SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1033

Primjena modela evolucijskog učenja na probleme optimizacije

Edi Smoljan

Zagreb, lipanj 2015.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 26. veljače 2015.

Predmet: **Analiza i projektiranje računalom**

DIPLOMSKI ZADATAK br. 1033

Pristupnik: **Edi Smoljan (0036459026)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Primjena modela evolucijskog učenja na probleme optimizacije**

Opis zadatka:

Opisati Learnable Evolution Model (LEM) i moguće načine primjene. Istražiti mogućnosti primjene LEM-a na probleme koji kao prikaz imaju permutaciju, vektor bitova ili stablo. Izraditi programsko okruženje za ispitivanje različitih inačica LEM-a koje rade s različitim prikazima rješenja. Ispitati inačice LEM-a nad skupom standardnih ispitnih problema kontinuirane i kombinatoričke optimizacije i usporediti rezultate s onima dobivenim postojećim evolucijskim algoritmima. Ocijeniti prostornu i vremensku složenost svake inačice. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 13. ožujka 2015.
Rok za predaju rada: 30. lipnja 2015.

Mentor:

Izv. prof. dr. sc. Domagoj Jakobović

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Siniša Srbljić

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

Zahvaljujem svojim roditeljima koji su mi pružali veliku podršku u dosadašnjem dijelu života.

Također, zahvaljujem mentoru prof. dr. sc. Domagoju Jakobovića na svoj pruženoj pomoći i usmjeravanju tijekom studija.

Sadržaj

1. Uvod

Za danas široko rasprostranjene probleme optimizacije, kojima je rješenje gotovo nemoguće pronaći metodama matematičke numeričke optimizacije poput gradijentnog spusta, pribjegava se stohastičkim metodama ili heuristikama. One često daju bolje rješenje od klasičnih metoda ali svejedno ne daju garanciju da je pronađeno rješenje zapravo najbolje moguće. Najpopularniji algoritam takvog tipa je genetski algoritam koji je zapravo predložak (metaheuristika) koji za ispravan rad zahtijeva implementaciju zasebnih dijelova (heuristika) specifičnih za problem koji se algoritmom nastoji optimizirati. Funkcionalnost tih heuristika unutar algoritma je inspirirana Darwinovom teorijom evolucije: na skup rješenja se gleda kao na populaciju jedinki koje heuristikom križanja stvaraju nove jedinke (rješenja), postojeće jedinke se mogu transformirati heuristikom mutacije a brojnost populacije regulira heuristika prirodne selekcije koja u pravilu pruža trenutno boljim rješenjima veću vjerojatnost opstanka.

Jedna od slabosti darvinovskih postupaka stvaranja i modificiranja jedinki (točnije križanja i mutacije) je ta da se ti postupci u pravilu izvode potpuno nasumično i slijepo, pritom ne uzimajući u obzir dostupne moguće korisne informacije o obliku trenutno dobrih rješenja iz populacije. Zbog toga novorazvijeni postupci stohastičke optimizacije pokušavaju na neki način iskoristiti postojeće podatke u rješenjima pri stvaranju novih, potencijalno dobrih, rješenja. Jedan od tih postupaka je *Learnable Evolution Model* (u daljnjem tekstu LEM): metaheuristika koja dijeli mnogo dodirnih točaka s genetskim algoritmom, ali u isto vrijeme omogućava definiranje dijela za instanciranje novih rješenja na temelju trenutnih ili povijesnih podataka što se postiže korištenjem algoritama strojnog učenja i dubinske analize podataka.

Cilj ovog rada je opisati algoritam LEM i ispitati njegov rad u sprezi s različitim postupcima dubinske analize podataka (engl. *data mining*) nad različitim domenama primjene. Kako rješavanje problema u različitim domenama zahtijeva različit prikaz rješenja potrebno je algoritme dubinske analize podataka i načine stvaranja rješenja (instanciranja) prilagoditi problemu. Ovaj rad se sastoji od opisa LEM algoritma i opisa implementacije okruženja za ispitivanje ostvarenja rješenja različitih problema. Rad je raspodijeljen po najčešće korištenim prikazima jedinki u

optimizacijskim algoritmima što znači da su postupci instanciranja i dobiveni rezultati opisani za prikaze rješenja poljem realnih brojeva, poljem bitova, permutacijom i stablom.

2. Learnable Evolution Model (LEM)

2.1 Motivacija

LEM je jedan od najpoznatijih algoritama evolucijskog računanja koji koristi i nedarvinovske postupke pri napretku faza algoritma. Darvinovski algoritmi poput genetskog algoritma inspirirani su postavkama iznesenim u Darwinovoj teoriji evolucije: jedinke sadrže i prenose svoj genetski materijal na potomstvo, evolucija nastavlja pomoću rekombinacije (križanja) i transformacije (mutiranja) genetskog materijala te sam proces evolucije napreduje po pravilu opstanka najboljih. Algoritmi nastali po tim postavkama su često jednostavni i lako prilagodljivi širokom rasponu problema bez zahtijevanja specifičnog znanja o domeni problema. No, postoje i određeni problemi s ovakvim tipom evolucijskih algoritama: proces je polu nasumičan jer se njegovi glavni operatori oslanjaju na nasumičnost i generiranje novih jedinki ne uzima u obzir znanje naučeno u prethodnim generacijama pa zbog toga algoritmi darvinovskog evolucijskog računanja znaju biti ne previše učinkoviti.

Usporedno s Darwinovom teorijom evolucije razvijale su se i druge teorije koje su proučavale utjecaj naučenih osobina na proces evolucije: Lamarckovu teoriju [1] koja pretpostavlja da se osobine jedinke naučene za njezinog života mogu prenijeti potomcima putem genoma (što je u potpunosti pobijeno tijekom 20. stoljeća) i utjecaj Baldwinovog efekta [2] na klasičnu teoriju evolucije. Nedarvinovski algoritmi evolucijskog računanja nastoje modelirati proces "intelektualne evolucije" u kojemu se analiziraju prednosti i nedostaci prethodnih rješenja te se rezultati analiza koriste u izgradnji novih rješenja. To rezultira izuzetno ubrzanim napredovanjem koje se ne ravna po zakonima biološke evolucije.

LEM nastoji implementirati "intelektualnu evoluciju" tako da stvaranje novih jedinki za iduću generaciju procesa evolucije prepušta i nekom inteligentnom agentu koji stvara nove jedinke uzimajući u obzir podatke iz prethodnih generacija koristeći neke metode strojnog učenja, a ne nasumično rekombinacijom gena i mutacijom.

2.2 Opis algoritma

U ovom radu je opisana i implementirana LEM3 verzija LEM-a [3] iz 2005. godine. U nastavku je napisan pseudokod algoritma, a funkcionalnost pojedinih dijelova biti će pojašnjena nakon toga.

```
1  Stvori početnu populaciju rješenja
2  Dok god uvjet zaustavljanja nije zadovoljen
3     Evaluiraj jedinke;
4     Odaberi populaciju roditelja;
5     Izvrši jednu ili više akcija od:
6          Nauči hipoteze koje razlikuju dobre od loših jedinki i
                instanciraj ih (Učenje)
7          Stvori nove jedinke koristeći Darwinove operacije
                (Pretraživanje)
8          Promijeni prikaz jedinki
9          Randomiziraj populaciju (mutiranjem dijela populacije
ili             ponovnim pokretanjem procesa evolucije)
10 Izračunaj statistiku i prikaži rezultate
11 Završi LEM
```

LEM3 pseudokod

LEM algoritam je generacijski evolucijski algoritam i sama srž mu je ista srži genetskog generacijskog algoritma. U liniji 1 LEM može stvoriti početnu populaciju rješenja uzevši u obzir određene parametre ili početnu populaciju jednostavno prepustiti korisniku na stvaranje i samo je preuzeti. Linija 2 je standardna u bilo kojem iterativnom optimizacijskom algoritmu, s tim da je za uvjet zaustavljanja često naveden maksimalni broj evaluacija funkcije cilja, maksimalni broj iteracija, minimalna dobrota rješenja, broj iteracija bez napretka i slično što varira od implementacije do implementacije.

U liniji 4 odabire se subpopulacija rješenja koja će se koristiti u postupcima učenja novih rješenja: to može biti čitava trenutna populacija rješenja ili pak neki njezin dio odabran postupcima selekcije koji se koriste i u genetskom algoritmu poput proporcionalne ili turnirske.

Od linije 5 do 9 navedene su moguće akcije koje LEM može izvršiti u jednoj iteraciji, a može ih odabrati više odjednom.

U liniji 6 navedena je akcija učenja: zasebnom modulu se predaju dobre i loše jedinke, modul pokušava naučiti što dobre jedinke čini dobrima i što loše čini lošima pa po naučenim pravilima ili opisima instancira zadani broj novih jedinki. Način na koji se određuju dobre i loše jedinke ovisi o implementaciji LEM-a: moguće je za dobre jedinke uzeti određeni postotak dobrih, a za loše određeni postotak loših jedinki iz populacije roditelja sortiranih po dobroti rješenja ili za dobra rješenja uzeti ona koja imaju dobrotu bolju od neke dinamično određene granice, a za loša uzeti ona koja imaju dobrotu lošiju od neke druge dinamično određene granice.

Broj rješenja koja se stvaraju učenjem i pretraživanjem može biti konstantan ili dinamičan tijekom procesa optimizacije. Kako bi se ostvarila kontrola nad tim brojem, u radu [3] predložena je implementacija zasebnog modula nazvanog *Action profiling function* koji na vanjsko definirani način kontrolira broj jedinki koje instanciraju moduli učenja, pretraživanja ili koje se instanciraju nasumično. Ti brojevi se mogu mijenjati kroz vrijeme, a prilikom modifikacije moguće je uzeti u obzir neke statističke pokazatelje jedinki generiranih od strane zasebnih modula poput prosječne dobrote novostvorenih jedinki.

U liniji 7 za stvaranje određenog broja novih jedinki iz populacije roditelja primjenjuju se Darwinovi operatori križanja i mutacije te je u tom slučaju postupak identičan genetskom algoritmu. Linija 8 ukazuje na promjenu prikaza jedinki. U originalnom radu se je ovaj algoritam ispitivao nad problemom optimizacije kontinuiranih funkcija koristeći vektore bitova za prikaz rješenja. Kako je u određenim trenutcima napretka optimizacije potrebno prostor rješenja pretraživati sve finijom rezolucijom tako se i prikaz (struktura) jedinki mijenja dodavanjem većeg broja bitova. Ovaj se postupak u ovom radu ne koristi.

U liniji 9 se nakon svake iteracije provjerava stanje algoritma odnosno stagnira li postupak optimizacije i ovisno o stanju algoritma je li potrebno napraviti neki drastični korak poput mutacije dijela populacije, ponovnog pokretanja algoritma ili potpunog zaustavljanja optimizacije. Pseudokod odlučivanja je napisan u nastavku.

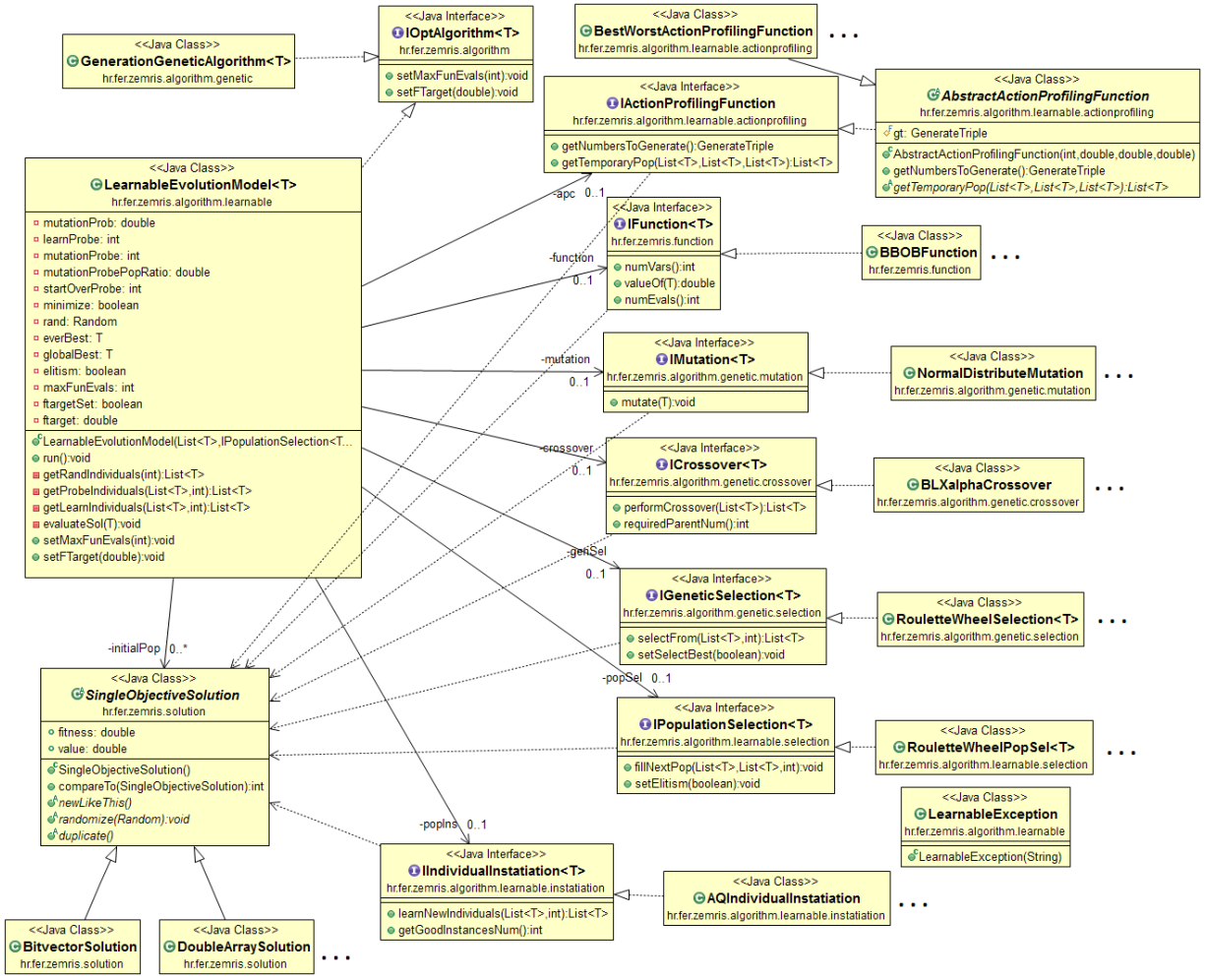
```
1  Ako nema napretka
2    Uvećaj learn-probe-counter
3    Ako je learn-probe-counter >= learn-probe
4          learn-probe-counter = 0
5          Ako je mutation-probe-counter < mutation-probe
6                Uvećaj mutation-probe-counter
7                Mutiraj jedinke
8                Evaluiraj mutirane jedinke
9          Inače ako je start-over-probe-counter < start-over-
probe
10               Uvećaj start-over-probe-counter
11               mutation-probe-counter = 0
12               Zapamti najbolje jedinke
13               Pokreni ponovo, ponovo inicijaliziraj populaciju
14               Evaluiraj sve jedinke
15         Inače
16               Zaustavi LEM3
```

Pseudokod odluke na kraju svake iteracije

Iz pseudokoda je vidljivo da se u algoritam LEM uvode dodatni parametri kojima se specificiraju različiti oblici stagnacije postupka i odlučuje o akciji koja se poduzima kad je neki oblik stagnacije detektiran. Parametar *learn-probe* određuje broj iteracija bez napretka nakon kojeg će se izvršiti prva akcija: mutiranje dijela populacije. Izvršavanje te akcije opet može biti podložno određenim korisničkim definiranim pravilima i postupcima: npr. korisnik može definirati proceduru koja odabire dio populacije za mutaciju, koliko jedinki je potrebno mutirati itd. Parametar *mutation-probe* služi za određivanje maksimalnog broja izvršenih mutacija dijela populacije kako je opisano ranije u jednom pokretanju algoritma. Kad je taj broj dosegnut čitava populacija rješenja se reinicijalizira i optimizacija kreće ispočetka. Jednom kad se populacija reinicijalizira *start-probe* puta i opet je detektirana stagnacija LEM završava s radom. Osim ovog načina LEM se može zaustaviti i nekim uobičajenijim načinom poput dosezanja broja maksimalnih evaluacija funkcije cilja, približavanju željenoj dobroti i slično.

2.3 Pregled programskog ostvarenja

U nastavku je prikazan dijagram razreda okruženja za ispitivanje različitih scenarija uporabe LEM algoritma (Slika 2.1). Okruženje je napravljeno po predlošku okruženja za evolucijsko računanje iskazanom u [4] i po uzoru na Evolutionary Computation Framework [5].



Slika 2.1 Dijagram razreda razvijenog okruženja

Svaki algoritam optimizacije ostvaren u okviru ovog okruženja mora implementirati sučelje *IOptAlgorithm<T>* koje propisuje metode za postavljanje maksimalnog dozvoljenog broja evaluacija funkcije cilja, metodu za postavljanje željene dobrote i metodu *run* koja pokreće postupak optimizacije. *GenerationGeneticAlgorithm* je jedan od algoritama koji implementiraju ovo sučelje, a jedan od njih je i također

LearnableEvolutionModel. Svaki algoritam radi nad određenim prikazom rješenja koje je ostvareno konkretnom implementacijom apstraktnog razreda *SingleObjectiveSolution* koji modelira jedinku u algoritmima jednokriterijske optimizacije. Konkretni razredi koji nasljeđuju taj apstraktni razred trebaju implementirati metode za stvaranje novih jedinki sličnih njima samima, dupliciranje jedinki (kloniranje) i nasumičnu inicijalizaciju jedinke. Neki konkretni razredi koji nasljeđuju *SingleObjectiveSolution* su *BitVectorSolution, PermutationSolution, DoubleArraySolution, TreeGASolution.*

Sama komponenta *LearnableEvolutionModel* je napravljena po obrascu strategije što znači da neke svoje dijelove delegira zasebnim objektima koji implementiraju određena sučelja. Slijedi popis tih sučelja:

IFunction: funkcija koju se optimizira mora implementirati ovo sučelje. Sadrži metode za evaluaciju jedinke, vraćanje broja evaluacija funkcije i dimenziju rješenja (gdje to ima smisla). Konkretni primjeri bili bi funkcija koja evaluira duljinu puta permutacije u TSP problemu (*TSPSolutionEvaluator*), Rosenbrock (*RosenbrockFunction*) funkcija u kontinuiranoj domeni itd.

IMutation: predstavlja mutaciju koja se koristi pri stvaranju jedinki pretraživanjem i koja se koristi pri mutiranju dijela populacije u slučaju da je detektirana stagnacija. Konkretne implementacije tog sučelja primaju jedinku i na neki način ju modificiraju metodom *mutate*. Konkretni primjeri su *NormalDistributeMutation* koja mutira vektor realnih brojeva dodajući broj koji je izvučen iz normalne distribucije jednoj njegovoj dimenziji, *ReplaceSubtreeMutation* koja u genotipu stabla mijenja neko podstablo nanovo stvorenim podstablom.

ICrossover: predstavlja operator križanja koji se koristi pri stvaranju jedinki pretraživanjem. Zahtijeva da konkretni razredi implementiraju metode *requiredParentNum* (vraća broj roditelja potrebnih za križanje) i *performCrossover* (vraća djecu nastalu križanjem). Konkretni primjeri su *BLXalphaCrossover* koji potomke stvara nasumičnim uzorkovanjem hiperkvadra omeđenog roditeljima proširenog za parametar *alfa*, razred *BitVectorOneBreakpointCrossover* koji implementira križanje dva binarna vektora s jednom točkom prekida itd.

IGeneticSelection: postupci selekcije jedinki za križanje i mutaciju u darvinovskom načinu stvaranja jedinki moraju implementirati ovo sučelje. Sučelje

propisuje implementaciju metoda za selekciju zadanog broja jedinki iz liste danih jedinki (*selectFrom*) i postavljanje načina rada selekcije boljih ili lošijih jedinki (*selectBest*). Konkretne implementacije ovog sučelja su razredi poput *KTournamentSelection* (implementira k-turnirsku selekciju) i *RouletteWheelSelection* (vjerojatnosna selekcija jedinki proporcionalna njihovoj dobroti). Ti razredi očekuju da ulazna lista jedinki bude sortirana uzlazno po dobroti.

IPopulationSelection: razredi koji implementiraju ovo sučelje koriste se za selekciju jedinki iz unije trenutne populacije i novostvorenih jedinki koje će ući u iduću generaciju. Sučelje propisuje implementaciju metode za punjenje spremnika sljedeće populacije potrebnim brojem jedinki iz unije (*fillNextPop)* i uključivanje odnosno isključivanje utjecaja elitizma. Konkretni razredi koji implementiraju ovo sučelje su *KTournamentPopSel* (k-turnirska selekcija) i *RouletteWheelPopSel* (proporcionalna selekcija).

IIndividualInstatiation: svaki modul koji instancira nove jedinke postupkom učenja mora implementirati ovo sučelje koje propisuje implementaciju stvaranja određenog broja novih jedinki postupkom učenja (metode *learnNewIndividuals*) i metode koja govori koji broj jedinki iz vraćene liste jedinki je stvoren direktno koristeći dobra pravila dobivena postupkom učenja jer je moguće da učenje ne rezultira dobrim pravilima (*getGoodInstancesNum*). O konkretnim implementacijama ovog sučelja biti će riječi kasnije kroz rad.

IActionProfilingFunction: sučelje propisuje implementaciju metoda potrebnih za ostvarenje konkretne *Action Profiling* funkcije. Iz metode *getNumbersToGenerate* potrebno je tri broja koja određuju koliko će LEM jedinki generirati postupcima učenja, pretraživanja i nasumično. Implementacija početne inicijalizacija razreda i metode *getNumbersToGenerate* je zajednička svim konkretnim razredima pa je izdvojena u apstraktni razred *AbstractActionProfilingFunction* koji nema implementiranu metodu *getTemporaryPop*. Metodom *getTemporaryPop* spajaju se dobivene novostvorene jedinke u jednu trenutačnu populaciju što je dobro mjesto za obavljanje analize dobrote jedinki stvorenih različitim metodama i modificiranje brojeva koji se vraćaju metodom *getNumbersToGenerate* u skladu s analizom. Konkretne implementacije tog apstraktnog razreda su *BestWorstActionProfilingFunction, StagnationAvoidActionProfilingFunction*

(mijenjaju brojeve kroz iteracije kako je opisano u 2.4) i *ConstantActionProfilingFunction* (broj jedinki koji se generira pojedinim postupkom ostaje konstantan kroz sve generacije).

2.4 Parametri LEM-a

Komponenta u kojoj je implementiran LEM algoritam je *LearnableEvolutionModel* i, osim komponenti navedenih prije, pri konstrukciji prima još nekoliko parametara čije su uobičajene vrijednosti za testove navedene uz njihove opise:

***mutationProb*:** vjerojatnost mutacije nove jedinke u stvaranju jedinki postupkom pretraživanja. U svim testovima u pravilu postavljen na 0.3.

learnProbe, mutationProbe, startOverProbe: objašnjeno u poglavlju **Error! Reference source not found.**. *learnProbe* u većini testova postavljen na 1000, *mutationProbe* na 10, *startOverProbe* na 3.

mutationProbePopRatio: postotak populacije koji se mutira kad je detektirana stagnacija. U svim testovima postavljeno na 0.5.

minimize: traži li se minimum ili maksimum funkcije. Ovisno o problemu.

elitism: označuje garantira li se prijenos najbolje jedinke u trenutnoj generaciji u sljedeću. U svim testovima je elitizam uključen.

maxFunEvals: maksimalni broj evaluacija funkcije cilja. Testovi se u pravilu izvršavaju s $500\ 000$ evaluacija funkcije cilja. Parametar *maxFunEvalsSet* označava je li ovaj parametar postavljen.

Populacija u je u većini testova fiksne veličine i broji 100 jedinki. Dodatno, u svim testovima se koristi jedna od prije navedenih dinamičkih *ActionProfiling* funkcija kojima je početni omjer generiranja jedinki *učenje:pretraživanje:nasumično* = $1:2:0$.

BestWorstActionProfilingFunction pri stvaranju uz uobičajene parametre veličine populacije (*popSize*) i početnog omjera jedinki za generiranje (*learnRatio, probeRatio* i *randRatio*) prima i parametre koji određuju minimalni udio jedinki generiranih postupkom učenja u novoj populaciji (*learnMin*), minimalni udio jedinki generiranih postupkom pretraživanja u novoj populaciji (*probeMin*) i broj generacija nakon kojeg se donosi odluka o smanjenju ili povećanju pojedinog udjela u

sljedećoj populaciji (*interval*). Razred za *interval* generacija određuje najbolje rješenje koje su postupci učenja i pretraživanja stvorili te medijan rješenja koja za svaki od ta dva postupka. Tada se odlučuje o promjeni udjela pojedinog postupka stvaranja jedinki na sljedeći način: ako je najbolje rješenje pronađeno postupkom pretraživanja lošije od medijana rješenja pronađenih postupkom učenja povećava se udio rješenja stvorenih učenjem za jednu jedinku i smanjuje se udio rješenja stvorenih pretraživanjem za isti broj. Slično, ako je najbolje rješenje pronađeno postupkom učenja lošije od medijana rješenja pronađenih postupkom pretraživanja povećava se udio rješenja stvorenih pretraživanjem, a smanjuje se broj rješenja nastalih učenjem. U slučaju da je najbolje rješenje pronađeno učenjem bolje od medijana rješenja pronađenog pretraživanjem i najbolje rješenje pronađeno pretraživanjem bolje od medijana rješenja pronađenog učenjem udjeli se mijenjaju težeći početno postavljenim udjelima.

StagnationAvoidActionProfilingFunction je dinamička *ActionProfiling* funkcija koja mijenja udjele načina stvaranja jedinki tako da se nastoji maknuti iz stanja stagnacije, odnosno nepronalaska boljeg rješenja. Pri inicijalizaciji uz uobičajene parametre veličine populacije i početnog omjera jedinki za generiranje dobiva i parametre koji određuju broj generacija nakon kojeg se donosi odluka o smanjenju ili povećanju pojedinog udjela u sljedećoj populaciji (*interval*), broj jedinki koji se dodaje ili oduzima nekom načinu pri donošenju odluke (*increment*), relativno povećanje dobrote pronađenog najboljeg rješenja s obzirom na prethodno najbolje rješenje koje je granica detekcije stagnacije (*stagnationThreshold*), relativno povećanje dobrote pronađenog najboljeg rješenja s obzirom na prethodno najbolje rješenje koje ukazuje na to da postupak dobro konvergira (*okThreshold*) i oznaku provodi li se minimizacija (*minimize*). Postupak promjene udjela je sljedeći: određuje se dobrota najboljeg generiranog rješenja u prethodnih *interval* generacija i uspoređuje se s najboljom dobrotom iz prethodnih *interval* generacija. Relativno povećanje dobrote rješenja se određuje tako da se razlika novog najboljeg i starog najboljeg podijeli s vrijednošću starog najboljeg. Ako je pronađeno povećanje manje od *stagnationTreshold* udjeli se mijenjaju za *increment* (učenje povećava, pretraživanje smanjuje ili obrnuto) prema načinu koji je manje zastupljen u udjelu stvaranja novih rješenja. Na primjer: ako je stagnacija detektirana u stanju gdje se većina jedinki generira postupkom pretraživanja onda

će se povećati udio jedinki koje se generiraju postupkom učenja i smanjiti udio koji se stvara postupkom pretraživanja. Udjeli se mijenjaju svakih *interval* generacija, čak i ako je postupak u stanju gdje optimizacija napreduje zadovoljavajuće s tendencijom povratka u stanje napretka optimizacije ako se promjenom izazove stagnacija postupka.

Svaki modul koji stvara jedinke učenjem treba na neki način znati koji dio predanih jedinki nad kojim se izvršava učenje je dobar, a koji loš. Zbog toga modul za učenje treba dobiti listu rješenja sortiranih uzlazno po dobroti i parametre *HPT* i *LPT* (*high i low population threshold*) koji označavaju koji postotak najboljih (*HPT*) ili najgorih (*LPT*) rješenja treba označiti kao pozitivne primjere za učenje dobrih ili loših pravila. I HPT, i LPT su u ovom radu kroz sve testove postavljeni na 0.3.

Ako u nekom testnom slučaju parametri opisani u ovom poglavlju poprimaju drukčije vrijednosti od ovih navedenih ta promjena je napisana uz opis testa.

3. Primjena na prikaz rješenja poljem realnih brojeva

Prikaz rješenja poljem realnih brojeva je intuitivan način prikaza rješenja pri optimizaciji funkcija definiranih u višedimenzionalnim kontinuiranim domenama. Za prikaz rješenja u okružju za ispitivanje korišten je *DoubleArraySolution* razred, a za operatore mutacije i križanja njemu prilagođeni operatori:

NormalDistributeAllMutation: mutira sve dimenzije rješenja svakoj dimenziji dodajući neki nasumični broj iz normalne distribucije centrirane na 0 sa disperzijom 0.05.

NormalDistributeMutation: mutira samo jednu nasumično odabranu dimenziju rješenja dodajući joj neki nasumični broj iz normalne distribucije centrirane na 0 sa disperzijom 1.0.

BLXalphaCrossover: križanje zahtijeva dva roditelja kojima se onda u višedimenzionalnom prostoru omeđuje hiperkvadar koji se proširuje za parametar *alpha* $\in [0, 1]$. Iz podrostora tako definiranog hiperkvadra uzorkuje se nasumično nova jedinka. Parametar *alpha* je u testovima postavljen na 0.1.

3.1 Učenje pravila C4.5 i AdaBoost algoritmom

Glavna zamisao instanciranja novih jedinki učenjem jest na neki način dokučiti obilježja prostora u kojemu se nalaze dobre jedinke, odnosno pravila iz kojih se dalje stvaraju nove jedinke. Pravilo je opisano minimalnom i maksimalnom vrijednošću za svaku dimenziju, uz mogućnost da te vrijednosti mogu biti neograničene, tj. beskonačne. Tako pronađen skup pravila zapravo predstavlja skup (u nekim dimenzijama moguće neograničenih) hiperkvadara u prostoru iz kojih se uzorkuju nova rješenja.

Učenje je izvedeno pomoću *Weka* okružja za dubinsku analizu podataka otvorenog koda [6]. Prvotno ideja je bila pravila naučiti samim C4.5 algoritmom [7] strojnog učenja čiji je pseudokod opisan u nastavku:

```
1 Provjeri bazne slučajeve:
2    Ako svi podaci u listi pripadaju istom razredu stvori list za
     taj razred
3    Ako je niti jedan atribut ne pruža informacijsku dobit ili
     pojava instance prethodno neviđenog razreda tada stvori čvor
     odluke na višoj razini u stablu koristeći očekivanu
     vrijednost  razreda
4 Za svaki atribut A
5    Izračunaj normaliziranu informacijsku dobit
6 Neka je A_best atribut s najboljom informacijskom dobiti
7 Stvori čvor odluke i podijeli podatke po a_best
8 Rekurzivno stvori stabla odluke nad podskupovima podataka
     dobivenim dijeljenjem po A_best i dodaj ta stabla trenutnom
     čvoru
```

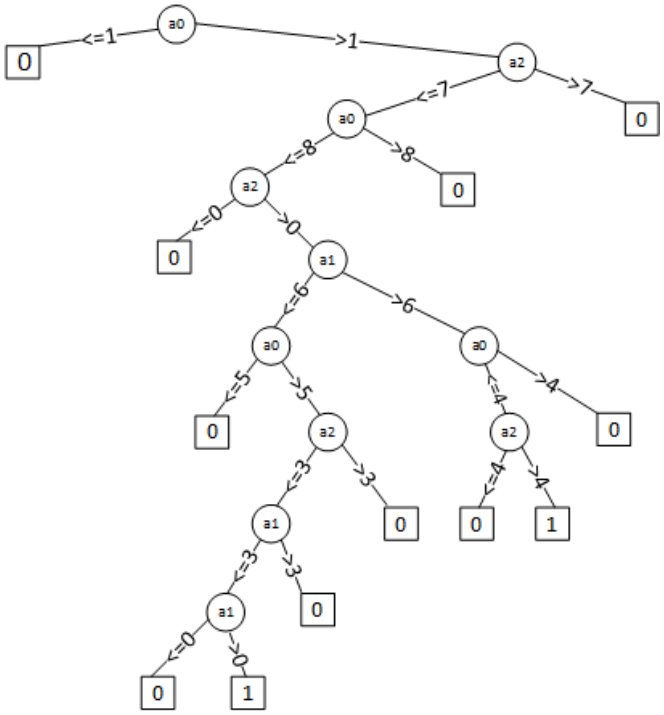
Pseudokod C4.5 algoritma

3.1.1 Opis učenja pravila

Algoritam C4.5 kao ulaz prima skup vektora realnih brojeva označenih s pozitivno ili negativno, a na izlazu vraća stabla odluke poput onog na slici Slika 3.1 iz kojih se mogu iščitati određena pravila. Pravila koja opisuju podatke označene sa '1' završavaju u listovima s oznakom '1', a pravila koje opisuju podatke označene s '0' u listovima s oznakom '0'. Pravila koja se mogu iščitati iz prikazanog stabla za oznaku '1' su:

Pravilo 1: donja granica $[5, 0, 0]$; gornja granica $[8, 3, 3]$

Pravilo 2: donja granica $[1, 6, 4]$; gornja granica $[4, \infty, 7]$



Slika 3.1 Primjer stabla odluke C4.5 algoritma

Kako uporaba samog C4.5 nije polučila zadovoljavajuće rezultate, a C4.5 se smatra slabim klasifikatorom (*engl. weak learner*) koji ima visoku pristranost i nisku varijancu, te je ustanovljeno da je i vremenski relativno nezahtjevan, pristupilo se pojačavanju (*engl. boosting*) koristeći AdaBoost algoritam [8] što je poboljšalo rezultate na manjem skupu ispitnih problema. Pseudokod AdaBoost algoritma je prikazan u nastavku.

```
1 Za dane označene primjere za učenje (x₁,y₁), ..., (xₘ,yₘ),
    yᵢ ∈ {-1, +1} inicijaliziraj D₁(i) = 1/m
2 Za t = 1, ..., T:
3    Treniraj slabog klasifikatora koristeći distribuciju Dₜ
4    Dobij slabu hipotezu hₜ sa ciljem smanjenja ponderirane
           greške εₜ
5    Izračunaj αₜ = 1/2 * ln ((1 - εₜ) / εₜ)
6    Za i = 1, ..., m
           Dₜ₊₁(i) = Dₜ(i) * exp(-αₜ * yᵢ * hₜ(xᵢ)) / Zₜ
           gdje je Zₜ normalizacijski faktor
7 Završna hipoteza H(x) = sign(sum(αₜ *  hₜ(x)))
```

Pseudokod AdaBoost algoritma

AdaBoost algoritam nastoji prenaučiti ulazne primjere zadajući im različite težine prilikom postupka učenja koristeći se zadanim slabim klasifikatorom. U ovom

slučaju ulazni podaci u AdaBoost algoritam su vektori realnih brojeva označeni u skladu s *HPT* parametrom (vidjeti 2.4) i kojima su težine postavljene na 1.0. Izlaz iz algoritma je više C4.5 stabala odluke različitih težina što povlači više generiranih pravila.

Komponenta koja ostvaruje učenje naziva se *BoostedC45IndividualInstatiation* i kao parametre pri inicijalizaciji prima *HPT*, *LPT* i realni broj $\in [0, 1]$ koji određuje broj iteracija AdaBoost algoritma (odnosno broj generiranih hipoteza). Taj broj se množi s brojem podataka koje treba naučiti po intuiciji da je za više podataka potrebno više naučenih hipoteza (iteracija) kako bi se podaci bolje opisali. To je parametar *iterationsPerc* i u testovima je postavljen na 0.05.

Asimptotska vremenska složenost C4.5 algoritma je $O(mn^2)$ gdje je *m* broj primjera za učenje, a *n* njihova dimenzija. Kako je asimptotska složenost AdaBoost algoritma $O(Tf)$ gdje je *T* broj iteracija algoritma (regulirano parametrom *iterationsPerc*) i *f* vremenska složenost učenja slabog klasifikatora slijedi da je ukupna vremenska složenost $O(Tmn^2)$.

3.1.2 Opis stvaranja novih jedinki

Jednom kad AdaBoost generira stabla odluke potrebno je iz njih izvući pravila. Za potrebe jednostavnog generiranja pravila zanemarene su težine kojima se množe pojedini izlazi pojedinih klasifikatora. Za svaki list u stablu odluke koji završava s oznakom željene klase stvara se novo pravilo skupljajući atribute na putu od lista do vrha stabla odluke. Tako stvorena pravila imaju u sebi dosta neograničenih dimenzija koje se naknadno ograničavaju tako da se za svaku dimenziju ako je neograničena s donje strane postavi na minimalnu vrijednost za tu dimenziju u skupu podataka, a za svaku dimenziju koja je neograničena s gornje strane na maksimalnu vrijednost.

Tako generiranim pravilima se dodjeljuje dobrota na temelju omjera pozitivnih (*HPT* dio podataka) i negativnih primjera (ostali dio podataka) koje pravilo pokriva. Prilikom stvaranja novih jedinki uzima se u obzir dobrota pravila i pri svakom stvaranju odabire se jedno pravilo iz skupa naučenih pravila proporcionalno dodijeljenoj dobroti. Kako izabrano pravilo zapravo opisuje potprostor u kojemu bi

se trebala nalaziti dobra rješenja, za stvaranje novog rješenja potrebno je taj potprostor samo uzorkovati.

3.2 Učenje pravila AQ algoritmom

U radu u kojem je predstavljen algoritam LEM ([3]) autori su za optimizaciju kontinuirane funkcije koristili *Algorithm Quasioptimal* (AQ) za učenje pravila u binarnoj domeni za optimizaciju kontinuirane funkcije. Kako se sam algoritam može prilagoditi različitim prikazima rješenja, odlučilo se da će se isprobati prilagoditi prikazu rješenja poljem realnih brojeva. Postupak je implementiran u razredu *AQIndividualInstantiation* i pomoćnim razredima koji implementiraju sam AQ algoritam (*PseudoAQ* i *OptPseudoAQ*).

3.2.1 Opis učenja pravila

Za učenje pravila koristi se AQ algoritam opisan u [9] i čiji je pseudokod napisan i objašnjen u nastavku.

```
1 AQ(Pozitivni_primjeri, Negativni_primjeri, maxstar, ε,
newPosTres)
2    P' = Pozitivni_primjeri; Pravila = []
3    Dok |P'| > 1 radi
4          p = nasumični primjer iz P'
5          r = STAR(p, Negativni_primjeri, maxstar, ε, newPosTres)
6          P' = P' - (P' ∩ r)
7          Pravila = Pravila + r
```

Pseudokod AQ algoritma

AQ algoritam očekuje skup pozitivnih i negativnih primjera na ulazu te parametar *maxstar* kojim se kontrolira vremenska i memorijska zahtjevnost algoritma. U petlji se iz skupa pozitivnih primjera nasumično odabire jedan i na temelju njega gradi novo pravilo pozivajući *STAR* funkciju čiji je pseudokod napisan i objašnjen u nastavku. Nakon toga se svi pozitivni primjeri pokriveni tim pravilom izbacuju iz liste kandidata za iduće pravilo i postupak se ponavlja dok god ima pozitivnih kandidata za generiranje pravila.

```
1 STAR(p, Negativni_primjeri, maxstar, ε, newPosTres)
2    star = []
3    Za svaki n iz Negativni_primjeri
4          r'  = proširi(p, n, ε)
5          r'' = star ∩ r'
6          sortiraj pravila iz r'' po dobroti
7          star' = []
8          radi
9                pr = izbaci_najbolje(r'')
10               ako broj_novih_pos_pokrivenih sa pr >= newPosTres
11                     star' += pr
12         dok |star'| < maxstar
13         star = star'
14   vrati najbolje pravilo iz star
```

Pseudokod STAR funkcije

STAR funkcija generira novo pravilo postupkom koji je zapravo *Beam Search* [10] širine *maxstar* svih mogućnosti generiranja pravila: u početku se *star* parametar (zapravo ograničena lista stanja) inicijalizira na praznu listu. Za svaki se negativni primjer iz liste danih negativnih primjera generira disjunktni skup mogućih proširenja pravila u tom trenutku. Proširenja se generiraju tako da se promatraju jedan pozitivni i jedan negativni primjer te se za svaku dimenziju linearno interpolira vrijednost između vrijednosti pozitivnog i negativnog primjera za tu dimenziju za parametar $\varepsilon \in [0, 1]$. Tako dobivena vrijednost postaje donjom granicom ako je manja od vrijednosti pozitivnog primjera, odnosno gornjom ako je veća. Novo dobivene vrijednosti donjih ili gornjih granica zapravo čine disjunkciju mogućih proširenja na ona pravila u izgradnji koja se nalaze u listi *star*. U liniji 5 se za svaku dimenziju iz disjunkcije i za svako pravilo iz *star* generira novo moguće pravilo kojemu se u liniji 6 dodjeljuje dobrota s obzirom na omjer pokrivenih pozitivnih i negativnih primjera.

Postupak odabira *maxstar* pravila iz nastalog skupa je sljedeći: gledajući od najboljih prema najgorima u listi kandidata za pravilo odredi koliko dodatnih novih primjera s obzirom na prethodna izabrana pravila to pravilo pokriva. Ako pravilo pokriva više ili jednako od *minPosTres* dodaj ga u privremeni *star'* dok god privremeni *star'* ima manje od *maxstar* elemenata. Jednom kad su obrađeni svi

negativni primjeri iz zadnje nastale liste *star* vraća se najbolje pravilo: ono čiji je omjer pokrivenih pozitivnih i negativnih primjera najveći.

Asimptotska vremenska složenost ovog postupka je $O(m^3 n^2 maxstar)$ gdje je *m* broj primjera za učenje, a *n* njihova dimenzija i učenje se za manji broj primjera odvija dosta sporo. Zbog toga je kao jedan od načina ubrzanja algoritma u *STAR* funkciji umjesto liste svih dostupnih negativnih primjera korištena lista četvrtine najbližih negativnih primjera danom pozitivnom primjeru u višedimenzionalnom prostoru koristeći euklidsku udaljenost kao mjeru udaljenosti. To je podosta ubrzalo postupak dobivanja pravila, ali u nekim slučajevima i rezultiralo boljim pravilima (najbliži susjedi su vjerojatno relevantniji prilikom proširenja) i u konačnici boljom konvergencijom LEM-a.

Za konkretno učenje pravila parametar *maxstar* je postavljen na 7, ε na 0.75, *newPosTres* na 1. Komponenta *AQIndividualInstantiation* u konstruktoru prima dodatno i prije opisane *HPT* te *LPT* parametre, a AQ algoritam se koristi tako da se za pozitivne primjere predaje *HPT* dio populacije, a za negativne preostali dio.

3.2.2 Opis stvaranja novih jedinki

Jednom kad se iz AQ algoritma dobiju pravila koja su po strukturi ekvivalentna pravilima koja se generiraju iz stabala prilikom korištenja C4.5 algoritma, postupak stvaranja novih rješenja je identičan postupku opisanom u 3.1.2.

3.3 Testiranje i rezultati

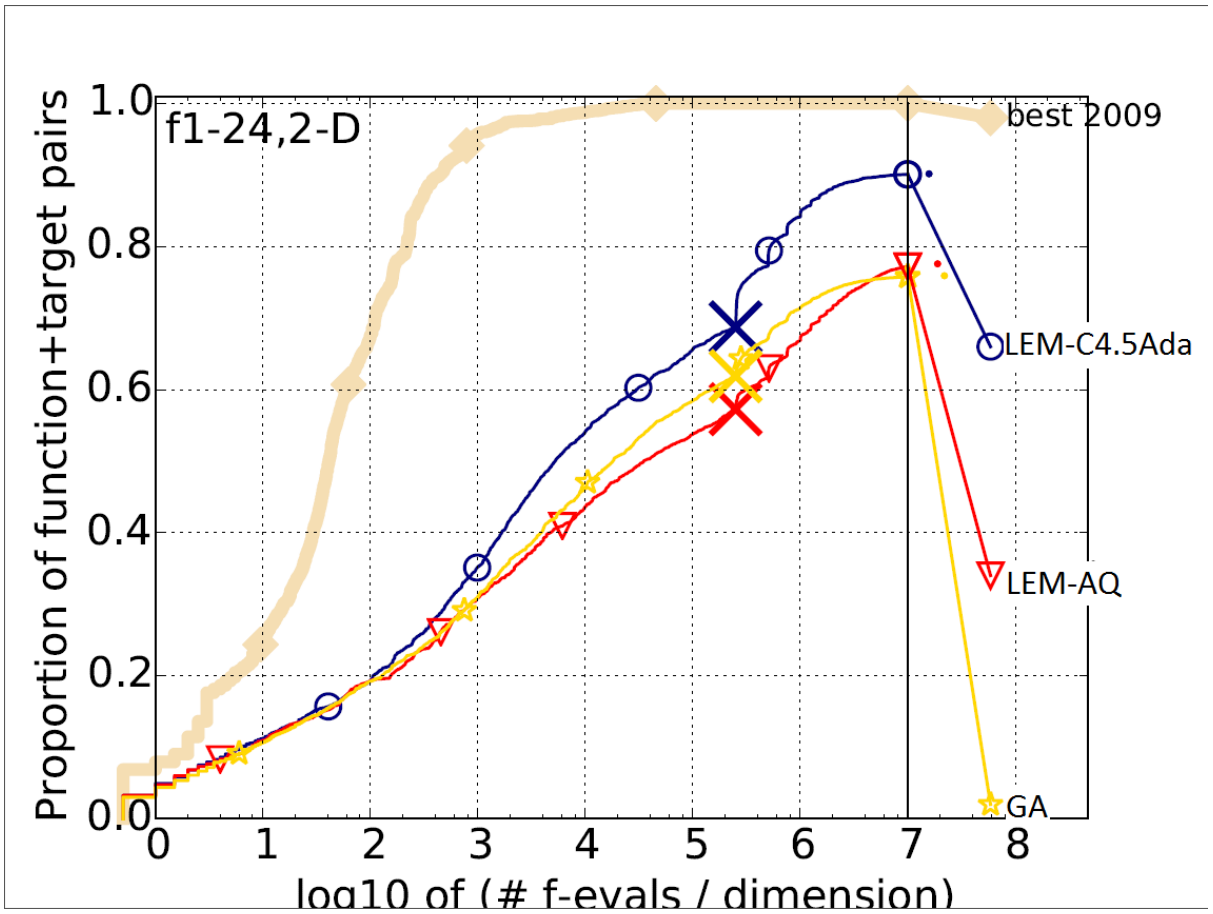
3.3.1 Opis testova

U nastavku su prikazani rezultati usporedbe ova dva opisana postupka s implementacijom generacijskog genetskog algoritma kojemu su parametri i komponente koje dijeli s LEM-om postavljene jednako kao i LEM-u nad COCO platformom za usporedbu optimizacijskih postupaka u realnoj domeni [11]. Testovi su izvršeni nad skupom funkcija bez dodanog šuma (f1-f24), a maksimalni broj evaluacija funkcija cilja je postavljen na $500\ 000$. U testovima je korišten *BestWorstActionProfilingFunction* opisan u poglavlju 2.4 s postavljenim parametrima *learnMin* na 0.07, *probeMin* na 0.15 i *interval* na 10. Početna

populacija se generira nasumičnim uzorkovanjem prostora koji je omeđen intervalom [-5.12, 5.12] za svaku dimenziju.

3.3.2 Rezultati

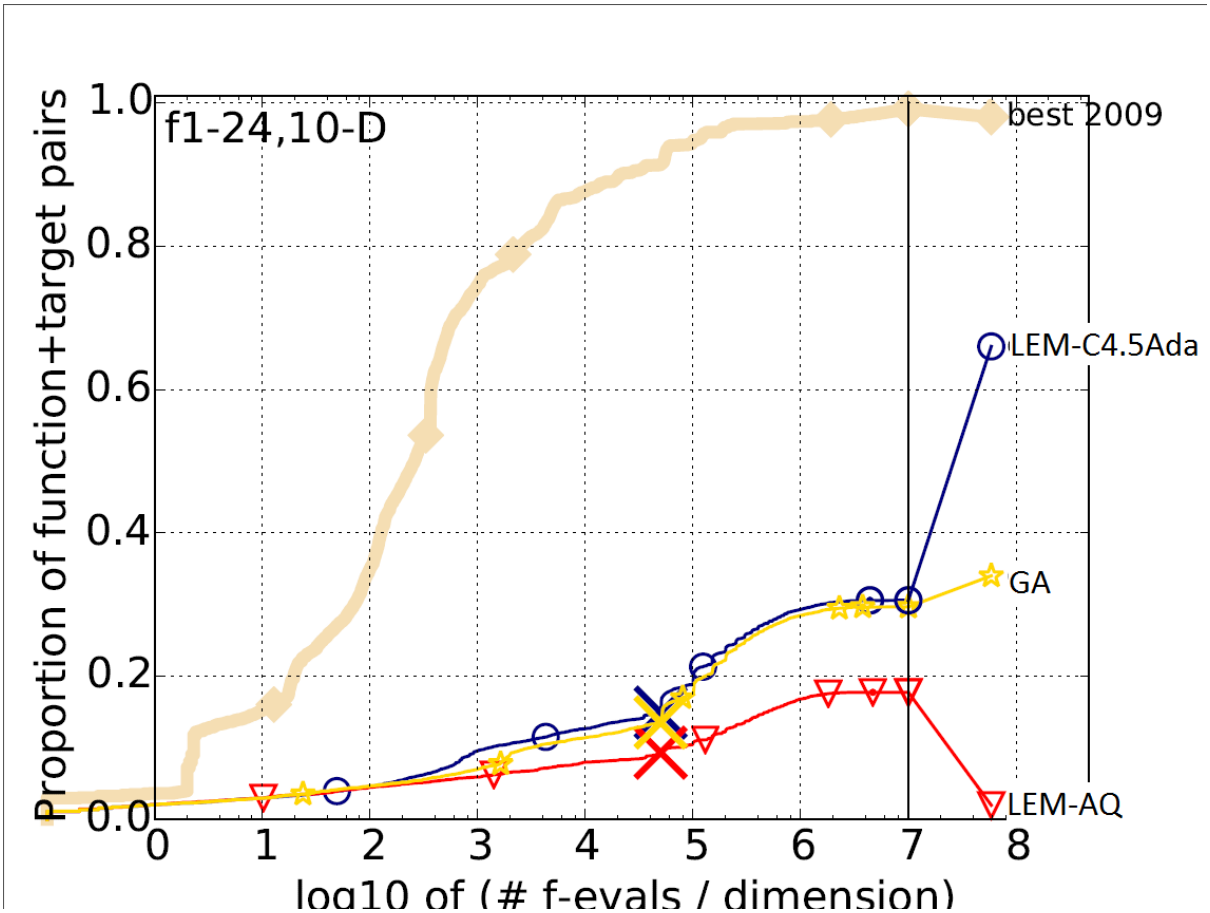
2D



Slika 3.2 Rezultat testiranja algoritama nad funkcijama bez šuma (2D)

Iz grafa na slici Slika 3.2 je vidljivo kako obje inačice LEM-a postižu bolju konvergenciju od običnog GA, s tim da je varijanta LEM-a s kombinacijom C4.5 algoritma i AdaBoost algoritma značajno bolja od one varijante s AQ algoritmom.

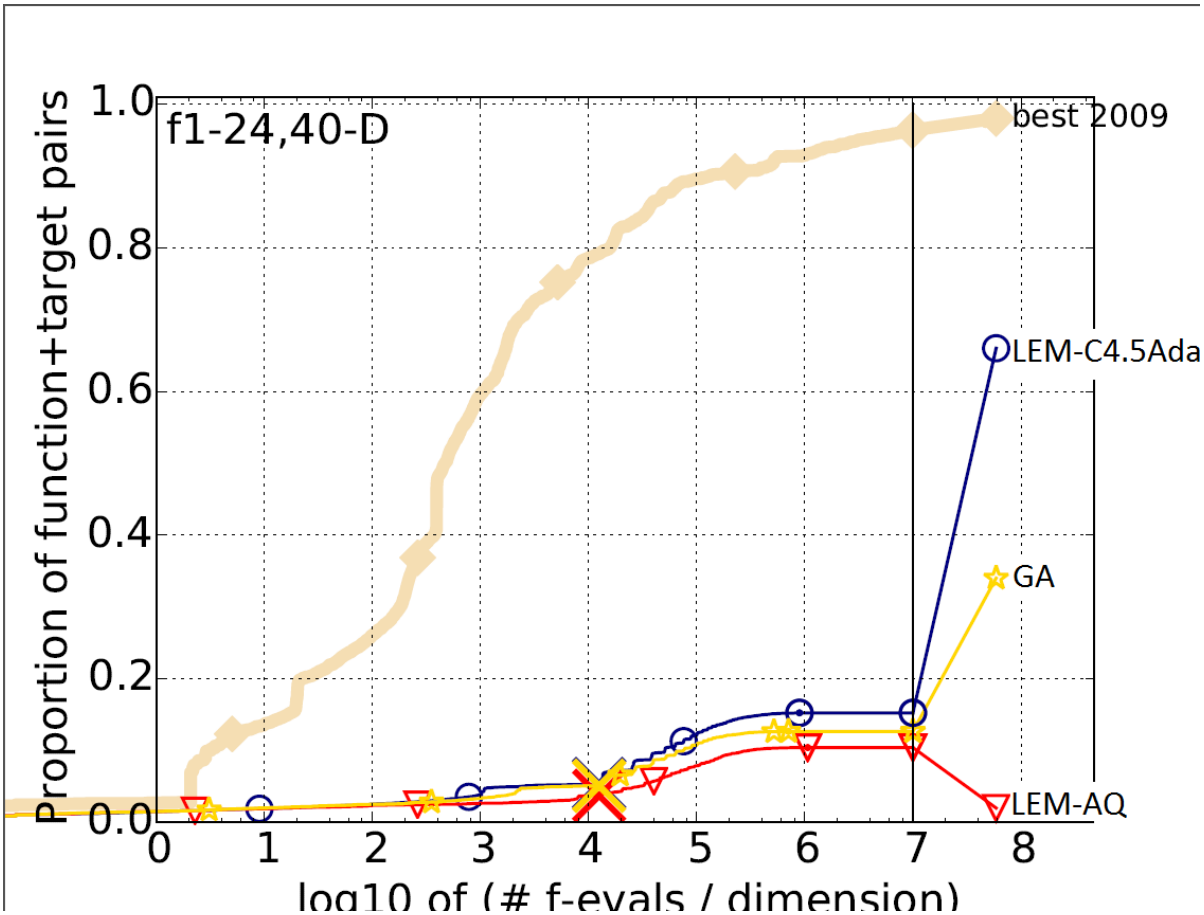
10D



Slika 3.3 Rezultat testiranja algoritama nad funkcijama bez šuma (10D)

Graf na slici Slika 3.3 pokazuje kako obje inačica LEM-a kombinacijom C4.5 algoritma i AdaBoost algoritma postiže konvergenciju sličnu onoj koju pokazuje genetski algoritam, no ipak nešto bolju. Varijanta LEM-a s AQ algoritmom se pokazala osjetno lošijom od druga dva načina optimizacije.

40D



Slika 3.4 Rezultat testiranja algoritama nad funkcijama bez šuma (40D)

Sa slike Slika 3.4 vidljivo je kako najbolju konvergenciju postiže varijanta LEM-a s kombinacijom C4.5 algoritma i AdaBoost algoritma, sličnu ali malo lošiju obični genetski algoritam. Sličnu ali malo lošiju konvergenciju od genetskog algoritma pokazuje varijanta LEM-a s AQ algoritmom.

4. Primjena na prikaz rješenja poljem bitova

Prikaz rješenja poljem bitova je koristan način prikaza rješenja u postupcima optimizacije problema koji je u svojoj prirodi diskretan. Jedan od takvih problema je pronalazak kriptografski ispravnih Booleovih funkcija [12]. Za prikaz rješenja u okružju za ispitivanje korišten je *BitVectorSolution* razred, a za operatore mutacije i križanja njemu prilagođeni operatori:

BitVectorMutMix: mutira jedno rješenje tako da ispremiješa bitove unutar nekog dijela rješenja. Dio rješenja koji će se ispremiješati određuje se nasumičnim odabirom donjeg i gornjeg indeksa bita.

BitVectorOneBitMutation: mijenja stanje (prebacuje) jednom nasumično odabranom bitu u rješenju.

BitVectorOneBreakpointCrossover: križanje s jednom točkom prekida: nasumično odabire indeks bita koji označava točku prekida kod oba roditelja. Prvi potomak nastaje tako da od prvog roditelja kopira bitove od indeksa 0 do isključivo točke prekida, a od drugog od točke prekida do kraja. Drugi potomak nastaje tako da od drugog roditelja kopira bitove od indeksa 0 do isključivo točke prekida, a od prvog od točke prekida do kraja.

BitVectorUniformCrossover: uniformno križanje: početno se generira nasumično polje bitova. Prvo dijete nastaje tako da, gledajući generirani vektor, na mjestima gdje je 0 uzima vrijednost prvog, a na mjestima gdje je 1 uzima vrijednost drugog roditelja na tom mjestu. Drugo dijete na mjestima gdje je 0 poprima vrijednost drugog, a na mjestima gdje je 1 poprima vrijednost prvog roditelja na tom mjestu.

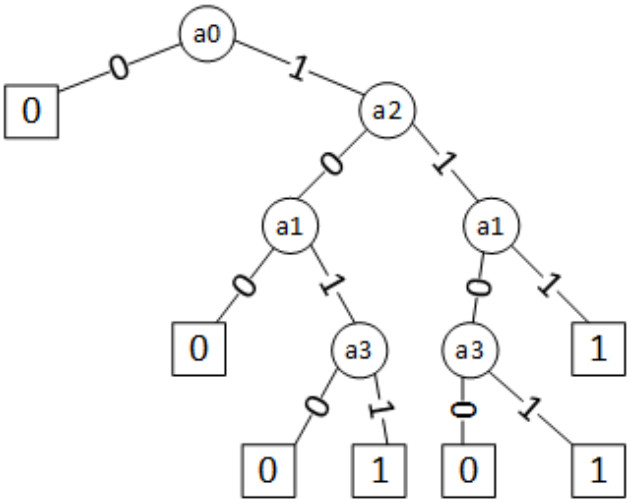
Instanciranje novih jedinki na temelju dostupnih podataka se može napraviti na dva različita načina. Prvi način jest pronalazak pravila koja opisuju razliku između pozitivnih i negativnih primjera za učenje te onda ta pravila iskoristiti za izravno generiranje novih ili modificiranje postojećih rješenja. Drugi način je uporabom nekog algoritma dubinske analize podataka iz njih izvući česte uzorke (postavljene bitove) te na temelju pronađenih uzoraka generirati novo rješenje sa sličnim brojem postavljenih bitova kao što imaju već dostupna rješenja na temelju kojih su pronađeni uzorci.

4.1 Učenje pravila C4.5 i AdaBoost algoritmom

Osim što se algoritam C4.5 koristi za stvaranje stabala odluke za podatke iz kontinuirane domene može se iskoristiti i za istu stvar nad podacima iz diskretne (odnosno binarne) domene. Isto kao i u slučaju kontinuirane domene u poglavlju 3.1, algoritam C4.5 rezultira slabim klasifikatorom pa se za poboljšavanje rezultata učenja primjenjuje učenje AdaBoost-om.

4.1.1 Opis učenja pravila

Stabla odluke koje algoritam C4.5 generira slična su stablu odluke opisanom na slici Slika 3.1 uz razliku da sad atributi poprimaju vrijednosti iz skupa $\{0, 1\}$ pa se odluka u unutarnjim čvorovima stabla temelji na tome poprima li atribut a_i vrijednost 0 ili vrijednost 1. Slika 4.1 to prikazuje.



Slika 4.1 Binarno stablo odluke

Promatraju se pravila koja završavaju s listom označenim s '1' jer je to oznaka koja se daje pozitivnim primjerima za učenje. Pravila koja se mogu izvući iz ovog stabla su:

Pravilo 1: $a_0 = 1 \,\&\, a_2 = 0 \,\&\, a_1 = 1 \,\&\, a_3 = 1$

Pravilo 2: $a_0 = 1 \,\&\, a_2 = 1 \,\&\, a_1 = 0 \,\&\, a_3 = 1$

Pravilo 3: $a_0 = 1 \,\&\, a_2 = 1 \,\&\, a_1 = 1$

Ulazni podaci u AdaBoost algoritam koji koristi C4.5 su binarni vektori koji su označeni u skladu s *HPT* parametrom: HPT dio populacije je označen s '1', dok je

ostatak označen s '0'. Izlaz iz algoritma je onoliko C4.5 stabala odluke koliko je određeno parametrom komponente *iterationsPerc*.

Učenje pravila i stvaranje novih jedinki je implementirano u razredu *BitVectorC45Instatiation* koji uz HPT i LPT (opisane u 2.4) parametre prima još i parametre *iterationsPerc* (opisan u 3.1.1) i parametar *foundRulesToApply* objašnjen u sljedećem potpoglavlju. Asimptotska vremenska složenost postupka učenja je $O(Tmn^2)$ gdje je *T* broj iteracija AdaBoost-a, *m* broj primjera za učenje (veličina populacije), a *n* broj atributa (dimenzija) jednog rješenja.

4.1.2 Opis stvaranja novih jedinki

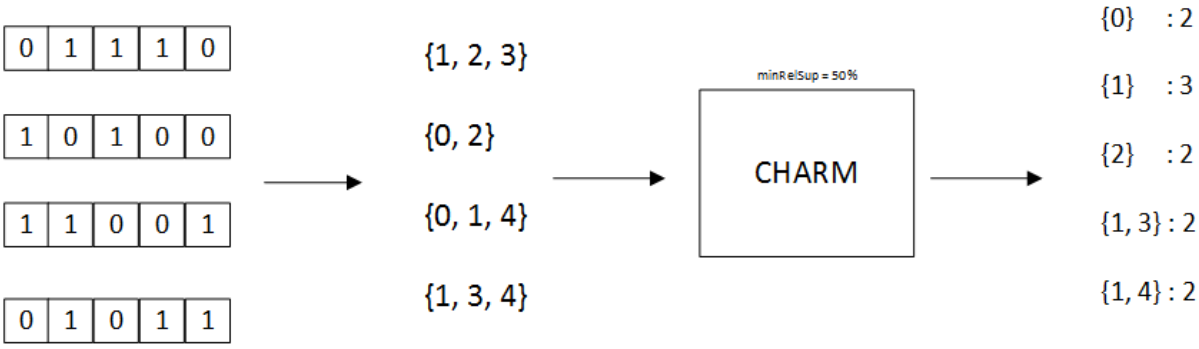
Osnovna zamisao pri stvaranju novih jedinki na temelju pravila je modificirati postojeće jedinke tako da zadovoljavaju neko (ili više) pronađenih pravila. Broj pravila koje će se primijeniti nad jedinkom tako ju modificirajući određuje se parametrom *foundRulesToApply* $\in [0, 1]$ tako da se broj pronađenih pravila koji opisuju primjere označene s '1' pomnoži njime. Garantirano je da će broj primijenjenih pravila biti veći ili jednak 1. Za svako novo rješenje koje se treba stvoriti bira se nasumično jedinka iz trenutne populacije, klonira se i za svako pravilo iz određenog broja nasumično odabranih pravila iz skupa pronađenih pravila rješenju se postavljaju bitovi određeni atributima u pravilu na vrijednosti navedene u pravilu.

4.2 Pronalaženje uzoraka u podacima CHARM algoritmom

Drugi način stvaranja novih rješenja je pronalaženjem uzoraka u podacima i iskorištavanjem tih uzoraka u svrhu izgradnje novih rješenja. Uzorke u polju bitova se može pronaći tako da se polje bitova predstavi skupom indeksa postavljenih bitova i primijeni neki algoritam pronalaska učestalih ili zatvorenih skupova stavki (*engl. frequent/closed itemset mining*). Za pronalaženje zatvorenih uzoraka isprobano je nekoliko algoritama iz *SPMF* Java biblioteke otvorenog koda za dubinsku analizu podataka [13], na kraju se odlučilo uzeti CHARM algoritam opisan u [14].

4.2.1 Opis pronalaženja uzoraka

Uzorci u HPT dijelu populacije rješenja prikazanih vektorom bitova se pronalaze tako da se svako rješenje umjesto vektorom bitova prikaže poljem cjelobrojnih indeksa koji predstavljaju indekse za koje polje bitova poprima vrijednost 1. Jednom kad je izvršena ta transformacija podataka pokreće se CHARM algoritam koji na svom izlazu daje sve česte uzorke u podacima koji zadovoljavaju kriterij da im je relativna frekvencija unutar skupa danih podataka veća ili jednaka od *minRelSup* i koji nisu dijelovi većih uzoraka jednake relativne frekvencije (kriterij zatvorenosti). Ovaj postupak je ilustriran na slici Slika 4.2.



Slika 4.2 Ilustracija postupka pronalaska uzoraka

No, postoji jedan problem kod algoritama pronalaska čestih skupova stavki: kad je dan skup podataka gust, što je ekvivalentno slučaju da ima populaciju rješenja koja imaju puno postavljenih jedinica, algoritmi postaju užasno spori. Jedan od načina borbe protiv sporosti algoritma je dinamičko povećavanje *minRelSup* parametra što rezultira većim pragom frekvencije pojavljivanja čestih uzoraka odnosno manjim brojem pronađenih uzoraka na kraju rada algoritma. U ovom radu dinamičko mijenjanje parametra *minRelSup* je postignuto mjerenjem izvođenja CHARM algoritma nad skupom podataka i brojanjem broja postavljenih bitova u svakoj iteraciji. Parametar *minRelSup* se tada mijenja tako da mu se u svakoj iteraciji dodaje (ako je vrijeme izvođenja poraslo), odnosno oduzima (ako je vrijeme izvođenja palo), nekakva mala vrijednost koja je proporcionalna s omjerom izmjerenog trajanja izvođenja s izmjerenim trajanjem izvođenja prve iteracije i relativnom razlikom broja postavljenih bitova. Intuicija oko ovoga je da se pretpostavlja kako je vrijeme izvođenja algoritma nad prvim danim skupom

podataka zadovoljavajuće pa ovaj postupak mijenjanja *minRelSup* parametra ide u smjeru zadržavanja tog vremena izvođenja.

Pronalazak uzoraka je implementiran u komponenti *BitVectorCharmInstatiation* koja kao parametre pri inicijalizaciji prima *HPT*, *LPT* i *minRelSup*. Vremenska asimptotska složenost postupka učenja je $O(l * |C|)$ gdje je l prosječna frekvencija pronađenih uzoraka, a $|C|$ veličina skupa pronađenih uzoraka što je ovisno o dobivenim podacima.

4.2.2 Opis stvaranja novih jedinki

Iz dobivenih čestih uzoraka u *HPT* dijelu populacije nove jedinke se stvaraju tako da se najprije pretpostavi kako se broj postavljenih jedinica u svakoj jedinki u populaciji ravna po normalnoj distribuciji: određuje se prosječan broj postavljenih jedinica po jedinki (parametar μ u normalnoj distribuciji) i varijanca broja jedinica po jedinki (parametar σ^2 u normalnoj distribuciji). Za svaku jedinku koju treba generirati iz određene normalne distribucije određuje se broj jedinica koje će ta jedinka imati i onda se nasumično odabiru uzorci iz skupa pronađenih uzoraka te se novom rješenju postavljaju bitovi u skladu s onim što se nalazi u uzorku sve dok se ne zadovolji traženi broj postavljenih bitova.

4.3 Testiranje i rezultati

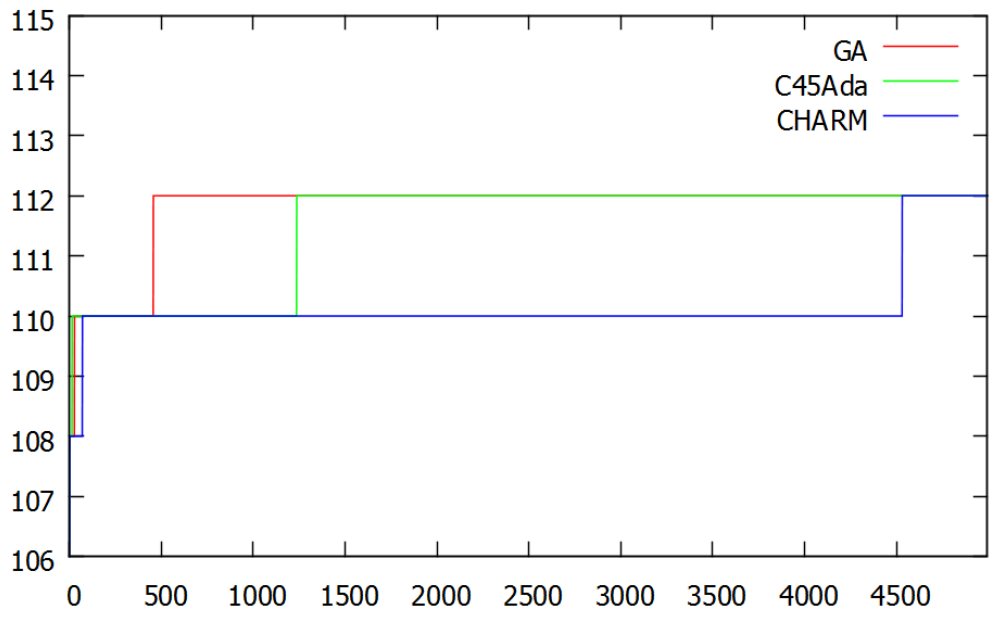
Testiranje optimizacije se izvodi na problemu izgradnje Booleovih funkcija optimiranjem nekih njihovih svojstava. Booleova funkcija od n varijabli poprima 2^n binarnih vrijednosti, pa se funkcija za n varijabli prikazuje vektorom od 2^n bitova. Neka svojstva Booleovih funkcija koja se uzimaju u obzir pri optimizaciji su balansiranost (ujednačenost broja 1 i broja 0 u funkciji), nelinearnost (minimalna udaljenost funkcije od skupa svih afinih (linearnih) funkcija nad *n* binarnih definiranih varijabli), algebarski stupanj (maksimalna težina monomijala u algebarskoj normalnoj formi funkcije), korelacijski imunitet (stupanj statističke nezavisnosti između linearnih kombinacija ulaznih bitova i izlaza) i algebarski imunitet (minimalni stupanj ne-nul funkcija *g* za koje vrijedi $f*g = 0$ ili $(1 \oplus f)*g=0$). Više o ovim svojstvima se može pročitati u [15].

4.3.1 Opis testova

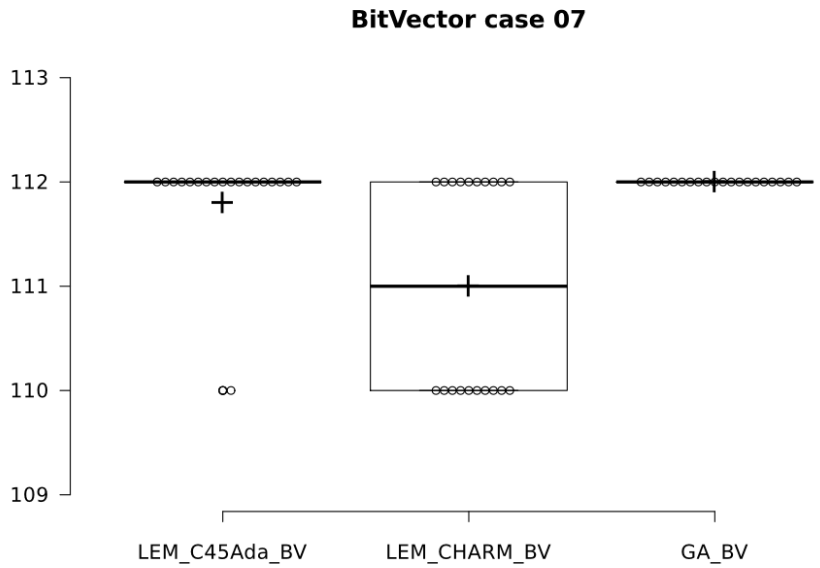
U nastavku su prikazani rezultati usporedbe ova dva opisana postupka s implementacijom generacijskog genetskog algoritma kojemu su parametri i komponente koje dijeli s LEM-om postavljene jednako kao i LEM-u nad 3 slučaja stvaranje Booleove funkcije: prvi način je optimizacijom funkcije koja maksimizira balans i nelinearnost funkcije (*BitVectorFunctionCase7*) te drugi način minimizira kombinaciju balansa, nelinearnosti, korelacijskog imuniteta i algebarskog imuniteta (*BitVectorFunctionCase9*). Maksimalni broj evaluacija funkcija cilja je postavljen na $500\ 000$. U testovima je korišten *BestWorstActionProfilingFunction* opisan u poglavlju 2.4 s postavljenim parametrima *learnMin* na 0.07, *probeMin* na 0.15 i *interval* na 10. Početna populacija se stvara nasumično tako da u svakom stvorenom početnom rješenju vjerojatnost da je neki bit postavljen iznosi 0.5. Parametar *minRelSup* u testovima sa *CHARM* algoritmom je postavljen na 0.6.

4.3.2 Rezultati

BitVectorFunctionCase7



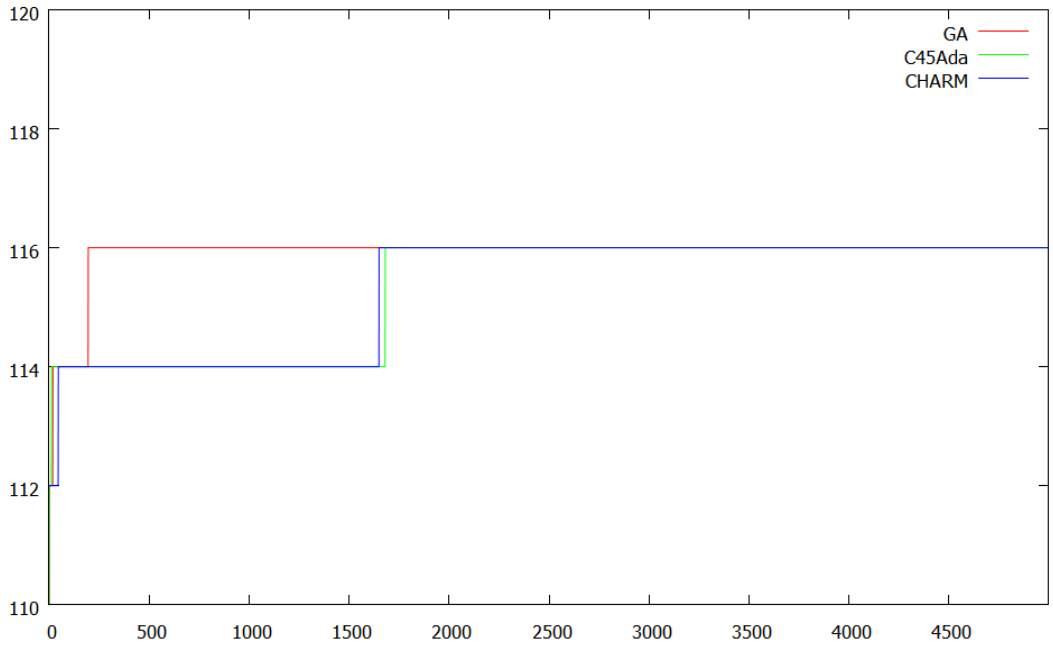
Slika 4.3 Kretanje medijana funkcije cilja kroz iteracije za sve postupke (case 7)



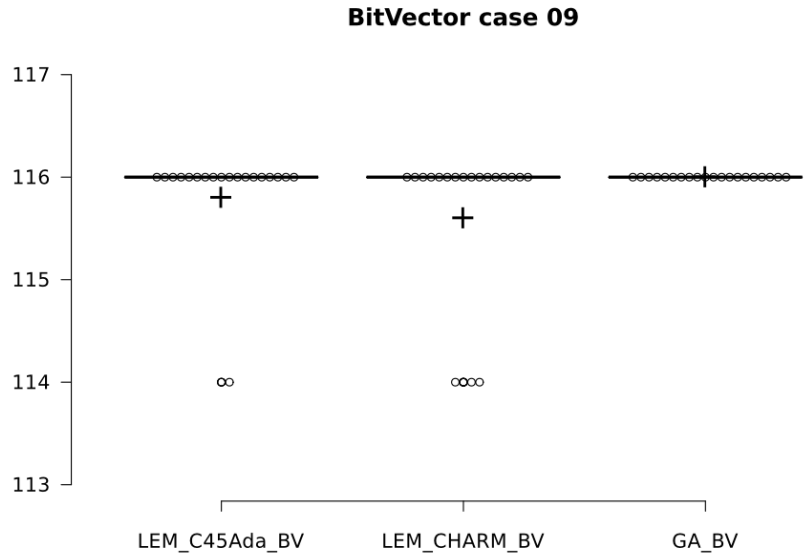
Slika 4.4 Boxplot pronađenih rezultata u 20 testova svakog postupka (case 7)

Iz slika Slika 4.3 i Slika 4.4 se može zaključiti kako oba postupka daju lošije rezultate od genetskog algoritma s kojim se uspoređuju, s time da varijanta LEM-a s AdaBoostom i C4.5 algoritmom pokazuje nešto bolju konvergenciju od varijante LEM-a s CHARM algoritmom.

BitVectorFunctionCase9



Slika 4.5 Kretanje medijana funkcije cilja kroz iteracije za sve postupke (case 9)



Slika 4.6 Boxplot pronađenih rezultata u 20 testova svakog postupka (case 9)

Iz slika Slika 4.5 i Slika 4.6 se vidi kako oba postupka daju lošije rezultate od genetskog algoritma s kojim se uspoređuju (iako im je medijan pronađenih rješenja isti te kako obje varijante LEM-a pokazuju sličnu brzinu konvergencije na ovom problemu.

5. Primjena na prikaz rješenja permutacijom

Neki problemi poput problema trgovačkog putnika (*engl. Traveling Salesman Problem)* ili problema popunjavanja kutija (*engl. Bin Packing*) imaju rješenja koja je pogodno prikazati permutacijom što zbog lakše manipulacije pojedinih rješenja, što zbog povezanosti rješavanja tih problema nekim heuristikama koje ovise o redoslijedu dospijeća dijelova rješenja. Permutacija je polje pozitivnih cjelobrojnih brojeva u granicama od $[0, n\text{-}1]$ koji su unutar polja ispremiješani. Za prikaz rješenja u okružju za ispitivanje korišten je *PermutationSolution* razred, a za operatore mutacije i križanja njemu prilagođeni operatori:

Permutation2OPTMutation: Permutaciju mutira određujući početni i završni indeks dijela koji će se mutirati i preokreće taj interval.

Permutation2ExchangeMutation: Određuje dva nasumična indeksa u permutaciji te za ta dva indeksa zamjenjuje vrijednosti.

PermutationCOSACrossover: križanje dvije permutacije se izvodi tako da se u prvoj permutaciji nasumično odabere jedno polje i vrijednost koja se nalazi u tom polju se traži u drugoj permutaciji. Indeks nakon nasumično odabranog označava početak intervala. Vrijednost iz druge permutacije koja se nalazi neposredno nakon pronađene pronalazi u prvoj permutaciji i indeks pronađene vrijednosti u prvoj permutaciji označava kraj intervala. Nova jedinka se stvara tako da se vrijednosti van određenog intervala kopiraju iz prvog roditelja na uobičajen način, a vrijednosti unutar intervala obrnuto.

PermutationOrderCrossover: križanje zahtijeva dva roditelja i generira dva potomka. U početku se nasumično odabiru indeksi početka i završetka intervala. Prvi potomak se generira tako da se na mjestima koji su u intervalu kopiraju vrijednosti iz intervala koje pripadaju prvom roditelju, a na mjestima van intervala uzimaju vrijednosti drugog roditelja koje se ne nalaze u novoizgrađenoj permutaciji u redoslijedu kojim se pojavljuju u drugom roditelju krenuvši od kraja intervala u novoizgrađenoj i permutaciji drugog roditelja. Drugi se potomak izgrađuje na isti način uz zamjenu uloga prvog i drugog roditelja.

Stvaranje novih permutacija postupkom učenja se temelji na pronalasku čestih uzoraka koji predstavljaju redoslijed vrijednosti u permutacijama. Zapravo postoje

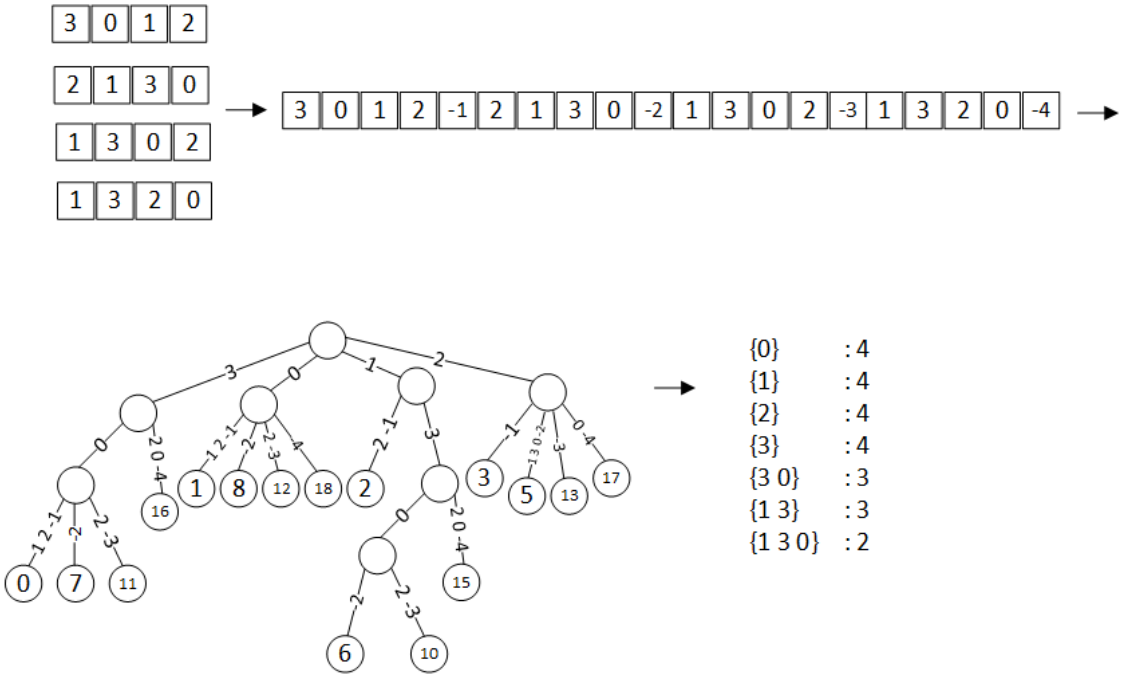
dvije vrste uzoraka: prvi su uzorci kod kojih između vrijednosti u uzorku nije predviđen prostor za druge vrijednosti koje se možda pojavljuju u podacima, dakle ti uzorci govore o čestim grupama vrijednosti koje se pojavljuju u podacima. Druga vrsta uzoraka govori isključivo o relativnom redoslijedu navedenih vrijednosti, no ne i o njihovom apsolutnom razmještaju u konkretnim podacima. Izrada novog rješenja temelji se na tome da se određeni postotak indeksa rješenja postavi na vrijednosti u redoslijedu kakvom su pronađene u uzorcima, a preostali postotak popuni nasumičnim vrijednostima.

5.1 Pronalazak uzoraka izgradnjom sufiksnog stabla

Prva vrsta uzoraka koji se mogu pronaći u skupu permutacija jesu grupe istih vrijednosti i njihovo se pronalaženje svodi na zadatak pronalaženja jednakih sljedova slova u nekom velikom tekstu. Pogodna struktura za pronalazak jednakih sljedova u velikom polju je sufiksno stablo koje se može izgraditi u vremenu linearnom s obzirom na broj znakova (vrijednosti) na ulazu.

5.1.1 Opis pronalaženja uzoraka

Postupak pronalaska ponavljajućih grupa vrijednosti prikazan je na slici Slika 5.1.



Slika 5.1 Prikaz postupka pronalaska uzorka korištenjem sufiksnog stabla

Najprije se od skupa permutacija napravi veliko polje kojim se daje na ulaz algoritma za izgradnju sufiksnog stabla. Između permutacija u velikom polju postavljaju se završni znakovi kako bi se permutacije međusobno odvojile. Algoritam gradi sufiksno stablo čitajući znak po znak iz polja. Jednom izgrađeno sufiksno stablo u svojim listovima sadrži indekse na kojima određeni sufiks (gledajući od korijena prema listu) započinje u ulaznom nizu. Na prethodnoj slici nisu dodani listovi sufiksa koji započinju na završne znakove (negativne vrijednosti). U takvom izgrađenom sufiksnom stablu svakom se čvoru izračunava broj listova koji pokriva. Taj broj onda označava na koliko se mjesta sufiks koji se nalazi između korijena i promatranog lista ponovio u ulaznom nizu. U ovom primjeru posebno su zanimljivi podnizovi koji se ponavljaju više od jednog puta i oni su navedeni u završnom koraku na gornjoj slici.

Konkretan algoritam kojim se pronalaze uzorci ovisi o algoritmu izgradnje sufiksnog stabla. Jedan od najpoznatijih algoritama te namjene je Ukkonenov algoritam koji je opisan u [21] i koji izgradnju sufiksnog stabla obavlja u vremenu linearnom s obzirom na broj znakova u ulaznom nizu. Zbog toga je postupak pronalaska uzoraka na ovaj način asimptotske vremenski složenosti $O(kN)$ gdje je k broj permutacija, a N njihova veličina.

Pronalazak uzoraka i stvaranje novih jedinki je implementiran u razredu *SuffixTreeIndividualInstatiation* koji prilikom konstrukcije prima parametre *HPT, LPT, minRelSup* i *maxPermToBuild*. Parametri *HPT* i *LPT* su prethodno objašnjeni, parametar *minRelSup* određuje minimalnu relativnu frekvenciju koju uzorak mora imati s obzirom na podatke u kojima se pronalazi (*HPT* dio populacije), a parametar *maxPermToBuild* biti će objašnjen kasnije.

5.1.2 Opis stvaranja novih jedinki

Jednom kad su pronađeni uzorci potrebno je te uzorke nekako ugraditi u nova rješenja. Prilikom stvaranja razred postavlja se parametar *maxPermToBuild* $\in [0, 1]$ koji određuje koji dio permutacije će se popuniti koristeći pronađene uzorke, a koji će se napuniti nasumično: pronađenim uzorcima se ispunjava $maxPermToBuild * N$ mjesta u permutaciji, gdje je *N* veličina permutacije. Postupak se izvodi tako da se za svako novo rješenje u svakom koraku odredi lista mogućih uzoraka koje je moguće ugraditi u rješenje u izgradnji: to su uzorci

koji duljine manje ili jednake od preostalog broja mjesta za popunjavanje uzorcima koji u sebi nemaju vrijednost koja je već dodana rješenju u izgradnji. Svim uzorcima iz liste mogućih se zatim određuje dobrota po formuli (5.1) koja je osmišljena za ovaj problem i nakon toga iz liste se proporcionalno dobroti bira jedan uzorak koji se ugrađuje u rješenje na nasumičnu poziciju. Pozicije koje su ostale neispunjene popunjavaju se nasumičnim vrijednostima.

$$pattern\_fitness = \frac{\log(1 + freq)}{\log(1 + \max Freq) * (1 + rem\_size - pattern\_size)} \qquad (5.1)$$

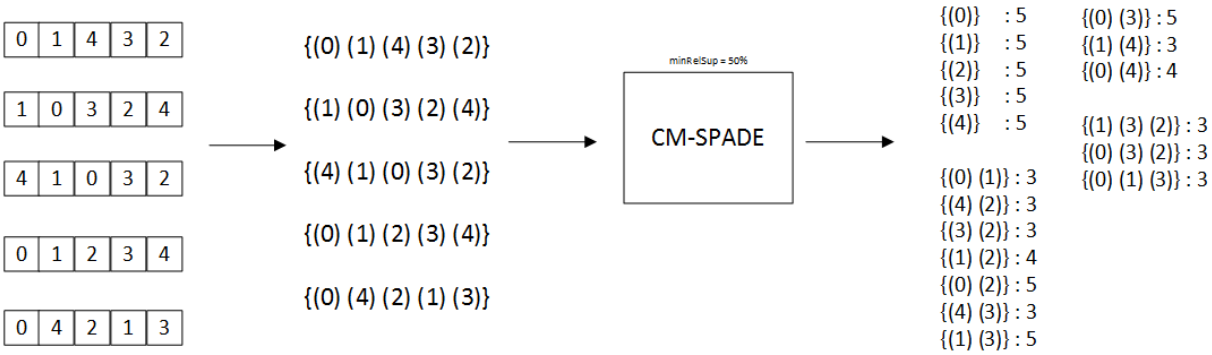
Ovom formulom se pokušava postići favoriziranje uzoraka koji su veličine koja je bliža preostalom broju mjesta koje treba popuniti u rješenju u izgradnji. Favoriziraju se uzorci koji se češće pojavljuju u skupu uzoraka, ali se logaritmom pokušava spriječiti dominacija prečestih uzoraka. Parametar *freq* označava učestalost promatranog uzorka, *maxFreq* učestalost najčešćeg uzorka iz liste mogućih u nekom koraku, *rem_size* preostali broj mjesta za popuniti u permutaciji, *pattern_size* veličina uzorka.

5.2 Pronalazak uzoraka primjenom CM-SPADE algoritma

Druga vrsta uzoraka koji se mogu pronaći u skupu permutacija su uzorci koji govore o relativnom redoslijedu elemenata u permutaciji, ali ne postavljaju nikakvo ograničenje na apsolutnu udaljenost između ta dva elementa. Dakle, ti uzorci otkrivaju informaciju o tome koji se element pojavljuje prije nekog drugog elementa, ali ne uzimaju u obzir koliko prije. Dubinska analiza ovakvih podataka ima primjenu u trgovinama kad se želi saznati redoslijed (i moguće vremensko uvjetovanje) kupovanja različitih proizvoda promatrajući proizvode koje su kupovali kupci u vremenskom slijedu (*engl. sequential mining*). Jedan od najbržih algoritama dubinske analize sekvencijalnih podataka je CM-SPADE koji je opisan u [23].

5.2.1 Opis pronalaženja uzoraka

Postupak pronalaska uzoraka prikazan je na slici Slika 5.2.



Slika 5.2 Prikaz dobivanja uzoraka algoritmom CM-SPADE

Prvo se skup permutacija pogodan za učenje uzoraka (HPT dio populacije) pretvori u format kojeg prima algoritam tako da se svaki broj u permutaciji izdvoji u zaseban skup. Algoritam kao parametar uzima *minRelSup,* odnosno minimalnu frekvenciju uzorka u odnosu na veličinu ulaznog skupa podataka. Algoritam CM-SPADE izbacuje uzorke u obliku $\{(a)\ (b)\ \ldots\}$ koji predstavljaju slijed kojim se pojavljuju podaci u uzorku i njihovu frekvenciju u skupu podataka. Tako se npr. u uzorku $\{(0)\ (1)\ (3)\} : 3$ može iščitati da 0 dolazi prije 1 i 3, 1 prije 3, a uzorak se pojavljuje 3 puta u ulaznom skupu podataka.

Komponenta koja ostvaruje učenje uzoraka i stvaranje novih jedinki na temelju njih je *CMSPADEIndividualInstatiation* i pri konstrukciji prima parametre *HPT*, *LPT*, *minRelSup, maxPermToBuild, maxLengthSkip i avgLengthSkip*. Opis parametara *HPT* i *LPT* se nalazi u prethodnim poglavljima, parametar *minRelSup* određuje minimalnu relativnu frekvenciju koju uzorak mora imati s obzirom na podatke u kojima se pronalazi (*HPT* dio populacije), a parametar *maxPermToBuild* je objašnjen u sljedećem potpoglavlju.

Pokazuje se kako je pronalazak ove vrste uzoraka u kontekstu evolucijskog algoritma veoma memorijski zahtjevan proces zbog toga što populacija rješenja u kasnijim iteracijama postupka optimizacije teži prema skupu veoma sličnih rješenja. Broj pronađenih čestih uzoraka u takvim situacijama je ograničen s 2^N gdje je N dimenzija rješenja. Ta činjenica se u praksi manifestira tako da jednom pokrenuti postupak dubinske analize podataka može zauzeti svu dostupnu memoriju u radnoj memoriji, ali i na disku. Kako bi se te situacije izbjegle poseglo se za dinamičkim mijenjanjem parametra *minRelSup* slično postupku opisanom u

poglavlju 4.2.1: u svakoj iteraciji se dodaje mala vrijednost koja je proporcionalna kvadratu omjera trenutne i prve iteracije pronalaska uzoraka ako je vrijeme izvođenja poraslo ili oduzima mala vrijednost koja je proporcionalna istom omjeru ako je vrijeme izvođenja palo. Parametar *minRelSup* nikad ne poprima vrijednost ispod početne postavljene vrijednosti. No prilikom testiranja pokazalo se kako niti ova intervencija ne daje dovoljno stabilan rad algoritma pa se za predviđanje memorijski zahtjevnog postupka koristi analiza uzoraka dobivenih izgradnjom sufiksnog stabla (opisano u poglavlju 5.1.1) zato što je to poprilično jeftina operacija. Tu su iskorištena posljednja dva parametra koje prima komponenta za učenje uzoraka: ako je veličina maksimalna veličina uzorka pronađenog sufiksnim stablom veća ili jednaka od *maxLengthSkip* ili je prosječna veličina uzorka pronađenog sufiksnim stablom veća ili jednaka *avgLengthSkip* postupak učenja ne pokreće algoritam dubinske analize podataka i stvara nasumična rješenja. Ta dva parametra određena su empirijski za svaki testni slučaj.

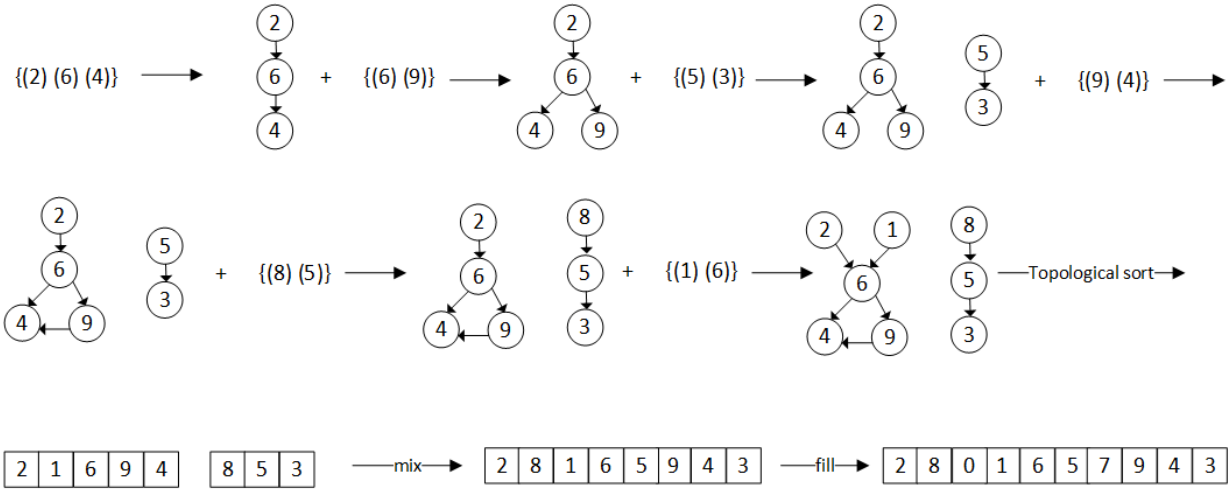
5.2.2 Opis stvaranja novih jedinki

Jednom kad se pronađu željeni uzorci potrebno je iz njih nekako izgraditi novo rješenje. Izgradnja rješenja sljedovima je u nekim segmentima slična izgradnji rješenja iz poglavlja 5.1.2. Parametrom *maxPermToBuild* $\in [0, 1]$ određuje se količina vrijednosti u novoizgrađenom rješenju koja će biti postavljena korištenjem uzoraka, dok će se ostali dio rješenja popuniti nasumičnim vrijednostima.

Gledajući više uzoraka odjednom, njihove međusobne relacije moguće je modelirati grafom s usmjerenim bridovima i iz jednom izgrađenog grafa izvući rješenje tj. dio permutacije koji se izgrađuje uzorcima. Prilikom izgradnje grafa uzorci se iz skupa uzoraka biraju na način kako je to opisano u poglavlju 5.1.2: u svakom koraku gradi se lista mogućih uzoraka koji su duljine manje ili jednakoj duljini preostaloj za izgradnju novog rješenja (dopuštaju se vrijednosti koje se već nalaze u grafovima), uzorcima se dodjeljuje dobrota opisana funkcijom (5.1) i nakon toga se proporcionalno dobroti bira uzorak kandidat za ulazak u graf. U praksi se pokazalo kako je postupak biranja uzoraka proporcionalno njihovoj dobroti previše spor kad broj pronađenih uzoraka prijeđe granicu od 15000 pa se u tom slučaju provodi nasumično biranje uzoraka iz skupa pronađenih. Uzorak se u graf dodaje tako da se promatraju parovi vrijednosti u uzorku i za se svaki par

vrijednosti, bez obzira postojale li one ili ne u grafu, dodaje brid od prve prema drugoj. Ako se dodavanjem uzorka u graf stvori ciklus, operacija dodavanja se poništava i bira se drugi uzorak.

Jednom kad je prestalo dodavanje novih uzoraka u graf nad svakom nepovezanom komponentom u grafu se izvrši algoritam topološkog sortiranja (*engl. topological sorting*, [25]) koji rezultira poljem vrijednosti u pravilnom redoslijedu za svaku komponentu. Dobivena polja se stapaju u jedno s obzirom na pronađeni redoslijed vrijednosti nasumičnim odabirom iz kojeg se pronađenog polja vadi iduća vrijednost za stavljanje u zajedničko polje. U stvoreno zajedničko polje na nasumičnim mjestima se dodaju praznine do željene veličine novog rješenja i te se praznine nadopunjuju nasumičnim vrijednostima. Postupak je ilustriran slikom Slika 2.1.



Slika 5.3 Ilustracija postupka stvaranja rješenja koristeći podatke o sljedovima

5.3 Testiranje i rezultati

LEM za slučaj permutacije i s dvije prethodno opisane komponente se testira nad standardnim problemima trgovačkog putnika koji se mogu pronaći u biblioteci TSPLIB [22]. Rezultati LEM-a uspoređuju se s rezultatima običnog genetskog generacijskog algoritma koji sve parametre koje dijeli s LEM-om ima postavljene na istu vrijednost kao i LEM.

5.3.1 Opis testova

Za testiranje oba postupka odabrani su sljedeći problemi iz TSPLIB-a: *eil51*, *eil101*, *kroA200* i *rat575* koji spadaju u skupinu simetričnih problema trgovačkog putnika. Maksimalni broj evaluacija funkcija cilja je postavljen na $1\ 000\ 000$. U testovima je korišten *StagnationAvoidActionProfilingFunction* opisan u poglavlju 2.4 s postavljenim parametrima *interval* na 100, *increment* na 10, *stagnationThreshold* na 10^{-5} te *okThreshold* na 10^{-4}. Parametar *minRelSup* u testovima s *CM-SPADE* algoritmom je postavljen na 0.7, dok je taj parametar u testovima sa sufiksnim stablom postavljen na 0.33. Parametar *maxPermToBuild* je za oba načina u svim testovima postavljen na 0.8. Parametri *maxLengthSkip* i *avgLengthSkip* u slučaju korištenja *CM-SPADE* algoritma su postavljeni kako to piše u tablici 5.1.

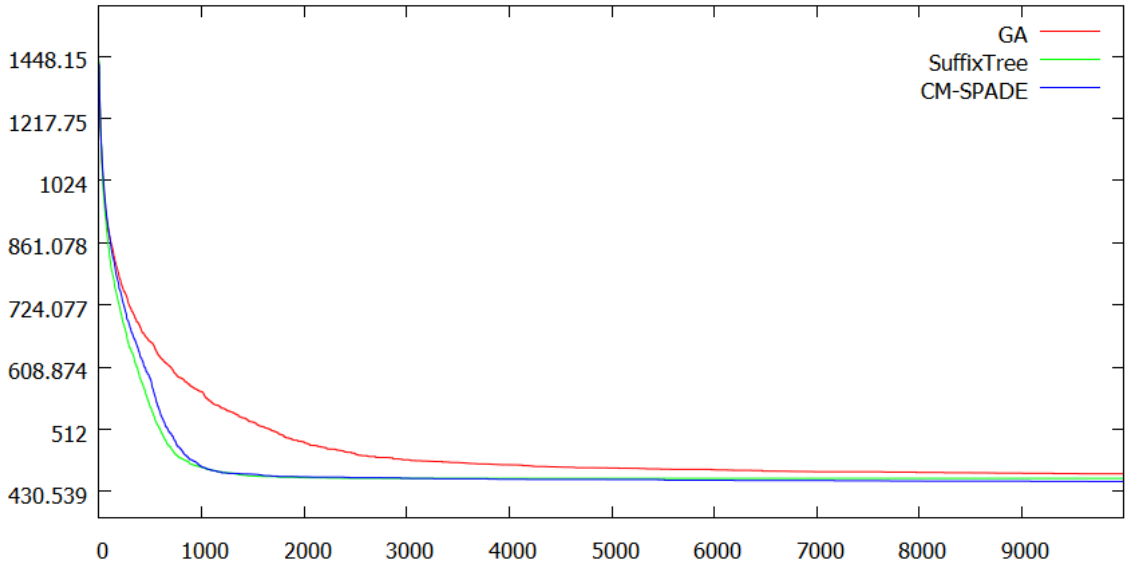
5.1 vrijednosti parametara maxLengthSkip i avgLengthSkip za sve slučajeve

	maxLengthSkip	*avgLengthSkip*
eil51	14	1.75
eil101	13	1.55
kroA200	12	1.35
rat575	11	1.15

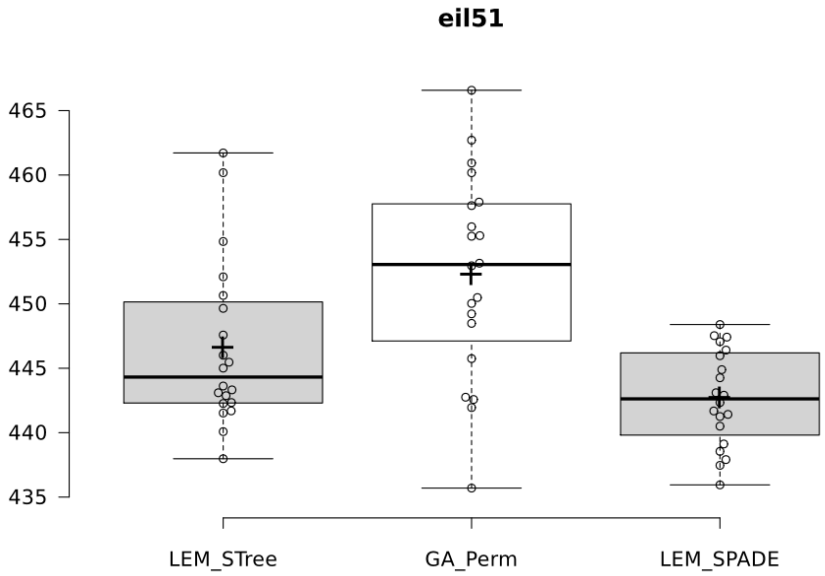
Početna populacija se sastoji od nasumično stvorenih permutacija. Dva prikazana načina stvaranja novih jedinki uspoređena su s genetskim algoritmom koji parametre koje dijeli s LEM-om ima postavljene kao što ih ima i LEM.

5.3.2 Rezultati

eil51



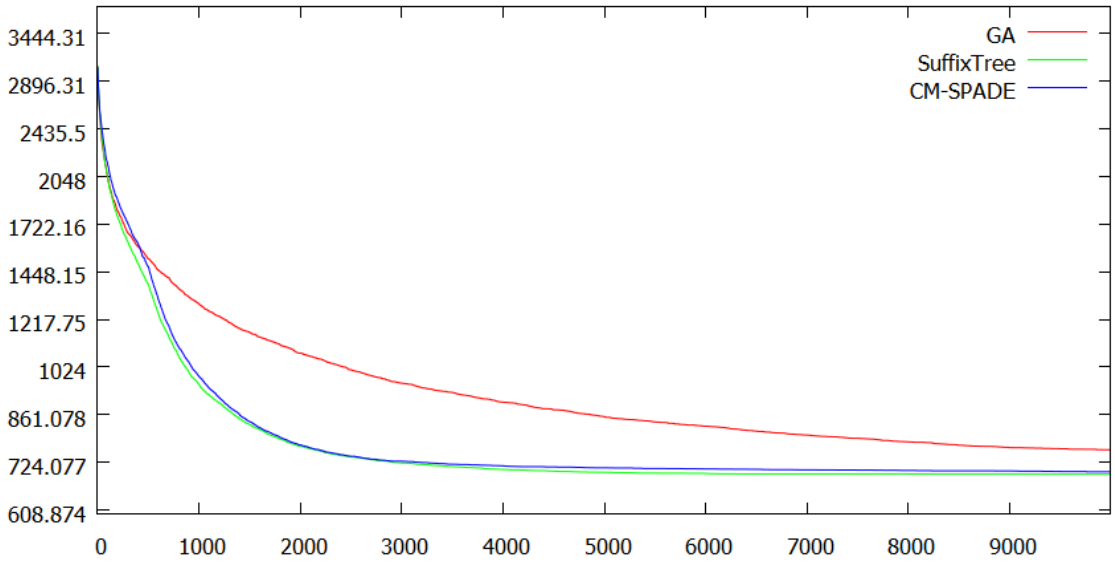
Slika 5.4 Kretanje prosjeka funkcije cilja kroz iteracije za sve postupke (eil51)



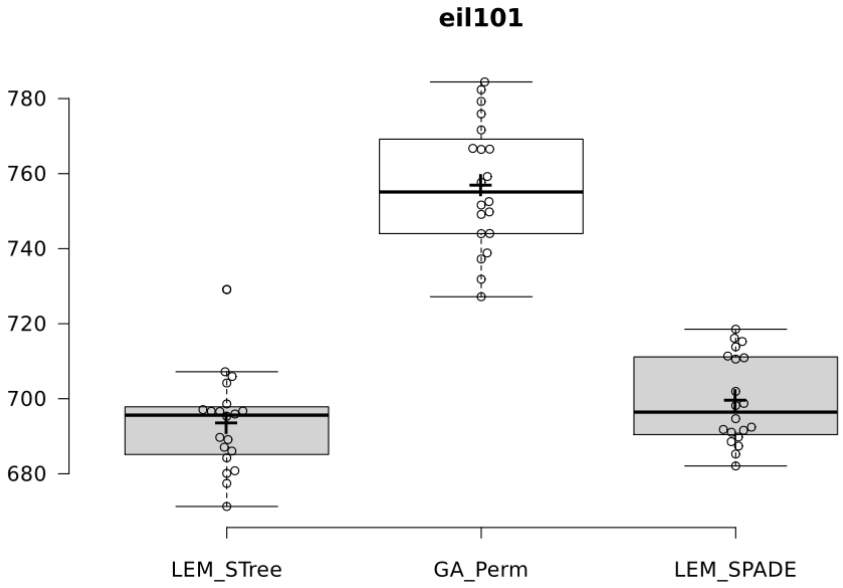
Slika 5.5 Boxplot pronađenih rezultata u 20 testova svakog postupka (eil51)

Slika 5.4 i Slika 5.5 pokazuju kako oba načina implementirana za LEM brže konvergiraju i dolaze do nešto boljeg rješenja od genetskog algoritma s kojim je LEM uspoređivan. Iz slike Slika 5.5 zaključujem kako je izvođenje SPADE varijante učenja završilo s nešto konzistentnijim i boljim rezultatima.

eil101



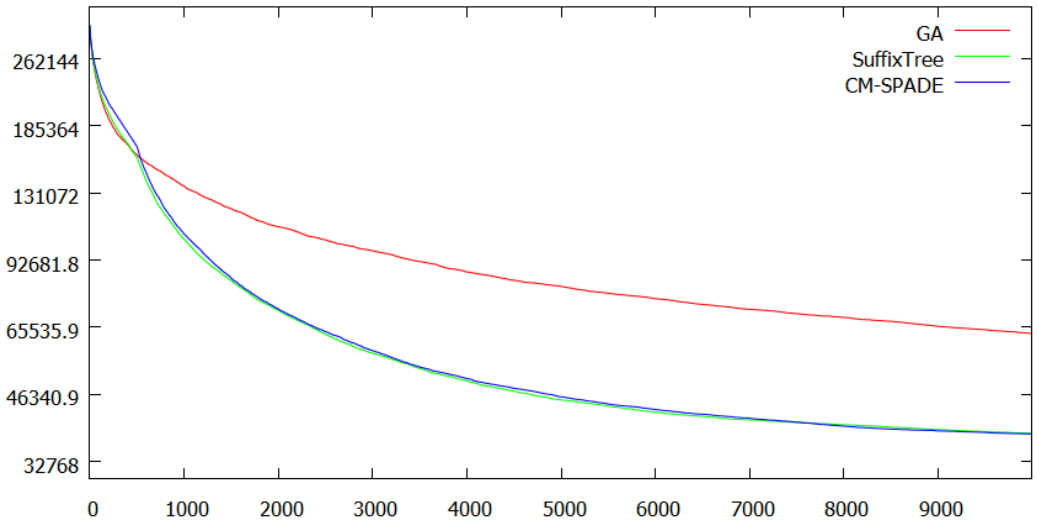
Slika 5.6 Kretanje prosjeka funkcije cilja kroz iteracije za sve postupke (eil101)



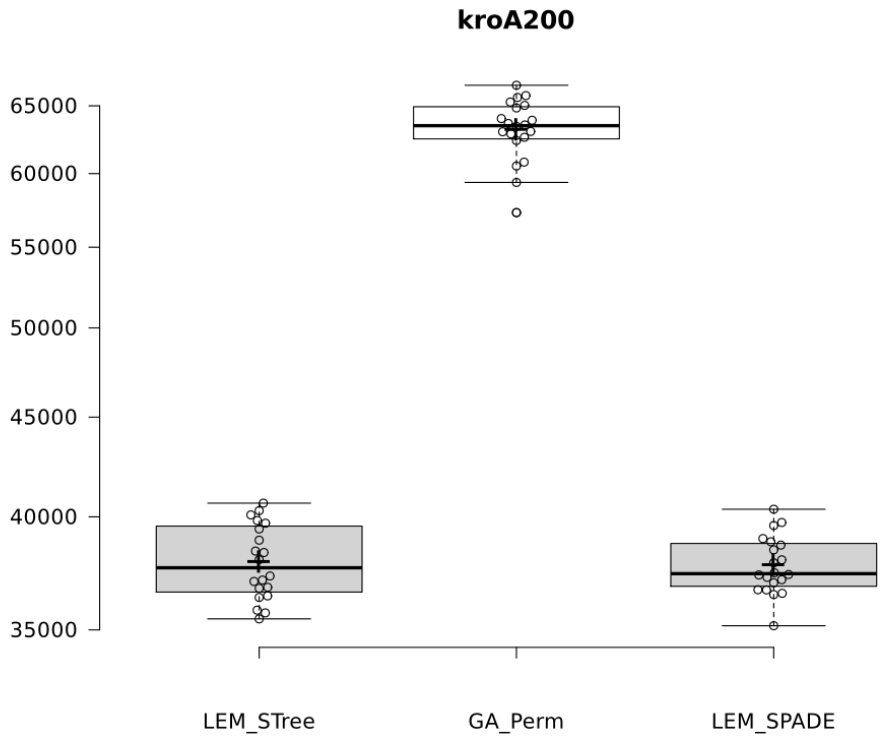
Slika 5.7 Boxplot pronađenih rezultata u 20 testova svakog postupka (eil101)

Iz slike Slika 5.6 može se zaključiti da oba implementirana postupka zadržavaju visoku razinu konvergencije dulje no što to uspijeva običan genetski algoritam što na kraju rezultira boljim rješenjima. Slika 5.7 pokazuje kako je u ovom slučaju postupak pronalaska uzoraka sufiksnim stablom nešto bolji od postupka pronalaska uzoraka SPADE-om, a oba postupka daju bolje rezultate od genetskog algoritma u gotovo svim slučajevima.

kroA200



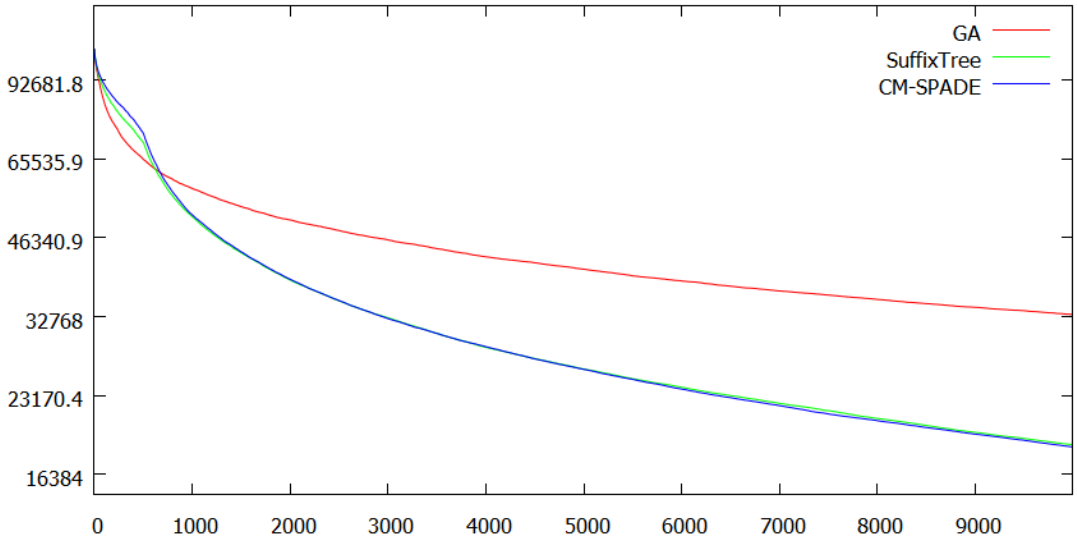
Slika 5.8 Kretanje prosjeka funkcije cilja kroz iteracije za sve postupke (kroA200)



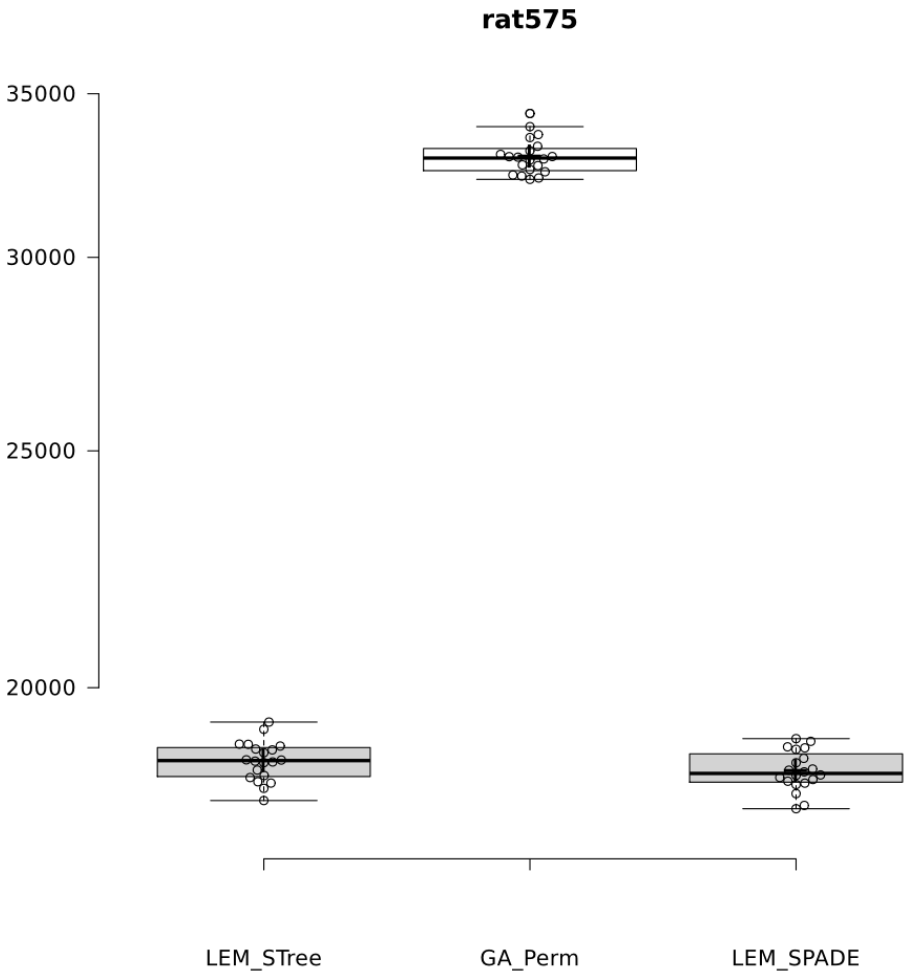
Slika 5.9 Boxplot pronađenih rezultata u 20 testova svakog postupka (kroA200)

Slika 5.8 opet pokazuje nešto lošiju konvergenciju LEM-a na samom početku s obzirom na genetski algoritam, no dosta bolju kako postupak optimizacije napreduje što rezultira puno boljim rješenjima. Na slici Slika 5.9 može se uočiti da se krajnji rezultati dva postupka pronalaska uzoraka u LEM-u ne razlikuju značajnije, s tim da SPADE inačica ima mrvicu bolji medijan rješenja.

rat575



Slika 5.10 Kretanje prosjeka funkcije cilja kroz iteracije za sve postupke (rat575)



Slika 5.11 Boxplot pronađenih rezultata u 20 testova svakog postupka (rat575)

Slika 5.10 i Slika 5.11 pokazuju sličnu situaciju kao i u slučaju testa *korA200* samo što je razlika između obje varijante LEM-a i genetskog algoritma još izraženija. I u ovom slučaju SPADE inačica ima mrvicu bolji medijan rješenja, no ne previše značajan.

6. Primjena na prikaz rješenja stablom

Genetsko programiranje je posebna grana genetskih algoritama koja se bavi evolucijom čitavih programa s obzirom na neki kriterij optimalnosti. Jedan od načina prikaza programa je koristeći stablastu strukturu koja u svojim čvorovima sadržava funkcije ili naredbe, a u listovima varijable (njihove trenutne vrijednosti) i konstante. Implementacija programa kao stabla je ostvarena kroz razrede *TreeGASolution* i *PrimitiveSet*. Razred *PrimitiveSet* je statički spremnik svih vrsta unutarnjih čvorova (funkcija) i listova koji se koriste u izgradnji stabla za određeni problem i zbog toga se ovaj razred treba inicijalizirati željenim primitivima prije svakog rješavanja problema. Potrebno je da primitivi (modelirani sučeljem *IPrimitive<T>*) koji se dodaju u *PrimitiveSet* budu istog tipa, odnosno da im je povratna vrijednost istog tipa. *TreeGASolution* razred služi za pohranu stabla ostvarenog rekurzivnom strukturom razreda *Node* koja sadržava primitiv promatranog čvora, polje referenci na sljedbenike (također razreda *Node*) i referencu na prethodnika. Inicijalizacija jednog objekta *TreeGASolution* se ravna po postavljenim parametrima maksimalne dubine (*maxDepth*) i maksimalnog broja unutarnjih čvorova (*maxNodes*) te se može izvesti na dva načina:

- ***full*** : stablo se izgrađuje potpuno koristeći pretraživanje u širinu (*engl. Breadth First Search*) dodajući nove čvorove i nasumično im dodjeljujući funkcijske primitive dok se ne dostigne jedan od *maxDepth* ili *maxNodes* kriterija. Nakon što se dosegne jedan zaustavni kriterij preostalim čvorovima koji čekaju na proširenje dodaju se listovi kojima se nasumično dodjeljuje neki primitiv lista.
- ***grow***: koristeći pretraživanje u širinu za svaki se novi čvor koji se dodaje u stablo odlučuje hoće li biti unutarnji čvor ili list nasumičnim biranje nekog primitiva iz skupa svih primitiva. Kad je jedan od kriterija zaustavljanja rasta stabla ostvaren preostalim se čvorovima koji čekaju na proširenje dodaju listovi kojima nasumično dodjeljuje neki primitiv lista.

Za operatore mutacije i križanja u okruženju za ispitivanje koriste se :

ReplaceSubtreeMutation: odabire neki unutarnji čvor u stablu te njega i stablo kojemu je taj čvor korijen zamjenjuje nasumično generiranim stablom.

ReplaceTreeNodePrimitiveMutation: odabire neki unutarnji čvor u stablu te mu mijenja dodijeljeni primitiv s nasumično odabranim primitivom koji zahtijeva isti broj operatora kao i mijenjani primitiv.

SwapSubtreesCrossover: križanje zahtijeva dva roditelja. Potomci nastaju tako da se roditelji kloniraju, svakom roditelju se nasumično odabere jedan unutarnji čvor (jedno podstablo) i ta podstabla se međusobno zamijene. Ako nastali potomak ne zadovoljava *maxDepth* i *maxNodes* kriterije ono se ne vraća iz operatora križanja.

Glavna zamisao instanciranja novih stabala učenjem jest iz danog skupa dobrih rješenja nekim algoritmom dubinske analize grafova (*engl. graph mining*) pronaći često ponavljajuće strukture i te pronađene strukture iskoristiti pri stvaranju novih jedinki. Strukture koje algoritmi dubinske analize grafova pronalaze su nepotpune zato što unutarnji čvorovi nemaju sve sljedbenike postavljene i ne završavaju u listovima. Jedan od mogućih načina izgradnje novih rješenja je modifikacijom *grow* tehnike izgradnje stabla tako da umjesto dodavanja jednog čvora u stablo dodaje čitavu strukturu koja zadovoljava uvjete dodavanja - da *maxDepth* i *maxNodes* ne budu premašeni dodavanjem.

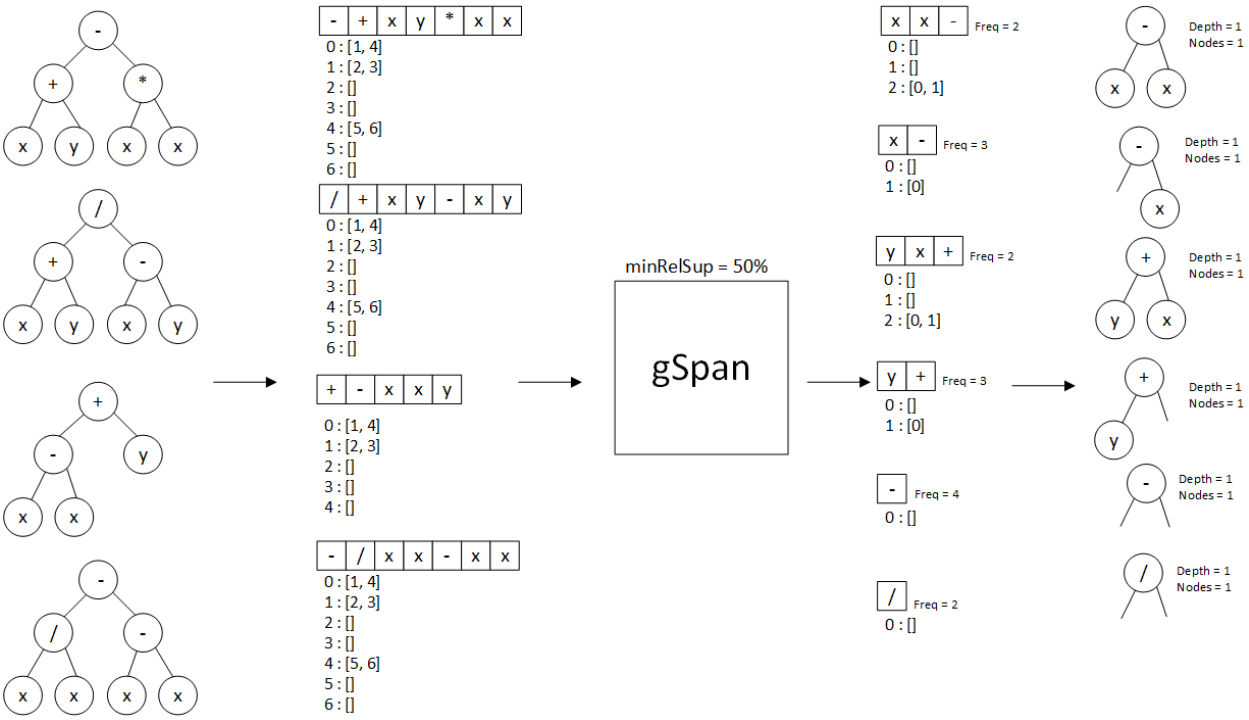
6.1 Pronalazak uzoraka algoritmom gSpan

Prvi način pronalaska uzoraka ostvaren je korištenjem algoritma gSpan (*graph-based substructure pattern mining*) [16] koji je implementiran u *ParSeMiS* Java biblioteci za dubinsku analizu grafova [17]. Algoritam *gSpan* ne uzima u obzir redoslijed ulaznih i izlaznih bridova iz čvorova u grafu što znači da pronađeni uzorci na izlazu algoritma ne moraju biti poredani hijerarhijski kao što su to ulazni podaci. Zbog toga se prije obrade izlaznim podacima treba odrediti korijen stabla i pretvoriti ih natrag u hijerarhijsku strukturu.

6.1.1 Opis pronalaženja uzoraka

Algoritam *gSpan* zahtijeva prikaz grafova strukturom koja u sebi sadržava polje čvorova i polje listi indeksa. Zbog toga je potrebno napraviti konverziju između načina prikaza stabla u LEM-u (početak poglavlja 6) i zahtijevanog načina prikaza grafa. Konverzija se postiže jednostavnim obilaskom početne strukture

pretraživanjem u dubinu (engl. *depth first search*) i dodavanjem oznaka primitiva te indeksa izlaznih bridova u za to predviđene strukture. Jednom kad je konverzija HPT dijela ulaznih podataka obavljena, pokreće se *gSpan* algoritam nad tim podacima i na svom izlazu generira podgrafove čija je relativna frekvencija u skupu podataka veća ili jednaka *minRelSup* odnosno koji se u HPT dijelu populacije pojavljuju u više ili jednako od *minRelSup* * *HPT_size* početnih rješenja. Dobivene je podgrafove onda potrebno transformirati u prikaz podataka sličan *TreeGASolution*-u opisanom na početku poglavlja koji prikazuje podstablo preko *Node* razreda i opisuje neka njegova svojstva poput dubine podstabla i broja čvorova podstabla. Ta svojstva se onda iskorištavaju prilikom stvaranja novih stabala iz pronađenih podstabala. Čitav postupak je ilustriran na slici Slika 6.1. U zadnjem dijelu postupka opis dubine uzorka i broja čvorova u uzorku uzima samo u obzir unutarnje (funkcijske) čvorove.



Slika 6.1 Vizualizacija postupka pronalaska uzoraka u skupu grafova

Pronalazak uzoraka i stvaranje novih jedinki je implementirano u komponenti *GSpanIndividualInstantiation* koja kao parametre pri inicijalizaciji prima *HPT*, *LPT* i *minRelSupport*. Vremenska asimptotska složenost postupka učenja je $O(kFS + rF)$ gdje je *k* maksimalni broj izomorfizama koji postoje između čestog uzorka i grafa u skupu grafova, *F* je broj čestih podgrafova, *S* je veličina skupa grafova, a *r*

maksimalni broj kodova duplikata čestog podgrafa koji izrastaju iz drugih minimalnih kodova.

Slično kao i kod analize sekvencijalnih podataka u poglavlju 5.2.1 dolazi se do problema zahtjevnosti postupka. U ovom se slučaju se to pokušava kompenzirati dinamičkim mijenjanjem parametra *minRelSupport* kako je to opisano u tom poglavlju.

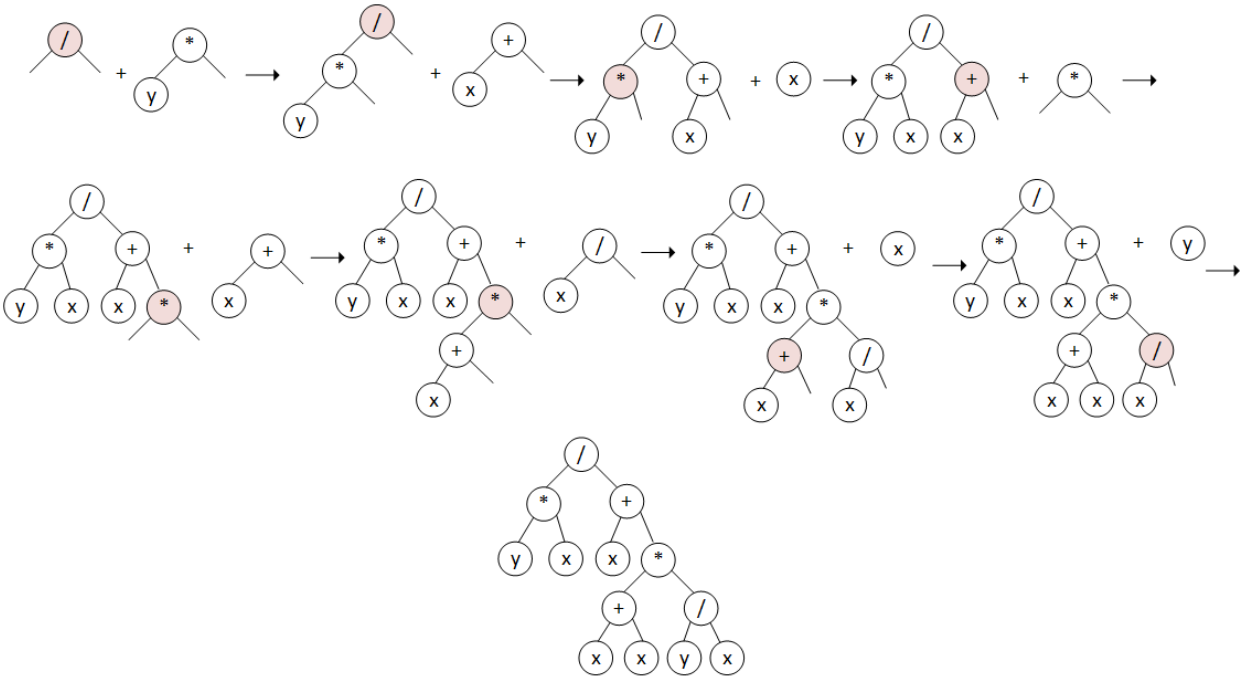
6.1.2 Opis stvaranja novih jedinki

Jednom kad su iz podataka izvučeni česti uzorci kreće se sa stvaranjem novih jedinki koje zadovoljavaju kriterije maksimalne dubine i maksimalnog broja unutarnjih čvorova u stablu. Naknadno se u listu čestih uzoraka dodaju i sve moguće vrijednosti listova. Postupak se izvodi tako da se prvo odabire jedan od uzoraka kojim će započeti izgradnja. Taj uzorak (i svi uzorci u postupku) se bira proporcionalno dobroti koja se određuje izrazom (6.1). U tom izrazu *freq* je frekvencija uzorka, *maxFreq* maksimalna frekvencija gledajući sve uzorke, *rem_nodes* broj mogućih čvorova koji se mogu dodati u nekom trenutku u rješenje u izgradnji, *pattern_nodes* broj čvorova koje ima uzorak, *rem_depth* preostala dubina iz mjesta na koje se dodaje novi uzorak, *pattern_depth* dubina uzorka.

$$score = \frac{\log(1 + freq)}{\log(1 + \max Freq) * (1 + rem\_nodes - pattern\_nodes) * (1 + rem\_depth - pattern\_depth)} \quad (6.1)$$

Ovakvom formulom dobiva se dobrota $\in [0, 1]$ koja favorizira uzorke kojima je dubina slična dubini preostaloj u rješenju u izgradnji, broj unutarnjih čvorova sličan preostalom broju čvorova u rješenju u izgradnji te koja se pojavljuju često (no ipak se ne dopušta linearna ovisnost o tome da taj atribut ne postane previše izražen). U slučaju da je broj uzoraka prevelik (npr. 1000) proporcionalan odabir s ovako definiranom dobrotom nema smisla jer zbog brojnosti uzoraka i dobrotom definiranom u rasponu $[0, 1]$ bolji uzorci imaju tek malo veću vjerojatnost odabira nad slabijim uzorcima. Zbog toga se dodijeljena dobrota potencira s logaritmom broja pronađenih uzoraka kako bi se naglasila razlika između boljih i lošijih uzoraka. Uzorci koje je nemoguće dodati u stablo zbog toga što ne zadovoljavaju kriterije dubine ili broja čvorova ne razmatraju se za proširenje.

Nakon odabira početnog uzorka djeca korijena uzorka se nasumično izmiješaju zbog toga što uzorci pronađeni *gSpan*-om ionako ne zadržavaju redoslijed bridova i počinje izgradnja rješenja obilaskom trenutno izgrađenog rješenja pretraživanjem u širinu (*BFS*) do tada izgrađenog stabla. Nailaskom na čvor u stablu u izgradnji koji nema sve sljedbenike postavljene uzima se iz skupa dopuštenih uzoraka proporcionalno dobroti novi uzorak i on se ugrađuje u stablo. Postupak stvaranja završava kad više nije moguće dodati nove uzorke, tj. kad su jedino što je preostalo u stablu u izgradnji listovi ili kad se došlo do maksimalne dubine ili maksimalnog broja čvorova. U potonjem slučaju preostalim se čvorovima dodaju nasumični listovi. Ovaj je postupak ilustriran na slici Slika 6.2.



Slika 6.2 Prikaz postupka izgradnje stabla iz uzoraka

6.2 Pronalazak uzoraka prilagođenim algoritmom Apriori

Kako algoritam *gSpan* nije u potpunosti prilagođen dubinskoj analizi podataka strukturiranih u obliku stabla (potpuno acikličkih grafova s jasnom hijerarhijskim uređenjem) moguće je postojanje algoritma koji je prilagođeniji takvom zadatku. Dodatno, taj algoritam bi trebao, za razliku od *gSpana*, uzeti u obzir i položaj sljedbenika (čvorova djece) u odnosu na prethodnika (čvor roditelja) što može dodatno smanjiti prostor pretraživanja uzoraka.

Algoritam koji obavlja dubinsku analizu stabala napisan je koristeći *Apriori* svojstvo [18] čestih uzoraka: ako je neki uzorak čest, onda su česti i svi poduzorci koji mogu nastati iz tog uzorka. Implementacijski je zanimljiv obrat te tvrdnje: ako neki uzorak nije čest, onda nisu česti ni naduzorci koji sadržavaju taj uzorak. To se svojstvo koristi kako bi se efektivno smanjio prostor pretraživanja uzoraka.

6.2.1 Opis pronalaženja uzoraka

U nastavku je napisan pseudokod osmišljenog algoritma koji pronalazi česte uzorke u skupu stabala.

```
1 mine_trees(stabla, minfreq, maxdepth)
2       početni_uzorci = stvori uzorke dubine 1 iz stabala
3 prethodni_uzorci = x : početni uzorci t.d. freq(x) >= minfreq
4 sljedeći_uzorci = []
5 pronađeni_uzorci = [] + prethodni_uzorci
6 dubina = 1
7 dok (dubina <= maxdepth & prethodni_uzorci ≠ [])
8       za uzorak : prethodni_uzorci
9             kandidati = generiraj_za_ekspanziju(uzorak, dubina,
                                                  stabla)
10            jed_proš = nađi_jednostavne_ekspanzije(kandidati,
                  stabla, minfreq)
11            sva_proš = nađi_presjeke(jed_proš, minfreq)
12            za proš : sva_proš
13                  novi_uzorak = uzorak + proš
14                        sljedeći_uzorci += novi_uzorak
15                  pronađeni_uzorci += (novi_uzorak,
                                         freq(novi_uzorak))
16      zamijeni(prethodni_uzorci, novi_uzorci)
17      novi_uzorci = []
18      dubina++
20 vrati pronađeni_uzorci
```

Pseudokod algoritma za pronalaženje uzoraka u stablima

Algoritam kao parametre prima skup stabala nad kojim se pronalaze uzorci, minimalnu frekvenciju uzorka i maksimalnu dubinu pronađenih uzoraka. Početno se stvaraju uzorci veličine 1 iz skupa stabala: te uzorke čine samo oni koji se sastoje od jednog čvora (ili lista) u stablu. Uz sve uzorke se pohranjuju i mjesta u

stablima gdje se ti uzorci pojavljuju (indeks stabla i indeks početnog čvora korijena uzorka), pa ni ovi nisu iznimka. Glavna ideja algoritma je postepeno proširivanje uzoraka pretraživanjem skupa stabala u širinu (*BFS*) krenuvši od početnog skupa uzoraka razinu po razinu. To je ostvareno u petlji koja počinje na liniji 7 u kojoj se pronalaze svi uzorci koji nastaju proširenjem uzoraka pronađenih u prethodnoj razini. U liniji 9 se za svaki uzorak koji je bio pronađen u prethodnoj razini pronalazi skup završnih čvorova, tzv. kandidata, iz kojih će se obaviti proširenje uzorka na iduću razinu. Sva moguća proširenja se pronalaze tako da se najprije pronalaze sva jednostavna proširenja (linija 10) koja za svakog kandidata opisuju neko proširenje jednim čvorom iz nove razine, a nakon toga se računaju svi mogući presjeci jednostavnih proširenja kako bi se uzeli u obzir uzorci nove razine koji nastaju proširenjem više kandidata istodobno (linija 11). Tijekom obavljanja jednostavnih proširenja i proširenja nastalih presjecima svaki se uzorak povezuje s listom pozicija (indeks grafa i indeks čvora u grafu) na kojima se taj uzorak ili proširenje nalazi čime se olakšava pronalazak novih proširenja i presjeka.

Na kraju se sva proširenja pretvaraju u uzorke koji se dodaju u listu pronađenih uzoraka zajedno s njihovom frekvencijom. Na kraju vanjske petlje mijenjaju se skupovi prethodnih i sljedećih uzoraka, te se skup sljedećih uzoraka briše i dubina povećava za 1. Postupak završava kad skup prethodnih uzoraka postane prazan ili kad je pređena najveća veličina uzoraka određena s *maxdepth* koju se želi pronaći.

Procijenjena asimptotska vremenska složenost ovog postupka je $O(P * 2^N * M)$ gdje je P broj pronađenih uzoraka, M broj ulaznih grafova, a N prosječni broj čvorova u grafu. Član 2^N nastaje zbog provjere svih presjeka u nekom proširenju gdje je broj jednostavnih proširenja proporcionalan s N i to mora biti provjereno u najgorem slučaju kad svi presjeci zadovoljavaju *minfreq* kriterij. Pronalazak uzoraka i stvaranje novih jedinki je implementirano u razredu *CustomTreeMiningAlgIndividualInstatiation* koji u konstruktoru prima parametre *HPT, LPT, minRelSupport* (minimalna frekvencija izražena relativno na veličinu skupa podataka koji se rudari) i *maxDepth* (maksimalna dubina koju imaju pronađeni uzorci).

Problemi s memorijom su izraženi i kod ovog načina pronalaska čestih podgrafova pa je implementirano dinamičko mijenjanje parametra *minRelSupport* kao i u slučaju pronalaska čestih podgrafova algoritmom *gSpan* (poglavlje 6.1.1).

6.2.2 Opis stvaranja novih jedinki

Stvaranje novih rješenja na temelju uzoraka je implementirano na identičan način kako je to opisano u poglavlju 6.1.2 uz jednu bitnu razliku, a to je da sljedbenici nekog čvora u uzorcima dobivenima gore opisanim algoritmom pronalaska uzoraka imaju i oznaku koji su sljedbenik po redu svog prethodnika, tako da miješanje sljedbenika nekog čvora prilikom obrade tog čvora ne dolazi u obzir.

6.3 Testiranje i rezultati

Testiranje opisanih postupaka u LEM-u se izvodi na dva problema: prvi je pronalazak programa koji će voditi mrava po svijetu u kojem se nalaze tragovi hrane tako da pokupi što je više moguće hrane u što manje koraka. Drugi problem je simbolička regresija, odnosno pronalaženje najboljeg opisa funkcije iz zadanih empirijskih podataka.

6.3.1 Opis testova

Prvi problem je takozvani *Santa Fe Trail problem* [19] koji se koristi pri uspoređivanju različitih inačica genetskog programiranja. U ovoj implementaciji mozak mrava kojemu je u cilju pojesti što više hrane u svijetu predstavljenim 2D rešetkom se modelira koristeći program u obliku stabla kojemu su unutarnji čvorovi funkcije koje modeliraju blokove ili odluke, a listovi konkretne naredbe. Mrav se kreće u diskretnim trenucima i u svakom trenutku može vidjeti što se nalazi točno ispred njega, okrenuti se lijevo, okrenuti se desno ili krenuti naprijed. Mrav sakuplja hranu dolaskom na polje gdje se nalazi hrana. Skup listova i unutarnjih čvorova čine:

- ***Move***: akcija i list. Izvršavanjem ovog mrav obavlja kretanje unaprijed.
- ***Left***: akcija i list. Izvršavanjem ovog mrav obavlja okret ulijevo.
- ***Right***: akcija i list. Izvršavanjem ovog mrav obavlja okret udesno.

- ***IfFoodAhead***: funkcija i unutarnji čvor s dva sljedbenika. Ako se ispred mrava nalazi hrana onda se izvršava prvi sljedbenik, inače se izvršava drugi.
- ***Pr2***: funkcija i unutarnji čvor s dva sljedbenika. Izvršava svoje sljedbenike po redu: najprije prvog, a zatim drugog.
- ***Pr3***: funkcija i unutarnji čvor s tri sljedbenika. Izvršava svoje sljedbenike po redu: prvog, drugog pa trećeg.

Dobrota rješenja jednaka je broju pokupljenih jedinica hrane, a jedinka se evaluira uz pomoć razreda *AntTrailEvaluator* koji dopušta maksimalno 300 izvršenih akcija. Maksimalna dubina jedinki je postavljena na 5, a maksimalni broj unutarnjih čvorova na 50. Vjerojatnost mutacije za ovaj problem je postavljena na 0.5.

Drugi problem jest problem simboličke regresije [20] u kojemu se zadani skup ulaza i izlaza pokušava pronaći izraz funkcije koji je za dane ulaze generirao zadane izlaze po kriteriju minimizacije funkcije srednje kvadratne pogreške (*engl. Mean Squared Error*). Optimizacija se provodi nad skupovima podataka generiranih od strane funkcija opisanih u tablici 6.1, gdje svaka funkcija uzorkovana uniformno 100 puta po svakoj dimenziji.

6.1 Popis funkcija koje se pokušavaju pronaći simboličkom regresijom

Symb1	$\log(x+1)+\log(x^2+1)$	x ∈ [-10, 10]
Symb4	$x \cdot y \cdot \sin((x+1)\cdot(y-1))$	x ∈ [-10, 10], y ∈ [-10, 10]
Symb6	$\frac{x^3}{5}+\frac{y^3}{2}-x-y$	x ∈ [-10, 10], y ∈ [-10, 10]

Prilikom izgradnje stabla postupak optimizacije ima na raspolaganju sljedeće primitive:

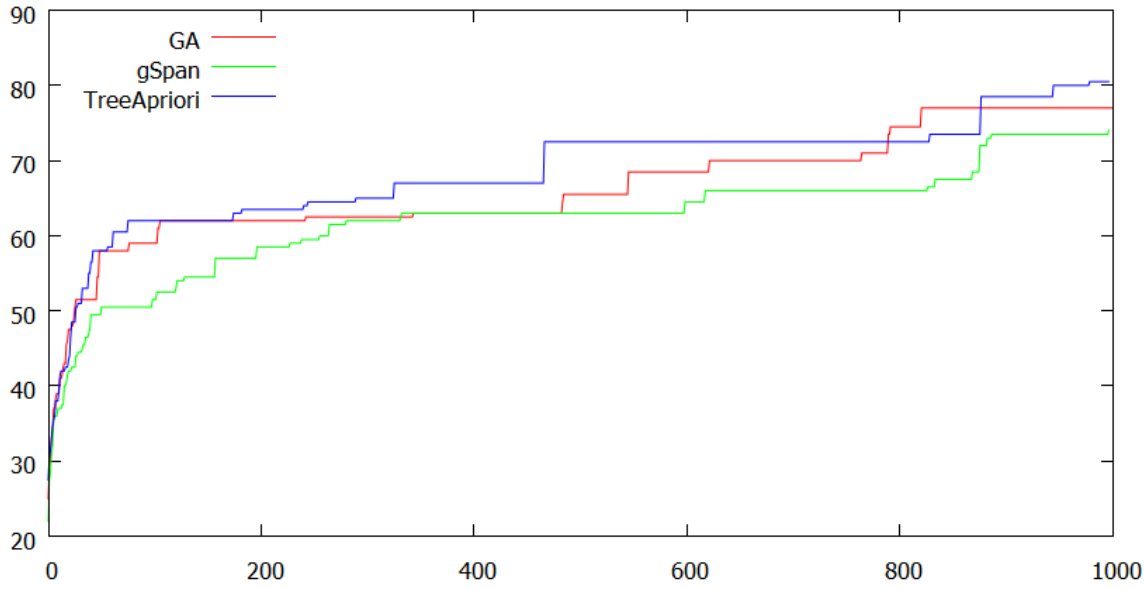
- ***Add***: funkcija koja ima dva sljedbenika i kao rezultat vraća zbroj rezultata izvođenja svojih sljedbenika.
- ***Cos***: funkcija ima jednog sljedbenika i kao rezultat vraća kosinus rezultata izvođenja tog sljedbenika.

- ***Div***: funkcija ima dva sljedbenika i kao rezultat vraća količnik rezultata izvođenja svojih sljedbenika gdje se rezultat prvog sljedbenika uzima kao djeljenik, a rezultat drugog kao djelitelj. Ako je djelitelj 0 funkcija vraća 1.0.
- ***Mul***: funkcija ima dva sljedbenika i kao rezultat vraća umnožak rezultata izvođenja svojih sljedbenika.
- ***Neg***: funkcija ima jednog sljedbenika i kao rezultat vraća negiran rezultat izvođenja tog sljedbenika.
- ***Sin***: funkcija ima jednog sljedbenika i kao rezultat vraća sinus rezultata izvođenja tog sljedbenika.
- ***Sub***: funkcija koja ima dva sljedbenika i kao rezultat vraća razliku rezultata izvođenja svojih sljedbenika. Rezultat izvođenja prvog sljedbenika je umanjenik, dok je rezultat izvođenja drugog umanjitelj.
- ***Var***: list koji pri stvaranju dobiva oznaku varijable (npr. *'x', 'y'*) i tablicu iz koje može iščitati vrijednost te varijable. Kao rezultat izvođenja vraća trenutnu vrijednost varijable.
- ***ConstantOne***: list koji kao rezultat vraća realnu konstantu 1.0.

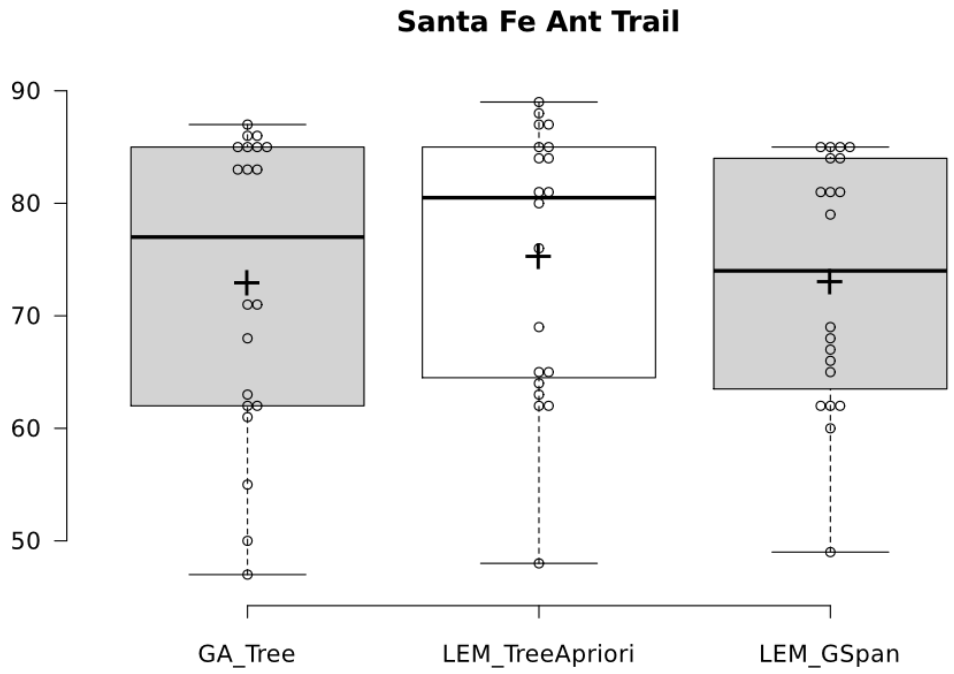
Kazna za neko rješenje je definirana kao srednja kvadratna pogreška funkcije nad zadanim podacima. Vjerojatnost mutacije za ovaj problem je postavljena na 0.35. Maksimalni broj evaluacija funkcija cilja u oba slučaja je postavljen na $500\ 000$. Praksa je u problemima genetskog programiranja postaviti veličinu populacije nešto većom nego u običnom genetskom algoritmu, pa je veličina populacije za sve slučajeve postavljena na 500. Algoritmu gSpan je početni *minRelSupport* postavljen na 0.5, dok je prilagođenom Apriori algoritmu taj parametar postavljen na početnu vrijednost od 0.2. Oba načina koriste *StagnationAvoid-ActionProfilingFunction* (opisan u poglavlju 2.4) s postavljenim parametrima *interval* na 20, *increment* na 50, *stagnationThreshold* na 10^{-5} i *okThreshold* na 10^{-4}. Kako je zbog povećanja veličine populacije smanjen broj generacija promijenjeni su i parametri koji utječu na donošenje odluke na kraju svake iteracije LEM-a: *learnProbe* je postavljen na 200, *mutationProbe* na 2, a *startOverProbe* na 1. Dva prikazana načina stvaranja novih jedinki uspoređena su s genetskim algoritmom koji parametre koje dijeli s LEM-om ima postavljene kao što ih ima i LEM.

6.3.2 Rezultati

AntTrail



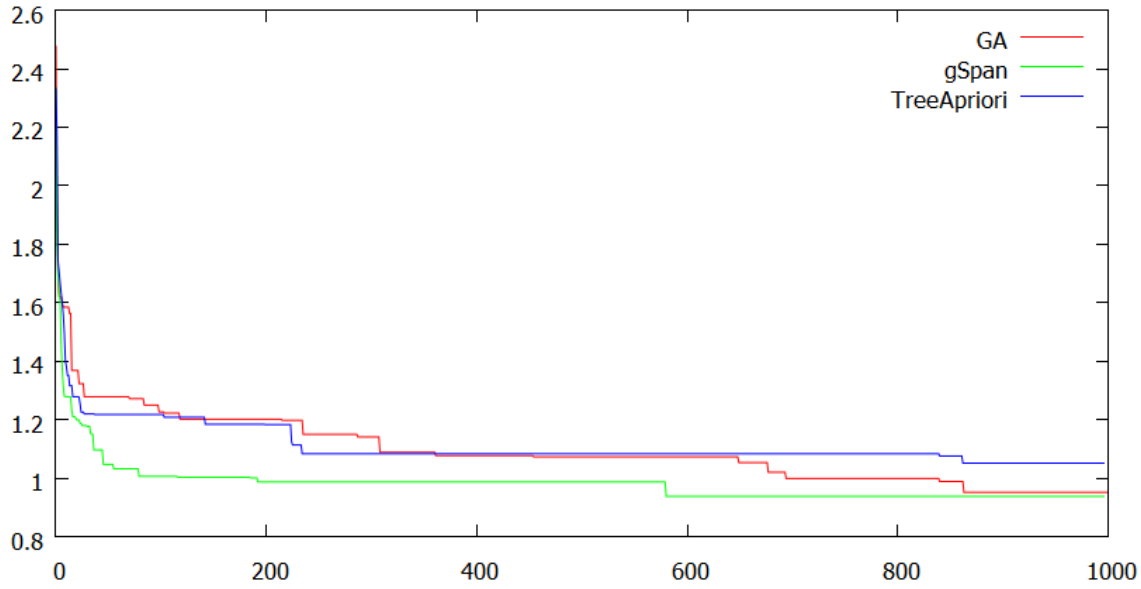
Slika 6.3 Kretanje medijana funkcije cilja kroz iteracije za sve postupke (AntTrail)



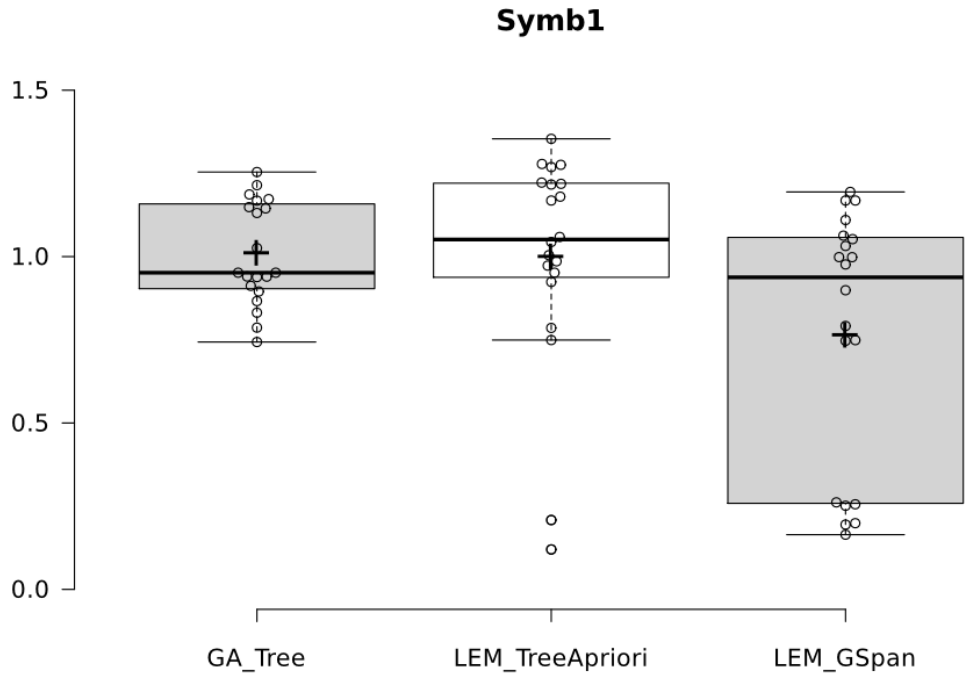
Slika 6.4 Boxplot pronađenih rezultata u 20 testova svakog postupka (AntTrail)

U ovom problemu maksimizacije sva 3 algoritma se ponašaju slično, s tim da algoritam koji uzorke pronalazi prilagođenim Apriorijem (TreeApriori) pokazuje nešto bržu konvergenciju na slici Slika 6.3 i bolji medijan rezultata na slici Slika 6.4 od genetskog algoritma. TreeApriori je ujedno jedini pronašao i najbolje rješenje.

Symb1



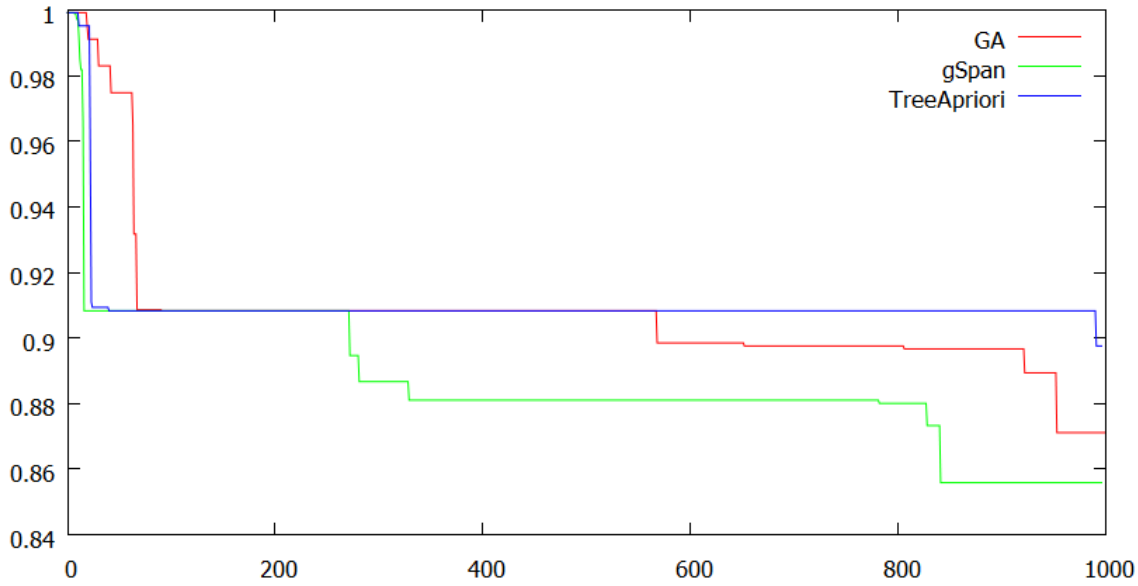
Slika 6.5 Kretanje medijana funkcije cilja kroz iteracije za sve postupke (Symb1)



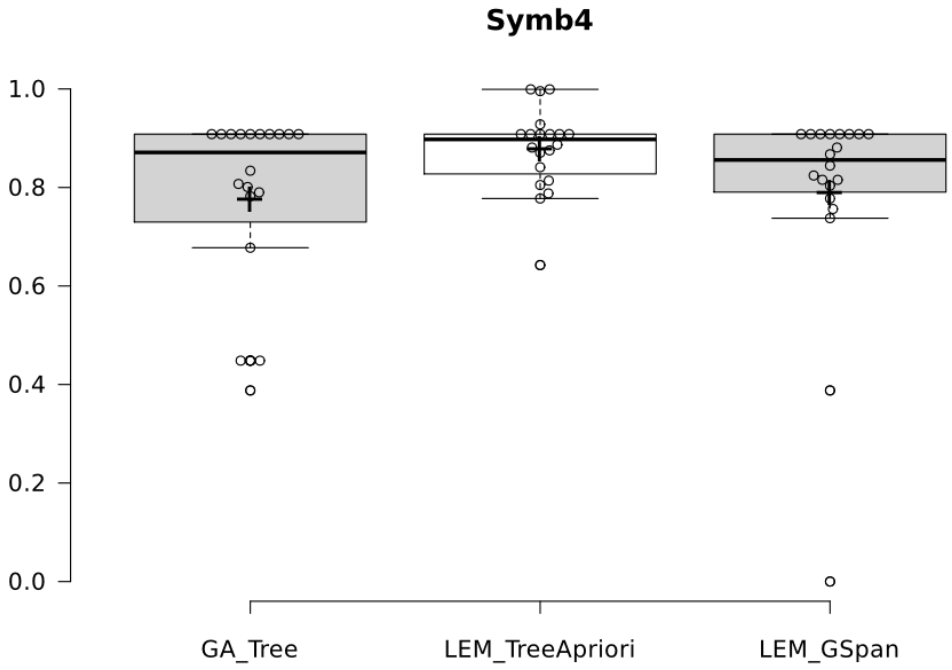
Slika 6.6 Boxplot pronađenih rezultata u 20 testova svakog postupka (Symb1)

Sa slike Slika 6.5 je jasno vidljivo da varijanta LEM-a s gSpan algoritmom ima najbržu konvergenciju i završava s najboljim rezultatom. To je potvrđeno na slici Slika 6.6 iz koje se može zaključiti kako gSpan varijanta LEM-a i genetski algoritam imaju slične medijane, ali prosjek dobrote rješenja je ipak na strani gSpan-a.

Symb4



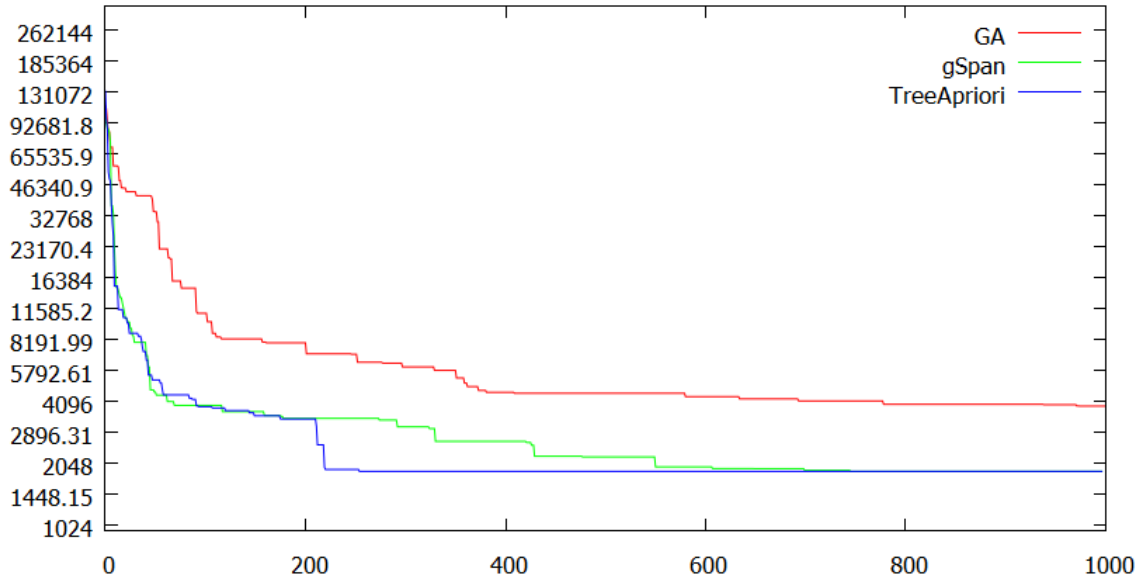
Slika 6.7 Kretanje medijana funkcije cilja kroz iteracije za sve postupke (Symb4)



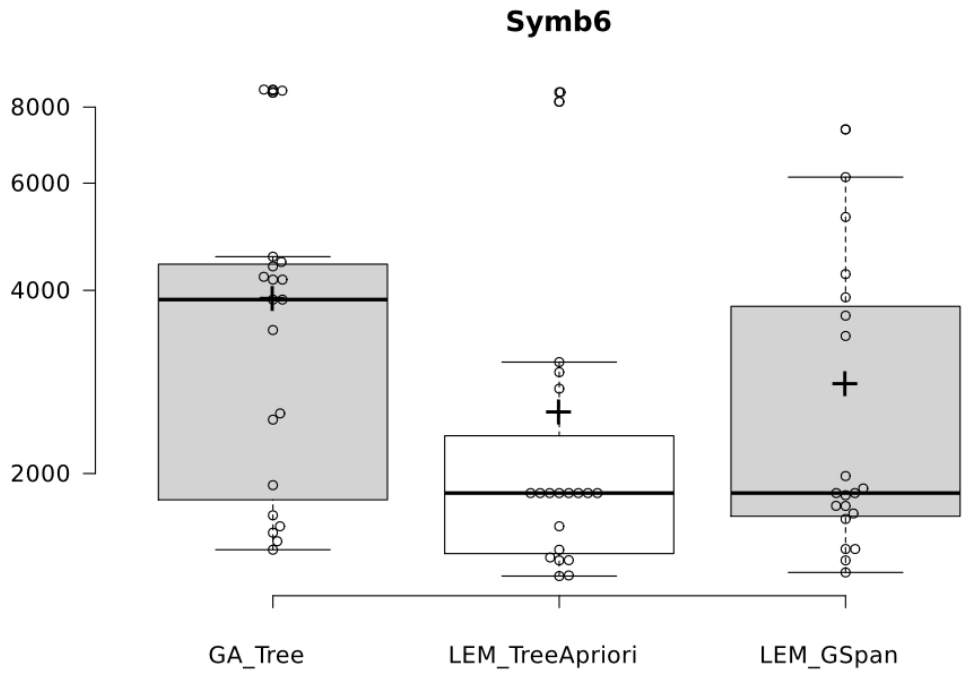
Slika 6.8 Boxplot pronađenih rezultata u 20 testova svakog postupka (Symb4)

Na slici Slika 6.7 je vidljivo kako je najbolju konvergenciju ostvarila varijanta LEM-a s gSpan-om čiji se medijan rješenja po slici Slika 6.8 tek neznatno razlikuje od medijana rješenja genetskog algoritma. Po istoj slici je vidljivo kako je prosjek pronađenih rješenja genetskim algoritmom bolji od onog pronađenog gSpan varijantom LEM-a.

Symb6



Slika 6.9 Kretanje medijana funkcije cilja kroz iteracije za sve postupke (Symb6)



Slika 6.10 Boxplot pronađenih rezultata u 20 testova svakog postupka (Symb6)

Iz slika Slika 6.9 i Slika 6.10 jasno je vidljivo kako se obje varijante LEM-a ponašaju bolje od običnog genetskog algoritma, s time da TreeApriori varijanta ima bolji prosjek dobrote pronađenih rješenja od gSpan varijante, iako su im medijani dobrote pronađenih rješenja isti.

7. Zaključak

U ovom radu prikazane su i ostvarene neke mogućnosti primjene LEM-a nad različitim zapisima rješenja postupka optimizacije, poput polja realnih brojeva, vektora bitova, permutacija i stabala.

Pokazalo se kako LEM koristeći različite osmišljene načine stvaranja jedinki u postupku optimizacije može postići bolje rezultate od genetskog algoritma kojemu su zajednički parametri postavljeni isto kao i LEM-u.

Tako se za prikaz rješenja poljem realnih brojeva pokazalo kako LEM može u slučaju malog broja dimenzija dati značajno bolje rezultate na COCO ispitnom skupu funkcija bez šuma, ali u slučaju većeg broja dimenzija rezultati su, iako bolji, slični baznom genetskom algoritmu. Testiranje na problemima koji kao prikaz rješenja imaju polje bitova je pokazalo da osmišljeni načini stvaranja novih jedinki ne uspijeva nadmašiti brzinu konvergencije niti dobrotu završnog rješenja do kojih dolazi genetski algoritam. Testovi nad simetričnim problemom trgovačkog putnika su pak pokazali kako osmišljeni postupci stvaranja novih jedinki dovode do rješenja koja su bolja od rješenja dobivenih uporabom genetskog algoritma i kako razlika u dobroti dobivenih rješenja između LEM-a i običnog genetskog algoritma raste proporcionalno s veličinom problema. Za probleme koji za prikaz imaju strukturu stabla, testovi su pokazali kako osmišljeni načini stvaranja jedinki rezultiraju rješenjima dobrote slične onim dobivenim genetskim algoritmom, iako imaju nešto bolju dobrotu.

Bez obzira na rezultate, mogućnosti za daljnje istraživanje ovog područja u vidu novih implementacija postupaka stvaranja jedinki na temelju pronađenih uzoraka u podacima su neiscrpne. Kako u ovom radu nisu korišteni podaci o lošim jedinkama pri stvaranju novih rješenja, to preostaje za daljnje istraživanje. Korištenje povijesnih podataka iz više prethodnih generacija je također mogućnost koja se može opširno ispitati, kao što je i utjecaj velikog broja parametara, kojim LEM raspolaže, na brzinu postupka optimizacije.

8. Literatura

[1] Lamarkizam, http://hr.wikipedia.org/wiki/Lamarkizam, 25.05.2015.

[2] Baldwinov efekt, http://en.wikipedia.org/wiki/Baldwin_effect, 25.05.2015.

[3] J. Wojtusiak, R.S. Michalski: The LEM3 Implementation of Learnable Evolution Model and Its Testing on Complex Function Optimization Problems, http://www.mli.gmu.edu/jwojt/papers/06-7.pdf, 25.05.2015.

[4] Čupić M.: Prirodom inspirirani optimizacijski algoritmi. Metaheuristike., 30. prosinca 2013., http://java.zemris.fer.hr/nastava/pioa/knjiga-0.1.2013-12-30.pdf, 25.05.2015.

[5] Evolutionary Computational Framework, 04.11.2014., *Applications of Evolutionary Computation Scheduling / Timetabling / Optimization / Machine learning*, http://gp.zemris.fer.hr/ecf/, 25.05.2015.

[6] Hall M., Frank E., Holmes G., Pfahringer B., Reutemann P., Witten I. H. (2009); The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.

[7] Quinlan, J. R. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993., http://dm.kaist.ac.kr/kse525/resources/papers/Qui93_decisiontree_C4.5.pdf, 25.05.2015.

[8] Schapire R. E. : Explaining AdaBoost, https://www.cs.princeton.edu/~schapire/papers/explaining-adaboost.pdf, 25.05.2015.

[9] Cervone G., Franzese P., Keesee A. P. K.: Algorithm quasi-optimal (AQ) learning, http://cervone.psu.edu/publications/2010_WIRES_AQLearning_Cervone.pdf, 25.05.2015.

[10] Beam search: http://en.wikipedia.org/wiki/Beam_search, 25.05.2015.

[11] COmparing Continuous Optimisers, http://coco.gforge.inria.fr/doku.php, 25.05.2015.

[12] Picek S., Jakobovic D., Golub M.: Evolving Cryptographically Sound Boolean Functions, http://bib.irb.hr/datoteka/640221.p191.pdf, 25.05.2015.

[13] Fournier-Viger, P., Gomariz, Gueniche, T., A., Soltani, A., Wu., C., Tseng, V. S. (2014). SPMF: a Java Open-Source Pattern Mining Library. Journal of Machine Learning Research (JMLR), 15: 3389-3393., http://www.philippe-fournier-viger.com/spmf/, 25.05.2015.

[14] Zaki M. J., Hsiao C.: CHARM: An Efficient Algorithm for Closed Itemset Mining, http://www.philippe-fournier-viger.com/spmf/Charm02.pdf, 25.05.2015.

[15] McLaughlin J., Clark J. A.: Evolving balanced Boolean functions with optimal resistance to algebraic and fast algebraic attacks, maximal algebraic degree, and very high nonlinearity., https://eprint.iacr.org/2013/011.pdf, 25.05.2015.

[16] Yan X., Han J.: gSpan: Graph-Based Substructure Pattern Mining, http://www.cs.ucsb.edu/~xyan/papers/gSpan.pdf, 25.05.2015.

[17] ParSeMiS - the Parallel and Sequential Mining Suite, 31.12.2011., https://www2.informatik.uni-erlangen.de/EN/research/zold/ParSeMiS/index.html, 25.05.2015.

[18] Topics in Database Systems: Mining Frequent Itemsets, https://www.cs.duke.edu/courses/spring02/cps296.1/lectures/20-dm.pdf, 25.05.2015.

[19] Santa Fe Trail problem, http://en.wikipedia.org/wiki/Santa_Fe_Trail_problem, 25.05.2015.

[20] Symbolic regression, http://en.wikipedia.org/wiki/Symbolic_regression, 25.05.2015.

[21] Gusfield D.: Linear-Time Construction of Suffix Trees, 1997., http://web.stanford.edu/~mjkay/gusfield.pdf, 25.05.2015.

[22] TSPLIB, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html, 25.05.2015.

[23] Fournier-Viger F., Gomariz A., Campos M., Thomas R.: Fast Vertical Mining of Sequential Patterns Using Co-occurrence Information, 2014.,

http://www.philippe-fournier-viger.com/spmf/PAKDD2014_sequential_pattern_mining_CM-SPADE_CM-SPAM.pdf, 25.05.2015.

[24] Campos M., Thomas R.: Fast Vertical Mining of Sequential Patterns Using Co-occurrence Information, 2014., http://www.philippe-fournier-viger.com/spmf/PAKDD2014_sequential_pattern_mining_CM-SPADE_CM-SPAM.pdf, 25.05.2015.

[25] Topological sorting, http://en.wikipedia.org/wiki/Topological_sorting, 25.05.2015.

9. Sažetak

Primjena modela evolucijskog učenja na probleme optimizacije

U ovom radu ispitane su mogućnosti implementacije LEM-a (*Learnable Evolution Model*) u svrhu rješavanja problema iz različitih domena korištenjem pravila ili uzoraka koji se pojavljuju u samim podacima. Opisano je testno okružje u kojemu je implementiran modularni algoritam i sve komponente koje su potrebne za rješavanje određenih problema. Implementacija LEM-a nad problemima čije je rješenje prikazano poljem realnih brojeva izvedena je na dva načina: pronalaskom pravila uz pomoć C4.5 poboljšanog AdaBoost-om i pronalaskom pravila AQ algoritmom. Implementacije su testirane nad COCO bibliotekom funkcija.

Problemima čije je rješenje prikazano poljem bitova napisana su dva načina stvaranja novih jedinki postupkom učenja: pronalaženjem pravila uz pomoć C4.5 poboljšanog AdaBoost-om i korištenjem CHARM algoritma pronalaska čestih zatvorenih skupova stavki. Implementacije su testirane nad problemom stvaranja Booleovih funkcija koje moraju čim bolje zadovoljiti određene kriterije.

Postupak učenja nad problemima čije je rješenje prikazano permutacijom ostvareno je koristeći dva načina: pronalaskom čestih grupa vrijednosti koristeći sufiksno stablo i pronalaskom sekvencijalnih uzoraka koristeći CM-SPADE algoritam. Načini su testirani nad podskupom problema iz poznate biblioteke TSPLIB.

Za probleme kojima je rješenje prikazano stablom učenje novih rješenja ostvareno je na dva načina: korištenjem algoritma gSpan i Apriori algoritmom prilagođenim za pronalazak čestih uzoraka u skupu stabala. Testiranje je obavljeno je nad problemom SantaFe traga i problemima simboličke regresije.

Ključne riječi: genetski algoritam, evolucijsko učenje, dubinska analiza podataka, strojno učenje, Learnable Evolution Model, optimizacija

10. Summary

Application of learnable evolution model to optimization problems

This thesis presents some implementation possibilities of the LEM (*Learnable Evolution Model*) algorithm with the purpose of solving problems from different domains using rules or patterns that occur in the data. The testing framework in which the modular algorithm is implemented and all components used for solving particular problems have been described. The implementation of LEM for solving problems which solution is represented by an floating point array has been done in two different ways: by discovering rules using the C4.5 algorithm boosted with AdaBoost and by discovering rules through the AQ algorithm. These implementations have been tasted on the COCO test platform.

For the problems which solution is represented with a vector of bits, two ways of constructing new solutions in the learning phase have been implemented: by discovering rules using the C4.5 algorithm boosted with AdaBoost and by using the CHARM frequent itemset mining algorithm. Those implementations have been tested on the problem of creating a Boolean function which has to satisfy a combination of criteria as good as possible.

The LEM learning phase for problems which solution is represented with a permutation has been realized in two ways: by finding frequent groups of values using a suffix tree and by mining sequential patterns using the CM-SPADE algorithm. These two realizations have been tested on a subset of standard TSP problems from TSPLIB.

For the problems which solution is represented by a tree data structure the LEM learning phase has been achieved in two ways: using the gSpan algorithm and a version of Apriori specially tuned for tree frequent pattern mining. These two methods have been tested on two different problems: the first being Santa Fe trail problem and the second a set of symbolic regression problems.

Key words: genetic algorithm, evolutionary learning, data mining, machine learning, Learnable Evolution Model, optimization