REPUBLIKA E SHQIPËRISË
UNIVERSITETI POLITEKNIK I TIRANËS
FAKULTETI I TEKNOLOGJISË SË INFORMACIONIT
DEPARTAMENTI I INXHINIERISË INFORMATIKE

INA PAPADHOPULLI

Për marrjen e gradës

"Doktor"

në "Teknologjitë e Informacionit dhe Komunikimit"

drejtimi "Inxhinieri Informatike"

DISERTACION

PËRCAKTIMI I NJË FUNKSIONI VLERËSIMI PËR TESTIMIN E BAZUAR NË KËRKIM TË KLASAVE NË JAVA, I CILI BAZOHET NË GJENDJET E ARRITURA TË OBJEKTIT

Udhëheqës Shkencor
PROF.AS. ELINDA MEÇE

Tiranë, 2016

PËRCAKTIMI I NJË FUNKSIONI VLERËSIMI PËR TESTIMIN E BAZUAR NË KËRKIM TË KLASAVE NË JAVA, I CILI BAZOHET NË GJENDJET E ARRITURA TË OBJEKTIT

Disertacioni
i paraqitur në Universitetin Politeknik të Tiranës
për marrjen e gradës
"Doktor"
në "Teknologjitë e Informacionit dhe Komunikimit"
drejtimi "Inxhinieri Informatike"

nga
Znj. Ina Papadhopulli
2016

Disertacioni i shkruar nga

Ina Papadhopulli

Master Shkencor, Universiteti Politeknik i Tiranës, Shqipëri, 2011

Diplomë e nivelit të parë, Universiteti Politeknik i Tiranës, Shqipëri, 2007

I aprovuar nga

Prof. Dr. Aleksandër Xhuvani, Kryetari i Jurisë së Disertacionit të Doktoraturës

Prof. Asoc. Igli Tafa, Anëtar i Jurisë së Disertacionit të Doktoraturës

Prof. Dr. Dhimitër Tole, Anëtar i Jurisë së Disertacionit të Doktoraturës

Prof. Dr. Luan Karçanaj, Anëtar i Jurisë së Disertacionit të Doktoraturës

Prof. Asoc. Marenglen Biba, Anëtar i Jurisë së Disertacionit të Doktoraturës

I pranuar nga

Prof. Asoc. Vladi Koliçi, Dekan, Fakulteti i Teknologjisë së Informacioni

Miratuar nga

Prof.Dr. Andrea Maliqari, Rektor i UPT

_____ , Këshilli i Profesorëve, FTI

Abstrakt

PËRCAKTIMI I NJË FUNKSIONI VLERËSIMI PËR TESTIMIN E BAZUAR NË KËRKIM TË KLASAVE NË JAVA, I CILI BAZOHET NË GJENDJET E ARRITURA TË OBJEKTIT

Testimi është një nga aktivitetet themelore të proçesit të zhvillimit të software-ve. Megjithatë, faza e testimit kërkon më shumë se gjysmën e burimeve të këtij proçesi. Një nga problemet kryesore është gjenerimi i rasteve të vlefshme të testimit, e cila është një detyrë e vështirë edhe që kërkon shumë kohë duke konsideruar kompleksitetin e software-ve që ekzistojnë sot. Për këtë arsye janë propozuar disa teknika për të automatizuar gjenerimin e rasteve të testimit.

Ky punim disertacioni është fokusuar tek problemi i gjenerimit automatik të rasteve të testimit për klasat në Java, mbi kriteret e testimit strukturor, duke përdorur teknikat e bazuara në kërkim.

Implementimi i teknikave të bazuara në kërkim për gjenerimin automatik të rasteve të testimit, ka dhënë rezultate mjaft premtuese për punën në të ardhmen. Ekzistojnë disa teknika që bazohen në kërkim dhe që mund të përdoren në kontekstin e testimit, por midis tyre, algoritmat gjenetikë sot konsiderohen si teknikat më efiçiente. Komponenti i algoritmit gjenetik që ndikon më shumë në rezultatet e përftuara, është funksioni i vlerësimit, i cili drejton kërkimin. Ky funksion përcakton sa afër është një individ për të përmbushur një qëllim të caktuar mbulimi. Si rrjedhojë e numrit të madh të konstrukteve të programeve, janë propozuar një shumëllojshmëri kriteresh mbulimi, të cilat kanë fortësi të ndryshme. Kriteri më i përdorur sot, është mbulimi i degëve. Megjithatë, arritja e një mbulimi të lartë ose të plotë të degëve, nuk nënkupton detyrimisht që set-i i rasteve të gjeneruara ka cilësi të mirë.

Në programet e orientuara nga objekti, gjendjet që arrin objekti gjatë ekzekutimit, ndikojnë tek sjellja e tij. Për këtë arsye, rastet e testimit që e vendosin objektin në gjendje të reja, paraqesin interes në kontekstin e testimit. Në këtë punim propozohet një funksion i ri vlerësimi,

që merr në konsideratë tre faktorë: nivelin e afërsisë, distancën e degëve dhe gjendjet e reja në të cilat një rast testimi vendos objektin nën testim. Në versionin e propozuar, qëllimet e mbulimit mbeten sërisht degët e klasës nën testim, por gjatë kërkimit, gjendja e objektit evolon me qëllim gjenerimin e individëve që nxjerrin tipare të fshehura të klasës dhe për rrjedhojë kanë probabilitet të lartë për të zbuluar gabime. Funksioni i propozuar i vlerësimit është implementuar në mjetin eToc.

Eksperimentet e zhvilluara në disa klasa me kod të hapur, të cilat kanë tipare dhe kompleksitet të ndryshëm, tregojnë se mënyra e propzuar është efikase. Vlerësimi i performancës bazohet tek mbulimi i degëve, rezultati i mutacionit dhe tek gjatësia e bashkësisë së rasteve të gjeneruara. Rezultatet e arritura tregojnë se, përdorimi i funksionit të ri të vlerësimit kundrejt atij origjinal nuk sjell ndryshim tek mbulimi i degëve, sjell një rritje relative prej 15.6% tek vlera mesatare e rezultatit të mutacionit të arritur dhe një rritje mesatare prej 12.6% tek gjatësia mesatare e bashkësisë së rasteve të gjeneruara.

Fjalë Kyçe—Testimi Strukturor, Testimi i Bazuar në Kërkim, Algoritmat Gjenetikë, Funksioni i Vlerësimit, Gjendja e Objekteve, Kriteret e Mbulimit, Mbulimi i Degëve, Rezultati i Mutacioni

Falenderime

Në rradhë të parë falenderoj udhëheqësen shkencore, Prof. Asoc. Elinda Meçe, për orientimin dhe ndihmën e dhënë në realizimin me sukses të këtij punimi. Gjithashtu e falenderoj Prof. Asoc. Elinda Meçe edhe në rolin e shefes së Departamentit të Inxhinierisë Informatike për pozitivitetin dhe përkrahjen e vazhdueshme në zgjidhjen e çdo problemi të shfaqur gjatë gjithë ciklit të doktoraturës.

Dëshiroj të falenderoj Dekanin e Fakultetit të Teknologjisë së Informacionit, Prof. Asoc. Vladi Koliçi, për punën e tij në ndjekjen e mbarëvajtjes së studimeve të doktoraturës.

Falenderoj gjithë kolegët e mi me të cilët kam shkëmbyer ide që më kanë ndihmuar gjatë punës sime.

Një falenderim i përzemërt është për familjen time që më ka dhënë forcën edhe motivimin gjatë gjithë studimeve të mija. Pa përkrahjen dhe ndihmën e tyre ky punim nuk do të kishte qenë i mundur.

Ina Papadhopulli

Tiranë, 2016

PËRMBAJTJA

LISTA E FIGURAVE

LISTA E TABELAVE

FJALOR TERMINOLOGJIK

Algoritma Metaheuristikë: Optimizimi iterativ i zgjidhjeve kandidate për probleme të ndryshme. Këto algoritma nuk sigurojnë gjetjen e zgjidhjeve globale optimale, por gjejnë zgjidhje afër optimales.

Buxheti i Kërkimit: numri maksimal i vlerësimeve të funksionit objektiv (numri i kandidatëve të zgjedhur) që pajtohet me kohën maksimale të ekzekutimit të algoritmit.

Oracle: Një mekanizëm për të përcaktuar nëse një program ka kaluar me sukses apo ka dështuar pas ekzekutimit të një testi. Oracle përbëhet nga tre pjesë: gjeneruesi, krahasuesi dhe vlerësuesi.

Rast Testimi: Bashkësia e input-ve të nevojshme për të kryer një ekzekutim të software-it nën testim.

Set-i i Rasteve të Testimit: Bashkësia e gjithë rasteve të testimit për testimin e një software-i.

Mutant: Programi që rezulton pas aplikimit të një operatori mutacioni tek programi original.

Kriter Mbulimi: Një kriter mbulimi është një rregull ose një bashkësi rregullash që imponon kërkesat e testimit mbi një bashkësi testesh.

Mbulim: Nëse kemi një bashkësi kërkesash për një test *KT* për një kriter mbulimi *K*, atëherë një bashkësi testesh *T*, përmbush *K*, vetëm në qoftë se për çdo kërkesë testimi *kt* në *KT*, ekziston të paktën një test *t* në *T* që përmbush *kt*.

Framework: një program pjesërisht i përfunduar. Ky program ofron një strukturë të ripërdorëshme që mund të përdoret nga aplikacione të tjera, të cilat duke e zgjeruar frameëork-un përmbushin kërkesat e tyre specifike.

Testimi i Njësive: testimi i njësive të veçanta hardware ose software ose testimi i grupeve të njësive që kanë lidhje midis tyre.

Teknikë Testimi: përcakton se cilat metoda/mënyra do të aplikohen ose cilat llogaritje do të kryhen për të testuar një tipar të veçantë të një software.

Detektim: vrojtimi se sjellja e programit është e ndryshme nga sjellja që pritej të kishte.

Instrumentimi i kodit: teknikë gjatë të cilës shtohen shprehje tek source code ose tek byte code i programit që është nën testim për të matur mbulimin e arritur gjatë testimit.

KAPITULLI 1

Hyrje

Gjatë viteve të fundit ka patur një rritje në interesin e studiuesve për Testimin e Software-ve të Bazuar në Kërkim (TSBK) dhe veçanërisht në teknikat për gjenerimin e rasteve të testimit që përmbushin një kriter strukturor mbulimi, si për shembull mbulimin e shprehjeve ose të degëve. Fusha e inxhinierisë së software-it që bazohet në kërkim, riformulon problemet e inxhinierisë së software-it, si probleme optimizimi dhe përdor algoritmat meta-heuristikë për zgjidhjen e këtyre problemeve. TSBK ishte problemi i parë i inxhinierisë së software-it që u tentua të zgjidhej duke përdorur optimizimin [1][2] dhe akoma sot, mbetet fusha më atraktive e studimit në komunitetin e inxhiniersisë së software-it që bazohet në kërkim.

Testimi i software-ve mund të shikohet si një sekuencë e tre hapave kryesore:

1. Ndërtimi i rasteve të testimit që janë efektive në zbulimin e gabimeve, ose që janë të përshtatshme bazuar në një kriter përshtatshmërie testimi.
2. Ekzekutimi i rasteve të testimit.
3. Përcaktimi nëse rezultati pas ekzekutimit është i saktë apo jo.

Fatkeqësisht, për sa i përket hapit të tretë, mund të themi se deri më sot nuk është arritur të automatizohet, pasi gjenerimi automatik i “oracle” nuk është ende i mundur. Është e qartë se hapi i dytë mund të kryhet plotësisht në mënyrë automatike. Për sa i përket hapit të pare, ka patur mjaft studime për automatizimin e tij pjesërisht ose plotësisht, megjithatë rezultatet e përftuara varen nga një sërë faktorësh dhe ende nuk ka një vlerësim të saktë mbi efektshmërinë e tyre [3].

Automatizimi i hapit të parë ka një impakt të konsiderueshëm në testim, pasi gjenerimi i rasteve të testimit është një detyrë e vështirë dhe që kërkon shumë kohë.

Panvarësisht studimeve të shumta, ekzistojnë ende probleme që kërkojnë zgjidhje para se automatizimi i plotë i kësaj faze të kryhet praktikisht. Ky punim, ka si qëllim të sjellë një përfitim në aspektin e vlerësimit të rasteve të testimit të gjeneruara gjatë kërkimit dhe do të fokusohet kryesisht në kriterin e mbulimit të degëve. Ky kriter është shumë i përdorur edhe kërkohet të përmbushet nga standartet e testimit për shumë aplikacione kritike të sigurisë [4].

Ekzistojnë dy fusha të ndryshme studimi për gjenerimin automatik të rasteve të testimit: ekzekutimi simbolik dhe testimi i bazuar në kërkim.

Ky punim fokusohet në testimin e bazuar në kërkim dhe konkretisht punimi konsiston në implementimin e një funksioni të ri vlerësimi për drejtimin e kërkimit, në mjetin eToc [1], i cili përdor një vlerësim të bazuar tek degët e mbuluara. Më gjerësisht problematika që adreson ky studim dhe kontributet e tij do të trajtohen në paragrafin pasardhës.

1.1 Motivimi dhe Kontributet e Studimit

Testimi automatik në nivel njësie është një proçes i vështirë, mjaft i gjerë dhe kompleks. Ende ka shumë sfida të hapura që kërkojnë zgjidhje në mënyrë që ky testim të jetë sa më efiçent. Sipas një studimi nga “National Institute of Standart & Technology” [5], rreth 80% e burimeve të zhvillimit të një software i dedikohen identifikimit dhe korrigjimit të gabimeve. Gjithashtu, sipas të njëjtit studim, kostoja e gabimeve të software-ve në SHBA është 59.5 bilion dollar në vit dhe 1/3 e kësaj vlerë i atribuohet infrastrukturës së dobët të testimit. Për përmirësimin e kësaj infrastrukture, janë zhvilluar disa mjete për testimin automatik. Meqë testimi është një veprimtari krijuese, studimi i mënyrave të përdorura për ta automatizuar këtë veprimtari paraqet një interes të veçantë. Në ditët e sotme ekzistojnë disa metoda si kërkimi metaheuristik, gjenerimi i testeve në mënyrë të rastësishme, analizimi statik, etj, të cilat përdoren për gjenerimin automatik, por aplikimi i këtyre metodave në software-t realë është akoma i limituar. Studime të ndryshme kanë paraqitur një shumëllojshmëri kriteresh mbulimi që duhet të përmbushen nga rastet e testimit të gjeneruara për të arritur një cilësi të lartë testimi. Megjithatë, shumë kritere janë

të vështira për t'u përmbushur, ose edhe të panevojshme në disa raste. Për këtë arsye përcaktimi i cilësisë së rasteve të gjeneruara është i vështirë. Gjithashtu edhe subjektet e zgjedhura për të kryer eksperimentet për vlerësimin e teknikave të reja, ndikojnë në rezultatet e përftuara, pasi për programe të ndryshme rezultatet e arritura janë të ndryshme.

Kontributet kryesore të këtij punimi janë:

- Studim i literaturës për evidentimin e problemeve me të cilat përballet sot testimi i bazuar në kërkim dhe përmbledhja e disa prej mënyrave që janë propozuar dhe implementuar për zgjidhjen e tyre.
- Studimi i kritereve të mbulimit që përdoren në ditët e sotme dhe vlerësimi i tyre individual dhe i kombinuar duke u bazuar tek rezultati i mutacionit të arritur. Vlerësimi është kryer duke përdorur si subjekte projekte reale dhe me kod të hapur.
- Propozimi i një funksioni të ri vlerësimi, për algoritmin gjenetik që përdoret për gjenerimin automatik të rasteve të testimit për klasat në Java. Ky funksion krahas nivelit të afërsisë dhe distancës së degëve merr në konsideratë edhe numrin e gjendjeve të reja të arritura nga objekti i klasës që është nën testim.
- Implementimi i funksionit të propozuar të vlerësimit.
- Studimi eksperimental i efekteve të përdorimit të funksionit të propozuar të vlerësimit mbi mbulimin e degëve, rezultatin e mutacionit të arritur dhe gjatësinë e rasteve të testimit. Eksperimentet janë kryer mbi klasa të marra nga projekte reale.
- Diskutim i përgjithshëm mbi rezultatet e arritura dhe renditja e çështjeve që mbeten për t'u zgjidhur në të ardhmen.

1.2 Struktura e Dizertacionit

Struktura e organizimit të këtij punimi është si në vijim:

- Kapitulli i dytë bën një përmbledhje të koncepteve shkencore që përbëjnë bazën e studimit tonë. Në këtë kapitull, bazuar tek literatura e publikuar në katër vitet e fundit, do të paraqitet fokusi i studimeve mbi fushën e testimit. Gjithashtu kapitulli shpjegon dy teknikat kryesore pët testimin automatik: ekzekutimin simbolik dhe testimin e bazuar në kërkim. Më pas bëhet një përmbledhje e problematikave me të

cilat përballet sot testimi i bazuar në kërkim dhe listohen disa prej zgjidhjeve të propozuara në literaturë. Në këtë kapitull, përshkruhen shkurtimisht tipet kryesore të teknikave meta-heuristike dhe kryesisht algoritmat gjenetikë, për të cilat shpjegohen gjithë komponentët përbërës të tyre. Gjithashtu bëhet një përmbledhje e funksioneve të vlerësimit të përdorura për algoritmat gjenetikë.

- Kapitulli i tretë paraqet një përmbledhje të kritereve më të rëndësishme të mbulimit për testimin strukuror. Gjithashtu në këtë kapitull jepen rezultatet e eksperimenteve të kryera për të vlerësuar efiçensën e përdorimit të disa kritereve të kombinuara së bashku. Pjesa e mbetur e kapitullit shpjegon disa prej karakteristikave të Java-s dhe si përfundim përshkruhen cilat duhet të jenë përshtatjet që i duhen bërë algoritmave gjenetikë (gjithë komponentëve të tyre) që të mund të aplikohen në nje kod në Java.
- Kapitulli i katërt prezanton një funksion të ri vlerësimi për algoritmat gjenetikë që përdoren për testimin në nivel njësie të programeve në Java (ky funksion mund të shërbejë edhe për gjuhë të tjera të orientuara nga objekti). Gjithashtu në këtë kapitull përshkruhen në mënyrë të detajuar gjithë ndryshimet që duhet të kryhen në një mjet testimi automatik (mjeti eToc), në mënyrë që të mund të implementohet funksioni i propozuar i vlerësimit.
- Kapitulli i pestë ka si qëllim të testojë rezultatet e funksionit të ri të vlerësimit të përshkruar në kapitullin katër. Tek ky kapitull do të paraqiten pyetjet për kërkim dhe për secilën prej tyre do të shpjegohet organizimi i eksperimenteve të kryera dhe do të paraqiten dhe diskutohen rezultatet e përftuara prej tyre.
- Kapitulli i gjashtë përmbyll këtë tezë disertacioni duke bërë një përmbledhje të konkluzioneve të arritura dhe duke përmendur disa prej drejtimeve të punës në të ardhmen që nisin nga ky punim.

KAPITULLI 2

Përmbledhje e Literaturës

Në këtë kapitull bëhet një përmbledhje e fushës së testimit në nivel njësie. Kapitulli shpjegon testimin automatik dhe përkatësisht teknikat për ekzekutim simbolik dhe teknikat që bazohen në kërkim. Gjithashtu këtu do të përshkruhen shkurt tipet kryesore të teknikave meta-heuristike dhe fokusi kryesor do të jetë mbi algoritmat gjenetikë dhe sfidat me të cilat përballet sot përdorimi i tyre si algoritma kërkimi në mjetet automatike për testim strukturor.

2.1 Testimi

Testimi është një disiplinë që ndihmon gjithë profesionistët e fushave të teknologjisë së informacionit të zhvillojnë software të cilësisë së lartë [6].

Testimi i software-ve është proçesi i verifikimit (kontrolli ose testimi i njësive, për konsistencë duke vlerësuar rezultatet kundrejt kërkesave të para-përcaktuara: *Po e ndërtojmë si duhet sistemin?*) dhe i vlerësimit (kontrollon korrektësinë e sistemit – p.sh. proçesi i kontrollit që çfarë është specifikuar është ajo që kërkon përdoruesi: *Po ndërtojmë sistemin e duhur?*)

Testimi është një proçes shumë i rëndësishëm edhe i ndërlikuar që shoqëron gjithë proçesin e ndërtimit të software-it dhe vazhdon edhe pasi ky proçes ka përfunduar. Është llogaritur që 60-80% e burimeve i dedikohen testimit. Testimi nuk shihet më si një aktivitet që fillon vetëm pasi ka mbaruar faza e kodimit. Në ditët e sotme dihet se është shumë më e kushtueshme të korrigjohen gabime që detektohen në etapat e fundit të zhvillimit të software-it. Edhe pse rëndësia e testimit është e padiskutueshme, akoma sot, ky proçes zhvillohet sipas rastit, është i kushtueshëm dhe efektiviteti i tij është i papërcaktuar [7].

Duke marrë parasysh përvojat e kaluara, testimi i pamjaftueshëm dhe i paefektshëm mund të rezultojë në probleme sociale dhe humbje financiare/njerëzore. Për këtë arsye, fakti që software-t duhet të testohen nuk ka nevojë për diskutim të mëtejshëm.

Zgjidhja e përkryer (që jep siguri 100%) do të ishte testimi i të gjitha rasteve të mundshme, por kjo gjë është praktikisht e pamundur. Kjo është arsyeja se pse testimi mund të tregojë praninë e defekteve në një sistem; por nuk mund të provojë asnjëherë se nuk ka gabime të mbetura në të. Testimi është gjithmonë një balancë mes kostos së testimit të mëtejshëm dhe kostos së mundshme të gabimeve të pazbuluara në një program. Asnjëherë nuk mund të jemi të sigurtë për cilësinë e një software. Gabime komplekse ekzistojnë edhe në sisteme kritike dhe që kanë vite që përdoren. Qëllimi kryesor i testimit është të ngjallë besueshmëri tek software.

Ekzistojnë një shumëllojshmëri teknikash të testimit dhe ka gjasa që shumë teknika të tjera do të zhvillohen në të ardhmen. Pra, në kushtet aktuale detyra kryesore është të bëhet një zgjedhje midis teknikave të disponueshme të testimit për të gjetur defektet maksimale/gabimet në një kontekst të dhënë. Rëndësia e zgjedhjes së teknikave të përshtatshme për testim sot pranohet gjerësisht në komunitetin e testuesve të software-ve. Shumë studime janë fokusuar në rëndësinë e informacionit mbi teknikat që janë zbuluar, në mëyrë që testuesi ta ketë të lehtë të përzgjedhë teknikën ose grupin e teknikave që do të përdorë. Efektiviteti i ketyre teknikave në shumë raste është i paqartë pasi rezultatet e paraqitura në artikujt korrespondues varen nga software-t e zgjedhur për testim, gjuha e programimit, tipet e gabimeve, subjekti që e kryen testimin etj [8][9][10]. Në figurën 2.1 jepet klasifikimi i teknikave të testimit të software-ve bazuar tek [8].

<table>
<tr><td rowspan="10">Kërkohet ekzekutimi i software-it që është nën testim?</td><td rowspan="2">JO</td><td rowspan="2">Testimi Statik</td><td rowspan="2">Kërkon teknika ndonjë mjet automatik?</td><td>JO</td><td colspan="5">Manuale</td></tr>
<tr><td>PO</td><td colspan="5">Automatike</td></tr>
<tr><td rowspan="8">PO</td><td rowspan="8">Testimi Dinamik</td><td rowspan="8">Kërkohet ekzaminimi i kodit?</td><td rowspan="4">JO</td><td rowspan="4">Testimi "Black Box"</td><td rowspan="8">Kriteret për përzgjedhjen e të dhenave të testimit</td><td>Në mënyrë "random"</td><td colspan="4">Testimi Random</td></tr>
<tr><td>Në përdorimin e pritur të software-it</td><td colspan="4">Testimi Statistikor</td></tr>
<tr><td>Në specifikimet funksionale</td><td colspan="4">Testimi Funksional</td></tr>
<tr><td rowspan="2">Në specifikimet strukturore</td><td rowspan="2">Testimi Strukturor</td><td rowspan="2">Cfarë kontrollohet?</td><td>Të dhënat</td><td>Testimi Data Flow</td></tr>
<tr><td rowspan="4">PO</td><td rowspan="4">Testimi "White Box"</td><td>Kontrolli</td><td>Testimi Control Flow</td></tr>
<tr><td>Bazuar tek mutantët e shpëtuar</td><td colspan="4">Testimi Mutation</td></tr>
</table>

Figura 2.1: Klasifikimi i teknikave të testimit të software-ve [8]

2.1.1 Klasifikimi i Teknikave të Testimit

Teknikat e testimit klasifikohen duke u bazuar tek:

1. Niveli i testimit

   Bazuar tek [11] ekzistojnë pesë nivele testimi:

   - Testim në Nivel Njësie (Unit Testing)
   - Testimi i Bashkimit të Njësive (Integration Testing)
   - Testimi i Sistemit (System Testing)
   - Tesimi i Pranimit (Acceptance Testing)
   - Testimi në Prapavajtje (Regression Testing)

   Përqindjet e kërkimit për testimin në këto pesë nivele, në katër vitet e fundit, jepen në figurën 2.2 [12]. Rezultatet janë përftuar pasi kemi kryer studimin e artikujve të marra nga pesë konferencat më të njohura në fushën e testimit: International Conference in Software Engineering (ICSE), International Conference on Fundamental Approaches to Software Engineering (FASE), International Symposium on Software Testing and Analysis (ISSTA), International Conference on Automated Software Engineering (ASE), European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE).

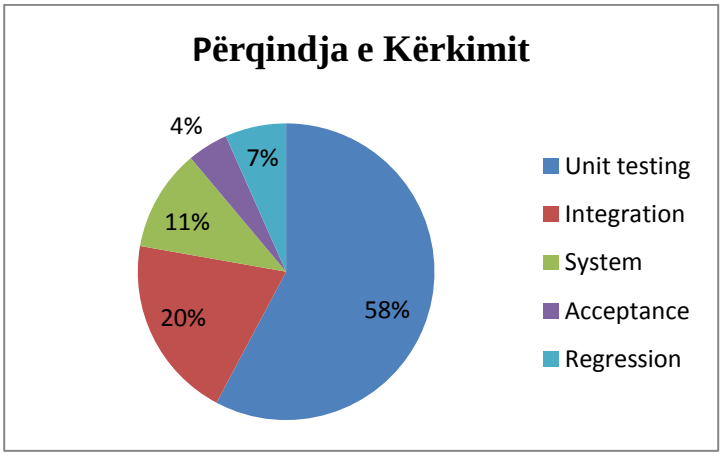

Figura 2.2: Përqindja e kërkimit bazuar në nivelet e testimit

Duket qartë se më shumë se 50% e kërkimit në kohët e fundit është fokusuar në testimin në nivel njësie, ndërsa më shumë se 50% e pjesës së mbetur fokusohet në testimin e bashkimit të njësive. Në të dy këto nivele kodi është i dukshëm për testuesit, prandaj arrijmë në konkluzionin se edhe pse ekzistojnë shumë teknika për

testimin e kodit akoma ekziston nevoja për përmirësimin e tyre. Shumica e kërkimit në këto nivele është drejt zhvillimit të mjeteve për gjenerimin automatik të rasteve të testimit dhe oracles në gjuhë të ndryshme dhe që përmbushim kritere të ndryshme mbulimi.

2. Strategjia për përzgjedhjen e të dhënave për testim

Ka dy kategori për ndarjen e praktikave të testimit: Testimi Statik dhe Testimi Dinamik [13]. Gjatë testimit dinamik zgjidhen input-et, ekzekutohet software mbi këti input-e dhe krahasohen rezultatet me oracle për të përcaktuar nëse rezultatet janë apo jo të sakta. Ekzistojnë pesë strategji për përzgjedhjen e të dhënave në testimin dinamik:

- Testim i Rastësishëm (Random Testing)
- Testim Statistikor (Statistical Testing)
- Testim Funksional (Functional Testing)
- Testim Strukturor (Structural Testing)
- Testim i Mutacioneve (Mutation Testing)

Përqindjet e kërkimit për testimin bazuar në strategjinë për përzgjedhjen e të dhënave, në katër vitet e fundit, jepen në figurën 2.3 [12].

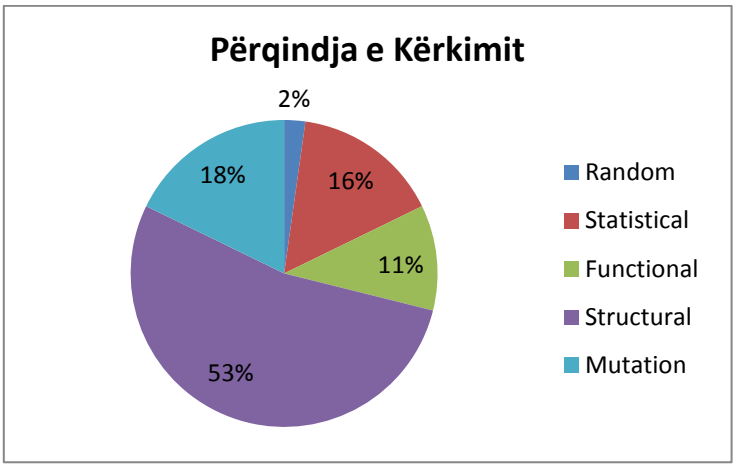

Figura 2.3: Përqindja e kërkimit bazuar në strategjinë për përzgjedhjen e të dhënave

Nga grafiku vëmë re se në katër vitet e fundit, afërsisht 71% e kërkimit mbi testimin e software-ve fokusohet në testimin strukturor dhe testimin e mutacioneve (testimin white-box). Disavantazhi kryesor i testimit strukturor është se ky lloj testimi nuk mund të gjejë

gabime të tipit të mos-implementimit të një ose disa kërkesave funksionale. Meqe pika e fillimit për zhvillimin e rasteve të testimit strukturor është vetë kodi, atëherë nuk ka asnjë mundësi që të gjenden kërkesa (të nivelit të lartë ose të ulët) që nuk janë implementuar tek kodi. Bazuar tek [14], testimi i mutacioneve, përfshin shumë kritere të tjera mbulimi. Sot ky kriter cilësohet si kriteri më i mirë për vlerësimin e cilësisë të setit të rasteve të testimit. Por testimi i mutacioneve është i vështirë për tu aplikuar realisht dhe kërkon të kryhen shumë veprime llogaritëse. Prandaj fokusi është të gjenden mënyra se si mund të lehtësohet proçesi i testimit të mutacioneve ose të përdoren kritere të tjera ekzistuese (ose një kombinim i disa kritereve) për vlerësimin e setit të rasteve të testimit.

2.2 Testimi Automatik

Automatizimi i testimit është detyra e krijimit të një paraqitjeje të interpretueshme në mënyrë automatike të një rasti testimi manual. Kjo paraqitje mund të jetë një program në një gjuhë të caktuar programimi. Testet manuale nuk e kanë avantazhin e ekzekutimit efiçient dhe të përsëritur si testet automatike [15].

Testimi automatik është i rëndësishëm në gjenerimin e rasteve të testimit për software të mëdhenj. Gjenerimi manual i rasteve të testimit sot është praktikisht i pamundur për software –t reale për arsye se këto sisteme janë shumë komplekse dhe përmbajnë deri në miliona rreshta kod. Përdorimi i mjeteve automatike për të përmbushur qëllimet e testimit çon në reduktimin e kohës dhe të punës së testuesve, megjithatë ka akoma shumë sfida që duhet të kalohen në mënyrë që testimi automatik të mund të kryhet për çdo software dhe të ketë nevoja minimale për mbikqyrjen e testuesve [16].

Studiuesit kanë propozuar disa mënyra për gjenerimin e testeve në mnëyrë automatike. Këto mënyra mund të ndryshojnë për disa arsye: karaketeristika e sistemit që po testohet (p.sh. funksionale ose jo-funksionale), shkalla e sistemit (p.sh. teste njësie, teste integrimi, teste sistemi, teste pranimi), tipi i sistemit (p.sh. funksione, klasa), tipi dhe sasia e informacionit që po përdoret (p.sh. black-box, grey-box, white-box) ose teknika që po përdoret për gjenerimin e testeve (p.sh. ekzekutimi simbolik ose mënyra e bazuar në kërkim). Edhe pse domosdoshmëria për përdorimin e automatizimit gjatë testimit

është e qartë, ende nuk ka një studim të plotë mbi masën se sa automatizimi ndihmon në të vërtetë gjatë testimit të software-ve reale në ditët e sotme.

Gjatë testimit kryhen dy proçese kryesore: përzgjedhja e rasteve të testimit që do të përdoren midis hapësirës së pafundme të rasteve të testimit dhe përcaktimi i oracle (rezultatet që duhet të merren pas ekzekutimit të cdo rasti testimi të përzgjedhur). Automatizimi i procesit të dytë sot është akoma shumë larg zgjidhjes. Pavarësisht përpjekjeve të ndryshme nuk ka ndonjë mjet që të bëjë vlerësimin e rezultateve që merren pas ekzekutimit. Kjo detyrë duhet te kryhet manualisht. Për sa i përket proçesit të parë janë zhvilluar shumë mjete të cilat gjenerojnë në mënyrë automatike rastet e testimit. Mjetet klasifikohen sipas nivelit në të cilin veprojnë, strategjisë që përdorin për përzgjedhjen e rasteve të testimit, gjuhës së kodit që ato mund të testojnë.

Për vlerësimin e mjeteve të cilat bazohen në strukturën e kodit, ekzistojnë disa kritere mbulimi mbi të cilat ngrihet edhe funksionaliteti i këtyre mjeteve. Kriteri më i përdorur në literaturë është mbulimi i degëve [17].

Mënyra më e thjeshtë për të kryer testim automatik është testimi i rastësishëm, i cili ka treguar që është efektiv [18]. Megjithatë sot ka dy drejtime kryesore më të sofistikuara [19] për të kryer gjenerim automatik të rasteve të testimit në testimin strukturor: Ekzekutimi Simbolik (ES) [16] dhe Testimi Bazuar në Kërkim (TSBK)[21]. Bazuar në rezultatet e paraqitura në literaturë, mjetet e zhvilluara të cilat bazohen në këto dy drejtime, përmbushin kritere të ndryshme mbulimi. Edhe pse këto dy teknika kanë baza teorike krejtësisht të ndryshme dhe kanë disa dekada që janë prezantuar, akoma në ditët e sotme, në fushën e kërkimit, ka shumë interes për secilën prej tyre. Gjithashtu në tre vitet e fundit ekzistojnë edhe përpjekje që duken mjaft premtuese për të ardhmen [21][22][23][24], për të zhvilluar mjete që kombinojnë këto dy teknika të cilat janë ortogonale teorikisht. Edhe pse ka pak kohë që punohet në këtë drejtim, janë paraqitur disa mënyra të ndryshme të kombinimit të këtyre teknikave.

2.2.1 Ekzekutimi Simbolik

Ekzekutimi Simbolik (ES) është një teknikë për verifikimin e programeve. Kjo teknikë është propozuar fillimisht në vitet 70. Qëllimi kryesor i kësaj teknike është të eksplorojë rrugët e mundëshme të ekzekutimit të një aplikacioni dhe për këtë arsye kjo teknikë konsiderohet shumë efektive në gjenerimin automatik të set-ve të rasteve të testimit që kanë mbulim të gjerë. Edhe pse ka më shumë se tre dekada që është dhënë ideja e kësaj teknike, vetëm në vitet e fundit kjo teknikë ka fituar vëmendje të madhe nga ana e studiuesve në fushën e testimit. Kjo vëmendje ka ardhur si pasojë e faktit se kohët e fundit ekzekutimi simbolik mund të implementohet më lehtësisht si rrejedhojë e rritjes së fuqisë përpunese të kompjuterave dhe e ppërmirësimit të bërë në zgjidhësit e detyrimeve (constraints solvers). Sot kjo teknikë mund të implementohet praktikisht për testimin e software-ve reale që kanë deri në miliona rreshta kod. Bazuar tek [25] përdorimi i ES në praktikë është akoma i limituar, por bazuar tek progresi i bërë nga kërkimi në këtë fushë mendohet se impakti i ES në testimin automatik për software reale do të vazhdojë të rritet.

Ekzistojnë disa mjete për gjenerim automatik te rasteve të testimt të cilat bazohen në ES. Një mjet i tillë përbëhet nga dy komponentë: gjeneruesi i detyrimeve i cili ndërton detyrimet dhe zgjidhësi i detyrimeve që i zgjidh ato. Në ES vlerat e inputeve janë vlera simbolike dhe jo reale. Çdo simbol paraqet gjithë brezin e vlerave që një variabël hyrës mund të marrë. Output-i i kodit paraqitet si funksion i këtyre vlerave. Për ekzekutimin e një kodi në mënyrë simbolike gjithë varaiablat hyrëse inicializohen me simbole. ES mban një gjendje simbolike që përkthen variablat tek shprehjet simbolike. Zgjidhësi i detyrimeve përcakton nëse një rrugë është e pamundur; kjo do të thotë që nuk ka inpute që të bëjnë që programi të ndjekë këtë rrugë. Një shembull për ilustrim paraqitet në figurën 2.4.

Kur përdoret në aplikacionet reale, ekzekutimi simbolik klasik ka disa probleme (p.sh. kur kodi përmban cikle ose rekursivitet). Mjetet e zhvilluara së fundmi bashkojnë ekzekutimin simbolik me ekzekutimin konkret. Ky bashkim quhet Ekzekutimi Simbolik Dinamik (ESD). ESD është një nga arsyet pse ekzekutimi simbolik modern mund të arrijë të përdoret në testimin e software-ve reale.

Bazuar tek [26] ekzistojnë dy menyra për kryerjen e ESD:

1. Testimi Concolic
2. Testimi Execution-Generated

Vështirësitë kryesore të ES janë numri i pafundëm i path-ve (path explosion) [27][28][29][30] dhe zgjidhësit e dtyrimeve (constraint solvers) [31][32][33][34].

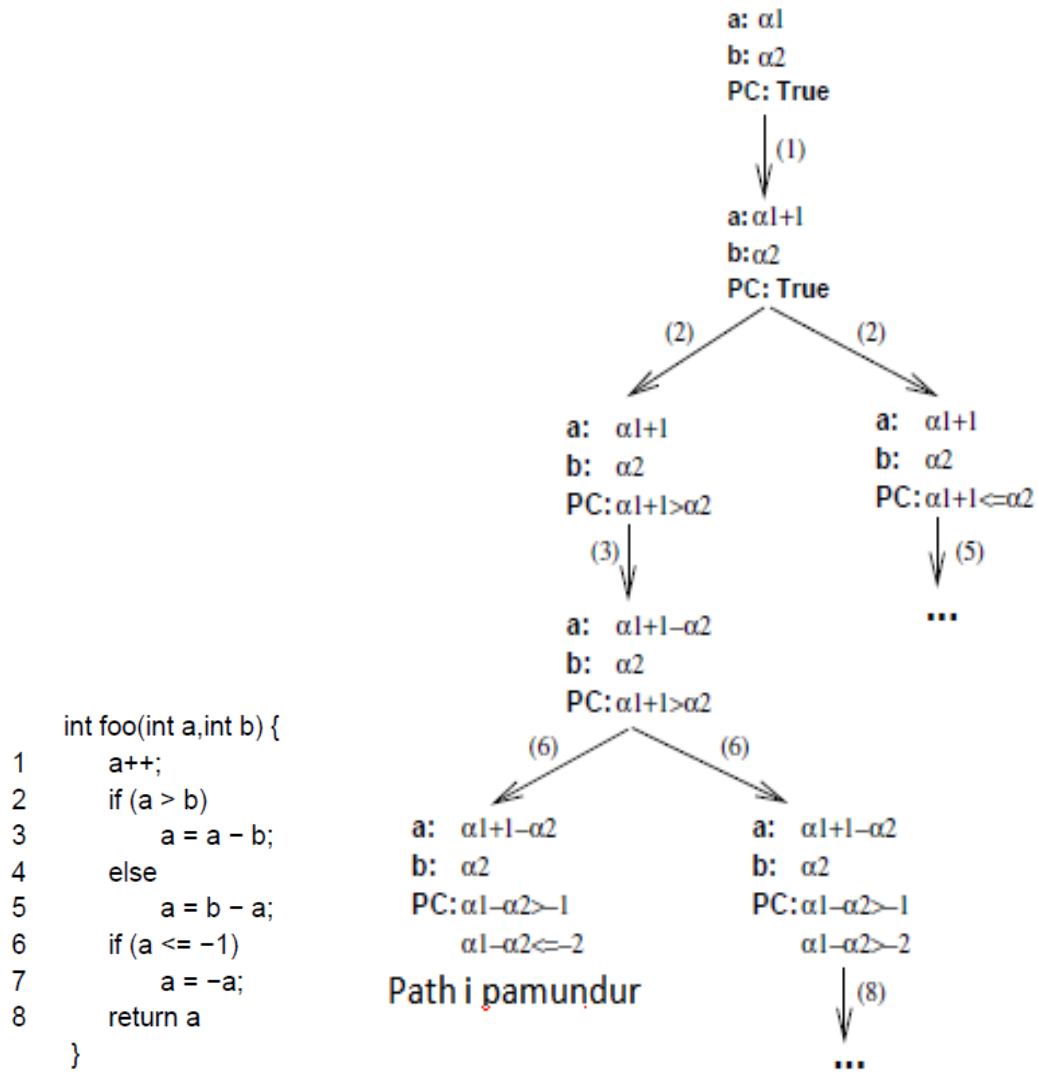

Figura 2.4: Shembulli i një metode (metoda `foo`), simbolet e përdorura, detyrimet e gjeneruara gjate ES

2.2.2 Testimi i Bazuar në Kërkim

Në rastin e testimit të bazuar në kërkim, problemi i kërkimit përkufizohet si më poshtë:

Të gjendet një vlerë x* që maksimizon (minimizon) funksionin objektiv (të vlerësimit) f, gjatë hapësirës së kërkimit X (figura 2.5):

$$f: X \rightarrow R$$

$$x^*: \forall x \in X, f(x^*) \geq f(x)$$

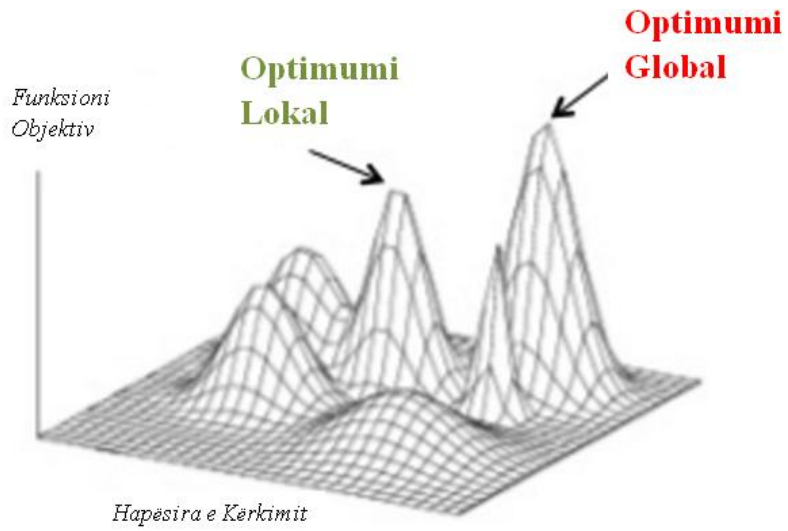

Figura 2.5: Pamja e funksionit objektiv në hapësirën e kërkimit

Algoritmat e kërkimit mund të grupohen në tre grupe:

1. **Kërkim Shterues**: kërkohen gjithë zgjidhjet e mundëshme
2. **Kërkim i Rastësishëm**: merren në mënyrë të rastësishme zgjidhje nga hapësira e kërkimit
3. **Kërkim Metaheuristik**: do të shpjegohet në paragrafin në vijim.

Teknikat metaheuristike të kërkimit janë struktura të nivelit të lartë, të cilat përdorin heuristikën, me qëllim gjetjen e zgjidhjeve për probleme kombinatore, me një kosto llogaritëse të arsyeshme. Përdorimi i këtyre teknikave në fushën e gjenerimit të rasteve të testimit është një mundësi shumë premtuese. Këto, në vetvete nuk janë algoritma të pavarur, por janë strategji që mund të përshtaten për probleme specifike. Në rastin e

gjenerimit të rasteve të testimit, kriteri i testimit transformohet në një *funksion objektiv* për kërkimin. Këto objektiva krahasojnë zgjidhjet e përftuara gjatë kërkimit në raport me qëllimin e përgjithshëm të kërkimit. Duke përdorur këtë informacion, kërkimi drejtohet drejt zonave më premtuese të hapësirës së kërkimit. Gjenerimi i rasteve të testimit bazuar në kërkim është vetëm një shembull i inxhinierisë së software-it që bazohet në kërkim.

Një algoritëm metaheuristik kur implementohet për zgjidhjen e një problemi specifik ka një buxhet kërkimi të fundëm. Si pasojë e këtij limiti këto algoritma kanë si qëllim të arrijnë një balancë midis (figura 2.6):

- Kërkimit lokal: shfrytëzim (intensifikim)
- Kërkimit global: eksplorim (diversifikim)

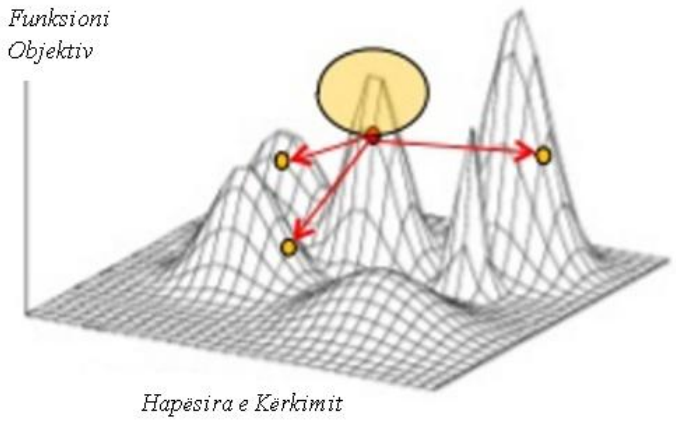

Figura 2.6: Kërkimi lokal kundrejt kërkimit global

Përdorimi i teknikave metaheuristike të kërkimit për një problem specifik, kërkon marrjen e disa vendimeve. P.sh. mënyra se si do të enkodohen zgjidhjet në mënyrë që të manipulohen gjatë kërkimit. Një mënyrë e mirë enkodimi do të siguronte që zgjidhje të ngjashme në hapësirën e pa enkoduar, të jenë gjithashtu "fqinjë" në hapësirën e enkoduar. Në këtë mënyrë kërkimi do ta kishte të mundur të kalonte nga një zgjidhje në një zgjidhje tjetër me disa karakteristika të përbashkëta. Bazuar në feedback-un e përftuar nga funksioni objektiv, kërkimi synon të gjejë zgjidhje më të mira. Për rrjedhojë, një funksion objektiv i mirë është i domosdoshëm që kërkimi të kryhet me sukses. Zgjidhjet që janë më të mira duhet të kenë një vlerë të mirë (të lartë ose të ulët; varet nëse po kërkohet të minimizohet apo të maksimizohet funksioni objektiv) të funksionit objektiv.

2.3 Teknikat Meta-Heuristike

Paraqgrafet në vijim shpjegojnë disa prej teknikave metaheuristike kryesore që janë përdorur për gjenerimin e rasteve të testimit.

2.3.1 Hill Climbing

Hill Climbing është një algoritëm kërkimi lokal shumë i njohur. Ky algoritëm klasifikohet edhe si kërkim në fqinjësi. Në algoritmat e kërkimit, termi fqinjësi i referohet një bashkësie individësh që kanë të përbashkëta disa karakteristika. Për shembull, variabli i plotë (integer) x më vlerën 5, mund të ketë si fqinjë dy vlerat e plota {4, 6}.

Tek Hill Climbing zgjidhet një zgjidhje fillestare në mënyrë të rastësishme si pika e fillimit të kërkimit. Më pas fqinjët e kësaj zgjidhjeje shqyrtohen dhe nëse gjendet një zgjidhje më e mirë, zgjidhja aktuale zëvendësohet me zgjidhjen e re. Fqinjët e zgjidhjes së re shqyrtohen dhe nëse gjendet një zgjidhje më e mirë kjo zgjidhje zëvendësohet sërisht. Kjo proçedurë vazhdon deri sa të mos ketë mundësi për të gjetur zgjidhje më të mira tek fqinjët. Në këtë pikë kërkimi ose ka arritur në një zgjidhje optimale lokale ose ka gjetur zgjidhjen optimale globale.

Ka disa mënyra të implementimit të këtij algoritmi: *ngjitje e thjeshtë, ngjitje me shkallë, ngjitje stokastike*. Në strategjinë me ngjitje të thjeshtë, kërkimi zhvendoset në fqinjin e parë që është më i mirë se individi aktual i zgjedhur. Në ngjitjen me shkallë, shqyrohen gjithë fqinjët dhe më pas zgjidhet individi më i mirë nga fqinjët për të zhvendosur kërkimin. Ngjitja stokastike qëndron midis dy strategjive të sipër përmendura; individi i ri zgjidhet në mënyrë të rastësishme midis gjithë kandidatëve që kanë marrë vlerësimin më të lartë pas shqyrtimit. Një përshkrim në nivel të lartë i këtij algoritmi jepet në figurën 2.7.

Hill climbing është algoritëm i thjeshtë dhe jep rezultate të shpejta. Megjithatë ndodh shpesh që kërkimi të gjenerojë një zgjidhje që është optimale lokalisht por jo globalisht. Në këto raste kërkimi mbetet i bllokuar në “majë të kodrës”, pa mundur të eksplorojë zona të tjera të hapësirës së kërkimit. Gjithashtu kërkimi mund të ngecë përgjatë

"plateaux" në hapësirën e kërkimit, ku në këto rrethana, asnjë fqinj nuk mund të zgjidhet për të ofruar një zgjidhje më të mirë, se të gjithë fqinjët kanë të njëjtin vlerësim. Për rrjedhojë në disa raste, rezultatet e marra nga ky algoritëm varen drejtpërdrejtë nga zgjidhja fillestare. Një shtesë që i është bërë këtij algoritmi është që të përfshihen disa "rifillime" të kërkimit ne mënyrë që të minimizohen këto probleme (rifillohet kërkimi kur nuk po gjendet ndonjë zgjidhje më e mirë ose përmirësimi i përftuar është shumë i vogël).

```
Përzgjidh një zgjidhje fillestare s Є S
Përsërit
      Përzgjidh s'Є N(s), e tillë që obj(s')>obj(s)
      s ← s'
Deri sa obj(s)>= obj(s'), çdo s'Є N
```

Figura 2.7: Përshkrim në nivel të lartë i algoritmit hill climbing, për një problem me hapësirë zgjidhjeje S, strukturë fqinjësh N dhe me funksion objektiv obj që duhet maksimizuar [35]

2.3.2 Simulated Annealing

Simulated Annealing është një teknikë e ngjashme në parim me hill climbing, por është më pak e pavarur nga zgjidhja e zgjedhur në fillim të kërkimit. Kjo teknikë, në mënyrë probabilitare, pranon edhe zgjidhje që janë më të këqija se zgjidhja aktuale; në këtë mënyrë lëvizja në hapësirën e kërkimit bëhët më pak e kufizuar. Probabiliteti për të pranuar zgjidhje më inferiore ndryshon gjatë progresit të kërkimit dhe llogaritet me formulën:

$$p = e^{-\frac{\delta}{t}}$$

ku δ është diferenca midis vlerësimit të zgjidhjes aktuale dhe zgjidhjes fqinje më pak të vlerësuar që po merret në konsideratë, t është një parametër kontrolli që njihet edhe si temperatura [36]. Temperatura ftohet sipas një skedulimi të përcaktuar. Fillimisht temperatura është e lartë, në mënyrë që të lejojë lëvizje të lirë përgjatë hapësirës së kërkimit duke kufizuar kështu varësinë nga zgjidhja fillestare. Gjatë përparimit të kërkimit, temperatura ulet. Megjithatë nëse ftohja është shumë e shpejtë, atëherë nuk do

të kemi shumë eksplorim dhe mundësitë që kërkimi të ngecë në optimumin lokal rriten. Algoritmi bazë (në rastin e minimizimit të funksionit objektiv) jepet në figurën 2.8.

Emri "Simulated Annealing" ka ardhur nga analogjia që ekziston midis kësaj teknike dhe proçesit kimik te "annealing" – ftohja e materialit në një vaskë të ngrohur; karakteristikat strukturore të materialit të ftohur varen nga shkalla e ftohjes.

```
Përzgjidh një zgjidhje fillestare s Є S
Përzgjidh një temperaturë fillestare t > 0
Përsërit
      nr ← 0
      Përsërit
            Përzgjidh s'Є N(s), në mënyrë të rastësishme
            Δe ← obj(s')-obj(s)
            Nëse Δe < 0
                  s ← s'
            Nëse Jo
                  Gjenero numër të rastësishëm r, 0≤r<1
                  Nëse r < e^(-δ/t) Atëherë s ← s'
            Mbyll Nëse
            nr ← nr + 1
      Deri sa nr = nr_zgjidhjesh
      Zvogëlo t sipas skedulimit të përcaktuar
Deri sa të arrihet kushti i përfundimit
```

Figura 2.8: Përshkrim në nivel të lartë i algoritmit simulated annealing, për një problem me hapësirë zgjidhjeje S, strukturë fqinjësh N, me funksion objektiv `obj` që duhet minimizuar dhe me numër zgjidhjesh `nr_zgjidhjesh` që duhen konsideruar për çdo nivel tëmperature t [35]

2.3.3 Kërkimi Tabu

Kërkimi Tabu (KT) është gjithashtu një teknikë që kërkon në afërsi dhe është propozuar në vitin 1986 nga Glover. Megjithatë në ndryshim nga SA ku përdoret një kërkim i rastësishëm, tek KT, kërkimi është më i kontrolluar.

Karakteristika kryesore e KT është se ky algoritëm fut konceptin e memorjes tek metaheuristika. Kjo memorje mban informacion mbi proçesin e kërkimit duke ruajtur

disa nga lëvizjet e kryera (ose atributet e tyre). Në këtë mënyrë ndihmohet kërkimi që të ecë përpara dhe të mos kthehet tek e një lëvizje që është kryer më parë. Këto lëvizje që nuk lejohen të rishqyrtohen quhen "tabu" dhe ruhen në një listë [37].

KT përdor kërkimin lokal; kërkon zgjidhjet fqinje për të përcaktuar lëvizjen pasardhëse (intensifikim). Për mënjanimin e problemeve të kërkimit lokal (optimum lokal dhe plateaux), KT pranon zgjidhje më të këqija kur nuk gjenden zgjidhje më të mira (diversifikim), në mënyrë që hapësira të mëdha të kërkimit të mos mbetën pa eksploruar. Gjithashtu "mos-lejimet" nuk lejojnë kërkimin të kthehet në një pikët të mëparshme. Algoritmi i KT jepet në figurën 2.9.

```
Përzgjidh një zgjidhje fillestare s Є S
x_optimal ← s
shto (listTabuFIFO, x_optimal)
Përsërit
      x_ri ← Mutacion(x_optimal)
      Nëse x_ri NUK ËSHTË tek listTabuFIFO
            shto (listTabuFIFO, x_ri)
            Nëse vlerësimi (x_ri) > vlerësimi (x_optimal)
                  x_optimal ← x_ri
            Mbyll Nëse
      Mbyll Nëse
Deri sa të arrihet kushti i përfundimit
```

Figura 2.9: Përshkrim në nivel të lartë i algoritmit KT, për një problem me hapësirë zgjidhjeje S, lista e ruajtjes së lëvizjeve të mëparshme `listTabuFIFO` [35]

2.3.4 Optimizimi i Kolonisë së Milingonave

Qasjet e Optimizimit të Kolonisë së Milingonave (OKM) janë algoritma që bazohen tek mënyra si punojnë milingonat. Ideja kryesore është paralelizimi i metodave bazuar në kërkim me një strukturë dinamike memorjeje që përfshin informacionin mbi efektshmërinë e rezultateve të përftuara më parë [38]. Ideja e OKM ka ardhur nga një eksperiment i zhvilluar në Argjentinë ku një koloni milingonash ishte lidhur me ushqimin nëpërmjet dy rrugëve me gjatësi të ndryshme. Rrugët ishin rregulluar në mënyrë të tillë që secila prej tyre të kishte të njëjtin probabilitet për t'u zgjedhur. U zbulua se pas pak kohe, shumica e milingonave filluan të zgjidhnin rrugën më të shkurtër. Eksperimenti u

përsërit duke rritur gjatësinë e rrugës më të gjatë dhe rezultati i përftuar ishte një zvogëlim i numrit të milingonave që zgjidhnin rrugën më të gjatë.

Zgjedhja e rrugës më të shkurtër nga milingonat, është rezultat i zgjidhjeve probabilitare që bëjnë milingonat si pasojë e një substance kimike, feromonit, që ato e çlirojnë gjatë ecjes. Zgjedhjet e tyre varen nga sasia e substancës që ato ndjejnë. Tek rruga më e shkurtër sasia e fenomonit ishte më e madhe, pasi milingonat arrijnë tek ushqimi më shpejt dhe kur kthehen ndjejenë substancën e cliruar gjatë vajtjes. Gjithashu një tjetër karakteristikë është fakti që feromoni me kalimin e kohës avullon dhe bën që sistemi të harrojë vendimet e gabuara të së shkuarës. Si rrjedhojë e këtij proçesi natyror janë zhvilluar algoritmat OKM.

OKM mund të aplikohen në probleme diskrete optimizimi që mund të karakterizohen në format grafik. "Milingonat" ecin në graf. Ato grumbullojnë informacion gjatë kërkimit që lidhet më secin hark. "Milingonat" depozitojne feromonin në proporcion me cilësisë. Kjo ndihmon në drejtimin e proçesit të kërkimit për "milingonat" në të ardhmen. Algoritmi OKM jepet në Figurën 2.10.

```
Pop ← {x0, …,xN}
x_optimal ← NULL
Përsërit
      PËR x TEK Pop
          PËR y TEK Fqinjët(x)
              LlogaritProb(P(x, y))
          FUND
          x ← zhvendos (x, P(x, *))
      FUND
      PËR ÇDO zhvendos (x, y)
           depozitoFeromonin (x, y)
      FUND
      rifreskoFeromonin (x, y)
      x_optimal ← mëeMira (Pop, x_optimal)
FUND
```

Figura 2.10: Përshkrim në nivel të lartë i algoritmit OKM [38]

2.3.5 Agoritmat Evolutivë

Algoritmat Evolutivë (AE) e bazojnë kërkimin mbi një popullatë; një tërësi zgjidhjesh kandidate përpunohet njëkohësisht. Të gjithë AE ngihen mbi të njëjtën ide kryesore: *mbi një popullatë me individë, mjedisi shkakton përzgjedhjen natyrore (mbijetesën e më të*

fortit) dhe kjo sjell një rritje në cilësinë e popullatës. AE dallojnë nga njëri tjetri në disa detaje teknike si p.sh. paraqitja e zgjidhjeve kandidate (enkodimi) etj. Mënyrat tipike të enkodimit janë: stringa që ndërtohen nga një alfabet i fundëm tek Algoritmat Gjenetikë, vektorë me vlera reale tek Strategjitë Evolutive, makina me gjendje të fundme tek Programimi Evolutiv dhe pemët tek Programimi Gjentik. Algoritmat Evolutivë më të njohur dhe në të cilat do të bazohet ky studim, janë Algoritmat Gjenetikë.

2.4 Algoritmat Gjenetikë

Algoritmat Gjenetikë (AG) u shfaqën fillimisht në fillim të viteve 1960 si rrjedhojë e punës së biologëve për të krijuar një model të evolucionit natyror. Përdorimi i AG filloi të përhapej me shpejtësi edhe në fusha të tjera. Sot AG bazohen në konceptin e krijuar nga Holland në 1975 [39]. Holland ishte i pari që propozoi një kombinim të dy operatorëve gjenetikë: kryqëzimin dhe mutacionin. Përdorimi i këtyre operatorëve nxorri në pah rëndësinë e zgjedhjes së një mënyre të mirë enkodimi për zgjidhjet kandidate.

Proçesi i kërkimit është përsëritës; individët më të mirë përzgjidhen për gjeneratën pasardhëse. Në versionin origjinal të propozuar nga Holland, numri i herëve që një individ përzgjidhet për t'u bërë prind është në proporcion të drejtë me cilësinë (fitness) e tij kundrejt individëve të tjerë të popullatës. Pas përzgjedhjes tek këta individë, aplikohen kryqëzimi dhe mutacioni. Si pasojë krijohen një grup i ri kandidatësh të cilët garojnë - duke u bazuar tek një funksion vlerësimi – për një vend në gjeneratën pasardhese. Ky proçes përsëritet deri sa një kandidat të ketë arritur vlerësimin e duhur ose të ketë përfunduar limiti i paravendosur i kërkimit. Në këtë proçes ka dy forca themelore që krijojnë bazën e sistemeve evolutive:

- Operatorët e ndryshimit (kryqëzimi dhe mutacioni) të cilët krijojnë diversitetin e nevojshëm
- Veprimet e përzgjedhjes që sigurojnë rritjen e cilësisë

Duhet theksuar se shumë prej komponentëve të proçesit evolutiv janë stokastike. Gjatë përzgjedhjes individët më të mirë kanë më shumë mundësi të përzgjidhen, megjithatë edhe individët e dobët kanë një mundësi të bëhen prindër edhe të mbijetojnë.

Gjatë kryqëzimit, zgjedhja e pjesëve që do rikombinohen është e rastësishme. Gjithashtu gjatë mutacionit, pjesët që do ndryshohen dhe pjesët e reja për zëvendësim zgjidhen në mënyrë të rastësishme. Pamja e përgjithshme e një algoritmi gjenetik jepet në figurën 2.11. Popullata fillestare zgjidhet në mënyrë të rastësishme.

```
x_optimal = NULL
Pop = {x0, …, xN} //Random
PËRSËRIT //me një buxhet kërkimi m
    kryejVlerësimin (Pop)
    x_optimal = mëIMirë (Pop, x_optimal)
    Pop = përzgjidh (Pop)
    Pop = riprodhim (Pop)
FUND
RETURN x_optimal
```

Figura 2.11: Përshkrim në nivel të lartë i AG

Algoritmat Gjenetikë kanë një sërë komponentësh të cilët duhet të përcaktohen në mënyrë që të ndërtohet një algoritëm specifik. Gjithashtu, që algoritmi të jetë funksional duhet që të përcaktohet edhe proçedura që bën inicializimin (përzgjedhja e popullsisë fillestare) dhe kushti i përfundimit të kërkimit. Komponentët kryesorë të AG janë:

- Paraqitja e individëve
- Funksioni i vlerësimit
- Popullata
- Mekanizmi i përzgjedhjes
- Operatorët e ndryshimit (kryqëzimi dhe mutacioni)
- Zëvendësimi

Paragrafët në vijim trajtojnë secilin prej komponentëve të algoritmave gjenetikë.

2.4.1 Paraqitja e Individëve

Hapi i parë në përcaktimin e AG është lidhja e "botës reale" me "botën e AG", që do të thotë, të vendoset një urë midis kontekstit të problemit origjinal dhe hapësirës së zgjidhjes së problemit në të cilën do të kryhet kërkimi. Objektet që formojnë zgjidhje të mundshme brenda kontekstit të problemit origjinal referohen si fenotipe, ndërsa enkodimi i tyre, që janë individët brenda AG, referohen si gjenotipe (ose kromozome). Proçesi i

paraqitjes së individëve bën një paraqitje të fenotipeve në një grup gjenotipesh. Për shembull, nëse kemi një problem optimizimi mbi numra të plotë, atëherë fenotipet janë bashkësia e grupit të dhënë të numrave të plotë. Më pas një mënyrë paraqitjeje do të ishte ti shprehnim këto numra të plotë sipas kodit binar dhe nëse kemi për shembull fenotipin 18, atëherë gjenotipi që e paraqet atë do të ishte 10010. Përdorimi i kodit binar është vetëm një prej mënyrave të enkodimit të individëve. Kjo mënyrë ka avantazhet dhe disavantazhet e saj. P.sh. një avantazh është se paraqitja binare e shpërbën kromozomin në numrin më të madh të blloqeve ndërtuese të tij duke bërë që rikombinimi dhe mutacioni të jenë më efektivë [40], ndërsa sipas [41], përdorimi i një alfabeteti më të sofistikuar do të sillte performancë më të mirë.

Hapësira e fenotipeve mund të jetë shumë e ndryshme nga hapësira e gjenotipeve. Gjithë kërkimi evolutiv kryhet në hapësirën e gjenotipeve. Zgjidhja përftohet duke dekoduar gjenotipin më të mirë pas përfundimit të kërkimit (për çdo gjenotip duhet të ketë maksimumi një fenotip korrespondues se në të kundërt nuk do të bëhej dot konvertimi).

2.4.2 Funksioni i Vlerësimit (Fitness Function)

Funksioni i vlerësimit (ose funksioni objektiv) është një pjesë shumë e rëndësishme e algoritmit të kërkimit. Ky funksion jep mundësinë e vlerësimit të individëve dhe në këtë mënyrë lejon kërkimin të drejtohet drejt individëve më të mirë me shpresën e gjetjes së një zgjidhjeje. Funksioni i vlerësimit formon bazën e përzgjedhjes dhe për rrjedhojë ndihmon përmirësimin [42]. Më saktësisht, ky funksion përcakton çfarë kuptojmë me përmirësim. Në mënyrë teknike mund të themi se ky është një funksion ose proçedurë që bën një matje të cilësisë të gjenotipeve. Në varësi të problemit, gjatë kërkimit mund të synohet ose të minimizohet ose të maksimizhet funksioni i vlerësimit. Duke qëndruar tek shembulli i mësipërm supozojmë se kërkohet të maksimizohet x^2 për numrat e plotë; atëherë vlerësimi për gjenotipin 10010 mund të përcaktohet si fuqia e dysh-it te fenotipit korrespondues: $18^8 = 324$.

Në kontekstin e testimit të mbulimit, në literaturën e mëparëshme, janë propozuar funksione vlerësimi që mund të ndahen në dy kategori: të bazuara në mbulim dhe në

kontroll. Në rastin e parë funksioni i vlerësimit synon të maksimizojë mbulimin, ndërsa në rastin e dytë këto funksione janë të ndërtuara për të ndihmuar kërkimin që të mbulojë disa konstrukte kontrolli. Më poshtë do të trajtohen shkurtimisht këto dy qasje.

Qasjet e orientuara drejt mbulimit zakonisht i sigurojnë kërkimit një drejtim të vogël. Ato synojnë të shpërblejnë ose penalizojnë rastet e testimit, duke u bazuar tek cilat pjesë të programit janë mbuluar ose janë lënë pa mbuluar [43]. Në këtë rast, konstrukte komplekse, si p.sh. kushte të mbivendosura ose predikata, kontrolli i të cilave varet nga një numër relativ i vogël vlerash inputi nga domain i madh, me shumë mundësi do të mbulohen vetëm rastësisht.

Qasjet e bazuara në strukturë tentojnë të jenë më të fokusuara. Algoritmi i kërkimit, do të kërkojë në mënyrë sistematike brenda një programi, të mbulojë gjithë strukturat, që kërkohen nga një kriter përshtatshmërie testimi. Informacioni që përdoret për të drejtuar kërkimin mund të bazohet tek distanca e nyjeve, të orientohet nga kontolli, ose një kombinim i të dyjave.

Në srategjitë e orientuara nga kontrolli, rrjedha e kontrollit dhe grafet e varësisë së kontrollit përdoren për të drejtuar kërkimin. Për shembull, Pargas [44] përdor nyjet kritike të kushteve për vlerësimin e individëve. Ideja është që sa më shumë nga këto nyje një test të kalojë aq më afër është me target-in e kërkuar. Një kusht ose shprehje që është target varet nga nyjet kritike të kushteve të tij. Hapësira e vlerësimit në këtë rast me shumë gjasa do të jetë shumë e gjërë pasi kërkimi nuk ka informacion se sa afër ishte një rast testimi për të ekzekutuar degën e kërkuar të nyjes kritike.

Një strategji e kombinuar përdor edhe distancën e degëve (llogaritja e saj do të shpjegohet në paragrafet në vijim) edhe varësinë e kontrollit. Tracey [45] ishte i pari që paraqiti një kombinim të të dyjave. Sa herë që ekzekutimi merr një anë të padëshiruar në një nyje kritike, p.sh. ndjek një rrugë që nuk çon tek target-i, llogaritet një distancë që mat sa afër ishtë ekzekutimi për të kaluar tek ana tjetër në nyjen e kushtit. Formula për funksionin e vlerësimit do të ishte:

$$\frac{nyjet\_ekzekutuara}{nyjet\_kritike} \times distanca\ e\ degeve$$

Edhe pse në shumë raste puna e Tracey ofron një funksion vlerësimi më të përmirësuar në krahasim me Pargas, prapë llogaritjet e vlerësimit mund të çojnë akoma në optimume të panevojshme lokale [46].

Wegener et al. [47] propozoi një funksion tjetër vlerësimi që bazohet në *distancën e degëve* dhe në *nivelin e afërsisë*. Niveli i afërsisë dokumenton sa nyje në varësinë e kontrollit të kushtit nuk u ekzekutuan nga një rast testimi i caktuar. Sa me pak nyje të jenë ekzekutuar aq më "larg" është rasti i testimit nga ekzekutimi i kushtit. Gjithashtu Wagener propozoi edhe që vlerat e distancës së degëvë të përkthehen në mënyrë logaritmike në një interval nga 0 në 1. Kjo distancë quhet distanca e normalizuar. Llogaritja e funksionit objektiv bëhet sipas formulës:

$$(nyjet\_kritike - nyjet\_ekzekutuara - 1) + distanca\_degeve\_normalizuar$$

Për mbulimin e shprehjeve dhe degëve funksioni i propozuar nga Wagener është funksioni që përdoret më shumë.

Për ilustrimin e koncepteve të distancës së degëve dhe nivelit të afërsisë do të marrim si shembull klasën Tekëndësh në Java që jepet në figurën 2.12.

```
class Trekendesh{
  void llogaritTipinTrekendesh() {
1     if (a == b)
2         if (b == c)
3           tipi = BARABRINJES;
4           else tipi = DYBRINJESHEM;
5     else if (a == c)
6             tipi = DYBRINJESHEM;
7     else if (b == c)
8             tipi = DYBRINJESHEM;
9     else kontrolloKendDrejte();
10    return;
}
```

Figura 2.12: Kodi i klasës Trekëndësh

Supozojmë se target-i është shprehja në rreshtin e tetë të metodës `llogaritTipinTrekendesh()`.Në figurën 2.13 jepet grafi për rrjedhën e kontrollit dhe të varësive për klasën Trekëndësh.

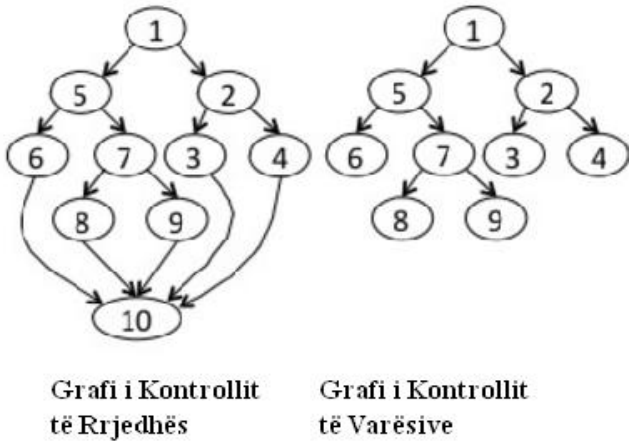

Figura 2.13: Grafi i kontrollit të rrjedhës dhe i kontrollit të varësive për klasën Trekëndësh

Nëse do të kishim dy raste testimi (vektor me tre input-e):

RT1 = (2, 2, 2)

RT2 = (2, 3. 4)

Si do llogaritej niveli i afërsisë?

Rruga (Rr) e ndjekur për secilin rast dhe niveli i afërsisë (NA) do të ishin (figura 2.14):

RT1 = (2, 2, 2) **Rr(RT1) = <1, 2, 3, 10>** **NA = 2**

RT2 = (2, 3. 4) **Rr(RT2) = <1, 5, 7, 9, 10>** **NA = 0**

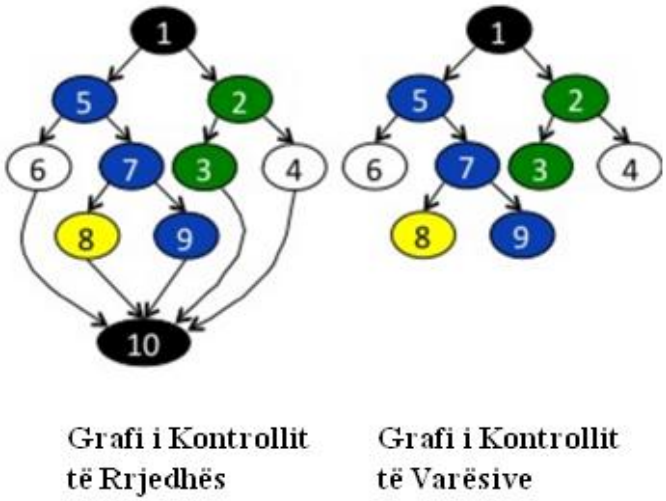

Figura 2.14: Grafi i kontrollit të rrjedhës dhe i kontrollit të varësive për klasën Trekëndësh, në rastin e inputeve RT1 dhe RT2

Si do të llogaritej distanca e degëve?

Predikata e nyjes së kontrollit e cila është më afër target-it konvertohet në një distancë e cila mat sa larg është një rast testimi nga zgjedhja e degës së dëshiruar. Kjo distancë në mënyrë që të mund të përdoret nga algoritmi gjenetik normalizohet sipas nje funksioni të caktuar. Në shumicën e rasteve ky funksion është:

$$DD(normalizuar) = 1 - 1.001^{-d}$$

ku **d** është distanca e degëve. Kjo distancë llogaritet më formula të ndryshme në varësi të predikatës që përman nyeja. Tabela 2.1, jep distancën d në rastin e variablave numerike/booleane a dhe b, ndërsa Tabela 2.2 jep distancën d në rastin e predikatave të përbëra **p** dhe **q**.

TABELA 2.1: LLOGARITJA E DISTANCËS SË DEGËVE [45] PËR PREDIKATAT RELACIONALE. K ËSHTË NJË KONSTANTE QË SHTOHET GJITHMONË NË RAST SE KUSHTI ËSHTË FALSE

Predikata Atomike	Distanca d
a	**d = {0 nëse a == true; K në të kundërt}**
!a	**d = {K nëse a == true; 0 në të kundërt}**
a == b	**d = {0 nëse a == b; abs(a-b) + K në të kundërt}**
a != b	**d = {0 nëse a != b; K në të kundërt}**
a < b	**d = {0 nëse a < b; a - b + K në të kundërt}**
a <= b	**d = {0 nëse a <= b; a - b + K në të kundërt}**
a > b	**d = {0 nëse a > b; b - a + K në të kundërt}**
a >= b	**d = {0 nëse a >= b; b - a + K në të kundërt}**

TABLE 1TABELA 2.2: LLOGARITJA E DISTANCËS SË DEGËVE [45] PËR PREDIKATAT LOGJIKE.

Predikata e Përbërë	Distanca d
!p	**Mohimi përhapet brenda p**
p & q	**d = d(p) + d(q)**
p | q	**d = min(d(p), d(q))**
p XOR q = p &!q | !p & q	**d = min(d(p) + d(!q), d(!p) + d(q))**

Në rastin e variablave booleane distanca mund të marrë vetëm vlerat 0 dhe 1 pasi këto variabla janë vetëm **true** ose **false.** Në rastin kur variablat që krahasohen janë referenca objektesh dhe përdoret operatori '= =', kemi dy mundësi: referencat shenjojnë tek i njëjti objekt dhe distanca është 0, ose referencat shenjojnë në objekte të ndryshme dhe distanca është 1. Edhe nëse përdoret metoda equals() për krahasimin e objekteve rezultati është i njëjtë.

Marrim sërisht në konsideratë klasën Trekendësh (figura 2.12). Nëse target-i do të ishte prapë rreshti i tetë dhe do të kishim rastin e testimit (vektor me tre input-e):

RT1 = (2, 2, 2)

Duke ju referuar grafit të kontrollit të varësive dhe nëse ndjekim rrugën e ekzekutimit të këtij rasti testimi (figura 2.15) do të kishim:

Distanca d (a != b) = K = 1

DD (a != b) = 1 / (1+1) = 0.5

funksioni_vlerësimit (RT1) = NA + DD = 2 +0.5 = 2.5

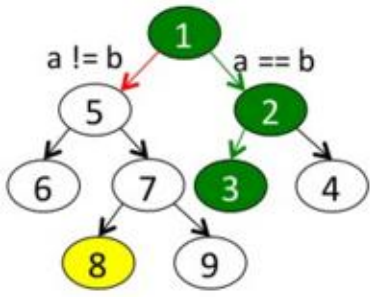

Figura 2.15: Distanca e nyjeve për rastin RT1 = (2, 2, 2)

Nëse do të kishim rastin e testimit (vektor me tre input-e):

RT2 = (2, 3, 4)

Duke ju referuar grafit të kontrollit të varësive dhe nësë ndjekim rrugën e ekzekutimit të këtij rasti testimi (figura 2.16), do të kishim:

Distanca d (b == c) = abs(b - c) + K = 2
DD (b == c) = 2 / (2+1) = 0.66
funksioni_vlerësimit (RT2) = NA + DD = 0 + 0.66 = 0.66

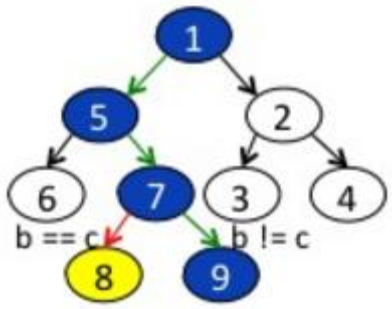

Figura 2.16: Distanca e nyjeve për rastin RT2 = (2, 3, 4)

Pra, rasti RT2 është më i mirë së rasti RT1 për ekzekutimin e rreshtit të tretë të klasës Trekëndësh.

2.4.3 Popullata

Popullata është një bashkësi gjenotipesh. Ajo përbën njësinë e evolucionit. Algoritmat gjenetikë përmbajnë një popullatë zgjidhjesh në vend të një zgjidhjeje të vetme. Për rrjedhojë kërkimi ka shumë pika fillimi dhe eksplorimi i hapësirës së kërkimit është më i madh se tek kërkimet lokale. Individët e popullatës janë objekte statike që nuk ndryshojnë ose adaptohen; ndërsa popullata, po. Ajo kalon nga një gjeneratë në një tjetër. Nëse është përcaktuar paraqitja e individëve, atëherë përcaktimi i popullatës është i thjeshtë pasi duhet vetëm të specifikohet sa individë do të ketë në të. Në ndryshim nga operatorët e ndryshimit të cilët veprojnë në një ose dy individë, operatorët e përzgjedhjes veprojnë në nivel popullate. P.sh. individi më i keq i *popullatës* do të përzgjidhet për t'u zëvendësuar me një të ri.

Diversiteti i popullatës është një matës i numrit të zgjidhjeve të ndryshme që janë të pranishme. Nuk ekziston një matës i vetëm për diversitetin; për matjen e tij mund të përdoret numri i vlerave të ndryshme të vlerësimit, numri i fenotipeve të ndryshme ose numri i gjenotipeve të ndryshme etj [48].

2.4.4 Mekanizmi i Përzgjedhjes

Detyra e këtij mekanizmi është të përzgjedhë individë, bazuar tek kualiteti i tyre, në mënyrë që këta individë të përzgjedhur të bëhen prindër për gjeneratën e ardhëshme. Përzgjedhja e prindërve është zakonisht probabilitare. Kjo do të thotë që individët më të mirë kanë mundësinë më të madhe për t'u bërë prindër, megjithatë edhe individët me cilësi jo të mira kanë një mundësi të vogël të përzgjidhen, pasi në të kundërt do të kishim një diversitet të ulët (popullësia do të ishte shumë e ngjashme pasi riprodhimi do të bëhej vetëm mbi një grup të vogël individësh) dhe kërkimi do të mbetej në optimume lokale. Janë zhvilluar shumë skema për të realizuar përzgjedhjen [49][50]. Disa prej tyre janë:

1. Renditja: përzgjedh çdo herë individët më të mirë të popullatës.
2. Rrota e ruletës: përzgjedh individët bazuar tek vlerësimi i tyre kundrejt popullatës. Probabiliteti që një individ të zgjidhet është:

$$p_1 = \frac{vleresim}{\sum_{i=0}^{gjat(popullates)} vleresim_i}$$

3. Përdoret skema e rrotës së ruletës për përzgjedhjen e dy individëve dhe më pas prej tyre zgjidhet individi me vlerësimin më të mirë.
4. Uniforme: zgjidhet një individ në mënyrë të rastësishme nga popullata.

2.4.5 Operatorët e Ndryshimit

Qëllimi i këtyre operatorëve është të krijojnë individë të rinj duke u nisur nga individët e vjetër. Këto operatorë, bazuar në numrin e individëve që marrin si input, ndahen në dy grupe: operatorët e kryqëzimit dhe të mutacionit.

2.4.5.1 Kryqëzimi

Kryqëzimi (ose Rikombinimi) është një operator ndryshimi binar. Ky operator shkrin informacion nga të dy prindërit duke formuar një gjenotip të ri. Ky operator është stokastik; zgjedhja e pjesëve që do të rikombinohen (ose pika e kryqëzimit) dhe mënyra se si do të kombinohen është e rastësishme. Operatorët e kryqëzimit që marrin më shumë se dy inpute janë matematikisht të mundura dhe janë të lehta për t'u implementuar (nuk kanë ekuivalente biologjike), megjithatë nuk përdoren shpesh.

Ideja e kryqëzimit është se duke bashkuar dy prindër me cilësi të mira do të përftohet një individ me cilësi edhe më të mira; megjithatë kjo nuk është gjithmonë e vërtetë. Ekzistojnë disa algoritma për kryerjen e kryqëzimit:

1. **Uniforme**: gjenet që do të përzgjidhen prej secilit prej prindërve do të zgjidhen rastësisht.
2. **Tek-çift**: do të zgjidhen gjenet me indeks tek nga prindi A dhe gjenet me indeks çift nga prindi B.
3. **Një pikë**: do të zgjidhet rastësisht një pikë bashkimi dhe gjithë gjenet në të majtë do të zgjidhen nga prindi A, ndërsa gjenet në të djathtë do të zgjidhen nga prindi B (figura 2.17 (a)).
4. **Dy pika:** do të zgjidhen rastësisht dy pika dhe më pas do të zgjidhen nga prindi A gjenet që kanë indeks më të vogël se pozicioni i parë dhe indeks më të madh se pozicioni i dytë; gjenet e mbetura do të zgjidhen nga prindi B (figura 2.17 (b)).

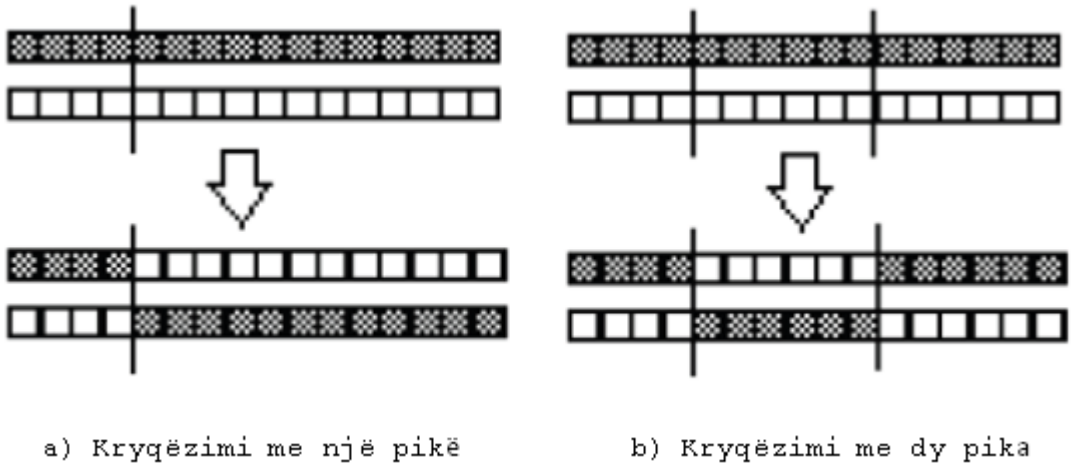

Figura 2.17: Skema e algoritmit të kryqëzimit në një pikë (a) dhe në dy pika (b)

5. **Përputhje e pjesshme:** nga kryqëzimi do të përftohen dy fëmijë. Tek fëmija C1 do të kopjohet prindi A ndërsa tek fëmija C2 do të kopjohet prindi B. më pas do të zgjidhen rastësisht pjesë për të shkëmbyer midis C1 dhe C2.
6. **Kryqëzim i renditur:** nga kryqëzimi do të përftohen dy fëmijë. Gjenet e dy prindërve kopjohen tëk dy fëmijët dhe tek secili prej fëmijëve fshihen *n* gjene rastësisht. Më pas kryhet një zhvendosje me *n* pozicione dhe tek pozicionet e lira vendosen gjenet e fëmijës tjetër. Ky algoritëm ilustrohet në figurën 2.18.

Figura 2.18: Skema e algoritmit të kryqëzimit të renditur

2.4.5.2 Mutacioni

Mutacioni është një operator ndryshimi unar. Ai aplikohet në një gjenotip dhe gjeneron një fëmijë që është i ndryshëm nga prindi. Mutacioni është përherë një veprim stokastik; output-i varet nga një seri zgjedhjesh të rastësishme. Mutacioni supozohet të krijojë një ndryshim të rastësishëm dhe të paanshëm. Mutacioni ka gjithashtu një rol teorik; ai siguron që hapësira është e lidhur. Kjo është e rëndësishme pasi teoremat që vërtetojnë se algoritmi gjenetik (nëse ka kohë të mjaftueshme), do të gjejë optimumin global, zakonisht bazohen tek vetia se çdo gjenotip që përfaqëson një zgjidhje të mundshme për një problem mund të arrihet nga operatorët e ndryshimit. Mënyra më e thjeshtë për të përmbushur këtë kusht është të lejohet që operatori i mutacionit të mund të

“kërcejë” kudo, p.sh. duke lejuar secilën alele që pas mutacionit të konvertohet, me probabilitet të ndryshëm nga zero, në çdo alele tjetër.

Disa prej mënyrave kryesore se si mund të kryhet mutacioni janë [51]:

1. **Mutacioni me hedhje:** një gjen i kromozomit do të ndryshohet me një vlerë të rastësishme.
2. **Mutacioni me shkembim:** do të shkëmbejë në mënyrë të rastësishme disa gjene tek kromozomi.
3. **Mutacioni Gausian:** do të zgjedhë rastësisht një vlerë të re e cila do të përftohet duke përdorur shpërndarjen Gausiane rreth vlerës aktuale.

Duhet theksuar se operatorët e ndryshimit varen nga paraqitja e individëve; d.m.th. për paraqitje të ndryshme duhet të përcaktohen operatorë të ndryshëm.

2.5 Sfidat dhe zgjidhjet për TSBK

Në këtë paragraf janë përmbledhur nga literatura sfidat kryesore edhe disa prej zgjidhjeve të propozuara, të cilat janë aplikuar në mjetet më me rendësi bazuar në TSBK. Këto sfida janë:

1) **Inputet e pamundura:** rastet e testimit të gjeneruara në mënyrë automatike në nivel njësie mund të jenë të pamundura; që do të thotë se ato përfaqësojnë një ekzekutim që nuk do të ndodhë asnjëherë në praktikë. Dështimet që shkaktojnë këto raste testimi janë të vështira për t’u identifikuar dhe eliminuar dhe për rrjedhojë sjellin harxhim të burimeve të testimit. Zgjidhje propozohen në citimet [52][53].
2) **Rritja e pakontrolluar e gjatësisë së rasteve të testimit:** është një fenomen kompleks gjatë kërkimit ku gjatësia e individëve rritet në mënyrë jo normale me kalimin e kohës, duke e bërë kështu vazhdimin e kërkimit të pamundur. Gjatësia është një parametër që duhet të ketë një limit, jo vetëm për vazhdimin e punës së algoritmit, por edhe për faktin se vlerësimi i rezultateve deri sot kryhet manualisht dhe me rritjen e gjatësisë rritet edhe vështirësia e gjenerimit të oracle. Bazuar tek [54] edhe pas shumë dekadave punë në këtë drejtim, rritja e gjatësisë së një rasti

testimi është akoma një problem i hapur ma natyrë dhe dinamikë ende të pa kuptuar plotësisht. Nuk ka një dëshmi të qartë mbi zgjedhjen e gjatësisë dhe mënjanimin e rritjes së pakontrolluar. Për kontrollin e këtij problemi përdoren një kombinim i teknikave të ndryshme. Zgjidhje propozohen në citimet [54][55].

3) **Parametrizimi i Algoritmit:** algoritmat gjenetikë duhet të parametrizohen që të japin zgjidhje të vlefshme. Vlera të ndryshme të parametrave mund të jenë optimale për probleme të ndryshme ose edhe për instanca të ndryshme të të njëjtit problem. Teknikat më të fundit përdorin optimizimin që përshtatet, i cili përdor feedback për të rregulluar vlerat e parametrave gjatë kërkimit. Ka shumë parametra që duhet të përcaktohen por më të rëndësishmet janë madhësia e popullatës, raporti i kryqëzimeve dhe raporti i mutacioneve. Zgjidhje propozohen në citimet [56][57][58][59].

4) **Optimumi Lokal:** ky problem shfaqet kur kërkimi konvergjon para kohe në një optimum lokal. Ky fenomen ndodh kur ka një diversitet të limituar të popullatës e për rrjedhojë në gjeneratën e fundit ka individë që e kanë funksionin e përshtatshmërisë më të mirë se fqinjët e tyre por që nuk janë optimal globalisht edhe në këtë mënyrë pengojnë eksplorimin e mëtejshëm. Në këtë kategori bën pjesë edhe problemi që shfaqet kur individët kanë të njëjtin funksion të përshtatshmërisë me fqinjët dhe kërkimi kthehet në zgjedhje të rastësishme. Zgjidhja në këtë rast është kombinimi i algoritmave gjenetikë me testimin e bazuar ne kufizime (p.sh. të përdoret Ekzekutimi Dinamik Simbolik si operator për të kryer mutacionin duke siguruar kështu marrjen e një individi të ri dhe një diversitet më të madh të popullatës). Nje zgjidhje propozohet në citimin [60].

5) **Një qëllim mbulimi i pamundur:** kur kërkohet të përmbushet një kriter i caktuar mbulimi, p.sh. mbulim i degëve, mund të shfaqen tre probleme:

    - Qëllimi i mbulimit është i pamundur që do të thotë se nuk ka rast testimi që ta përmbushë atë.
    - Disa qëllime mbulimi janë më të vështira për t'u përmbushur që do të thotë se sasia e limituar e burimeve nuk do të shpërndahet në mënyrë të

barabartë, pasi do të harxhohet më shumë kohë për mbulimin e qëllimeve më të vështira.

- Ekzistenca e mbulimit indirekt që do të thotë se nëse një qëllim mbulimi përmbush edhe qëllime të tjera këto qëllime nuk merren në konsideratë, duke sjellë një keqpërdorim të burimeve.

Zgjidhja në këtë rast [61], është gjenerimi i bashkësisë së rasteve të testimit si një e tërë kundrejt përmbushjes së një kriteri mbulimi dhe jo gjenerimi i rasteve të testimit në mënyrë të veçantë drejtuar qëllimeve të ndryshme të mbulimit. Zgjidhje propozohen në citimet [61][62].

6) **Flamuri Boolean:** vlerësimi i kandidatëve, siç u shpjegua edhe më sipër, bazohet tek funksioni i përshtatshmërisë i cili bazohet tek distanca në rrjedhën e kontrollit. Flamujt boolean e humbin informacionin mbi distancën dhe për rrjedhojë nuk japin asnjë drejtim për përshtatshmërinë gjatë kërkimit. Zgjidhja këtu është [63] transformi i kodit burim në mënyrë të tillë që informacioni i humbur gjatë kërkimit të përhapet tek kushtet ku keto flamuj përdoren. Zgjidhje propozohen në citimet [63][64][65][66].

7) **Varësitë nga Mjedisi:** kur kodi që po testohet ndërvepron me mjedisin si p.sh. me sistemin e skedarëve, me rrjetin, me përdoruesin, etj, atëherë kodi nuk mund të mbulohet tërësisht dhe testet e gjeneruar mund të jenë të paqëndrueshme se pasojë e këtyre ndërveprimeve. Bazuar tek [67] në një studim për rastet e kodit të orientuar nga objekti, është vënë re se kontrolli i mjedisit mund të ketë një ndikim deri në +80%/+90% tek mbulimi i degëve. Zgjidhja për këtë problem është të izolohet njësia që po testohet nga mjedisi dhe gjendja e mjedisit të përfshihet si pjesë e hapësirës së input-ve. Zgjidhje propozohen në citimet [67][68].

Tabela 2.3 përmbledh sfidat e TSBK, disa prej zgjidhjeve që janë propozuar së fundmi për ti kapërcyer këto sfida dhe mjetet më me influencë që janë zhvilluar mbi këto zgjidhje. Zgjidhjet e paraqitura në tabelë janë propozuar në periudhën 2010-2015.

TABELA 2.3: SFIDAT, ZGJIDHJET E PROPOZUARA DHE MJETET E ZHVILLUARA MBI TSBK

Sfidat e TSBK	Citimi që përmban zgjidhjen e propozuar	Mjete të zhvilluara si pasojë e zgjidhjeve
Inputet e Pamundura	**[52][53]**	**EXSYST, BugEx**
Rritja e Pakontrolluar e Gjatësisë së Rasteve të Testimit	**[54][55]**	**Evosuite**
Parametrizimi i Algoritmit	**[55][56][57][58]**	**Evosuite**
Optimumi Lokal	**[59]**	**AUSTIN**
Një qëllim mbulimi i pamundur	**[60][61]**	**Evosuite**
Flamuri Boolean	**[62][63][64][65]**	**Evosuite**
Varësitë nga mjedisi	**[66][67]**	**Evosuite**

KAPITULLI 3

Testimi Evolutiv në Nivel Njësie

Në këtë kapitull do të bëhet një përmbledhje e kritereve të mbulimit që nevojitet të përmbushen për tëstimin e klasave në Java (dhe në përgjithësi për gjuhët e orienturara nga objekti). Këtu do të shpjegohen shkurtimisht disa tipare të Java-s dhe do të përshkruhen cilat duhet të jenë përshtatjet që i duhen bërë algoritmave gjenetikë (gjithë komponentëve të tyre) që të mund të aplikohen në nje kod në Java.

3.1 JUnit

Duke patur parasysh rëndësinë e testimit në nivel njësie, në vitin 1997, dy studiues Erich Gamma dhe Kent Beck krijuan JUnit; një framework i thjeshtë dhe efektiv për testimin e njësive në Java. Shkrimi i testeve të thjeshta nuk është i vështirë, megjithatë kur testet janë më komplekse, shkrimi dhe mirëmbajtja e tyre bëhet më e vështirë. JUnit e bën më të lehtë krijimin, ekzekutimin dhe rishikimin e testeve. Gjithashtu disa prej mjeteve që gjenerojnë automatikisht teste në nivle njësie përdorin formatin e JUnit, pasi ekzekutimi i tyre eshte i thjeshtë edhe testet janë të standartizuara dhe mund të kuptohen lehtësisht nga testuesit.

JUnit është me kod të hapur (junit.org) dhe është i vendosur në SourceForge [68]. JUnit sot është framework-u standart për të zhvilluar teste në nivel njësie për programe në Java. Ekzistojnë edhe framework-e të tjerë xUnit të gatshme për gjuhët ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, Visual Basic etj

Qëllimet e JUnit janë [69]:

- Ky framework duhet të ndihmojë në shkrimin e testeve që kanë vlerë.

- Ky framework duhet të ndihmojë në shkrimin e testeve që e ruajnë vlerën me kalimin e kohës.
- Ky framework duhet të ndihmojë në uljen e kostos së shkrimit të testeve duke përdorur ripërdorimin e kodit.

Supozojmë se kemi një klasë të thjeshtë me emër `Llogaritesi`:

```
public class Llogaritesi
{
	public double add(double numer1, double numer2)
	{
		return numer1 + numer2;
	}
}
```

Një program i shkruar me JUnit për tesimin e klasës Llogaritesi do të ishte:

```
import junit.framework.TestCase;
public class TestLlogaritesi extends TestCase
{
	public void testAdd()
	{
		Llogaritesi l = new Llogaritesi();
		double rezultati = l.add(10, 50);
		assertEquals(60, rezultati, 0);
	}
}
```

Ky shembull është një rast shumë i thjeshtë i testimit të njësisë dhe shërben këtu vetëm për të dhënë idenë e një testi të shkruar me JUnit.

Si fillim duhet që klasa e ndërtuar të trashëgohet nga klasa TestCase që ndodhet në paketën `junit.framework`. Kjo klasë përmban kodin që i duhet JUnit për të ekzekutuar automatikisht testet. Gjithë metodat e klasës duhet të kenë emrin sipas modelit `testXXX()`. Ky lloj emërtimi (p.sh. `testAdd()`), i bën të qartë framework-ut se kjo metodë është një test njësie dhe mund të ekzekutohet automatikisht. Më pas duhet të ndërtohet një instancë e klasës që është nën testim (në këtë shembull klasa Llogaritesi). Nëpërmjet kësaj instance thirret një metodë e klasës duke i dhënë argumentat e duhura. Në fund për të kontrolluar rezultatin përdoren metodat e trahëguara nga klasa

`TestCase`, si p.sh. metoda `assertEquals()` që krahason dy vlera që merr si argument. Output-i i pritur i kalohet si argument i parë kësaj metode. Nëse pohimi del i vërtetë, atëherë testi kalon, në të kundërt testi dështon. Edhe përjashtimet e padeklaruara, që gjenerohen nga programi që po testohet, cilësohen si teste të dështuara. Gjithë metodat që do ekzekutohen duhet të jenë `public` dhe të kthejnë `void`.

3.2 Kriteret e mbulimit për testimin në nivel njësie të bazuar në kërkim

Në testimin e bazuar në kërkim, funksioni i vlerësimit bazohet në një objektiv optimizimi dhe ky objektiv zakonisht përcaktohet nga një ose disa kritere mbulimi. Këto kritere, jo vetëm drejtojnë kërkimin në rastin e testimit bazuar në kërkim, por gjithashtu shërbejnë edhe për vlerësimin e rasteve të testimit [71]. Ekzistojnë kritere të ndryshme dhe me fortësi të ndryshme. Më poshtë shpjegohen kriteret e mbulimit strukturor dhe testimit bazuar tek mutacionet, që përdoren më shumë në ditët e sotme.

Është e nevojshme këtu të theksojmë ndryshimin midis testimit strukturor dhe mbulimit strukturor. Qëllimi i mbulimit strukturor është të përcaktojë cilat struktura të kodit nuk janë ushtruar nga rastet e testimii. Testimi strukturor është proçesi i aplikimit në software të ratesve të testimit që janë shkruar nga kodi burim dhe jo nga kërkesat.

1. **Mbulimi i Metodave.** Është kriteri më minimal për klasat dhe kërkon që të gjithë metodat e një klase që po testohet, të ekzekutohen nga seti i rasteve të testimit.
2. **Mbulimi i Metodave në Nivel të Lartë.** Në rastin e testimit në regres duhet që secila metodë të thirret në mënyrë direkte. Për këtë arsye, ky kriter kërkon që të gjitha metodat e klasës që po testohet të mbulohen nga seti i rasteve të testimit në mënyrë të tillë që thirrja e çdo metode të shfaqet si një shprehje në një rast testimi.
3. **Mbulimi i Metodave në Nivel të Lartë pa Përjashtime.** Zakonisht klasat kanë pak metoda dhe një set i rasteve të testimit arrin mbulim të madh të metodave duke i thirrur këto metoda kur objekti është në një gjendje të pavlefshme ose duke i thirrur ato me parametra të pavlefshëm. Për mënjanimin e këtyre rasteve, kriteri Mbulimi i Metodave në Nivel të Lartë pa Përjashtime, kërkon që gjithë metodat të

mbulohen duke u thirrur direkt dhe gjithashtu merr në konsideratë vetëm ekzekutimet që përfundojnë normalisht (nuk gjenerohen përjashtime).

Funksionet e vlerësimit për tre kriteret e mësipërme janë diskrete dhe nuk kanë mundësi që të ofrojnë drejtim (të tregojnë sa larg është një rast testimi nga përmbushja e mbulimit të një target-i) gjatë kërkimit. Vlerat e vlerësimit llogariten thjeshtë duke numëruar metodat që janë mbuluar nga një set rastesh testimi.

4. **Mbulimi i Rreshtave.** Kriteri bazë do të ishte mbulimi i shprehjeve, por për arsye se mjetet moderne të testimit për Java ose C# nuk përdorin si input kodin burim por bytecode-in, atëherë mbulimi i rreshtave është më i përshtatshëm pasi instruksionet në bytecode mund të mos përputhen direkt me shprehjet në kodin burim. Çdo shprehje në një klasë ka një rresht të përcaktuar, që përcakton pozicionin e shprehjes në kodin burim të klasës. Kodi burim i klasës përbëhet nga rreshta që nuk përmbajnë komente dhe rreshta që nuk përmbajnë kod (rreshta bosh ose komente). Një set rastesh tesimi për testim në nivel njësie, përbush këtë kriter nëse mbulon të gjithë rreshtat që përmbajnë kod. Ky lloj mbulimi është shumë i lehtë për t'u vizualizuar, interpretuar nga një mjet analizues (p.sh. mjeti EclEmma [72]). Ndoshta kjo është arsyeja se pse ky kriter është kaq shumë i përdorur. Për të mbuluar çdo rresht në kod duhet që secili prej blloqeve bazë të arrihet gjatë ekzekutimit. Siç u shpjegua në kapitullin 2, arritja e një blloku në testimin e bazuar në kërkim mund të shprehet me anë të nivelit të afërsisë dhe distancës së degëve.
5. **Mbulimi i Degëve.** Koncepti i mbulimit të degëve është shumë popullor dhe është i implementuar në shumë mjete testimi. Mbulimi i degëve interpretohet si maksimizimi i numrit të degëve të kushteve që mbulohen nga një set rastesh tesimi. Kjo do të thotë që një set rastesh testimi e përmbush këtë kriter nëse për secilin kusht përmban të paktën një rast testimi që kur ekzekutohet e bën kushtin të vërtetë dhe të paktën një rast testimi që kur ekzekutohet e bën kushtin jo të vërtetë. Supozojme se kemi një kusht të përbërë (`A or B`), ku A dhe B jane kushte booleane. Që të përmbushet mbulimi i dy degëve që dalin nga ky kusht, do të mjaftonin dy rastet e testimit (TF) dhe (FF). Shikojmë që me këtë kriter efekti i

B-së nuk do të testohet; kështu këto teste nuk mund të bëjnë diferencën midis kushtit `(A or B)` dhe kushtit `(A)`.

6. **Mbulimi i Kushteve.** Për të eleminuar problemin e kriterit të mbulimit të degëve, ky kriter kërkon që secili prej kushteve booleane në një kusht të përbërë të marrë gjithë vlerat e mundshme, por nuk kërkon që kushti i përbërë të marre gjithë vlerat e mundshme. Në këtë rast për kushtin `(A or B)`, do të mjaftonin dy rastet (TF) dhe (FT) për ta përmbushur këtë kriter, edhe pse në të dy rastet rezultati i shprehjes (A or B) do të dalë `true`.
7. **Mbulimi i Kushteve/Degëve.** Ky kriter kombinon kërkesat e mbulimit të degëve dhe mbulimit të kushteve. Në këtë rast, për kushtin `(A or B)`, do të mjaftonin dy rastet (TT) dhe (FF) për ta përmbushur këtë kriter. Megjithatë këto dy raste nuk mund të bëjnë diferencën midis kushtit `(A or B)` dhe kushtit `(A and B)`.
8. **Mbulim i Modifikuar i Kushteve/Degëve.** Ky kriter e shton fortësinë e kriterit të mbulimit të kushteve/degëve pasi kërkon që për çdo kusht të tregohet që në mënyrë të pavarur ndikon në degën e zgjedhur. Kërkesa e pavarësisë siguron që efekti i çdo kushti testohet në mënyrë relative nga kushtet e tjera. Në këtë rast për kushtin `(A or B)`, do të duheshin tre rastet (TF), (FT) dhe (FF) për ta përmbushur këtë kriter. Arritja e këtij kriteri kërkon në mënyrë të konsiderueshme më shumë raste testimi se kriteret e sipërpërmendura. Testimi strukturor i software-ve që përdoren në aeronautikë dhe që bëjnë pjesë në nivelin më kritik (Niveli A), duhet të përmbushë kriterin e mbulimit të Modifikuar të Kushteve/Degëve.
9. **Mbulim i Shumëfishtë i Kushteve.** Ky si kriter kërkon që rastet e testimit të sigurojnë gjithë kombinimet e mundshme te kushteve booleane në një kusht të përbërë [73]. Nga ana teorike ky kriter do të ishte më i miri, por nuk është praktik në shumë raste. Për një kusht të përbërë me n kushte booleane do të duheshin 2^n raste testimi për përmbushjen e këtij kriteri.
10. **Mbulimi i Path-ve.** Ky kriter kërkon të mbulohen gjithë path-et e mundshme në një kod burim. Ky është një kriter shumë i fortë, por duke qenë se mund të këtë

një numër të pafundëm path-esh dhe disa prej tyre mund të jënë të pamundura atëherë ky kriter nuk mund të përmbushet praktikisht.

11. **Mbulimi Direkt i Degëve.** Kur një rast tesimi mbulon një degë në një metodë publike pa e thirrur në mënyrë direkte metodën që përmban degën, është më e vështirë të kuptohet se si testi është i lidhur me degën që mbulon. Gjitashtu është e vështirë që të bëhen pohimet për outputin e gjeneruar nga ekzekutimi i testit [3]. Kriteri i mbulimit direkt të degëve kërkon që secila degë e e një metode publike të mbulohet nga një thirrje direkte e metodës, por ky kriter nuk bën ndonjë kufizim për metodat private. Funksioni i vlerësimit për këtë kriter do të ishte i njëjtë me funksionin e vlerësimit për kriterin e mbulimit të degëve përveç faktit që vetëm metodat e thirrura direkt do të merren në konsideratë.
12. **Mbulimi i Output-it.** Ka disa metoda në Java që njihen si observues ose inspektues, këto metoda thjesht kthejnë vlerën e një pjestari të klasës ku bëjnë pjesë. Për këto metoda si kriteri i mbulimit të metodave, kriteri i mbulimit të rreshtave, ose kriteri i mbulimit të degëve janë identike. Në këto raste, për të rritur efikasitetin e testimit, duhet të gjenerohen raste testimi që mbulojnë jo vetëm input-in e metodës po edhe outputin (vlerën e kthimit të metodës). Diversiteti i output-it ndihmon në përmirësimin e mundësisë së detektimit të gabimeve [74].

    Objektivat e mbulimit të output-it varen nga tipi i kthimit të metodës. Funksioni që bën përkthimin e vlerave të kthimit në vlera abstrakte për përcaktimin e objektivave të mbulimit është:

$$tipi\_output = \begin{Bmatrix} \{true, false\} & \text{nëse Tipi} \equiv \text{Boolean} \\ \{-, 0, +\} & \text{nëse Tipi} \equiv \text{Numër} \\ \{alfabetik, shifër, *\} & \text{nëse Tipi} \equiv \text{Char} \\ \{null, \neq null\} & \text{në të kundërt} \end{Bmatrix}$$

13. **Mutacion i Dobët.** Mjetet e testimit automatik zakonisht gjenenrojnë vlera për përmbushjen e disa detyrimeve ose kushteve, por jo vlera që do të përcaktonte një testues. Testuesit për shembull, përdorin vlerat kufi. Testimi automatik mund të detyrohet të gjenerojë këto vlera nëse përdoret kriteri i mutacionit të lehtë. Në këtë lloj testimi, klasës që po testohet i bëhen disa modifikime të vogla. Më pas shikohet nëse rasti i testimit arrin të dallojë midis klasës origjinale dhe klasës me mutacion. *Përkufizimi i mbulimit në mënyrë të dobët të mutantëve është:*

Nëse kemi një mutant $m \in M$ që modifikon një pozicion l në një program P dhe një test t, atëherë themi që t mbulon në mënyrë të dobët m vetëm nëse gjëndja e ekzekutimit të P me t është e ndryshme nga gjëndja e ekzekutimit të m direkt pas l.
Numri i muatantëve të gjeneruar varet nga klasa që po testohet dhe mund të jetë shumë i madh [75].

14. **Mutacion i Fortë.** Edhe në këtë lloj testimi, ashtu si në kriterin e mutacionit të dobët, klasës që po testohet i bëhen disa modifikime të vogla. Më pas shikohet nëse rasti i testimit arrin të dallojë midis klasës origjinale dhe klasës me mutacion.
Përkufizimi i mbulimit në mënyrë të lehtë të mutantëve është:
Nnëse kemi një mutant $m \in M$ në një program P dhe një test t, atëherë themi që t mbulon në mënyrë të fortë m vetëm nëse output i P me t është i ndryshëm nga output i t në m.
Është e qartë që ky kriter është më i fortë se kriteri i mutacionit të dobët.
15. **Mbulim i Përjashtimeve**. Kriteret e mbulimit të sipërpërmendura do të kapnin përjashtimet që mund të gjenerohen vetëm nëse përjashtimet hidhen në mënyrë direkte më një shprehje `throw`. Megjithatë, në rastin e përjashtimeve të paqëllimshme (p.sh. thirrja e një metode nga një instancë null do të gjeneronte një përjashtim të tipit `NullPointerException`) ose kur përjashtimi hidhet nga trupi i një metode të thirrur nga klasa që po testohet. Fatkeqësisht, nuk është e mundur të dihet që në fillim numri i përjashtimeve të padeklaruara. Si kriter mbulimi këtu konsiderohen gjithë përjashtimet e mundshme në çdo metodë. Megjithatë në ndryshim nga kriteret e tjera këtu nuk mund të përdoret përqindja. Funksioni i vlerësimit është diskret dhe llogaritet si numri i përjashtimeve eksplicite dhe implicite të gjeneruara gjatë ekzekutimit të gjithë testeve në setin e rasteve të testimit.

Ekzistojnë edhe kritere të tjera, por kriteret e sipërpërmendura janë më të njohurat për tesimin e software-ve të shkruara në gjuhë të orientuara nga objekti në nivel njësie. Megjithatë jo gjithë këto kritere kanë të njëjtën fortësi dhe mund të përmbushen praktikisht. Gjithashtu disa kritere përfshihen nga kritere të tjera. Një renditje e kritereve strukturore bazuar në fortësinë e tyre do të ishte si në figurën 3.1.

Dy kriteret e para në figurën 3.1, nuk mund të realizohen praktikisht prandaj kriteri strukturor më i fortë sot është mbulimi i modifikuar i kushteve/degëve.

Figura 3.1: Renditja e kritereve strukturore bazuar në fortësinë e tyre

Dy kriteret e para në figurën 3.1, nuk mund të realizohen praktikisht prandaj kriteri strukturor më i fortë sot është mbulimi i modifikuar i kushteve/degëve.

Megjithatë kriteri më i fortë për testimin në nivel njësie është Mutacioni i Fortë. Arsyeja është se ky kriter prodhon më shumë kërkesa që duhet të plotësohen se çdo kriter tjetër. Numri i këtyre kërkesave është vështirë të përcaktohet se varet nga kodi që po testohet dhe numri i operatorëve të muatcionit. Ky kriter njihet edhe si "standarti i artë" [75]. Duke qenë se ky kriter është i vështirë për t'u përmbushur si manualisht edhe automatikisht atëherë ky kriter përdoret kryesisht për vlerësimin e kritereve të tjera në studimet eksperimentale.

Operatorët kryesorë të mutacionit janë:

- Zëvendësimi i operatorëve binarë: zëvendësohen operatorët binarë si ato atitmetikë, logjikë, të kushtëzimit, të relacionit, të zhvendosjes etj.
- Zëvendësimi i operatorëve unarë: zëvendësohen operatorët unarë si mohimi
- Zëvendësimi i vlerave konstante: zëvendësohen konstantet numertike dhe të tipit string me vlera të parapërcaktuara
- Ndryshimi i kushteve të kërcimit
- Fshirja e shprehjeve: fshihen shprehje të veçanta si p.sh. thirrja e metodave.

Efikasiteti i këtij kriteri është tashmë i vërtetuar, por ky kriter është shumë i kushtueshëm për t'u aplikuar. Llogaritja e rezultatit të mutacionit (mutation score) për një set rastesh testimi, kërkon që për çdo mutant të përcaktohet nëse ka ndonjë rast testimi që e vret apo jo këtë mutant. Ka disa mutantë për çdo shprehje të një klase; kjo sjell që edhe për klasa shumë të thjeshta të kemi një numër të konsiderueshëm mutantësh. Gjithashtu një problem tjetër për këtë lloj kriteri janë edhe muatntët ekuivalentë të cilët ndryshojnë vetëm sintaksën e programit dhe jo semantikën e tij duke u bërë kështu të padetektueshëm nga asnjë test. Për ilustrim, në figurën 3.2 (a) paraqitet një metodë në Java dhe në figurën 3.2 (b) dhe 3.2 (c), paraqiten rastet e dy mutantëve ekuivalentë me metodën e saktë [75]. Një sërë studimesh i janë dedikuar kriterit të mutacionit për të ulur kompleksitetin e tij, megjithatë akoma në ditët e sotme ky kriter mbetet jo i përdorshëm praktikisht.

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

(a)

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

(b)

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

(c)

Figura 3.2: (a) Metoda max në Java; (b), (c) mutantë ekuvalentë me rastin (a)

Gjithashtu për TSBK duhet marrë në konsideratë edhe nëse një kriter është apo jo i përshtatëshëm për t'u përdorur si bazë për funksionin e vlërësimit të algoritmit të kërkimit të implementuar nga një mjet për tëstim automatik.

Kriteri më i përdorur është mbulimi i degëve [23]; ky kriter është tashmë kriteri default i vendosur në literaturë për ndërtimin e mjeteve automatike. Megjithatë ka raste kur ky kriter prodhon sete ratesh testimi të dobëta dhe për këtë arsye përcaktimi i një kriteri mbulimi të përshtatshëm për tu përdorur në testimin automatik është akoma një fushë e hapur studimi.

Duke qenë se kritere të ndryshme zakonisht veprojnë mbi pjesë të ndryshme të kodit atëherë një zgjidhje do të ishte edhe mundësia e kombinimit të disa kritereve; në këtë rast shuma e kërkesave të secilit prej kritereve do të shërbejë si funksion vlerësimi për proçesin e kërkimit. Çështja në vijim bën një vlerësim eksperimental të disa prej kritereve të mbulimit dhe të kombinimit të tyre kur përdoren për testimin e klasave në Java.

3.3 Vlerësimi i Kritereve të Mbulimit dhe i Kombinimit të tyre

Qëllimi i kësaj çështjeje është të studiojë efikasitetin e kritereve të mbulimit më të përdorura sot. Duke qenë se secili prej kritereve ka një target të veçantë mbulimi, atëherë interes paraqet edhe kombinimi i tyre, prandaj në këtë çështje do të vlerësohet edhe kombinimi i disa kritereve të cilat kur implementohen nuk kanë konflikt me njëra tjetrën. Gjithashtu do të bëhet një analizë e avantazheve dhe disavantazheve të këtij kombinimi.

Për matjen e efekteve që ka kombinimi i kritereve të ndryshme në gjenerimin automatik, në këtë punim u përdor mjeti EvoSuite [76].

3.3.1 EvoSuite

EvoSuite është një mjet për gjenerimin automatik të rateve të testimit në nivel njësie për programet në Java. Disa nga karakteristikat kryesore të EvoSuite listohen më poshtë:

- Përdorimi: Nga rreshti i komandave ose si një plug-in në Eclipse
- Input-i: Bytecode i klasës që do testohet

- Output-i: Raste testimi sipas formatit JUnit
- Teknologjia e përdorur për gjenerim: Algoritmat Gjenetikë
- Disponueshmëria: Me kod të hapur
- Sistemi Operativ ku mund të përdoret: Mac, Linux

Karakteristika kryesore e EvoSuite është se ky mjet gjatë kërkimit nuk kërkon më vete të gjejë një rast testimi që të mbulojë një target të caktuar, por bën optimizimin e gjithë set-it të rasteve të testimit drejt përmbushjes së gjithë qëllimeve në të njëjtën kohë (*whole test suite generation*). P.sh. për kriterin e mbulimit të degëve, funksioni i vlerësimit është i tillë që vlerëson gjithë setin e rasteve të testimit kundrejt kriterit të mbulimit **të të gjithë** degëve ne klasën që po testohet. Bazuar tek autorët e EvoSuite [77], kjo teknikë është më efektive se mënyra ku kërkohet të përbushet vetëm një qëllim në një kohë. Arsyeja është se disa qëllime mbulimi janë të pamundura ose më të vështira për t'u përmbushur dhe në një skenar testimi ku burimet janë të kufizuara, kërkesa për të përmbushur këto qëllime sjell një ndarje jo të barabartë të burimeve. Bazuar tek [61][77], gjenerimi i plotë i setit të rasteve të testimit arrin mbulim më të madh dhe gjeneron sete testimi më të vogla se gjenerimi i veçantë i rasteve të testimit.

EvoSuite, gjithashtu, gjeneron edhe pohime për të kapur sjelljen aktuale të klasave që po testohen.

Në EvoSuite janë implementuar disa lloje kritere mbulimi [78]. Ato janë: *Mbulimi i Rreshtave, Mbulimi i Degëve, Mbulimi Direkt i Degëve, Mbulimi i Output-it, Mutacion i Dobët, Mbulimi i Përjashtimeve, Mbulimi i Metodave në Nivel të Lartë, Mbulimi i Metodave në Nivel të Lartë pa Përjashtime.*

Gjithë kriteret e sipërpërmendura nuk kanë konflikt me njëra tjetrën (mund të shtohen raste të resja testimi në një set për të rritur mbulimin e një kriteri, pa ulur mbulimin e kritereve të tjera) e për rrjedhojë funksioni i vlerësimit eshtë një kombinim linear i gjithë funksioneve të veçanta të vlerësimit.

Kriteri "by default" që përdoret për gjenerim është mbulimi i degëve. Përdoruesi ka mundësi të zgjedhë një prej kritereve të tjera. Gjithashtu ai mund të zgjedhë edhe një kombinim të disa ose të gjithë kritereve.

3.3.2 Vlerësimi Eksperimental

Pyetjet për kërkim (PK) që kërkojmë ti japim përgjigje mbi kombinimin e disa kritereve të mbulimit janë:

- *PK1: Si ndikon kombinimi i disa kritereve në mbulimin e secilit kriter?*
- *PK2: Si ndikon kombinimi i gjithë kritereve tek rezultati i mutacionit i arritur nga seti i rastve të testimit?*
- *PK3: Cili prej kritereve (përveç Mutacionit të Dobët), i përdorur më vete arrin rezultatin e mutacionit më të lartë?*
- *PK4: Si ndikon numri i mutantëve të klasës që po testohet tek rezultati i mutacionit?*
- *PK5: Si ndikon kombinimi i disa kritereve në numrin dhe në gjatësinë e rasteve të testimit të setit të gjenenruar?*
- *PK6: Si ndikon shtimi i kriterit Mutacion i Dobët tek gjatësia e rasteve të testimit?*

3.3.3 Organizimi i Experimentit

A) Karakteristikat e Sistemit

Për eksperimentet është përdorur një kompjuter desktop me sistem operativ Linux me 32 bit, memorje kryesore 1GB dhe një proçesor Intel Core 2 Duo CPU E7400 2.8GHz x 2.

B) Përdorimi i EvoSuite

EvoSuite është përdorur nga rreshti i komandës (Plugin i Eclipse nuk përdoret në sistemin operativ Linux).

C) Përzgjedhja e Subjekteve për Testim

Përzgjedhja e klasave për testim është shumë e rëndësishme pasi përzgjedhja ndikon në rezultatet e përftuara nga eksperimentet. Për zhvillimin e eksperimenteve u përzgjodhën software që janë me kod të hapur. Përzgjodhëm 4 projekte me një total prej

367 klasa të testueshme. Projektet dhe adresat se nga janë shkarkuar listohen në tabelën 3.1.

Tre prej projekteve të përzgjedhura u morrën nga SourceForge [79], që sot është magazina më e madhe e projekteve me kod të hapur. Dy prej projekteve u përzgjodhën nga CodeCreator [80] dhe dy projektet e tjera janë pjesë e librarisë së Java-s.

Si pasojë e burimeve fizike të limituara në dispozicion dhe të kohës së konsiderueshme që nevojitet për ekzekutimin e gjithë eksperimenteve mbi një klasë, u zgjodhën në mënyrë të rastësishme 150 klasa nga projektet e listuara tek tabela 3.1. Për të tejkaluar faktin që rezultati që gjenerohet nga algoritmi gjenetik mbi një klasë është i rastësishëm secili eksperiment u përsërit 5 herë. Për 12 klasa EvoSuite nuk arrin të gjenerojë output. Nuk është qëllimi i këtij studimi të investigojë mbi shkaqet që cojnë në mosgjenerimin e asnjë output-i në disa raste. Gjithashtu është me rëndësi të përmendet këtu se rezultatet e përftuara varen në shumë raste nga "menaxhuesi i sigurisë" së EvoSuite, i cili kufizon ekzekutimin e setit të rasteve të testimit të gjeneruara për arsye sigurie. Ky menaxher ka për detyrë të ruajë sistemin ku po kryhen eksperimentet nga veprime të pasigurta si p.sh. aksesimi i sistemit të file-ve.

TABELA 3.1: EMRAT E PROJEKTEVE TË PËRZGJEDHURA PËR EKSPERIMENTET, NUMRI I KLASAVE QË ATO PËRMBAJNË, BURIMI NGA JANË SHKARKUAR

Emri i Projektit	***Numri i Klasave***	***Burimi***
MathPareser	48	SourceForge
MathQuickGame	25	SourceForge
java.util.Regex	92	jdk 1.8.0/src
Java.lang	69	jdk 1.8.0/src
Refactoring	87	SourceForge
Library	22	CodeCreator.org
StudentManagementSystem	24	CodeCreator.org
Total	***367***	

D) Parametrat e AG

Gjatë eksperimenteve vlerat e algoritmit gjenetik u lanë në vlerat default të vendosura nga autorët e EvoSuite. Për sa i përket buxhetit të kërkimit, ai u vendos në vlerat e 1 min ose 5 min në varësi të eksperimentit. Vlerat e parametrave më të rëndësishëm janë:

- Madhësia e popullatës: 50 sete rastesh testimi
- Gjatësia e kromozomemeve: 40 raste testimi
- Probabiliteti i mutacioneve: 0.75
- Probabiliteti i kryqëzimeve: 0.75

Përcaktimi i parametrave të AG për përftimin e rezultateve optimale është e vështirë. Një numër i madh studimesh i dedikohen kësaj çështjeje [56][58]

3.3.4 Rezultatet e Eksperimenteve

PK1: Si ndikon kombinimi i disa kritereve në mbulimin e secilit kriter?

Eksperimenti 1: Për secilën prej klasave u përdor EvoSuite me konfigurimet e mëposhtme:

1. *Gjithë kriteret e përzgjedhura me një buxhet kërkimi prej 1 min*
2. *Gjithë kriteret e përzgjedhura me një buxhet kërkimi prej 5 min*
3. *Secili kriter i përzgjedhur më vete me një buxhet kërkimi prej 5 min*

Rezultatet e eksperimenteve jepen në tabelën 3.2.

TABELA 3.2: REZULTATET E MBULIMIT PËR SECILIN KONFIGURIM, MESATARJA PËR GJITHË EKZEKUTIMET PËR SECILËN KLASË NËN TESTIM

Kriteri	Line	Branch	Exception	Weak Mutation	Output	Top-Level	Method No Exception	Direct Branch
TË GJITHA(1 min)	59.7	52.6	83.7	62.4	47.9	93.4	89.6	49.8
TË GJITHA (5 min)	60.7	53.4	83.9	69.7	48.2	93.8	90.7	53.1
Vetëm një (5 min)	61.1	53.5	83.9	74.3	48.6	81	80.5	52.8

Bazuar tek rezultatet mund të themi se për secilin kriter, nëse merret mesatarja për mbulimin e arritur për gjithë klasat, atëherë përdorimi i një buxheti kërkimi prej 5 min siguron një mbulim më të mirë se rasti kur përdoret një buxhet kërkimi prej 1 min. Ky rezultat është i pritshëm pasi individët përmirësohen gjatë kërkimit dhe një kohë më e gjatë kërkimi rezulton në zgjidhje më të mira.

Për sa i përket kritereve të mbullimit të: Rreshtave, Degëve, Output-it, Mutacionit të Dobët; përdorimi i kombinimit të të gjithë kritereve rezulton në një mbulim më të ulët sepse në këtë rast funksioni i vlerësimit është një kombinim i gjithë funksioneve të vlerësimit dhe si rrjedhojë ka për qëllim të mbulojë pjesë të ndryshme të programit (p.sh. Jo vetëm të mbulojë rreshtat). Ndryshimi më i madh në rezultatin e përftuar kur përdoret kombinimi i kritereve është në rastin e kriterit të Mutacionit të Dobët. Vihet re që për kriterin e mbulimit të Metodave në Nivel të Lartë pa Përjashtime dhe për kriterin e mbulimit të Metodave në Nivel të Lartë përdorimi i kombinimit të të gjithë kritereve sjell një rritje në mbulim, përkatësisht më 13% dhe me 12%. Arsyeja është se kërkimi në këto raste përfiton nga drejtimi që ofron funksioni i kombinuar i vlerësimit, pasi siç është përmendur më sipër, këto kritere nuk kanë mundësi të drejtojnë kërkimin.

PK1: Për kriteret të cilat kanë një funksion vlerësimi që drejton kërkimin, përformanca midis përdorimit të kombinimit të gjithë kritereve dhe përdorimit të një kriteri të vetëm konvergjon afërsisht drej të njëjtës vlerë nësë rritët buxheti i kërkimit. Për kriteret që nuk ofrojnë drejtim gjatë kërkimit, kombinimi i gjithë kritereve sjell një rritje të performancës.

PK2: Si ndikon kombinimi i gjithë kritereve tek rezultati i mutacionit i arritur nga seti i rastve të testimit?

PK3: Cili prej kritereve (përveç Mutacionit të Dobët), i përdorur më vete arrin rezultatin e mutacionit më të lartë?

Eksperimenti 2: Për secilën prej klasave u përdor EvoSuite me konfigurimet e mëposhtme:

1. *Gjithë kriteret e përzgjedhura me një buxhet kërkimi prej 5 min*
2. *Gjithë kriteret (përvec Mutacionit të Dobët) të përzgjedhura me një buxhet kërkimi prej 5 min*
3. *Secili kriter i përzgjedhur më vete me një buxhet kërkimi prej 5 min*

Për arsye se, si u shpjegua në paragrafin 3.2, Rezultati i Mutacionit është matësi që përdoret në kriterin më të fortë (Mutacioni i fortë), këtu është përdorur për të matur cilësinë e setit të gjeneruar. Vlerat e rezultatit mesatar të mutacionit të përftuar për secilin prej konfigurimeve jepen në tabelën 3.3.

TABELA 3.3: REZULTATET E MUTACIONIT PËR SECILIN KONFIGURIM, MESATARJA PËR GJITHË EKZEKUTIMET PËR SECILËN KLASË NËN TESTIM, ME BUXHET KËRKIMI 5 MIN

Kriteri	Line	Branch	Exception	Weak Mutation	Output	Top-Level	Method No Exception	Direct Branch	TË GJITHA (pa Weak Mutation)	TË GJITHA (5 min)
Rezultati i Mutacionit	15.6	25.8	0.2	28.3	16.6	23	15.3	21.2	24.5	26.1

Nga vlerat e përftuara mund të themi se kriteri që arrin rezultatin e mutacionit më të lartë është Mutacioni i Dobët. Kjo gjë pritej pasi qëllimi i këtij kriteri është të vrasë të gjithë mutantët. Përdorimi i kombinimit të gjithë kritereve arrin rezultatin e mutacionit më të mirë se përdorimi i secilit prej kritereve më vete (përvec Mutacionit të Dobët). Mbulimi i Përjashtimeve ka një rezultat mutacioni afërsisht të barabartë me zero. Bazuar tek rezultatet e mutacionit të përftuara në secilin prej konfigurimeve mund të themi se nëse nuk ka mundësi të përdoren të gjithë kriteret ose kriteri i Mutacionit të Dobët, atëherë duhet të përdoret kriteri i mbulimit të degëve, pasi është kriteri i dytë që arrin gjenerimin e set-ve të rasteve të testimit me cilësinë më të mirë.

Gjithashtu ajo që vihet re nga rezultatet e eksperimenteve është që setet e përftuara automatikisht janë akoma larg arritjes së një rezultati mutacioni të lartë, panvarësisht kriterit /kritereve të përzgjedhura.

PK2: Nëse ka një buxhet kërkimi të mjaftueshëm, kombinimi i të gjithë kritereve, arrin rezultate mutacioni më të larta se përdorimi i secilit kriter më vete (përvec kriterit të Mutacionit të Dobët).

PK3: Nga eksperimentet tona themi se kriteri i dytë më i mirë për të përftuar rezultate mutacioni të larta është kriteri i mbulimit të degëve.

PK4: Si ndikon numri i mutantëve të klasës që po testohet tek rezultati i mutacionit?

Eksperimenti 3: Për secilën prej klasave u përdor EvoSuite me konfigurimin e mëposhtëm:

1. Kriteri i mutacionit të dobët i përzgjedhur, me një buxhet kërkimi prej 5 min

Meqë kriteri që arrin rezultatin më të lartë të mutacionit nga kriteret e implementuara në EvoSuite është mutacioni i dobët, grafiku në figurën 3.3 jep varësinë e rezultatit të mutacionit kundrejt numrit të mutantëve për këtë kriter. Këto rezultate u përftuan nga eksperimenti 3.

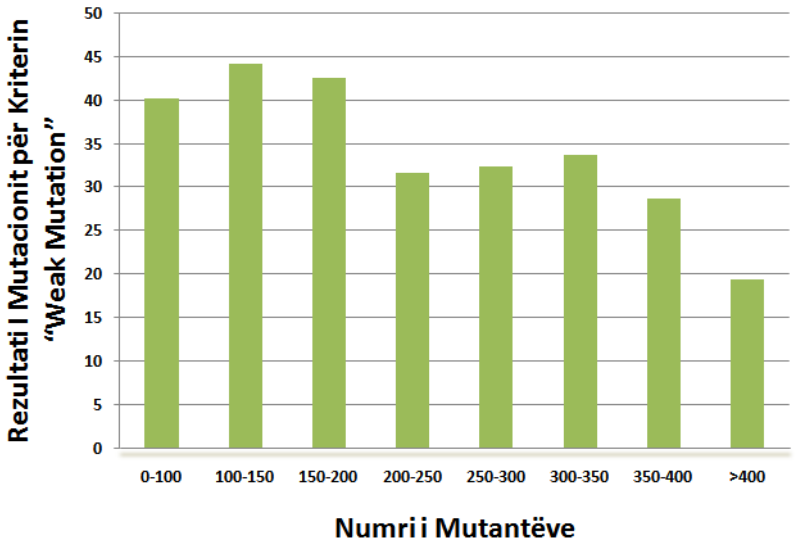

Figura 3.3: Varësia e rezultatit të mutacionit të arritur nga kriteri "Weak Mutation", kundrejt numrit të mutantëve

Si pasojë e faktit se numri i mutantëve nuk është i vetmi faktor që ndikon në rezultatin e mutacionit, nuk pritej një varësi e theksuar midis tyre në intervalet e ulëta,

por nëse numri i mutantëve është i lartë, ai qartazi ndikon në rezultatin e mutacionit të përftuar.

PK4: Duke patur një buxhet të kufizuar të kërkimit, atëherë numri i mutantëve në klasën që po testohet ndikon në rezultatin e mutacionit të përftuar.

PK5: Si ndikon kombinimi i disa kritereve në numrin dhe në gjatësinë e rasteve të testimit të setit të gjenenruar?

PK6: Si ndikon shtimi i kriterit të Mutacionit të Dobët tek gjatësia e rasteve të testimit?

Këtu me termin *madhësi e rastit të testimit*, i referohemi numrit të shprehjeve në një rast testimi pas fazës së minimizimit (pa përfshirë pohimet). Minimizimi ndodh automatikisht pas fazës së gjenerimit dhe rastet e testimit që ruhen në skedarin përkatës, janë ato të përftuara pas minimizimit.

Gjatësia e rasteve të testimit kërkohet të jetë sa më e vogël në mënyrë që puna e testuesve që do punojnë me to të jetë sa më e thjeshtë.

Eksperimenti 4: Për secilën prej klasave u përdor EvoSuite me konfigurimet e mëposhtme:

1. *Gjithë kriteret e përzgjedhura me një buxhet kërkimi prej 5 min*
2. *Gjithë kriteret (përvec Mutacionit të Dobët) të përzgjedhura me një buxhet kërkimi prej 5 min*
3. *Secili kriter i përzgjedhur më vete me një buxhet kërkimi prej 5 min*

Gjatë 35 ekzekutimeve të EvoSuite, faza e minimizimit kaloi kohën maksimale të përcaktuar duke mos arritur të kryejë minimizimin. Këto raste nuk u morrën në konsideratë. Nuk është qëllim i këtij punimi të investigojë mbi arsyet pse kjo fazë nuk përfundoi me sukses.

Numri i rasteve të testimit të gjeneruara nuk është i rëndësishëm në krahasim me gjatësinë e tyre, pasi gjenerimi i shumë rasteve të shkurtra testimi nuk është një problem për testuesin që duhet të detektojë gabimet.

Rezultatet janë paraqitur në tabelën 3.4. Nga rezultatet e përftuara duket qartë se përdorimi i kombinimit të të gjithë kritereve rezulton në sete me raste testimi që kanë më shumë shprehje krahasuar rastet te testimit në setet e gjeneruara nga përdorimi i secilit

kriter më vete. Ky është një disavantazh i konsiderueshëm i përdorimit të disa kritereve të kombinuara sepse rrit kompleksitetin e rasteve të testimit duke i bërë ato të vështira dhe mbase të papërdorshme nga testuesit.

TABELA 3.4 MADHËSIA E SET-IT PËR SECILIN KONFIGURIM, MESATARJA PËR GJITHË EKZEKUTIMET PËR SECILËN KLASË NËN TESTIM, ME BUXHET KËRKIMI 5 MIN

Kriteri	Line	Branch	Exception	Weak Mutation	Output	Top-Level	Method No Exception	Direct Branch	TË GJITHA (pa Weak Mutation)	TË GJITHA (5 min)
Mbulimi	61.1	53.5	83.9	74.3	48.6	81	80.5	52.8	59.7	69.3
Rezultati i Mutacionit	15.6	25.8	0.2	28.3	16.6	23	15.3	21.2	24.5	26.1
Nr. i testeve	12.3	12.9	6	14.2	7.5	12.4	11.7	10	14.2	15.1
Madhësia e testeve	14.2	15.7	6.7	19.4	10	10.3	12.4	10.2	22.3	27.8

PK5: Përdorimi i kombinimit të disa kritereve rrit madhësinë e setit të rasteve të testimit me më shumë se 50% krahasuar me madhësinë mesatare të setit të rasteve të testimit të përftuar kur përdoret secili kriter më vete.

PK6: Shtimi i kriterit të mutacionit të dobët rrit madhësinë e setit të rasteve të testimit afërsisht me 20%.

3.4 Testimi i Klasave në Java

3.4.1 Gjuhët e Orientuara nga Objekti

Modeli i orientuar nga objekti është krijuar si rrjedhojë e mendimit se është e natyrshme të specifikohet, projektohet edhe zhvillohet një software bazuar në aspektin e objekteve. Një supozim i tillë justifikohet nga fakti se programet kompjuterike modelojnë entitetet e botës reale së bashku me ndërveprimet midis tyre dhe njerëzit tentojnë ta

shikojnë ambjentin të bazuar në objekte. Më poshtë jepet klasifikimi i propozuar nga Wegner dhe Shriver [81] të cilët përcaktojnë tre nivele ku bën pjesë një gjuhë bazuar në karakteristikat e saj:

1. **Bazuar në objekt:** gjuhët që krijojnë mundësi për përcaktimin e objekteve, të cilët janë entitete të karakterizuara nga gjëndja dhe funksionalitetet për aksesim dhe modifikim të objektit.
2. **Bazuar në klasa:** gjuhët e bazuara në objekt që krijojnë mundësi për përcaktimin e klasave; klasat janë implementime të tipeve abstrakte të të dhënave dhe përcaktojnë template-in e objekteve të cilat janë instance të klasës.
3. **Të orientuara nga objekti:** gjuhët e bazuara në klasa që krijojnë mundësinë për përcaktimin në mënyrë inkrementuese të klasave dhë mbështesin polimorfizmin dhe lidhjen dinamike. Në këtë nivel bën pjesë Java.

Tiparet kryesore të gjuhëve të orientuara nga objekti janë:

1. **Abstraksioni i të dhënave:** ka të bëjë me mundësinë për të përcaktuar objekte si implementime të tipeve abstrakte të të dhënave.
2. **Enkapsulimi:** ka të bëjë me mundësinë për bashkëmbylljen e të dhënave që kanë lidhje midis tyre, rutinave dhe përcaktimeve në një entitet të vetëm.
3. **Fshehja e informacionit:** ka të bëjë me mundësinë që ka programuesi të përcaktojë nëse një tipar që është enkapsuluar brenda një moduli të mund të shihet/aksesohet nga modulet e tjera; në këtë mënyrë ndahen qartazi ndërfaqja e modulit dhe implementimi i tij.
4. **Trashëgimia:** lejon përcaktimin e tipeve abstrakte të të dhënave të cilat derivohen nga tipe ekzistuese abstrakte; tipet e reja shtojnë tipare të reja dhe mund të ndryshojnë disa tipare të prindit (ose prindërve nëse lejohet trashëgimia e shumfishtë).
5. **Polimorfizmi:** lejon entitetet në program ti referohen objekteve të tipeve të ndryshme (por me lidhje trashëgimie) gjatë ekzekutimit.
6. **Lidhja dinamike:** ka të bëjë me mundësinë e përcaktimit gjatë ekzekutimit të identitetit të një veprimi që është thirrur në një një entitet në mënyrë

polimorfike; rezultati i veprimit mbi një variabël polimorfik varet nga tipi aktual i objektit që variabli referon gjatë kohës që bëhet thirrja.

3.4.1.1 Objektet

Objektet janë baza e sistemeve të orientuara nga objekti. Ato përfaqësojnë entitetet që përbëjnë sistemin; ndërveprimet dhe karakteristikat e tyre përcaktojnë sjelljen e gjithë sistemit. Në mënyrë intuitive, një objekt mund të konsiderohet si një abstraksion që paraqet një entitet të botës reale ose një element konceptual. Objekti karakterizohet nga tre veçori:

1. **Gjendja:** gjendja e objektit karakterizohet nga vlerat që lidhen me atributet e tij. Atributet mund të jenë tipe primitive ose referenca në objekte të tjera. Gjendja e objekteve duhet të jetë e modifikueshme dhe të inspektohet vetëm nëpërmjet shërbimeve që objekti ofron.
2. **Shërbimet:** këto paraqesin funksionalitetet që objekti ofron. Këto shërbime quhen metoda dhe mund të përdoren për ndryshimin dhe inspektimin e gjendjes së objektit ku ato bëjnë pjesë.
3. **Identiteti:** është një veçori e brendshme e objekteve e cila siguron që dy instanca të një klase të jenë gjithmonë të dallueshme edhe nëse gjendjet e tyre janë identike.

Një objekt fillimisht krijohet; alokon burimet që i duhen dhe përcakton gjendjen fillestare. Më pas metodat e tij thirren për inspektim/modifikim të gjëndjes së tij. Pas thirrjes së një metode objekti kalon në një gjendje tjetër (jo detyrimisht e ndryshme nga gjendja e mëparshme).

3.4.1.2 Klasat

Gjuhët e orientuara nga objekti sigurojnë konstrukte specifike që i lejojnë programuesit të përcaktojë tipe abstrakte të dhënash. Përcaktimi i një klase nënkupton përcaktimin e një tipi të ri të dhënash. Klasa përcakton numrin dhe tipin e atributeve së bashku me veprimet e lejuara mbi këto attribute.

Klasat detyrojnë përdorimin e enkapsulimit dhe të fshehjes së informacionit. Klientët e një klase mjafton të njohin ndërfaqen e klasës (bashkësia e veprimeve që mund të thirren në këtë klasë) dhe nuk kanë nevojë të njohin mënyrën se si këto veprime janë implementuar. Fshehja e informacionit realizohet me ndarjen e pjesës private nga ajo publike (programuesit kanë në dispozicion modifikuesit e aksesit që i lejojnë të deklarojnë disa pjestarë të klasës si të aksesueshëm vetëm nga objekte të kësaj klase, ndërsa pjestarë të tjerë si të aksesueshëm publikisht). Përgjithësisht ekzistojnë disa nivele aksesueshmërie për pjestarët e klasës (jo vetëm private dhe publike).

Klasat janë implementime të tipeve abstrakte të të dhënave, ndërsa metodat janë veprime të këtyre tipeve abstrakte të të dhënave. Metodat karakterizohen nga firma e tyre. Metodat klasifikohen në disa kategori të ndryshme në varësi të qëllimit të tyre:

1. **Konstruktorë:** këto sigurojnë një instancë specifike të klasës ku bëjnë pjesë. Ka disa tipe konstruktorësh: konstruktorët default që nuk marrin asnjë argument dhe vendosen automatikisht nga kompilatori nëse programuesi nuk ka përcaktuar asnjë konstruktor për klasën; konstruktorët e tjerë që përcaktohen nga programuesi dhe që kanë një numër variabël argumentash dhe këto argumenta përdoren për inicializimin e atributeve të objektit të ri të formuar.
2. **Observues:** këto metoda i sigurojnë thirrësit informacion mbi gjendjen e objektit mbi të cilin po thirren. Këto metoda nuk ndryshojnë gjendjen e objektit.
3. **Modifikues:** këto metoda kanë për qëllim ti mundësojnë thirrësit modifikimin e gjendjes së objektit mbi të cilin thirren duke ndryshuar vlerat e atributeve të tij (një ose disa prej tyre).
4. **Destruktorët** lirojnë burimet që nuk nevojiten më. Në Java, duke qenë se ekziston “garbage collector”, nuk ka destruktorë.

Duhet theksuar se një metodë mund të jetë edhe observues edhe modifikues, nëse lejon inspektimin dhe modifikimin e gjendjes së objektit.

3.4.1.3 Atributet

Atributet, ose variablat e instancës, së bashku me vlerat e tyre përcaktojnë gjendjen e objektit. Për të detyruar fshehjen e informacionit, atributet nuk duhet të jenë asnjëherë të aksesueshme direkt nga klasat e tjera. Cdo klasë, nëse është nevoja, duhet të ketë metoda për inspektimin dhe ndryshimin e atributeve. Atributet mund të jenë të tipeve primitive ose gjithashtu të tipit klasë (referencë në një objekt tjetër). Në këtë rast gjendja e objektit jepet nga vlerat e atributeve primitive dhe gjendja e objekteve që ai referencon.

3.4.2 Testimi në Nivel Njësie i Klasave

Gjatë testimit të njësive, çdo klasë do të konsiderohet si e izoluar. Zakonisht klasat që kërkohen nga klasa që po testohet janë të gatshme. Nëse jo, ato zëvendësohen nga "stubs" ose "mocks" [70].

Në rastin e gjuhëve të bazuara në proçedura, një rast testimi përmban vetëm disa vlera konkrete; p.sh. $(v_1, v_2 \ldots v_N)$. Për sa i përket Java-s, ose në përgjithësi gjuhëve të orientuara nga objekti, një rast testimi përmban krahas vlerave konkrete edhe ndërtime objektesh dhe thirrje metodash.

Proçedura tipike që ndiqet për testimin në nivel njësie të klasave, përbëhet nga katër hapa të cilat janë:

1. Krijimi i një objekti të klasës që po testohet duke përdorur një prej konstruktorëve që përmban klasa.
2. Një sekuencë prej zero ose më shumë metodash thirren mbi këtë objekt që të arrihet gjendja e kërkuar e tij.
3. Metoda që kërkohet të testohet thirret.
4. Gjendja përfundimtare e arritur nga objekti, analizohet për të aksesuar rezultatin e rastit të testimit.

Kjo proçedurë mund të aplikohet në mënyrë funksionale (black-box testing), duke marrë rezultatet e pritura nga specifikimet e klasës. Një tjetër mundësi është të vlerësohet saktësia e testimit nëpërmjet kritereve të mbulimit. Hapat e përmendura më sipër përsëriten deri sa të arrihet niveli i pranueshëm i mbulimit.

Vihet re se në tre hapat e para, nevojiten parametra për të thirrur si konstruktorin, metodat për ndryshimin e gjendjes, ose metodën që po testohet. Nëse disa prej këtyre parametrave janë objekte, atëherë edhe këto objekte duhet të krijohen dhe të vendosen në gjendjen e duhur duke përsëritur në mënyrë rekursive tre hapat e para, deri sa gjithë objektet të jenë të gatshme. Pra, një rast testimi do të jetë një sekuencë krijimesh të objekteve (objekti i klasës që po testohet + objekte që duhen si parametra), thirrje metodash (për të sjellë objektet në gjendjen e kërkuar) dhe thirrja përfundimtare e metodës që po testohet.

Për shembull supozojmë se po testohet metoda `m`, e klasës `T`. Supozojmë se metoda `m` merr si argument një numër të plotë dhe një objekt të tipit klasë `A`. Një rast testimi për këtë shembull do të ishte:

```
T t = new T(); //krijimi i objektit të klasës nën testim
A a = new A(); //krijimi i një objekti që duhet si parametër
a.s(6.2);      //thirrja e metodës për të vendosur objektin në
               //gjendjen e kërkuar
t.m(5, a);     //thirrja e metodës që po testohet
```

Në këtë rast, krijimi i objektit `a` është i domosdoshëm, pasi parametri i dytë i metodës `m` është një objekt i klasës `A`. thirrja e metodës `s` nga objekti `a`, synon ndryshimin e gjendjes së këtij objekti para se ti jepet si argument metodës `m`. Sekuenca e inputeve në këtë rast do të ishte `<6.2, 5>`. Në fund, duhet të shtohet edhe një pohim mbi gjendjen e objektit `t` pas thirrjes së metodës `m`, në mënyrë që rasti i testimit të jetë i plotë.

Karakteristikat e rasteve të testimit në gjuhët e orientuara nga objekti mund të përmblidhen si më poshtë:

- Numri dhe renditja e thirrjes së metodave është variabël.
- Numri i vlerave të inputit është gjithashtu variabël.
- Disa parametra në thirrjen e metodave janë objekte, për rrjedhojë kërkojnë ndërtime të mëtejshme objektesh.

- Gjëndja e objektit që po testohet dhe e objekteve që kalohen si parametra, ndikon tek rezultati i përftuar.

Si përfundim, mund të themi se për testimin e njësive në Java, rastet e testimit nuk do të jenë vetëm sekuenca vlerash inputesh si në rastin e gjuhëve të bazuara në proçedura.

3.5 Implementimi i Algoritmit Gjenetik

3.5.1 Ndërtimi i mjetit

Bllok-skema për gjenerimin e rasteve të testimit duke përdorur algoritmat gjenetikë jepet në figurën 3.4.

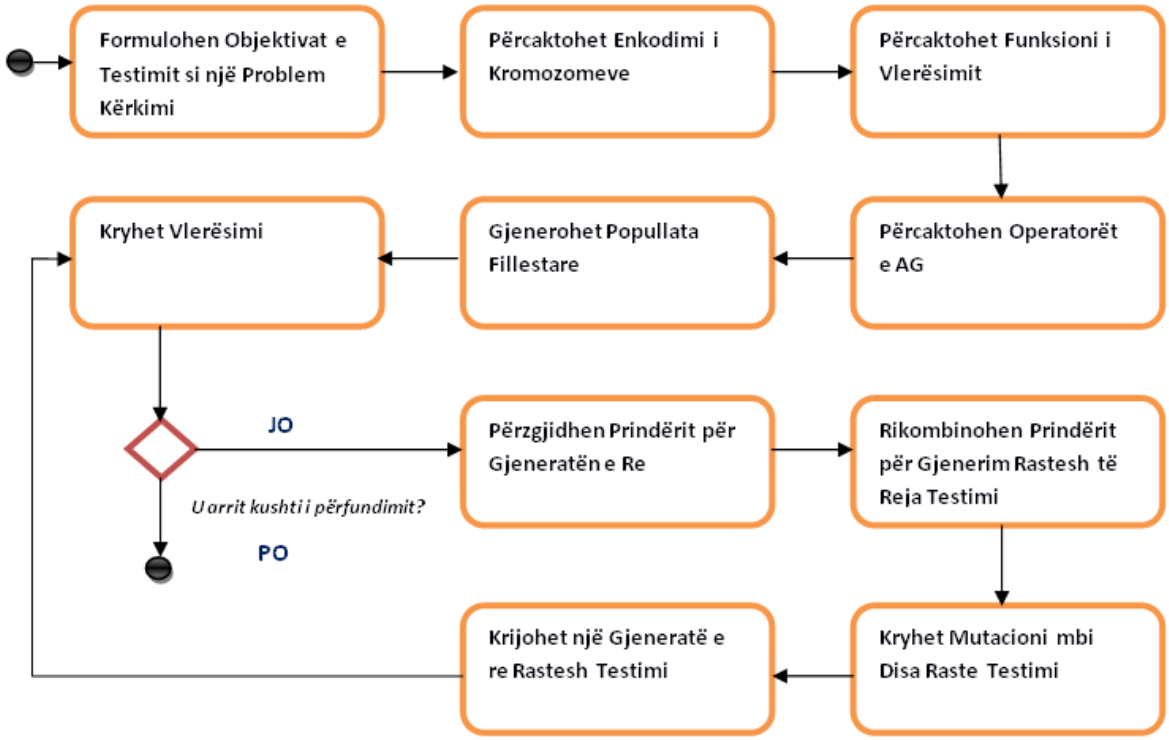

Figura 3.4: Sekuenca e hapave për gjenerimin e rasteve të testimit duke përdorur AG [82]

Algoritmi gjenetik që do të përdoret është ai i propozuar nga [1] dhe paraqitet në figurën 3.5. Në këtë paragraf do të shpjegohen implementimet e secilit prej hapave në mjetin eToc, që do të përdoret për ndërtimin e funksionit të propozuar në këtë punim.

Të dhënat e inputit gjenerohen që të përmbushin një kriter të caktuar (në këtë rast do të përdoret mbulimi i degëve). Bashkësia e degëve që duhet të mbulohen përcaktohet në rreshtin e dytë. Më pas, një set rastesh testimi gjenerohet në mënyrë të rastësishme (popullataTanishme). Mënyra se si do të paraqitet një rast testimi jepet në paragrafin në vijim. Si u shpjegua në paragrafin 3.3.2, një rast testimi është një sekuencë ndërtimi objektesh dhe thirrje metodash.

```
gjenerimRateshTestimi(klasaNënTestim: Klasa)
1  qëllimePërMbulim ← qëllime(klasaNënTestim)
2  popullataTanishme ← gjeneroPopullatëRastësishme(madhësiaPopullatës)
3  while qëllimePërMbulim != 0 and kohaEkzekutimit()<kohaMaxEkzekutim
4     t ← zgjidhQëllim(qëllimePërMbulim), tentativa ← 0
5     while not mbuluar(t) and përpjekje < maxPërpjekje
6         ekzekuto rastet tek popullataTanishme
7         update qëllimePërMbulim
8         if mbuluar(t) break
9         llogarit fitness[t] për rastet tek popullataTanishme
10        nxirr popullataRe nga popullataTanishme bazuar tek fitness[t]
11        kryej muatacione tek popullataRe
12        popullataTanishme ← popullataRe
13        përpjekje ← përpjekje + 1
14    end while
15 end while
```

Figura 3.5: AG për gjenerimin e rasteve të testimit bazuar tek nivelet e mbulimit [1]

Algoritmi gjeneron raste të tjera testimi për sa kohë ka akoma degë të pambuluara, ose është arritur koha maksimale e ekzekutimit, kohaMaxExekutim (çdo algoritëm gjenetik ka si parametër një kohë limit ekzekutimi, pasi nuk ka siguri që qëllimi i kërkimit të përmbushet dhe për rrjedhojë ekzekutimi do të jetë i pafundëm). Në këtë rast qëllimet (targets) për t'u përmbushur janë degët që duhet të mbulohen. Një qëllim është diçka që nuk është mbuluar akoma dhe përmbushja e këtij qëllimi do të rriste të paktën një kriter mbulimi.

Në rreshtin 4, zgjidhet një degë për t'u mbuluar dhe nga popullata aktuale do të bëhen një numër i caktuar (maxPërpjekje) gjenerimesh të njëpasnjëshme. Ekziston një numër maksimal përpjekjesh për arsye se disa degë (në përgjithësi disa target) mund të jenë të pamundura për t'u mbuluar dhe duhet të këtë një limit në numrin e tentativave për

mbulim, pasi përndryshe do të harxhoheshin vetëm për këtë degë një pjesë e konsiderueshme e burimeve të kufizuara që ka testimi.

Për sa kohë dega e përzgjedhur, nuk është mbuluar dhe nuk është arritur numri maksimal i përpjekjeve, ekzekutohen gjithë rastet e testimit në popullatën aktuale (rreshti 6) dhe pas ekzekutimit përcaktohen kush janë qëllimet e pambuluara (qëllimePërMbulim) dhe rifreskimi ndodh në rreshtin 7. Nëse dega e përzgjedhur u mbulua, dilet nga cikli i brendëshëm (rreshti 8) dhe zgjidhet një degë e re. Nëse qëllimi nuk është mbuluar, me anë të funksionit të vlerësimit përcaktohet cilësia e secilit prej rasteve të testimit (rreshti 9). Një popullatë e re del nga popullata e vjetër (rreshti 10); probabiliteti për t'u përzgjedhur është më i lartë për individët që kanë cilësi më të mira në lidhje më targetin e përzgjedhur. Më pas, veprimi i mutacionit kryhet mbi popullatën e re (rreshti 11). Numri i përpjekjeve për këtë target rritet me 1 dhe ri-testohet kushti i ciklit të brendshëm.

Gjatë ekzekutimit, sa herë që një target i pambuluar mbulohet nga një rast testimi, ky rast ruhet sepse është një prej rasteve që do të nevojiten për të arritur nivelin përfundimtar të mbulimit. Për rrjedhojë, rezultati i algoritmit është seti i rasteve të testimit që mbulojnë e pakta një target. Një set i tillë mund të ketë raste testimi të tepërta, pasi rastet e shtuara së fundmi mund të mbulojnë qëllime që më parë ishin mbuluar nga raste të tjera. Është e drejtë të pranohet se: nëse gjithë objektivat që mbulohen nga një rast testimi, mbulohen edhe nga rastet e tjera që janë shtuar më vonë, atëherë ky rast testimi është i tepërt dhe duhet të hiqet nga seti pasi nuk jep kontribut në mbulimin e përftuar. Për të zgjidhur këtë situatë duhet që pas gjenerimit të setit (përfundimit të algoritmit gjenetik), të bëhet një proçesim i setit me qëllim minimizimin e tij.

Dy zgjedhjet më kritike për algoritmin e dhënë në figurën 3.4, janë funksioni i vlerësimit dhe operatorët e mutacionit. Këto janë dy faktorët që drejtojnë evolucionin e popullatës aktuale në një popullatë të re e cila duhet të ketë probabilitet më të madh për të mbuluar qëllimin e përcaktuar. Funksioni i vlerësimit përcakton probabilitetin që një individ të mbijetojë dhe të marrë pjesë në një formë më të evoluar në popullatën e re, ndërsa operatorët e mutacionit përcaktojnë se si individët e rinj gjenerohen nga individët e vjetër.

Parametrat e algoritmit gjenetik te dhënë në figurën 3.5 janë: koha maksimale e ekzekutimit (kohaMaxEkzekutim), numri maksimal i përpjekjeve për secilin target (maxPërpjekje) dhe madhësia e popullatës (madhësiaPopullatës). Këtyre parametrave i vendosen vlera default, por mund të ndryshohen edhe nga përdoruesi nëse niveli i mbulimit të arritur nuk eshte i kënaqshëm.

3.5.2 Enkodimi i Rasteve të Testimit (Kromozomet)

Struktura e kromozomeve në rastin e gjuhëve të bazuara në proçedura, është e thjeshtë dhe konsiston vetëm në një sekuencë vlerash inputi që duhet ti jepen programit gjatë ekzekutimit. Në të kundërt, një rast testimi që do të përdoret për testimin e njësive të një klase nuk është vetëm një sekuencë vlerash (çështja 3.3.2). Zakonisht metoda `main` nuk ekziston fare në një klasë që po testohet dhe po konsiderohet e izoluar. Klasa duhet të testohet duhet ju referuar ndërfaqes së saj të konstruktorëve dhe metodave dhe pa asnjë njohuri shtesë mbi sekuencën e thirrjeve të metodave që do të ndodhë në aplikacionin final që do e përdorë atë. Pra, një rast testimi është një sekuencë thirrjesh konstruktorësh, metodash duke përfshirë dhe parametrat e tyre. Pohimet mbi gjendjen e pritur pas ekzekutimit e bëjnë rastin e testimit të plotë. Koromozomet që do të përdoren për testimin evolutiv të klasave duhet të përcaktojnë edhe sekuencën e veprimeve që duhet të kryhen edhe vlerat përkatëse të parametrave.

Për të përcaktuar sintaksën e kromozomeve do të përdoret përcaktimi i bërë nga Tonella [1]. Ky përcaktim jepet në figurën 3.6.

```
<kromozomi>   ::=<veprime> @ <vlera>
<veprime>     ::= <veprim>   {: <veprime>}?
<veprim>      ::= $id = konstruktor ({<parametra>}?)
                |   $id = klasë # null
                |   $id = metodë ({<parametra>}?)
<parametrat>  ::= <parametër>   {, <parametra>}?
<parametër>   ::= tip-builtin {<gjenerues>}?
                |    $id
<gjenerues>   ::=      [poshtë; lartë]
```

```
                |   [klasëGjen]
<vlera>         ::= <vlerë>  {, <vlera>}?
<vlerë>         ::= integer
                |   real
                |   bloolean
                |   string
```

Figura 3.6: Sintaksa e Kromozomeve

Në figurën 3.6, kllapat gjarpëruese dhe pikëpyetjet janë meta-karaktere që përdoren për zgjerime opsionale, ndërsa jo-terminalet janë të paraqitura me kllapa këndore.

Një kromozom ndahet në dy pjesë, që ndahen nga karakteri '@'. Pjesa e parë (jo-terminali <veprime>), përmban një sekuencë konstruktorësh dhe thirrje metodash të ndara me ':'. Pjesa e dytë përmban vlerat aktuale të inputit që ndahen me presje. Këto vlera do të përdoren gjatë thirrjeve dhe korrespondojnë me kromozomet e përdorura në testimin evolutiv të programeve proçeduriale.

Secili <veprim> mund të ndërtojë një objekt, që i jepet si referencë një variabël kromozomi ($id), ose mund të thërresë një metodë mbi një objekt të identifikuar me $id. Një rast i veçantë i vlerëdhënies së objekteve përfshin vlerën `null`, e cila korrespondon në një referencë që nuk shenjon në asnjë objekt.

Parametrat e thirrjeve të metodave ose konstruktorëve (<parametrat>), janë tipe built-in si p.sh. int, double, booelan ose variabla kromozomesh ($id). Klasa String konsiderohet si tip built-in, pasi vlerat mund të gjenerohen edhe pa qenë nevojë e thirrjes në mënyrë eksplicite të konstruktorit. Metoda për gjenerim mund të zgjedhë në mënyrë të rastësishme një numër brenda një intervali *[poshtë, lartë]*. Në mungesë të një specifikimi për gjeneruesin, përdoret gjeneratori by default.

Jo të gjithë kromozomet e gjeneruara sipas rregullave të figurës 3.6 janë të formuara saktë. Një kromozom do të konsiderohet i mirë-formuar nëse:

1. Variablave të kromozomit ($id) i është dhënë një vlerë para se të përdoren (si parametra ose referenca për thirrje metodash)
2. Për çdo tip built-in tek pjesa e veprimeve, ka një vlerë inputi tek pjesa e dytë që i korrespondon këtij tipi tek pozicioni korrespondues.

Gjithashtu, një kromozom paraqet një sekuencë aktuale ekzekutimi për klasën që po testohet nëse gjithë metodat dhe konstruktorët që përbëjnë kromozomin ekzistojnë dhe kanë një firmë që përputhet me tipet e përcaktuara në kromozom. Për shembull kromozomi:

```
$a = A() : $f = F() : $a.m(int, $f) @ 7
```

është i mirë-formuar dhe paraqet një sekuencë ekzekutimi aktuale nëse klasat A dhe `F` kanë një konstruktor pa parametra dhe në klasën `A` ekziston një metodë `m`, ku parametri i parë i të cilës është i tipit `int` dhe parametri i dytë është objekt i tipit klasë `F` (ose nënklasë e F-së). Vlera `7` është gjeneruar nga gjeneruesi default për vlerat integer.

Një tjetër rast kromozomi do të ishte si në figurën 3.7 [83]. Në figurë jepet gjithashtu edhe rasti i testimit për ekzekutim i shkruar në JUnit.

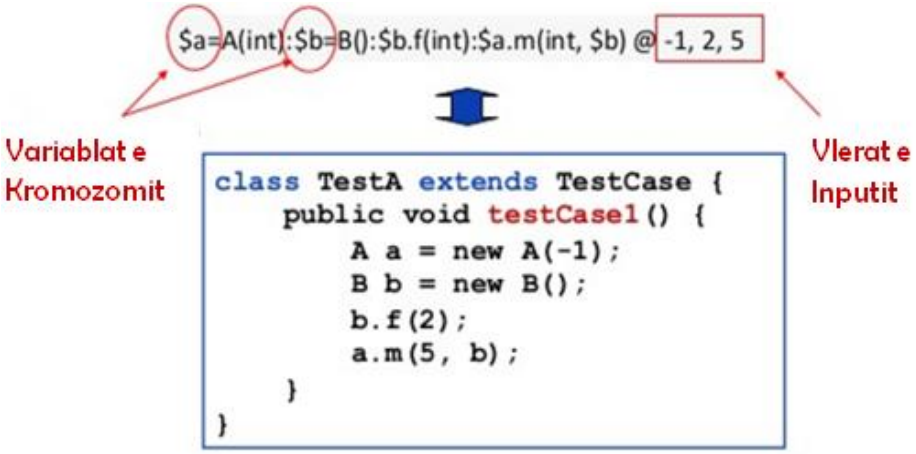

Figura 3.7: Paraqitja e kromozomeve në rastin e testimit të klasave në Java dhe përkthimi në rast testimi JUnit [83]

Hapat për ndërtimin e kromozomit janë:

1. Ndërtohet një objekt i klasës që është nën testim duke zgjedhur rastësisht njërin prej konstruktorëve të klasës:
   `$a = A(int) @ -1`
2. Shtohet thirrja e metodës që është nën testim:
   `$a = A(int) : $a.m (int, $b) @ -1, 5`
3. Shtohen gjithë ndërtimet e kërkuara të objekteve:

```
$a = A(int) : $b = B() : $a.m (int, $b) @ -1, 5
```

4. Shtohen në mënyrë të rastësishme thirrje metodash për të ndryshuar gjendjen e objekteve të krijuara:

```
$a = A(int) : $b = B() : $b.f (int) : $a.m (int, $b)
@ -1, 2, 5
```

Hapat 3 dhe 4 përsëriten deri sa gjithë variablat e kromozomit që përdoren si parametra të konstruktorëve ose metodave janë të inicializuara saktë (kromozom i mirë-formuar).

Gjenerimi i të dhënave për input bëhët në mënyrë të rastësishme duke përdorur një prej 4 gjeneruesve:

- Default:
  - `A.m(int)`: përzgjedhje e nje numri int default
  - `A.m(boolean)`: përzgjedhje e true ose false
  - `A.m(String)`: zgjidhen karakteret në mënyrë uniforme nga [a-zA-Z0-9], dhe gjatësia e stringës pas zgjedhjes së çdo karakteri vjen duke u zvogëluar
- I parametrizuar
  - `A.m(int[-2;2])`: zgjidhet një int midis dy parametrave [-2;2]
- Përzgjedhje nga një bashkësi vlerash
  - `A.m(int<bashkësi>)`: zgjidhet një int nga bashkësia e vlerave int të programit
- I përshtatur
  - `A.m(int[MyIntGenerator])`: thirret metoda newIntValue për gjenerimin e numrave int nga klasa MyIntGenerator

3.5.3 Operatorët e Mutacionit

Kromozomet transformohen duke zgjedhur rastësisht gjatë kërkimit operatorë mutacioni. Operatorët e mutacionit për testimin evolutiv të klasave në Java janë:

1. **Mutacioni i një prej vlerave të inputit.** Një vlerë inputi zëvendësohet nga një vlerë tjetër e të njëjtit tip.
2. **Ndryshimi i konstruktorëvë.** Një prej konstruktorëve (që ndërton klasën nën testim ose ndërton objekte të tjera) zëvendësohet me një konstruktor të

mbingarkuar të së njëjtës klasë. Vlerat ose objektet që nevojiteshin para mutacionit dhe që nuk përdoren më, fshihen, gjithashtu shtohen vlera ose objekte të reja që kërkohen për kostruktorin e ri.

3. **Shtimi/Heqja e thirrjeve të metodave.** Shtohen thirrje të reja metodash. Gjithashtu shtohen vlera ose objekte që nevojiten si parametra. I njëjti shpjegim vlen edhe për heqjen e metodave; zgjidhet në mënyrë të rastësishme një metodë për t'u hequr, hiqen vlerat që kjo metodë përdorte si parametra dhe hiqet gjithashtu dhe variabli i kromozomit që përdorej vetëm për thirrjen e kësaj metode (nëse ka). Për shembull në figurën 3.8 [83], shtimi i thirrjes së metodës f(int) nga objekti b, sjell dhe shtimin e vlerës 2. Ky operator përdoret në mënyrë të përsëritur; probabiliteti i shtimit/heqjes së metodave vjen duke u ulur në varësi të numrit të metodave të shtuara/hequra.

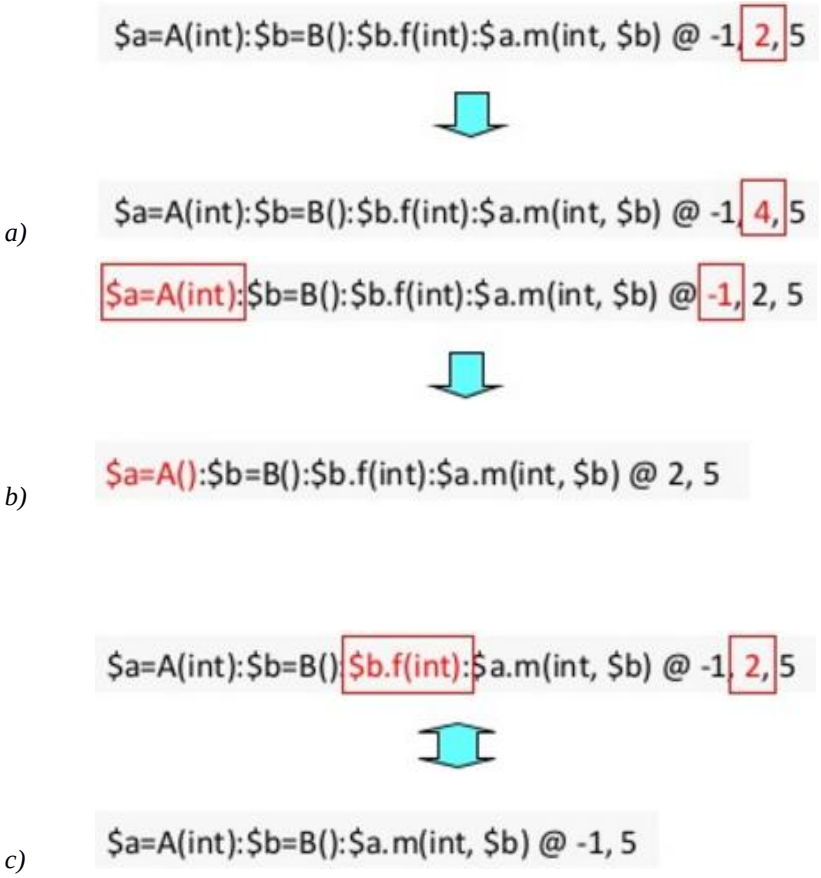

Figura 3.8: Operatorët e mutacionit. a) Mutacioni i një prej vlerave të inputit, b) Ndryshimi i konstruktorëve, c) Shtim/Heqje e thirrjeve të metodave [83]

3.5.4 Kryqëzimi në një pikë

Dy kromozome do të bashkohen pasi do të priten në një pikë të zgjedhur rastësisht. Kjo pikë duhet të jetë **pas** ndërtimit të objektit të klasës që është nën testim dhe **para** thirrjes së metodës së fundit. Është e domosdoshme që pas bashkimit, thirrje të panevojshme konstruktorësh/metodash të hiqen dhe nëse ka konflikt midis variablave, ky konflikt të zgjidhet duke bërë një ri-emërtim të tyre. Kryqëzimi mund të konsiderohet edhe si mutacion i dy kromozomeve. Në figurën 3.9, pjesa pas pikës së prerjes për të dy kromozomet, është vendosur brenda kuadratit. Vihet re që thirrja e konstruktorit të klasës C, edhe pse ka mbetur tek kromozomi i parë duhet të hiqet prej tij edhe të vendoset tek kromozomi i dytë (tek i pari është i panevojshëm, tek i dyti është i domosdoshëm).

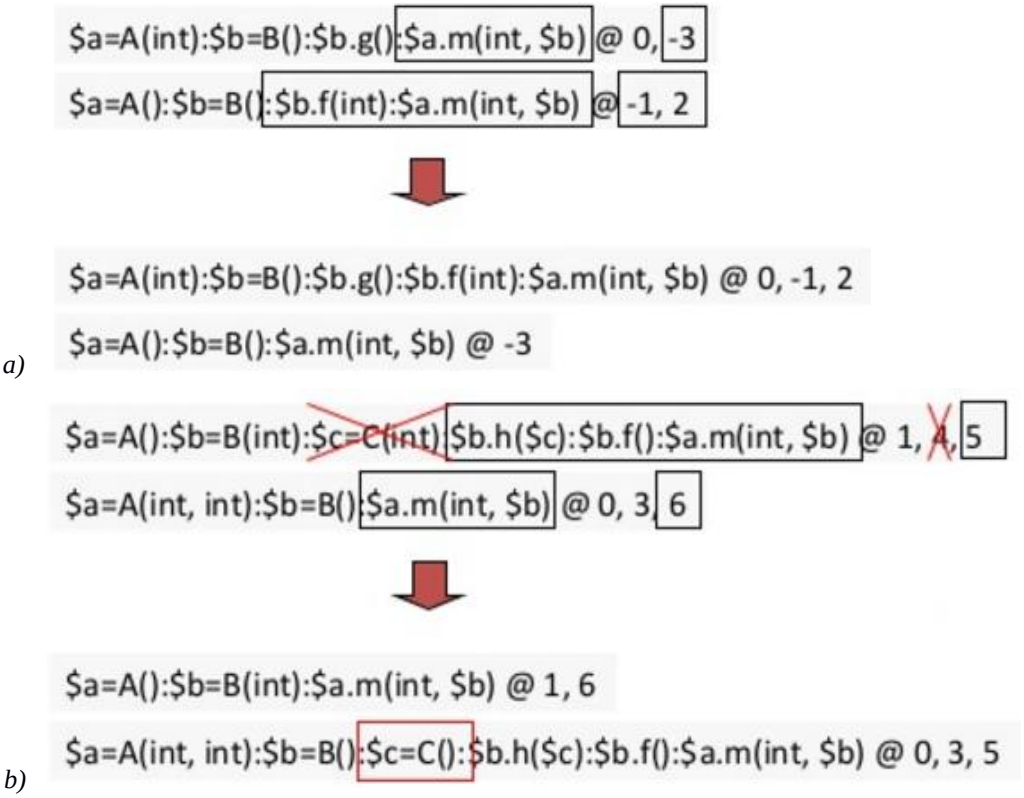

Figura 3.9: a) Kromozome të mirë-formuara pas kryqëzimit, b) Kromozomet kanë nevojë për modifikime pas kryqëzimit [59]

3.6 Ilustrimi i një Rasti ku Mbulimi i Degëve nuk Mjafton

Kriteri i mbulimit të degëve është kriteri më i përdorur sot për gjenerimin automatik te testeve. Gjithashtu duke u bazuar tek eksperimentet e shpjeguara në çështjen 3.3.3, u vërtetua se kriteri që arrin rezultatin e mutacionit më të lartë (d.m.th. kriteri që gjeneron teste me cilësi më të lartë), dhe që mund të implementohet realisht, është kriteri i mbulimit të degëve. Megjithatë, rezultati i mutacionit të arritur është më pak se 30%. Panvarësisht se, në këtë rezultat ndikojnë edhe mutantët ekuivalentë, prapë mund të themi se rezultati i arritur, duke përdorur vetëm kriterin e mbulimit të degëve nuk është shumë i mirë. Në këtë çështje do të ilustrohet konkretisht një rast ku arrihet një mbulim degësh 100%, por jo një rezultat mutacioni i lartë. Qëllimi është të kuptohen arsyet se pse një mbulim i plotë degësh nuk siguron një cilësi të lartë të rasteve të gjeneruara.

Supozojmë se kemi nje klasë të tillë në Java:

```
public class Staku {
            private int madhesia = 0;
            private int st [] = new int [4];
            void push (int x){
                  if (madhesia < st.length)
                        st[madhesia++] = x;
            }
            int pop (){
                  return st[madhesia--];
            }
   }
```

Klasa Staku është shkruar qëllimisht me gabime. Nëse do të përdorim EvoSuite për gjenerimin e rasteve të testimit për një klasë të thjeshtë si klasa Stack (8 rreshta kod, 2 metoda, 2 atribute), duke përdorur si kriter mbulimin e degëve, rastet e gjeneruara automatikisht do të ishin:

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.RunWith;
```

```
public class Staku_ESTest {
      @Test
      public void test0()  throws Throwable  {
                  Staku she0 = new Staku();
                  she0.push(1);
                  she0.push(0);
                  int i = she0.pop();
                  assertEquals(0, i);
                  she0.push(0);
                  she0.push(0);
                  she0.push((-1916));
                  she0.push((-1916));
            }
}
```

Analiza e mbulimit, e cila përftohet automatikisht nga EvoSuite, në këtë rast do të ishte:

Qëllime mbulimi: 4

Qëllime të mbuluara: 4

Mbulimi i degëve: 100%

U gjenerua 1 test me gjatësi 8

Rezultati i mutacionit: 29%

Duhet theksuar se për procesin e kërkimit u përcaktua një buxhet kërkimi prej 200s (dhe u harxhuan vetëm 3s, pasi klasa Staku është shumë e thjeshtë. U zgjodh një buxhet kërkimi relativisht i madh (200s) për arsye se qëllimi i këtij eksperimenti është studimi i cilësisë së testeve të gjeneruara për kriterin e mbulimit të degëve dhe do të ishte e pa saktë nëse kërkimi do të penalizohej nga koha e limituar.

Duke përdorur EclEmma (si plugin në Eclipse) [72], për të përftuar mbulimin e përftuar nga ky set rastesh testimi, rezultati do të ishte si në figurën 3.10. Ngjyra jeshile tregon se një shprehje është ekzekutuar, ndërsa rombi jeshil përpara kushtit `if`, tregon se të dyja degët e nyjes janë ekzekutuar. Dy metodat push dhe pop, janë ekzekutuar.

Vëme re se përftojmë një mbulim 100% të degëve. Testi i përftuar kalon me sukses. Nëse ekzekutohet testi, rezultatet e përftuara janë ato që priteshin. Meqë mbulimi është i plotë dhe rezultatet përputhen, atëherë duhet të themi se testimi eshtë kryer!

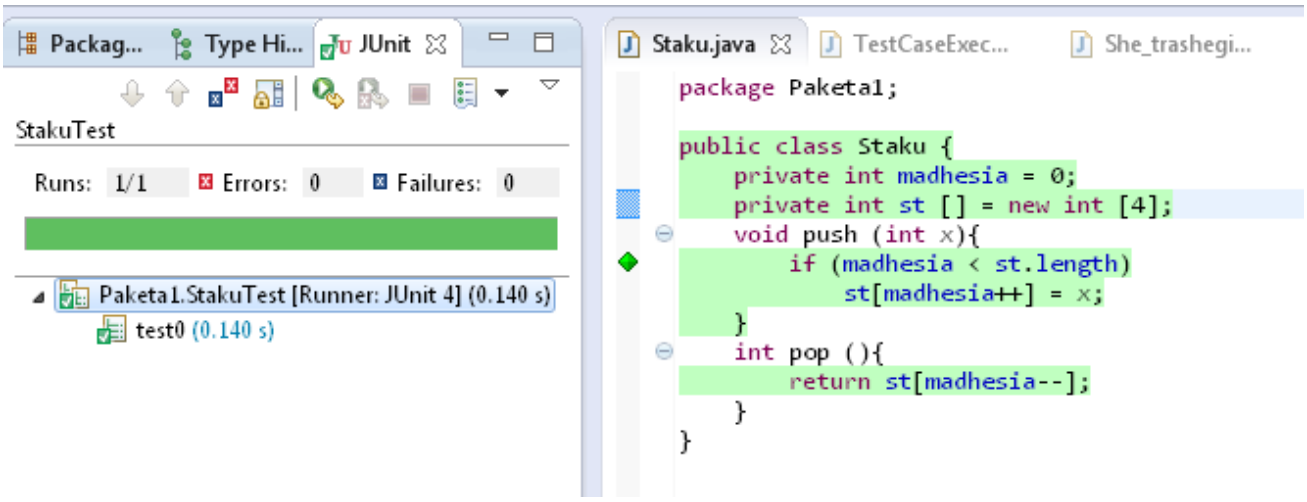

Figura 3.10: Mbulimi i përftuar pas ekzekutimit të testeve të gjeneruara për mbulimin e degëve

A mjafton ky kriter për këtë klasë? A ka raste testimi që mund të gjejnë gabime tek klasa Staku? Nëse vëmë re kodin e klasës, duket qartë se metoda pop(), nuk mund të thirret e para dhe pastaj të thirret metoda push(int), pasi do të gjenerohej një përjashtim sepse do të tejkaloheshin kufijtë e matricës.

Një rast tesimi për të vënë re këtë gabim do të ishte:

```
public void test1()  throws Throwable  {
            Staku staku0 = new Staku();
            int int0 = staku0.pop();
            staku0.push(15);
      }
```

Ekzekutojmë rastin test1(). Ky rast dështon dhe afishohet mesazhi:

```
java.lang.ArrayIndexOutOfBoundsException: -1
      at Staku.push(Staku.java:7)
      at Staku_ESTest.test2(Staku_ESTest.java:44)
```

Gjithashtu vëmë re se metoda pop(), kthen një vlerë dhe dekrementon me një fushën madhësia, pa bërë një herë një kontroll mbi gjendjen aktuale të stack-ut (nëse vlera e fushës `madhësia` është më e madhe se 0). Një rast tesimi për të vënë re këtë gabim do të ishte:

```
public void test2()  throws Throwable  {
      Staku staku0 = new Staku();
      staku0.pop();
      staku0.pop();
}
```

Ekzekutojmë rastin test2(). Ky rast dështon dhe afishohet mesazhi:

```
java.lang.ArrayIndexOutOfBoundsException: -1
      at Staku.pop(Staku.java:10)
      at Staku_ESTest.test3(Staku_ESTest.java:51)
```

Ekziston edhe një problem tjetër tek klasa Stack. Po sikur metoda `push(int)` të thirret 4 herë? Atëherë vlera e variablit madhësia do të ishte 4. Nëse pas 4 thirrjeve të `push(int)`, thirret `pop()`, do të tejkalonim kufijtë e matricës `st[]`, pasi do të tentonim të nxirrnim elementin tek pozicioni me indeks 4. Një rast tesimi për të vënë re këtë gabim do të ishte:

```
public void test3()  throws Throwable  {
      Staku staku0 = new Staku();
      staku0.push(2967);
      staku0.push(2967);
      staku0.push(0);
      staku0.push(2967);
      staku0.pop();
}
```

Ekzekutojmë rastin test3(). Ky rast dështon dhe afishohet mesazhi:

```
java.lang.ArrayIndexOutOfBoundsException: 4
      at Staku.pop(Staku.java:11)
      at Staku_ESTest.test4(Staku_ESTest.java:60)
```

Nëse katër testet (1 i gjeneruar automatikisht + 3 të shkruara për të kapur gabimet) e dhëna më sipër do të ishin gjeneruar automatikisht, atëherë do të kishim patur tre dështime dhe do të ishin identifikuar gabimet që ekzistojnë në klasën Staku; panvarësisht se mbulimi i degëve do të mbetej 100%. Klasa e korrigjuar do të ishte si më poshtë (me ngjyrë të verdhë jepen shprehjet ku janë bërë korrigjimet).

```
public class Staku {
            private int madhesia = 0;
            private int st [] = new int [4];
            void push (int x){
                  if (madhesia < st.length)
                        st[madhesia++] = x;
            }
      int pop (){
            if (madhesia > 0)
            return st[--madhesia];
            else
                  return -1;
      }
}
```

Duke përdorur si shembull një klasë të thjeshtë si klasa Staku, provuam konkretisht se kriteri i mbulimit të degëve, edhe kur plotësohet 100%, nuk siguron kapjen e disa lloje gabimesh. Ne dimë se mbulimi i degëve nuk është kriteri më i fortë, por është kriteri më i përdorur sot për vlerësimin e rasteve të gjeneruara gjatë kërkimit automatik. Në këtë tezë doktorature ne do të përpiqemi ti japim përgjigje pyetjes së mëposhtme:

Ka mundësi që funksionit të vlerësimit, krahas kriterit të mbulimit të degëve ti shtojmë edhe ndonjë kriter tjetër (pa shtuar shumë kompleksitetin) që të përftojmë teste më cilësore për programet ne Java?

Analizë e testeve të klasës Staku:

Nëse i kthehemi sërisht shembullit të mësipërm dhe studiojmë rastin e testimit `test1()`, vëmë re se përjashtimi i gjeneruar nga thirrja e metodës `push(int)`, vjen si pasojë e faktit se atributi `madhësia`, para thirrjes së kësaj metode, është -1 (është bërë -1, pas thirrjes së metodës `pop()`). Metoda `push(int)`, është thirrur edhe në rastin e testimit `test0()`, që u gjenerua automatikisht dhe shërben për të përmbushur kriterin e mbulimit të degëvë. Tek ky rast metoda `push(int)` thirret pesë herë dhe nuk gjeneron asnjë përjashtim. Atributi `madhësia`, këto pesë herë është përkatësisht 0, 1, 2, 3, 4. Pra, kur **e njëjta** metodë u thirr mbi gjendje të ndryshme të objektit, rezultati ishte i ndryshëm. Në këto dy teste rezultatin e ndryshon gjendja e objektit në momentin kur u thirr metoda `push(int)` dhe jo parametri i kësaj metode.

Marrim në studim tani rastin e testimit `test2()`. Vëmë re se përjashtimi në këtë rast gjenerohet nga thirrja e metodës `pop()` për herë të dytë. Arsyeja është se atributi `madhësia` para thirrjes së dytë të metodës është negativ, ndërsa para thirrjes së parë është 0. Në rastin e testimit `test0()`, që u gjenerua automatikisht dhe shërben për të përmbushur kriterin e mbulimit të degëvë, metoda `pop()`, nuk gjeneron përjashtim, pasi po thirret nga objekti me gjendje të atributit `madhësia` = 2. Mund të themi sërisht se kur e njëjta metodë u thirr mbi gjendje të ndryshme të objektit, rezultati ishte i ndryshëm.

Si pasojë e faktit që një metodë në Java, mund të ketë sjelljë të ndryshme kur thirret mbi gjendje të ndryshme të objektit thirrës, atëherë do të ishte me interes që rastet e testimit të gjeneruara të përfshinin thirrje të metodave mbi gjëndje të ndryshme. Prandaj është e nevojshme që gjatë kërkimit të evolojë edhe gjendja e objektit të klasës nën testim, në mënyrë që të dalin në pah sjellje të reja të objektit (idealisht të shfaqen gjithë sjelljet e objektit) dhe kështu të rritet efektiviteti i detektimit të gabimeve nga këto teste. Ideja është të shpërblehen ato raste testimi që i vendosin objektet në gjendje "interesante", d.m.th. në gjendje të reja dhe që shfaqin interes për testimin strukturor. Për të arritur këtë, natyrisht që duhet patur një model i sjelljes së objektit. Ky model jep lidhjen midis metodave dhe gjendjeve, d.m.th. tregon për secilën metodë, se si ndikon në gjendjen e objektit dhe gjithashtu tregon si duhet të jetë gjendja e objektit, që një metodë e caktuar të mund të thirret.

Testimi i bazuar në gjendje është propozuar që në vitet 70 nga Chow [84]. Kritere mbulimi që marrin parasysh edhe gjendjen e sistemit, janë propozuar nga autorë të ndryshëm [85][86][87][88].

3.7 Përftimi i Modelit të Sjelljes së Objekteve

Edhe pse modelet e sjelljes së objekteve konsiderohen si të rëndësishme, ato zakonisht janë jo të plota. Nga studimi i literaturës, arritëm në përfundimin se punimi më i përshtatshëm në këtë drejtim, duke marrë në konsideratë kontekstin ku duam të përdorim modelin e sjelljes së objekteve, është punimi i propozuar nga Dallmeier at al. i emërtuar ADABU [89].

ADABU, monitoron ekzekutimin e testeve në një sistem për të nxjerre modelet e sjelljeve të objekteve. Qëllimi është të nxirret një model i sjelljeve normale të sistemit. Si formalizim për këtë model, ADABU përdor makinat me gjendje të fundme, në të cilat nyjet paraqesin gjendjet "me kuptim" të objekteve, ndërsa harqet tregojnë tranzicionet midis gjendjeve. Gjendjet e objekteve përftohen duke thirrur metodat që kthejnë vlerat e atributeve (si fillim bëhet një analizë për të ndarë metodat në dy grupe: **observuese** dhe **ndryshuese**), para dhe pas thirrjes së metodave që ndryshojnë gjendjen. Në këtë mënyrë do të merret një informacion i plotë mbi gjithë tranzicionet e gjendjeve të objektit për secilën gjendje fillestare të mundshme. Për shembull në figurën 3.11, paraqitet modeli i sjelljes për klasën `Vector` [89]. Klasa përbëhet nga dy gjendje: *bosh* dhe *jo bosh*. Këto dy gjendje merren nga metoda observuese `isEmpty`. Harqet tregojnë efektet e thirrjes së metodave dhe lokalizojnë gjendjet e reja që objekti mund të marrë. Duhet theksuar se modeli nuk është një specifikim i detajuar i sistemit, megjithatë zhvilluesit e sistemit mund ta analizojnë këtë model për të kuptuar sjelljen e përgjithshme të klasës dhe të lokalizojnë gabime (p.sh. prezenca e një harku me etiketën e metodës add, që drejtohet nga gjendja empty do të ishte padyshim rrjedhojë e një gabimi). Përdorimi i këtij modeli konkretisht në këtë punim do të shpjegohet në kapitullin 4.

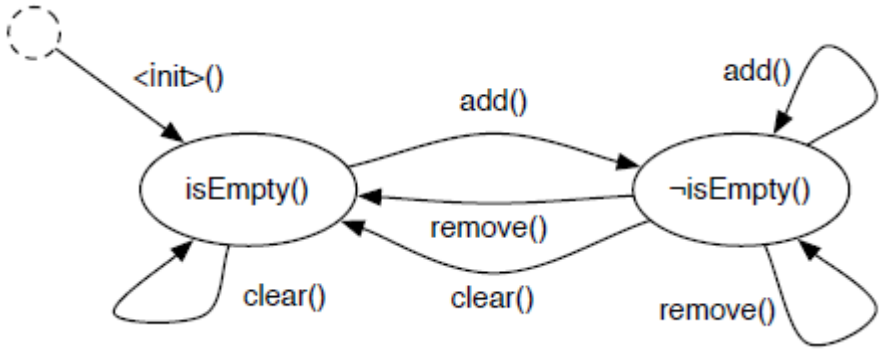

Figura 3.11: Modeli i sjelljes së objekteve i gjeneruar me ADABU për klasën `Vector` [89]

KAPITULLI 4
Implementimi

Në kapitujt e mëparshëm u përshkrua se si algoritmat meta-heuristikë (dhe veçanërisht algoritmat gjenetikë), mund të aplikohen për zgjidhjen e problemit të gjenerimit automatik të rasteve të testimit, për testimin strukturor të klasave në Java. Në këto kapituj u paraqit një pamje të përgjithshme e fushës së testimit që bazohet në kërkim, e cila vazhdon të ketë shumë vëmendje nga komuniteti akademik. Në këtë kapitull do të prezantohet një funksion i ri vlerësimi për algoritmat gjenetikë që përdoren për testimin në nivel njësie të programeve në Java (ky funksion mund të shërbejë edhe për gjuhë të tjera të orientuara nga objekti). Për implementimin e këtij funksioni do të përdoret mjeti eToc[1], i cili do të modifikohet për t'ju përshtatur këtij funksioni të ri vlerësimi.

Kontributet kryesore të këtij kapitulli janë:

1. Përcaktimi i një funksioni të ri vlerësimi që merr në konsideratë jo vetëm mbulimin e dëgëve por edhe gjendjet e arritura nga instanca e klasës nën testim.
2. Përshkrimi dhe implementimi i gjithë ndryshimeve që sjell përdorimi i këtij funksioni vlerësimi të propozuar.

Pjesa e mbetur e kapitullit është e organizuar si në vijim: Çështja 4.1 paraqet faktorët që do të konsiderohen gjatë vlerësimit dhe formulën matematike të funksionit të propozuar të vlerësimit. Çështja 4.2 shpjegon cilat do të jenë qëllimet e kërkimit nëse do të përdoret funksioni i prpozuar. Çështjet 4.3, 4.4 dhe 4.5 japin në mënyrë të detajuar implementimin e gjithë komponentëve për përdorimin e funksionit të ri të vlerësimit duke theksuar ndryshimet që duhet të kryhen mbi këto komponente (krahasuar me implementimin origjinal të mjetit eToc).

4.1 Funksioni i propozuar i vlerësimit

Siç u shpjegua edhe në paragrafet e mësipërme, funksioni i vlerësimit shërben për të drejtuar kërkimin. Qëllimi është që kërkimi të drejtohet drejt individëve më të mirë, prandaj akoma sot, panvarësisht shumë studimeve të kryera, përcaktimi i funksionit të vlerësimit është ende një fushë e hapur kërkimi. Sa më mirë të vlerësohen individët, aq më shumë mundësi ka të merret një output cilësor. Funksioni i propozuar në këtë punim merr në konsideratë edhe gjendjet e arritura nga objekti i klasës që është nën testim. Ky funksion do të paraqitet si një ekuacion matematik me varësi nga:

- Niveli i afërsisë
- Distanca e degëve
- Gjendjet e reja të arritura

Çështjet më poshtë shpjegojnë se si do të ndikojë secili prej faktorëve tek funksioni i vlerësimit.

4.1.1 Niveli i afërsisë

Për secilin qëllim, niveli i afërsisë do të llogaritet si largësia në nyje midis qëllimit dhe nyjes më afër këtij qëllimi që është arritur gjatë ekzekutimit [47]. Për secilin ekzekutim, me anë të intrumentimit që do të jetë kryer në fazën e parë (do të shpjegohet në çështjen pasardhëse), do të jenë ruajtur në një listë të gjithë nyjet ku ka kaluar ekzekutimi. Gjithashtu do të jenë ruajtur edhe varësitë për të gjithë qëllimet që duhen mbuluar. Atëherë duke bërë një krahasim të path-it të përftuar nga ekzekutimi dhe të path-it të varësive për një qëllim të caktuar, do të përftohet niveli i afërsisë si diferenca midis numrit të nyjeve tek path-i i varësive dhe numrit të nyjeve të përbashkëta të path-it të varësive dhe path-it të nyjeve të mbuluara gjatë ekzekutimit.

Niveli i afërsisë do të paraqitet me një numër të plotë dhe do të shpërblejë ato individë që shkojnë më afër qëllimit që duhet mbuluar. Niveli i afërsisë është faktori më i përdorur për ndërtimin e funksionit të vlerësimit për kriteret strukturore. Pasha e këtij

faktori është e madhe pasi nga rastet që kanë nivel afërsie më të vogël ka më shumë mundësi që pas rikombinimit ose mutacionit të përftohen individë që e mbulojnë qëllimin, ose që kanë nivel afërsie edhe më të vogël se gjenerata paraardhëse. Megjithatë, ky faktor nuk i jep shumë drejtim kërkimit pasi vërtetë tregon afërsinë e ekzekutimit me qëllimin për t'u mbuluar, por nuk jep informacion se sa larg ishte ekzekutimi nga zgjedhja e degës së kundërt tek nyja ku ekzekutimi nuk ndoqi path-in e kërkuar për arritjen e qëllimit (nyja kritike). Pra, duke përdorur vetëm nivelin e afërsisë, nuk kemi informacion se si duhet të jetë rasti i testimit që të mbulohet kjo nyje. Në këtë situatë, hapësira e kërkimit mund të ketë "plateaux". Nëse predikata e nyjes ku ndodhi devijimi nga path-i i kërkuar është e thjeshtë, atëherë ka shumë mundësi që mbulimi i dy degëve të kësaj nyjeje të ndodhë rastësisht. Mirëpo, nëse predikata është komplekse një kërkim i tillë i padrejtuar, ka shumë gjasa të dështojë në mbulimin e nyjes dhe për rrejdhojë edhe në mbulimin e qëllimit. Për të ilustruar hapësirën e vlërësimit për funksione të ndryshme objektivë, marrim në konsideratë metodën që jepet në figurën 4.1.

```
1 void shembull_vleresimi (int i, int j){
2      if (i >= 10 && i <= 20){
3             if (j >= 0 && j <= 10){
4                  //qëllimi për t'u arritur
5      }
6 }
```

Figura 4.1: Shembull për krahasimin e funksioneve të ndryshëm të vlerësimit

Qëllimi që kërkohet të mbulohet, tek path-i i tij i varësive ka dy nyje (tek rreshti 2 dhe tek rreshti 3). Nëse do të përdoret vetëm niveli i afërsisë, nuk do të kemi informacion se si mund të ndryshohet rrjedha e ekzekutimit tek këto nyje kontrolli, por vetëm informacion mbi numrin e nyjeve që ekzdekutimi është larg qëllimt. Hapësira e vlerësimit për qëllimin e kësaj metode jepet në figurën 4.2.

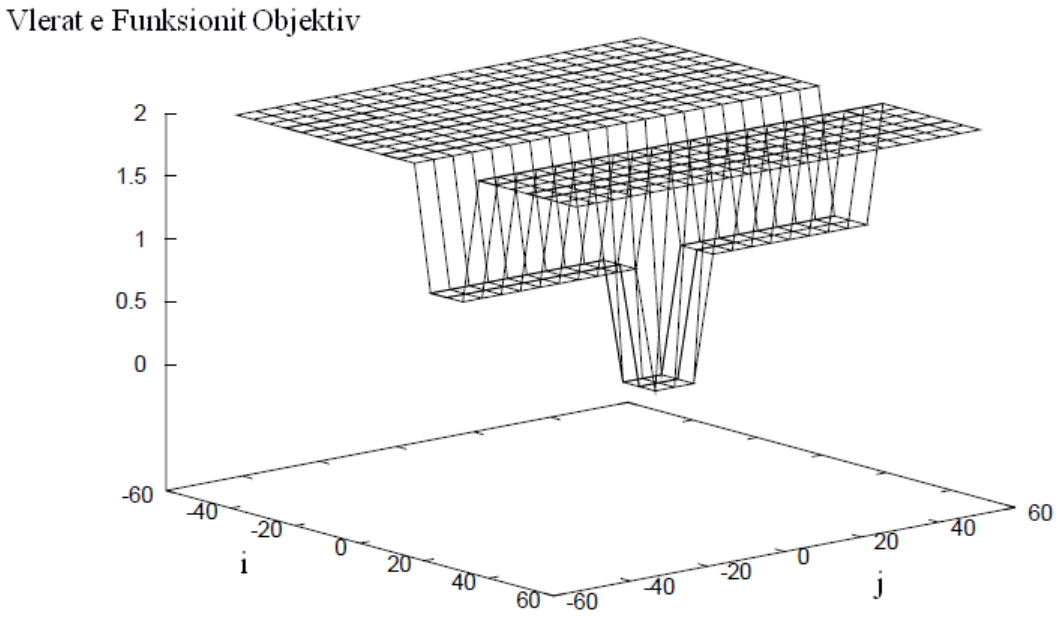

Figura 4.2: Hapësira e funksionit objektiv që mat vetëm nivelin e afërsisë, për shembullin e figurës 4.1 [46]

Duket qartë nga grafiku se hapësira ka "plateaux" (pjesët e rrafshta në hapësirën e vlerave të funksionit). Për individët që nuk përmbushin një ose më shumë nga predikatat e degëve, nuk jepet asnjë drejtim në mënyrë që të zbritet më poshtë tek hapësira e zgjidhjeve që janë më afër ekzekutimit të qëllimit; për rrjedhojë lëvizja do të mbetet e rastit. Për këtë arsye, tek funksioni i vlerësimit, do të shtojmë edhe faktorin tjetër që përdoret për kriteret strukturore që është distanca e degëve.

4.1.2 Distanca e Degëve

Distanca e degëve (DD) paraqet një vlerë që tregon sa larg është një predikatë që të vlerësohet si e vërtetë. Nëse shtohet ky faktor tek funksioni i vlerësimit, atëherë hapësira e kërkimit do të jetë më e lëmuar se në rastin kur përdoret vetëm niveli i afërsisë. Distanca e degëve përdoret për të zgjidhur detyrimet e predikatave që janë në path-in e kontrollit të varësive të një qëllimi. Distanca e degëve, tek funksioni i vlerësimit, llogaritet duke përdorur kushtin e shprehjes së kontrollit ku rrjedha e ekzekutimit devijoi nga path-i i kërkuar për një qëllim që duhet të mbulohet. Në paragrafin 2.5.2 u paraqit si do të llogaritet distanca e degëve për predikata më operatorë të ndryshëm, bazuar tek

Tracey et al. [45]. Për ndërtimin e funksionit të vlerësimit niveli i afërsisë dhe distanca e degëve do të kombinohen në mënyrë lineare. Megjithatë, niveli i afërsisë është më i rëndësishëm dhe për rrjedhojë distanca e degëve duhet të normalizohet tek ky funksion. Kjo distancë do të normalizohet në një vlerë midis 0.0 dhe 1.0, ku 0.0 tregon "true", ose që dega e kërkuar është arritur. Vlerat afër 1-it tregojnë se kushti është larg përmbushjes. Vlerat e ndërmjetme duhet të jenë të tilla që në mënyrë të lehtë të drejtojnë kërkimin drejt përmbushjes së kushtit. Formula që përdoret zakonisht për normalizimin e vlerës së distancës është:

$$DD(normalizuar) = 1 - 1.001^{-DD}$$

ku DD, është distanca para normalizimit.

Megjithatë, Arcuri [90] ka vënë në dukje disa të meta në këtë formulë, që ndikojnë në rezultatin e përftuar nga kërkimi. Ai propozon një funksion tjetër normalizimi që është dhe më i lehtë për t'u llogaritur. Ky funksion është:

$$DD(normalizuar) = \frac{DD}{DD + \beta}$$

ku β është një konstante > 0 dhe DD është distanca para normalizimit. Në këtë punim, do të përdoret funksioni i propozuar nga Arcuri dhe β do të merret me vlerë 1.

Nëse kemi p.sh. nyjen `if (x < 2)`, atëherë nëse x ka vlerën 1, distanca për degën true është 0.0, ndërsa distanca për degën false është DD(x >= 2) = 2 – x + K = 2 – 1 + K = 1 + 1 = 2, d.m.th. DD(normalizuar) = 2/3 = 0.66.

Marrim në konsideratë sërisht metodën `shembull_vleresimi` që jepet në figurën 4.1. Nëse, si funksion objektiv, do të përdoret një kombinim i nivelit të afërsisë dhe distancës së degëve, hapësira e vlerësimit do të jetë si në figurën 4.3.

Siç vëmë re nga grafiku, hapësira e funksionit të vlerësimit ka një formë të ngjashme me rastin e figurës 4.2. Megjithatë informacioni shtesë që siguron distanca e degëve, parandalon formimin e "plateaux" tek secili prej niveleve të afërsisë. Kalimi nga një nivel në një tjetër më poshtë, bëhet në mënyrë të pandërprerë.

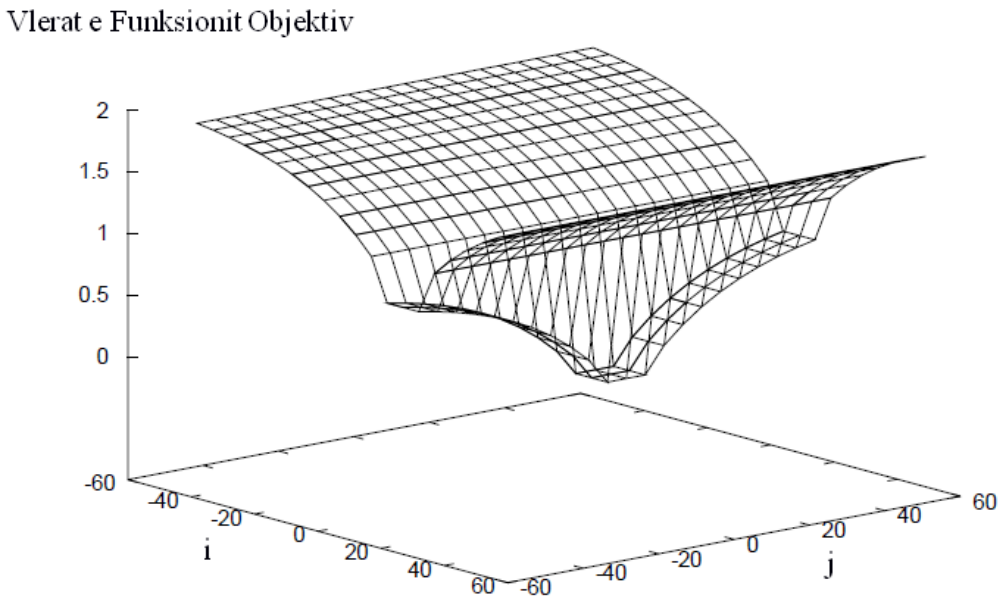

Figura 4.3: Hapësira e funksionit objektiv që mat nivelin e afërsisë dhe distancën e degëve, për shembullin e figurës 4.1 [46]

Problemi kryesor që ka llogaritja e distancës së degëve është në rastin kur kodi përmban *flamuj boolean.* Për shembull, marrim në konsideratë fragmentin e kodit:

```
boolean f = x > 0;
..
if (f) {..}
```

Në këtë rast, nëse duam të gjejmë distancën për degën true dhe `f` ka qenë false, atëherë nuk kemi asnjë informacion për distancën e degës true pasi ajo varet nga kushti `x > 0` dhe ky kusht ka humbur. Hapësira e vlerësimit do të ishte si në figurën 4.4 (c). Në rastet e flamujve booleanë distanca do të jetë gjithmonë 1 ose 0 dhe faktori distancë e degëve nuk do të ketë asnjë ndikim në rezultatin e vlerësimit.

Për të zgjidhur këtë problem mund të bëhen transformime në kod [91], por kjo çështje është jashtë qëllimit të këtij punimi.

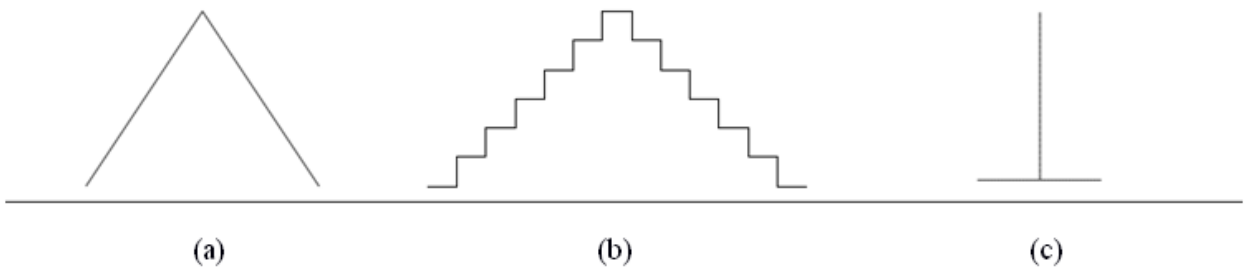

Figura 4.4: Pamja e vlerësimit në tre raste: (a) Pamja më e mirë me drejtim të kudondodhur drejt optimumit global. (b) Pamje e pranumeshme me drejtim në disa raste, drejt optimumit global. (c) Pamja më e keqe me "plateau dual" dhe pa drejtim drejt optimumit global [67]

4.1.3 Gjendjet e reja të arritura

Meqë kërkojmë që gjendja e objektit të evolojë gjatë kërkimit për të nxjerrë në pah gjithë tiparet e një klase, atëherë është e nevojshme që tek funksioni i vlerësimt, të shfaqet edhe faktori që mat gjendjet e reja të arritura, pasi në këtë mënyrë kërkimi do të fokusohet më shumë drejt rasteve të testimit që çojnë në gjendje të reja të objektit të klasës nën testim.

Me termin *gjendje* këtu i referohemi *bashkësisë së vlerave të gjithë atributeve të klasës në një çast të caktuar ekzekutimi + metodën që u thirr mbi këtë gjendje*. Gjatë testimit, paraqet interes gjendja e objektit para thirrjes së një metode. Gjithashtu, po e njëjta gjendje para thirrjes së një metode tjetër, do të paraqeste sërisht interes, prandaj në këtë punim do t'i referohemi si një gjendje e re.

Marrim në konsideratë klasën Staku (Çështja 3.6). Për këtë klasë gjendjet:

- `atributi` **`madhësia = 0`** `dhe atributi` **`st = !null,`** `para thirrjes së metodës` **`push`**
- `atributi` **`madhësia = 0`** `dhe atributi` **`st = !null,`** `para thirrjes së metodës` **`pop`**

do të cilësohen si gjendje të ndryshme dhe të dy rastet e testimit që i arrijnë këto dy gjendje do të ishin me interes në kontekstin e testimit.

Për rrjedhojë, numri total i gjendjeve do të llogaritet si prodhimi i gjithë kombinimeve të mundshme të vlerave të atributeve (pas abstraksionit që do të shpjegohet në çështjen 4.2.1.2) me numrin e metodave publike të klasës në testim.

Është e qartë që nuk mund të kërkohet që një rast testimi të arrijë gjithë gjendjet e mundshme. Gjtihashtu, duke patur parasysh që ka mjaft raste ku një ose disa gjendje janë të pamundura të arrihen, atëherë qëllimi i kërkimit nuk do të jetë arritja e gjithë gjendjeve të mundshme nga objekti i klasës nën testim.

Funksioni objektiv duhet të "shpërblejë" testet që kanë vendosur objektin në një ose më shumë gjendje të reja, megjithatë pasha e këtij faktori nuk eshtë e njëjtë me nivelin e afërsisë dhe për rrjedhojë duhet të kryhet një normalizim. Faktori i gjendjeve të reja tek funksioni i vlërësimit (GJRV) do të paraqitet me formulën:

$$GJRV = \frac{nr\_gjendjesh\_total - nr\_gjendjesh\_reja}{nr\_gjendjesh\_total}$$

Meqë gjatë kërkimit, synojmë minimizimin e funksionit të vlerësimit, atëherë sa më i madh numri i gjendjeve të reja, aq më e vogël duhet të jetë vlera e GJRV.

Si përfundim, po japim formulën e plotë të funksionit të vlerësimit të propozuar në këtë punim. Kjo formulë siç u shpjegua në paragrafet e mësipërme, merr në konsideratë nivelin e afërsisë, distancën e degëve dhe gjendjet e reja të arritura nga objekti i klasës nën testim.

$$Vlerësimi = niveli\_afërsisë + \frac{DD}{DD+1} + \frac{nr\_gjendjesh\_total - nr\_gjendjesh\_reja}{nr\_gjendjesh\_total}$$

4.2 Qëllimet e Kërkimit

Pas përcaktimit të funksionit të vlerësimit duhet t'i përgjigjemi pyetjes:

Me përdorimin e funksionit të vlerësimit të propozuar në çështjen 4.1, është e nevojshme të vendoset si kusht gjatë kërkimit që duhet të arrihen të gjitha gjendjet që

përftohen nga kombinimi i vlerave abstrakte të të gjithë atributeve + metodave të klasës që është nën testim?

Në këtë punim, edhe pse për vlerësimin e rasteve të gjeneruara gjatë kërkimit do të meren në konsideratë edhe gjendjet e arritura, sërisht qëllimet e kërkimit do të mbeten mbulimet e degëve të klasës nën testim. Shtimi i qëllimeve të tjera, si p.sh. mbulimi i gjithë gjendjeve të klasës, do të sillte një rritje në kohën e kërkimit dhe në madhësinë e set-it të rasteve të gjeneruara. Megjithatë, nëse do të ishin vetëm këto arsye, atëherë duhej të bëhej një vlerësim i avantazheve të përftuara kundrejt disavantazheve, në mënyrë që të mund të jepnim një përgjigje të saktë mbi efektin e shtimit të qëllimeve shtesë të mbulimit të gjithë gjendjeve. Por, problemi është se mbulimi i gjithë gjendjeve është praktikisht i pamundur në shumicën e rasteve.

Dihet se ekziston mundësia që një atribut të mos mund të marrë vlera që bëjnë pjesë në të gjitha grupet që përftohen nga abstraksioni (çështja 4.2.1.2). Kjo vjen, jo si pasojë e faktit se kërkimi që kryhet gjatë ekzekutimit të algoritmit gjenetik nuk arrin në vlera të caktuara, por si pasojë e mënyrës se si mund të jetë implementuar një klasë e caktuar që nuk lejon arritjen e këtyre gjendjeve. Për shembull, tek klasa Staku (çështja 3.6), atributi madhësia mund ti marrë të tre vlerat abstrakte (<0, =0, >0), pasi edhe pse ky atribut është deklaruar me akses `private`, mënyra se si janë implementuar metodat push dhe pop bën që ky atribut të marrë këto tre vlera. Nga ana tjetër, ne dimë që nëse kjo klasë do të ishte implementuar saktë, atëherë atributi madhësia nuk do mund të merrte asnjëherë vlerën abstrakte <0, pasi sido të thirreshin metodat push dhe pop (me çfarëdo argumentash), kjo vlerë nuk do arrihej. Prandaj, vendosja e një kushti që gjatë kërkimit duhet të arrihen të gjitha gjendjet që përftohen nga kombinimi i vlerave abstrakte të të gjithë atributeve të klasës që është nën testim, do të ishte jo i saktë dhe jo efektiv. Nëse kërkimi do të kërkojë të përmbushë një qëllim të pa mundur (d.m.th. nuk ka rast testimi që ta përmbushë qëllimin), atëherë do të harxhoheshin burimet e limtuara (koha). Prandaj si qëllime mbulimi do te mbeten vetëm degët e klasës nën testim.

4.3 Implementimi i funksionit të ri të vlerësimit

Siç u përmend edhe në paragrafin e mëparshëm për implementimin e funksionit të ri të vlerësimit do të përdoret mjeti eToc[1]. U zgjodh mjeti eToc, pasi ky mjet ka shërbyer si bazë edhe për ndërtimin e mjeteve të tjera. Gjithashtu eToc është i përshtatshëm për qëllimin e studimit tonë pasi kryhen testim automatik bazuar në kërkim duke përdorur algoritmat gjenetikë. Gjithashtu ky mjet ka të implementuar kriterin e mbulimit të degëve, gjë që paraqet interes për punimin tonë, pasi mbi këtë kriter do të implementohet funksioni i propozuar i vlerësimit që do të bazohet në gjendjet e reja të arritura.

Në vijim do të paraqiten ndryshimet e bëra në mjetin eToc dhe detaje të implementimit të këtyre ndryshimeve. Në figurën 4.5 jepet një pamje në nivel të lartë e organizimit të këtij mjeti.

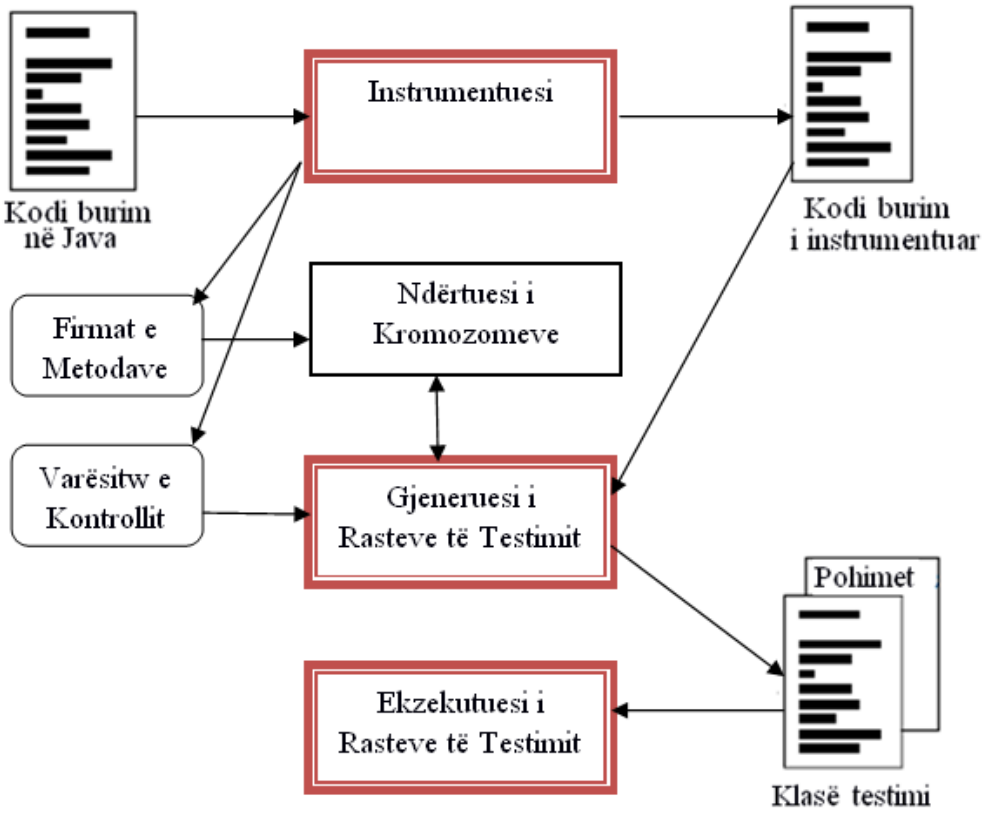

Figura 4.5: Arkitektura në nivel të lartë e mjetit eToc[1]. Me ngjyrë të kuqe tregohen modulet që do të ndryshohen në versionin e propozuar

Modulet që janë theksuar me ngjyrë të kuqe në figurën 4.5, është e nevojshme të ndryshohen për implementimin e funksionit të ri të vlerësimit. Çështjet më poshtë do të trajtojnë implementimin e secilit prej moduleve në versionin e ri të propozuar në këtë tezë.

4.3.1 Instrumentuesi

Moduli i instrumentimit shërben për të gjeneruar një version të instrumentuar të klasës origjinale që do të testohet. Ky modul merr si input skedarin ku ndodhet kodi burim dhe gjeneron një skedar të ri që përfshin krahas kodit burim, i cili mbetet i pandryshuar edhe pjesë të tjera kodi. Instrumentimi ndodh plotësisht në mënyrë automatike. Ky kod i ri i gjeneruar do të përdoret vetëm për qëllime testimi dhe testuesit nuk do të kenë nevojë të veprojnë mbi këtë kod të transformuar. Shprehjet e shtuara gjatë instrumentimit **nuk** duhet të ndryshojnë sjelljen e klasës që është nën testim, pra nuk duhet të kenë asnjë efekt anësor tek programi origjinal. Krahas instrumentimit të kodit, ky modul duhet të japë edhe informacion mbi metodat (gjenerohet nje skedar që mban firmat e të gjithë metodave të klasës) dhe mbi varësitë e kontrollit (gjenerohet një skedar që mban informacion mbi varësinë e secilës dëgë nga nyjet përkatëse). Këto të dhëna do të përdoren nga modulet e tjera. Instrumentuesi ka një detyrë mjaft të rëndësishme, pasi jo vetëm gjeneron informacion mbi klasën, por edhe e pajis klasën me instrumenta që do të përdoren më pas për të bërë vlerësimin e individëve të gjeneruar. Meqë qëllimi i këtij punimi është implementimi i një funksioni të ri vlerësimi, moduli i instrumentimit duhet të implementohet në mënyrë të ndryshme nga moduli që shërbente vetëm për instrumentimin e degëve, pasi duhet të kryhet edhe instrumentimi për të përftuar informacion mbi gjendjet e reja të arritura.

Në rastin e Java-s, mund të instrumentohen edhe kodi burim edhe byte kodi. Të dy rastet kanë avantazhet dhe disavantazhet e tyre. Në këtë punim është zgjedhur të instrumentohet kodi burim, pasi në këtë rast, kodi i instrumentuar është shumë më i thjeshtë për t'u kuptuar gjatë inspektimit dhe në këtë mënyrë zhvillimi dhe debugging bëhet më i thjeshtë [92].

Për të kryer instrumentimin është përdorur OpenJava (OJ) [93]. OpenJava është një sistem makro i avancuar për Java. Në ndryshim nga shumë makro të tjera OJ nuk përdor

Pemët Abstrakte të Sintaksës (PAS), pasi PAS janë një paraqitje plotësisht tekstuale e programit dhe janë të pavaura nga informacioni logjik ose kontekstual. OJ prodhon një objekt që përfaqëson strukturën logjike të përcaktimit të klasës, për çdo klasë që bën pjesë tek kodi burim. Ky objekt quhet metaobjekt. Programuesi ka mundësi që të përshtasë sipas nevojës përcaktimin e klasës së metaobjekteve për të bërë zgjerimin e makros (macro expansion). Kjo klasë quhet metaklasë. Për të përdorur OJ ndërtuam metaklasën që do të kryente zgjerimin e macros (ndryshimet tekstuale që do i bëhen klasës nën testim).

Cilat janë transformimiet tekstuale që duam ti bëjmë klasave që janë nën tetsim?

Meqë funksioni jonë i vlerësimit do të varet nga degët e arritura dhe nga gjendja e përftuar e objektit, atëherë transformimet duhet të jenë të tilla, që gjatë ekzekutimit të merret ky informacion për secilin rast testimi që përftohet gjatë kërkimit. Transformimet që do të kryhen do të ndahen në tre grupe:

1. Transformime për përftimin e informacionit mbi degët e ekzekutuara
2. Transformime për përftimin e informacionit mbi gjendjen e arritur
3. Transformime për përftimin e informacionit mbi distancën e degëve

4.3.2 Transformime për përftimin e informacionit mbi degët e ekzekutuara

Për të ndërtuar atë pjesë të metaklasës që bën ndryshimet për përftimin e informacionit mbi dëgët e arritura jemi referuar tek [1], pasi mjeti eToc, e ka funksionin e vlerësimit që bazohet tek degët e ekzekutuara.

Krijohet një objekt i tipit `java.util.List`.

```
static java.util.List trace = new java.util.LinkedList();
```

Këtij objekti, sa herë arrin ekzekutimi para tek një dege, i shtohet nje objekt i tipit `java.lang.Integer`. Që të mund të realizohet kjo, shtohet para shprehjes së parë të çdo metode/dege shprehja e mëopshtme:

```
trace.add(new java.lang.Integer(nrDege));
```

Shprehjet që kosiderohen nyje janë: *Grupet e Case tek Switch, blloqet catch të përjashtimeve, shprehja DoWhile, shprehja For, shprehja If, shprehja Try, shprehja While.*

Në fund të ekzekutimit, në listë është identifikuesi i çdo dege që ka përshkruar ekzekutimi.

Për ilustrim po marrim sërisht klasën Staku (çështja 3.6). Pas instrumentimit klasa Stack do të bëhej:

```
public class Staku {
	private int madhesia = 0;
	private int st [] = new int [4];
	void push (int x){
		trace.add(new java.lang.Integer(1));
		if (madhesia < st.length){
			trace.add(new java.lang.Integer(2));
			st[madhesia++] = x;
		}
		else {
			trace.add(new java.lang.Integer(3));
		}
	}
	int pop (){
		trace.add(new java.lang.Integer(4));
		return st[madhesia--];
	}
	static java.util.List trace;
	public static void newTrace()
	{
		trace = new java.util.LinkedList();
	}
	public static java.util.List getTrace()
	{
		return trace;
	}
    }
```

Figura 4.6: Kodi burim i klasës Staku pas instrumentimit të kryer për vlerësimin e bazuar tek mbulimi i degëve

4.3.3 Transformime për përftimin e informacionit mbi gjendjen e arritur

Këto transformime synojnë shtimin e instrumentave për përftimin e gjendjes së një objekti (bashkësia e vlerave të fushave të klasës, instancë e të cilës është objekti) në një çast të caktuar. Zakonisht proçesi për marrjen e modeleve të gjendjes konsiston në dy faza [89]. Si fillim, nëpërmjet një analize statike identifikohen metodat që nuk kanë "efekte anësore" në program. Këto metoda quhen edhe inspektues të programit. Inspektuesit kthejnë informacion mbi gjendjen e objektit. Gjatë fazës së dytë, ekzekutohet programi edhe thirren inspektuesit për të marrë informacion mbi gjendjen e objektit.

Një metodë që të mund të përdoret si inspektuese duhet të plotësojë kriteret e mëposhtme:

- **Nuk kthen void:** Meqë inspektuesit duhet ti kthejnë thirrësit informacion, atëherë është e qartë që tipi i tyre i kthimit nuk mund të jetë void.
- **Të mos ketë parametra:** Nëse një inspektues merr një argument *a*, atëherë ka shumë mundësi që vlera e kthyer nuk do të varet vetëm nga gjendja e vetë objektit, por edhe nga gjendja e *a*-së.
- **Të mos ketë efekte anësore:** Ekzekutimi i një inspektuesi nuk duhet të ketë efekte anësore tek gjendja e programit. Kjo siguron që thirrja e inspektuesit nuk ka asnjë impakt në ekzekutimin e programit dhe mund të thirret në çdo rast që lind nevoja për të.

Analiza statike për të përcaktuar metodat që plotësojnë këto tre kushte mund të kryhet në mënyra të ndryshme. Sipas [94], vërtetimi i kushtit të tretë është i vështirë edhe kërkon një analizë të gjithë programit. Ka edhe nje mënyrë më të thjeshtë [95], në të cilën, një metodë klasifikohet si pa efekte anësore, nëse ajo nuk modifikon objektet që ekzistonin para thirrjes së kësaj metode. Megjithatë, këto mënyra, edhe kur janë të thjeshtuara kërkojnë një analizë të programit (në këtë rast të klasës që është nën testim), gjë që kërkon kohë dhe shton kompleksitetin e mjetit për testimin automatik. Gjithashtu, qëllimi në kontekstin tonë, nuk është nxjerrja e mirfilltë e një modeli të sjelljes së

objekteve, pasi jo gjithë informacioni i përftuar do të ishte me interes. Qëllimi këtu është të kuptojmë pas ekzekutimit të një rasti testimi, se cilat ishin gjendjet në të cilat u vendos objekti gjatë ekzekutimit të këtij rasti. Prandaj, në këtë punim ne propozojmë një mënyrë të thjeshtë e cila është e përshtatshme për qëllimin ku do të përdoret. Në vend që të bëjmë një analizë paraprake të programit për të identifikuar inspektuesit, do të identifikohen nëpërmjet OpenJava gjithë fushat e klasës dhe do të shtohen metoda `get` për secilën fushë, përveç fushave të deklaruara `final`, pasi gjendja e tyre nuk ndryshon nga asnjë metodë dhe për rrjedhojë monitorimi i këtyre fushave nuk paraqet interes. Metodat get janë të përshtatshme dhe plotësojnë tre kushtet e sipërpërmendura pasi kthejnë vlerën e një fushe të klasës, nuk marrin parametra dhe nuk kanë efekte anësore tek klasa ku bëjnë pjesë. Këto metoda do të thirren të gjitha në rast se duam të marrim gjendjen e objektit. Kujtojmë se një rast testimi konsiston në ndërtime objektesh, thirrje metodash për vendosjen e objekteve të ndërtuara në një gjendje të caktuar dhe në fund thirrjen e metodës ku ndodhet qëllimi për t'u përmbushur. Për këtë arsye, thirrjet e gjithë metodave get të ndërtuara, do të vendosen si shprehje të para të çdo metode që ndodhet në klasën nën testim.

Instrumentimi në këtë rast do të konsistojë në shtimin e metodave get për secilën fushë. Përdorim OpenJava dhe tek metaklasa mbikalojmë (i bëjmë override) metodës `translateDefinition()`. Më poshtë jepet vetëm pjesa e kodit për këtë metodë që shton metodat get. Si fillim me anë të metodës `getDeclaredFields(Object)`, marrim të gjitha fushat dhe i vendosim në një vektor të tipit `OJField`. Më pas, me anë të një cikli `for`, kapim gjithë elementët e vektorit, të cilët përfaqësojnë gjithë fushat e klasës. Për secilën fushë kontrollojmë nëse ka modifikuesin `final`; nëse jo, atëherë ndërtojmë një metodë publike që kthen tipin e fushës dhe me emer *get+emrin e fushës+1*. Karakteri '1' shtohet në fund të emrit që të mos kemi konflikt emrash, pasi ka mundësi të ketë një tjetër metode ekzistuese get me emër të njëjtë në klasë (duke marrë parasysh se si emërtohen metodat get në Java), dhe do të kishim gabim në kompilim. Shprehja e vetme që përmbajnë metodat get të ndërtuara, është kthimi i vlerës së një fushe të klasës. Kodi i metaklasës për të kryer instrumentimin me qëllim ndërtimin dhe shtimin e metodave get jepet në figurën 4.7.

```
void translateDefinition() throws MOPException {
    OJField[] f = this.getDeclaredFields(this);
    for (int i = 0; i < f.length; ++i) {
            OJModifier modif = f[i].getModifiers();
            if (!modif.isFinal() && !f[i].getName().equals("trace")) {
                OJMethod n = new OJMethod(this,
                f[i].getModifiers(),f[i].getType(),"get"+f[i].getName()+"1",
                null, null, null, null);
                Statement s = makeStatement ("return" +f[i].getName() + ";");
                n.getBody().insertElementAt(s, 0);
                this.addMethod(n);
               }
    }
```

Figura 4.7: Kodi i metaklasës për të kryer instrumentimin me qëllim ndërtimin dhe shtimin e metodave get

Vlera e kthimit, për secilën metodë të ndërtuar, do të shtohet në një objekt të tipit `String` të quajtur `gjendjaPlote`. Në fund ky objekt do të ketë gjithë gjendjet e kthyera nga metodat get, të vendosura njëra pas tjetrës dhe të ndara me hapësirë.

Do të ndërtohet gjithashtu një fushë statike e tipit `LinkedList` (emri i saj do të vendoset `gjendjetArritura`), që do të mbajë gjithë gjendjet e arritura gjatë ekzekutimit. Objektet e kësaj liste do të jenë të tipit `String`.

Më pas do të krijohet një metodë (emri i saj do të vendoset `ktheGjendje()`), e cila thërret gjithë metodat get të shtuara dhe thirrja e kësaj metode do të bëhet si shprehje parë e çdo metode të klasës. Në momentin që kjo metodë thirret, qëllimi i saj është të ruajë gjendjen e objektit (që ruhet tek stringa gjendjaPlote) në listën gjendjetArritura. Ne jemi të interesuar ta thërresim metodën `ktheGjendje()` kryesisht tek metodat e klasës që nuk janë inspektuese por ndryshuese, pasi sjellja e këtyre metodave varet më shumë nga gjendja e objektit dhe do të paraqiste më shumë interes për testimin. Megjithatë edhe për sa i përket metodave inspektuese që mund të ketë klasa (përveç metodave get që shtuam ne) dhe për të cilat ne nuk kemi ndonjë informacion, pasi siç u shpjegua më sipër, nuk bëhet analiza e klasës për identifikimin e tyre, ekzistojnë raste ku sjellja e këtyre metodave varet nga gjendja e objektit para thirrjes. Për këtë arsye, metoda `ktheGjendje()` do të thirret për çdo metodë të klasës që ekzistonte para instrumentimit. Kjo metodë do të ndërtohet me modifikues aksesi `public` dhe me tip

kthimi `void`. Gjithashtu është e qartë se metodat get nuk duhet të llogariten kur të përcaktohen qëllimet për mbulimin e degëve, prandaj shtimi i identifikuesve të degëve duhet të bëhet para shtimit të metodave get. Kodi i metaklasës për të kryer instrumentimin me qëllim ndërtimin dhe shtimin e metodave get dhe metodës ktheGjendje dhe për thirrjen e metodës ktheGjendje nga çdo metodë e klasës jepet në figurën 4.8.

```
public void translateDefinition() throws MOPException {
            OJModifier mod = OJModifier.forModifier( OJModifier.STATIC );
            FieldDeclaration fd = new FieldDeclaration( new ModifierList(
                                  ModifierList.STATIC ), TypeName.forOJClass(
                                  OJClass.forName( "LinkedList" ) ) ,
                                  "gjendjetArritura", null);
            OJField fusha = new OJField( getEnvironment(), this, fd );
            addField( fusha );
            OJMethod gjendja = new OJMethod (this, null, null, null, null,
                               null, null, null);
            gjendja.setName("ktheGjendje");
            gjendja.setModifiers(OJModifier.PUBLIC);
            gjendja.setReturnType(OJPrimitive.VOID);
            String gjendjaPlote = "";
            String tipiFushave = "";

            OJField[] f = this.getDeclaredFields();
            for (int i = 0; i < f.length; ++i) {
                OJModifier modif = f[i].getModifiers();
                if (!modif.isFinal() && !f[i].getName().equals("trace")) {
                   OJMethod n = new OJMethod(this,
                   f[i].getModifiers(),f[i].getType(),"get"+f[i].getName()+"1
                   ", null, null, null, null);
                   Statement s = makeStatement("return"+ f[i].getName()+";");
                   n.getBody().insertElementAt(s, 0);
                   this.addMethod(n);
                   String shprehje = n.getName();
                   gjendjaPlote = gjendjaPlote+"
                                  String.valueOf("+shprehje+"())";
                   tipiFushave = tipiFushave + f[i].getType();
                   }
            }
```

```
                Statement ruajGjendje = makeStatement (fusha.getName() +
                ".add(gjendjaPlote)");
                gjendja.getBody().insertElementAt(ruajGjendje, 0);
                Statement printoGjendje = makeStatement ("ktheGjendje();");
                OJMethod[] methods = getDeclaredMethods();
                for (int i = 0; i < methods.length; i++) {
                   methods[i].getBody().insertElementAt(printoGjendje, 0 );
            }
      }
```

Figura 4.8: Kodi i metaklasës për të kryer instrumentimin me qëllim ndërtimin dhe shtimin e metodave get dhe metodës ktheGjendje dhe për thirrjen e metodës ktheGjendje nga çdo metodë e klasës.

Po ilustrojmë sërisht instrumentimin e kodit burim të klasës Staku. Në figurën 4.9 jepet klasa Staku pas instrumentimit ekzistues edhe instrumentimit të propozuar në këtë punim dhe të kryer nga kodi në figurën 4.8.

```
public class Staku {
      private int madhesia = 0;
      private int st [] = new int [4];
      void push (int x){
            ktheGjendje();
            trace.add(new java.lang.Integer(1));
            if (madhesia < st.length){
                  trace.add(new java.lang.Integer(2));
                  st[madhesia++] = x;
            }
            else {
                  trace.add(new java.lang.Integer(3));
            }
      }
      int pop (){
            ktheGjendje();
            trace.add(new java.lang.Integer(4));
            return st[madhesia--];
      }
      public int getmadhesia1(){
            return madhesia;
      }
```

```
	public Object getst1(){
		return st;
	}
	public void ktheGjendje (){
		gjendjetArritura.add(String.valueOf(getmadhesia1())+"
	"+getst1());
	}

	static java.util.List gjendjetArritura;
	 public static void newGjendjetArritura()
	{
		gjendjetArritura = new java.util.LinkedList();
	}
	public static java.util.List getGjendjetArritura()
	{
	return gjendjetArritura;
	}
	static java.util.List trace = new java.util.LinkedList();

	public static java.util.List getTrace()
	{
		return trace;
	}
}
```

Figura 4.9: Kodi burim i klasës Staku pas instrumentimit ekzistues edhe instrumentimit të propozuar për përftimin e gjendjeve të objektit gjatë ekzekutimit të rasteve të testimit

Siç u shpjegua në paragrafin 4.3, me termin gjendje do t'i referohemi jo vetëm bashkësisë së vlerave të fushave të klasës, por tek gjendja do të shtojmë edhe identifikuesin e metodës që u thirr me këtë gjendje. Prandaj tek String-a gjendjetArritura, krahas vlerave të fushave do të ruhet në fund edhe emri i metodës.

Sipas [93], overhead-i i instrumentimit është i pakonsiderueshëm, mesatarisht, më pak se 8 sekonda për klasë. Numri i shprehjeve të klasës nën testim, rritet në mënyrë të konsiderueshme (rritja varet nga numri i fushave jo-final, nga numri i metodave dhe nga numri i degëve), megjithatë kjo rritje nuk është problematike, pasi kodi i instrumentuar

shërben vetëm për funksionimin e algoritmit gjenetik dhe nuk përdoret në asnjë rast nga testuesi.

Qëllimi i instrumentimit të kryer është të mundësojë monitorimin e gjendjeve, në të cilat arrin objekti gjatë ekzekutimit të rasteve të testimit. Megjithatë, ka një problem, pasi numri i gjendjeve që arrin objekti mund të jetë i pafundëm dhe gjithashtu jo të gjitha gjendjet e arritura kanë të njëjtën rëndësi për sa i përket testimit. Për shembull tek klasa Staku, kemi një atribut me emër `madhësia` të deklaruar me tip `int`. Vlerat që mund të marrë ky atribut janë të pafundme (gjithë bashkësia e numrave `int`). Njëkohësisht, nëse studiojmë klasën Staku, vëmë re se vlerat 1 dhe 2 për atributin madhësia nga pikëpamja e testimit do të ishin të njëjta. Të dyja metodat e klasës, edhe metoda push edhe metoda pop sillen njëlloj për të dyja vlerat e atributit madhësia, pra nuk ka asnjë përfitim që të thirren këto dy metoda, një herë kur objekti është në gjendjen me atributin madhësia = 1 edhe një herë kur objekti është në gjendjen me atributin madhësia = 2. Nuk do të përfitonim asgjë në lidhje me mbulimin e degëve dhe po ashtu nuk do të mund të detektonim gabimet e përmendura në çështjen 3.6. Pra, mund të themi se vërtetë jemi në dy gjendje të ndryshme, por normalisht duhet të kemi një vlerësim identik (nuk duhet të kemi një rritje në fitness) nga pikëpamja e gjendjeve për dy raste testimi që do të vendosnin atributin madhësia në gjendjet 1 dhe 2. Kjo do të thotë se jo çdo gjendje e re duhet të shpërblehet me një vlerësim më të mirë. Arrijmë në përfundimin se gjendjet duhet të grupohen; gjendjet që ndodhen në grupe të ndryshme do të paraqesnin një interes nga pikëpamja e testimit dhe për rrjedhojë duhet të vlerësohen në mënyra të ndryshme.

Si do të grupohen gjendjet në mënyrë që të practohet vlerësimi bazuar në gjendjet e arritura?

Gjendja e objektit do të paraqitet si një vektor $v = (x_1, \ldots x_n, m)$, ku çdo x_i përfaqëson vlerën e kthimit të një prej metodave get të vendosura gjatë instrumentimit dhe m përfaqëson emrin e një metode të klasës. Ashtu siç u shpjegua më sipër nëse do të përdornim direkt vlerat e kthimit nga metodat get do të kishim një numër shumë të madh ose në shumicën e rasteve, të pafundëm gjendjesh dhe vlerësimi i kryer mbi bazën e këtyre gjendjeve do të ishte i pasaktë. Për rrjedhojë për implementim do të përdorim konceptin e abstraksionit të specifikuar nga [89] (çështja 3.6). Ky model edhe pse është

përdorur për qëllime të tjera, është i përshtatshëm edhe në kontekstin e testimit. Vlerat që do të ruhen dhe do të përdoren për përcaktimin e vlerësimit do të jenë abstraksione të vlerave të kthyera nga metodat `get`. Abstraksioni do të kryhet mbi këto tre parime:

- Nëse tipi i vlerës së kthyer nga metoda get është konkret (`int, double, long` etj), atëherë përkthimi do të bëhet në tre gjendje abstrakte:

  $x_i < 0$, $x_i = 0$ dhe $x_i > 0$.

  Për shembull, tek klasa `Staku`, vlerat e kthyera nga metoda `getmadhesia1()`, do të përkthehen në këto tre gjendje *(tek klasa Staku, pas korrigjimit të gabimeve gjendja madhësia < 0 nuk do të arrihet asnjëherë).*
- Nëse tipi i vlerës së kthyer nga metoda get është një objekt, atëherë përkthimi do të bëhet në dy gjendje abstrakte: $x_i = null$ dhe $x_i \neq null$
- Nëse tipi i vlerës së kthyer nga metoda get është `boolean`, nuk ka nevojë të bëhet përkthim, pasi këtu numri i vlerave është i kufizuar dhe të dyja vlerat paraqesin interes për testimin.

Në tabelën 4.1 paraqiten gjendjet (pa metodat) që do të mund të përftoheshin gjatë kërkimit, për gjenerimin e rasteve të testimit, pas përdorimit të abstraksionit, tek klasa Staku (pa i bërë korrigjimet). Vëmë re se ka tre kombinime gjendjesh pa përdorimin e metodave. Meqë kemi dy metoda tek kjo klasë, atëherë numri total i gjendjeve është: 3x2 = 6 gjendje.

TABELA 4.1: GJENDJET QË DO TË PËRFTOHESHIN GJATË KËRKIMIT, PAS ABSTRAKSIONIT, TEK KLASA STAKU

	getmadhesia1()	getst1()
gjendja1	= 0	≠ null
gjendja2	> 0	≠ null
gjendja3	< 0	≠ null

4.3.4 Transformime për përftimin e informacionit mbi distancën e degëve

Siç u shpjegua në paragrafin 4.1.2, distanca e degëve është një madhësi që tregon sa larg ishte ekzekutimi që të ndiqte një degë të caktuar. Meqë kjo distancë do të përdoret si një faktor për përcaktimin e vlerësimit, atëherë pas çdo ekzekutimi që kryhet gjatë kërkimit duhet të përftohet distanca e atyre degëve që përbëjnë interes në një rast të caktuar. Për rrjedhojë, duhet të kryhet sërisht instrumentimi i kodit në mënyrë që të përftohet ky informacion. Instrumentimi që do të përdoret në këtë punim bazohet tek [96]. Funksioni për llogaritjen e distancës do të jetë ai i paraqitur në tabelën 2.1 dhe tabelën 2.2.

Sa herë që në kodin i cili po instrumentohet do të gjendet një konstrukt kërcimi (p.sh. `if`), do të gjenerohet një fragment kodi që konsiston në thirrjen e një metode që bën llogaritjen e distancës së degëvë dhe do të zëvendësojë predikatën fillestare. Thirrja e kësaj metode do të bëjë njëkohësisht që të llogariten dhe të ruhen distancat e degëve dhe gjithashtu te përcaktohet vlera e vërtetë e predikatës duke përcaktuar kështu edhe rrjedhën e kontrollit. Gjatë llogaritjes së distancës, do të gjenerohet një matricë me dy elementë; përkatësisht distanca e të dy degëve. Meqë një nga dy degët do të ekzekutohet, atëherë një nga këto dy vlera do të ketë vlerën 0.

Për secilin konstrukt të Java-s që përcakton rrjedhën e programit, do të ketë një instrumentim të veçantë. Megjithatë ideja bazë e instrumentimit do të konsistojë në shtimin e një shprehjeje mbi predikatën. Kjo shprehje do të jetë thirrja e metodës për llogaritjen e distancës. Predikata do të zëvendësohet me një kontroll të thjeshtë për të përcaktuar nëse distanca është e barabartë me vlerën 0.0, pasi kjo do të thotë që vlera e predikatës është “true”.

Si shembull për ilustrimin e instrumentimit po marrim rastin e një predikate `if-then` dhe të një predicate `while`. Në figurat 4.10 dhe 4.11 jepet përkatësisht fragmenti i kodit pas instrumentimit për predikatën `if-then` dhe për predikatën `while`. DC është klasa që përmban metodat statike (p.sh. distLess) që shërbejnë për llogaritjen e distancave për operatorë të ndryshëm sipas tabelës 2.1 dhe 2.2.

```
if (v < 10) {
	//shprehje
}
```

```
double [] distanca = (DC.distLess(v, 10));
KlasaInstrumentim.ET.setDistanca(00, 01, distanca);
if (distanca[0] == 0) {
	//shprehje
}
```

Figura 4.10: Fragmenti i kodit pas instrumentimit për predikatën `if-then`

```
while (v < 10) {
	//shprehje
}
```

```
double [] distanca = (DC.distLess(v, 10));
KlasaInstrumentim.ET.setDistanca(00, 01, distanca);
boolean ePara_0 = true;
while (ePara_0 ? distanca[0] == 0 : v<10) {
	ePara_0 = false;
	{
		//shprehje
	}
}
```

Figura 4.11: Fragmenti i kodit pas instrumentimit për predikatën `while`

4.4 Skedarët që do të gjenerohen gjatë instrumentimit

Gjatë instrumentimit do të gjenerohen tre skedarë që do të përdoren nga fazat e mëvonshme. Këto tre skedarë do të jenë:

1. Skedari **.sign** që do të mbajë informacion për firmat e gjithë metodave të klasës nën testim

2. Skedari **.path** që do të mbajë gjithë varësitë e kontrollit për secilën prej degëve që janë identifikuar në klasën nën testim
3. Skedari **.tgt** që do të mbajë të gjithë targets dhe metodat ku ato bëjnë pjesë.

Skedari .sign do të ndërtohet duke marrë nga klasa nën testim emrat e metodave, listën e parametrave të tyre dhe tipin e parametrave. Për të marrë këtë informacion do të përdoren metodat e OpenJava. Duke qenë se metodat e klasës do të përdoren për ndërtimin e kromozomeve, metodat me modifikuesin `private` nuk do të merren në konsideratë, pasi ato nuk do të jenë pjesë e kromozomeve. Firma e secilës metodë do të shkruhet automatikisht në një skedar që do të ketë emrin e klasës në testim dhe prapashtesën .sign. Nëse marrim sërisht në konsideratë klasën Staku, atëherë skedari i gjeneruar (Staku.sign) që mban firmat e metodave do të kishte përmbajtjen si më poshtë:

```
Staku.push(java.lang.Integer)
Staku.pop()
```

Skedari .path do të ketë si përmbajtje numrin integer për secilën prej dëgëve dhe për secilin numër do të ketë të përcaktuara edhe numrat e degëve nga të cilat varet dega me numrin përkatës. Metoda për printimin e path-ve jepet në figurën _:

```
private void printoPathin( int tgt )
   {
    pathFile.print( tgt + ":" );
    java.util.Iterator
kontrolli=BranchTraceVisitor.getControlDependences().iterator();
    while (kontrolli.hasNext()) {
           java.lang.Integer br = (java.lang.Integer) kontrolli.next();
           pathFile.print( " " + br );
        }
        pathFile.println();
    }
```

Për klasën Staku (referohemi tek klasa Staku pas instrumentimit të degëve), skedari i gjeneruar (Staku.path) do të ketë përmbajtjen si më poshtë:

```
1:
2: 1
3: 1
4:
```

Numrat 1, 2, 3, 4 janë identifikuesit e degëve. Për shembull, dega 3 varet nga dega 1.

Skedari **.tgt** do të mbajë të gjithë qëllimet që duhet të përmbushen dhe metodat ku ato bëjnë pjesë. Ky skedar do të përdoret gjithashtu për krijimin e kromozomeve pasi siç është shpjeguar në çështjen 3.4.2, një kromozom i ndërtuar për të përmbushur një qëllim të caktuar, duhet të ketë si metodë të fundit të tij, metodën që përmban qëllimin që duhet përmbushur.

Për klasën Staku, skedari i gjeneruar (Staku.tgt) do të ketë përmbajtjen si më poshtë:

```
Staku.push(java.lang.Integer): 1, 2, 3
Staku.pop(): 4
```

4.5 Ndërtuesi i Kromozomeve

Moduli i ndërtimit të kromozomeve, do të merret nga mjeti eToc [1] dhe nuk do ti bëhen modifikime, pasi siç u shpjegua në paragrafet e mësipërme, mënyra e ndërtimit të kromozomeve nuk do të ndryshojë. Ky modul merr si input skedarin .sign dhe .tgt të gjeneruar gjatë fazës së instrumentimit. Ky modul përdor facilitetet e pasqyrimit (reflection) që ofron Java. Ndërtimi i koromozomeve do të bëhet si u shpjegua në çështjen 3.4.2. Duke përdorur informacionin e skedarit .sign do të ndërtohet një objekt i klasës nën testim. Më pas bazuar tek targeti i vendosur, do të merret informacion nga skedari .tgt dhe do të thirret metoda që e përmban targetin. Hapi pasardhës është shtimi, para thirrjes së metodës që ka targetin, i gjithë ndërtimeve të kërkuara të objekteve (duke përdorur pasqyrimin). Në fund do të shtohen rastësisht thirrje metodash për të ndryshuar gjendjen e objekteve të krijuara. Në këtë modul janë të implementuara edhe operatorët e mutacionit dhe të kryqëzimit.

4.6 Ekzekutuesi i Rasteve të Testimit

Ky modul është përgjegjës për gjenerimin e rasteve të testimit pas ndërtimit të kromozomeve. Veprimet e enkoduara në një kromozom në këtë fazë konvertohen në thirrje metodash dhe konstruktorësh. Tek kromozomi i formuar, merret një prej

veprimeve të tij dhe në varësi të rastit (nëse kemi të bëjmë me ndërtimin e një objekti ose thirrjen e një metode), gjendet nëpërmjet reflektimit konstruktori ose metoda që kanë parametrat ashtu si përcaktohen tek kromozomi. Nëse konstruktori ose metoda gjendet, ndërtohet një objekt ose thirret një metodë sipas veprimit që merr si argument. Kjo do të vazhdojë deri sa të mbarojnë gjithë veprimet e enkoduara në kromozom. Nëse ndonjë prej veprimeve nuk mund të kryhet atëherë kromozomi nuk është i vlefshëm. Ky modul ka si qëllim jo vetëm gjenerimin e rasteve të testimit nga një kromozom por edhe ekzekutimin e këtij rasti. Gjithashtu, pas ekzekutimit, thirret metoda getTrace(), e cila u vendos gjatë instrumentalizimit të kodit (çështja 4.3.2) dhe vlerat që kjo metodë kthen ruhen në një bashkësi të tipit HashSet. Secili kromozom ka si atribut set-in e përftuar nga thirrja e metodës getTrace(), pas ekzekutimit të tij. Ky atribut do të duhet për të llogaritur numrin e degëve të mbuluara nga një rast ekzekutimi për një target të caktuar dhe numri i dëgëve të mbuluara do të jetë një prej faktorëve që do të përdoren për përcaktimin e cilësisë së një rasti testimi.

Në këtë punim ky modul është ndryshuar në mënyrë që krahas gjenerimit të rastit të testimit, ekzekutimit të këtij rasti dhe përftimit të bashkësisë së degëve të ekzekutuara nga një rast testimi, të përcaktojë edhe gjendjet e arritura gjatë ekzekutimit, pasi këto gjëndje do të përdoren nga funksioni i vlerësimit të individëve. Siç u shpjegua në çështjen 4.3.3, gjendja e arritur nga objekti para thirrjes së çdo metode që bën pjesë në një rast testimi do të ruhet në një LinkedList me emër gjendjetArritura. Ruajtja do të bëhet nga metoda ktheGjendje e cila do të thirret si shprehje e parë e secilës metodë që bën pjesë në klasën nën testim.

4.7 Gjeneruesi i Testimit

Ky modul është moduli përfundimtar i mjetit dhe ka disa funksione. Gjeneruesi i testimit është përgjegjës për leximin e skedarëve që janë gjeneruar nga instrumentuesi: skedarit që mban targets dhe skedarit që mban path-et për secilën degë dhe për leximin e skedarit ku janë vendosur parametrat. Parametrat mund të ndryshohen nga përdoruesi. Ato ndikojnë në rezultatin e përftuar nga kërkimi dhe përcaktimi i tyre nuk është i thjeshtë [56]. Parametrat që ruhen në skedar janë:

- Koha maksimale e kërkimit
- Numri maksimal i përpjekjeve për mbulimin e qëllimi të caktuar
- Madhësia e popullatës

Për secilin qëllim, gjenerohet një popullatë në mënyrë të rastësishme. Më pas ekzekutohen rastet e testimit dhe kontrollohet nëse qëllimi është mbuluar; nëse po kalohet në një qëllim tjetër, në të kundërt gjenerohet një popullatë e re duke përdorur mekanizmat e rikombinimit dhe mutacionit dhe kontrollohet sërisht nëse qëllimi është mbuluar. Nëse jo, llogaritet fitness për secilin individ dhe rifreskohet numri i qëllimeve të mbuluara. Kjo proçedurë vazhdon deri sa të mbulohet qëllimi ose të ketë arritur numri maksimal i përpjekjeve për mbulimin e këtij qëllimi.

Pas përfundimit të proçesit të gjenerimit të rasteve të testimit (nuk ka më qëllime për të mbuluar ose qëllimet e pambuluara janë të pamundura), bëhet minimizimi i rasteve të gjeneruara. Minimizimi normalisht heq ato raste të cilat nuk kontribuojnë në shkallën e mbulimit të përftuar, pasi mbulojnë vetëm qëllime që mbulohen edhe nga raste të tjera. Në këtë punim proçesi i minimizimit do të ndryshohet. Siç u shpjegua në çështjen 4.2, qëllimet e kërkimit do të mbeten të njëjtat, pra degët e klasës nën testim, por duke marrë parasysh që një gjendje e re e arritur ka interes nga pikëpamja e testimit, atëherë gjithë rastet që kanë arritur një ose më shumë gjendje të reja do të ruhen gjatë minimizimit, panvarësisht nëse ato mbulojnë ndonjë qëllim të pambuluar ose jo. Secili prej rasteve të testimit do të ketë një ose disa numra që do të identifikojnë indeksin e gjendjeve tek lista e gjendjeve të arritura. Këto numra, që për çdo gjendje të re janë unike (pasi vetëm gjendjet e reja shtohen në listë), do të shërbejnë për të përcaktuar nëse një rast testimi arrin apo jo gjendje të reja dhe ky fakt do të përdoret gjatë minimizimit.

Proçedura e minimizimit është një algoritëm i thjeshtë “greedy”, i cili në mënyrë iterative përzgjedh rastin e testimit që sjell rritjen më të madhe në numrin e qëllimeve të mbuluara. Ky rast shtohet në një set të minimizuar rastesh testimi (fillimisht bosh) deri sa të arrihet niveli përfundimtar i mbulimit. Më pas për gjithë rastet e mbetura do të përcaktohet nëse ndonjë prej tyre ka gjendje të reja, duke u krahasuar me rastet që janë shtuar në set-in e minimizuar të rasteve të testimit. Si përfundim të gjithë rastet që nuk shtohen gjatë fazës së minimizimit fshihen pasi janë të tepërta. Kjo mënyrë e propozuar

në këtë punim ka disavantazhin se do të shtojë numrin e rasteve të gjeneruara, por nga ana tjetër pritet të shtojë edhe aftësinë e tyre për të detektuar gabimet. Në kapitullin 5, me anë të eksperimenteve do të vëmë re se si qëndron raporti i shtimit të numrit të rasteve të gjeneruara kundrejt rritjes së cilësisë së tyre.

Marrim sërisht në konsideratë klasën Staku (çështja 3.6). Dy rastet e testimit në figurën 4.12, mbulojnë të njëjtat degë (mbulojnë metodën pop()), prandaj në versionin origjinal të minimizimit një prej këtyre rasteve do të ishte fshirë. Në versionin e propozuar në këtë punim, të dy rastet do të shtohen tek set-i i minimizuar i rasteve të testimit, pasi rasti (a) do të shtohet se mbulon metodën pop() dhe vlerat e atributeve kur kjo metodë thirret janë: `madhesia = 0, st = !null`, ndërsa rasti (b) do të shtohet pasi vërtetë nuk përmbush ndonjë qëllim të ri, por arrin një gjendje të re që është: metoda pop() do të thirret me vlerë të atributeve: `madhësia < 0, st = !null`

```
a) public void test1()  throws Throwable  {
        Staku staku0 = new Staku();
        int int0 = staku0.pop();
   }

b) public void test3()  throws Throwable  {
        Staku staku0 = new Staku();
        staku0.pop();
        staku0.pop();
   }
```

Figura 4.12: Dy raste testimi për klasën Staku që do të ruhen gjatë fazës së minimizimit

Moduli i gjeneruesit të testimit është quajtur kështu, pasi ka edhe një funksion tjetër; kthen rastet e testimit në format JUnit dhe i shkruan në një skedar. Ky skedar është edhe outputi final i mjetit. Shtimi i pohimeve tek rastet e testimit JUnit duhet të bëhet manualisht. Siç vihet re, ky proçes manual ndodh vetëm në fund, pasi kërkimi ka përfunduar dhe është gjeneruar bashkësia e rasteve të testimit. Ekzekutimi i rasteve në

këtë fazë kryhet nga mjeti JUnit i cili gjeneron informacion mbi rastet e testimit që përfundojnë me sukses dhe rastet që dështojnë. Është e sigurtë se bashkësia e gjeneruar do të sigurojë nivelin e mbulimit që u arrit nga algoritmi gjenetik, pasi po këto raste u ekzekutuan edhe gjatë kërkimit. Nëse niveli i mbulimit nuk është niveli i kërkuar, duhet vërtetuar nëse një ndryshim i vlerave të parametrave mund të sjellë një mbulim më të mirë.

KAPITULLI 5

Eksperimentet dhe Rezultatet e Përftuara

Në kapitullin e mëprarshëm u paraqit një funksion i ri vlerësimi që mund të përdoret tek algoritmi gjenetik që përdoret për gjenerimin e rasteve të testimit për klasat në Java. Në këtë kapitull do të testohen rezultatet e këtij funksioni të ri vlerësimi. Këtu do të shpjegohet organizimi i eksperimenteve të kryera dhe do të paraqiten rezultatet e përftuara prej tyre. Qëllimi i këtij kapitulli është ti japë përgjigje pyetjeve të mëposhtme:

Pyetje për Kërkim 1: Si ndikon përdorimi i funksionit të propozuar të vlerësimit tek mbulimi i degëve?

Pyetje për Kërkim 2: Si ndikon përdorimi i funksionit të propozuar të vlerësimit te rezultati i mutacionit të arritur?

Pyetje për Kërkim 3: Si ndikon përdorimi i funksionit të propozuar të vlerësimit tek madhësia e set-it të rasteve të testimit të gjeneruara?

Pjesa e mbetur e këtij kapitulli është organizuar si vijon:

Çështja 5.1 tregon mënyrën se si janë organizuar eksperimentet duke paraqitur karakteristikat e sistemit ku janë kryer eksperimentet, subjektet e zgjedhura për testim dhe parametrat që janë përcaktuar për algoritmin gjenetik. Çështja 5.2 paraqet dhe shpjegon rezultatet e eksperimenteve. Në këtë çështje i jepet përgjigje tre pyetjeve për kërkim.

5.1 Organizimi i Eksperimenteve

A) Karakteristikat e Sistemit

Për eksperimentet është përdorur një kompjuter desktop me sistem operativ Linux me 32 bit, memorje kryesore 1GB dhe proçesor Intel Core 2 Duo CPU E7400 2.8GHzx2.

B) Përzgjedhja e Subjekteve për Testim

Përzgjedhja e subjekteve për testim (klasa ne Java), është shumë e rëndësishme pasi kjo përzgjedhje ndikon në rezultatet e përftuara nga eksperimentet. Në këtë studim janë përzgjedhur software që janë me kod të hapur.

Në mënyrë që teknikat për gjenerimin automatik të rasteve të testimit në nivel njësie të arrijnë të gjejnë një përdorim të gjerë, ato duhet të vlerësohen duke përdorur klasa reale dhe jo klasa të thjeshta ose që kanë ndonjë veçori të caktuar që ndikon në rezultatin e eksperimenteve. Gjithashtu nuk duhet bërë ndonjë ndërhyrje mbi to për të zvogëluar kompleksitetin e tyre.

Për të përftuar rezultate të vlefshme, ne zgjodhëm 6 projekte në Java. Tabela 5.1 jep për secilën prej klasave të përzgjedhura për eksperimentet, informacion mbi: emrin e projektit ku klasa bën pjesë, numrin e rreshtave (pa komente dhe rreshta bosh) të kodit të klasës, numrin e metodave publike, cyclomatic complexity, numrin e degëve, numrin e mutantëve, numrin e atributeve jo të pandryshueshme (`final`) dhe URL-në ku mund të shkarkohet projekti. Informacioni mbi numrin e rreshtave (pa komentet dhe rreshtat bosh) dhe cyclomatic complexity është përftuar duke përdorur mjetin Metrics 1.3.6, si plugin në Eclipse [97].

Siç vihet re nga tabela, një klasë (`StringTokenizer`) u morr nga paketa e Java-s `java.util`, që bën pjesë tek jdk 1.8.0. Kjo paketë është përdorur në shumë studime për vlerësimin e mënyrave të gjenerimit automatik të rasteve të testimit [85]. Gjithashtu pesë projekte u morën nga SourceForge, që është sot magazina më e madhe e projekteve me kod të hapur (ka më shumë se 300,000 projekte dhe më shumë se dy miljon përdorues të regjistruar). Një projekt u morr nga Apache Software Foundation që ekziston që nga 1999 dhe ka më shumë se 350 projekte (ndër të cilat dhe Apache HTTP Server).

Si pasojë e burimeve fizike të limituara që kishim në dispozicion dhe të kohës së konsiderueshme që nevojitej për ekzekutimin e gjithë eksperimenteve mbi një klasë, u zgjodhën në mënyrë të rastësishme 25 klasa nga projektet e listuara tek tabela 5.1. Në eksperimente u përdor edhe klasa Staku që është marrë si shembull gjatë kapitujve të mëparshëm.

TABELA 5.1: KARAKTERISTIKAT E KLASAVE TË PËRZGJEDHURA PËR EKSPERIMENTET: PROJEKTI KU BËJNË PJESË, NUMRI I RRESHTAVE TË KODIT, NUMRI I METODAVE PUBLIKE, NUMRI I DEGËVE, NUMRI I MUTANTËVE, NUMRI I ATRIBUTEVE JO-FINAL, CYCLOMATIC COMPLEXITY, URL-JA E PROJEKTIT

Projekti	Klasa	LOC	Degë	Mutantë	Atribute jo-final	Metoda publike	Cyclomatic Complexity	URL e projektit
	Staku	12	4	22	2	2	1.5	
Commons CLI	Option	155	131	140	9	42	1.52	https://commons.apache.org
	TypeHandler	124	28	28	0	9	2.66	
	AlreadySelectedException	26	4	1	2	2	1	
	OptionGroup	86	21	19	2	8	1.875	
Math4J	Rational	61	36	161	2	19	1.526	https://sourceforge.net/projects /math4j
	ExponentialFunction	40	11	31	1	9	1	
	ArrayUtil	320	167	1769	0	36	3.48	
	PolyFunction	245	100	827	2	12	3.63	
	Complex	102	24	682	2	20	1.091	
jdk (java.util v.1.8.0)	StringTokenizer	313	78	434	7	6	3.12	
Genetic Algorithm in Java	GAAlgorithm	65	14	6	6	8	2	https://sourceforge.net/projects /gaj
	Genome	14	9	21	3	4	1.4	
	Population	62	13	44	4	11	1.08	
ObjectExplorer4J	ExplorerFrame	158	26	74	8	9	1.44	https://sourceforge.net/projects /objectexplorer
	ObjectViewManager	114	41	41	8	17	1.571	
NewzGrabber	DirectoryDialog	177	47	155	16	13	2.235	https://sourceforge.net/projects /newsgrabber
	NewsFactory	121	45	88	4	7	4	
	SongInfo	55	12	59	3	4	2	
	BatchJob	28	11	29	8	10	1.27	
	StringSorter	63	12	47	1	4	2.2	
	OptionsPanel	363	75	214	15	4	9.8	
Jipa	Label	18	11	42	3	4	1.8	https://sourceforge.net/projects /jipa/
	Variable	40	23	87	3	4	2.1	
Total		276	943	5021	111	264		

C) Parametrat e AG

Tabela 5.2, jep vlerat e parametrave kryesorë të algoritmit gjenetik. Për sa i përket buxhetit të kërkimit, ai u vendos në varësi të eksperimentit dhe do të tregohet për secilin eksperiment. Përcaktimi i parametrave të AG për përftimin e rezultateve optimal, është një problem i vështirë [56] dhe nuk është qëllim i këtij studimi, prandaj janë përdorur vlerat default të përcaktuara tek [1].

TABELA 5.2: PARAMETRAT E ALGORITMIT GJENETIK

Parametri	Vlera
Madhësia e popullatës	10
Koha maksimale e kërkimit	**600s**
Numri maksimal i gjenerimeve për një qëllim	10

5.2 Rezultate dhe Diskutime

Pyetje për Kërkim 1: Si ndikon përdorimi i funksionit të propozuar të vlerësimit tek mbulimi i degëve?

Shtimi i faktorit të gjendjeve të reja të arritura, tek funksioni i vlerësimit, u krye me qëllim që gjatë kërkimit, ti jepej përparësi rasteve që arrinin gjendje të pazbuluara të objektit të klasës nën testim. Megjithatë qëllimet e kërkimit edhe në këtë rast do të jenë mbulimet e degëve. Prandaj për të testuar efektshmërinë e funksionit të ri të vlerësimit duhet edhe të studiojmë efektin e tij mbi mbulimin e degëve krahasuar me funksionin e vlerësimit për kriterin e mbulimit të degëve.

Proçedura e eksperimentit: Për secilën prej klasave të përzgjedhura për eksperimentet (tabela 5.1), u ekzekutua mjeti eToc me funksionin e vlerësimit origjinal dhe me funksionin e vlerësimit të propozuar. Për të tejkaluar faktin që rezultati që gjenerohet nga algoritmi gjenetik mbi një klasë është i rastësishëm, secili eksperiment, mbi secilën klasë, u përsërit 5 herë.

U përdor një kohë kërkimi prej **2 min dhe 10 min**. Sipas [1], zakonisht është e pranueshme që aktiviteti për gjenerimin e rasteve të testimit, të zgjasë midis disa minutave deri në një orë. Dihet që koha e kërkimit është një faktor që ndikon në rezultatet e gjeneruara nga algoritmi gjenetik, prandaj koha u zgjodh e tillë në mënyrë që rezultatet të mos ishin pasojë e një kohe të limituar kërkimi, pasi qëllimi i këtij eksperimenti është monitorimi i mbulimit të arritur të degëve dhe nuk ka për fokus primar kohën në të cilën arrihet ky mbulim. U zgjodhën dy kohë kërkimi për të parë nëse rasti me funksionin e propozuar të vlerësimit kërkon më shumë kohë krahasuar me rastin e pamodifikuar për të arritur të njëjtin mbulim degësh. Për të përftuar mbulimin e arritur është përdorur si plugin në Eclipse, mjeti EclEmma [72].

TABELA 5.3: MBULIMI I DEGËVE (MD) DUKE PËRDORUR FUNKSIONIN E VLERËSIMIT ORIGJINAL (FVO) DHE FUNKSIONIN E VLERËSIMIT TË PROPOZUAR (FVP), ME NJË BUXHET KËRKIMI (BK) PREJ 2 MIN DHE 10 MIN

Klasa	Dege	Mutantë	MD për FVO me BK = 2 min	MD për FVP me BK = 2 min	MD për FVO me BK = 10 min	MD për FVP me BK = 10 min
Staku	4	22	100	100	100	100
Option	131	140	69	69	69	69
TypeHandler	28	28	75	75	75	75
AlreadySelectedException	4	1	100	100	100	100
OptionGroup	21	19	100	100	100	100
Rational	36	161	94	94	94	94
ExponentialFunction	11	31	100	100	100	100
ArrayUtil	167	1769	100	99	100	100
PolyFunction	100	827	-	-	85	87
Complex	24	682	100	100	100	100
StringTokenizer	78	434	65	65	69	69
GAAlgorithm	14	6	93	93	93	93
Genome	9	21	44	44	55	55
Population	13	44	92	92	100	100
ExplorerFrame	26	74	8	15	8	15
ObjectViewManager	41	41	54	54	54	54
DirectoryDialog	47	155	6	6	6	6
NewsFactory	45	88	-	-	-	-
SongInfo	12	59	50	50	50	50
BatchJob	11	29	100	100	100	100
StringSorter	12	47	100	100	100	100
OptionPanel	75	214	-	-	37	37
Label	11	42	100	100	100	100
Variable	23	87	100	100	100	100
Mesatarja			***60.5***	***69***	***74.8***	***75.2***

Klasat e përzgjedhura kanë një numër degësh të ndryshëm, që shtrihet në intervalin nga 4 deri në 167 degë, në mënyrë që të arrijmë në një konkluzion sa më të saktë dhe të përgjithshëm mbi efektin e funksionit të propozuar të vlerësimit tek mbulimi i arritur i degëve.

Siç mund të vëmë re nga tabela, mbulimi i arritur i degëve është i konsiderueshëm. Mesatarja për 24 klasat, me një buxhet kërkimi prej 10 min, duke përdorur funksionin origjinal të vlerësimit është 74.8%, ndërsa duke përdorur funksionin e propozuar është 75.2%. Ndryshimi midis këtyre dy konfigurimeve është vetëm 0.4%. Një diferencë e tillë kaq e vogël, pritej, pasi tek konfigurimi i ri, nuk është ndryshuar bashkësia e qëllimeve të mbulimit; kjo bashkësi përbëhet sërisht nga degët e klasës që është nën testim, por gjatë kërkimit do të vlerësohen më shumë rastet që e vendosin objektin në gjendje të reja. Vëmë re se për klasën ArrayUtil (klasa me numrin më të madh të degëve, 167), mbulimi i përftuar më konfigurimin e dytë është 1% më i ulët, ose 2 degë më pak. Klasa ArrayUtil nuk ka asnjë atribut, prandaj bazuar tek mënyra se si është implementuar funksioni i propozuar i vlerësimit, mund të themi me siguri se në këtë rast funksioni i propozuar është identik me funksionin origjinal. Gjithashtu, edhe instrumentimi i kryer për monitorimin e gjendjeve, kur klasa nuk ka asnjë atribut, nuk shton asnjë shprehje tek klasa. Ndryshimi me 1% tek mbulimi i përftuar vjen si pasojë e faktit që rezultatet e përftuara nga algoritmi gjenetik janë të rastit dhe jo të përcaktuara, edhe pse në tabelë është marrë mesatarja e mbulimit pas pesë gjenerimeve. Pra, ky ndryshim i detyrohet rastit dhe jo implementimit të funksionit të vlerësimit.

Vëmë re se për klasat StringTokenizer, Genome dhe Population, përdorimi i një buxheti kërkimi prej 10 min sjell një rritje në përqindjen e mbulimit përkatësisht me 3%, 10% dhe 8%.

Për klasën PolyFunction dhe OptionPanel, buxheti prej 2 min, nuk mjafton për të përftuar një set rastesh testimi pasi faza e gjenerimit prodhon shumë pak individë të cilët nuk arrijnë të përmirësohen dhe për rrjedhojë faza e minimizimit dështon. Ky fakt ndodh edhe në rastin e përdorimit të funksionit origjinal edhe në rastin e përdorimit të funksionit të propozuar të vlerësimit. Koha 2 min është një kohë e shkurtër për gjenerimin e rasteve të testimit, prandaj është e kuptueshme që për disa klasa duhet një kohë më e madhe. Numri i konsiderueshëm i degëve që kanë këto klasa, përkatësisht 100 dhe 75, mund të jetë një prej arsyeve që kërkimi ka nevojë për një kohë më të madhe se 2 min.

Në rastin e klasës NewsFactory, proçesi i gjenerimit dështon si për buxhetin 2 min, ashtu edhe për buxhetin 10 min për të dy funksionet e vlerësimit. Për këtë klasë u provua të ndryshohej madhësia e popullatës (një prej parametrave të algoritmit gjenetik) dhe të bëhej nga 10 në 30 individë. Megjithatë edhe pas këtij ndryshimi proçesi dështoi. Edhe pse numri i degëve dhe numri i mutantëve nuk është shumë i madh, asryet e këtij dështimi mund të jenë të ndryshme. Bazuar edhe tek eksperimentet e autorëve të tjerë me mjete të ndryshme të testimit automatik [78], ekzistojnë klasa ku deri më sot gjenerimi automatik dështon plotësisht. Nuk është qëllim i këtij punimi, evidentimi i arsyeve konkrete të këtij dështimi, pasi meqë dështimi ndodh edhe për rastin kur përdoret funksioni origjinal i vlerësimit, kuptojmë se shkaku nuk është funksioni i propozuar i vlerësimit.

Në rastin e klasës ExplorerFrame, vëmë re se edhe pse mbulimi i arritur është i ulët, rasti me funksionin e vlerësimit të propozuar arrin nje mbulim 7% (2 degë) më të lartë se rasti me funksionin origjinal. Një rezultat i tillë mund të shpjegohet në dy mënyra. Një arsye është se rezultatet e përftuara janë të rastit. Arsyeja tjetër është se edhe pse qëllimet e mbulimit janë të njëjta për të dy funksionet, rasti me funksionin e propozuar, i jep përparësi rasteve që plotësojnë edhe një kusht shtesë, për rrjedhojë mund të ndodhë që rastet e përftuara gjatë kërkimit dhe pas minimizimit, të jenë të ndryshme dhe më komplekse se rasti i përdorimit të funksionit origjinal, prandaj në këtë rast mund të jetë shfaqur fenomeni i ekzistencës së mbulimit indirekt. Duke qenë se kjo klasë ka 8 gjendje ekziston mundësia që rezultati të vijë si pasojë e arsyes së dytë të sipërpërmendur. I njëjti shpjegim vlen edhe për klasën StringTokenizer.

Nga rezultatet e përftara shohim se në 92% të klasave mbulimi i arritur i degëve është identik për të dy funksionet (në rastin e buxhetit të kërkimit prej 10 min). Në 8% të klasave ndryshimi i mbulimit të degëve është shumë i vogël dhe mund të jetë si pasojë e rezultateve të rastit që gjeneron algoritmi gjenetik.

Për të vizualizuar më qartë shpërndarjen e rezultateve të eksperimenteve po paraqesim diagramën box plot për të katër konfigurimet (Mbulimi i Degëve (MD) duke përdorur funksionin e vlerësimit origjinal (FVO) dhe funksionin e vlerësimit të propozuar (FVP), me një buxhet kërkimi (BK) prej 2 min dhe 10 min). Diagrama box plot është një mënyrë e standartizuar për paraqitjen e shpërndarjes së të dhënave bazuar në pesë vlera: minimumi, vlera e mesit (Q2),

maksimumi, vlera e mesit (Q1) midis minimumit dhe Q2 dhe vlera e mesit (Q3) midis Q2 dhe maksimumit.

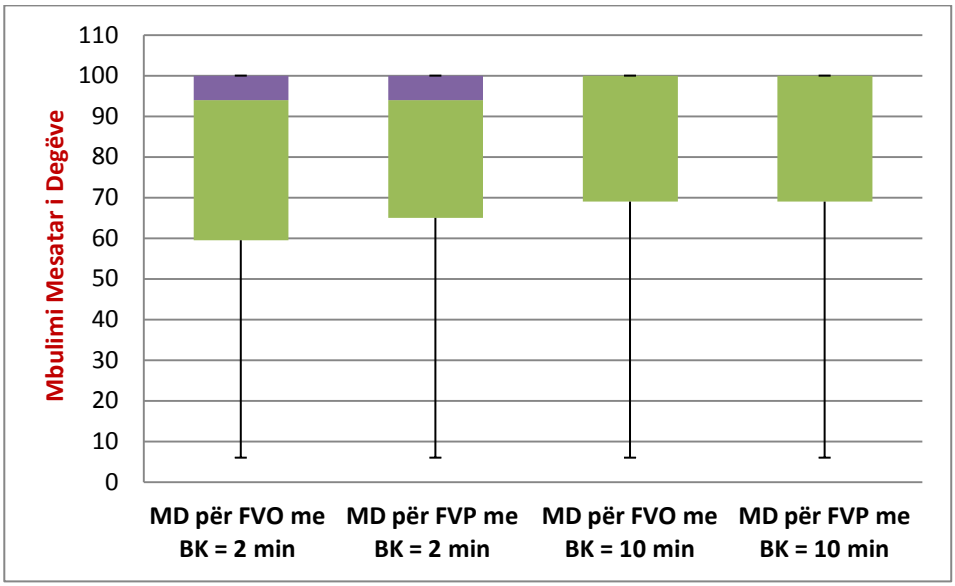

Figura 5.1: Shpërndarja e mbulimit mesatar të degëve

<u>Pyetje për Kërkim 1:</u> Në eksperimentet tona, nuk ka ndryshim në mbulimin e degëve të arritur duke përdorur funksionin e propozuar të vlerësimit kundrejt mbulimit të arritur duke përdorur funksionin origjinal të vlerësimit.

Pyetje për Kërkim 2: Si ndikon përdorimi i funksionit të propozuar të vlerësimit te rezultati i mutacionit të arritur?

Siç u shpjegua në çështjen 3.2, numri i mutantëve të vrarë nga rastet e testimit për një klasë, kundrejt numrit total të mutantëve për atë klasë (d.m.th. rezultati i mutacionit), është kriteri më i fortë për vlerësimin e rasteve të testimit. Për këtë arsye, me anë të eksperimenteve të shpjeguara më poshtë do të përpiqemi të vlerësojmë funksionin e propozuar të vlerësimit duke përdorur këtë kriter. Rezultati i mutacionit paraqet aftësinë e testeve për të detektuar gabimet dhe për rrjedhojë tregon cilësinë e tyre.

Proçedura e eksperimentit: Për secilën prej klasave të përzgjedhura për eksperimentet (tabela 5.1), do të llogaritet rezultati i mutacionit i arritur nga rastet e testimit të gjeneruara duke përdorur mjetin eToc, me funksionin origjinal të vlerësimit dhe me funksionin e propozuar. Duke

marrë në konsideratë faktin që rezultati i kërkimit është i rastësishën, secili eksperiment u përsërit 5 herë. Për secilën klasë do të gjenerohen mutantët duke përdorur mjetin MuClipse v1.3 [98], i cili përdoret si një plugin në Eclipse. MuClipse do të gjenerojë mutantët duke përdorur operatorët tradicionalë të muatacionit dhe ato në nivel klase (operatorët që dëshirojmë të përdorim, zgjidhen para fazës së gjenerimit të mutantëve). Operatorët e mutacionit shpjegohen tek [99]. Operatorët që mjaftojnë për të vlerësuar efektivitetin e set-it të rasteve të testimit përcaktohen tek [100]. Me anë të MuClipse, marrim kodin burim të secilit prej mutantëve dhe gjithashtu pas ekzekutimit përftojmë edhe rezultatin mbi numrin e mutantëve të vrarë ose jo. Në mënyrë që ratstet e testimit të gjeneruara automatikisht të mund të përdoren nga MuClipse për të llogaritur rezultatin e mutacionit, atyre duhet t'i shtohen pohimet (assertions), pasi ato nuk gjenerohen automatikisht. Për këtë arsye pohimet u shtuan manualisht në secilën prej metodave të testimit të gjeneruara. Ekzistojnë sot edhe përpjekje për të gjeneruar automatikisht oracle (rezultatet e duhura pas ekzekutimit me nje rast testimi) dhe për rrjedhojë edhe pohimet, megjithatë saktësia e tyre ende nuk është provuar në mënyrë bindëse [101] [102], prandaj për të mos rrezikuar që rezultatet e eksperimentit të jenë të pa sakta, shtimi i pohimeve u bë manualisht. Koha për realizimin e këtij proçesi varet nga klasa nën testim edhe për disa klasa kjo kohë ishte e konsiderueshme (më shumë se 30 minuta). Proçedura për shtimin e pohimeve është:

- Çdo herë që një thirrje metode kthen një vlerë, shtohet një pohim i cili krahason vlerën e kthyer me vlerën që pritet të kthehet.
- Çdo herë që një thirrje metode mund të hedhë një përjashtim, shtohet një bllok try-catch që përfshin këtë thirrje metode.

Mutantët e gjeneruar do të ekzekutohen me anë të JUnit dhe prezenca e dështimeve do të tregojë se rastet e testimit ishin në gjendje të kapnin gabimet që përmbajnë mutantët.

Koha për ekzekutimin e rasteve të testimit të gjeneruara kundrejt gjithë mutantëve është e madhe. Vetëm për klasën Staku, u gjeneruan 4 mutantë në nivel klase dhe 18 mutantë nga operatorët tradicionalë, pra në total 22 mutantë.

Në tabelën 5.4, jepen për secilën klasë, rezultati i mutacionit të përftuar duke ekzekutuar setin e rasteve të testimit të gjeneruara duke përdorur mjetin eToc me funksionin e vlerësimit origjinal edhe me funksionin e vlerësimit të propozuar. Rezultatet jepen për rastin kur është përdorur për kërkim koha 2 min edhe për rastin kur është përdorur koha 10 min.

TABELA 5.4: REZULTATI I MUTACIONIT (RM) DUKE PËRDORUR FUNKSIONIN E VLERËSIMIT ORIGJINAL (FVO) DHE FUNKSIONIN E VLERËSIMIT TË PROPOZUAR (FVP), ME NJË BUXHET KËRKIMI (BK) PREJ 2 MIN DHE 10 MIN

Klasa	Mutantë	RM për FVO me BK = 2 min	RM për FVP me BK = 2 min	RM për FVO me BK = 10 min	RM për FVP me BK = 10 min	Ndryshimi
Staku	22	29	72	29	72	10
Option	140	41	49	41	49	12
TypeHandler	28	46	46	46	46	0
AlreadySelectedException	1	100	100	100	100	0
OptionGroup	19	84	89	84	89	1
Rational	161	75	79	75	79	8
ExponentialFunction	31	60	55	60	60	0
ArrayUtil	1769	9	9	9	9	0
PolyFunction	827	-	-	31	38	58
Complex	682	34	37	34	37	22
StringTokenizer	434	15	21	19	23	20
GAAlgorithm	6	33	33	33	50	1
Genome	21	0	4	0	4	1
Population	44	32	32	32	32	0
ExplorerFrame	74	0	3	0	3	2
ObjectViewManager	41	17	24	17	24	3
DirectoryDialog	155	0	0	0	0	0
SongInfo	59	22	27	24	27	2
BatchJob	29	62	69	62	69	2
StringSorter	47	17	17	17	17	0
OptionPanel	214	-	-	3	9	12
Label	42	55	55	55	55	0
Variable	87	55	56	56	59	3
Mesatarja		***34.1***	***38.1***	***35.9***	***41.5***	

Rezultatet e mutacionit duke përdorur të dy funksionet e vlerësimit janë larg vlerave optimale (100%). Një përqindje e përafërt vihet re edhe në studime të tjera [103]. Arsyet kryesore në këtë rast janë dy: e para që qëllimet e mbulimit janë degët dhe jo vrasja e mutantëve dhe e dyta ekzistenca e mutantëve ekuivalentë (që sillen njëlloj me programin e pa modifikuar). Megjithatë, interesi ynë fokusohet tek ndryshimi i arritur në rezultatin e mutacionit midis rastit kur përdoret funksioni i vlerësimit origjinal dhe ai i propozuar.

Nga tabela 5.4, vëmë re se ka një ndryshim në rezultatet e përftuara kur kemi buxhetin e kërkimit prej 2 min dhe 10 min. Eshtë e pritshme që rezultati të përmirësohet kur kemi një kohë

kërkimi më të madhe, po kjo nuk ndodh në 100% të rasteve. Në eksperimentin tonë ky përmirësim vihet re në 6/23 (26%) të rasteve. Më shumë rëndësi për ti dhënë përgjigje pyetjes së kërkimit, ka rasti kur përdoret koha 10 min. Rezultati mesatar i mutacionit të përftuar duke përdorur funksionin origjinal është 35.9%, ndërsa dukë përdorur funksionin e propozuar në këtë punim, është 41.5%, pra një ndryshim prej 5.6%. Përmirësimi i arritur është 5.6%/35.9% = 15.6%. Në 15 klasa nga 23 në total (65%), ka një përmirësim midis rezultateve të arritura duke përdorur dy funksionet e vlerësimit dhe në 8 raste (35%), rezultati është identik. Nuk ka asnjë rast kur funksioni i propozuar sjell një ulje në rezultatin e mutacionit të përftuar. Edhe pse jemi të vetëdijshëm se rezultatet e eksperimentit varen nga subjektet nën testim (panvarësisht se për të zvogëluar varësinë janë zgjedhur subjekte me karakteristika të ndryshme), mund të themi se rezultatet janë premtuese.

Për të vizualizuar më qartë shpërndarjen e rezultateve të eksperimenteve, në figurën 5.2 paraqitet diagrama box plot për rezultatin e mutacionit të arritur duke përdorur katër konfigurimet (Rezultati i Mutacionit (RM) duke përdorur funksionin e vlerësimit origjinal (FVO) dhe funksionin e vlerësimit të propozuar (FVP), me buxhet kërkimi (BK) prej 2 min dhe 10 min).

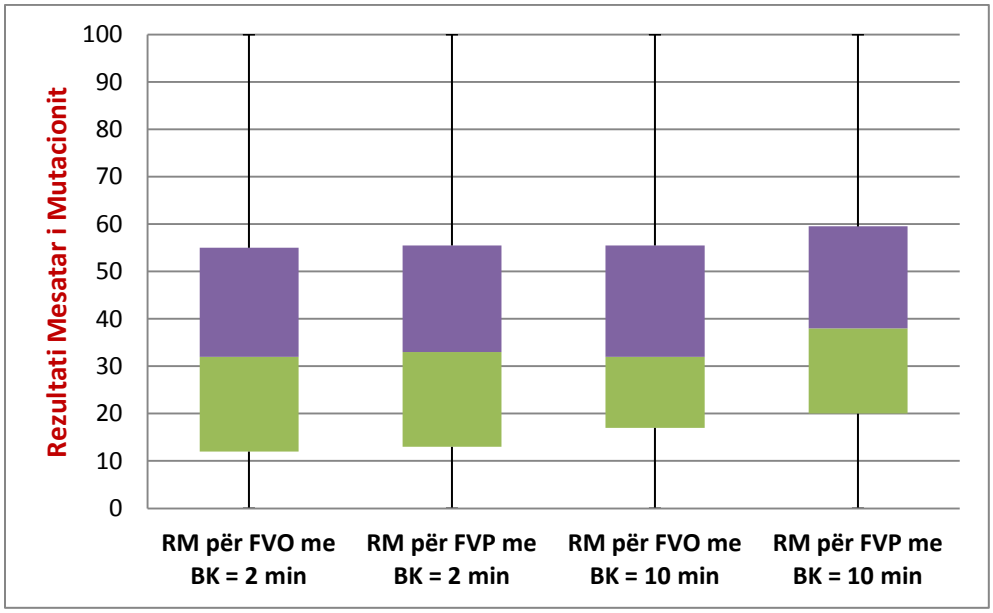

Figura 5.2: Shpërndarja e rezultatit të mutacionit

Nga grafiku i paraqitur në figurën 5.2, vihet re se për buxhetin e kërkimit prej 10 min, përdorimi i funksionit të propozuar të vlerësimit kundrejt funksionit origjinal, edhe pse ka të njëjtin minimum dhe maksimum, ka vlerën e mesit, vlerën Q1 dhe vlerën Q3 më të lartë.

<u>Pyetje për Kërkim 2:</u> Në eksperimentet tona, përdorimi i funksionit të propozuar të vlerësimit sjell një rritje relative prej 15.6% në rezultatin e mutacionit të përftuar kundrejt përdorimit të funksionit origjinal të vlerësimit.

Pyetje për Kërkim 3: Si ndikon përdorimi i funksionit të propozuar të vlerësimit tek madhësia e set-it të rasteve të testimit të gjeneruara?

Testet e gjeneruara automatikisht duhet të kontrollohen manualisht për të detektuar gabimet, pasi në ditët e sotme akoma nuk është e mundur të gjenerohet automatikisht oracle. Për këtë arsye, është shumë e rëndësishme që rastet e gjeneruara jo vetëm të kenë mbulim dhe cilësi të lartë, por të kenë edhe gjatësi sa më të vogël, në mënyrë që të mund të bëhet kontrolli manual i tyre.

Këtu me termin *madhësi e rastit të testimit*, i referohemi numrit të shprehjeve në një rast testimi pas fazës së minimizimit (pa përfshirë pohimet).

Gjithashtu, rëndësi ka edhe që minimizimi të kryhet në mënyrë sa më të saktë dhe të eleminojë rastet e testimit që nuk mbulojnë ndonjë qëllim që është i pa mbuluar nga rastet e tjera [104]. Megjithatë, përveç rasteve që duhen hequr gjatë minimizimit, numri i rasteve të testimit të gjeneruara pas kësaj faze nuk mund të merret si faktor për vlerësimin e cilësisë së setit të gjeneruar nga një teknikë e caktuar. Rëndësi ka gjatësia e rasteve, pasi gjenerimi i shumë rasteve të testimit që janë të shkurtra dhe të thjeshta nuk është një problem për testuesin që duhet të shtojë pohimet. Prandaj për vlerësimin e funksionit të propozuar fokusi do të jetë kryesisht tek gjatësia e rasteve të gjeneruara.

Proçedura e eksperimentit: Për secilën prej klasave të përzgjedhura për eksperimentet (tabela 5.1), u ekzekutua mjeti eToc me funksionin e vlerësimit origjinal dhe me funksionin e vlerësimit të propozuar. Duke marrë në konsideratë faktin që rezultati i kërkimit është i rastësishëm, secili eksperiment u përsërit 5 herë. U përdor një kohë kërkimi prej **10 min**. Në këtë eksperimeint nuk u përdor koha e kërkimit prej 2 min, pasi jemi të interesuar për numrin e rasteve dhe gjatësinë e tyre në skenarin "më të keq". Koha e fazës së minimizimit nuk është pjesë e buxhetit të kërkimit, prandaj nuk varet prej tij. Informacioni mbi numrin e rreshtave (pa

komentet dhe rreshtat bosh) të set-it të gjeneruar, është përftuar duke përdorur mjetin Metrics 1.3.6 si plugin në Eclipse [97].

Në tabelën 5.5, jepet mesatarja për 5 ekzekutime e numrit të rasteve të testimit dhe e gjatësisë së tyre për secilën prej klasave nën testim, me një buxhet kërkimi prej 10 min.

Gjatë ekzekutimit të eToc për gjenerimin e rasteve të testimit për klasën NewsFactory, faza e minimizimit kaloi kohën maksimale të përcaktuar duke mos arritur të kryejë minimizimin edhe në rastin e përdorimit të funksionit origjinal edhe në rastin e përdorimit të funksionit të propozuar. Ky rast nuk u morr në konsideratë.

Nga rezultatet e eksperimentit, të paraqitura në tabelën 5.5, vëmë re se kemi një rritje prej 314 – 290 = 24 ratsesh testimi në total, ose një rritje relative prej 24/290 = 8.2%. Kjo rritje në numrin e rasteve është e pranueshme, megjithatë siç u shpjegua edhe në paragrafin e mësipërm, numri i rasteve të testimit të gjeneruara nuk është një faktor i cili duhet të merret në konsideratë për të vlerësuar efektin e përdorimit të një teknike të re në vendosjen e pohimeve nga testuesi.

Rezultatet e përftuara tregojnë se nuk kemi një rritje të madhe në madhësinë e set-it të rasteve të gjeneruara. Përdorimi i funksionit të propozuar të vlerësimit e rrit mesataren e gjatësisë së set-it të gjeneruar nga 30.11 shprehje në 33.9 shprehje. Rritja relative është (33.9 – 30.1) / 30.1 = 12.6%.

Për 8 klasa (34% të klasave), nuk kemi ndryshim në gjatësinë e set-it të gjeneruar. Për këto klasa nuk ka ndryshim as në mbulimin e degëve as në rezultatin e mutacionit të përftuar; rastet e testimit për këto 8 klasa janë identike, pra funksioni i propozuar nuk sjell asnjë ndryshim.

Në rastin e klasave ExponentialFunction dhe GAAlgorithm (8.7% e klasave), vëmë re se kemi një ulje në numrin e rasteve dhe gjatësinë e set-it të gjeneruar, panvarësisht se nuk kemi ulje tek mbulimi i degëve ose tek rezultati i mutacionit. Në këto raste është shfaqur fenomeni i ekzistencës së mbulimit indirekt, pasi rastet e gjeneruara në të dy rastet janë të ndryshme.

TABELA 5.5: MESATARJA PËR 5 EKZEKUTIME E NUMRIT TË RASTEVE TË TESTIMIT DHE E GJATËSISË SË TYRE PËR SECILËN PREJ KLASAVE NËN TESTIM, DUKE PËRDORUR FUNKSIONIN E VLERËSIMIT ORIGJINAL (FVO) DHE FUNKSIONIN E VLERËSIMIT TË PROPOZUAR (FVP), ME NJË BUXHET KËRKIMI PREJ 10 MIN

Klasa	Atribute jo-final	Mutantë	Dege	Nr. test me FVO	Gjatësia e rastve me FVO	Nr. test me FVP	Gjatësia e rastve me FVP
Staku	2	22	4	2	8	4	15
Option	9	140	131	62	147	71	166
TypeHandler	0	28	28	12	24	12	24
AlreadySelectedException	2	1	4	3	5	3	5
OptionGroup	2	19	21	8	27	7	35
Rational	2	161	36	12	24	12	31
ExponentialFunction	1	31	11	8	16	7	15
ArrayUtil	0	1769	167	64	141	64	141
PolyFunction	2	827	100	27	89	30	98
Complex	2	682	24	13	27	12	31
StringTokenizer	7	434	78	8	18	16	33
GAAlgorithm	6	6	14	10	21	8	19
Genome	3	21	9	3	6	4	10
Population	4	44	13	11	29	11	29
ExplorerFrame	8	74	26	2	2	2	3
ObjectViewManager	8	41	41	2	3	2	3
DirectoryDialog	16	155	47	5	11	5	11
NewsFactory	4	88	45	-	-	-	-
SongInfo	3	59	12	5	12	8	19
BatchJob	8	29	11	10	20	9	22
StringSorter	1	47	12	6	17	6	17
OptionPanel	15	214	75	7	21	8	19
Label	3	42	11	4	16	4	16
Variable	3	87	23	6	9	9	19
Total			***786***	***290***	***693***	***314***	***781***

Edhe në rastin e klasave ku kemi një rritje në numrin e shprehjeve të set-it të gjeneruar, vihet re një rritje jo e konsiderueshme. Ndryshimi më i madh ka ndodhur për klasën Option, ku kemi një rritje prej 19 shprehjesh.

Set-i i rasteve më i madh, është gjeneruar për klasën ArrayUtil (64 raste/141 shprehje). Një gjë e tillë pritej pasi kjo është klasa me numrin më të madh të degëve (167 degë). Kjo klasë ka

dhe numrin më të madh të mutantëve, por ky fakt, për të dy funksionet, nuk ndikon në rastet e gjeneruara.

Rritja në numrin e rasteve të gjeneruara dhe në gjatësinë e tyre kur përdoret funksioni i propozuar i vlerësimit, kundrejt rastit kur përdoret funksioni origjinal, vjen si pasojë e dy arsyeve: e para se në implementimin e propozuar, gjatë minimizimit, do të ruhen rastet që nuk sjellin ndonjë rritje në mbulimin e degëve, por e kanë vendosur objektin në një ose disa gjendje të reja dhe arsyeja e dytë është se rastet e gjeneruara mund të jenë të ndryshme, si pasojë e përdorimit të dy funksioneve të vlerësimit të ndryshme e për rrjedhojë mund të kenë edhe numër shprehjesh jo të njëjtë (jo detyrimisht numër më të madh).

Për të vizualizuar më qartë shpërndarjen e rezultateve të eksperimenteve, në figurën 5.3 paraqitet diagrama box plot për mesataren e gjatësisë së rasteve të testimit në dy konfigurimet (Gjatësia e Rasteve duke përdorur funksionin e vlerësimit origjinal (FVO) dhe funksionin e vlerësimit të propozuar (FVP), me një buxhet kërkimi (BK) prej 10 min).

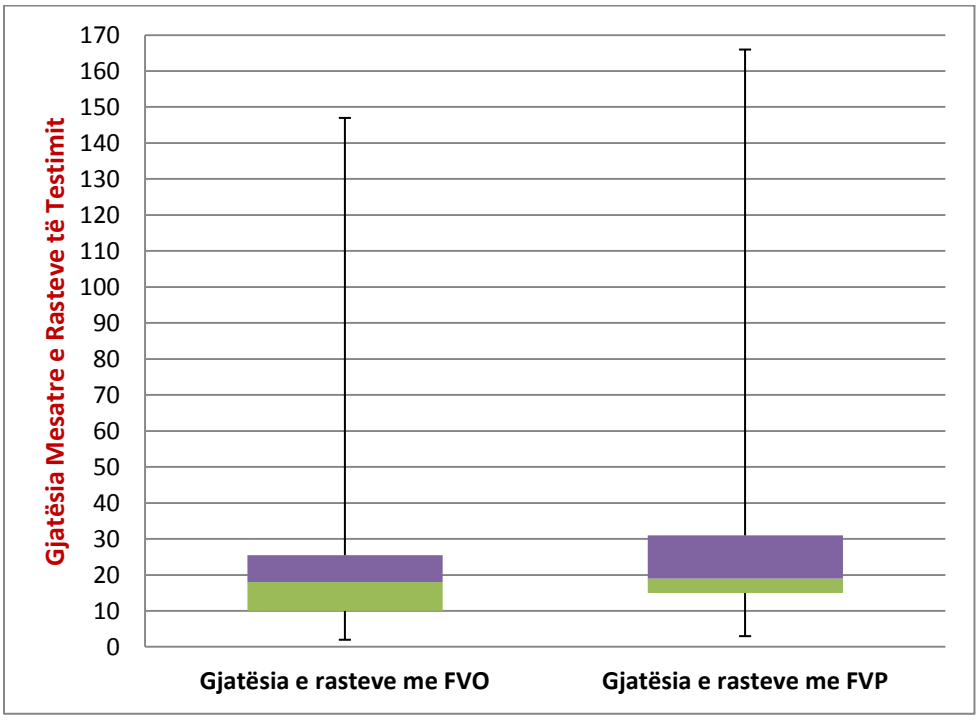

Figura 5.3: Shpërndarja e gjatësisë së rasteve të testimit

Nga grafiku i paraqitur në figurën 5.3, vihet re se shpërndarja e vlerave të gjatësisë së rasteve të testimit duke përdorur funksionin e propozuar të vlerësimit kundrejt funksionit origjinal, ka

vlerën minimale, vlerën maksimale, vlerën e mesit, vlerën Q1 dhe vlerën Q3 më të lartë. Megjithatë ndryshimet nuk janë të mëdha, veçanërisht për vlerat Q1 dhe Q3.

Pyetje për Kërkim 3: Në eksperimentet tona, përdorimi i funksionit të propozuar të vlerësimit sjell një rritje në gjatësinë e bashkësisë së rasteve të testimit me 12.6%.

KAPITULLI 6

Konkluzione dhe Puna në të Ardhmen

Në këtë punim disertacioni, trajtohet funksioni i vlerësimit që përdoret për të drejtuar kërkimin gjatë gjenerimit automatik të rasteve të testimit për klasat në Java. Kriteri më i përdorur i mbulimit, mbulimi i degëve, është i thjeshtë për t'u implementuar, por mund të gjenerojë teste "të dobëta". Për këtë arsye në këtë punim propozohet një funksion i ri vlerësimi i cili, gjatë ekzekutimit të një rasti testimi, merr në konsideratë edhe gjendjet e reja të arritura nga objekti nën testim. Bazuar në idenë e programimit të orientuar nga objekti, dihet se rastet e testimit që e vendosin objektin në gjendje të reja mund të nxjerrin në pah tipare/sjellje të fshehura të klasës nën testim dhe për rrjedhojë paraqesin interes në kontekstin e testimit.

Funksioni i propozuar do të llogaritet si një kombinim linear i tre faktorëve: nivelit të afërsisë, distancës së degëve (të normalizuar) dhe numrit të gjendjeve të reja të arritura (të normalizuar). Qëllimet e mbulimit do të mbeten degët, pasi nëse do të shtonim si qëllime mbulimi kombinimin e gjendjeve të mundshme të objektit, atëherë shumë prej qëllimeve do të mbeteshin të pamundura. Kjo për arsye se një objekt nuk mund të arrijë gjithë gjendjet, edhe për rrjedhojë do të kishim një keqpërdorim të burimeve të testimit. Gjithashtu, për përcaktimin ose jo të një gjendjeje të arritur si "gjendje e re", u krye abstarksioni i të dhënave, në mënyrë që termi gjendje e re të përshtatej me kontekstin e testimit. Instrumentimi për të mundësuar marrjen e informacionit që i nevojitet funksionit të propozuar të vlerësimit, u krye në nivel kodi burim, duke përdorur mundësitë që ofron pasqyrimi në Java. Gjatë minimizimit u quajtën të vlefshme jo vetëm rastet e gjeneruara që sjellin një rritje në mbulimin e dëgëve, por edhe ato raste që e vendosin objektin në gjendje të reja.

Performanca e funskionit të propozuar të vlerësimit, u vlerësua duke llogaritur mbi bashkësinë e steteve të gjeneruara, vlerat e tre madhësive:

- mbulimin e arritur të degëve
- rezultatin e mutacionit
- gjatësinë e rasteve të gjeneruara

Eksperimentet u kryen mbi 24 subjekte me kod të hapur dhe me karakteristika të ndryshme (numër degësh, numër atributesh jo-finale, LOC, cyclomatic complexity, numër mutantësh, numër metodash publike).

6.1 Përfundimet e arritura

Përfundimet e arritura gjatë punimit janë:

1. Për përmirësimin e cilësisë së rasteve të gjeneruara, kërkohet të gjenden funksione të reja vlërësimi, pasi rezulatati i mutacionit të përftuar nga eksperimentet duke përdorur funksionet e vlerësimit të cilat janë implementuar, gjithashtu edhe një kombinim linear të tyre, është ende më i vogël se 30%.
2. Mbulimi i degëve është i thjeshtë për t'u implementuar, por arritja e një mbulimi të lartë ose të plotë të degëve nuk siguron që testet kanë cilësi të lartë.
3. Rastet e testimit që e vendosin objektin në gjendje të reja nxjerrin në pah sjellje të fshehura të tij dhe mund të përdoren për zbulimin e gabimeve.
4. Përdorimi i funksionit të propozuar të vlerësimit kundrejt funksionit origjinal që bazohet në mbulimin e degëve:
    a. Nuk sjell ulje në mbulimin e degëve
    b. Sjell një rritje relative prej 15.6% në rezultatin e mutacionit
    c. Sjell një rritje relative prej 12.6% në gjatësinë mesatare të rasteve të gjeneruara
5. Si pasojë e kompleksitetit dhe e shumëllojshmërisë së klasave në Java, është e nevojshme të bëhen ende shumë përmirësime në mënyrë që të ketë konfidencialitet në përdorimin real të automatizimit për testimin e tyre.

6.2 Puna në të Ardhmen

Ky punim mund të zgjerohet më tej në të ardhmen në disa drejtime:

1. Të konsiderohet kombinimi i mënyrës së propozuar që bazohet në kërkim, me algoritma që bazohen tek testimi dinamik simbolik, pasi kombinimi i këtyre dy teknikave në ditët e sotme konsiderohet mjaft premtues në fushën e testimit strukturor automatik.
2. Hulumtimi i mundësisë së kombinimit të funksionit të propozuar me funksione vlerësimi që bazohen në kritere të tjera mbulimi, duke marrë në konsideratë gjatësinë e rasteve të testimit të gjeneruara.
3. Instrumentimi i kodit për përftimin e numrit të gjendjeve të reja të arritura, mund të kryhet në nivel byte-code, pasi në këtë mënyrë programi nuk ka nevoje të ri-kompilohet dhe ulet koha e llogaritjeve.
4. Eksperimentet për vlerësimin e funksionit të propozuar të kryhen edhe me subjekte të tjera nën testim, në mënyrë që rezultatet e arritura të jenë sa më reale dhe të besueshme.
5. Shqyrtimi i arsyeve (karakteristikave të kodit), të cilat vështirësojnë gjenerimin automatik me teknikat TSBK.
6. Paralelizimi i algoritmit gjenetik që të mund të përftohen rezultatet më shpejt.
7. Të shqyrtohet mundësia e përdorimit të funksionit të propozuar edhe në algoritma të tjerë metaheuristikë.

REFERENCAT

[1] P. Tonella, “Evolutionary Testing of Classes”. In Proceedings of International Symposium on Software Testing and Analysis (ISSTA) 2004

[2] Webb Miller and David L. Spooner. Automatic generation of oating-point test data. IEEE Transactions on Software Engineering, Shtator 1976.

[3] G. Fraser, P. McMinn, A. Arcuri, M. Staats, “Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study”. ACM Transactions on Software Engineering and Methodology, 2015

[4] British Standards Institute. BS 7925-2 software component testing, 1998.

[5] NIST (National Institute of Standards and Technology): The Economic Impacts of Inadequate Infrastructure for Software Testing, Report 7007.011

[6] Boris Beizer. Software Testing Techniques. Van Nostrand Reinhold, New York, 2nd edition, 1990. ISBN 0-442-20672-0.

[7] A. Bertolino, “Software testing research: Achievements, challenges, dreams”. In Future of Software Engineering, 2007. FOSE'07, faqe 85-103. IEEE, 2007.

[8] U. Faroq, “Evaluating Effectiveness of Software Testing Techniques with Emphasis on Enhancing Software Reliability”, March 2012

[9] H. Do, S. Elbaum, G. Rothermel, “Infrastructure Support for Controlled Experimentation with Software Testing and Regression Testing Techniques”, Empirical Software Engineering, Volume 10, Issue 4, Tetor 2005.

[10] S. Shamshiri, R. Just , J. Miguel Rojas, G. Fraser, P. McMinn, A. Arcuri, “Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges” , IEEE/ACM International Conference on Automated Software Engineering, 201–211, 2015.

[11] P. Ammann and J. Offutt, Introduction to Software Testing. Cambridge University Press, 2008

[12] I. Papadhopulli, N. Frasheri, "A Review of Software Testing Techniques", In Proceedings of RCITD, 2014.

[13] P. C. Jongersen, Software Testing – A Craftsman's Approach. 2002.

[14] R. Gopinath, C. Jensen, and A. Groce, "Code Coverage for Suite Evaluation by Developers". In Proceedings of the International Conference on Software Engineering, 2014.

[15] S.Thummalapenta, S. Sinha, "Automating Test Automation", IBM Research Report, Shtator 2011

[16] S. Tasharofi, M. Pradel, Yu Lin, R. Johnson, "Bita: Coverage-guided, Automatic Testing of Actor Programs", In Proceedings of Automated Software Engineering ASE, 2013

[17] T. Xie, N. Tillmann, J. Halleux, W. Schulte, "Fitness guided path exploration in dynamic symbolic execution". IEEE/IFIP International Conference on Dependable Systems & Networks, 2009

[18] C. Pacheco, S. Lahiri, M. Ernst, "Feedback-directed Random Test Generation". In Proceedings of International Conference in Software Engineering (ICSE) 2007

[19] I. Papadhopulli, N. Frasheri, "Today's Challenges of Symbolic Execution and Search-Based for Automated Structural Testing", In Proceedings of ICTIC, 2015.

[20] C. Cadar, P. Godefroid, "SE for software testing in practice–preliminary assessment". In Proceedings of ICSE, 2011.

[21] F. Gross, G. Fraser, A. Zeller, "Search-based system testing: high coverage, no false alarms". In Proceedings of The International Symposium on Software Testing and Analysis (ISSTA), 2012.

[22] K. Inkumsah, T Xie, "Improving structural testing of object oriented programs via integrating evolutionary testing and symbolic execution". In Proceedings of ASE, 2008

[23] K. Lakhotia, P. McMinnb, M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN". Journal of Systems and Software, 2010

[24] K. Lakhotia, N.Tillmann, M. Harman, J. Halleux, "FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution". In Proceedings of ICTSS, 2010.

[25] C. Cadar, P. Godefroid, "SE for software testing in practice–preliminary assessment". In Proceedings of ICSE, 2011.

[26] C. Cadar, K. Sen, "Symbolic Execution for Software Testing: Three Decades Later". Communications of ACM, faqe 82-90, 2013

[27] K. Inkumsah, T Xie, “Improving structural testing of object oriented programs via integrating evolutionary testing and symbolic execution”. In Proceedings of ASE, 2008

[28] J. Burnim, K. Sen, “Heuristics for Scalable Dynamic Test Generation”. In Proceedings of ASE, 2008

[29] R. Majumdar, K. Sen, “Hybrid Concolic Testing”. In Proceedings of ICSE, 2007

[30] P. Marinescu, C. Cadar “make test-zesti: A Symbolic Execution Solution for Improving Regression Testing”. In Proceedings of ICSE, 2012

[31] E. Bounimova, P. Godefroid, and D. Molnar, “Billions and Billions of Constraints: Whitebox Fuzz Testing in Production”. In Proceedings of ICSE, 2013.

[32] S. Makhdoom, M. Khan, “Incremental symbolic execution for automated test suite maintenance”. In Proceedings of ASE, 2014.

[33] S. Person, G. Yang, N.Rungta, S. Khurshid, “Directed Incremental Symbolic Execution”. In Proceedings of PLDI, 2011.

[34] M. Souza, M. Borges, M. Amorim, “CORAL: solving complex constraints for symbolic pathfinder”. In Proceedings of NFM, 2011.

[35] P. McMinn, “Search-based Software Test Data Generation: A Survey”, Software Testing, Verification and Reliability, faqe 105-156, June 2004.

[36] S. Kirkpatrick, C. D. Gellat, M. P. Vecchi, “Optimization by simulated annealing”, Science, faqe 671-680, 1983.

[37] E. Díaz, J. Tuya, R. Blanco, “Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search”, 18th IEEE International Conference on Automated Software Engineering, faqe 310 – 313, Tetor 2003.

[38] Vittorio Maniezzo, Luca Maria Gambardella, and Fabio De Luigi. Ant colony optimization. In Optimization Techniques in Engineering. Springer-Verlag, faqe 101-117. Addison-Wesley, 2004.

[39] J. H. Holland, “Adaptation in Natural and Artificial Systems”, University of Michigan Press, Ann Arbor, 1975.

[40] D. Goldberg “Genetic Algorithms in Search, Optimization and Machine Learning”, Addison-Wesley, 1989

[41] J. Antonisse, “A new interpretation of schema notation that overturns the binary encoding constraint”, In Proceedings of the 3rd International Conference on Genetic Algorithms and Their Applications, faqe 86-91, Morgan Kaufmann, California, USA, 1989.

[42] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), faqe 1329-1336, New York, USA, 2002.

[43] Marc Roper. Computer-aided software testing using genetic algorithms. In Proceedings of the 10th International Software Quality Week (QW '97), San Francisco, USA, 1997.

[44] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. Softw. Test, Verif. Reliab, 9(4): faqe 263-282, 1999.

[45] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. Software Practice and Experience, 30(1):61{79, 2000.

[46] Phil McMinn. Evolutionary Search For Test Data In The Presence Of State Behaviour. PhD thesis, University of Shefield, 2005.

[47] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, faqe 1233 - 1240, New York, 9-13 Korrik 2002.

[48] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, “A systematic review of the application and empirical investigation of searchbased test-case generation,” IEEE Transactions on Software Engineering (TSE), vol. 36, no. 6, faqe 742–762, 2010.

[49] D. Whitley, “The GENITOR algorithm and selection pressure: Why rankbased allocation of reproductive trials is best”, Proceedings of the 3rd International Conference on Genetic Algorithms, faqe 116-121, USA, 1989.

[50] K. Deb, D. Goldberg, “A comparative analysis of selection schemes used in genetic algorithms”, Foundations of Genetic Algorithms, faqe 69-93. Morgan Kaufmann, USA, 1991.

[51] B. Baudry, V. Le Hanh, J.-M. J´ez´equel, Y. Le Traon, “Building trust into OO components using a genetic analogy”, 11th International Symposium on Software Reliability (ISSRE), faqe 4–14, USA, October 2000.

[52] F. Gross, G. Fraser, A. Zeller, “Search-based system testing: high coverage, no false alarms”. In Proceedings of The International Symposium on Software Testing and Analysis (ISSTA), 2012.

[53] J. Rofler, G. Fraser, A. Zeller, A. Orso, “Isolating failure causes through test case generation”. In Proceedings of The International Symposium on Software Testing and Analysis (ISSTA), 2012.

[54] G. Fraser, A. Arcuri, “It is Not the Length that Matters, It is How You Control It”. In Proceedings of ICST, 2011.

[55] G. Fraser, A. Arcuri, “Handling test length bloat”. In Proceedings of ICST, 2013.

[56] A. E. Eiben, S. K. Smit, “Parameter tuning for configuring and analyzing evolutionary algorithms”. Journal: Swarm and Evolutionary Cmputation, pages 19-31, 2011.

[57] A. Aleti, I. Moser, “Entropy-based adaptive range parameter control for evolutionary algorithms”. In Proceedings of GECCO, 2013.

[58] A. Aleti, L. Grunske, “Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm”. Journal of Systems and Software, 2014.

[59] A. Arcuri, G. Fraser, “On parameter tuning in search based software engineering”. Lecture notes in Computer Science, faqe 33-47, 2011.

[60] K. Lakhotia, M. Harman, H. Gross, “AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems”. International Symposium on SBSE, 2010

[61] G. Fraser, A. Arcuri, “Evolutionary Generation of Whole Test Suites”. In Proceedings of QSIC, 2011.

[62] G. Fraser, A. Arcuri, P. McMinn“A Memetic Algorithm for whole test suite generation”. Journal of Systems and Software, 2014.

[63] M. Harman, L. Hu, R. Hierons, J. Wegener, “Testability Transformation”, IEEE Transactions on Software Engineering, 2004.

[64] S. Wappler, A. Baresel, J. Wegener, “Improving Evolutionary Testing in the Presence of Function-Assigned Flags”. In Proceedings of TAICPARTT-MUTATION, 2007.

[65] Y. Li, G. Fraser, “Bytecode Testability Transformation”. In Proceedings of SSBSE, 2011.

[66] S. Wappler, J. Wegener, A. Baresel, “Evolutionary testing of software with function assigned flags”. Journal of Systems and Software, 2009.

[67] A. Arcuri, G. Fraser, J. P. Galeotti, “Automated Unit Test Generation for Classes with Environment Dependencies”. In Proceedings of ASE, 2014.

[68] G. Fraser, A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software”. In Proceedings of ESEC/FSE, 2011.

[69] www.sourceforge.org/Junit/

[70] V. Massol, “JUnit in Action”, Manning Publications, 2004

[71] Z. H. Hall, “J.H.R.: Software unit test coverage and adequacy”. ACM Computing. Maj 1997

[72] http://www.eclemma.org/

[73] K. Hayrust, D. Veerhusen, J. Chilenski, L. Rierson, “A Practical Tutorial on Modified Condition/Decision Coverage”, NASA, Maj 2001

[74] N. Alshahwan, M. Harman, “Coverage and fault detection of the output-uniqueness test selection criteria”. In Proceedings of The International Symposium on Software Testing and Analysis (ISSTA). faqe 181-192. ACM 2014

[75] G. Fraser, A. Arcuri, “Achieving Scalable Mutation-based Generation of Whole Test Suites”. Empirical Software Engineering 2014.

[76] G. Fraser, A. Arcuri, “EvoSuite at the SBST 2015 Tool Competition”. In Proceedings of International Conference in Software Engineering (ICSE) 2015

[77] G. Fraser, A. Arcuri, “Whole Test Suites Generation”. In Proceedings of QSIC, 2012

[78] J. Miguel Rojas, J. Campos, M. Vivanti, G. Fraser, A. Arcuri, “Combining Multiple Coverage Criteria in Search-Based Unit Test Generation” in Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 436-439, 2011

[79] www.souceforge.org

[80] www.codecreator.org

[81] P. Wegner, E. Shriver, “Dimensions of Object-Based Language Design”, MIT press, 1987

[82] S. Ali, L. Briand, H. Hemmati, R. Panesar-Walawege, “A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation”, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 36, 2010

[83] P. Tonella, “Search-Based Test Case Generation”, TAROT Testing School Presenetation, 2013

[84] T.S. Chow. Testing Software Design Modelled by Finite State Machine. IEEE Transactions on Software Engineering, 4(3): faqe 178-187, Maj 1978.

[85] M. Miraz, "Evolutionary Testing of Stateful Systems: a Holistic Approach", Tezë Disertacioni, Universiteti Politeknik i Torinos, 2010

[86] K. Ghani, "Searching for Test Data" Tezë Disertacioni, Universiteti York, 2009

[87] Christoph Csallner, Yannis Smaragdakis. "Jcrasher: an automatic robustness tester for java". Softw. Pract. Exper.,34(11):faqe 1025-1050, 2004.

[88] Christoph Csallner, Yannis Smaragdakis, Tao Xie. "DSD-Crasher: A hybrid analysis tool for bug finding". ACM Transactions on Software Engineering and Methodology (TOSEM), 17(2):faqe 1-37, 2008.

[89] V. Dallmeier, C. Lindig, A. Vasilowski, "Mining Object Behaviour with ADABU". In Proceedings of the International Workshop on Dynamic Systems Analysis, 2006

[90] A. Arcuri, "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing". Third International Conference on Software Testing, Verification and Validation, 2010"

[91] Dave Binkley, Mark Harman, and Kiran Lakhotia. FlagRemover: A Testability Transformation For Loop Assigned Flags. Transactions on Software Engineering and Methodology,

[92] Li, N., Meng, X., O_utt, J., Deng, L.: Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In: Proc. of ISSRE'13. faqe 380-389. IEEE (2013)

[93] M. Tatsubori1, S. Chiba2, M. Killijian, K. Itano, "OpenJava: A Class-based Macro System for Java", Reflection and Software Engineering, Springer, Vellimi 1826, 2001

[94] A. Rountev, "Precise identification of side-effect-free method in java", 20th IEEE International Conference on Software Maintenance (ICSM '04), faqe 82–91, 2004.

[95] A. Salcianu, M. Rinard, "Purity and side effect analysis for java programs", In Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, faqe 199–215, Janar 2005.

[96] K. Jansen, "Instrumentation and transformation of Java source code for automated testing with search-based testing algorithms", Teze Masteri, Universiteti i Oslos, Gusht 2010

[97] http://metrics.sourceforge.net/

[98] http://muclipse.sourceforge.net/

[99] Y.S. Ma, J.Offutt, “Description of Class Mutation Mutation Operators for Java”, Gusht 2014

[100] A. Siami Namin, J. H. Andrews, D. J. Murdoch, “Sufficient mutation operators for measuring test effeectiveness”, In Proceedings of the International Conference on Software Engineering (ICSE), faqe 351-360, 2008.

[101] M. Staats, G. Gay, M. Heimdal, “Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing”, In Proceedings of International Conference on Software Engineering ICSE, faqe 870–880, 2012

[102] S. Mirshokraie, A. M. K. Pattabiraman, “PYTHIA: Generating Test Cases with Oracles for JavaScript Applications”, In Proceedings of Automated Software Enegineering ASE, 2013

[103] D. Le, M. A. Alipour, R. Gopinath, A. Groce “MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs”, In Proceedings of The International Symposium on Software Testing and Analysis (ISSTA), Qershor 2014

[104] J., D., Gupta, “Improving Fault Detection Capability by Selectively Re-taining Test Cases During Test Suite Reduction”. IEEE Transactions in Software Engeneering, faqe 108-123, 2007