	1,4	4,6
3	1,3	3,5	1,5
4	1,3	3,5	3,6
5	1,3	3,6	4,6
6	2,4	1,4	1,5
7	2,4	1,4	4,6
8	2,4	2,5	1,5
9	2,4	2,5	2,6
10	2,4	2,6	4,6
11	2,5	3,5	1,5
12	2,5	3,5	2,6
13	2,6	3,6	3,5
14	2,6	3,6	4,6

Slika 3.5: Generisanje za $\mathcal{T}_6$ bazirano na $\mathcal{T}_5$ i odgovarajućeg izraza dekompozicije

3.1.3 Kompleksnost algoritma

Sa R_i ćemo označiti odnos količina ulazih podataka *Hurtado-Noy* algoritma (H_{input}) i naše *metode dekompozicije* (D_{input}). Tačnije, R_i predstavlja odnos broja parova temena koji se preuzimaju u procesu generisanja "potomaka" u $\mathcal{T}_n$ od "roditelja" iz $\mathcal{T}_{n-1}$. *Hurtado-Noy* algoritam radi sa kompletnom strukturom koja određuje triangulaciju (obuhvata unutrašnje dijagonale i spoljašnje ivice), dok *metod dekompozicije* preuzima samo unutrašnje dijagonale koje definišu triangulacije iz prethodnog nivoa. Kompleksnost skladištenja kod *Hurtado-Noy* algoritma 1.3.1 je dat sa $2(2n-5)C_{n-3}$, i označava broj preuzetih temena parova. Sa druge strane, kod *dekompozicije* se zahteva $2(n-4)C_{n-3}$ parova temena. Prema tome, njihov odnos je jednak

$$R_i = \frac{H_{input}}{D_{input}} = \frac{2n-5}{n-4}. \tag{3.5}$$

Prednost *metode dekompozicije* se ogleda u procesiranju i skladištenju manje količine podataka.

Razmotrimo sada kompleksnost naše metode. Poznato je da generišemo C_{n-2} triangulacija za n-tougao tako što upotpunjujemo $2C_{n-3}$ preuzetih triangulacija i generišemo $C_{n-2} - 2C_{n-3}$ 'novih' triangulacija. Posmatraju se potrebne operacije za oba slučaja. Primenom Algoritma 3.1.4 se kompletiraju nove triangulacije. On poziva Algoritam 3.1.3 sa kompleksnošću od $n-4$. Algoritam 3.1.4 kompletira dva preostala reda pozivajući dve dodatne operacije. Da bi se izvršile $2C_{n-3}$ triangulacije potrebno je $C_{n-3}(n-4+2) = (n-2)C_{n-3}$ operacija. Za ostatak od $C_{n-2} - 2C_{n-3}$ triangulacija, prema Algoritmu 3.1.5, potrebno je da za svaku kreiramo odgovarajuću transformaciju u $n-3$ redova, iz čega sledi da je potrebno $(n-3)(C_{n-2} - 2C_{n-3})$ operacija. Za kreiranje dekompozicije C_{n-2} potrebno je $\sum_{i=2}^{n-3} C_i$ prolazaka u Algoritam 3.1.2.

Ukupan broj operacija za *metod dekompozicije* jednak je

$$N_D = (n-2)C_{n-3} + (n-3)(C_{n-2} - 2C_{n-3}) + \sum_{i=2}^{n-3} C_i = (n-3)C_{n-2} - (n-4)C_{n-3} + \sum_{i=2}^{n-4} C_i.$$

Sa druge strane, za *Hurtado-Noy* algoritam je potrebno za svaku triangulaciju na nivou $n-1$ izvršiti $2n-5$ provera, da bi se pronašle dijagonale koje su incidentne sa temenom $n-1$. Ukupan broj ovih provera jednak je $(2n-5)C_{n-3}$. Dalje, moramo ići kroz dijagonale i kopirati iste bez transformacija, dok je neke od njih neophodno transformisati. Vrši se i dodavanje dve nove dijagonale za svaku incidentnu dijagonalu koja se pronađe. Na taj način se kreira $2n-3$ parova koji opisuju jednu novu triangulaciju. Ukupan broj incidentnih dijagonala jednak je C_{n-2}. Dakle, kad je u pitanju ovaj algoritam, ukupan broj operacija je

$$N_H = (2n-5)C_{n-3} + (2n-3)C_{n-2}.$$

Nije teško proveriti tačnost nejednakosti $N_H > N_D$, za sve vrednosti od $n \geq 4$.

3.1.4 Komparativna analiza i eksperimentalni rezultati

Aplikacije za *Hurtado-Noy* i za *metodu dekompozicije* su realizovane u programskom jeziku *Java* u *NetBeans* okruženju.

Eksperimentalni rezultati su dati u Tabeli 3.1. Navedeno je ukupno vreme za obe metode (kolone H_{all} i D_{all}). Pored toga, ukupno vreme je podeljeno na dva dela: (1) vreme koje je potrebno za preuzimanje skupova temena koja definišu triangulacije iz $\mathcal{T}_{n-1}$ (H_i i D_i). U tabeli je navedena i njihova razlika Dif_i; (2) vreme koje je potrebno za dopunu preuzetih skupova kako bi se dobile triangulacije u $\mathcal{T}_n$ (H_g i D_g). Na kraju je dato i postignuto ubrzanje (U).

Tabela 3.1: Eksperimentalni rezultati testiranja: *dekompozicija i Hurtado-Noy*

n	**Broj. Tr.**	H_i	H_g	H_{all}	D_i	D_g	D_{all}	Dif_i	**U**
5	**5**	0.01	0.24	**0.25**	0.002	0.17	**0.18**	**0.008**	**1.39**
6	**14**	0.03	0.31	**0.34**	0.009	0.26	**0.27**	**0.021**	**1.26**
7	**42**	0.06	0.37	**0.43**	0.02	0.33	**0.35**	**0.04**	**1.23**
8	**132**	0.08	0.41	**0.49**	0.03	0.40	**0.43**	**0.05**	**1.14**
9	**429**	0.14	0.53	**0.67**	0.06	0.55	**0.61**	**0.08**	**1.10**
10	**1,430**	0.30	0.88	**1.18**	0.12	0.91	**1.03**	**0.18**	**1.15**
11	**4,862**	1.10	2.71	**3.81**	0.36	2.75	**3.11**	**0.74**	**1.23**
12	**16,796**	3.25	6.91	**10.16**	1.37	7.81	**9.18**	**1.88**	**1.11**
13	**58,796**	14.91	27.70	**42.61**	6.40	29.54	**35.94**	**8.51**	**1.19**
14	**208,012**	39.96	65.19	**105.15**	17.37	74.71	**92.08**	**22.59**	**1.14**
15	**742,900**	112.37	161.71	**274.08**	49.50	187.61	**237.11**	**62.87**	**1.16**
16	**2,674,440**	263.22	335.00	**598.22**	116.99	438.94	**555.93**	**146.23**	**1.08**

Testiranje je izvršeno u *NetBeans* modulu *"CPU Analyze Performance"* na konfiguraciji performansi: *CPU - Intel(R) Core(TM)2Duo CPU, T7700, 2.40 GHz, L2 Cache 4 MB, RAM - 2 Gb, Graphic - NVIDIA GeForce 8600M GS.*

Kod testiranja *Hurtado-Noy* metode značajno vreme se troši u momentu preuzimanja $(2n-5)$ parova temena koji određuju kompletnu strukturu triangulacije iz skupa $\mathcal{T}_{n-1}$. Kod *metode dekompozicije* se preuzima $n-4$ parova temena.

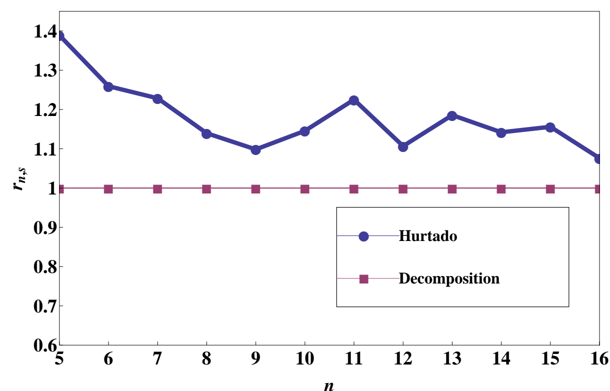
Primenom tzv. *Bencmark* profila performansi, koji je opisan u radu [2], mogu se uvideti bolje performanse za *metod dekompozicije.* Osnovna metrika se odnosi na CPU vreme za konstrukciju (generisanje) svih triangulacija za $n \in \{5, 6, ..., 16\}$.

Prateći oznake date u radu [2] broj metoda (rešavaoci problema) u ovom slučaju je $n_s = 2$ (*Decomposition* i *Hurtado*) a broj numeričkih eksperimenata je $n_p = 12$. Na osnovu $t_{n,s}$, dobijamo broj iteracija koje su potrebne za rešavanje problema triangulacije za n-tougao primenom rešavaoca s. Količnik

$$r_{n,s} = \frac{t_{n,s}}{\min\{t_{n,s} : s \in \{Hurtado, Decomposition\}\}}$$

predstavlja odnos performansi za dve metode.

Taj odnos, za dve testirane metode, je prikazan na sledećem grafikonu (Slika 3.6).



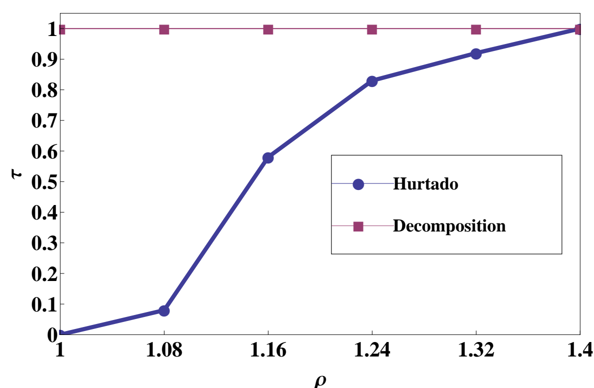
Slika 3.6: Odnos performansi $r_{n,s}$ na osnovu CPU vremena

Performanse rešavaoca s su definisane funkcijom

$$\rho_s(\tau) = \frac{1}{n_p}\text{size}\{p \in \mathcal{P} : r_{n,s} \leq \tau\}, \;\; s \in \{Hurtado, Decomposition\}$$

gde $\tau \in \mathbb{R}$ i $\mathcal{P}$ predstavlja skup problema.

Sledeći grafikon (Slika 3.7) prikazuje odnos u komparativnoj analizi ($\rho_s(\tau)$) za dve navedene metode.



Slika 3.7: Funkcija $\rho_s(\tau)$: *metod dekompozicije* i *Hurtado-Noy*

Na osnovu numeričkih podataka koji su dobijeni eksperimentalnim ispitivanjem obe *Java* aplikacije, može se zaključiti sledeće:

- Ukupno vreme (u sekundama) za svih 12 testiranja (za $n \in \{5, 6, ..., 16\}$) kod *Hurtado-Noy* algoritma je 1037.39 dok je kod *dekompozicije* 936.22. Postignuto je prosečno ubrzanje 1.108, odnosno poboljšanje za 10,8 %.
- Prosečno vreme izvršavanja po jednom nivou kod *Hurtado-Noy* algoritma je 94.31 dok je kod *dekompozicije* 85.11.
- Primenom *dekompozicije* postignuto je povećanje broja triangulacija u sekundi za 1.132, odnosno poboljšanje za 13,2 %.

3.1.5 Detalji implementacije u *Javi*

Implementacija za *metod dekompozicije* se sastoji od klasa: `GenerateTriangulations`, `Triangulation`, `DataBase`, `Node`, `LeafNode` i `Point`.

U klasi `GenerateTriangulations` je definisana metoda koja je zadužena za dekompoziciju Katalanovog broja i za generisanje triangulacija na osnovu dobijenih izraza iz postupka dekompozicije. Navešćemo delove metode `createExpr()` po koracima koji odgovaraju Algoritmu 3.1.1:

Korak 1: Postavljanje početnog izraza

```
public Vector<Node> createExpr (int k) throws IOException {
Vector<Vector> expr = new Vector<Vector>();
expr.add(new Vector<LeafNode>());
expr.get(0).add(new LeafNode());
```

Korak 2: k-iteracije

```
for (int n=0; n <=k; n++) {
```

Korak 2.1: izračunavanje sume u termu

```
Vector<Node> exprSum = new Vector();
```

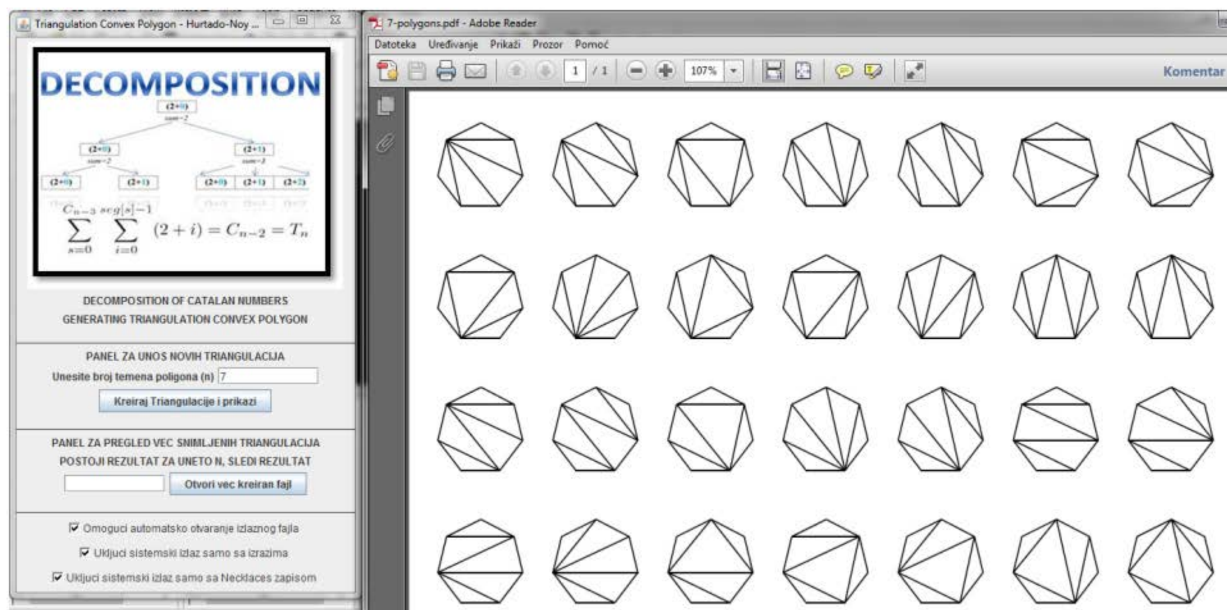
Korak 2.2: kreiranje izraza za naredne nivoe

```
for (int j = 0; j < expr.get(n).size(); j++) {
 Node t = (Node)expr.get(n).get(j);
 Node s = new Node(new LeafNode(), t.copy());
 exprSum.add(s);

 for (int e = 0; e < t.leftBranch(); e++) {
  s = t.copy();
  Node r = s;
  for (int i=0; i<e; i++){
    s = s.getLeft();
    }

 s.setLeft(new Node(new LeafNode(), s.getLeft()));
 exprSum.add(r);
    }
}
```

Korak 2.3: spajanje termova

```
expr.add(exprSum);
return expr.get(k);
}
```

Na Slici 3.8 su prikazana dva prozora pri izvršavanju *Java* aplikacije na primeru generisanja triangulacija za sedmougao. Levi prozor prikazuje interfejs aplikacije (GUI) a na desnom prozoru su prikazane izgenerisane triangulacije.



Slika 3.8: *Java* aplikacija za generisanje triangulacija na osnovu *dekompozicije*

3.2 Blok metod za generisanje triangulacija poligona

Blok metod ima cilj da ubrza proces generisanja triangulacija poligona P_n koristeći skup triangulacija $\mathcal{T}_{n-1}$. Generalna strategija koja je korišćena u Blok metodi je da se glavni problem razlaže na manje potprobleme koji su međusobno zavisni. Svaki potproblem se rešava samo jednom a koristi se više puta i na taj način se izbegava nepotrebno ponavljanje istih izračunavanja. U cilju izbegavanja generisanja istih triangulacija koristili smo rekurziju sa memoizacijom.

3.2.1 Uvodne napomene i osnovne postavke

Pretpostavimo da se skup $\mathcal{T}_{n-1}$ može koristiti više puta kao *blok* za generisanje triangulacija u skupu $\mathcal{T}_n$. Obzirom da važi

$$C_n = \frac{4n-2}{n+1} C_{n-1} \tag{3.6}$$

nije teško proveriti da je nejednakost $T_n > 2T_{n-1}$ zadovoljena za sve $n > 4$.

Dakle, broj triangulacija $T_n = C_{n-2}$ se može rekurzivno izraziti kao:

$$T_n = 2T_{n-1} + \text{rest}(R_n). \tag{3.7}$$

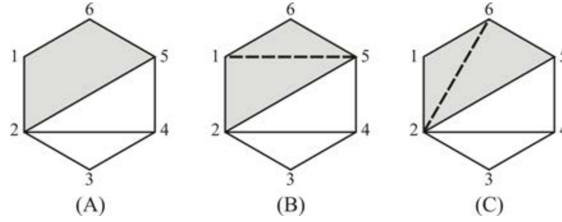
Sledeći stav daje formalnu osnovu za našu metodu.

Stav 3.2.1. *Svaka triangulacija iz skupa $\mathcal{T}_{n-1}$ se pojavljuje kao polazni deo generisanja za tačno dve triangulacije u skupu $\mathcal{T}_n$.*

Dokaz. Ako uzmemo u obzir poligon P_{n-1} i njegovu ivicu $\delta_{1,n-1}$, onda je očigledno da u $\mathcal{T}_{n-1}$ navedena ivica pripada trouglovima $(1, i, n-1)$, $2 \leq i \leq n-2$. Novo teme n, koje se ne nalazi u poligonu P_{n-1}, je u poziciji da formira konveksan poligon (sa temenima $1, ..., n-1$). Dijagonale $\delta_{1,i}$ i $\delta_{i,n-1}$ čine temena $2, \ldots, i-1, i+1, \ldots, n-2$ zatvorenim u odnosu na teme n. Temena $1, i, n-1$, i n formiraju četvorougao. Četvorougao se može triangulisati na dva načina, tako da se svaka triangulacija iz $\mathcal{T}_{n-1}$ pojavljuje kao polazni deo u generisanju dve triangulacije u skupu $\mathcal{T}_n$. Napomenućemo da postoje i druge triangulacije u skupu $\mathcal{T}_n$ koje se dobijaju na drugačiji način. □

Memoizacija je korisna i važno je sredstvo za rešavanje teških rekurzivnih izračunavanja [14]. Problem triangulacije poligona je po svojoj prirodi rekurzivan i dugotrajan posao. Umesto da imamo ulaz u tabelu za rešavanje svakog potproblema, u ovom slučaju imamo prethodno snimljene generisane triangulacije za $\mathcal{T}_{n-1}$. Kao što je prethodno objašnjeno, imamo značajan broj slučajeva gde su triangulacije iz $\mathcal{T}_{n-1}$ početni deo za dve triangulacije u $\mathcal{T}_n$. Ovde nije potrebno nastaviti dalje sa rekurzijom (sve do trougla), već u stilu memoizacije koristimo ono što već imamo i adekvatnom dopunom dobijamo dve nove triangulacije. U slučaju tradicionalne rekurzije, generisanje za $\mathcal{T}_{n-1}$ bi se dva puta ponovilo u nezavisnim izračunavanjima podstabla. Ovo ponavljanje je izbegnuto primenom ove metode.

Glavna ideja je predstavljena na Slici 3.9, gde je dat postupak transformacije iz triangulacije petougla u dve odgovarajuće triangulacije šestougla.



Slika 3.9: Dopuna triangulacije $\delta_{2,4}$; $\delta_{2,5}$ u skupu triangulacija $\mathcal{T}_6$

Na prvom delu slike (A) možemo uočiti da dijagonale $\delta_{2,4}$ i $\delta_{2,5}$ zatvaraju temena 3 i 4, dok temena 1, 2, 5 i 6 formiraju četvorougao koji se može triangulisati na dva načina (B i C). Obe dobijene triangulacije u $\mathcal{T}_6$ sadrže triangulaciju iz $\mathcal{T}_5$ koja je određena sa $\delta_{2,4}$ i $\delta_{2,5}$.

3.2.2 Algoritam za blok metod

U sekciji 3.1.2 smo definisali pojam zatvorenih temena (Definicija 3.1.2). Slično definiciji za dužinu ivice iz rada [27], definisaćemo rastojanje između dva temena poligona.

Definicija 3.2.1. *Rastojanje između dva celobrojna temena i, j gde $i, j \in \{1, 2, \ldots, m\}$ se može definisati izrazom*

$$d(i, j) = d(j, i) = \min\{|i - j|, m - |i - j|\}.$$

U proceduri koja se koristi za pronalaženje i eliminaciju zatvorenih temena, najpre treba krenuti od *uva* triangulacije (videti [30]). *Uvo* triangulacije je prepoznato kada za dijagonalu δ_{i_p,i_q} imamo $d(p, q) = 2$. Obzirom da triangulacija ima najmanje dva *uva* i u najgorem slučaju jedno *uvo* može biti teme n, onda uvek imamo najmanje jedno *uvo* između ostalih temena. Za ovu svrhu kreiramo listu uređenih parova u obliku

$$L = \{(1, 1), (2, 2), \ldots, (n, n)\}. \tag{3.8}$$

Nakon eliminacije $n - l$ parova lista L postaje oblika:

$$L = \{(s, i_s),\ s = 1, \ldots, l\},\ 4 \leq l \leq n,\ i_l = n. \tag{3.9}$$

Vrednosti i_s gde je $s = 1, ..., l$ su oznake temena, dok vrednosti $1, ..., l$ predstavljaju relativne pozicije temena u listi l.

Algoritam 3.2.1 Eliminacija parova

Ulaz: Lista L oblika (3.9) i temena i_p; i_q, gde je $d(p, q) = 2$.

1: Uklanja iz liste L par smešten između (p, i_p) i (q, i_q), kružno.

2: Umanjuje za jedan članove prvog para u parovima koji slede onaj eliminisani.

Dalje, proveravamo prvih $n-4$ kolona u tabeli za $\mathcal{T}_n$ tražeći *uvo* triangulacije. Onda eliminišemo odgovarajući element liste uređenih parova i povećavamo relativnu poziciju parova prateći eliminisani par.

Algoritam 3.2.2 Formiranje četvorougla

Ulaz: Lista L oblika (3.8), ceo broj n i niz od $n-4$ dijagonala (red u tabeli za $\mathcal{T}_n$).

1: Pronalazi dijagonalu δ_{i_p,i_q} gde je $d(p,q)=2$ u listi L.

2: Poziva Algoritam 3.2.1 za parametre i_p i i_q.

3: Ponavlja korake 1 i 2, $n-4$ puta.

Posle eliminisanja $n-4$ parova, imamo četiri temena u listi pomoću kojih formiramo četvorougao koji može biti triangulisan na dva načina.

Algoritam 3.2.3 Algoritam za *blok metod*

Ulaz: Ceo broj n i skup triangulacija $\mathcal{T}_{n-1}$ sa brojem redova $row_{n-1}=C_{n-3}$ i brojem kolona $col_{n-1}=n-4$.

1: Kreira praznu tabelu za $\mathcal{T}_n$ sa brojem redova $row_n=C_{n-2}$ i kolona $col_n=n-3$.

2: Popunjava tabelu za $\mathcal{T}_n$ sa triangulacijama iz $\mathcal{T}_{n-1}$ duplirajući svaki red.

3: Popunjava ostatak za unete blokove (zadnja kolona tabele za prvih $2row_{n-1}$ redova):

    for $(i=1; i\leq 2row_{n-1}; i+=2)$

        Kreira listu L oblika (3.8).

        Poziva Algoritam 3.2.2 za red i iz tabele za $\mathcal{T}_n$.

        Za preostala četiri temena u listi L kreira dijagonalu δ_{i_1,i_3} i smešta u poslednju kolonu za red i.

        Zatim kreira dijagonalu δ_{i_2,i_4} i smešta u poslednju kolonu za red $i+1$.

4: Popunjava ostatak tabele za $\mathcal{T}_n$ koja sadrži $T_n - 2T_{n-1}$ redova.

  4.1: Popunjava prvih $n-4$ kolona u poslednjih $row_n - 2row_{n-1}$ redova.

    $i = 2row_{n-1}+1$

        Kreira listu L oblika (3.8).

        Eliminiše temena koja su susedna sa temenom n na osnovu Algoritma 3.2.1 za parametre 1 i $n-1$.

        Popunjava red i dijagonalama $\delta_{2,n}, \delta_{3,n}, ..., \delta_{n-2,n}$.

        Prvih $n-4$ kolona u ostatku $row_n - 2row_{n-1} - 1$ se popunjava dijagonalama koje sadrže poslednje teme n, dok su za prvo teme vrednosti iz skupa $\{2,3,...,n-2\}$.

        Broj ovakvih kombinacija je $\binom{n-3}{n-4} = n-3$.

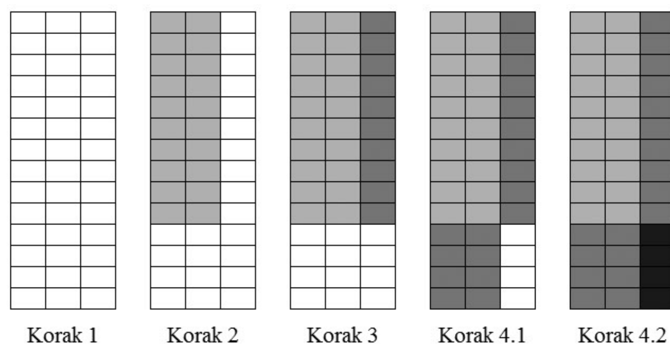
  4.2: Popunjava zadnju kolonu u poslednjih $row_n - 2row_{n-1}$ redova.

    for $(i = 2row_{n-1}+2; i \leq row_n; i++)$

        Kreira listu L oblika (3.8); Poziva Algoritma 3.2.2 za red i iz tabele za $\mathcal{T}_n$.

        Za preostala četiri temena u listi L kreira dijagonalu δ_{i_1,i_3} i smešta je u poslednju kolonu za red i.

Algoritam 3.2.3 se sastoji od četiri koraka (Slika 3.10 sa postupkom popunjavanja tabele).



Slika 3.10: Popunjavanje tabele na osnovu algoritma za *blok metod*

Korak 1 – prazna tabela,
Korak 2 – sadrži kopije preuzetog bloka,
Korak 3 – popunjava se poslednja kolona za preuzete kopije bloka,
Korak 4.1 – popunjavaju se dve kolone sa novim triangulacijama,
Korak 4.2 – popunjava se poslednja kolona za nove triangulacije.

Primer 3.2.1. Navešćemo postupak primene *blok metode* za izvođenje skupa triangulacija $\mathcal{T}_6$ na osnovu skupa $\mathcal{T}_5$.

1. Prvo se kreira prazna tabela dimenzija $C_{n-2} \times (n-3)$. U konkretnom slučaju za $n = 6$, kreira se tabela dimenzija 14 x 3 (korak 1).

2. Zatim se tabela popunjava blokom T_5 (korak 2), tačnije svaki njen red se preuzima dva puta.

3. U trećem koraku se popunjava poslednja kolona za 10 redova, jer je $2row_b = 10$. Za prvih 5 neparnih redova (1,3,5,7 i 9) se kreira lista:

$$L = \{(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)\}.$$

   Zatim se pronalaze dijagonale koje se završavaju u temenima čija je razlika 2. Takva dijagonala je u prvom redu $\delta_{1,3}$. Vrši se eliminacija temena između njih (to je teme 2). Nakon ove eliminacije, lista sadrži $L = \{(1, 1), (2, 3), (3, 4), (4, 5), (5, 6)\}$.

   Dalje, temena gde se završava dijagonala $\delta_{2,4}$ imaju takođe razliku 2. Lista se nakon druge eliminacije transformiše u: $L = \{(1, 1), (2, 4), (3, 5), (4, 6)\}$. Dakle, izvršena je eliminacija $(n - 4) = 2$ zatvorena temena i preostaju četiri dozvoljena temena a to su: 1,4,5 i 6.

Preostala temena formiraju četvorougao koji se može triangulisati na dva načina i to dijagonalom $\delta_{1,5}$ ili $\delta_{4,6}$. Sledi, u zadnjoj koloni u prvom redu se skladišti $\delta_{1,5}$, a u zadnjoj koloni u drugom redu se skladišti $\delta_{4,6}$. Dve dobijene triangulacije u $\mathcal{T}_6$ su: $\delta_{1,3}, \delta_{1,4}, \delta_{1,5}$ i $\delta_{1,3}, \delta_{1,4}, \delta_{4,6}$. Na isti način popunjavamo i preostale redove.

Popunjen deo tabele sadrži 5 redova sa dijagonalama koje se ne završavaju u poslednjem temenu, i 5 redova koji sadrže tačno jednu dijagonalu koja se završava u poslednjem temenu (u ovom slučaju to je teme 6).

4. U četvrtom koraku se popunjava ostatak tabele, odnosno poslednjih $row_n - 2row_b$ redova. Broj "novih" triangulacija kod *blok metode* se određuje na osnovu $rest(R_n)$ iz formule (3.7). U ovom slučaju to su 4 triangulacije.

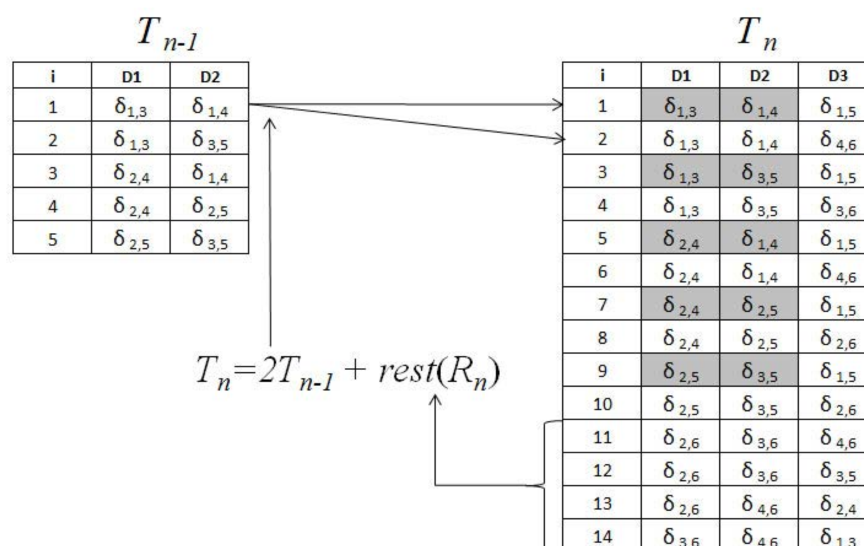
   Preostali nepopunjen deo tabele sadrži triangulacije sa kombinacijom dijagonala gde najmanje dve dijagonale u jednom redu moraju sadržati poslednje teme n (Korak 4.1). To je garancija da se neće desiti ponavljanje istih kombinacija dijagonala koje su preuzete iz bloka, a samim tim i neće se generisati iste triangulacije.

   Uvek jedan red u nepopunjenom delu tabele sadrži sve dijagonale koje se završavaju u poslednjem temenu n. Nakon kreiranja liste L i eliminacije susednih temena sa poslednjim temenom 6 (to su 1 i 5) lista se transformiše u $L = \{(1,2),(2,3),(3,4),(4,6)\}$. Ovde postoji jedna moguća triangulacija sa tri dijagonale koje se završavaju u poslednjem temenu a to je: $\delta_{2,6}, \delta_{3,6}, \delta_{4,6}$. Postoje još tri kombinacije sa tačno dve dijagonale koje se završavaju u poslednjem temenu a to su: $\delta_{2,6}, \delta_{3,6}; \delta_{2,6}, \delta_{4,6}; \delta_{3,6}, \delta_{4,6}$.

   U koraku 4.2, na isti način kao u koraku 3, popunjava se preostali deo za poslednja $row_n - 2row_b$ reda. Ponovo se kreira lista $L = \{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$. Dijagonala $\delta_{2,6}$ eliminiše teme 1, jer je zatvoreno i nalazi se između njih, pa se lista transformiše u novi oblik: $L = \{(1,2),(2,3),(3,4),(4,5),(5,6)\}$. Dijagonala $\delta_{3,6}$ se takođe završava u temenima čija je razlika 2 i to ukazuje na eliminaciju temena 2. Lista se, nakon eliminacije, transformiše u: $L = \{(1,3),(2,4),(3,5),(4,6)\}$. Eliminisano je $(n-4) = 2$ zatvorena temena i preostaju četiri dozvoljena a to su 3,4,5 i 6. Oni formiraju četvorougao koji se može triangulisati na dva načina i to dijagonalom $\delta_{4,6}$ ili $\delta_{3,5}$. U zadnjoj koloni u prvom redu nepopunjenog dela se skladišti $\delta_{4,6}$ a u zadnjoj koloni u drugom redu se skladišti $\delta_{3,5}$. Tačnije, to su nove triangulacije: $\delta_{2,6}, \delta_{3,6}, \delta_{3,5}; \delta_{2,6}, \delta_{3,6}, \delta_{4,6}$.

   Na isti način popunjavamo i preostale redove u nepopunjenom delu tabele.

   Kompletan postupak generisanja skupa triangulacija $\mathcal{T}_6$ koji je baziran na skupu $\mathcal{T}_5$ je prikazan na Slici 3.11.



Slika 3.11: Generisanje skupa triangulacija $\mathcal{T}_6$

3.2.3 Komparativna analiza i eksperimentalni rezultati

U ovoj sekciji je navedena komparativna analiza *blok metode* i *Hurtado-Noy* algoritma. Blok metoda, kao i prethodna *metoda dekompozicije*, je implementirana u *Java NetBenas IDE* okruženju.

Eksperimentalni rezultati su dati u Tabeli 3.2. Navedena su vremena izvršavanja (u sekundama) za obe metode. Date su i veličine datoteka koje se kreiraju pri testiranju obe metode (H_o i B_o u Kb). U tim datotekama se skladište temena koja određuju triangulacije u skupu $\mathcal{T}_n$, koji se kasnije mogu koristiti kao blokovi za generisanje skupa $\mathcal{T}_{n+1}$ itd.

Tabela 3.2: Eksperimentalni rezultati testiranja: *blok i Hurtado-Noy*

n	**Broj triang.**	**Hurtado**	**Blok**	**U**	H_o	B_o
5	5	0.25	**0.16**	1.56	0.26	**0.08**
6	14	0.34	**0.26**	1.31	0.95	**0.32**
7	42	0.43	**0.34**	1.26	3.47	**1.26**
8	132	0.49	**0.41**	1.20	12.87	**4.95**
9	429	0.67	**0.46**	1.46	48.26	**19.31**
10	1,430	1.18	**0.54**	2.19	182.33	**75.08**
11	4,862	3.81	**0.85**	4.48	692.84	**291.72**
12	16,796	12.46	**1.32**	9.44	2645.37	**1133.73**
13	58,786	50.51	**4.40**	11.48	10140.59	**4408.95**
14	208,012	119.05	**24.13**	4.93	39002.25	**17160.99**
15	742,900	318.63	**85.30**	3.74	150437.25	**66861.00**
16	2,674,440	/	**529.29**	/	/	**240699.60**

Testiranje za obe metode je urađeno po nivoima. Dakle, ukupno zabeleženo vreme obuhvata sve aktivnosti od momenta preuzimanja prve triangulacije iz skupa $\mathcal{T}_{n-1}$ do momenta kad se na izlazu algoritma daje poslednja triangulacija u skupu $\mathcal{T}_n$.

Kao i kod prethodne komparativne analize, i ovde je primenjen tzv. *Benchmark* profil performansi [2]. U ovom slučaju, osnovna metrika se odnosi na CPU vreme za generisanje svih triangulacija za $n \in \{5, 6, ..., 15\}$, gde je broj rešavaoca $n_s = 2$ (*Block* i *Hurtado*), a broj numeričkih eksperimenata je $n_p = 11$. Na osnovu $t_{n,s}$ dobijamo broj iteracija koje su potrebne za rešavanje problema triangulacije za n-gon primenom rešavaoca s.

Količnik

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \{Hurtado, Block\}\}}$$

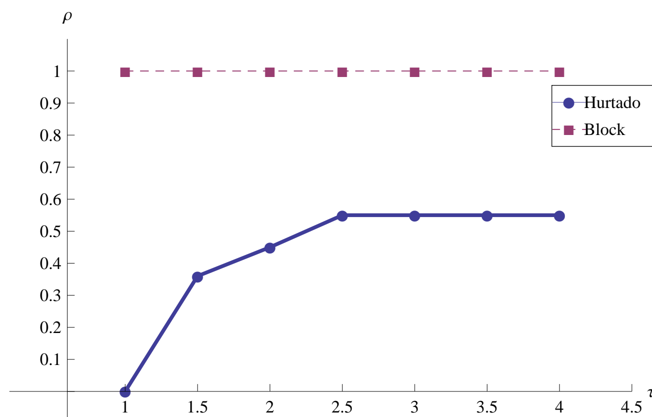
predstavlja odnos performansi dva metoda za triangulaciju konveksnog mnogougla.

Performanse metoda triangulacije, označenog sa s, definisane su funkcijom

$$\rho_s(\tau) = \frac{1}{n_p}\text{size}\{p \in \mathcal{P} : r_{p,s} \leq \tau\}, \ s \in \{Hurtado, Block\},$$

gde $\tau \in \mathbb{R}$ i $\mathcal{P}$ predstavlja skup problema.

Sledeći grafikon (Slika 3.12) prikazuje odnos performansi ($\rho_s(T)$), na osnovu podataka iz Tabele 3.2, za: *blok metodu* i *Hurtado-Noy* algoritam.



Slika 3.12: Odnos performansi (CPU vreme): *blok metoda* i *Hurtado-Noy*

Na osnovu numeričkih podataka, koji su dobijeni eksperimentalnim ispitivanjem obe *Java* aplikacije, može se utvrditi da je prosečno vreme izvršavanja po jednom nivou kod *Hurtado-Noy* algoritma 46.16 dok je kod *blok metode* 10.74. Obrada kompletne strukture triangulacija na ulazu znatno usporava rad *Hurtado-Noy* algoritma. Sve ovo bitno utiče i na opterećenost memorije u toku izvršavanja, pa i na brzinu generisanja triangulacija.

Broj triangulacija drastično raste sa povećanjem vrednosti n i te triangulacije je potrebno skladištiti u neku izlaznu datoteku. Za $n > 16$ prilikom testiranja, na računaru navedenih

performansi, RAM memorija je iscrpljena (Tabela 3.2).

Ovaj problem se može prevazići primenom virtuelne memorije (*virtual memory paging file*). Za vrednosti $n > 16$, skup triangulacija $\mathcal{T}_n$ se ne može smestiti u isto vreme u radnoj memoriji. Činjenica je da od neke vrednosti ($n > 20$) moramo da uključimo i vreme čitanja/pisanja podataka sa diska. Sve to utiče na značajno povećanje vremena izvršavanja. To se i dešava sa *Hurtado-Noy* algoritmom koji na ovaj način pokušavamo da ubrzamo.

3.2.4 Detalji implementacije i rad sa bazama podataka u *Javi*

U implementaciji ove metode je stavljen akcenat na mogućnost rada sa bazama podataka u *Java NetBeans* okruženju (*JDBC*). *Java* sa svojim *JDBC API*-jem se pokazala kao efikasan alat za realizaciju ove metode. Implementacija *blok metode* se sastoji od sledećih klasa: `GenerateTriangulations`, `Triangulation`, `DataBase`, `Node`, `LeafNode` i `Point`.

U klasi `GenerateTriangulations` je definisana metoda `Block()` za generisanje triangulacija. Struktura ove metode se može podeliti na četiri dela (koji prate korake Algoritma 3.2.3). Najpre se uzimaju triangulacije $\mathcal{T}_{n-1}$ (Korak 1). Posle toga, $\mathcal{T}_{n-1}$ se kopira dva puta (Korak 2). U koraku 3, poslednja kolona u tabeli se popunjava dodatnim dijagonalama. Taj posao obavlja metoda `contains()` koja je član klase `Node`. Ova metoda odgovara Algoritmu 3.2.2 i ima za cilj da pronađe temena koja nisu zatvorena i koja mogu formirati četvorougao.

Metoda `contains()` poziva metodu `elimination()` koja odgovara Algoritmu 3.2.1. Poslednji korak pronalazi ostatak triangulacija (Korak 4.1), gde se metoda `contains()` ponovo poziva kako bi se završio Korak 4.2.

U nastavku je naveden *Java* izvorni kôd za metodu `Block (int CatNum)`:

```
public Vector<Node> Block (int CatNum) throws IOException {

//step 1
     Vector<Vector> bl = new Vector<Vector>();
     this.openFile("baze/[T"+(CatNum+1)+" BAZA].jdb");
     bl.add(new Vector<LeafNode>());
     bl.get(0).add(new LeafNode());

//step 2
     int lim=2; lim=CatNum;
     for (int n=0; n <=CatNum; n++) {
          Vector<Node> L = new Vector();
          bl.add(L);
     for (int row = 0; row < bl.get(n).size(); row++) {
          String as="row"+"#"+(row+1);
          if(n==lim) System.out.println("row "+(row+1));
          Node t = (Node)bl.get(n).get(row);
          Node s = new Node(new LeafNode(),t.copy());
          L.add(s);

//step 3
```

```
    int remainingBranch= n−2*n1;
    for (int k = 0; k < t.remainingBranch(); k++) {
        s = t.copy();
        Node r = s;
        L.contains(r);
        Vector<Node> R = new Vector();
        r.equals(R);

//step 4
    for (int i=0; i < k; i++){
        //step 4.1
             s = s.getLeft(); L.lastElement();
             s.setLast(new Node(new LeafNode(), s.getLeft()));
        //step 4.2
             L.contains(r);
             L.add(r); }
            }
        }
    return bl.get(CatNum);
    }
```

Java izvorni kôd za metodu `contains()` koja odgovara Algoritmu 3.2.2:

```
public int contains() {
 int a=this.contains()−this.left.contains();
 int b=this.right.contains()−(this.contains()−this.left.contains());
     for (i=0; i<=(n−4); i++){
           if((a−b)==2) { L.elimination();}
           }
     return L.remainingBranch();
     }
```

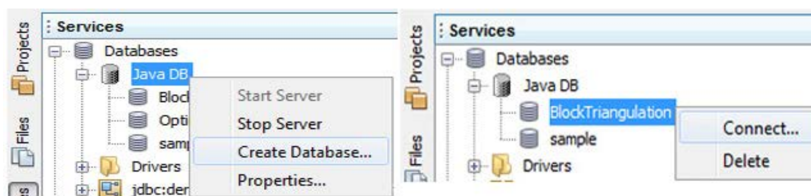
Java izvorni kôd za metodu `elimination()` koja odgovara Algoritmu 3.2.1:

```
 public int elimination(){
     int t=this.right.elimination()−this.left.elimination();
     return t;
     }
```

JAVA API za rad sa bazama podataka je JDBC (*Java DataBase Connection*). Ovaj API je baziran na SQL jeziku. Pomoću klasa iz paketa *AWT* i *Swing* mogu se izvršavati SQL naredbe i manipulisati rezultatima. *JDBC* poziva SQL interfejs za *Javu*: `import java.sql.Connection;` i `import javax.sql.DataSource;`.

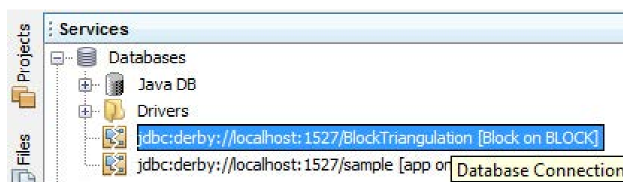
Za implementaciju *blok metode* se uspostavlja baza podataka *Java DB*-*Sun*-ove distribucije *Apache Derby* u okviru *JDK* 6. Na ovaj način, moguće je da se koristi bilo koja druga baza podataka koja obezbeđuje *JDBC* drajver. *NetBeans* okruženje obezbeđuje lak interfejs za kreiranje baza podataka i uspostavljanje konekcije, jer je odgovarajući *Java DB* drajver dostupan u okviru *NetBeans*-a [7].

Najpre je potrebno pokrenuti server. Nakon toga, sledi kreiranje baze podataka za skladištenje blokova triangulacije (Slika 3.13). Kreirana baza sadrži tabele koje su u ovoj implementaciji nazvane $T[n]$, gde je n broj temena poligona P_n za koji se beleže triangulacije.



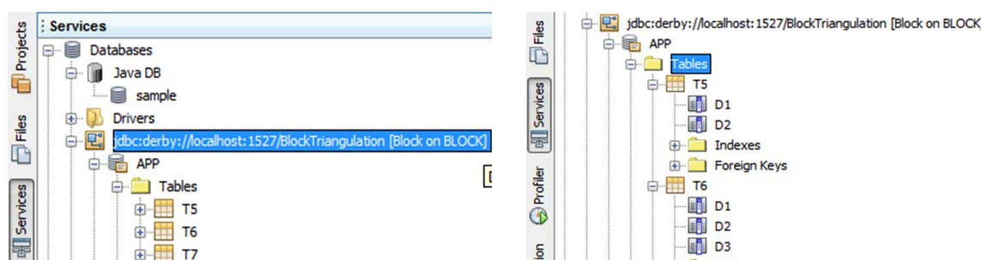
Slika 3.13: Kreiranje baze podataka za blokove triangulacija

JDBC URL je oblika: *jdbc* :< *subprotocol* >:< *subname* >, gde je < *subprotocol* > ime vrste mehanizma pristupa bazi koji je podržan od jednog ili više drajvera. U slučaju implementacije za ovu metodu *JDBC URL* je: *derby://localhost:1527/BlockTriangulation[Block]*.



Slika 3.14: Mehanizam pristupa bazi podataka

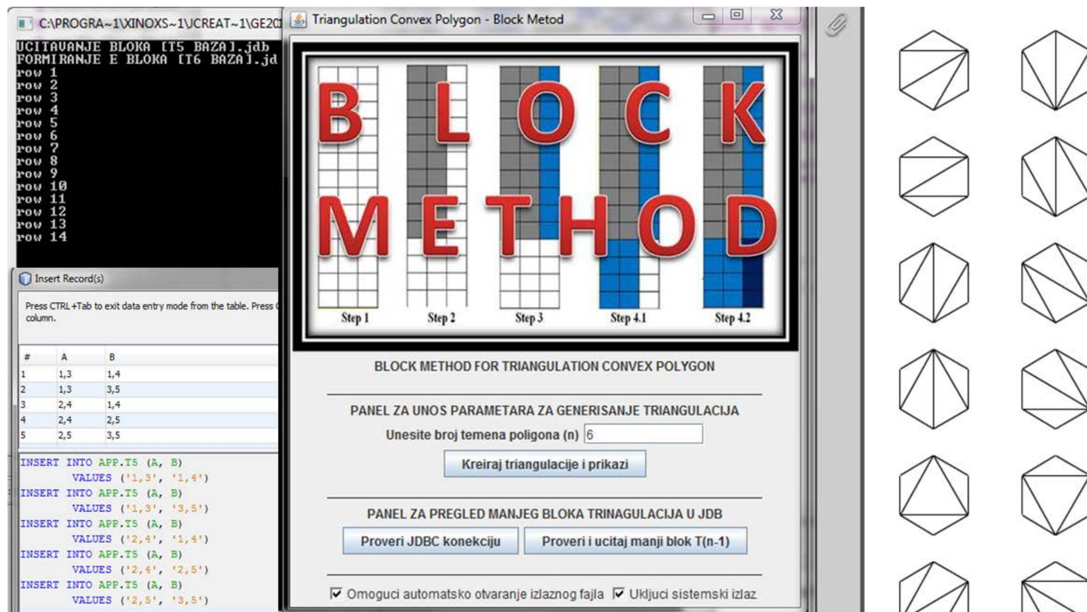
Na Slici 3.15 (levo) su prikazane tabele baze *BlockTriangulation*. Svaka tabela sadrži $n-3$ kolona (Slika 3.15, desno), što je ekvivalentno broju dijagonala poligona P_n. Tabela $T5$ ima dve kolone (D1 i D2); $T6$ ima tri kolone itd.



Slika 3.15: Generisanje tabela baze *BlockTriangulation*

Pre početka generisanja triangulacija za P_n, neophodno je proveriti da li je uspostavljena *JDBC* konekcija. Nakon izveštaja o uspešno uspostavljenoj konekciji, sledi provera da li postoji blok koji odgovara tabeli $T[n-1]$ u bazi podataka. Tek tada je moguće izvršiti generisanje triangulacija i obaviti njihovo skladištenje u $T[n]$. Na kraju disertacije (Prilog II-E) je naveden deo klase `Database` koja je zadužena za proveru *JDBC* konekcije sa bazom podataka.

Generisanje triangulacija šestougla na osnovu bloka triangulacija koje odgovaraju petouglu je prikazano na Slici 3.16.



Slika 3.16: *Java* aplikacija za *blok metodu*

Poglavlje 4

Metode za notaciju i skladištenje triangulacija poligona

Ovo poglavlje obrađuje triangulacije sa aspekta zapisivanja (pridruživanja odgovarajuće notacije triangulacijama) i njihovog skladištenja. Predstavljene su nove metode sa ciljem uštede memorijskog prostora. Napravljena je veza između problema triangulacije poligona i kombinatornih problema, kao što su: problem glasačkih listića (*ballot problem*) i problem puteva u mreži (*lattice path*).

Prva tehnika zapisivanja je nazvana *ballot notacija*. Druga notacija je nazvana *alfanumerička (AN)*. Dobijeni eksperimentalni rezultati pokazuju da se primenom navedenih notacija ostvaruje značajna uštedu memorije u procesu skladištenja triangulacija. Naučni radovi autora na kojima se bazira četvrto poglavlje su [49, 55].

4.1 Ballot notacija za triangulaciju poligona

Prva metoda za konstrukciju i skladištenje triangulacija konveksnog poligona, koja je predstavljena u ovom poglavlju, je *ballot* notacija. Osnovna motivacija za ovu metodu je proizašla iz dva kombinatorna problema: *ballot problem* i *lattice path*. Metoda se zasniva na tzv. kretanju kroz poligon na osnovu *ballot* zapisa.

Predstavljena su dva algoritma: *triangulacija u ballot* i *ballot u triangulaciju*. Da bi metoda za kodiranje *ballot* zapisa (ili odgovarajuće triangulacije) bila još kompaktnija korišćena je struktura steka. Svi navedeni algoritmi (metode) su implementirani u programskom jeziku *Java*.

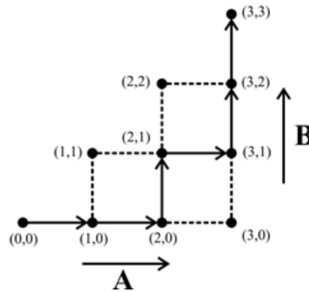
4.1.1 Uvodne napomene i osnovne postavke

Triangulacija poligona je postupak koji oduzima veliko vreme za generisanje i zahteva veliki memorijski prostor za skladištenje rezultata. Prema tome, veoma je važno obezbediti efikasan mehanizam za generisanje triangulacija poligona i njihovo trajno skladištenje.

Ballot problem u kombinatorici je opisan u radu [39]. Polazna osnova su dva kandidata za koje se glasa (A i B). Problem se odnosi na pronalazak broja različitih načina glasanja u kojima se $2n$ glasova može rasporediti na takav način da zadovoljava sledeće uslove:

- Da broj glasova koji je dobio kandidat A bude veći ili jednak broju glasova koji je dobio kandidat B, i to u svakom trenutku prebrojavanja glasova.
- Da svaki kandidat dobije n glasova, što znači da je na kraju broj glasova izjednačen.

Sada ćemo razmatriti slučaj kada se od $2n$ glasača rasporedi n glasova na kandidata A i n glasova na kandidata B. *Ballot* problem može biti grafički ilustrovan pomoću putanja na mreži koja se sastoji od n^2 tačaka u Dekartovom koordinatnom sistemu [39].



Slika 4.1: Kretanje kroz putanju na mreži i odgovarajući *ballot* zapis $AABABB$

Problem se odnosi na način pronalaženja broja putanja između početnih $(0, 0)$ i odredišnih tačaka (n, n), koje ne prelaze liniju koju definišu ove dve tačke. Svaka putanja na mreži može biti kodirana pomoću određenog niza vektora koji definišu pomeraj nadesno ili nagore. Pomeraj nadesno označava kretanje iz tačke $P(x, y)$ do tačke $P_1(x + 1, y)$, dok pomeraj nagore označava pomeraj od tačke $P(x, y)$ do $P_2(x, y + 1)$. U kombinaciji sa *ballot* problemom bilo koji pomeraj nagore odgovara glasu B, dok bilo koji pomeraj nadesno odgovara glasu A. Svaka putanja koja se sastoji od $2n$ pomeraja u $n \times n$ mreži odgovara *ballot* zapisu dužine $2n$ (videti Sliku 4.1).

Primer 4.1.1. *Tabela 4.1 je formirana od niza glasanja redosleda* ***AABABB****. Ovaj ballot zapis odgovara kretanju kroz putanju mreže sa Slike 4.1. Celi brojevi redom označavaju do tada izbrojan broj pojava specifičnog glasa.*

Tabela 4.1: Redosled glasanja u *ballot* zapisu $AABABB$

Kandidat	***A***	***A***	***B***	***A***	***B***	***B***
A	1	2	2	3	3	3
B	0	0	1	1	2	3

Dve matrice reda n, označene sa $\mathcal{A}_n$ i $\mathcal{B}_n$, mogu biti formirane na osnovu svih validnih nizova glasanja za oba kandidata. Elementi u matricama $\mathcal{A}_n$ i $\mathcal{B}_n$ označavaju broj glasova u trenutku dodavanja svakog novog glasa. Sa druge strane, matrice $\mathcal{A}_n$ i $\mathcal{B}_n$ sadrže prvu i drugu koordinatu za tačke koje definišu putanju unutar $n \times n$ mreže, respektivno. Ove matrice mogu biti transformisane u jednu matricu, koju možemo označiti sa $\mathcal{R}_n$, čiji redovi odgovaraju *ballot* zapisu.

Primer 4.1.2. *Razmotrimo sada konkretan slučaj od $2n = 6$ glasača, gde tri glasača daju glas kanidatu A i tri glasaju za kandidata B. Dve matrice $\mathcal{A}_3$ i $\mathcal{B}_3$ se formiraju kako bi zabeležili sve ballot zapise. Matrica $\mathcal{A}_3$ sadrži prvu koordinatu a matrica $\mathcal{B}_3$ drugu koordinatu za tačke koje definišu putanju na mreži. Ove matrice mogu biti predstavljene u formi matrice sa ballot zapisima ($\mathcal{R}_3$).*

$$\mathcal{A}_3 = \begin{bmatrix} 1 & 2 & 3 & 3 & 3 & 3 \\ 1 & 2 & 2 & 3 & 3 & 3 \\ 1 & 1 & 2 & 3 & 3 & 3 \\ 1 & 1 & 2 & 2 & 3 & 3 \\ 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}, \quad \mathcal{B}_3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 1 & 2 & 3 \\ 0 & 1 & 1 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 & 2 & 3 \\ 0 & 0 & 1 & 2 & 2 & 3 \end{bmatrix}, \quad \mathcal{R}_3 = \begin{bmatrix} \mathrm{A\ A\ A\ B\ B\ B} \\ \mathrm{A\ A\ B\ A\ B\ B} \\ \mathrm{A\ B\ A\ A\ B\ B} \\ \mathrm{A\ B\ A\ B\ A\ B} \\ \mathrm{A\ A\ B\ B\ A\ B} \end{bmatrix}.$$

Elementi matrice $\mathcal{A}_3$ i $\mathcal{B}_3$ ukazuju na broj pojavljivanja znakova A i B, resepktivno, u vrsti matrice $\mathcal{R}_3$. Broj validnih kombinacija, tj. broj redova u matrici $\mathcal{R}_3$ je jednak $C_3 = 5$.

U sledećoj sekciji, predstavljena je primena *ballot problema* u konstrukciji i skladištenju triangulacija konveksnog poligona. Data su dva inverzna algoritma. Prvi algoritam se odnosi na transformaciju iz *ballot* zapisa u triangulaciju i služi za generisanje triangulacija konveksnog poligona na osnovu odgovarajućih *ballot* zapisa koji su zabeleženi u matrici. Inverzni algoritam generiše *ballot* zapise koji odgovaraju triangulacijama.

4.1.2 Ballot problem i triangulacija poligona

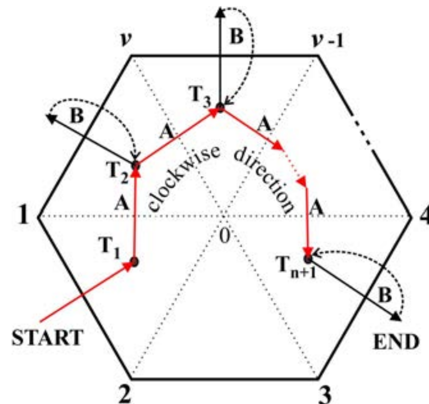
Postupak kretanja kroz mrežu u kombinaciji sa *ballot* zapisom može poslužiti kao dobra ideja za kreiranje metode za notaciju i skladištenje triangulacija. Bitno je obezbediti i reverzni postupak na osnovu koga možemo dobiti grafičku prezentaciju triangulacija.

Koristimo osnovnu postavku da broj validnih *ballot* zapisa dužine $2n$ odgovara konveksnom poligonu sa $v = n + 2$ temena. Konveksni poligon je podeljen na v trougaonih oblasti, sa v imaginarnih linija koji povezuju temena sa centrom poligona koji je označen sa 0.

Trouglovi su označeni sa $\triangle(0, k, l)$, gde su k i l susedna temena poligona. Unutar svakog trougla izabrana je jedna unutrašnja tačka, osim za trougao $\triangle(0, 2, 3)$. Te tačke su označene sa T_j, gde je $j = 1, \ldots, n + 1$.

Može se pretpostaviti da je poligon pravilan i da su trouglovi karakteristični jednakokraki trouglovi, dok za njihove centre T_j možemo izabrati njihova težišta.

Slika 4.2 prikazuje generalnu formu kretanja kroz poligon, gde se kretanje zasniva na pojavljivanju znaka A i B u *ballot* zapisu. Glavna putanja (unutar poligona) je definisana pojavom znaka A, dok je skretanje sa unutrašnje putanje prema odgovarajućoj spoljašnjoj stanici definisano pojavom znaka B. Na ovaj način se uvek može odrediti oblast kroz koji se vrši kretanje.



Slika 4.2: Generalna forma kretanja kroz poligon

Karakteristike kretanja kroz poligon koje je zasnovano na *ballot* zapisu su:

1. Osnovno pravilo je da kretanje počinje od stranice $\delta_{1,2}$ i vodi prema tački T_1 (početak putanje je uvek označen znakom A). Kretanje se vrši u smeru kazaljke na satu.

2. Dva moguća kretanja su dozvoljena od centra T_j: kretanje prema središtu sledećeg neposećenog trougla u smeru kazaljke na satu (odgovara znaku A) ili kretanje prema spoljašnoj ivici sledećeg neposećenog trougla u smeru kazaljke na satu (odgovara znaku B). Trougao se smatra posećenim ako smo prešli preko njegove spoljašnje ivice. Takođe, trougao je posećen i ako smo prešli direktno na njegovu spoljašnju ivicu.

3. Procedura se završava kada su posećene sve spoljašne stranice poligona (označene znakom B) osim završne ivice $\delta_{2,3}$, koja je susedna sa početnom ivicom i formira zatvorenu konturu poligona.

Sada ćemo predstaviti algoritam za konstrukciju triangulacije na osnovu datog *ballot* zapisa. Algoritam 4.1.1 na ulazu očekuje *ballot* zapis, tj. red matrice $\mathcal{R}_n$, koji sadrži $2n$ znakova (A ili B). Poznato je da postoji C_n različitih validnih *ballot* zapisa [39].

Algoritam 4.1.1 Transformacija iz *ballot* zapisa u triangulaciju (kretanje kroz poligon)

Ulaz: *ballot* zapis, označen sa $b = \{b_1, \ldots, b_{2n}\}$.

1: Kreira poligon sa $v = n + 2$ temena
   Pronalazi centre T_j, $j = 1, \ldots, n + 1$.
   Postavlja niz $visited_j$, $j = 1, \ldots, n + 1$, na *false*.
   Započinje kretanje od $\delta_{1,2}$ stranice u smeru kazaljke na satu.
   Postavlja $k = 1$, $j = 1$, i početnu tačku kretanja na $P_k = T_1$ (zato što je uvek $b_1 = A$).

2: for $i = 2$ to $2n$
   2.1: Ako je trenutni znak u *ballot* zapisu $b_i = A$, onda pronalazi najmanji $j_2 \geq j$ gde je $visited_{j_2} = false$.
   Postavlja $j = j_2$, $k = k + 1$, $P_k = T_j$.
   2.2: Ako je trenutni znak u *ballot* zapisu $b_i = B$, onda pronalazi najmanji $j_2 \geq j$ gde je $visited_{j_2} = false$ i postavlja $visited_{j_2} = true$.
      2.2.1: Ako su dva susedna znaka B u b onda se vraća za još jedan čvor nazad tako što će se postaviti $j = j - 1$.

3: Crta unutrašnje dijagonale na osnovu Algoritma 4.1.2.

Izlaz: Generisana triangulacija na osnovu odgovarajućeg *ballot* zapisa sa ulaza.

Algoritam 4.1.2 Pronalaženje mogućih ukrštanja u poligonu

Ulaz: Putanja P_k, $k = 1, \ldots, n$ unutar poligona.

1: for $i = 1$ to m; gde je m broj svih dijagonala poligona, $m = v(v - 3)/2$
   1.1: Biranje dijagonale iz skupa unutrašnjih dijagonala ($\{\delta_{1,3}, \ldots, \delta_{1,v}\}$, $\{\delta_{2,4}, \ldots, \delta_{2,v}\}$; itd.).
      1.1.1: Ako izabrana dijagonala preseca samo jedan deo putanje P_k i ako se ne ukršta sa prethodnom dijagonalom, ONDA crta dijagonalu.
      1.1.2: U suprotnom bira sledeću dijagonalu (ide na korak 1.1).

Izlaz: $v - 3$ unutrašnjih dijagonala.

Implementacija procedure pronalaženja mogućih ukrštanja u poligonu (Algoritam 4.1.2) u Javi može da se odradi uz pomoć klase *Triangulator* iz paketa *GeometryInfo* (*import com.sun.j3d.utils.geometry*). Konkretnije, proces pronalaska preseka može biti realizovan primenom referentne metode klase *Triangulator* u kombinaciji sa metodom *intersects()* iz klase *Polygon.*

Primer 4.1.3. *Ilustrujmo kako Algoritam 4.1.1 kreira triangulaciju petougla na osnovu ballot zapisa **AABABB** (koji odgovara kretanju kroz putanju na mreži sa Slike 4.1). Proces ilustrovan na Slici 4.3 je opisan pomoću sledećih koraka:*

(1) Prvi znak u ballot zapisu je A, kretanje počinje od početne strane $\delta_{1,2}$ i $P_1 = T_1$.

(2) Drugi znak je A, i kretanje ide unutar poligona u smeru kazaljke na satu prema tački $P_2 = T_2$.

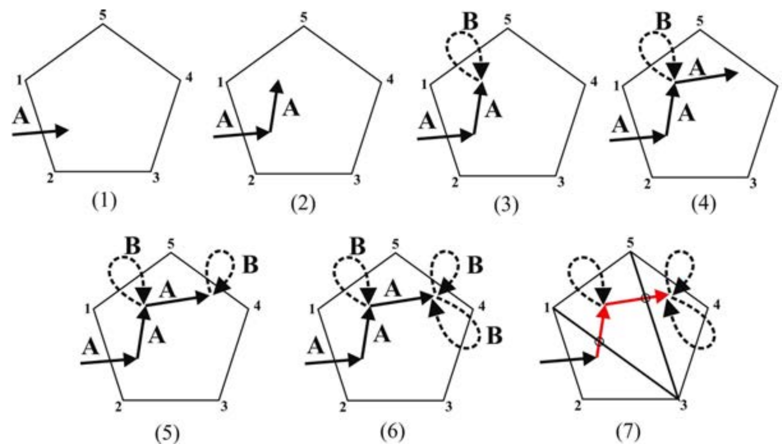
(3) Sledeći znak u ballot zapisu je B - kretanje obilazi oko sledeće neposećene spoljašnje ivice ($\delta_{1,5}$) i vraća se u tačku T_2 (dostignuta u prethodnom koraku). Postavlja parametar $visited_2 = true$.

(4) Sledeći znak je A - kretanje nastavlja u smeru kazaljke na satu, prema $P_3 = T_3$.

(5) Sledeći znak je B - kretanje obilazi oko sledeće leve spoljašnje stranice $\delta_{4,5}$ i vraća se u čvor T_3. Postavlja parametar $visited_3 = true$.

(6) Sledeći znak u zapisu je B - kretanje se nastavlja obilazeći oko sledeće leve spoljašnje stranice ($\delta_{3,4}$). Vraća se nazad za još jednu tačku, tj. u T_2. Postavlja parametar $visited_4 = true$. Obzirom da nema više znakova u ballot zapisu kretanje je završeno.

(7) Po pređenom putu P unutar poligona, odgovarajuća triangulacija može biti konstruisana. Na osnovu Algoritma 4.1.2 se iscrtavaju sve moguće unutrašnje dijagonale. U ovom slučaju, moguće nepresecajuće unutrašnje dijagonale koje seku datu putanju označenu sa A (tj. sa T_1, T_2, T_3) su $\delta_{1,3}$ i $\delta_{3,5}$.



Slika 4.3: Konstrukcija triangulacije zasnovana na *ballot* zapisu $AABABB$

Sada ćemo predstaviti algoritam za kodiranje određene triangulacije konveksnog poligona sa odgovarajućim *ballot* zapisom (suprotno Algoritmu 4.1.1). Ova procedura se može nazvati "kretanje kroz definisanu triangulaciju". Kretanje se opet vrši u smeru kazaljke na satu. Jedan od dva znaka A ili B se pojavljuje kao rezultat svakog pojedinačnog poteza. Kada se pređe preko unutrašnje dijagonale - piše se znak A, u suprotnom za spoljašnju ivicu - piše se znak B.

U procesu kretanja, sve spoljašnje ivice i dijagonale poligona tretiraju se kao linije (ivice) kroz koje se prolazi. Koristi se rastojanje između dva temena poligona da bi se napravila razlika između spoljašnje ivice i unutrašnje dijagonale (videti Definiciju 3.2.1).

Algoritam 4.1.3 očekuje triangulaciju konveksnog poligona na ulazu i proizvodi odgovarajući *ballot* zapis.

Algoritam 4.1.3 Transformacija iz triangulacije u *ballot* zapis (kretanje kroz triangulaciju)

Ulaz: Triangulacija konveksnog poligona.

1: Kretanje počinje od početne stranice $\delta_{1,2}$ (označen znakom A), u smeru kazaljke na satu.

2: for $i = 2$ to $2n$

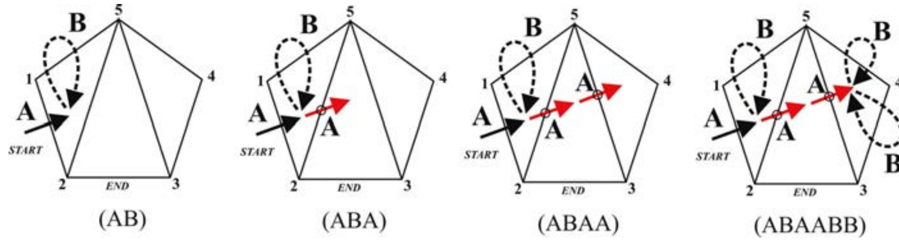
2.1: Ako je sledeća stranica $\delta_{p,q}$, gde je $d(p,q) > 1$ (unutrašnja dijagonala) onda se smešta znak A u izlazni string i registruje prolazak preko te dijagonale.

2.2: Ako je sledeća stranica $\delta_{p,q}$, gde je $d(p,q) = 1$ (spoljašnja ivica), onda se smešta znak B u izlazni string i vraća se u čvor kako bi se moglo nastaviti kretanje kroz triangulisani poligon.

2.2.1: Ako smo poslali dva uzastopna znaka B u izlazni string onda se vraća za još jedan čvor unazad.

Izlaz: Odgovarajući *ballot* zapis.

Primer 4.1.4. *Ilustrovaćemo primenu Algoritma 4.1.3 u procesu kretanja kroz triangulaciju petougla definisanu unutrašnjim dijagonalama $\delta_{2,5}$ i $\delta_{3,5}$ (Slika 4.4). Na ovaj način se dobija odgovarajući ballot zapis:* ***ABAABB***.



Slika 4.4: Kretanje kroz triangulaciju $\delta_{2,5}$ i $\delta_{3,5}$

4.1.3 Primena steka za skladištenje triangulacija

U ovoj sekciji, predstavićemo mogućnost primene strukture steka za skladištenje triangulacija. Stek je apstraktan tip i struktura podataka, zasnovana na principu *LIFO (last in, first out)*, sa dve osnovne operacije: *push* i *pop*.

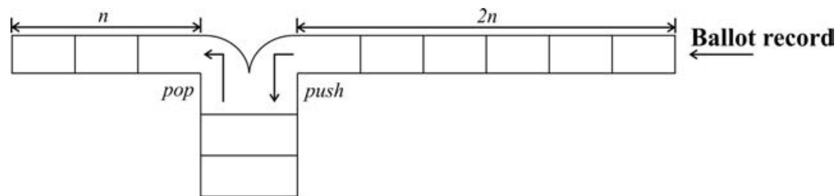
Napravićemo vezu između operacija nad stekom i znakova koji se pojavljuju kod *ballot notacije*:

1. Ako je trenutni znak u *ballot* zapisu A, poziva se operacija *push* i stavlja se broj pojavljivanja ovog znaka u *ballot* zapisu sa leve strane.

2. Kada se pojavljuje znak B, poziva se operacija *pop* i izbacuje se broj iz steka na izlazu.

U [21] i [39] prikazano je da broj permutacija koje se koriste u steku odgovara Katalanovom broju. Prema tome, primenom steka se može jedinstveno preslikati svaki *ballot* zapis dužine $2n$ na jednu permutaciju skupa $\{1, \ldots, n\}$.

Na ulazu se očekuje *ballot* zapis, i pošto operacija *push* snima samo broj pojavljivanja znaka A dobija se dva puta kraći izlaz (na Slici 4.5 je prikazan odnos ulaza i izlaza steka).

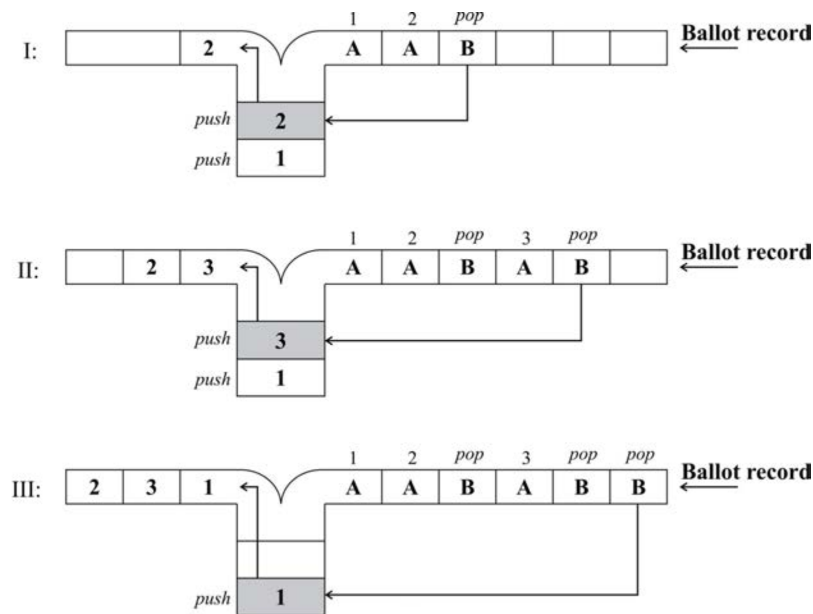


Slika 4.5: Odnos ulaza i izlaza steka, $2n : n$

Primer 4.1.5. *Neka* ***AABABB*** *bude ulazni ballot zapis. Prva dva znaka u ovom ballot zapisu su AA, što znači da se stavlja broj 1 i broj 2 na stek. Naredni znak je B, tako da se izbacuje broj 2 i šalje na izlaz (Slika 4.6- I).*

Sledeći znak u ballot zapisu je A, tako da se šalje broj 3, i nakon pojavljivanja sledećeg znaka B ova vrednost se izbacuje i šalje na izlaz. Sada, izlazni string sadrži brojeve: 2 i 3 (Slika 4.6- II).

Konačno, posle poslednjeg znaka u ballot zapisu (znak B), vrednost 1 se izbacuje i šalje na izlaz steka. Stek sada sadrži tri vrednosti: 2,3,1 (Slika 4.6- III).



Slika 4.6: Primer obrade *ballot* zapisa pomoću steka

Izlazne vrednosti dobijene iz steka (koristićemo skraćeno SVB - stek vrednosti na osnovu *ballot*) su određene pomoću sekvence operacija nad stekom zasnovanih na *ballot* zapisu.

Primer 4.1.6. *Za triangulacije petougla imamo pet različitih ballot zapisa i pet njima odgovarajućih SVB vrednosti:*

$$ABABAB \to 123, \quad ABAABB \to 132, \quad AABBAB \to 213$$
$$AABABB \to 231, \quad AAABBB \to 321.$$

Sada je potrebno omogućiti da se iz SVB vrednosti dobijenih pomoću steka dobije odgovarajući *ballot* zapis (obrnut proces). Ovo je naročito bitno zbog konstrukcije triangulacija, jer ako imamo skladištene SVB neophodno je imati odgovarajuće *ballot* zapise na osnovu kojih ćemo izvršiti generisanje triangulacija. U nastavku je predložen algoritam za mapiranje SVB zapisa u *ballot* zapis.

Algoritam 4.1.4 Mapiranje SVB u *ballot* zapis

Ulaz: Niz P_i, $i = 1, \ldots, n$ sadrži permutaciju skupa $\{1, \ldots, n\}$ dobijenu koristeći stek.

1: for $i = 1$ to n do
- 1.1: Ako je $P_i > 0$ pošalji P_i znakova A na izlaz.
- 1.2: Pošalji jedan znak B na izlaz.
- 1.3: Ako je $P_i > 0$ za $j = i + 1$ sve do n umanji P_j za P_i.

Izlaz: Odgovarajući *ballot* zapis.

4.1.4 Komparativna analiza i eksperimentalni rezultati

U cilju evaluacije prezentovane metode predstavljena je komparativna analiza sa *Hurtado-Noy* algoritmom [29]. Obe metode su implementirane u *Javi*. U ispitivanju je akcenat stavljen na uštedu memorijskog prostora (u toku izvršavanja i pri trajnom skladištenju rezultata).

Radi bolje komparacije, obe metode su analizirane kroz tri faze:

- *I (Input)*: ulazni podaci koji su potrebni da bi se otpočeo proces generisanja,
- *G (Generate)*: proces generisanja triangulacija,
- *O (Output)*: izlazni podaci, koji uključuju grafički prikaz triangulacije i skladištenje.

Faze za Hurtada (H): H_i - triangulacije opisane sa $2v - 5$ parova temena; H_g - generisanje triangulacije na osnovu hijerarhije (stablo triangulacija); H_o - izlazna datoteka sa skupom od $2v - 3$ parova temena koja opisuju triangulaciju.

Faze za ballot (B): B_i - *ballot* zapis; B_g - Kretanje kroz triangulaciju na osnovu *ballot* zapisa (Algoritam 4.1.1); B_o - Izlazna datoteka sa SVB zapisima.

Tabela 4.2 prikazuje rezultate testiranja obe metode za $v = 5, \ldots, 16$. Testiranje je izvršeno u *NetBeans* modulu *"Profile Main Project / CPU Analyze Performance"* na konfiguraciji: *CPU - Intel(R) Core(TM)2Duo T7700, 2.40 GHz, L2 Cache 4 MB (On-Die, ATC, Full-Speed), RAM 2 Gb, Graphic card - NVIDIA GeForce 8600M GS.*

Rezultati testiranja su analizirani kroz dva aspekta:

- **Brzina izvršavanja**: Vreme koje se odnosi na fazu generisanja triangulacija bez njihovog skladištenja (samo faza H_g i B_g) i ukupno vreme za sve tri faze: ulaz, generisanje triangulacija i njihovo skladištenje (oznake u tabeli H_{all} i B_{all}).
- **Memorija**: Zauzetost radne memorije (bafera) za vreme generisanja svih triangulacija za dato n (faze H_o; B_o).

Tabela 4.2: Eksperimentalni rezultati za primenu *ballot* notacije

v	**Broj triang.**	**CPU vreme (u sek.)**				**Memorija (u Kb)**	
		H_{all}	H_g	B_{all}	B_g	H_o	B_o
5	5	**0.25**	0.21	**0.24**	0.22	0.12	**0.02**
6	14	**0.34**	0.29	**0.33**	0.31	0.46	**0.08**
7	42	**0.43**	0.35	**0.41**	0.38	1.76	**0.29**
8	132	**0.49**	0.40	**0.47**	0.44	6.46	**1.05**
9	429	**0.67**	0.55	**0.63**	0.60	24.49	**3.86**
10	1,430	**1.18**	0.89	**1.03**	0.98	93.13	**14.30**
11	4,862	**3.81**	2.19	**3.51**	3.09	355.67	**53.40**
12	16,796	**12.46**	5.81	**11.29**	10.11	1,363.43	**196.00**
13	58,786	**50.51**	15.24	**43.55**	38.81	4,121.13	**502.00**
14	208,012	**119.05**	46.34	**108.07**	97.13	11,523.62	**1,441.00**
15	742,900	**318.63**	124.18	**274.19**	244.02	29,874.29	**4,295.00**
16	2,674,440	/	/	**677.17**	572.87	/	**12,752.00**

U slučaju *Hurtado-Noy* algoritma, vrši se snimanje skupa od $(2v-3)$ parova temena, jer je svaka triangulacija u trenutku generisanja opisana kao struktura koja sadrži toliko parova temena. Implementacija i eksperimentalni rezultati ovakvog načina skladištenja dati su u radu [65] a deo tih rezultata je prikazan u Tabeli 4.2 (kolone H_{all} i H_o). Kod *ballot* metode, skladište se izlazni *SVB* zapisi sa steka (kolona B_o) i svaki od ovih izlaza odgovara tačno jednom *ballot* zapisu.

Dobijeni rezultati predstavljeni u Tabeli 4.3 daju komparaciju CPU vremena i zahteva za skladištenje za testirane metode. Konkretnije, razmatraju se razlike u vremenima za fazu generisanja triangulacija $(H_g - B_g)$ ali i za istovremeno generisanje i skladištenje $(H_{all} - B_{all})$. Takođe, date su razlike u veličini zahtevane memorije $(H_o - B_o)$ kao i procentualni udeo vremena potrebnog za skladištenje (H_s; B_s) za vreme izvršavanja programa.

Tabela 4.3: Komparacija metoda: *ballot* i *Hurtado-Noy*

v	**CPU vreme (u sek.)**		**Memorija (u Kb)**	**Skladištenje (u %)**	
	$H_{all} - B_{all}$	$H_g - B_g$	$H_o - B_o$	H_s	B_s
5	**0.01**	-0.01	**0.10**	16.00	**8.33**
6	**0.01**	-0.02	**0.38**	17.71	**8.46**
7	**0.02**	-0.03	**1.47**	18.60	**8.55**
8	**0.02**	-0.04	**5.41**	19.37	**8.71**
9	**0.04**	-0.05	**20.63**	20.91	**8.79**
10	**0.15**	-0.09	**78.83**	24.58	**8.85**
11	**0.30**	-0.90	**302.27**	42.52	**10.97**
12	**1.17**	-4.30	**1,167.43**	53.37	**11.25**
13	**6.96**	-23.57	**3,619.13**	59.83	**11.48**
14	**10.98**	-50.79	**10,082.62**	61.08	**11.62**
15	**44.44**	-119.84	**25,579.29**	63.19	**11.97**

Na osnovu dobijenih rezultata se može zaključiti:

- *Ballot* metod daje bolje rezultate za slučaj gde je uključeno i generisanje i skladištenje istovremeno ($H_{all} - B_{all}$). Za ukupno vreme za svih 11 testiranja (za $v \in \{5, 6, ..., 15\}$) postignuto je prosečno ubrzanje od 1.09, odnosno smanjenje vremena za 9%. Postizanje boljih rezultata je omogućeno primenom *SVB* izlaza ($H_o - B_o$), iz razloga što se zahteva znatno manje podataka za vreme izvršavanja. Razlika u veličini zahtevane memorije za *Hurtado* i *ballot* metod je data u Tabeli 4.3.

- *Hurtado-Noy* algoritam daje bolje rezultate ako se uzme u obzir samo vreme za fazu generisanja triangulacija ($H_g - B_g$).

- Sa S smo označili udeo vremena za skladištenje (u %) u odnosu na ukupno vreme (H_{all}, B_{all}). Na osnovu numeričkih podataka, vrednost S se značajno razlikuje kod dve navedene metode. Primenom *SVB* izlaza, B_s je znatno manji za *ballot* metod (od 8% do 12% za $v \in \{5, \ldots, 15\}$). Kod skladištenja preko *Hurtado-Noy* algoritma, vrednost H_s se kreće od 16% do 63%. Ova ušteda u vremenu koje je potrebno za skladištenje međurezultata značajno utiče na ukupne rezultate (vreme generisanja triangulacija u oba oblika: grafički prikaz i skladištenje).

Prednosti ovakvog tipa snimanja trenutnih rezultata su višestruke. Dobija se značajnija ušteda u memorijskom prostoru (ovde se podrazumeva i radna memorija i veličina izlazne datoteke).

Ušteda u radnoj memoriji se reflektuje kroz postizanje boljeg vremena u procesu generisanja triangulacija, kao i na obezbeđivanje efikasnog generisanja triangulacija za poligone P_n, gde je $n > 20$.

4.1.5 Detalji implementacije u *Javi*

Implementacija metode je realizovana kroz sledeće klase: `Triangulation, BallotPath`, `NoteTriangulation`, `DrawTriangulations, Nodes, LNodes, ListBallot` i `ValidBallot`.

A) Klasa `Triangulation` se koristi za generisanje triangulacija. *Java* metod `BallotNotation()`, pripada klasi `NoteTriangulations` (Prilog II-C) i kreira triangulacije na osnovu zapisa dobijenih od metode koja pripada klasi `BallotPath`. Ova metoda odgovara opisanom Algoritmu 4.1.1:

```
public void BallotNotation() {
//Require
    ListingBallot = new Vector[2n];
    polygon = new BallotPath();

//STEP 1 - create polygon with v-vertices
    for( int v=1 ; v<n+2 ; v++ ) {
        //drawing regular convex polygons
              double edge = (a*n+b-c*v)*Math.PI/(d*n+e);
              Fsinus[v]   = (int)Math.floor(60*Math.sin(edge));
              Fkosinus[v] = (int)Math.floor(60*Math.cos(edge));   }
//STEP 2
    //Processing of ballot record at the entrance
    for( int i = 0 ; i <2n+1 ; i++ )
            ListingBallot[i] = new Vector();
            ListingBallot[0].addElement( new LNodes() );
//step 2.1
         if (ballChar=="A"){
            for( int p= 0; p< ListingBallot[i].size(); p++ ) {
            for( int q= 0; q< ListingBallot[r-i-1].size() ; q++ ) {
                ListingBallot[r].add(p,new Nodes(
                    (ListBallot)ListingBallot[r-i-1].elementAt(q),
                    (ListBallot)ListingBallot[i].elementAt(p)));}
            }
        }
//step 2.2:
        else{
            ListingBallot[r].add(b,a);
            // two consecutive B's
            if (a==b) p--; //step 2.2.1
            }
//STEP 3 - Algorithm for intersect
                ValidPath = new Vector[n];
                path = new BallotPath();
                polygon.POLYGON_ARRAY(i);
                polygon.TRIANGLE_ARRAY();
                if(polygon.intersects()==true || validD.intersects()==false){
                ValidD = new Vector[n]; }
                else{ ValidD = new Vector[0];
                for (Enumeration diag = ListingBallot[n].elements();
```

```
                diag.hasMoreElements(); ) {
                    diagonal=(ValidBallot) diag.nextElement();
                    validD.move(diagonal); }  }
            }
        }
}
```

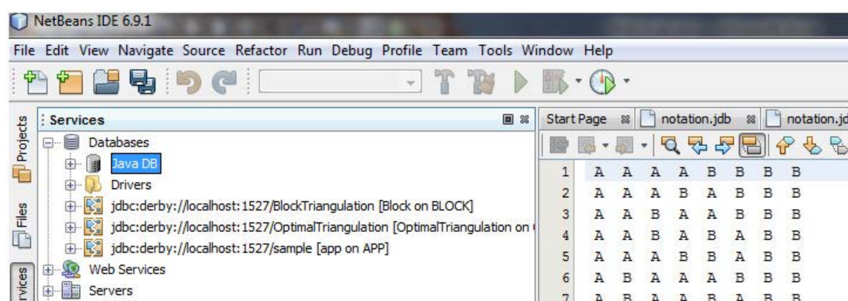
B) Kretanje kroz poligon je zasnovano na validnim *ballot* zapisima. Ovo kretanje je podržano primenom klase `Node` i njene klase naslednice `LNode`. Metode `ballot()` i `path()` iz klase `Node` obezbeđuju korelaciju između *ballot* notacije i kretanja kroz triangulaciju.

```
String ballot(){
        return First.notation()+"A"+ Second.notation()+"B";
}
String path(){
        Nodes.a++;
        Nodes.b++;
        return First. path()+ Nodes.a + Second.path()+ Nodes.b;
}
***
public void move( int aMove, ListBallot t ) {
    if (t instanceof Nodes) {
        move(aMove,((Nodes)t).First);
        move(aMove+((Nodes)t).First.leaves(),((Nodes)t).Second); }
        addElement( new Point(aMove, aMove+t.leaves()));
}
```

C) *SVB* izlazi i njima odgovarajući *ballot* zapisi se evidentiraju primenom gotove klase `BufferedWriter` (iz paketa `java.io.BufferedWriter` i `java.io.BufferedOutputStream`).

```
for (i=0; i<n; i++){
BufferedWriter bufferedWriter = null;
        try { bufferedWriter = new BufferedWriter(
                new FileWriter("SVB_output.jdb",true));
                bufferedWriter.write(SVB);
                bufferedWriter.newLine();
```

Sadržaj izlazne datoteke iz testiranja aplikacije je prikazan na Slici 4.7:



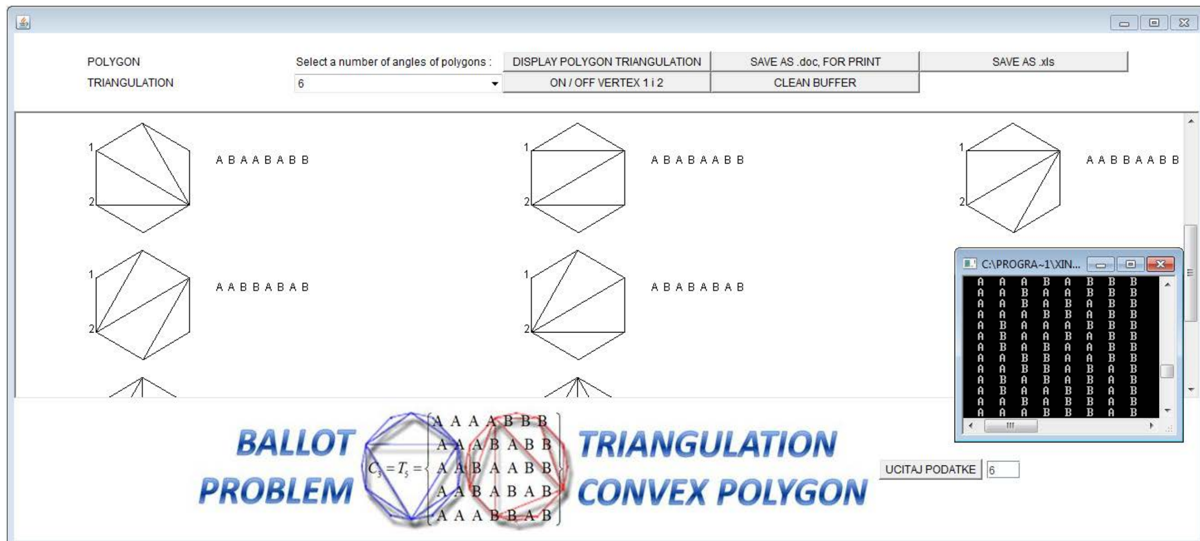
Slika 4.7: *Ballot* zapisi za određene triangulacije šestougla

D) Klasa `DrawTriangulations` u saradnji sa klasom `Triangulation`, tačnije sa njenom metodom `DisplayTriangulations()`, je zadužena za predstavljanje triangulacija na *JPanel* aplikacije. Metoda `BallotNotation()` obezbeđuje vezu između grafičkog prikaza triangulacije `(Graphics g)` sa odgovarajućim *ballot* zapisom `(String Ballot)`.

Primerak klase `Triangulation` koji je nazvan `polygon`, poziva metodu `BallotNotation` sa argumentima `g` i `t.notation()`, gde je: `g` grafička prezentacija triangulacije, `t` primerak iz `ValidBallot`, a naredba `t.notation()` spaja ova dva oblika.

```
public void DisplayTriangulations(Graphics g){
    ValidBallot t;
    int v=ListingBallot[order].size();
    polygon.init();
    ***
    for(Enumeration e=ListingBallot[n].elements(); e.hasMoreElements();){
    t=(ValidBallot) e.nextElement();
    polygon.copy(t);
    polygon.BallotNotation(g,""+t.notation()); }
}
```

Java aplikacija sadrži centralni *JPanel* i *Toolbar* sa dodatnim opcijama za snimanje i prikazivanje triangulacija, kao i učitavanje matrice sa *ballot* zapisima (u *txt* ili *jdb* formatu).



Slika 4.8: *Ballot notacija* za određene triangulacije šestougla

4.2 Alfanumerička notacija

Notacija u vidu uparenih zagrada (*balanced parentheses*, u daljem tekstu BP) je jedna od postojećih tehnika zapisivanja triangulacija konveksnih poligona. U radovima [18, 13, 34] su predstavljeni načini prezentacije Katalanovih brojeva primenom ove notacije.

U ovoj sekciji će biti opisana nova predložena tehnika za skladištenje i notaciju triangulacija (rezultati se baziraju na autorskom radu [49]). Predložena notacija je nazvana *alfanumerička* (skraćeno AN notacija) i bazira se na BP notaciji. Dobijena notacija predstavlja skraćeni oblik prethodne, a sve u cilju uštede memorijskog prostora. Data su dva algoritma koja realizuju dve transformacije (iz BP zapisa u AN i obrnuto). Predloženi metod je implementiran u *Javi*.

4.2.1 Uvodne napomene i osnovne postavke

Osnovna primena BP notacije, pored reprezentacije Katalanovih brojeva, je i u formiranju zapisa koji su ekvivalentni binarnim stablima [26, 47, 50, 71].

Kod binarnog stabla svaki čvor ima najviše dva potomka, gde se ti potomci obično tretiraju kao "levi" (L) i "desni" (R) čvor. U kombinaciji sa BP notacijom, čvor je predstavljen pomoću podudarajućeg para zagrada `"()"` i sva podstabla na korenu čvora su kodirana između uparenih zagrada [20, 73].

String od n parova zagrada je uparen ako svaka otvorena zagrada ima odgovarajuću zatvorenu zagradu. Ustanovljena su dva neophodna uslova kako bi string x bio uparen [41]:

- (i) $L(x) = R(x)$;
- (ii) Ako je $x = yz$ za neki string z, onda važi $L(y) \geq R(y)$.

Na primer, `"(()())"` je validan zapis, dok zapis `"())()("` nije validan. Najjednostavniji način za predstavljanje niza BP stringova je da koristimo *bit-string*, gde bit `1` predstavlja `"("` a bit `0` predstavlja `")"`. Na primer, *bit-string* koji odgovara BP zapisu `()((()))(())` je `101110001100`.

Broj različitih dobro formiranih nizova zagrada predstavljen *bit-stringovima* $b_{2n}b_{2n-1}\dots b_1$ jednak je Katalanovom broju [25, 39]. Na primer, postoji pet validnih BP zapisa, koji se sastoje od ukupno 6 zagrada, tačnije od 3 para uparenih zagrada ($2n$=6; $C_3 = 5$):

$$((()))\quad ()()()\quad (())()\quad ()(())\quad (()()).$$

Da bi smanjili zahteve memorije, koristićemo *bit-string* kao osnovu za uvođenje novih pravila za postupak skraćenja BP notacije.

4.2.2 Transformacija iz BP u AN zapis

Procedura transformacije iz BP u AN notaciju je realizovana Algoritmom 4.2.1. Ovaj algoritam se sastoji od 4 faze: *zamena, eliminacija, selekcija i konverzija.*

Algoritam 4.2.1 Transformacija iz BP u AN notaciju

Ulaz: BP string, sa dužinom od m znakova.

1: **Zamena**
 for $(i = 1; i \leq m; i++)$
 $(\rightarrow 1 \quad \text{i} \quad) \rightarrow 0$
 Izlaz1 = $BP1$. // *dobija se bit-string*

2: **Eliminacija**
 Brisanje prvog elementa (1) i poslednjeg elementa (0) u $BP1$.
 Izlaz2 = $BP2$. // *skraćeni bit-string*

3: **Selekcija**
 if (Slučaj 1 - biraju se grupe od po 2 bita){
 Prva grupa uzima prva dva elementa u $BP2$.
 Druga grupa uzima poslednja dva elementa u $BP2$.
 RestBP2 = $BP2$ bez Prve i Druge grupe od po dva bita.
 Prva i Druga grupa od po dva bita $\rightarrow$ Alpha2.
 Izlaz3 = Alpha2 + RestBP2.
 }
 if (Slučaj 2 - biraju se grupe od po 3 bita){
 Prva grupa uzima prva tri elementa u $BP2$.
 Druga grupa uzima poslednja tri elementa u $BP2$.
 RestBP2 = $BP2$ bez Prve i Druge grupe od po tri bita.
 Prva i Druga grupa od po tri bita $\rightarrow$ Alpha3.
 Izlaz3 = Alpha3 + RestBP2.
 }

4: **Konverzija**
 if (Slučaj 1){
 RestBP2 $\rightarrow$ DecimalEq2.
 Izlaz4 = Alpha2 + DecimalEq2.
 }
 if (Slučaj 2){
 RestBP2 $\rightarrow$ DecimalEq3.
 Izlaz4 = Alpha3 + DecimalEq3.
 }

Izlaz: AN notacija

Sledi opis Algoritma 4.2.1 po fazama:

1. **Zamena**: Forma binarnog ekvivalenta za datu BP notaciju je nazvana *bit-string*. Bit-string se generiše u ovoj fazi postupkom zamene otvorene zagrade bitom `1` i zatvorene zagrade bitom `0`.

2. **Eliminacija**: U ovoj fazi se vrši uklanjanje prvog i poslednjeg bita u *bit-string* notaciji. Ispravnost ove eliminacije je obezbeđena činjenicom da svaki BP zapis počinje otvorenom i završava se zatvorenom zagradom, odnosno svaki *bit-string* počinje bitom `1` a završava se bitom `0`.

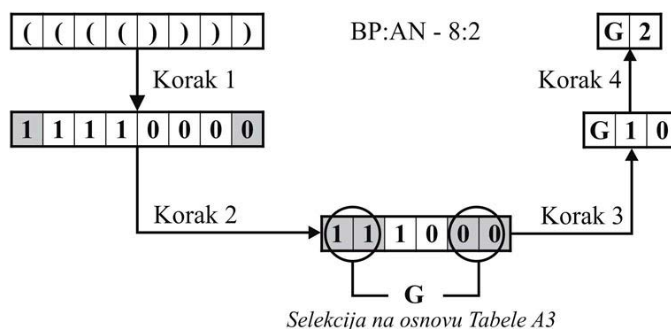
3. **Selekcija**: Ovaj korak koristi jedan od dva predložena pristupa (slučaja).

   1. Slučaj kada se biraju grupe od po 2 bita: prvi deo AN notacije (alfa notacija) se dobija tako što se uzimaju dve grupe od po dva bita (prva grupa na početku *bit-stringa* i druga grupa na kraju). Alfa notacija se dobija na osnovu predloženog šifrarnika (Tabela 4.4). Centrani deo *bit-stringa* (ako postoji) se prenosi u narednu fazu.

   2. Slučaj kada se biraju grupe od po 3 bita: Alfa notacija se dobija tako što se uzimaju dve grupe od po tri bita. Alfa notacija se dobija na osnovu predloženog šifrarnika (Tabela 4.7). Centrani deo *bit-stringa* (ako postoji) se prenosi u narednu fazu. Ovaj slučaj obezbeđuje još veće skraćenje BP notacije, jer se uzimaju veće grupe bitova i na taj način se dobija kraća AN notacija.

4. **Konverzija**: Drugi deo AN notacije (numerička notacija) se dobija prevođenjem centralnog dela *bit-stringa* u odgovarajući decimalni broj. Konačan oblik AN notacije se dobija spajanjem dva dela: alfa i numerički deo.

Primer 4.2.1. *Slika 4.9 predstavlja postupak transformacije BP zapisa* `(((())))` *na osnovu definisanih koraka Algoritma 4.2.1. U ovom primeru, koristimo Tabelu 4.4 kao šifrarnik za dobijanje alfa notacije, odnosno za dobijanje znaka "G". Centralni deo "10" se konvertuje u decimalni broj "2". Dobija se AN notacija "G2", gde je odnos BP:AN=8:2.*



Slika 4.9: Primer transformacije iz BP u AN notaciju

Tabela 4.4: Šifrarnik za prvi slučaj

Broj kombinacija	Prvi binarni par		Alfa notacija	Poslednji binarni par	
	1	2		m - 1	m
1	0	1	A	0	0
2	0	1	B	0	1
3	0	1	C	1	0
4	1	0	D	0	0
5	1	0	E	0	1
6	1	0	F	1	0
7	1	1	G	0	0
8	1	1	H	0	1
9	1	1	M	1	0

Primer 4.2.2. *Tabela 4.5 predstavlja više primera transformacije iz BP u AN notaciju za vrednosti* $n \in \{1, 2, 3, 4\}$*, gde važi da je broj kombinacija uparenih zagrada jednak Katalanovom broju* C_n*. Svaki od dobijenih AN zapisa odgovara tačno jednoj triangulaciji poligona. Za vrednosti* n*, a na osnovu odnosa broja triangulacija i Katalanovog broja (videti* (1.3)*), u ovom primeru imamo zapise koji odgovaraju poligonima sa temenima* $v \in \{3, 4, 5, 6\}$*.*

Tabela 4.5: Transformacija iz BP u AN, za slučaj gde se uzimaju po dva bita

n	Katalanov broj za n	BP	Binarni ekvivalent	Prvi skraćeni oblik	Drugi skraćeni oblik	AN
3	1	( )	1 0			0
4	1	( ) ( )	1 0 1 0	0 1		1
	2	(( ))	1 1 0 0	1 0		2
5	1	((( )))	1 1 1 0 0 0	1 1 0 0		G
	2	(( ) ( ))	1 1 0 1 0 0	1 0 1 0		F
	3	(( )) ( )	1 1 0 0 1 0	1 0 0 1		E
	4	( ) (( ))	1 0 1 1 0 0	0 1 1 0		C
	5	( ) ( ) ( )	1 0 1 0 1 0	0 1 0 1		B
6	1	(((( ))))	1 1 1 1 0 0 0 0	1 1 1 0 0 0	G 1 0	G 2
	2	((( ) ( )))	1 1 1 0 1 0 0 0	1 1 0 1 0 0	G 0 1	G 1
	3	(( ) (( )))	1 1 0 1 1 0 0 0	1 0 1 1 0 0	D 1 1	D 3
	4	((( )) ( ))	1 1 1 0 0 1 0 0	1 1 0 0 1 0	M 0 0	M 0
	5	(( ) ( ) ( ))	1 1 0 1 0 1 0 0	1 0 1 0 1 0	F 1 0	F 2
	6	( ) ((( )))	1 0 1 1 1 0 0 0	0 1 1 1 0 0	A 1 1	A 3
	7	( ) (( ) ( ))	1 0 1 1 0 1 0 0	0 1 1 0 1 0	C 1 0	C 2
	8	(( )) (( ))	1 1 0 0 1 1 0 0	1 0 0 1 1 0	F 0 1	F 1
	9	( ) ( ) (( ))	1 0 1 0 1 1 0 0	0 1 0 1 1 0	C 0 1	C 1
	10	((( ))) ( )	1 1 1 0 0 0 1 0	1 1 0 0 0 1	H 0 0	H 0
	11	(( ) ( )) ( )	1 1 0 1 0 0 1 0	1 0 1 0 0 1	E 1 0	E 2
	12	( ) (( )) ( )	1 0 1 1 0 0 1 0	0 1 1 0 0 1	B 1 0	B 2
	13	(( )) ( ) ( )	1 1 0 0 1 0 1 0	1 0 0 1 0 1	E 0 1	E 1
	14	( ) ( ) ( ) ( )	1 0 1 0 1 0 1 0	0 1 0 1 0 1	B 0 1	B 1

Primer 4.2.3. *Tabela 4.6 predstavlja više primera transformacije iz BP u AN za slučaj kada se u fazi selekcije biraju grupe od po tri bita. Prikazana je transformacija za neke zapise za $n \in \{6, 7, 8, 9\}$, gde se koristi Tabela 4.7 kao šifrarnik za dobijanje alfa znaka u AN.*

Tabela 4.6: Transformacija iz BP u AN, za slučaj gde se uzimaju po tri bita

n	Katalanov broj za *n*	BP (nekoliko primera)	Binarni ekvivalent	Prvi skraćeni oblik	Drugi skraćeni oblik	AN
6	132	(((((( ))))))	1 1 1 1 1 1 0 0 0 0 0 0	1 1 1 1 1 0 0 0 0 0	D 1 1 0 0	D 1 2
		((((( ) ( )))))	1 1 1 1 1 0 1 0 0 0 0 0	1 1 1 1 0 1 0 0 0 0	D 1 0 1 0	D 1 0
7	429	((((((( )))))))	1 1 1 1 1 1 1 0 0 0 0 0 0 0	1 1 1 1 1 1 0 0 0 0 0 0	D 1 1 1 0 0 0	D 5 6
		(((((( ) ( ))))))	1 1 1 1 1 1 0 1 0 0 0 0 0 0	1 1 1 1 1 0 1 0 0 0 0 0	D 1 1 0 1 0 0	D 5 2
		((((( ) (( ))))))	1 1 1 1 1 0 1 1 0 0 0 0 0 0	1 1 1 1 0 1 1 0 0 0 0 0	D 1 0 1 1 0 0	D 4 4
8	1430	(((((((( ))))))))	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 0 0 0 0 0 0 0	D 1 1 1 1 0 0 0 0	D 2 4 0
		((((((( ) ( )))))))	1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0	1 1 1 1 1 1 0 1 0 0 0 0 0 0	D 1 1 1 0 1 0 0 0	D 2 3 2
		(((((( ) (( )))))))	1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0	1 1 1 1 1 0 1 1 0 0 0 0 0 0	D 1 1 0 1 1 0 0 0	D 2 1 6
9	4862	((((((((( )))))))))	1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0	D 1 1 1 1 1 0 0 0 0 0	D 9 9 2
		(((((((( ) ( ))))))))	1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0	D 1 1 1 1 0 1 0 0 0 0	D 9 7 6
		((((((( ) (( ))))))))	1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0	1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0	D 1 1 1 0 1 1 0 0 0 0	D 9 4 4

Tabela 4.7: Šifrarnik za drugi slučaj

Prva grupa od tri bita			Alfa notacija	Poslednja grupa od tri bita			Prva grupa od tri bita			Alfa notacija	Poslednja grupa od tri bita		
1	2	3		*m* - 2	*m* - 1	*m*	1	2	3		*m* - 2	*m* - 1	*m*
0	1	0	A	0	0	0	0	1	0	Y	1	1	0
0	1	1	B	0	0	0	0	1	1	Z	1	1	0
1	0	1	C	0	0	0	1	0	1	a	1	1	0
1	1	1	D	0	0	0	1	1	1	b	1	1	0
1	1	0	E	0	0	0	1	1	0	c	1	1	0
1	0	0	F	0	0	0	1	0	0	d	1	1	0
0	1	0	G	0	1	1	0	1	0	e	1	0	0
0	1	1	H	0	1	1	0	1	1	f	1	0	0
1	0	1	I	0	1	1	1	0	1	g	1	0	0
1	1	1	J	0	1	1	1	1	1	h	1	0	0
1	1	0	K	0	1	1	1	1	0	i	1	0	0
1	0	0	L	0	1	1	1	0	0	j	1	0	0
0	1	0	M	1	0	1	0	1	0	k	0	1	0
0	1	1	N	1	0	1	0	1	1	l	0	1	0
1	0	1	O	1	0	1	1	0	1	m	0	1	0
1	1	1	P	1	0	1	1	1	1	n	0	1	0
1	1	0	Q	1	0	1	1	1	0	o	0	1	0
1	0	0	R	1	0	1	1	0	0	p	0	1	0
0	1	0	S	1	1	1	0	1	0	q	0	0	1
0	1	1	T	1	1	1	0	1	1	r	0	0	1
1	0	1	U	1	1	1	1	0	1	s	0	0	1
1	1	1	V	1	1	1	1	1	1	t	0	0	1
1	1	0	W	1	1	1	1	1	0	u	0	0	1
1	0	0	X	1	1	1	1	0	0	v	0	0	1

4.2.3 Obrnuta transformacija iz AN u BP zapis

Na osnovu Algoritma 4.2.1 definisaćemo obrnuti postupak, odnosno ustanovićemo faze koje realizuju transformaciju iz AN u prvobitnu BP notaciju. U obrnutoj transformaciji, faza *eliminacije* iz Algoritma 4.2.1 postaje faza *dodavanja.* Takođe, faze se izvode obrnutim redosledom. Dakle, transformacija se realizuje kroz sledeće faze: *konverzija, selekcija, dodavanje i zamena.* Ovaj obrnuti proces je realizovan Algoritmom 4.2.2.

Algoritam 4.2.2 Obrnuta transformacija iz AN u BP notaciju

Ulaz: AN notacija.

1: **Konverzija**
   Podela AN notacije na delove: *alpha* i *numeric.*
   numeric → binaryEq.
   Izlaz1 = *alpha* + binaryEq.

2: **Selekcija**
   if (Slučaj 1 - biraju se grupe od po 2 bita){
      alpha → Prva i Druga grupa od po dva bita. *// na osnovu Tabele 4.4*
      Izlaz2 = Prva grupa + binaryEq. + Druga grupa.
   }
   if (Slučaj 2 - biraju se grupe od po 3 bita){
      alpha → Prva i Druga grupa od po tri bita. *// na osnovu Tabele 4.7*
      Izlaz3 = Prva grupa + binaryEq.+ Druga grupa.
   }

3: **Dodavanje**
   Prvi element (1) i poslednji element (0) dodaj na Izlaz2 ILI Izlaz3.
   Izlaz4 = *bit-string.*

4: **Zamena**
   m je dužina *bit-stringa.*
   for $(i = 1; i \leq m; i++)$
      1→ (i 0 →)

Izlaz: BP notacija.

Algoritam 4.2.2 na ulazu očekuje AN notaciju koja se sastoji od dva dela: znaka α i decimalnog broja n. Označimo opšti oblik AN notacije sa $\alpha\, n$.

Kod transformacije iz AN→ BP notaciju, ako se bazira na prvom slučaju (na osnovu Tabele 4.4) znak α uzima vrednosti iz skupa $\{A, B, \ldots, M\}$. Kod drugog slučaja, gde se koristi Tabela 4.7, moguće vrednosti za α su definisane skupom $\{A, B, \ldots, Z, a, b, \ldots, v\}$.

Sledi opis Algoritma 4.2.2 po fazama:

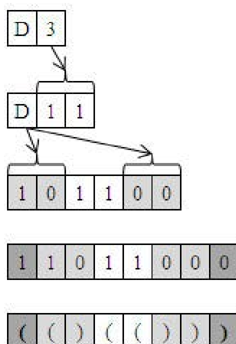
1. **Konverzija**: Decimalni broj n se konvertuje u binarni ekvivalent b, i on predstavlja centralni deo dobijene BP notacije.

2. **Selekcija**: Znak α se transformiše u odgovarajuće dve grupe od po dva bita (na osnovu Tabele 4.4) ili u odgovarajuće grupe od po tri bita (na osnovu Tabele 4.7). Označimo te parove sa p_1 i p_2. Prvi par je smešten na početku, a drugi na kraju trenutnog zapisa b (koji je dobijen iz faze konverzije). Na ovaj način, iz ove faze dobijamo notaciju u obliku $p_1\, b\, p_2$.

3. **Dodavanje**: U ovoj fazi se vrši dopuna zapisa koji je dobijen iz druge faze. Tačnije, vrši se dodavanje bita `1` na početku, a takođe i bita `0` na kraju tog zapisa. Dakle, iz ove faze se dobija notacija sledećeg oblika: $1\, p_1\, b\, p_2\, 0$. Na ovaj način dobijamo *bit-string* notaciju.

4. **Zamena**: Na kraju se vrši zamena bita `1` otvorenom zagradom `"("` i bita `0` zatvorenom zagradom `")"`. Iz ove faze se dobija izvorna BP notacija.

Primer 4.2.4. *Slika 4.10 predstavlja proces transformacije iz AN zapisa (u konkretnom primeru "D3") u BP zapis, gde se dobija niz uparenih zagrada* `(()(()))`. *U ovom primeru je korišćen prvi slučaj, gde se biraju grupe od po dva bita (na osnovu Tabele 4.4).*



Slika 4.10: Ilustracija obrnute transformacije (iz AN u BP notaciju)

4.2.4 Eksperimentalni rezultati

Ispitaćemo broj znakova kao meru efikasnosti predložene notacije. Na osnovu ulaza (BP) i izlaza (AN) Algoritma 4.2.2, može se uspostaviti sledeći odnos (R):

$$R = \frac{BP}{AN}, \tag{4.1}$$

gde je $BP = 2n$ i $AN = 2(n - i - 1)$. U ovom slučaju, n je indeks u izračunavanju Katalanovog broja (C_n), a i je broj bitova u grupi ($i \in \{2, 3\}$).

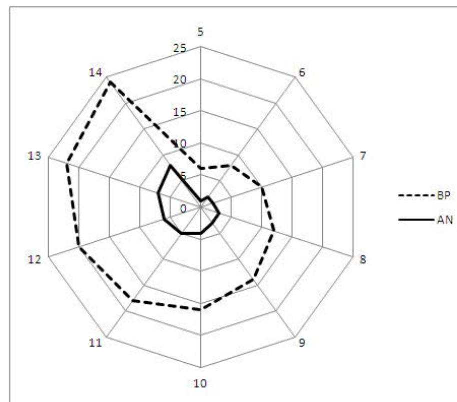
Za potrebe dobijanja ekperimentalnih rezultata, urađena je Java aplikacija sa mogućnošću skladištenja u obe notacije (`http://muzafers.uninp.edu.rs/triangulacija.html`).

Tabela 4.8 predstavlja odnos BP i AN notacija sa dva aspekta (na osnovu (4.1)): odnos broja znakova i odnos u veličini izlazne datoteke (u Kb) u kojoj se skladište triangulacije poligona sa v temena.

Tabela 4.8: Eksperimentalni rezultati za AN i BP notaciju

	Broj znakova			**Veličina datoteke**		
$\boldsymbol{v}$	**BP**	**AN**	**R**	**BP**	**AN**	**R**
5	6	1	6.00	0.5	0.3	1.67
6	8	2	4.00	1	0.6	1.67
7	10	2	5.00	2	1	2.00
8	12	3	4.00	3	1.4	2.14
9	14	3	4.67	7	3	2.33
10	16	4	4.00	26	9	2.89
11	18	5	3.60	95	31	3.06
12	20	6	3.33	386	122	3.16
13	22	7	3.14	1378	446	3.08
14	24	8	3.00	4792	987	4.85

Sledeći grafikon (Slika 4.11) predstavlja odnos broja znakova notacija AN i BP za $v \in \{5, 6, \ldots, 14\}$.



Slika 4.11: Odnos u broju znakova za AN i BP notaciju

Na osnovu rezultata eksperimentalnih ispitivanja uočene su prednosti primene AN notacije, a posebno je razlika evidentna za vrednosti $v > 10$ (Slika 4.11). Sa druge tačke gledišta, veličina izlazne datoteke naglo opada primenom AN notacije.

Na primer, za $v = 14$, datoteka koja sadrži BP notacije triangulacija je veličine *4792 kb* dok za odgovarajuće AN notacije iznosi *987 kb*, što je skoro pet puta manje. Za sve

navedene rezultate testiranja (Tabela 4.8, vrednosti $v \in \{5, \ldots, 14\}$) dobijamo prosečan koeficijent skraćenja (R) koji je približno jednak 2.69.

Pored toga što AN notacija može naći primenu u skladištenju triangulacija konveksnog poligona, u opštem slučaju, predstavljena notacija može da posluži i kao model za skraćenje zapisa i za neke druge probleme koji se zasnivaju na Katalanovim brojevima (*binarni zapisi za Lukasievićev algoritam, problem glasačkih listića, problem puteva u mreži* itd.).

4.2.5 Detalji implementacije u *Javi*

Predstavljeni metod, opisan Algoritmima 4.2.2 i 4.2.1, je implementiran u *Java NetBeans* okruženju. Implementacija ove metode je urađena kroz sledeće klase: `Triangulations, Nodes, LNodes`, `NoteTriangulations, StructureTriangulationBP, ProfileListBP`, `DrawTriangulation, ListBP`.

- Klasa `Triangulations` je zadužena za generisanje triangulacija na osnovu zapisa koji proizvodi klasa `NoteTriangulations` (Prilog II-C).

- Klasa `DrawTriangulations` u saradnji sa klasom `Triangulations` je zadužena za grafički prikaz triangulacija na *JPanel* aplikacije.

- Klase `StructureTriangulationBP` i `ProfileListBP` su apstraktne i zadužene su za uspostavljanje veze sa generisanim triangulacijama i njihovim notacijama. Klase `Nodes` i `LNodes` su standardne klase za rad sa temenima poligona. Klasa `ListBP` određuje redosled iscrtavanja triangulacija na osnovu liste znakova u zapisu.

- Glavna klasa `NoteTriangulations` sadrži metodu `ANnotation()` koja realizuje Algoritam 4.2.2.

U nastavku je dat deo izvornog koda metode `ANnotation()` podeljen po fazama:

Faza 1: Zamena znakova iz BP notacije u binarni ekvivalent

```
BP1 = BP.replace("(","1");
BP1 = BP.replace(")","0");
binaryEq = BP1;
```

Faza 2: Eliminisanje prvog i poslednjeg bita

```
BP2 = binaryEq.substring(1,LabelBP.length()-1);
```

Faza 3: Selektovanje prve i poslednje grupe bitova. Prikazan je deo izvornog koda za slučaj kada se biraju grupe od po dva bita (na osnovu Tabele 4.4).

```
int aLenght = BP2.length();
String aFirst = BP2.substring(0,2);
String aLast = BP2.substring(aLenght-2,aLenght);
String str0="00", str1="01", str2="10", str3="11";

// codebook  table A3 (From A to M)
if (aFirst.equals(str1) && aLast.equals(str0)) Alfa="A";
```

```
if (aFirst.equals(str2) && aLast.equals(str0)) Alfa="B";
if (aFirst.equals(str1) && aLast.equals(str2)) Alfa="C";
...
if (aFirst.equals(str3) && aLast.equals(str0)) Alfa="M";
```

Faza 4: Konvertovanje centralnog dela u odgovarajući decimalni broj.

```
CentralBin = BP2.substring(2,BP2.length()-2);
long numSk = Long.parseLong(CentralBin);
long remSk;
while(numSk > 0){
remSk = numSk % 10;
numSk = numSk / 10; }
int CentDec= Integer.parseInt(CentralBin,2);
CentralDec = Integer.toString(CentDec);
```

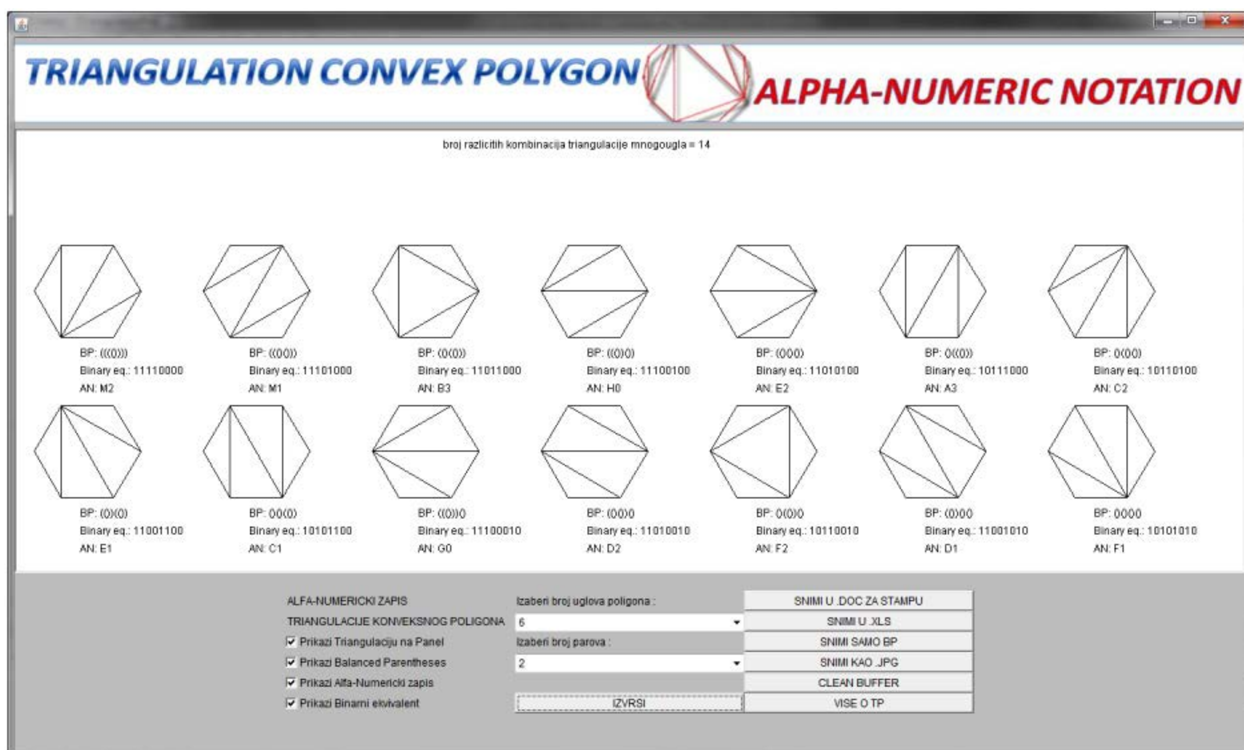
Prikazani rezultati u uporednoj analizi su omogućeni primenom metode za skladištenje oba tipa zapisa triangulacija u *Java* aplikaciji. U nastavku sledi deo *Java* kôda koji omogućava uporedni pregled izlaznih datoteka za obe notacije, a sve u cilju utvrđivanja procenta uštede memorijskog prostora.

Java primer 4.2.1. *Deo glavne klase* `NoteTriangulations`*, koji je zadužen za snimanje navedenih notacija u dve izlazne datoteke, je dat u nastavku:*

```
for (i=0; i<n; i++){
try {
// izlazni fajl za AN notaciju
bufferedWriter=new BufferedWriter(new FileWriter("notationAN.jdb",true));
bufferedWriter.write(LabelAlfa+CentralDec); bufferedWriter.newLine();

// izlazni fajl za BP notaciju
bufferedWriter1=new BufferedWriter(new FileWriter("notationBP.jdb",true));
bufferedWriter1.write(LabelBP); bufferedWriter1.newLine(); }
catch (FileNotFoundException ex) { ex.printStackTrace(); }
catch (IOException ex) { ex.printStackTrace();
}
```

Java aplikacija sadrži centralni *JPanel* i *Toolbar* sa dodatnim opcijama. *JPanel* prikazuje grafičku prezentaciju triangulacija i njihov odgovarajući zapis. *Toolbar* aplikacije sadrži opcije za čuvanje dobijenih rezultata u različitim izlaznim formatima (*mdb, xls, doc, jpg*). Pored biranja broja temena za koji želimo da prikažemo notaciju i konstrukciju triangulacija, postoji i opcija biranja slučaja za skraćenje: grupa po dva ili po tri bita.

Glavna klasa `NoteTriangulations` omogućava da se odgovarajućoj triangulaciji pridruži više zapisa (npr. *BP notacija, AN notacija i bit-string*, videti Sliku 4.12).



Slika 4.12: AN i BP notacije triangulacija šestougla

Poglavlje 5

Primena objektno-orijentisane analize i dizajna na problem triangulacije poligona

Triangulacija poligona je kompleksan problem koji zahteva složene klase za efikasnu objektno-orijentisanu implementaciju. Objektno-orijentisana analiza i dizajn (*OOAD*) omogućava sveobuhvatan uvid u implementaciju konkretnog problema [43].

Ovo poglavlje sadrži neke pristupe OOAD metodologije za problem triangulacije poligona. Ovaj problem je analiziran sa tri aspekta, gde je svaki pristup definisan odgovarajućim tipom inženjeringa ove metodologije:

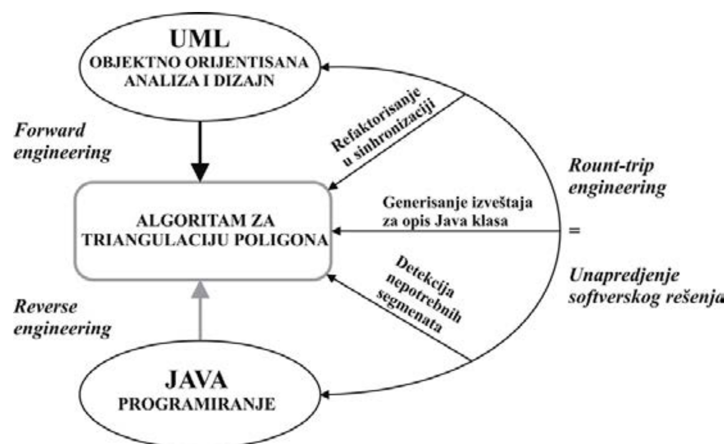
1. Direktna analiza (direktni inženjering, *eng. forward engineering*) se zasniva na modeliranju kroz statički, dinamički i fizički model (pogled). Ovaj pristup omogućava generisanje izvornog koda u nekom od objektno-orijentisanih programskih jezika na osnovu razvijenih modela. U konkretnom slučaju je primenjeno *UML* modeliranje i prikazano je generisanje u *Java* programskom jeziku.

2. Povratna analiza (obrnuti ili reverzni inženjering, *eng. reverse engineering*) se odnosi na tumačenje i analizu izvornog koda koji se generiše iz definisanih modela. Tačnije, ovaj pristup obezbeđuje vizuelizaciju izvornog koda.

3. Sinhronizacija direktne i povratne analize (*round-trip* inženjering) se odnosi na integraciju i koordinaciju između razvijenih modela i dobijenog programskog koda.

Implementacija data u drugom poglavlju (realizacija Algoritma 1.3.1) predstavlja fazu programiranja bez prethodne analize i kreiranja vizuelnog plana. Način implementacije nekog problema bez analize i dizajna (bez pratećih modela koji predstavljaju vizuelizaciju problema) može negativno da utiče na funkcionalnost i efikasnost dobijenog softverskog rešenja.

Bolje razumevanje i detaljna analiza algoritma se postiže primenom direktnog pristupa koji zahteva kreiranje odgovarajućih pogleda (modela) na sam problem. Pored toga, na ovaj način dobijamo razvijenu strategiju planiranja implementacije problema koja je nezavisna od tehnologije i alata za implementaciju. Ovakav pristup se bavi postupkom definisanja modela koji omogućavaju prelazak u fazu kodiranja (programiranja). Nakon završetka faze programiranja, naredna dva pristupa (obrnuti i *round-trip* inženjering) igraju ključnu ulogu u održavanju i evoluciji dobijenog softverskog rešenja.



Slika 5.1: Tri OOAD pristupa (inženjeringa)

Bitno poboljšanje se postiže primenom obrnutog inženjeringa i sinhronizacije *UML* modela i *Java* izvornog koda. U poslednjoj sekciji ovog poglavlja date su neke prednosti za sva tri pristupa, sa posebnim akcentom na unapređenje implementacija primenom sinhronizacije prva dva pristupa.

5.1 Direktna analiza za algoritam triangulacije poligona (*Forward engineering*)

UML je jezik kojim se kreira apstraktni model sistema kroz skup grafičkih dijagrama. Koristi se za specifikaciju, vizuelizaciju, projektovanje i dokumentaciju razvoja sistema. Opšta klasifikacija standardnih *UML* pogleda (modela) je izvršena na statički, dinamički i fizički model. Modeliranje u *UML*-u ima širok spektar različitih područja primene [33, 66, 70].

U ovom slučaju, putem *UML* modeliranja pratićemo faze implementacije za date nove metode za triangulaciju konveksnih poligona. Taj postupak praćenja počinje od apstraktnih ideja, kroz pojedinačne klase, aktivnosti, stanja i ponašanja sistema, pa sve do fizičke distribucije. Najpre ćemo navesti proces *UML* modeliranja, zatim ćemo dati primere generisanja u *Java* izvorni kôd i na kraju daćemo predloge poboljšanja softverskog rešenja kroz sinhronizaciju dva dela implementacije: programiranja u *Javi* i modeliranja u *UML*-u.

5.1.1 Modeliranje u *UML-u*: statični, dinamički i fizički model

Deo implementacije koji se odnosi na modeliranje (metode koje su predstavljene u disertaciji) sadrži ukupno 48 dijagrama, od toga 36 dijagrama se odnosi na dinamički model. Tabela 5.1 predstavlja broj tih dijagrama, klasifikovanih po modelu i vrsti. Svi navedeni dijagrami su dobijeni iz razvojnih okruženja (*Visual Paradigm for UML* i *NetBeansUML*) i mogu se preuzeti sa : `http://muzafers.uninp.edu.rs/triangulation/UMLTriang.rar`

Tabela 5.1: Pregled *UML* dijagrama po modelima

Model	*Vrsta UML dijagrama*	*Broj dijagrama*
Statički	Class diagrams	2
	Object diagrams	8
Dinamički	Use case diagrams	8
	Activity diagrams	7
	State-Chart diagrams	7
	Sequence diagrams	6
	Collaboration diagrams	4
	Interaction overview diagram	4
Fizički	Deployment diagram	1
	Component diagram	1

Dijagram klasa opisuje statičku strukturu sistema. Klase se modeluju i međusobno povezuju korišćenjem ovih dijagrama, dok su objekti opisani svojim osobinama i vezama sa ostalim objektima. Svaki od objekata sadrži niz operacija koje može izvršiti, čime se modeluje njegovo ponašanje [3].

1. Opšta struktura metoda za generisanje triangulacija (metode koje su date u trećem poglavlju: *metod dekompozije* i *blok metoda* kao i *Hurtado-Noy* algoritam) je predstavljena *UML klasnim dijagramom* na Slici 5.2. Zajedničke klase su: *GenerateTriangulations, Triangulation, Node, LeafNode, Point* i *PostScriptWriter*.

   Glavna klasa *GenerateTriangulations* sadrži *Java* metodu za generisanje triangulacija:

   - ako govorimo o *blok metodi*, onda je definisana *Java* metoda *Block()* koja realizuje Algoritam 3.2.3,

   - ako govorimo o *metodi dekompozicije* onda je to *Java* metoda *CreateExpr()* koja realizuje Algoritam 3.1.1,

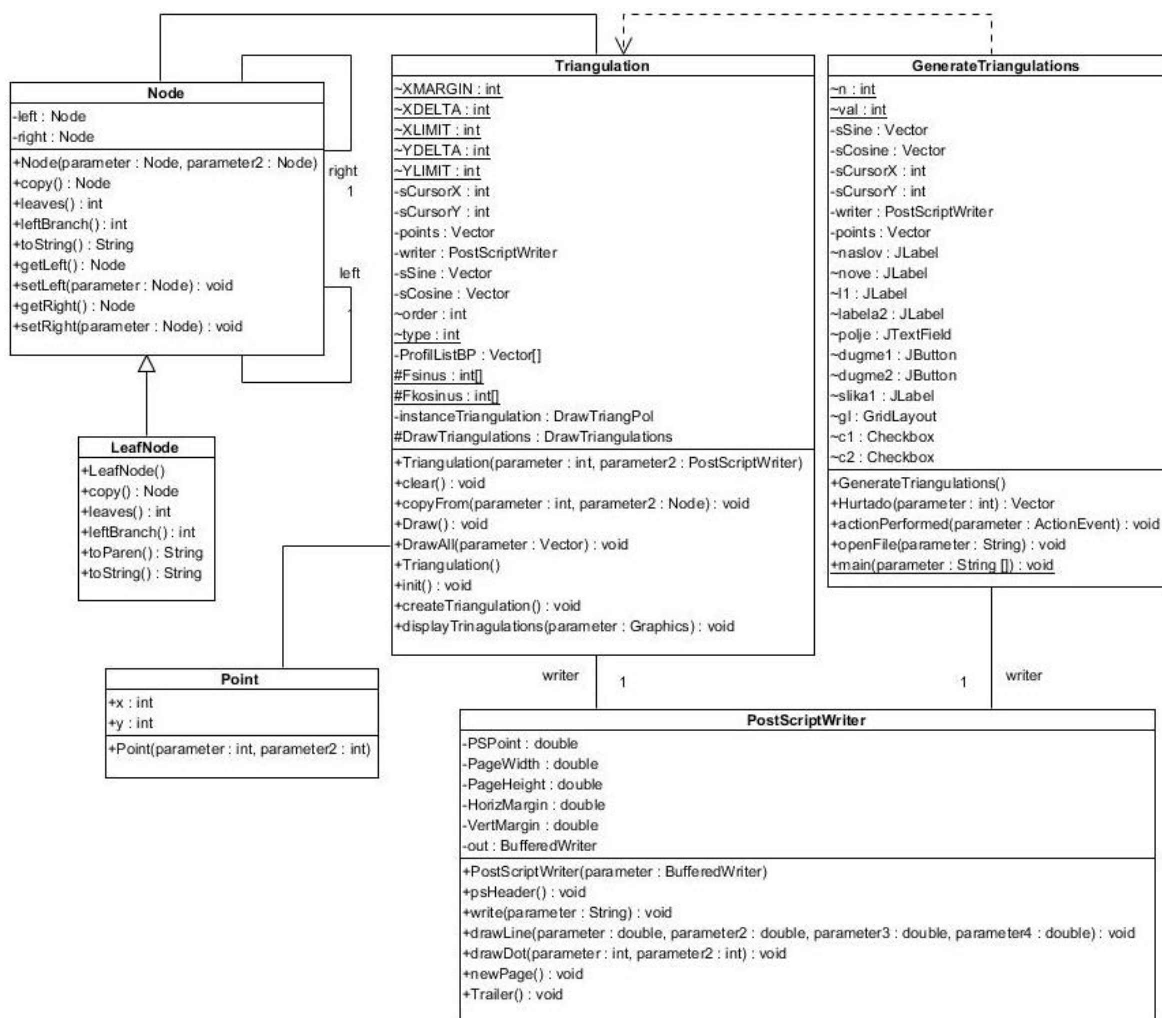
   - i ako govorimo o implementaciji *Hurtado-Noy* algoritma onda je to *Java* metoda *Hurtado()* koja realizuje Algoritam 1.3.1.

Tri tipa relacija (veza) je definisano u *Class Diagram*-u, a to su: zavisnost, generalizacija i asocijacija.

(i) Zavisnost se najčešće koristi kada jedna klasa koristi drugu kao argument. Primer veze zavisnosti je odnos između klase *Triangulation* i glavne klase *GenerateTriangulations* (Slika 5.2).

(ii) Asocijacija je veza koja pokazuje da su objekti jedne klase povezani sa objektima druge (na primer *Triangulation* sa klasom *Node*, ili *Triangulation* sa klasom *Point*).

(iii) Generalizacija definiše hijerarhiju u kojoj podklasa nasleđuje jednu ili više nadklasa (na primer, takav odnos je prikazan između klasa *Node* i *LeafNode*).



Slika 5.2: Dijagram klasa (metode za generisanje triangulacija)

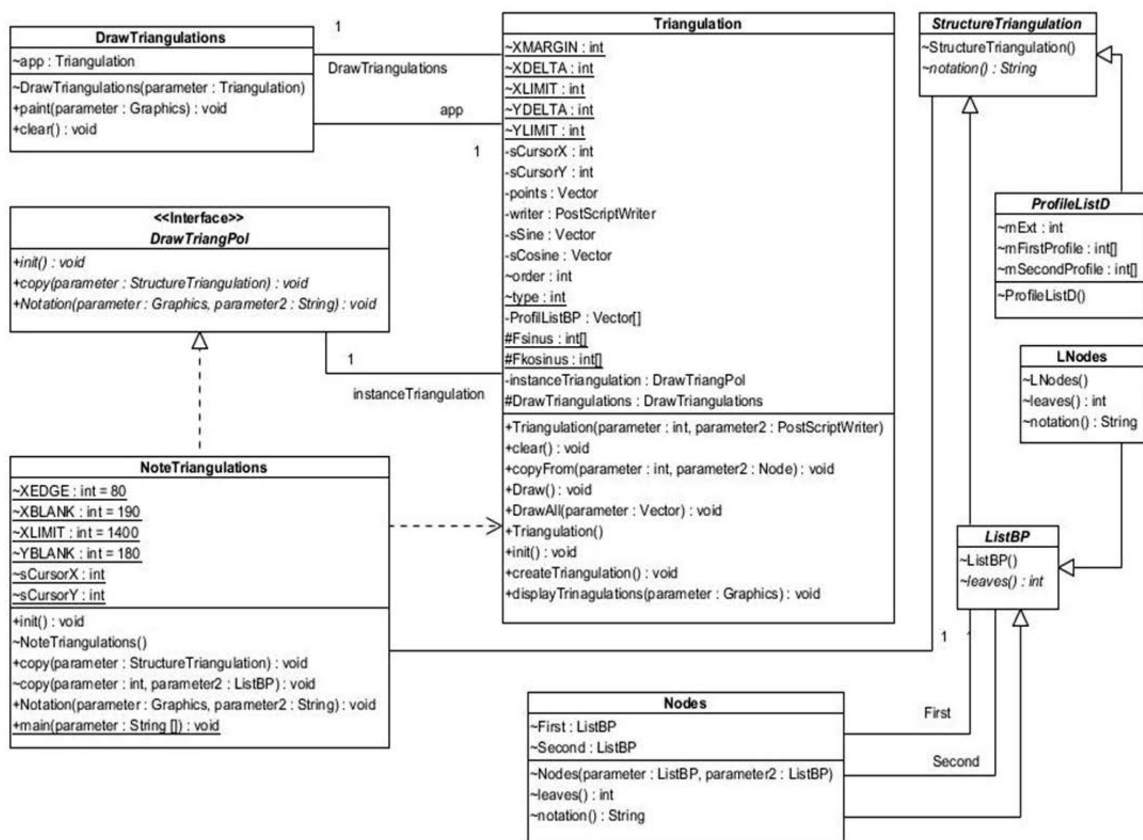
2. Opšta struktura metoda za notaciju i skladištenje triangulacija (metode koje su date u četvrtom poglavlju: *metod za alfanumeriču notaciju i ballot notacija*) je predstavljena *UML klasnim dijagramom* na Slici 5.3.

   Glavna klasa *NoteTriangulations* sadrži izvršnu metodu kao i metodu za pridruživanje odgovarajuće notacije triangulacijama, i to:

   - ako govorimo o *alfanumeričkoj notaciji* onda je definisana *Java* metoda *ANnotation()* koja realizuje Algoritam 4.1.1,

   - ako govorimo o *ballot notaciji* onda je definisana *Java* metoda *BallotNotation()* koja realizuje Algoritam 4.2.2.

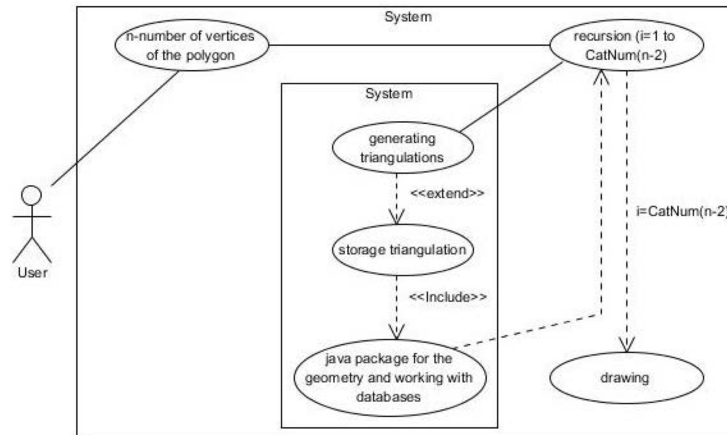
   Kao i u prethodnom dijagramu klasa 5.2, i u dijagramu 5.3 su definisana tri tipa relacija: zavisnost, generalizacija i asocijacija.



Slika 5.3: Dijagram klasa (metode za notaciju triangulacija)

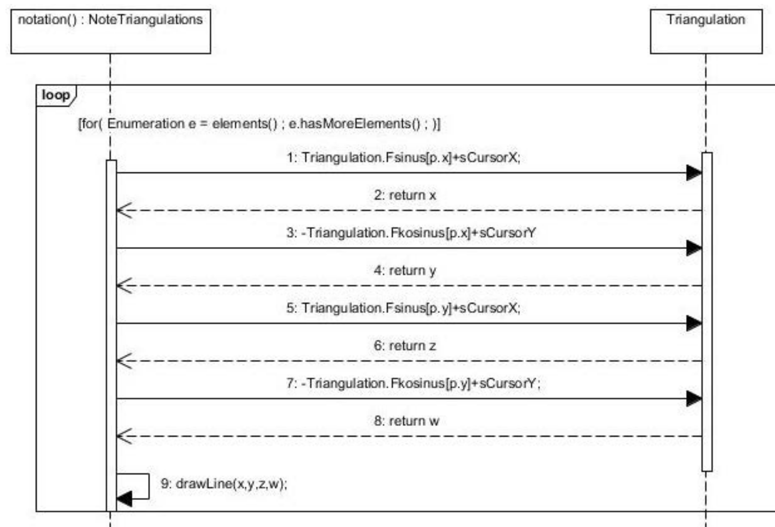
Operacije (*Java* metode) iz navedenih dijagrama klasa su dodatno opisane dijagramima ponašanja i dijagramima interakcije formirajući dinamički model sistema. Dijagrami ponašanja uključuju aktivnosti i stanja, dok dijagrami interakcije obuhvataju sekvence i komunikacije (saradnju) [1, 37].

Dijagrami slučaja korišćenja (*Use Case*) predstavljaju funkcionalne zahteve koje treba sistem da ispuni (Slika 5.4). U implementaciji problema triangulacije poligona možemo izvršiti sledeću klasifikaciju slučajeva korišćenja: (*i*) aktivnosti (operacije) koje generišu triangulacije, (*ii*) aktivnosti za pridruživanje odgovarajuće notacije triangulaciji, (*iii*) aktivnosti za trajno skladištenje triangulacija.



Slika 5.4: Dijagram slučaja korišćenja

Bitno je i odgovarajućim dijagramom predstaviti i komunikaciju (saradnju) i tok podataka između dve metode ili klase. Navedeni dijagram sekvenci (Slika 5.5) prikazuje samo jednu interakciju između klase *Triangulation* i glavne klase *NoteTriangulations*, sa akcentom na redosled i tok slanja poruka i jasan prikaz saradnje između njih.

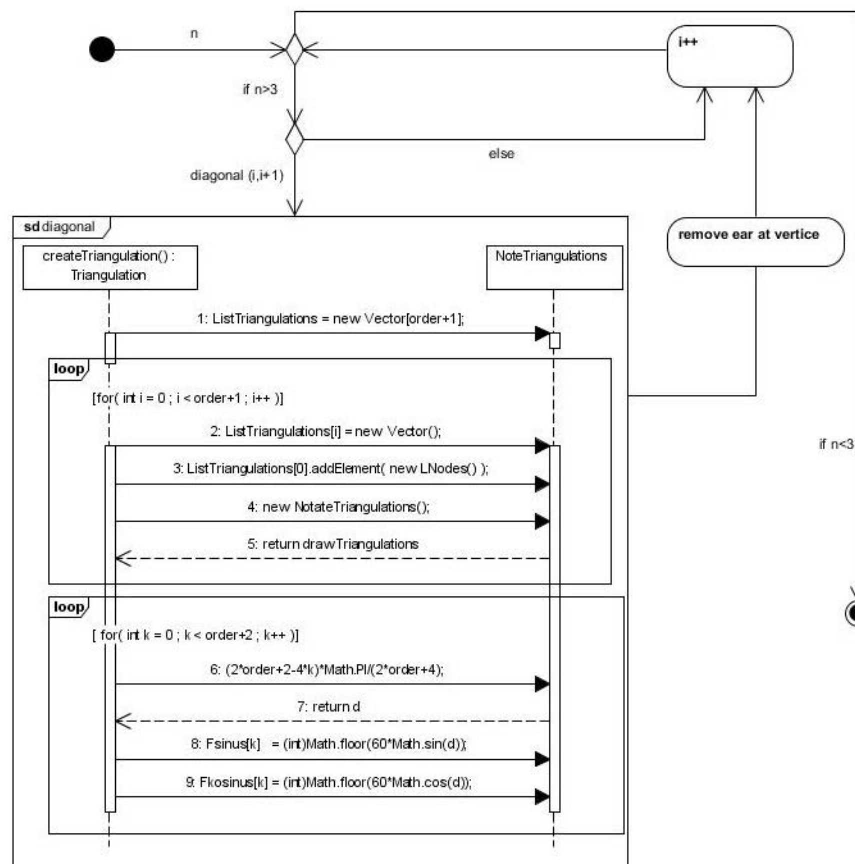


Slika 5.5: Dijagram sekvenci: Klase *Triangulation* i *NoteTriangulations*

Dijagrami stanja (*State-Chart*) se koriste za apstraktni opis ponašanja algoritama. Jedan od primera prelaska iz jednog stanja u drugo je proces generisanja triangulacija primenom

odgovarajuće metode (npr. *metode dekompozicije* ili *blok metode*) i prelazak u stanje njihovog zapisivanja primenom odgovarajuće metode za notaciju (npr. *AN* ili *ballot notacija*). Svaka od ovih tranzicija stanja ima tri opciona dela: (*i*) pobuđen događaj: pokretanje metode za generisanje triangulacija; (*ii*) zaštitni uslov: izgenerisan tačan broj triangulacija koji je jednak C_{n-2}; (*iii*) aktivnost: iscrtavanje triangulacija na osnovu pridruženih notacija.

Dijagram 5.6 predstavlja interakciju klase *Triangulation* sa klasom *NoteTriangulations*. Ovaj dijagram prikazuje postupak pridruživanja odgovarajuće notacije u trenutku kreiranja triangulacija.

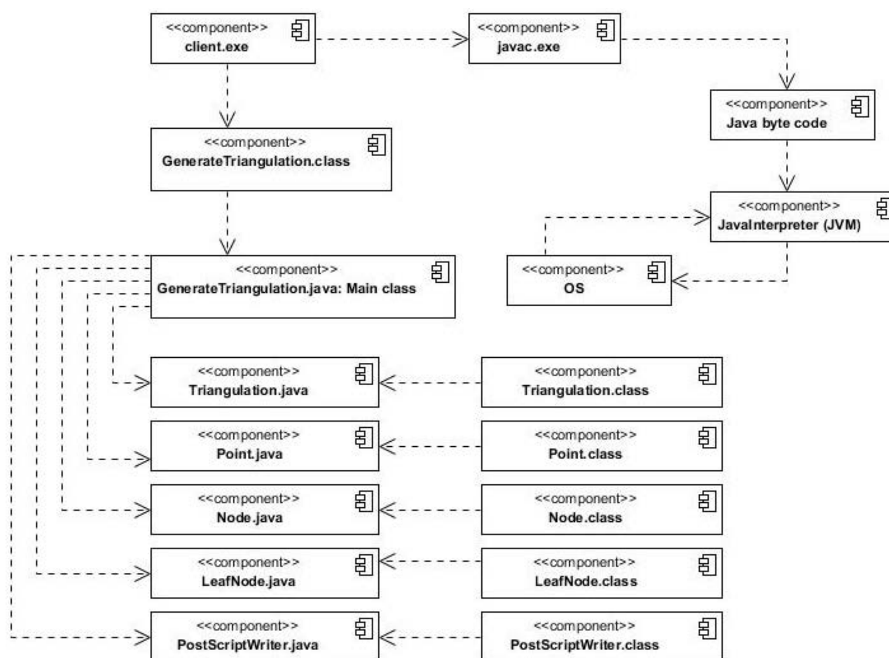


Slika 5.6: Dijagram interakcije

Dijagramima aktivnosti se vizuelizuju sledeće metode: *CreateExpr, Block, Hurtado, BallotNotation, ANnotation, DisplayTriangulations, DrawAll.* Dijagram saradnje (komunikacije) se odnosi na prikazivanje interakcije između objekata klasa. Takođe, razvijeni su dijagrami saradnje za objekte iz dijagrama klasa koji su prikazani na slikama 5.2 i 5.3.

Fizički model se realizuje kroz dijagrame komponenti i dijagrame razvoja. Dijagram komponenti prikazuje strukturu sistema i opisuje zavisnost komponenti sistema (Slika 5.7). Komponente prikazanog dijagrama su izvorni kodovi, biblioteke, dinamičke komponente i izvršni programi (*.exe*). Datoteke sa izvornim kodom programa (segmenti implementacije metoda za triagulaciju poligona, sa *java* ekstenzijom) se kompajliraju u *class* binarne da-

toteke. Dakle, pri izvršavanju svaki *java* datoteka kreira prateću datoteku sa *class* ekstenzijom. *Java* kompajler konvertuje *Java* izvorni kôd u bajt programski kôd. Bajt programski kôd se interpretira pomoću *JVM-a* jer on sadrži naredbe koje prepoznaje *JVM* i koje preslikava u naredbe konkretnog OS.

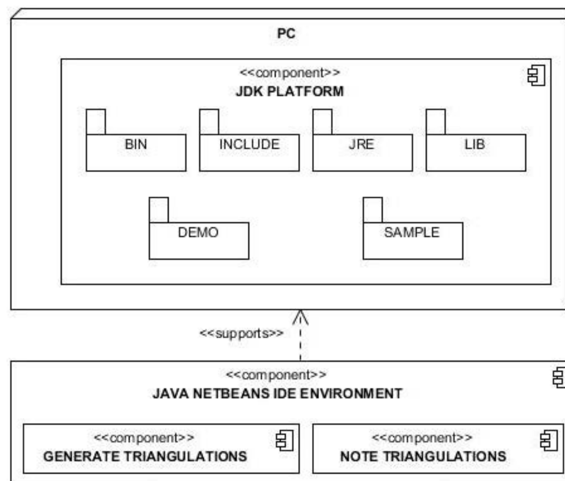


Slika 5.7: Dijagram komponenti

Dijagrami realizacije ili razvoja (*deployment diagram*) prikazuju komunikaciju između hardverskih i softverskih komponenata. Između ove dve komponente postoji veza zavisnosti (<< *supports* >>). Na softverskoj strani su *Java* aplikacije za rad sa metodama za triangulaciju poligona. Na hardverskoj strani je *Java* JDK platforma (*Java Development Kit*) sa svojim komponentama koje pružaju podršku implementacijama koje realizuju metode (Slika 5.8).

Prilikom preuzimanja izvršnih *Java* aplikacija, za sve metode koje su opisane u ovoj disertaciji (ukupno pet *Java* aplikacija), potrebno je imati *Java* platformu kako bi aplikacije bile funkcionalne. Komplet JDK postoji za čitav niz hardverskih platformi i operativnih sistema, a paket i njegova dokumentacija su potpuno nezavisni i instaliraju se posebno. Direktorijum *BIN* sadrži celokupan rezultat prevođenja aplikacije. Direktorijum *LIB* sadrži sve eksterne biblioteke koje aplikacija koristi (opisano u drugom poglavlju disertacije) a JRE služi samo za pokretanje *Java* aplikacija koje realizuju pomenute metode.

Proces kreiranja aplikacije ne obuhvata samo prevođenje, već on uključuje i druge korake (prevođenje i izvršavanje test klasa, kreiranje *javadoc*-a, izračunavanje pokrivenosti kôda, generisanje izlaznih datoteka koje sadrže grafički prikaz triangulacija i kreiranje datoteka za zapise triangulacija u obliku neke određene notacije).



Slika 5.8: Dijagram realizacije

5.1.2 Generisanje *Java* izvornog koda iz *UML* modela

Ideja o generisanju izvornog kôda se uvek povezivala sa alatima i tehnikama koje su zasnovane na modeliranju u *UML*-u. Za potrebe generisanja izvornog koda u *Javi*, korišćena su okruženja: *Visual Paradigm for UML* i *NetBeansUML* [74, 52].

Proces generisanja izvornog koda na osnovu kreiranog modela dovodi do opšte strukture softverskog rešenja (klase, metode, atributi). Na ovaj način, omogućeno je da brzo i efikasano izvršimo transformciju modela u prilagodljiv *Java* izvorni kôd. Primenom pristupa direktnog razvoja na dijagrame klasa (Slika 5.2 i 5.3) dobijamo opšte strukture svih klasa (tj. deklaracije promenljivih, zaglavlja njihovih metoda itd.). Dobijeni kôd je potrebno učiniti funkcionalnijim u tom smislu što će morati da se dopuni detaljima koji se baziraju na dinamičkom modelu, tačnije na dijagramima ponašanja i dijagramima interakcija.

Primer 5.1.1. *Sada ćemo navesti jedan primer generisanja jednog segmenta Java izvornog koda za klase* `Triangulations` *i* `GenerateTriangulations` *iz Dijagrama klasa (Slika 5.2). Znak "*" označava da postoji mogućnost dopune izvornog koda u cilju postizanja potrebne i/ili željene funkcionalnosti.*

```
public class Triangulation {

    //attributes
    public Object private int sCursorX;
    public Object private int sCursorY;
    public Object private Vector<Point> points;
    public Object private PostScriptWriter writer;

    ***
```

```
    //operations
    public void Draw() {*}
    public void DrawAll(Vector<Node> trees)() {*}
    public void clear() {*}
    public void copyFrom(Object int aOffset, Object Node t) {*}
}

public class GenerateTriangulation implements Triangulation {

    //operations
    public void Vector<Node> createTriangulations(Object int limit) {*}
    public void static void main(Object String args[]) {*}
}
```

Kompletna struktura izvornog koda (generisana iz *UML* modela) može se preuzeti sa: `http://muzafers.uninp.edu.rs/triangulation/GeneralStructureHurtado.rar`

5.2 Povratna analiza i sinhronizacija (*Reverse/Round-trip engineering*)

U ovoj sekciji, navešćemo analizu sledeća dva pristupa na konkretnom primeru implementacija za generisanje i notaciju triangulacija.

1. Obrnuti inženjering i vizuelizacija izvornog koda omogućavaju bolje razumevanje programa. Glavne prednosti ovog pristupa su: analiza uticaja uvođenja odgovarajućih promena u implementaciji, upoznavanje nepoznatog koda, ponovno korišćenje, integracija određenih modula izvornog koda, održavanje softvera i sl. Ovaj pristup nalazi mnoge primene u identifikovanju objektno-orijentisanog izvornog koda [5, 72].

   Na primeru implementacija za triangulaciju poligona, primenjen je ovaj postupak i to kroz dve faze: (*i*) Faza raščlanjavanja kompletnog izvornog koda u formalne jedinice (klase), gde se najpre dobija statički model (*dijagrami slučajeva korišćenja* i *dijagrami klasa*); (*ii*) Na osnovu modelovanih atributa i operacija, njihovi se opisi dalje razlažu na dijagrame koji se odnose na dinamički model (*aktivnosti, stanja, interakcije, sekvence*).

2. *Round-trip* inženjering predstavlja sinhronizaciju direktne i povratne analize, što se pokazalo kao najbolja praksa u testiranju i održavanju implementacija [6, 42]. Prednosti ovakvog pristupa se ogledaju u direktnoj koordinaciji između izvornog koda i odgovarajućih modela. Takođe, i ovde se mogu istaći bitne napredne mogućnosti. Pre svega, može se ostvariti manipulisanje nad definisanim modelima ili postići odgovarajuće promene u izvornom kodu koje su prouzrokovane modifikacijom odgovarajućih modela. Na ovaj način se ostvaruje analiza uticaja uvođenja odgovarajućih promena u implementaciji.

Primer 5.2.1. *Dat je jedan primer sinhronizovanja UML modela sa Java klasama* `GenerateTriangulations` *i* `Triangulations`*. NetBeansUML modul u postupku ovakve integracije prijavljuje na standardni izlaz sledeće podatke:*

```
   "...Initial reverse engineering into a new project:
Begin processing Reverse Engineering, Parsing 56 elements,
Analyzing attributes and operations for 72 symbols,
Resolving 54 attribute types,
Integrating 72 elements, Building the query cache...".
```

Ovi izlazni podaci pokazuju: da je 72 elementa modela korišćeno za generisanje datoteka sa Java izvornim kodom (sa tačnim brojem elemenata i atributa koji su identifikovani i klasifikovani). Tabela 5.2 predstavlja ispunjenost određenih uslova u ovom procesu sinhronizacije.

Tabela 5.2: Zahtevi u *round-trip* inženjeringu za konkretne klase

	Zahtevi	*Triangulation*	*GenerateTriangulations*
1	Navigacija na izvorni kôd	+	+
2	Generisanje modela zavisnosti	+	+
3	Generisanje izvornog koda	+	+
4	Generisanje izveštaja	+	+
5	Navigacija na elemente	+	+
6	Refaktorisanje	+	+
7	Pronalaženje i zamene u UML modelu	+	
8	Primena DP* i model čitljivosti koda	+	
9	Manipulacija sa A, O, I, R*	+	+

** A - Atributi, O - Operacije, I - Implementacije, R - Relacije/Veze, DP - Dizajn šabloni.*

U *NetBeansUML* modulu postoji mogućnost automatskog identifikovanja izvornog koda samo ako je ispravno uspostavljena sinhronizacija između *UML* i *Java NetBeans* projekta (Zahtevi 1,2,3 i 5 iz Tabele 5.2). Na taj način se smanjuje složenost problema triangulacije, a programiranje i dodavanje novih funkcionalnosti je veoma olakšano.

Za obrnuti i *round-trip* inženjering je važno pomenuti mogućnost generisanja izveštaja modela za sve klase. Izveštaj opisuje sve klase koje su definisane u implementaciji (kroz analizu njihovih atributa, metoda, relacija sa drugim klasama i sl.). Pored analize klasa, izveštaj sadrži i korišćenje paketa i interfejsa u implementaciji (Zahtev 4). Podaci o preuzimanju ovih izveštaja su dati na kraju disertacije (Prilog I).

Glavne kategorije obrnutog inženjeringa su automatsko restrukturiranje i automatska transformacija: (i) Prva kategorija se odnosi na refaktorisanje i remodularizaciju koji se primenjuje u cilju dobijanja boljeg izvornog koda: bolja struktura, modularnost, izbegavanje

ponovljenih slučajeva, izvlačenje metoda itd. (Zahtev 6 i 7); (*ii*) Druga kategorija se odnosi na primenu standarda u kodiranju (programiranju) koji se primenjuje u cilju dobijanja čitljivijeg izvornog koda (Zahtev 8).

Refaktorisanje menja unutrašnju strukturu kako bi se lakše razumela i jednostavnije modifikovala implementacija kroz manipulaciju atributima, operacijama i relacijama između njih (Zahtev 9).

5.2.1 Dobijeni rezultati u unapređenju implementacija

Tabela 5.3 predstavlja rezultate unapređenja softverskog rešenja za metode koje se koriste za generisanje triangulacija. Vršena su ispitivanja i analize izvornog koda primenom sinhronizacije *Java* programiranja i *UML* modeliranja, a sve sa ciljem da se dobije jednostavniji izvorni kôd (bez ponovljenih slučajeva, sa što većom modularizacijom, sa integracijom određenih delova itd.). Zatim, cilj je bio i da se smanji obimnost implementacije i da se utiče na funkcionalnost i rad segmenata implementacije posebno.

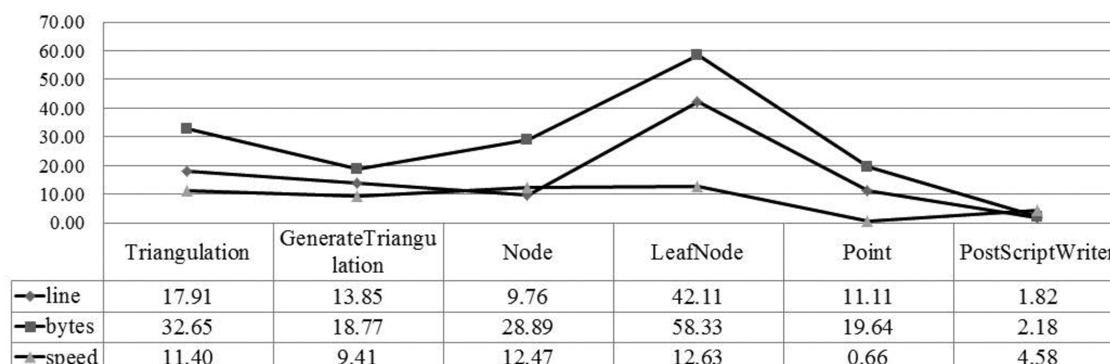
U analizi odnosa "pre i posle" koriste se tri kriterijuma: broj linija izvornog koda (*line*), veličina *Java* datoteke (*bytes*) i merenje vremena u sekundama (*speed*). Ova analiza je rađena pojedinačno za svaki segment implementacije (za svaku klasu). Podaci dobijeni nakon poboljšanja su označeni sa '*'.

Tabela 5.3: Poboljšanje implementacija za generisanje triangulacija

Klasa	*line*	*line**	*bytes*	*bytes**	*speed*	*speed**
Triangulation	79	67	2275	1715	29.12	26.14
GenerateTriangulation	222	195	8544	7194	37.56	34.33
Node	45	41	745	578	4.51	4.01
LeafNode	27	19	342	216	3.21	2.85
Point	10	9	134	112	1.52	1.51
PostScriptWriter	56	55	1830	1791	2.74	2.62
Ukupno	**439**	**386**	**13870**	**11606**	**78.66**	**71.46**
Poboljšanje (%)	**13.73**		**19.51**		**10.08**	

U procesu analize koda implementacija (*code review*), primenom modula *NetBeans- "Unnecessary Code Detector"*, ostvareno je sledeće poboljšanje: *nekorišćen import/uvoz* (4.09%), *nepročitane lokalne varijable* (3.02%), *nepročitani parametri* (3.18%), *nepotrebna metoda ili konstruktor* (4.23%), *nepročitane metode (private), konstruktori ili tipovi* (5.87%) *i nepročitani lokalni ili privatni članovi* (6.57%).

Sledeći grafikon (Slika 5.9) prikazuje poboljšanje izvornog koda za svaki segment implementacije, za tri postavljena kriterijuma (*broj linija koda; veličina datoteke; brzina*).



	Triangulation	GenerateTriangulation	Node	LeafNode	Point	PostScriptWriter
line	17.91	13.85	9.76	42.11	11.11	1.82
bytes	32.65	18.77	28.89	58.33	19.64	2.18
speed	11.40	9.41	12.47	12.63	0.66	4.58

Slika 5.9: Poboljšanje (u %) primenom sihronizacije direktne i povratne analize

Tabela 5.4 sadrži uočene prednosti u toku ovih ispitivanja za sva tri pristupa.

Tabela 5.4: Prednosti za sva tri tipa inženjeringa

Pristup	Prednosti
Direktni	Multidimenzionalnost sistema i visok nivo apstrakcije Efikasna transparentnost strukture sistema Identifikovanje funkcionalnih celina Generisanje izvornog koda
Obrnuti	Analiza i interpretacija implementacije problema Logički dizajn i bolja vidljivost izvornog koda "Demontaža" Java koda Efikasnije održavanje softverskog rešenja
Round-trip	Sinhronizovane promene od modela do izvornog koda ili obrnuto Generisanje izveštaja koji opisuju klase (dijagrami ili tabelarno) Kombinovanje prednosti prva dva pristupa

Prednosti primene sinhronizacije direktne i povratne analize za implementacije koje realizuju metode za generisanje triangulacija su:

1. *Suočavanje sa kompleksnošću problema:* Bolje razumevanje definisanih klasa i njihovih metoda, identifikovanje međuzavisnosti, načina komunikacije i protoka podataka.

2. *Generisanje alternativnih pogleda problema:* Analiza izvornog koda sa više aspekata i kroz nekoliko modela (uglavnom statičkih i dinamičkih) pruža mogućnost dopune izvornog koda, dodavanje novih metoda, simplifikacije i redefinisanje metoda, sinteze i analize metoda itd.

3. *Otkrivanje ponovljenih slučajeva:* Nakon lociranja i uklanjanja izvornog koda ili modula koji se ne koristi, smanjuje se složenost problema i pojednostavljuje izvorni kôd. Dakle, postižu se bolji rezultati koji se odnose na obimnost samih klasa kao i na njihovu funkcionalnost.

Na osnovu datih analiza i ispitivanja može se zaključiti da je najbolja praksa u primeni tehnike sinhronizacije. Dobijeni rezultati ukazuju na poboljšanje softverskih rešenja kroz tri postavljena kriterijuma. Ova metodologija sa navedena tri pristupa se može primeniti kao nova tehnika u analizi i dizajnu algoritama za slične probleme. Generalno, ovim je predložen pristup koji je pogodan za analizu i dizajn implementacija nekih drugih algoritama računarske geometrije.

Poglavlje 6

Zaključna razmatranja

U disertaciji su analizirane nove metode i algoritmi za rešavanje problema triangulacije poligona. Rezultati disertacije se mogu svrstati u tri kategorije. Prvu kategoriju čine predložene nove tehnike i metode za generisanje triangulacija konveksnih poligona. Drugu kategoriju čine rezultati koji se odnose na nove metode za notaciju i skladištenje triangulacija, dok treću kategoriju čine rezultati iz oblasti primene *OOAD* metodologije (sa tri tipa inženjeringa) u cilju efikasne analize i dizajna algoritama za triangulaciju poligona.

Sledi kratak pregled rezultata izloženih poglavlja disertacije:

1. Data je analiza programskog jezika *Java* sa aspekta mogućnosti primene u računarskoj geometriji kroz posedovanje gotovih klasa i biblioteka. Ispitano je, koliko je ovaj programski jezik pogodan za implementaciju algoritama računarske geometrije. Urađena je komparativna analiza implementacija za *Hurtado-Noy* algoritam, u tri objektno-orijentisana programska jezika (*Java, C++, Python*). Ova analiza je imala za cilj da ukaže na adekvatan izbor aktuelnog programskog jezika, kao efikasnog alata za implementaciju, a sve radi postizanja dobrih rezultata u realizaciji novih metoda za generisanje i notaciju triangulacija poligona.

2. Predstavljene su dve nove metode za generisanje triangulacija konveksnog poligona.

   A) Prva metoda se bazira na *dekompoziciji Katalanovog broja*. Ovaj postupak proizvodi izraze u obliku sume termova $(2+i), i \in \{0, \ldots, n-4\}$, sa ciljem primene u postupku generisanja triangulacija. Na osnovu ove ideje je uspostavljeno težinsko stablo triangulacija. U cilju izbegavanja ponavljanja istih generisanja korišćena je rekurzija sa memoizacijom. Ova tehnika rešavanja problema se bazira na generisanju skupa triangulacija $\mathcal{T}_n$ koristeći skup $\mathcal{T}_{n-1}$. Kod *metode dekompozicije* se iz prethodnog nivoa preuzima $(n-4)$ parova temena unutrašnjih dijagonala koje određuju triangulaciju, a na osnovu izraza dekompozicije se preuzete triangulacije adekvatno dopunjuju. Dobijen izraz iz postupka dekompozicije jasno pokazuje koliko se prenosi triangulacija iz $\mathcal{T}_{n-1}$ u naredni nivo. Drugi sabirak u termu, označen sa i, pokazuje koliko se novih triangulacija generiše u skupu $\mathcal{T}_n$.

Kroz navedenu uporednu analizu ove metode sa *Hurtado-Noy* algoritmom, ukazano je na prednosti nove predložene metode. Ovde je bitno istaći dva aspekta koja su veoma važna za implementaciju algoritama, a to su: (1) podaci koji se očekuju na ulazu algoritma i (2) opterećenost memorije. Na osnovu numeričkih podataka koji su dobijeni eksperimentalnim ispitivanjem *Java* aplikacija, može se utvrditi sledeće: (1) Ukupno vreme (u sekundama) za svih 12 testiranja (za $n \in \{5, 6, ..., 16\}$), kod *Hurtado-Noy* algoritma je 1037.39 dok je kod *metode dekompozicije* 936.22. Znači, postignuto je prosečno ubrzanje za 1.108 odnosno poboljšanje u brzini generisanja za 10,8 %. (2) Prosečno vreme izvršavanja po jednom nivou kod *Hurtado-Noy* algoritma je 94.31 dok je kod *metode dekompozicije* 85.11. (3) Primenom *metode dekompozicije* postignuto je povećanje broja generisanih triangulacija u sekundi za 1.132 odnosno poboljšanje za 13,2 %.

B) Generalna strategija *blok metode* je razlaganje problema na manje potprobleme koji su međusobno zavisni. Svaki potproblem se rešava samo jednom a koristi se više puta. Na taj način je izbegnuto nepotrebno ponavljanje istih generisanja, jer se adekvatno dopunjuju prethodno dobijeni rezultati. Metoda se zasniva na odnosu broja triangulacija dva uzastopna poligona. *Java* sa svojim *JDBC API*-jem se pokazala kao efikasan alat za realizaciju ove metode. Navedene su mogućnosti efikasnog skladištenja i rad sa bazama podataka u *Java NetBeans* okruženju.

U komparativnoj analizi je zabeleženo prosečno vreme izvršavanja po jednom nivou kod *Hurtado-Noy* algoritma 46.16 dok je kod *blok metode* 10.74. Takođe, i ovde se ističe prednost *blok metode* u odnosu na postojeću, a to je da radi samo sa skupom temena unutrašnjih dijagonala. Kod *Hurtado-Noy* algoritma se na uzlazu očekuje kompletna struktura, jer je svaka triangulacija iz prethodnog poligona P_{n-1} opisana unutrašnjim dijagonalama i spoljašnjim ivicama. Ta obrada kompletne strukture znatno usporava rad njihove metode. Sve ovo bitno utiče i na opterećenost memorije u toku izvršavanja, pa i na brzinu generisanja triangulacija.

3. Date su nove metode za notaciju i skladištenje triangulacija sa glavnim ciljem da se uštedi memorijski prostor i obezbedi jedinstven sistem zapisivanja (notacija) triangulacija.

   A) Prva metoda za notaciju se bazira na kombinatornim problemima: *ballot* problemu i problemu puteva u mreži (*lattice path*). Metodu za *ballot* notaciju čine dva inverzna algoritma (iz *ballot* zapisa u grafički prikaz triangulacije i obrnuto). Uvedena je primena steka koji znatno ubrzava proces skladištenja triangulacija i koji daje znatno kompaktniji zapis u odnosu na postojeće notacije (*SVB* izlazi). Eksperimentalni rezultati implementirane metode ukazuju na poboljšanje u konstrukciji i notaciji triangulacija u odnosu na postojeće algoritme. I ovde je rađena komparativna analiza sa *Hurtado-Noy* algoritmom gde se primenom *SVB* izlaza može znatno uticati na uštedu memorije pri generisanju triangulacija poligona.

Na osnovu numeričkih podataka, može se videti da *Hurtado-Noy* algoritam daje bolje rezultate ako se uzme u obzir samo vreme generisanja triangulacija. Međutim, *ballot metod* daje bolje rezultate ako se posmatra ukupno vreme gde je uključeno i generisanje i skladištenje. Za ukupno vreme, za svih 11 testiranja (za $v \in \{5, 6, ..., 15\}$), postignuto je prosečno ubrzanje 1.09 odnosno smanjenje vremena za 9%. Primenom *SVB* izlaza, udeo vremena za skladištenje je manji za *ballot metod* (u opsegu od 8% do 12%, redom za svih 11 testiranja). Kod *Hurtado-Noy* algoritma, ta vrednost za ista testiranja se kreće u opsegu od 16% pa sve do 63%. Ovo značajno utiče na rezultate koji predstavljaju ukupno vreme izvršavanja programa, tačnije za slučaj istovremenog generisanja i skladištenja triangulacija.

B) Predstavljena metoda za *alfanumeričku* notaciju može naći primenu ne samo u zapisivanju triangulacija konveksnog poligona već i nekih drugih kombinatornih problema koji se baziraju na Katalanovim brojevima. Dati metod predstavlja način skraćenja postojeće BP notacije (notacija *uparenim zagradama*). Za ovaj metod su data dva algoritma koja realizuju dve transformacije (iz BP zapisa u AN i obrnuto).

Na osnovu rezultata eksperimentalnih ispitivanja, uočene su prednosti primene AN notacije, a posebno se ističe razlika u broju znakova koja je evidentna za vrednosti $n > 10$. Za sve navedene rezultate testiranja (za vrednosti $n \in \{5, \ldots, 14\}$) dobijamo prosečan koeficijent skraćenja koji je približno jednak 2.69. Sa druge tačke gledišta, veličina izlazne datoteke naglo opada primenom AN notacije. Utvrđeno je da se primenom *AN notacije* može uštedeti memorijski prostor za oko 65% za vrednosti $n > 9$ u odnosu na *BP notaciju*. Prednosti primene ove metode se mogu posmatrati kroz dva aspekta: (*i*) kroz uštedu memorijskog prostora u procesu trajnog snimanja triangulacija u vidu neke izlazne datoteke; (*ii*) i kroz uštedu radne memorije u procesu generisanja triangulacija. Ostvarivanjem ova dva cilja obezbeđujemo brže generisanje i omogućavamo postupak obrade za poligone P_n, gde je $n > 20$.

4. Naveden je novi pristup kroz objektno-orijentisanu analizu i dizajn algoritama za triangulaciju poligona. Problem triangulisanja poligona je analiziran sa tri aspekta: pristupa direktnog razvoja (*forward engineering*), sa aspekta povratne analize (*reverse engineering*) i sinhronizacije prva dva pristupa (*round-trip engineering*).

A) Bolje razumevanje i detaljna analiza algoritma za triangulaciju se postiže primenom direktnog pristupa kroz kreiranje odgovarajućih pogleda (modela) na sam problem. Na ovaj način smo dobili razvijenu strategiju planiranja implementacije problema koja je nezavisna od tehnologije i alata za implementaciju. Analiza problema triangulacije poligona je realizovana kroz *UML* modele, a dati su i primeri generisanja izvornog koda iz *UML*-a u programski jezik *Java*.

B) Uspostavljena je sinhronizacija programiranja u *Javi* i modeliranja u *UML*-u preko naprednih okruženja. Ovim je omogućen uvid u analize uticaja uvođenja odgovarajućih promena, upoznavanje koda, ponovno korišćenje, integracija određenih mo-

dula izvornog koda i sl. Neke od uočenih prednosti uspostavljene sinhronizacije za date implementacije u disertaciji su: (*i*) Suočavanje sa kompleksnošću problema, bolje razumevanje definisanih klasa i njihovih metoda, identifikovanje međuzavisnosti, načina komunikacije i protoka podataka; (*ii*) Generisanje alternativnih pogleda na problem i analiza izvornog koda sa više aspekata i kroz nekoliko modela; (*iii*) Otkrivanje ponovljenih slučajeva u izvornom kodu itd.

Data su ispitivanja i analize izvornog koda implementacija, primenom sihronizacije programiranja i modeliranja, a sve sa ciljem da se dobije bolja struktura sa što većom modularizacijom, bez ponovljenih slučajeva i sa integracijom određenih delova. Navedene analize su imale za cilj i da se smanji obimnost i složenost izvornog koda i da se utiče na funkcionalnost i rad segmenata implementacije zasebno. Dobijeni rezultati ukazuju na unapređenje softverskih rešenja, gde je postignuto poboljšanje kroz tri postavljena kriterijuma: broj linija izvornog koda (13,13%), veličina izlazne datoteke (19,51 %) i brzina izvršavanja (10,08 %).

Otvaraju se nova pitanja za dalja istraživanja u vezi ove oblasti, a odnose se na mogućnosti primene predstavljenih metoda. Budući pravci razvoja datih metoda se odnose na ispitivanje i prilagođavanje za problem kvadragulacije poligona (podela poligona na kvadrate) u vidu novih softverskih rešenja: *Java_BlockQuadr, Java_DecompositionQuadr* (moguća primena *blok metode* i/ili *metode dekompozicije* za navedeni problem). Takođe, mogu biti ispitane mogućnosti primene *ballot* i *alfanumeričke notacije* u zapisivanju i skladištenju kvadragulacija poligona.

U budućnosti se može razmatrati i o implementacijama novih metoda za pronalaženje i skladištenje optimalnih triangulacija, kao i o problemu minimalne težine triangulacija. Konačno, u cilju efikasne analize i dizajna novih algoritama za kvadragulaciju poligona i optimalne triangulacije, po predloženom modelu može biti upotrebljena objektno-orijentisana metodologija (sa tri tipa inženjeringa).

Prilozi

I) Podaci o preuzimanju *izveštaja za klase* i implementacije metoda

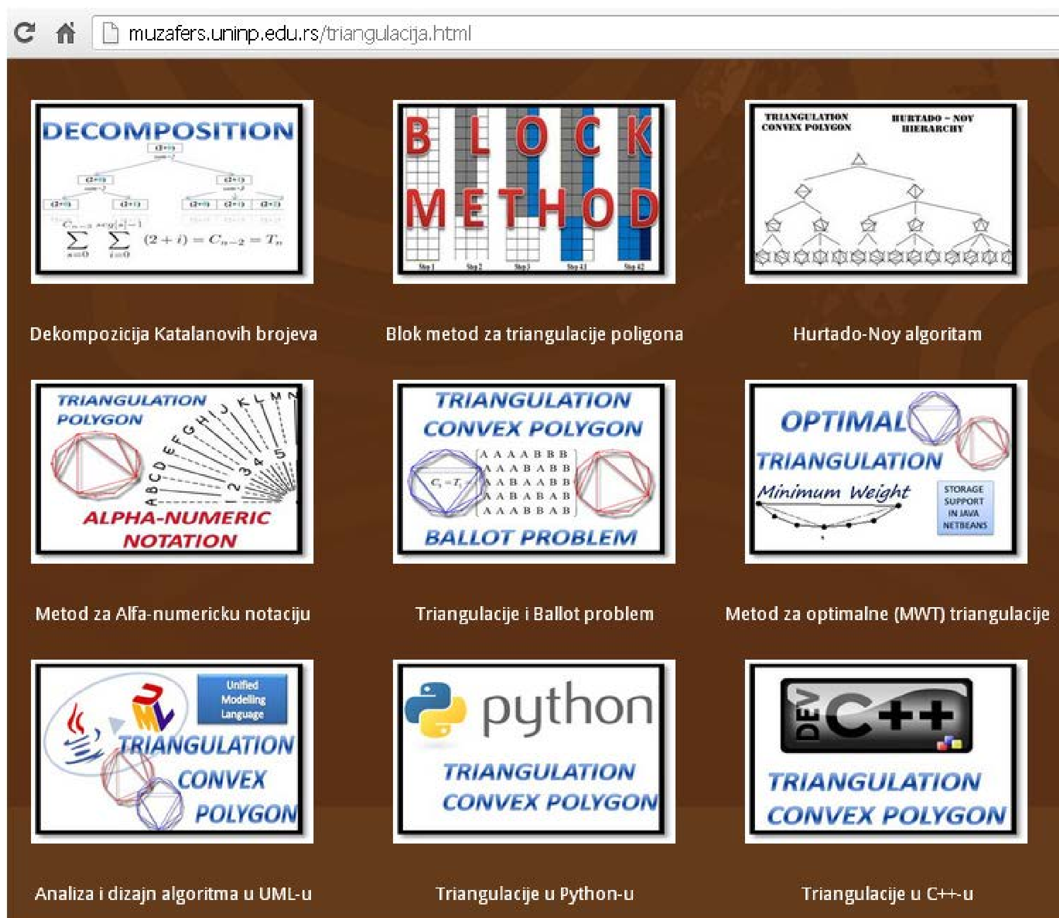
Sve opisane metode po poglavljima su realizovane kroz konkretne aplikacije:

1. Tri *Java* aplikacije za generisanje triangulacija konveksnih poligona (*Hurtado-Noy, metod dekompozicije i blok metod*);
2. Dve *Java* aplikacije za notaciju triangulacija (*alfanumerička i ballot notacija*);
3. *UML* dijagrami i kompletna dokumentacija u vidu opisa klasa i modela;
4. Implementacije *Hurtado-Noy* algoritma u *Python*-u i *C++*-u.

Za sve klase koje su sastavni deo navedenih implementacija izgenerisan je izveštaj u vidu detaljnih opisa kroz atribute, osnovne operacije, dijagrame, elemenate itd.
Na *http://muzafers.uninp.edu.rs/reportTriangulation*, može se pristupiti kompletnom izveštaju klasa. Na Slici 6.1 je prikazan deo jednog izveštaja za klasu `Triangulation`. Ovu praksu opisa standardnih klasa i paketa koristi *Java Platform Standard Edition* na svom zvaničnom sajtu: *http://docs.oracle.com/javase/6/docs/api/*.

Slika 6.1: *Izveštaj modela iz NetBeans UML-a za klasu Triangulation*

Na web strani *http://muzafers.uninp.edu.rs/triangulacija.html* mogu se preuzeti implementacije za sve pomenute metode u disertaciji (Slika 6.2).



Slika 6.2: *Web strana za preuzimanje izvršnih (exe) aplikacija*

II) *Java* izvorni kôd

U ovom prilogu su navedeni neki delovi klasa za realizaciju novih tehnika i metoda koje su opisane u trećem i četvrtom poglavlju disertacije. Istaknuti su važniji segmenti sledećih klasa: `Triangulation, GenerateTriangulations, NoteTriangulation, Point, Node, LeafNode, Database i PostScriptWriter`.

A) Klasa: *Triangulation*

Klasa `Triangulation` je zadužena za generisanje svih triangulacija u grafičkom obliku. Glavne metode klase su `Draw` i `DrawAll`.

- Metoda `Draw()` iscrtava pojedinačne triangulacije konveksnih poligona. Pomoću metode `drawLine()` se vrši spajanje temena, linijama po odgovarajućem rasporedu. Jedna kombinacija unutrašnjih dijagonala čini jednu triangulaciju poligona.

- Metoda `DrawAll()` poziva `Draw` metodu. Broj izvršavanja `Draw()` metode u okviru petlje je jednak Katalanovom broju za dato n. Na taj način se iscrtavaju sve triangulacije poligona P_n. Metoda `DrawAll()` povezuje sve triangulacije u jednu izlaznu datoteku (u različitim formatima).

```
1    import java.io.IOException;
2    import java.util.Vector;
3
4    public class Triangulation {
5      ***
6      private int sCursorX, sCursorY;
7      private Vector<Point> points;
8      private PostScriptWriter writer;
9      private Vector<Double> sSine;
10     private Vector<Double> sCosine;
11
12     public Triangulation(int edges, PostScriptWriter writer) {
13        this.sCursorX = Triangulation.XLIMIT;
14        this.sCursorY = Triangulation.YLIMIT;
15        this.points = new Vector<Point>();
16        this.writer = writer;
17        this.sSine = new Vector();
18        this.sCosine = new Vector();
19
20        for (int k=0; k < edges; k++) {
21           double d = (4*k+edges)*Math.PI/(2*edges);
22           this.sSine.add(x*Math.sin(d));
23           this.sCosine.add(y*Math.cos(d)); }
24     }
25
26     public void clear() {
27           this.points.removeAllElements(); }
```

```
28
29      public void copyFrom(int aOffset, Node t) {
30         if (!(t instanceof LeafNode)){
31            this.copyFrom(aOffset,t.getLeft());
32               this.copyFrom(aOffset+t.getLeft().leaves(),t.getRight()); }
33               this.points.add(new Point(aOffset, aOffset + t.leaves()));
34         }
35
36       public void Draw() throws IOException {
37           if (Triangulation.XLIMIT <= this.sCursorX) {
38              this.sCursorX = Triangulation.XMARGIN;
39              this.sCursorY -= Triangulation.YDELTA; }
40
41           for (int i = 0; i < this.points.size(); i++) {
42           Point p = this.points.get(i);
43           this.writer.drawLine (
44           -this.sCosine.get(p.x)+this.sCursorX,
45           this.sSine.get(p.x)+this.sCursorY,
46           -this.sCosine.get(p.y)+this.sCursorX,
47           this.sSine.get(p.y)+this.sCursorY );          }
48           this.sCursorX += Triangulation.XDELTA;
49         }
50
51       public void DrawAll(Vector<Node> trees) throws IOException {
52         this.writer.psHeader();
53
54         for (int i = 0; i < trees.size(); i++) {
55           Node t = trees.get(i);
56              this.clear();
57              this.copyFrom(0, t);
58              this.Draw();
59                   if (i != 0 && i %55 == 0) {
60                      this.sCursorX = Triangulation.XLIMIT;
61                      this.sCursorY = Triangulation.YLIMIT;
62                      this.writer.newPage(); }
63          }
64         this.writer.Trailer(); }
65    }
```

B) Klasa: *GenerateTriangulations*

Klasa `GenerateTriangulations` sadrži glavnu izvršnu metodu, kao i metodu za generisanje triangulacija. Za metode *blok, dekompoziciju i Hurtado* koristi se ista struktura ove klase, jedino se razlikuje metoda za određen način generisanja triangulacija.

U slučaju implementacije *Hurtado-Noy* algoritma to je metoda `Hurtado()`, u slučaju *dekompozicije Katalanovog broja* to je `CreateExpr()`, dok je *Java* metoda `Block()` u implementaciji opisane *blok metode.*

B.1. Klasa `GenerateTriangulations` sadrži komponente za *GUI* aplikacije iz *SWING* i *AWT* paketa:

```
1   import java.io.*;
2   import java.util.*;
3   import java.io.BufferedWriter;
4   import java.io.FileWriter;
5   import java.util.Vector;
6   import javax.swing.*;
7   import javax.swing.event.*;
8   import javax.swing.JToolBar;
9   import javax.swing.border.*;
10  import java.awt.*;
11  import java.awt.geom.*;
12  import java.awt.event.*;
13  import java.awt.Toolkit;
14
15  public class GenerateTriangulations extends JFrame implements
        ActionListener{
16  static int p;
17  JLabel nove = new JLabel("PANEL ZA UNOS NOVIH TRIANGULACIJA");
18  JLabel pregled = new JLabel("PANEL ZA PREGLED SNIMLJENIH TRIANGULACIJA");
19  JLabel labela2 = new JLabel("Unos broja temena poligona za generisanje");
20  JTextField polje = new JTextField(30);
21  JButton dugme = new JButton("Kreiranje Triangulacija");
22  JButton dugme2 = new JButton("Otvori vec kreiran fajl");
23  ***
```

B.2. Metoda `actionPerformed()` služi za dodeljivanje funkcionalnosti komponentama *GUI* aplikacije. U ovom slučaju je dodeljena funkcionalnost na dugme koje izvršava `DrawAll` metodu. Ova metoda omogućava pregled već snimljenih rezultata (grafički prikaz i notaciju).

```
1   public void actionPerformed(ActionEvent evt) {
2   if (evt.getSource()==dugme){
3     try{
4   p = Integer.parseInt(polje.getText());
5   FileWriter fstream = new FileWriter(p+"-polygons.ps");
6   BufferedWriter out = new BufferedWriter(fstream);
7   PostScriptWriter writer = new PostScriptWriter(out);
8   GenerateTriangulations app = new GenerateTriangulations();
9   Vector<Node> btrees = app.Hurtado(p-2);
10  Triangulation instanceTriangulation = new Triangulation(p,writer);
11  instanceTriangulation.DrawAll(btrees);
12  out.close(); }
13  catch (Exception e){ e.printStackTrace();}
14  this.openFile(String.valueOf(p) +"-polygons.ps"); }
15  }
```

C) Klasa: *NoteTriangulation*

Klasa `NoteTriangulation` se koristi za dve metode zapisivanja (*alfanumerička* i *ballot* notacija) koje su opisane u četvrtom poglavlju.

Obe metode koriste istu struktura ove klase, jedino se razlikuje metoda za određenu notaciju triangulacija (`ANnotation()` ili `BallotNotation()`).

C.1. Metoda `createTriangulation()` se koristi za formiranje triangulacije na osnovu dobijenog zapisa iz pomenutih metoda koje realizuju određenu notaciju.

```
1     class NotateTriangulations extends Vector implements DrawTriangPol{
2                           ***
3         public void createTriangulation() {
4             ProfilListBP = new Vector[order+1];
5             for( int i = 0 ; i < order+1 ; i++ )
6                 ProfilListBP[i] = new Vector();
7                 ProfilListBP[0].addElement( new LNodes() );
8                 instanceTriangulation = new NotateTriangulations();
9
10             for( int k = 0 ; k < order+2 ; k++ ) {
11                  double d = (2*order+2-4*k)*Math.PI/(2*order+4);
12                  Fsinus[k]   = (int)Math.floor(60*Math.sin(d));
13                  Fkosinus[k] = (int)Math.floor(60*Math.cos(d));
14                }
15
16            for( int n = 1 ; n < order+1 ; n++ ) {
17              for( int i = 0 ; i < n ; i++ ) {
18               for( int j = 0 ; j < ProfilListBP[i].size() ; j++ ) {
19                for( int k = 0 ; k < ProfilListBP[n-i-1].size() ; k++ ) {
20            ProfilListBP[n].addElement(
21              new Nodes(
22                (ListBP)ProfilListBP[i].elementAt(j),
23                (ListBP)ProfilListBP[n-i-1].elementAt(k) ));}
24                    }
25                }
26             }
27         }
```

C.2. Upotreba `BufferedWriter` iz `java.io.BufferedWriter` za snimanje rezultata u formi određene notacije je realizovana sledećim segmentom izvornog koda u *Javi*:

```
1   n=1;
2   for (i=0;i<n;i++){
3      BufferedWriter bufferedWriter = null;
4        try {
5            bufferedWriter = new BufferedWriter(
6            new FileWriter("notation.jdb",true));
7            bufferedWriter.write(LabelBP+""+LabelBP1+""+LabelAlfa+CentralBin)
                 ;
```

```
8                bufferedWriter.write(LabelAlfa+CentralDec);
9                bufferedWriter.newLine(); }
10               catch (FileNotFoundException ex) {
11               ex.printStackTrace(); }
12               catch (IOException ex) {
13               ex.printStackTrace(); }
14               finally {
15               try {
16                   if (bufferedWriter != null) {
17                       bufferedWriter.flush();
18                       bufferedWriter.close();
19                       bufferedWriter1.flush();
20                       bufferedWriter1.close();
21                   }
22               } catch (IOException ex) {
23                   ex.printStackTrace(); }
24           }
25       }
```

D) Klase: *Node, LeafNode, Point*

Klase koje su zadužene za rad sa temenima (kretanje kroz triangulaciju) su: `Node` i njena klasa naslednica `LeafNode`, kao i klasa `Point` koja je zadužena za rad sa koordinatama temena poligona.

Glavne metode klase `Node` su `setLeft, getLeft, getRight, setRight.` Ove metode obezbeđuju manipulaciju čvorovima.

```
1    public class Node {
2         private Node left;
3         private Node right;
4         public Node(Node left , Node right) {
5            super();
6            this.left = left;
7            this.right = right;}
8
9        public Node copy() {
10       return new Node(this.left.copy(),this.right.copy());
11       }
12       public int leaves() {
13       return this.left.leaves() + this.right.leaves();
14       }
15       public int leftBranch() {
16       return this.left.leftBranch() + 1;
17       }
18       public String toString(){
19       return 'move' + this.left.toString() + this.right.toString() + 0;
20       }
```

```
21          public Node getLeft() {
22          return left;
23          }
24          public void setLeft(Node left) {
25          this.left = left;
26          }
27          public Node getRight() {
28          return right;
29          }
30          public void setRight(Node right) {
31          this.right = right;
32          }
33      }
34
35      public class LeafNode extends Node {
36      public LeafNode() { super(null, null);}
37        public LeafNode copy() {return new LeafNode();}
38        public int leaves() {return 1;}
39        public int leftBranch() {return 0;}
40        public String toParen() {return "+";}
41        public String toString() {return "1";}
42      }
43
44      public class Point {
45        public int x,y;
46        public Point(int x, int y) {
47           super();
48           this.x = x;
49           this.y = y; }
50      }
```

E) Klasa: *Database*

Kod *blok* metode je omogućen rad sa *JDBC API*-jem, odnosno sa okruženjem koje obezbeđuje rad sa bazama podataka u *Javi*. Klasa `Database` omogućava povezivanje na bazu i učitavanje zapisa triangulacija, kao i skladištenje rezultata u različitim formatima.

```
1       import java.sql.*;
2
3       class Database {
4         java.sql.Connection dbConn=null;
5         Statement naredba1=null;
6
7       public void poveziSaBazom(){
8           try {
9           Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
10          String sourceURL= "jdbc:odbc:BlockTriangulation";
11          dbConn= DriverManager.getConnection(sourceURL);
```

```
12          System.out.println ("Veza sa bazom je: "+dbConn);
13          System.out.println ("Veza uspostavljena\\n");
14          naredba1=dbConn.createStatement(); }
15          catch (Exception e){
16          System.out.println ("Izuzetak izbacen"+ e.getMessage()); }
17      }
18
19      public  void zatvoriBazu(){
20          try {
21          if (naredba1 != null) naredba1.close();
22          if (dbConn != null) dbConn.close(); }
23          catch (SQLException sqle) {
24          System.out.println("SQLIzuzetak - close(): " + sqle.getMessage()); }
25           }
26      }
```

F) Klasa za kreiranje izlazne datoteke

Klasa za kreiranje izlazne datoteke je `PostScriptWriter` i ona sadrži ključne metode za grafički prikaz i snimanje generisanih triangulacija.

```
1       import java.io.BufferedWriter;
2       import java.io.IOException;
3
4       public class PostScriptWriter {
5           public PostScriptWriter(BufferedWriter out) {
6            this.out = out; }
7           public void psHeader() throws IOException {
8            this.out.write("\%!PS-Adobe-3.0\r"); }
9           public void write(String msg) throws IOException {
10           this.out.write(msg); }
11          public void drawLine(double x1, double y1, double x2, double y2)
12          throws IOException {
13              this.out.write(
14              String.format((int)x1, (int)y1, (int)x2, (int)y2)); }
15          public void drawDot(int xc, int yc) throws IOException {
16              this.out.write(String.format("\%d \%d 2.25 dot\r"+"k",xc,yc)); }
17          public void newPage() throws IOException{
18              this.out.write("showpage\r"); }
19      }
```

Spisak algoritama

1.3.1 *Hurtado-Noy* algoritam . . . 8
3.1.1 Dekompozicija Katalanovog broja C_{n-2} u sumu termova $(2+i)$ . . . 27
3.1.2 Izračunavanje suma u termovima $(2+i)$ . . . 28
3.1.3 Formiranje četvorougla . . . 30
3.1.4 Kompletiranje dva kopirana reda . . . 30
3.1.5 Kompletiranje "novih" triangulacija . . . 30
3.2.1 Eliminacija parova . . . 38
3.2.2 Formiranje četvorougla . . . 39
3.2.3 Algoritam za *blok metod* . . . 39
4.1.1 Transformacija iz *ballot* zapisa u triangulaciju (kretanje kroz poligon) . . . 52
4.1.2 Pronalaženje mogućih ukrštanja u poligonu . . . 52
4.1.3 Transformacija iz triangulacije u *ballot* zapis (kretanje kroz triangulaciju) . . . 54
4.1.4 Mapiranje *SVB* u *ballot* zapis . . . 56
4.2.1 Transformacija iz BP u AN notaciju . . . 63
4.2.2 Obrnuta transformacija iz AN u BP notaciju . . . 67

Spisak tabela

1.1 Vrednosti prvih 30 Katalanovih brojeva 4

2.1 Statistika popularnosti programskih jezika (OOP jezici, 2007-2013) 10
2.2 Statistika popularnosti programskih jezika (svi prog. jezici, 1998-2013) . . . 10
2.3 Klase za implementaciju *Hurtado-Noy* algoritma 15
2.4 Komparativna analiza za tri implementacije 20

3.1 Eksperimentalni rezultati testiranja: *dekompozicija i Hurtado-Noy* 33
3.2 Eksperimentalni rezultati testiranja: *blok i Hurtado-Noy* 42

4.1 Redosled glasanja u *ballot* zapisu *AABABB* 49
4.2 Eksperimentalni rezultati za primenu *ballot* notacije 57
4.3 Komparacija metoda: *ballot* i *Hurtado-Noy* 58
4.4 Šifrarnik za prvi slučaj 65
4.5 Transformacija iz BP u AN, za slučaj gde se uzimaju po dva bita 65
4.6 Transformacija iz BP u AN, za slučaj gde se uzimaju po tri bita 66
4.7 Šifrarnik za drugi slučaj 66
4.8 Eksperimentalni rezultati za AN i BP notaciju 69

5.1 Pregled *UML* dijagrama po modelima 75
5.2 Zahtevi u *round-trip* inženjeringu za konkretne klase 83
5.3 Poboljšanje implementacija za generisanje triangulacija 84
5.4 Prednosti za sva tri tipa inženjeringa 85

Spisak slika

1.1 Triangulacija konveksnog poligona za $n \in \{3, ..., 6\}$ . . . 5
1.2 Prostupak "cepanja" dijagonale i konstrukcija potomka . . . 7
1.3 Primer generisanja dva potomka: $S^1(\tau^l_{n-1})$ i $S^{n-1}(\tau^l_{n-1})$ . . . 7
1.4 *Hurtado-Noy* hijerarhija . . . 8

2.1 Neke operacije klase *GeometryInfo* . . . 13
2.2 Sadržaj paketa `Geom`: *TriangulationUtilis* i *SeidelTriangulator* . . . 14
2.3 Merenje performansi za *BinaryTrees* algoritam u *Javi, Python*-u i *C++*-u . . . 15
2.4 Generisanje triangulacija po *Hurtado-Noy* hijerarhiji za $n \in \{3, ..., 6\}$ . . . 17
2.5 *Hurtado-Noy* redosled triangulacija za nepravilan konveksni šestougao . . . 19
2.6 Broj generisanih triangulacija u sekundi za $n \in \{7, 8, \ldots, 14\}$ . . . 20
2.7 Komparativna analiza: *Java, Python i C++* . . . 21

3.1 Težine ivica koje povezuju "roditelja" i njegove "potomke" . . . 24
3.2 Težinsko stablo triangulacija za $n \in \{3, ..., 6\}$ . . . 24
3.3 Izabrani deo stabla sa više detalja . . . 25
3.4 Početni deo za $\mathcal{T}_6$ generisan kopiranjem triangulacija iz $\mathcal{T}_5$ . . . 29
3.5 Generisanje za $\mathcal{T}_6$ bazirano na $\mathcal{T}_5$ i odgovarajućeg izraza dekompozicije . . . 31
3.6 Odnos performansi $r_{n,s}$ na osnovu CPU vremena . . . 34
3.7 Funkcija $\rho_s(\tau)$: *metod dekompozicije* i *Hurtado-Noy* . . . 34
3.8 *Java* aplikacija za generisanje triangulacija na osnovu *dekompozicije* . . . 36
3.9 Dopuna triangulacije $\delta_{2,4}$; $\delta_{2,5}$ u skupu triangulacija $\mathcal{T}_6$ . . . 38
3.10 Popunjavanje tabele na osnovu algoritma za *blok metod* . . . 40
3.11 Generisanje skupa triangulacija $\mathcal{T}_6$ . . . 42
3.12 Odnos performansi (CPU vreme): *blok metoda* i *Hurtado-Noy* . . . 43
3.13 Kreiranje baze podataka za blokove triangulacija . . . 46
3.14 Mehanizam pristupa bazi podataka . . . 46
3.15 Generisanje tabela baze *BlockTriangulation* . . . 46
3.16 *Java* aplikacija za *blok metodu* . . . 47

4.1 Kretanje kroz putanju na mreži i odgovarajući *ballot* zapis *AABABB* . . . 49
4.2 Generalna forma kretanja kroz poligon . . . 51
4.3 Konstrukcija triangulacije zasnovana na *ballot* zapisu *AABABB* . . . 53

4.4 Kretanje kroz triangulaciju $\delta_{2,5}$ i $\delta_{3,5}$ 54
4.5 Odnos ulaza i izlaza steka, $2n : n$ 55
4.6 Primer obrade *ballot* zapisa pomoću steka 55
4.7 *Ballot* zapisi za određene triangulacije šestougla 60
4.8 *Ballot notacija* za određene triangulacije šestougla 61
4.9 Primer transformacije iz BP u AN notaciju 64
4.10 Ilustracija obrnute transformacije (iz AN u BP notaciju) 68
4.11 Odnos u broju znakova za AN i BP notaciju 69
4.12 AN i BP notacije triangulacija šestougla 72

5.1 Tri OOAD pristupa (inženjeringa) 74
5.2 Dijagram klasa (metode za generisanje triangulacija) 76
5.3 Dijagram klasa (metode za notaciju triangulacija) 77
5.4 Dijagram slučaja korišćenja 78
5.5 Dijagram sekvenci: Klase *Triangulation* i *NoteTriangulations* 78
5.6 Dijagram interakcije 79
5.7 Dijagram komponenti 80
5.8 Dijagram realizacije 81
5.9 Poboljšanje (u %) primenom sihronizacije direktne i povratne analize 85

6.1 *Izveštaj modela iz NetBeans UML-a za klasu Triangulation* 91
6.2 *Web strana za preuzimanje izvršnih (exe) aplikacija* 92

Literatura

[1] J. Belt, *Automated Consistency Checking between UML State Charts and Sequence Diagram*, presentation of Masters Project for CIS 798, 2005.

[2] E. D. Dolan, J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming **91**, (2002), 201–213.

[3] D. Berardi, D. Calvanese, G. Degiacomo, *Reasoning on UML class diagrams*, Artificial Intelligence **168**, (2005), 70–118.

[4] M. Berg, O.Cheong, M. Kreveld, M. H. Overmars, *Computational geometry: Algorithms and applications: 3rd edition*, Springer Verlag, 2008.

[5] J. Bringer, H. Chabanne, *Code Reverse Engineering Problem for Identification Codes*, IEEE transactions on information theory **58 (4)**, (2012), 2406–2412.

[6] B. Bruegge, A. Dutoit, *Object-Oriented software engineering using UML, patterns and Java*, Pearson Education, New Jersey, (2004), 568-576.

[7] H. Bock, *The Definitive Guide to NetBeansTM Platform 7*, Apress, 2011.

[8] D.M. Campbell, *The computation of Catalan numbers*, Mathematics Magazine **57 (4)**, (1984), 195–208.

[9] B. Chazelle, *Triangulation a Simple Polygon in Linear Time*, Discrete Computational Geometry **6**, (1991), 485-524.

[10] B. Chazelle, L. Palios, *Decomposition Algorithms in Geometry*, Algebraic Geometry and its Application (ed. Ch. L. Bajaj) **27**, (1994), 419–447.

[11] B. Chen, H. Harry, *Interpretive OpenGL for Computer Graphics*, Computers and Graphics **29 (2)**, (2005), pp. 331–339.

[12] J. X. Chen, C. Chen, *Foundations of 3D Graphics Programming: Using JOGL and Java3D*, Springer, 2008.

[13] K.G. Choo, *Catalan Numbers*, Mathematics Department of University of Sydney, preuzeto sa http://www.maths.usyd.edu.au/u/kooc/catalan/cat1fbp.pdf

[14] T. H. Cormen, C.E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms, Second Edition*, The MIT Press, 2001.

[15] R. Dantas, *NetBeans IDE 7 Cookbook*, Pack publishing, 2011.

[16] G. Davis, *Learning Java Bindings for OpenGL*, Author-House publisher, 2004.

[17] T. Davis, *Catalan Numbers*, Mathematical Circles Topics, November 24, 2010, preuzeto sa: http://www.geometer.org/mathcircles/catalan.pdf

[18] T. A. Dowling, *Catalan Numbers*, Department of Mathematics of Ohio State University, preuzeto sa http://www.mhhe.com/math/advmath/rosen/r5/ instructor/applications/ch07.pdf.

[19] D. Du, F. Hwang, *Computing in Euclidean Geometry*, World Scientific, 1992.

[20] D. J. Evans, F. Abdollahzadeh, *Eficient construction of balanced binary trees*, The Computer Journal **26 (3)**, (1983), 193-195.

[21] S. Even, *Graph Algorithms*, Cambridge University Press, New York, 2011.

[22] W. Feller, *An Introduction to Probability Theory and its Applications, Volume I (3rd ed.)*, Wiley publisher, (1968), p. 69.

[23] M.R. Garey, D.S. Johnson, F.P. Preparata, R.E. Tarjan, *Triangulating a simple polygon*, Inform. Process. Lett. **7**, (1978), 175–180.

[24] D. M. Geary, *Graphic Java 2: Swing, volume 2 / 3rd edition*, Prentice Hall Professional, 1999.

[25] F.R. Geary, N. Rahman, R. Raman, V. Raman, *A Simple Optimal Representation for Balanced Parentheses*, Theoretical Computer Science **368 (3)**, (2006), 231–246.

[26] S. Gog, J. Fischer, *Advantages of shared data structures for sequences of balanced parentheses*, DCV'10: Data Compression Conf. 2010, 406-415.

[27] S. S. Gupta, K. Mukhopadhyaya, B. B. Bhattacharya, B. P. Sinha, *Geometric Classification of Triangulations and Their Enumeration in a Convex Polygon*, Computers Math. Applic. **27 (7)**, (1994), 99–115.

[28] P. Hilton, J. Pedersen, *Catalan Numbers, Their Generalization, and Their Uses*, The mathematical intelligencer **13 (2)**, (1991), 64–75.

[29] F. Hurtado, M. Noy, *Graph of Triangulations of a Convex Polygon and Tree of Triangulations*, Computational Geometry **13** (1999), 179–188.

[30] F. Hurtado, M. Noy, *Ears of triangulations and Catalan numbers*, Discrete Mathematics **149** (1996), 319–324.

[31] D. R. Heffelfinger, *Java EE 6 Development with NetBeans 7*, Pack publishing, 2011.

[32] O. Hjelle, M. Daehen, *Triangulations and Applications, Mathematics And Visualization*, Springer, 2006.

[33] S. Huynh, Y. Cai, W. Shen, *Automatic Transformation of UML Models into Analytical Decision Models*, Technical Report DU-CS-08-01, Drexel University, 2008.

[34] X. Jiang, *Applications of Catalan numbers*, Department of Mathematical Sciences, Sweet Briar College, Version of July 12, 2012, preuzeto sa: http://www.sbc.edu/sites/default/files/Honors/XiaotongJiang.July20_0.pdf

[35] H. C. Kazi, B. Stanley, D. Lilja, *Techniques for Obtaining High Performance in Java Programs*, ACM Computing Surveys **32 (3)**, (2000), 213–240.

[36] F. Klawonn, *Introduction to Computer Graphics: Using Java 2D and 3D: Second Edition*, Springer, 2012.

[37] A. Knapp, S. Merz, *Model Checking and Code Generation for UML State Machines and Collaborations Tools for System Design and Verification*, Institut fur Informatik: Universitat Augsburg publications **11**, (2002), 59–64.

[38] D. E. Knuth, *The Art of Computer Programming (vol4: Generating all trees: history of combinatorial generation*), Addison-Wesley Professional, 2006.

[39] T. Koshy, *Catalan Numbers with Applications*, Oxford University Press, New York, 2009.

[40] T. Koshy, G. Zhenguang, *Some divisibility properties of Catalan numbers* Mathematical Gazette **95**,(2011), 96102.

[41] C.K. Kozen, *Automata and Compatibility*, Springer Science and Business Media, New York, 1997.

[42] K. Lano, *Advanced Systems Design with Java, UML, and MDA*, Elsevier publisher, 2005.

[43] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition,* Prentice Hall, 2004.

[44] M. J. Laszlo, *Computational geometry and computer graphics in C++*, Prentice Hall, Book, 1996.

[45] J. R. Loera, J.Rambaou, F. Santos, *Triangulations: Structures for Algorithms and Applications*, Springer Verlag, 2010.

[46] J. R. Loera, J.Rambaou, F. Santos, *Triangulations Of Point Sets: Applications, Structures, Algorithms*, Universidad de Cantabria, (July 21, 2003), preuzeto sa: http://personales.unican.es/ santosf/MSRI03/chapter1.pdf

[47] H. Lu, C. Yeh, *Balanced Parentheses strike back*, ACM Transactions on Algorithms **4 (3)**, (2008), 1-13.

[48] S. Mašović, M. Saračević, H. Kamberović, *Java technology in the design and implementation of web applications*, Technics Technologies Education Management **7 (2)**, (2012), 504–512.

[49] S. Mašović, M. Saračević, P. S. Stanimirović, *Alpha-Numeric notation for one Data Structure in Software Engineering*, Acta P.Hungarica: Journal of Applied Sciences, (2013) (accepted).

[50] I. Munro, V. Raman, *Succinct representation of Balanced Parentheses, static trees and planar graphs*, In Proceedings of the 38th Annual Symposium on Foundations of Computer Science, IEEE, Miami Beach, Florida, 1997, 118-126.

[51] A. Narkhede, D. Manocha, *Fast Polygon Triangulation based on Seidel's Algorithm*, Department of Computer Science, UNC Chapel Hill, preuzeto sa: http://www.cs.unc.edu/ dm/CODE/GEM/chapter.html

[52] *NetBeans IDE UML Features, User's Guide*, link: http://netbeans.org/features/uml/

[53] M. Saračević, P.S. Stanimirović, S. Mašović, E. Biševac, *Implementation of the convex polygon triangulation algorithm*, Facta Universitatis, series: Mathematics and Informatics **27(2)**, (2012), 213–228.

[54] M. Saračević, P.S. Stanimirović, S. Mašović, *Implementation of some algorithms in computer graphics in Java*, Technics Technologies Education Management **8(1)**,(2013), 293–300.

[55] M. Saračević, P.S. Stanimirović, P. V. Krtolica, S. Mašović, *Ballot Notation for Convex Polygon Triangulation* (submitted).

[56] M. Saračević, P.S. Stanimirović, S. Mašović, *Forward, Reverse and Round-trip Engineering for Polygon Triangulation Algorithm* (submitted).

[57] M. Saračević, S. Mašović, H. Kamberović, *Application of JAVA and UML tools to better quality of some matrices computations*, Communications in Dependability and Quality Management **15 (3)**,(2012), 21–31.

[58] M. Saračević, S. Mašović, H. Kamberović, *Netbeans profiler as a tool for quality software assurance*, 3rd DQM international conference on life cycle engineering and management (2012), 142-147.

[59] M. Saračević, S. Mašović, H. Kamberović, *Implementacija nekih algoritama računarske grafike u JAVA NETBEANS okruženju*, XVI International Scientific and professional conference – Information Technology (2012), 136-140.

[60] M. Saračević, *Objektno-orijentisano programiranje i modelovanje - JAVA i UML, sa praktičnim primerima*, Izdavački centar Univerziteta u Novom Pazaru, 2011.

[61] M. Saračević, S. Mašović, E. Međedović, *Application of object-oriented analysis and design in navigation systems and transport networks*, 10th International Conference - Research and Development in Mechanical Industry (2010), 656–664.

[62] P. Schneider, D. Eberly, *Geometric Tools for Computer Graphics*, Morgan Kaufmann Publishers, San Francisco, 2003.

[63] D. Singmaster, *An elementary evaluation of the Catalan numbers*, American Math. Monthly **85**, (1978), 366–368.

[64] H.A. Sowizral, M.F. Deering, *The Java 3D API and virtual reality*, IEEE Computer Graphics and Applications **19 (3)**, (1999), 12–15.

[65] P.S. Stanimirović, P.V. Krtolica, M. Saračević, S. Mašović, *Block Method for Triangulation Convex Polygon*, ROMJIST - Journal of Information Science and Technology **15(4)**, (2012), 344–354.

[66] P.S.Stanimirović, M.B.Tasić, M.Saračević, S.Mašović, *UML-based modeling for Moore-Penrose inverse computation*, Metalurgia International **17(12)**, (2012), 99–106.

[67] P.S. Stanimirović, P.V. Krtolica, M. Saračević, S. Mašović, *Decomposition of Catalan numbers and Convex Polygon Triangulations*, International Journal of Computer Mathematics, (2013), DOI:10.1080/00207160.2013.837894.

[68] R. P. Stanley, *Catalan addendum to Enumerative Combinatorics, Vol. 2*, MIT Mathematics, version of 13 July 2012, preuzeto sa *Richard Stanleys home page* http://www-math.mit.edu/ rstan/ec/.

[69] C. Sukić, M. Saračević, *UML and JAVA as effective tools for implementing algorithms in computer graphics*, TEM (Technology, Education, Management): Journal of Association for Information Communication Technology Education and Science **1(2)**,(2012), 111–117.

[70] C. Riva, P. Selonen, T. Systa, J. Xu, *UML-based reverse engineering and model analysis approaches for software architecture maintenance*, In 20th IEEE international conference on software maintenance, 2004, 50–59.

[71] F. Ruskey, A. Williams, *Generating balanced parentheses and binary trees by prefix shifts*, In CATS '08: Fourteenth Computing Vol. 77, 2008.

[72] P. Tonella, *Reverse engineering of object oriented code*, In 27th International Conference on Software Engineering, 2005, 724–725.

[73] J. Tsay, *Designing a systolic algorithm for generatng well-formed parenthesis strings*, Parallel Processing Letters **14**, (2004), 83-97.

[74] *Visual Paradigm for UML, User's Guide*, link: http://www.visual-paradigm.com/

[75] L. Zhen-ping, H. Huai-jian, L. Qiang, *Study of the technology of 3D modeling and visualization system based on Python*, Changjiang Water Resources Com., China, 2008.

[76] J. Zukowski, *Java AWT reference*, O'Reilly Media publisher, 1997.

Biografija autora

Muzafer Saračević je rođen 1984. godine u Novom Pazaru. Osnovnu školu i gimnaziju je završio u rodnom gradu, kao nosilac Vukovih diploma. Studije na Fakultetu za informatiku i informacione tehnologije Univerziteta u Novom Pazaru, upisao je 2003. godine na smeru za diplomirane inženjere informacionih tehnologija. Osnovne studije je završio 2007. godine kao student generacije, na nivou svih fakulteta Univerziteta u Novom Pazaru, sa prosečnom ocenom 9,88. Iste godine je angažovan kao saradnik u nastavi. Postdiplomske (master) studije je završio na Fakultetu tehničkih nauka Univerziteta u Kragujevcu, sa prosečnom ocenom 10, na studijskom programu: Tehnika i informatika. Doktorske studije je upisao 2008. godine na Prirodno-matematičkom fakultetu Univerziteta u Nišu, na Departmanu za računarske nauke. Sve ispite i studijsko-istraživačke radove položio je u roku sa prosečnom ocenom 9,25.

Autor je oko 70 naučnih i stručnih radova, od toga preko 20 naučnih radova u međunarodnim i domaćim časopisima, od kojih je 6 u časopisima sa SCI/SCIE liste. Autor je jedne zbirke zadataka i jednog praktikuma. Usavršavao se iz oblasti dizajna i programiranja baza podataka u trajanju od dva semestra (dva nivoa). Radio je recenzije za tri međunarodna i dva domaća časopisa, kao i za jednu međunarodnu konferenciju.

Od 2007. godine, kao saradnik u nastavi na Fakultetu za informatiku i informacione tehnologije Univerziteta u Novom Pazaru, izvodio je vežbe iz sledećih predmeta: *Dizajn aplikativnog softvera, Informacioni sistemi, Baze podataka, Operativni sistemi, Multimedijalni sistemi, Web dizajn, Verovatnoća i statistika.* Od 2009. godine, angažovan je kao asistent na Departmanu za tehničke nauke, na predmetima: *Strukture podataka i algoritmi, Uvod u programiranje, Programski jezici, Objektno-orijentisano programiranje i Softversko inženjerstvo.* Radio je tri godine kao nastavnik matematike i dve godine u jednoj računarskoj kompaniji kao menadžer za hardverske proizvode.

Oblasti interesovanja su mu: *Objektno-orijentisano programiranje i modeliranje; Računarska geometrija i grafika; Alati za dinamičku matematiku; Alati, tehnologije i sistemi za e-učenje; Projektovanje informacionih sistema ...*