SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1484

Problemi i algoritmi kombinatoričke optimizacije

Marina Krček

Zagreb, lipanj 2017.

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

Zagreb, 3. ožujka 2017.

DIPLOMSKI ZADATAK br. 1484

Pristupnik: **Marina Krček (0036470282)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Problemi i algoritmi kombinatoričke optimizacije**

Opis zadatka:

Opisati vrste postupaka kombinatoričke optimizacije i načine usporedbe učinkovitosti različitih postupaka. Istražiti postojeće skupove ispitnih problema kombinatoričke optimizacije. Ostvariti programski sustav za ispitivanje učinkovitosti stohastičkih algoritama u ovisnosti o parametrima i veličini problema. Ocijeniti učinkovitost različitih algoritama i prikaza rješenja na zadanim ispitnim skupovima te na postojećim problemima iz inženjerske prakse. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 10. ožujka 2017.
Rok za predaju rada: 29. lipnja 2017.

Mentor:

Izv. prof. dr. sc. Domagoj Jakobović

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Siniša Srbljić

Sadržaj

1. Uvod

Combinatorial Black-Box Optimization Competition (CBBOC) je natjecanje koje se održava kao priprema za konferenciju *Genetic and Evolutionary Computation Conference* (GECCO). Održalo se dvije godine za redom – 2015. te 2016. godina. Cilj natjecanja bilo je prikazati karakteristike heurističkih algoritama u primjeni kombinatoričke optimizacije.

Na natjecanju su se pojavili algoritmi koji su postigli vrlo dobre rezultate na općenitim problemima, dakle ne za specifične primjene, nego široku upotrebu. Iz tog razloga cilj je rada bio proučiti probleme korištene na natjecanju te implementirati algoritme pobjednike za daljnju upotrebu unutar okruženja ECF (*Evolutionary Computation Framework*).

U radu su opisani ispitni problemi kombinatoričke optimizacije s natjecanja te su ispitani postojeći algoritmi unutar ECF-a nad tim problemima. Provedena je usporedba s rezultatima algoritama s natjecanja.

Algoritmi pobjednici implementirani su unutar ECF-a kako bi se ispitali nad već postojećim problemima iz prakse.

2. Kombinatorička optimizacija

U primijenjenoj matematici i teoriji računarske znanosti, kombinatorička optimizacija predstavlja područje koje se bavi pronalaženjem optimalnog (najboljeg) rješenja u konačnom skupu mogućih rješenja [1][2]. Primjenjuje se u nekoliko područja, kao što su umjetna inteligencija, strojno učenje, matematici te programskom inženjerstvu.

Primjeri primjene:

- razvoj najbolje zračne mreže ruta i odredišta,
- izbor optimalne rute za dostavu paketa (pisma),
- određivanje koji taxi će preuzeti kojeg klijenta.

Postoji puno algoritama koji se izvode u polinomnom vremenu, a koriste se za određene razrede diskretnih optimizacijskih problema. Takvi problemi kombinatoričke optimizacije su primjerice pronalaženje najkraćeg puta, razapinjućeg stabla (engl. *spanning tree*), protoka u mrežama (engl. *network flow*) te potpuno sparivanje u grafovima.

Međutim, mnogi problemi ipak su presloženi da bi se riješili u razumnom vremenu, optimalno rješenje nije isplativo čak niti ostvarivo tražiti potpunom pretragom („grubom silom“) svih mogućih rješenja. Iz tog razloga, u praksi često se koriste aproksimativne metode.

Aproksimativne metode [3] su algoritmi koji pronalaze 'dobro' rješenje koje nije nužno optimalno, ali se postiže relativno brzo. Takva rješenja su zadovoljavajuća upravo kada je traženje pravog optimuma neizvedivo ili neostvarivo u stvarnom vremenu (ili vremenskom ograničenju). Postoje aproksimacijski i heuristički algoritmi. Aproksimacijski algoritmi daju rješenja dokazive kvalitete i dokazive granice vremena izvođenja, dok heurističke metode ne jamče optimalnost rješenja, ali obično brzo dolaze do dovoljno dobrog rješenja.

S obzirom da se na probleme kombinatoričke optimizacije može gledati kao na pretragu za najboljim objektom u skupu diskretnih objekata, za rješavanje takvih problema mogu se koristiti algoritmi pretraživanja ili metaheuristike. Općeniti algoritmi pretraživanja ne garantiraju pronalazak pravog optimalnog rješenja za svaki problem, niti da će se izvesti u određenom vremenu (razumnom vremenu).

2.1 Problemi kombinatoričke optimizacije

Problemi kombinatoričke optimizacije koriste diskretan prostor (diskretne varijable odluke), prostor je konačan, a funkcija cilja može biti bilo kakva (nelinearna, linearna, neanalitička, itd.) [2].

Mnogi problemi iz stvarnog života pripadaju kategoriji kombinatoričkih problema.

Primjeri najpoznatijih kombinatoričkih problema:

- problem trgovačkog putnika (*Travelling salesman problem* - TSP),
- problem zadovoljivosti (*Satisfiability problem* - SAT),
- problem naprtnjače (*Knapsack problem*),
- bojenje grafa (*Graph coloring* - GC),
- problem Steinerovog stabla (*Steiner tree problem* - STP).

2.1.1 Problem trgovačkog putnika (TSP)

Zadano je n gradova i njihove međusobne udaljenosti. Zadatak je pronaći najkraću rutu koja svaki grad posjećuje jednom i samo jednom (Hamiltonov ciklus), te se vraća u polaznu točku. Postoji $(n-1)!$ mogućih ruta – $(n-1)!/2$ mogućih ruta ako orijentacija nije bitna, što je previše da bi se „grupom silom“ pretraživalo, pa se za pronalaženje rješenja koriste heuristički algoritmi.

2.1.2 Problem zadovoljivosti (SAT)

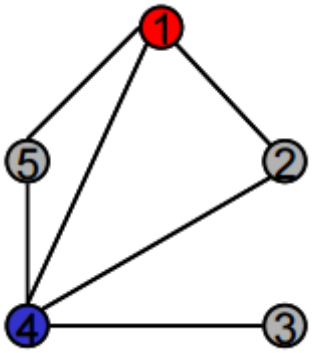
Zadana je *Boolean* funkcija (izraz), a problem zadovoljivosti traži kombinaciju vrijednosti koja bi se dodijelila varijablama kako bi cijeli izraz bio istinit.

2.1.3 Problem naprtnjače

Postoji n predmeta i naprtnjača u koju stane predmeta s ukupno maksimalno C jedinica. Svaki predmet ima vrijednost koji nosi i veličinu koju zauzima u naprtnjači. Cilj problema je pronaći podskup predmeta koji stane u naprtnjaču, a imaju maksimalnu ukupnu vrijednost. Ukratko, maksimizira se vrijednost zbroja predmeta, uz ograničenje veličine naprtnjače.

2.1.4 Bojanje grafa (GC)

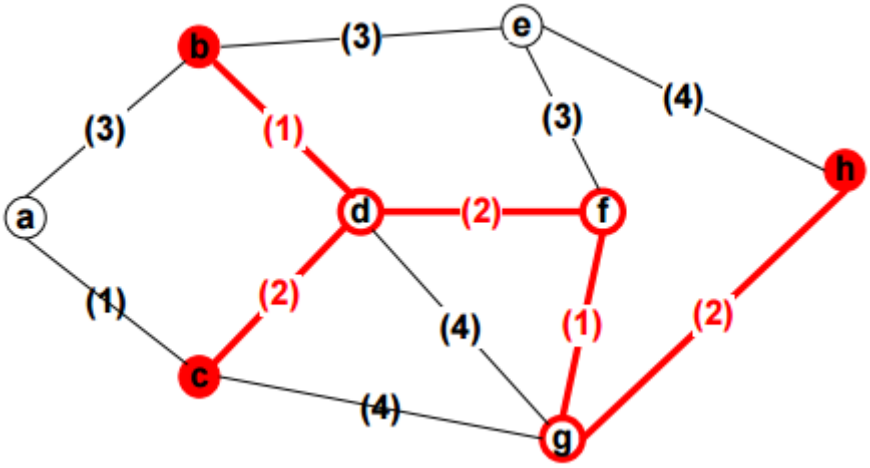
Zadanom grafu obojiti čvorove minimalnim brojem boja tako da nijedan par susjednih čvorova nije obojen istom bojom, što prikazuje slika 2.1.



Slika 2.1: Bojanje grafa

2.1.5 Problem minimalnog Steinerovog stabla

Steinerovo stablo za zadani graf $G = (V, E)$ i podskup čvorova D u V je povezan i acikličan podgraf od G koji obuhvaća sve čvorove u D. Minimalno Steinerovo stablo je takav podgraf minimalne težine u grafu s težinama. Primjer takvog stabla označen je crvenom bojom na slici 2.2.



Slika 2.2: Steinerov graf

2.2 Kombinatorička eksplozija

Konačni prostor pretraživanja implicira da se optimalno rješenje sigurno može naći iscrpnim pretraživanjem (na silu, engl. *brute-force*), no to vodi do problema kombinatoričke eksplozije.

Tablica 2.1 prikazuje vrijeme koje je potrebno za iscrpno pretraživanje svih mogućih kombinacija u problemu trgovačkog putnika. Iz ovih se razloga koriste heuristike koje daju rješenje u razumnom vremenu.

Tablica 2.1: Kombinatorička eksplozija na primjeru TSP problema

Broj gradova (n)	**Broj mogućih rješenja (kombinacija) – n!**	**Vrijeme potrebno da se ispitaju sva rješenja (*brute-force*)**
20	20! ≈ 2.5*10^18	1 sat
21	21! ≈ 5.1*10^19	20 sati
22	22! ≈ 1.2*10^21	17.5 dana
25	25! ≈ 1.5.*10^25	6 stoljeća

3. Skup problema CBBOC

CBBOC je natjecanje koje se održalo u sklopu konferencije *Genetic and Evolutionary Computation Conference* (GECCO) kako bi se prikazale karakteristike heurističkih algoritma korištenih za kombinatoričke probleme gdje je puni naziv natjecanja (engl.) *Combinatorial Black-Box Optimization Competition*. Održalo se 2015. te 2016. godine, no u ovom radu korišteni su podaci s natjecanja iz 2015. jer su tamo u trenutku izrade rada bili podaci i o rezultatima natjecanja – algoritmima pobjednicima te najboljim rješenjima.

Natjecanjem se htjela prikazati detaljna usporedba uspješnosti različitih heurističkih metoda nad kombinatoričkim problemima, gdje su stvarni problemi koji predstavljaju kombinatoričke probleme bili podijeljeni u tri skupine – bez učenja, s kratkim učenjem i s dugim učenjem. Kategorije se razlikuju po broju dozvoljenih evaluacija tijekom učenja (treniranja). Nakon učenja, ispitni problemi imaju dozvoljen broj evaluacija jednak za sve algoritme. Broj evaluacija rješenja tijekom izvođenja algoritma je mjera vremena izvođenja. Ograničavanjem broja evaluacija želi se stvoriti dojam stvarnog problema gdje se optimizacija mora izvršiti u nekom ograničenom i razumnom vremenu.

Na natjecanju su usporedbu algoritama odredili tako da su za svaki primjerak problema rangirali algoritme po postignutom rezultatu (ukoliko su neki algoritmi izjednačeni, drugi kriterij je bio broj evaluacija). Ukupni rezultat algoritma je prosječni rang postignut kroz sve ispitne primjerke, čime se preferiraju algoritmi koji postižu dobre rezultate nad svim problemima.

Problemi na kojima se ispituju algoritmi generirani su pomoću NK modela. Na stranicama natjecanja [4] može se preuzeti njihov generator problema te već generirani problemi na kojima se provodilo natjecanje. Također, na stranicama se nalaze i rezultati natjecanja – dobivena rješenja te algoritmi s najboljim ukupnim rezultatima.

Dva su algoritma implementirana u sklopu izrade ovog rada – CMA-VNS i P3 algoritam. CMA-VNS je bio rangiran na 1. mjestu u sve tri kategorije, dok je P3 bio na 2. mjestu u kategoriji problema bez učenja.

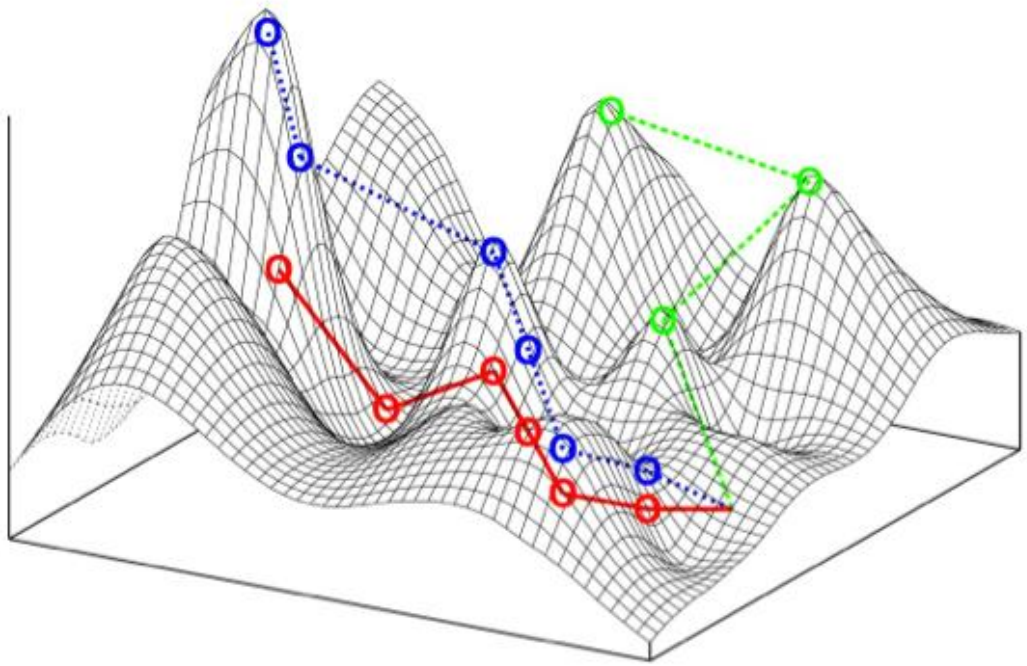
3.1 NK model

Problemi se generiraju korištenjem NK modela (engl. *NK-landscape*) s različitim brojem gena/bitova (N), veličinom epistaze (K), modelom epistaze, tablicom

vrijednosti (funkcijom dobrote, engl. *fitness*) i maksimalnim brojem evaluacija po razredu problema.

Jedno pokretanje generatora čini jednu klasu problema gdje se generiraju primjerci problema za učenje (treniranje) te za testiranje. Broj instanci za učenje i ispitivanje može se zadati, a ukoliko nije drugačije zadano generira se 200 instanci za učenje i 50 za ispitivanje.

NK model je matematički model koje je opisao Stuart Kauffman kao neravno/hrapavo područje dobrote (engl. *fitness*). Područje funkcije i broj njenih lokalnih minimuma i maksimuma može se prilagoditi s dva parametra – N i K. Ovaj model primjenjuje se u evolucijskoj biologiji, imunologiji, optimizaciji, itd. Model je vrlo popularan u području optimizacija jer je jednostavno pomoću njega napraviti instancu NP-kompletnog problema. NK model definira kombinatorički prostor čija je rezultirajuća struktura „krajolik“ (engl. *landscape*) prikazan na slici 3.1.



Slika 3.1: Vizualizacija 2D funkcije nastale NK modelom

Kombinatorički prostor sastoji se od nizova (bitova ili nekih drugih diskretnih vrijednosti) duljine N. Za svaku jedinku (niz) određuje se dobrota (engl. *fitness*).

Vrijednost dobrote ovisi o samom modelu, ali glavna značajka NK modela je da je dobrota jedne jedinke (rješenja) S suma utjecaja (doprinosa) svakog gena (varijable) S_i u jedinki što se formalno zapisuje formulom (3.1).

$$F(S) = \sum_i f(S_i) \tag{3.1}$$

Doprinos (utjecaj) svakog gena (varijable) S_i (*locus*) općenito ovisi o vrijednosti K drugih gena (varijabli) što se zapisuje formulom (3.2)

$$f(S_i) = f(S_i, S_1^i, \dots, S_K^i), \tag{3.2}$$

gdje je S_j^i drugi gen (varijabla) o kojem ovisi dobrota S_i.

Dakle, funkcija dobrote (3.2) je mapiranje između jedinke duljine K+1 i skalarne vrijednosti, koje je Weinberger kasnije prozvao 'fitness doprinosima'. Doprinosi su najčešće slučajno odabrani nekom određenom vjerojatnošću.

U nastavku slijedi primjer NK modela.

Primjer 1.:

Neka se koristi NK model s nizovima bitova i parametrima N = 5, K = 1. Dobrota niza neka je suma svih pojedinačnih doprinosa varijabli, a doprinos varijable neka ovisi o toj varijabli i prvoj idućoj, zapisano formulom:

$$f(S_i) = f(S_i, S_{i+1}),$$

gdje zbog cikličnosti vrijedi i:

$$f(S_5) = f(S_5, S_1).$$

Neka su zadane dobrote doprinosa:

$$f(0,0) = 0, f(0,1) = 1, f(1,0) = 2, f(1,1) = 0.$$

Kolika je dobrota niza nekog niza?

Rješenje:

$$F(00101) = f(0,0) + f(0,1) + f(1,0) + f(0,1) + f(1,0) = 0 + 1 + 2 + 1 + 2 = 6$$

$$F(11100) = f(1,1) + f(1,1) + f(1,0) + f(0,0) + f(0,1) = 0 + 0 + 2 + 0 + 1 = 3$$

U primjeru za svih $2^5 = 32$ mogućih rješenja vrijednost dobrote je 0, 3 ili 6, pa tako ukoliko se traži minimum, rješenje može biti niz nula ili jedinica (dobrota je 0). Najveća dobrota iznosi upravo 6, pa tako postoji mnogo više maksimuma od kojih je jedan i niz 00101 iz rješenja.

Parametar K određuje stupanj epistaze, a epistaza je pojava (u biologiji) gdje gen ovisi o drugom genu ili genima [5] te dolazi do toga da se gen maskira drugim genima. U NK modelu se to ponašanje (pojava) imitira doprinosima vrijednosti dobrote.

U slučaju kada je K = 0, dobrota niza (jedinke) jednostavna je za izračunati jer je potrebno zbrojiti dobrote pojedinih varijabli. Globalni optimum postoji i lako ga je identificirati. Na primjeru binarnih nizova ukoliko je $f(0) > f(1)$, globalni optimum je niz nula (duljine N), a u obrnutom slučaju, globalni optimum je niz jedinica (duljine N).

Kada je K veći od nule, znači da dobrota jedne varijable ovisi o K drugih varijabli, dobrota niza je suma dobroti podnizova. Povećanje parametra K povećava neravnost krajolika dobrote (engl. *fitness landscape*), tj. povećava broj lokalnih optimuma što otežava pronalazak željenog globalnog optimuma.

3.2 Generator problema

Organizatori natjecanja implementirali su u programskom jeziku *Python* generator problema koji koristi NK model i koji se može preuzeti sa stranica natjecanja, tj. postoji poveznica koja vodi na stranicu *GitHub* poslužitelja na kojem se nalazi izvorni kod generatora [6]. Za pokretanje je potrebna verzija 3 programskog jezika *Python*. Upute za korištenje generatora napisane su u repozitoriju.

Generator problema pokreće se datotekom *ProblemClassGenerator.py* koja ima dodatne opcije o kojima se više može saznati pokretanjem kroz komandnu liniju s opcijom *--help*.

Postoji opcija za postavljenje direktorija u kojem će se pojaviti generirani primjerci problema za taj razred problema, zatim broj instanci za učenje te broj instanci za ispitivanje. Za ove opcije postoje pretpostavljene vrijednosti, pa tako se primjerci razreda problema pohranjuju u direktorij pod nazivom '*problems*' u trenutnom direktoriju u kojem se pokrenulo izvršavanje, a za učenje se generira 200 primjeraka, dok za ispitivanje 50.

Prilikom generiranja stvaraju se dodatni direktoriji gdje ime jasno označava što se u direktoriju nalazi, pa tako primjerice postoji direktorij '*training*', '*testing*' i '*results*'.

Postoje i dodatne pomoćne datoteke od kojih u jednoj postoji opis razreda problema, a u druge dvije pohranit će se relativni putevi do datoteka s primjercima problema.

Dakle, stvorit će se datoteka *trainingFiles.txt* sadržaja sličnog onom prikazanom na slici 3.2.

```
200
training\00000.txt
training\00001.txt
training\00002.txt
training\00003.txt
training\00004.txt
training\00005.txt
training\00006.txt
training\00007.txt
training\00008.txt
training\00009.txt
training\00010.txt
training\00011.txt
```

Slika 3.2: Sadržaj datoteke trainingFiles.txt

Na slici 3.2 vidi se format datoteke gdje se u prvom retku nalazi broj primjeraka za učenje (200), a potom slijede putevi do datoteka s primjerkom problema. Na isti je način formatirana i datoteka *testingFiles.txt* s putevima do problema za ispitivanje.

Cijela struktura direktorija i datoteka predstavlja jedan razred problema s generiranim primjercima problema toga razreda.

Razred problema definira se:

- parametrom N, gdje je N slučajno odabran broj u intervalu [50, 300],
- parametrom K, gdje je K slučajno odabran broj u intervalu [1, 5],
- evaluacijskom konstantom koja može biti 1, 4 ili 16 (služi za izračun dozvoljenog broja evaluacija),
- funkcijom za definiranje interakcije varijabli, tj. za određivanje varijabli koje sudjeluju u doprinosu za pojedinu varijablu,
- funkcijom za drugačiji poredak (premještaj) unutar epistaza,
- funkcijom za generiranje vrijednosti dobrota,

- funkcijom za određivanje koliko jedinstvenih vrijednosti smije biti unutar cijele tablice s vrijednostima dobrota.

Funkcije za modeliranje povezanosti varijabli (utjecaja) su

- *NearestNeighbor* – varijabla ovisi o idućih K susjeda,
- *Unrestricted* – varijabla ovisi o K slučajno odabranih varijabli,
- *Separable* – jedinka je podijeljena u blokove veličine 2 * K, te se iz blokova pomoću *Unrestricted* načina biraju varijable o kojima ovisi,
- *Mesh* – stvara (K+1)-dimenzionalnu hiperkocku,
- *SAT_like* – stvara 4.27 * N skupova s K+1 slučajnih varijabli.

Funkcije za drugačiji poredak su:

- *Scatter* – slučajno izmijeni poredak unutar epistaze (engl. *shuffle*),
- *NoChange* – ne mijenja poredak.

Funkcije za generiranje dobrote:

- *Uniform* – uniformnom distribucijom u intervalu [0, 1],
- *Normal* – normalnom distribucijom u intervalu [0, 1],
- *Scaled* – male vrijednosti, slične u prosjeku uniformnoj i normalnoj distribuciji.

Funkcije za određivanje broja jedinstvenih vrijednosti unutar tablice:

- *TwoValues* – samo dvije različite vrijednosti dobrote će se pronaći u tablici dobrote,
- *PowKValues* – 2^K različitih vrijednosti za cijelu tablicu,
- *AllUnique* – omogućuje da sve vrijednosti u tablici budu različite.

Broj dozvoljenih evaluacija računa se po formuli (3.3):

$$\frac{2 \cdot N^2}{eval\_const}, \tag{3.3}$$

gdje je *eval_const* spomenuta evaluacijska konstanta čija vrijednost može biti 1, 4 ili 16.

S obzirom da je natjecanje novo i podložno promjenama u trenutku izrade rada generator generira primjerke problema s podacima prikazanim na slici 3.3.

U prvom retku se redom nalaze parametar N (broj varijabli), broj dozvoljenih evaluacija, broj varijabli u retku (K + 1) te broj redaka u datoteci. Nakon toga slijedi upravo toliko redaka gdje u retku prvih K + 1 brojeva označava varijable (indekse), a potom slijedi 2^K dobrota.

Na primjeru sa slike 3.3 instanca je razreda problema definiranog s N = 75, K = 1, eval_const = 1, funkcijom *Mesh* za utjecaje, *Scatter* za poredak varijabli, *Scaled* za odabir vrijednosti dobrota te *TwoValues* za jedinstvenost dobrota (ovi se podaci mogu pročitati u datoteci *meta.txt*).

U datoteci s instancom problema vidimo dozvoljen broj evaluacija izračunat preko formule (3.3):

$$\frac{2 \cdot N^2}{eval\_const} = \frac{2 \cdot 75^2}{1} = 11250.$$

```
075 011250 2 150
001 059 0.1640 0.3989 0.1640 0.3989
001 008 0.3989 0.1640 0.3989 0.3989
059 055 0.3989 0.1640 0.3989 0.3989
059 019 0.3989 0.1640 0.3989 0.3989
055 067 0.1640 0.3989 0.3989 0.1640
055 029 0.1640 0.3989 0.3989 0.3989
067 016 0.3989 0.1640 0.3989 0.1640
067 051 0.3989 0.1640 0.3989 0.1640
016 009 0.3989 0.1640 0.3989 0.3989
016 049 0.1640 0.1640 0.1640 0.1640
009 041 0.3989 0.1640 0.1640 0.1640
009 057 0.3989 0.3989 0.1640 0.1640
041 028 0.1640 0.1640 0.1640 0.1640
041 004 0.1640 0.1640 0.3989 0.1640
028 011 0.1640 0.3989 0.3989 0.1640
028 012 0.3989 0.3989 0.3989 0.1640
011 001 0.3989 0.3989 0.1640 0.1640
011 003 0.1640 0.1640 0.1640 0.3989
```

Slika 3.3: Primjer datoteke s instancom problema

No, od interesa je bilo usporediti rezultate dobivene na natjecanju s rezultatima i optimalnim rješenjima koje dosežu algoritmi unutar ECF okruženja.

Problemi koji su bili generirani od organizatora natjecanja za ispitivanje svih algoritama natjecatelja mogu se također preuzeti sa stranice natjecanja.

Očito je tada bila drugačija verzija generatora jer su instance tih problema formatirane po primjeru sa slike 3.4 u nastavku.

```
126 015876 2
000 091 077 0.2948 0.6308 0.6308 0.5361 0.5412 0.2948 0.2948 0.8351
001 052 122 0.8147 0.5361 0.5412 0.4521 0.2948 0.8351 0.8147 0.8147
002 022 092 0.5412 0.1871 0.1871 0.4521 0.8351 0.8147 0.8147 0.4521
003 054 008 0.8147 0.6308 0.8351 0.4521 0.8147 0.6308 0.8147 0.4521
004 076 099 0.8351 0.5361 0.5412 0.4521 0.5412 0.5361 0.5412 0.4521
005 019 115 0.1871 0.8351 0.2948 0.6308 0.8351 0.8351 0.2948 0.5412
006 075 069 0.2948 0.1871 0.2948 0.5361 0.8147 0.8351 0.1871 0.6308
007 081 105 0.1871 0.8147 0.8351 0.4521 0.2948 0.8351 0.8147 0.5361
008 105 064 0.1871 0.5361 0.8147 0.5361 0.6308 0.1871 0.2948 0.2948
009 002 106 0.8147 0.6308 0.5361 0.5412 0.2948 0.8147 0.5361 0.5361
010 118 063 0.1871 0.6308 0.4521 0.1871 0.4521 0.8351 0.6308 0.2948
011 045 071 0.6308 0.8351 0.5361 0.2948 0.6308 0.5361 0.5412 0.6308
012 037 061 0.5361 0.2948 0.4521 0.6308 0.5361 0.2948 0.5412 0.8147
013 122 105 0.5361 0.5361 0.4521 0.2948 0.4521 0.5361 0.4521 0.5361
014 003 071 0.4521 0.2948 0.8351 0.8351 0.5361 0.5361 0.6308 0.2948
015 094 089 0.4521 0.2948 0.5412 0.8147 0.6308 0.1871 0.5412 0.8351
016 055 060 0.6308 0.4521 0.5412 0.2948 0.8351 0.8351 0.8351 0.5361
017 052 003 0.6308 0.8351 0.2948 0.4521 0.5361 0.5361 0.8351 0.4521
018 099 100 0.1871 0.6308 0.8147 0.6308 0.1871 0.6308 0.8351 0.2948
019 014 073 0.5361 0.5412 0.6308 0.6308 0.2948 0.8351 0.8147 0.8351
020 051 042 0.8147 0.6308 0.5412 0.5412 0.4521 0.8351 0.2948 0.2948
021 030 115 0.8351 0.8351 0.8351 0.1871 0.5412 0.1871 0.1871 0.5412
022 010 080 0.5412 0.8351 0.6308 0.5412 0.5412 0.8147 0.4521 0.4521
```

Slika 3.4: Format instance problema za ispitivanje na natjecanju

Ovdje je razred problema zadan s N = 126, K = 2, *eval_const* = 1, funkcijom *Unrestricted* za utjecaje, *NoChange* za poredak varijabli, *Normal* za odabir vrijednosti dobrota te *PowKValues* za jedinstvenost dobrota.

Formula (3.3) za izračun dozvoljenog broja evaluacija također je drugačija i računa se po formuli (3.4):

$$\frac{N^2}{eval\_const} = \frac{126^2}{1} = 15876. \tag{3.4}$$

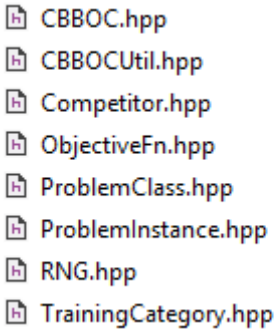
Također, treći parametar je K, ne K + 1, a u recima se nalaze prvo indeksi K + 1 varijabli koje maskiraju dobrotu. Nema ni posebno zapisanog broja redaka u prvom retku datoteke, već je broj redaka uvijek jednak broju varijabli, s obzirom da se koriste samo funkcije *NearestNeighbor*, *Unrestricted* i *Separable*.

Nadalje, organizatori su pripremili i API za pisanje algoritama gdje je funkcija izračuna dobrote bila implementirana. Postoje inačice za programske jezika *C++*, *Java* i *Python*. U radu je korišten API napisan *C++* programskim jezikom, a opis slijedi u nastavku.

3.3 API za algoritme natjecatelje

Izvorni kod pružene potpore za algoritme natjecatelje nalazi se također na *GitHub* poslužitelju 44[7]. Postoje i upute za njegovo korištenje [8], no u ovom radu nije korišten potpuni API, s obzirom da je cilj bio isprobati algoritme iz ECF-a.

Iskorištena je implementacija čitanja instanci problema iz opisanog formata datoteka te implementacija izračuna dobrote neke jedinke prilikom optimizacije. Slika 3.5 prikazuje datoteke iz API-ja.



Slika 3.5: API za natjecatelje

Kako bi se implementacija iskoristila za evaluaciju jedinki unutar ECF-a iskorišten je razred *ProblemInstance* koji zna pročitati format datoteke s instancom problema.

ProblemInstance razred učitavao je instance problema po formatu koji trenutno generira njihov generator problema, a s obzirom da koristimo primjere generirane tijekom natjecanja, modificirano je čitanje datoteka tako da očekuje samo tri parametra u zaglavlju primjerka problema.

API očekuje unos razreda problema (ime direktorija) u datoteku *classFolder.txt*, nakon toga API sam dalje čita sve što mu je potrebno iz strukture direktorija u razredu problema.

Postoji sučelje *Competitor* koje predstavlja algoritme natjecatelje; u radu nije korišten jer se koriste algoritmi iz ECF-a. Također, *ObjectiveFn* pohranjuje *ProblemInstance* te broj preostalih evaluacija, što korištenjem ECF-a nije potrebno dodatno pamtiti s obzirom da on to automatski radi, pa tako ni taj razred nije korišten. *ProblemClass* nije posebno korišten jer se ECF vrti nad pojedinim zadacima neovisno o razredu problema kojem zadatak pripada.

3.4 Prilagodba za ECF

Kako je već spomenuto od cijelog API-ja korišten je samo razred *ProblemInstance* koji zna pročitati datoteku s potrebnim podacima za evaluaciju funkcije.

U ECF-u ostvaren je novi evaluacijski operator koji koristi preuzeti razred. *ProblemInstance* koristi vektor *bool* vrijednosti za evaluaciju, dok ECF koristi svoje genotipe, no korišten je *BitString* genotip koji zapravo predstavlja vektor *bool* vrijednosti.

Evaluacijski operator u ECF-u (*EvaluateOp*) ima tri bitne metode koje je potrebno implementirati u novom operatoru, a navedene su u pseudokodu 3.1:

```
FitnessP evaluate(IndividualP individual);
void registerParameters(StateP);
bool initialize(StateP);
```

Pseudokod 3.1: Sučelje evaluacijskog operatora

Za korištenje programa nije potrebna određena struktura direktorija već je potrebno ECF-u poslati parametar koji će sadržavati točan put do datoteke i ime datoteke s instancom problema. Put može biti apsolutan, ali i relativan. Parametar koji je potrebno zadati u konfiguracijskoj datoteci je *problem.file*. Na ovaj način moguće je izvršiti optimizaciju nad instancom problema bilo kojeg razreda problema.

Moguće je pokrenuti optimizaciju više instanci odjednom. U tom je slučaju potrebno da se sve datoteke (instance problema) nalaze u tom određenom direktoriju te da su imena datoteka peteroznamenkasti brojevi s vodećim nulama. Tada se zadaje i parametar *testing.instances* koji označava koliko instanci treba optimizirati.

Navedeni parametri čitaju se prilikom inicijalizacije (*initialize* metoda), a u metodi *registerParameters* potrebni parametri se registriraju.

Metoda *evaluate* poziva metodu iz *ProblemInstance* razreda koja računa dobrotu poslane jedinke. Kod te metode je u nastavku (pseudokod 3.2).

```
metoda evaluiraj (jedinka)
     total = 0
     za i = 0 do broj_redaka radi
          indeksi_var = varijable_koje_pridonose
          ind_u_tablici = 0
```

```
        za j = 0 do indeksi_var.velicina radi
             ind_u_tablici <<= 1
             ind_u_tablici |= jedinka[indeksi_var[j]]
        total += dobrote[ind_u_tablici]
    vrati total
```

Pseudokod 3.2: Evaluacija jedinke

4. Algoritam P3

Problemi iz stvarnog svijeta za koje se koriste evolucijski algoritmi često su ograničeni zbog potrebe za definiranjem parametara specifičnih za određeni problem. Neke metode pokušavaju smanjiti potrebu za parametrima i njihovim ugođavanjem, no to uobičajeno smanji učinkovitost algoritma i njegovih rezultata. P3 algoritam, čije je puno ime *Parameterless Population Pyramid*, evolucijski je algoritam koji nema parametre i unatoč tome vrlo je učinkovit na raznim problemima [9].

P3 je iterativna metoda koja stvara kolekciju populacija i ne zahtijeva niti jedan parametar od korisnika. Koristi entropiju za shvaćanje i detekciju povezanosti rješenja kako bi se što učinkovitije pomiješala rješenja bez dodatnih informacija o funkciji (problemu) osim načina evaluacije. Za razliku od drugih neparametarskih algoritama, po [9], eksperimentalno se P3 pokazao za konstantni iznos bolji od optimalno konfiguriranih (parametriziranih) optimizacijskih metoda s kojima se mogao usporediti.

P3 nema samo jednu populaciju rješenja kao uobičajeni genetski algoritmi, već održava strukturu sličnu piramidi (od tuda i ime algoritma) tako da svaka razina piramide sadrži jednu populaciju rješenja (jedinki). U nastavku je prikazan pseudokod 4.1. u kojem je prikazano kako P3 iterativno gradi piramidu populacija te kako generira, tj. poboljšava rješenja kroz *hill climbing* i križanje. P3 može se koristiti za diskretne optimizacije.

```
metoda Iteracija-P3
     stvori slučajno rješenje
     provedi hill climber
     ako rješenje nije u sva_rješenja
          dodaj rješenje u P_0
          dodaj rješenje u sva_rješenja
     za sve P_i u piramidi radi
          križaj rješenje s P_i
          ako se dobrota rješenja poboljšala
               ako rješenje nije u sva_rješenja
                    dodaj rješenje u P_(i+1)
                    dodaj rješenje u sva_rješenja
```

Pseudokod 4.1: Jedna iteracija P3 algoritma

Dodatna pojašnjenja dijelova napisanog pseudokoda su nastavku.

4.1 Hill Climber

Kao prvi korak poboljšanja rješenja koristi se *First Improvement Hill Climber* (FIHC), gdje je ime samo opisno (postoji u drugim literaturama pod drugim imenom).

U jednom rješenju neka su sve vrijednosti varijabli slučajno razmještene. Za svaku se varijablu isprobaju sve moguće vrijednosti varijable (vrijednosti su diskretne). Ako barem jedna takva promjena pridonese poboljšanju rješenja, promjena će se zadržati. Proces se ponavlja sve dok promjena ne pridonese poboljšanju rješenja.

Kako bi se izbjeglo bespotrebno korištenje evaluacija, FIHC pamti varijable nad kojima su isprobane alternativne vrijednosti. Ukoliko algoritam ponovo pristupi već isprobanoj varijabli, nastavit će na iduću varijablu bez evaluacija.

FIHC je osmišljen kako bi pronašao optimum koristeći minimalni broj evaluacija, a ispitivanjem varijabli slučajnim poretkom ne stvara ovisnost (sklonost) o poziciji.

4.2 Piramida

P3 održava strukturu nalik piramidi čije su razine zapravo populacije pri čemu su populacije s boljim jedinkama na višim razinama.

Razine unutar piramide su disjunktni skupovi jedinstvenih rješenja koji predstavljaju jednu populaciju. Ova se karakteristika održava kroz cijelu izgradnju piramide jer se novo rješenje dodaje samo kada ne postoji niti u jednoj razini piramide. Provjera jedinstvenosti rješenja postiže se u konstantnom vremenu održavanjem seta svih jedinstvenih rješenja u piramidi.

Svaka populacija (razina piramide) održava tablicu s frekvencijama parova varijabla-vrijednost koja se koristi prilikom križanja rješenja s populacijom. Također se gradi kroz izvođenje algoritma tako da se u početku frekvencije stave na 0. Svaki puta kada se rješenje dodaje u neku razinu piramide, za sve se varijable provjerava vrijednost te se tablica parova ažurira. Ovo zahtijeva $O(N^2)$ operacija (ne evaluacija) svaki puta kada se rješenje dodaje u populaciju.

4.3 Križanje

P3 koristi entropiju vrijednosti varijabli kako bi stvorio grozd (klaster, engl. *cluster*) varijabli čije bi se vrijednosti trebale očuvati. Dva su koraka kod definiranja ovog

tipa križanja: stvaranje klastera (engl. *cluster creation*) i korištenje klastera (engl. *cluster usage*).

4.3.1 Stvaranje klastera (*cluster creation*)

Tijekom stvaranja klastera, P3 stvara binarno stablo klastera tako da listovi predstavljaju pojedine varijable, a svaki unutarnji čvor predstavlja podskup varijabli kojeg čine varijable iz njegove djece. Koristeći tablicu frekvencija vrijednosti po jednadžbi (4.1) računa se udaljenost između klastera varijabli po entropiji (jednadžba (4.2)(4.2)).

$$D(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{c_i \in C_i} \sum_{c_j \cup C_j} 2 - \frac{H(C_i) + H(C_j)}{H(C_i \cup C_j)} \tag{4.1}$$

$$H(C) = -\sum_{s \in S} p_c(s) \log(p_c(s)) \tag{4.2}$$

U nastavku je pseudokod 4.2 koji prikazuje stvaranje stabla povezanosti.

```
metoda Stvaranje_klastera
     nespojeni = {{0}, {1}, {2},…, {N-1}}
     korisni = nespojeni
     dok nespojeni.velicina > 1 radi
          C_i, C_j = min_{C_i,C_j iz nespojeni}(D(C_i, C_j))
          nespojeni = nespojeni – {C_i, C_j} + {C_i ∪ C_j}
          korisni = korisni + {C_i ∪ C_j}
          ako D(C_i, C_j) = 0 onda
               korisni = korisni – {C_i, C_j}
     sortiraj korisni prema veličini klastera, uzlazno
     makni najveći klaster iz korisni
     vrati korisni
```

Pseudokod 4.2: Stvaranje klastera (stabla povezanosti)

Krećući sa svim varijablama u pojedinim klasterima, jednadžba (4.2) se koristi kako bi pronašla koji klasteri iz *nespojeni* imaju najmanju udaljenost (najviše su povezani). Ti klasteri se povezuju u novi klaster i uklanjaju iz *nespojeni*, dok *korisni* prati sve stvorene klastere. Ovaj se proces ponavlja dok sve varijable nisu spojene u jedan klaster.

Nisu svi stvoreni klasteri u stablu korisni za križanje, pa tako npr. posljednji klaster koji sadrži sve varijable nije koristan jer se njime ne može stvoriti nova jedinka. Također, uklanjaju se i klasteri koje direktni roditelji sadržavaju u potpunosti. Dakle, kad je $D(C_i, C_j) = 0$, C_i i C_j se ne koriste prilikom križanja. Nad malim populacijama, na ovaj se način znatno smanjuje broj klastera koji se koristi prilikom križanja. Primjerice, za populaciju od jedne jedinke, klasteri ne postoje, dok za populaciju od dvije jedinke, postoje najviše dva klastera: sve varijable jednake i sve varijable različite.

4.3.2 Korištenje klastera (*cluster usage*)

Križanje se koristi kako bi se poznate informacije iz razine piramide kombinirale s novim rješenjem kandidatom prema pseudokodu 4.3.

```
metoda korištenje_klastera
    za svaki klaster C_i iz korisni radi
        za svaku jedinku S_i u P_i radi
            kopiraj iz S_i varijable C_i u rješenje
            ako se rješenje promijenilo onda
                ako se dobrota smanjila onda
                    vrati vrijednosti
                prekini petlju
```

Pseudokod 4.3: Korištenje klastera

Dakle, za svaki klaster stvoren prilikom stvaranja klastera, izabire se slučajan *donor*, jedinka iz trenutne razine (populacije) tako da se iz odabrane jedinke vrijednosti varijabli u usporedbi s novim rješenjem (rješenje nad kojim se križanje obavlja) razlikuju u barem jednoj varijabli iz klastera. Idući korak je kopirati vrijednosti varijabli iz *donora* u novo rješenje. Ukoliko se dobrota rješenja smanjila, promjene se uklanjaju.

Donor se bira tako da se razlikuje bar u jednoj varijabli od rješenja kandidata te se to pokazalo isplativije nego dozvoljavati promjene vrijednosti varijabli.

Križanje se završava nakon što se ispitaju svi korisni klasteri, gdje svaki zahtijeva najviše jednu evaluaciju. Klasteri se ispituju od klastera najmanje veličine do onih najveće jer se tako osigurava da su manji dijelovi optimizirani prije nego se naprave veće promjene unutar jedinke čime se zadržava različitost.

4.4 Algoritam P3 u ECF-u

Unutar ECF-a algoritam je implementiran tako da koristi *BitString* genotip s obzirom da koristi diskretan prostor pretraživanja. S obzirom da algoritam interno gradi opisanu strukturu populacija – piramidu, potrebno je ECF-ovu generiranu populaciju zamijeniti jedinkama iz piramide. Iz tog razloga interno se koristi i brojač koji označava indeks jedinke iz ECF-ove populacije koja se iduća treba zamijeniti novom jedinkom koja se dodaje u piramidu.

Algoritam nema parametara, pa se u konfiguracijskoj datoteci može zadati praznim *tagom* `<P3Alg/>`.

5. CMA-VNS algoritam

CMA-VNS algoritam čije je puno ime *Covariance Matrix Adaptation Variable Neighborhood Search* je hiper-heuristika koja koristi CMA-ES (*Covariance Matrix Adaptation Evolution Strategy*) evoluciju nakon koje slijedi iterativna lokalna pretraga (engl. *local search*). Razlog korištenja CMA-ES je učinkovita predikcija varijabli. Stoga lokalna pretraga, pretraga susjedstva varijable u ovom algoritmu, može biti olakšana naspram skupih i eventualno ograničenih evaluacija [10].

Pseudokod algoritma (5.1) je u nastavku.

```
metoda CMA-VNS(N, broj_evaluacija)
     parametri = premaPredefPravilima(N, broj_evaluacija)
     elitna = prazan_skup
     dok parametri.cma_eval radi
          rješenja = bipopCMAES(parametri.lambda)
          elitna += rješenja.novi_najbolji
     dok parametri.vns_eval radi
          backbone = najčešći_bitovi(elitna)
          s0 = bipopCMAES.predikcija_s_backbone
          rješenje = VNS(s0)
          ako je rješenje novo najbolje onda
               elitna += rješenje
```

Pseudokod 5.1: CMA-VNS algoritam

Algoritam ima dva dijela – CMA blok i VNS blok. Udio tih dijelova određen je hiper-heuristikom koja je unaprijed ispitala karakteristike nekolicine načina i naučila skup pravila iz rezultata.

CMA dio trenira (uči) bi-populacijski CMA-ES da predviđa rješenja nekog *black-box* problema (problem čije karakteristike i detalji nisu poznati). Najvažniji parametar λ (broj djece) određen je predefiniranim pravilima. Ukoliko su pronađena nova najbolja rješenja dodaju se u elitni skup osim rješenja dobivenih na samom početku optimizacije (preuranjeno).

VNS dio koristi tri mehanizma. Prvo, *backbone*, osnova (uobičajeni, najčešći bitovi u elitnom skupu) se gradi kako bi se ojačala pretraga. Zatim se procijenjeno rješenje generira CMA-ES metodom kako bi se kombiniralo s osnovom. Na kraju, VNS dio pokušava pronaći poboljšanja u generiranom rješenju. Ukoliko je novo rješenje bolje od do tada najboljeg, novo se rješenje dodaje u elitni skup.

5.1 CMA-ES

CMA-ES (*Covariance Matrix Adaptation Evolution Strategy*) temelji se na multivarijantnoj normalnoj distribuciji i uspješno pronalazi optimalna rješenja problema na način da postupno prilagođava očekivanje, standardnu devijaciju i kovarijantnu matricu distribucije [11]. Smatra se jednim od najboljih algoritama za optimizaciju kontinuiranih funkcija.

Algoritam postepeno povećava uspješnost generacije na način da povećava izglednost odabira uspješne jedinke [12]. Time će se populacija sve više približavati optimalnoj vrijednosti, dok će kovarijanca i disperzija biti sve manje, čime se pretraga fokusira na manje, lokalizirano područje. Drugo svojstvo algoritma je pamćenje evolucijskog puta što će omogućavati da pretraga optimalnog rješenja ide u pravome smjeru.

5.1.1 Varijante CMA-ES algoritma

Neke od poznatijih varijanta CMA-ES algoritma [13] su:

- LR-CMA-ES (engl. *local restart* CMA-ES) – moguće je ponovo pokrenuti izvođenje ako se primijeti da algoritam ne uspijeva dovoljno brzo konvergirati,
- IPOP-CMA-ES (engl. *increasing population size* CMA-ES) – varijanta u kojoj je omogućeno ponovno pokretanje s novom populacijom koja sadrži više jedinki od prošle,
- BIPOP-CMA-ES (engl. *bipopulation* CMA-ES) – postoje dvije populacije, jedna za globalno pretraživanje i druga manja za lokalno pretraživanje,
- G-CMA-ES (engl. *global* CMA-ES) – sličan IPOP varijanti, ali s mogućnošću ponovnog pokretanja algoritma ako se ne pronađe dovoljno dobro rješenje, te prilagodbom parametara za globalnu pretragu,
- L-CMA-ES (engl. *local* CMA-ES) – veličina koraka i veličina populacije su veoma mali čime se oponaša lokalna pretraga,
- A-CMA-ES (engl. *active* CMA-ES) – modifikacija u kojoj se ne koriste samo podaci o najboljim potomcima, nego i o najlošijim jer je ideja da se razdioba ne pomiče samo prema najboljem rješenju, već da se i odmiče od najlošijega,
- PS-CMA-ES (engl. *particle swarm* CMA-ES) - kombinacija PSO i CMA-ES optimizacijskih algoritama.

5.1.2 Parametri CMA-ES algoritma

Postoji puno parametara no nisu svi dostupni korisniku na mijenjanje, već se interno prilikom izvođenja algoritma postavljaju na izračunate vrijednosti.

Broj roditelja (μ) i broj potomaka (λ) su parametri koji određuju način selekcije za novu populaciju.

Očekivana vrijednost (engl. *mean*) je parametar koji određuje sredinu prostora oko koje se biraju nove jedinke prilikom stvaranja populacije.

Veličina koraka (engl. *step-size*, σ) je skalarna vrijednost koja određuje disperziju za sve dimenzije. Veličina koraka je promjenjiva vrijednost te se prilagođava s obzirom na uspješnost generacije. Veličina promjene koraka ovisi o njegovom evolucijskom putu (engl. *evolution path*). Ukoliko je duljina puta velika, veličina koraka treba se povećati kako bi se moglo obuhvatiti optimalno rješenje.

Kovarijantna matrica (engl. *covariance matrix*, C) ponaša se kao parametar disperzije za više dimenzija. Vrijednosti unutar matrice određuju kovarijancu između elementa tog retka i stupca. Mijenja se na način da pomiče prostor pretraživanja prema jedinkama koje su u prošlom koraku bile uspješnije.

Stope učenja (engl. *learning rate*) su konstante koje određuju koliko brzo i koliko jako će se događati promjene.

5.2 Parametri CMA-VNS algoritma

Parametri CMA-VNS algoritma su zapravo samo dimenzija prostora pretraživanja (broj varijabli jedinke) te najveći dozvoljeni broj evaluacija. Pomoću ta dva parametra i predefiniranih pravila CMA-VNS odredi interne parametre. Od tih internih parametara broj potomaka λ i veličina koraka σ predaju se CMA-ES algoritmu.

5.3 Dodatna svojstva CMA-VNS algoritma

Tijekom izvođenja algoritam pamti, tj. stvara *cache* složenosti O(1) za dohvaćanje dobrote jedinke. Stvara mapiranje jedinke s njenom vrijednosti te prije evaluacija provjerava postoji li već dobrota jedinke u *cache* mapi. Ukoliko već postoji smanjuje se vrijeme izvođenja i trošak evaluacija.

Koristi i tabu listu invertiranja bitova, što znači da izbjegava ponovno ispitivanje invertiranih vrijednosti koje su nedavno isprobane i nisu pružile napredak jedinke.

CMA-VNS ne koristi učenje već su parametri algoritma definirani predefiniranim pravilima. Dimenzija (N) i maksimalni broj evaluacija su dva parametra koja definiraju pravila. Ta su se pravila pokazala dovoljno učinkovitima bez ponovnog pokretanja i učenja za svaki NK problem s natjecanja.

5.4 CMA-VNS algoritam u ECF-u

Korištena je implementacija CMA-ES algoritma koja ne sadrži BIPOP-CMA-ES varijantu nego se može odabrati između tri implementirane, a to su: IPOP-CMA-ES, A-CMA-ES i uobičajeni CMA-ES. Uobičajeni CMA-ES je zapravo $(\mu/\mu_w, \lambda)$-CMA-ES, gdje μ_w označava da se prilikom izbora roditelja koristi težinska funkcija. CMA-VNS ne prima posebno parametre za CMA-ES dio već se koriste pretpostavljene vrijednosti osim za broj potomaka λ i veličinu koraka σ koje CMA-VNS odredi predefiniranim pravilima.

Pamćenje dobrota jedinki nije potrebno posebno ostvarivati s obzirom da jedinka u ECF-u uvijek pamti svoju dobrotu nakon što se evaluira. Unutar ECF-a koristi se elitni skup rješenja te ukoliko je jedinka u njemu, ne treba provoditi dodatne evaluacije.

S obzirom da su jedini parametri koje CMA-VNS zahtijeva dimenzija rješenja i najveći dozvoljeni broj evaluacija, a ti se parametri ionako već određuju unutar konfiguracijske datoteke ECF-a, CMA-VNS zapravo nema parametre. No, kako se koristi implementacija CMA-ES algoritma s više varijanti izvođenja, CMA-VNS algoritmu zapravo se zadaje parametar koji određuje s kojim CMA-ES algoritmom će se izvoditi.

CMA-VNS algoritam implementiran je da koristi *FloatingPoint* s obzirom da implementacija CMA-ES algoritma očekuje ili *Binary* ili *FloatingPoint* genotip. Opisan algoritam pretpostavljao je diskretan prostor pretraživanja – bitove. Ovdje se interno koristi pretvorba u bitove, ali vrlo jednostavna. Ukoliko je vrijednost realnog broja neke varijable veća ili jednaka određenoj vrijednosti interno zadanoj, tada vrijednost predstavlja istinitu *bool* vrijednost.

6. Rezultati

6.1 Ispitivanje postojećih algoritama

Kao što je već spomenuto, korišten je razred *ProblemInstance* za učitavanje funkcija (problema) iz datoteke te evaluaciju. S obzirom da je za evaluaciju potreban vektor binarnih vrijednosti u ECF-u (8) je korišten *BitString* genotip.

Algoritmi iz ECF-a koji rade s *BitSting* genotipom su:

- *Steady State Tournamet* (SST)*,*
- *Roulette Wheel* (RW),
- *Evolution Strategy* (ES),
- *Elimination*,
- *Genetic Annealing* (GA).

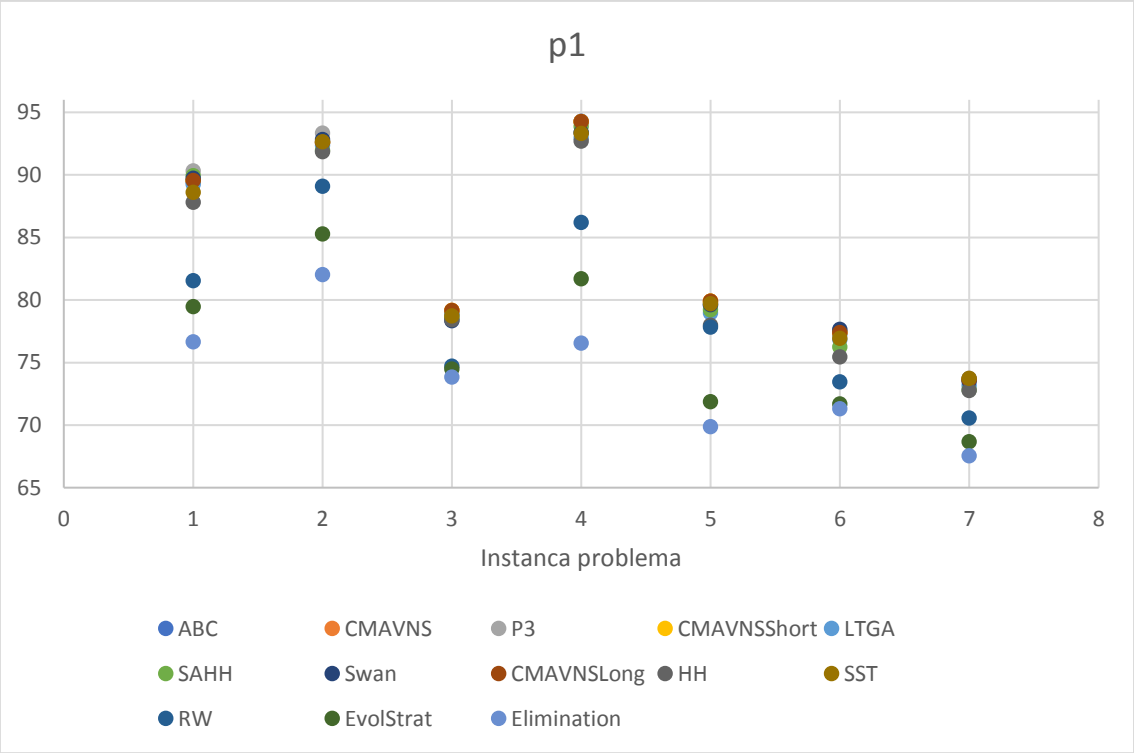
Posljednji navedeni algoritam *Genetic Annealing* radi s *BitStringom*, no implementiran je samo za minimizaciju, a nad ovim problemima vrši se maksimizacija, pa nije korišten.

Za ovo ispitivanje korišteni su problemi generirani za samo natjecanje i ispitivanje algoritama natjecatelja. Generirano je 20 različitih razreda problema koji se mogu preuzeti sa stranica natjecanja. Navedeni algoritmi koji se ispituju postavljeni su u jednaku situaciju kao i algoritmi natjecatelji, što znači da su ograničeni brojem evaluacija koje određuje razred problema te su pokretani samo jednom s obzirom da su se i algoritmi natjecatelji izvodili po jednom nad svakom ispitnom instancom. Ograničenja postavljena na algoritme služe kako bi se ostvarila što realnija situacija.

Za algoritme iz ECF-a korištene su pretpostavljene (uobičajene) vrijednosti parametara, više o samim algoritmima opisano je u dodatku rada.

Korišten je razred problema „p1“ koji je definiran brojem varijabli – 126, te maksimalnim dozvoljenim brojem evaluacija – 15876. Ostale karakteristike problema su K = 2, *eval_const* = 1, te funkcije *Unrestricted*, *NoChange*, *Normal* i *PowKValues* za stvaranje tablice vrijednosti dobrota.

Na slici 6.1 prikazani su rezultati za razred problema „p1“ za prvih nekoliko primjera. Na slici 6.1 se vide rezultati po tri najbolja algoritma (s natjecanja) iz svake kategorije – bez učenja, s kratkim te dugim učenjem i rezultati algoritama iz ECF-a.



Slika 6.1: Rezultati algoritama na problemu "p1"

Na apscisi se nalaze brojevi koji zapravo označavaju instancu problema, dok su na osi ordinata vrijednosti dobivene optimizacijom. Za svaku instancu su označeni algoritmi različitom bojom, a na dnu se nalazi legenda za lakše snalaženje. S obzirom da se rezultati dosta preklapaju, a algoritama je puno, u nastavku je i tablica (Tablica 6.1) s rezultatima algoritama iz ECF-a.

Tablica 6.1: Rezultati algoritama iz ECF-a nad problemom "p1"

funkcija	SST	RW	ES	Elim	P3	CMA-VNS	Najbolji
1	88,5978	81,5353	79,4705	76,6451	89,7213	89,4922	90,3235 (P3)
2	92,6345	89,0867	85,2651	82,0226	93,1272	91,9471	93,3318 (P3)
3	78,7282	74,7013	74,4862	73,8310	79,0388	79,1322	79,1587 (CMA-VNS)
4	93,3124	86,2067	81,6944	76,5525	94,2759	94,2634	94,2709 (ABC)
5	79,6862	77,8335	71,8721	69,8644	79,7461	79,6849	79,8947 (CMA-VNS)
6	76,9241	73,4623	71,6770	71,3091	77,4223	77,5981	77,6582 (Swan)
7	73,7400	70,5503	68,6728	67,5545	73,5250	73,6421	73,7081 (P3)

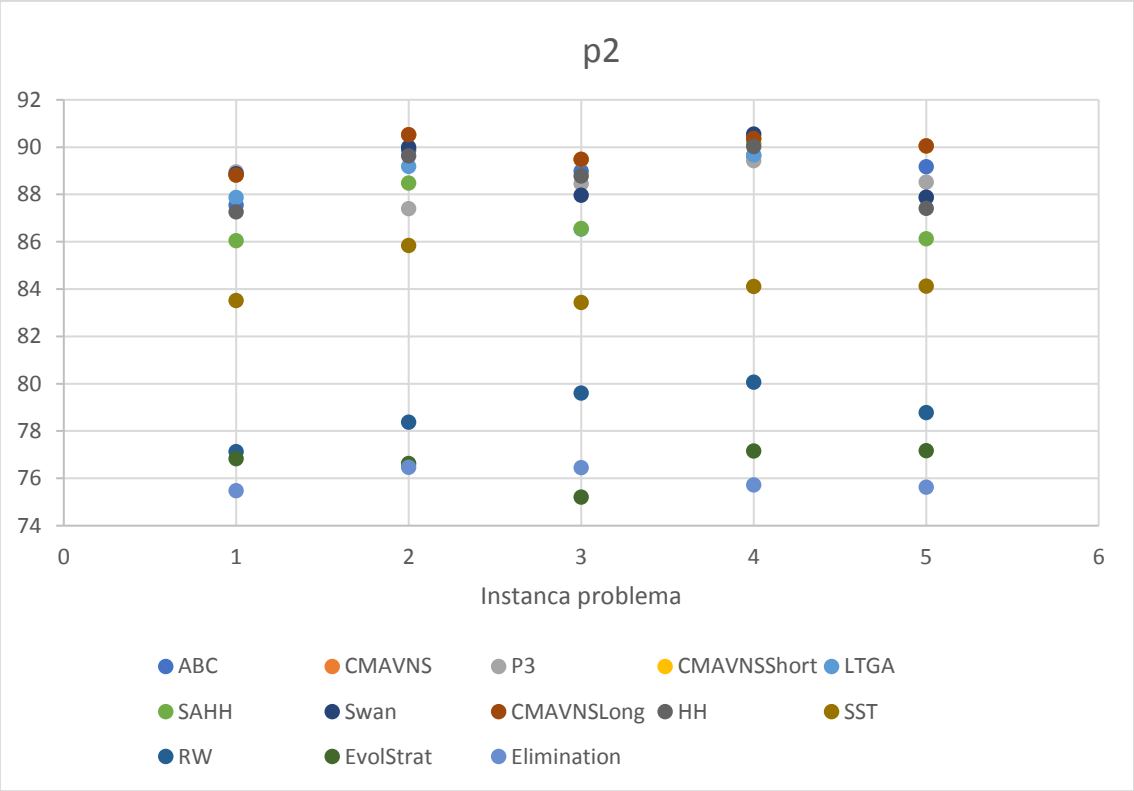
Rezultati algoritama s natjecanja dostupni su na stranici natjecanja, a u tablici su navedena najbolja rješenja s natjecanja te koji od algoritama je postigao tu vrijednost. Također, prikazani su i rezultati dobiveni algoritmima P3 i CMA-VNS implementiranih u ECF-u. Prema ovim rezultatima možemo zaključiti da *Steady State Tournament* algoritam iz ECF-a najbolje radi na ovakvom razredu problema, te da se može uspoređivati s ostalim algoritmima s natjecanja. Ostali algoritmi iz ECF-a dosta odskaču u rješenjima od drugih algoritama te se za ovakav tip problema možda ne bi koristili.

Na isti način isprobani su algoritmi i na nekoliko instanci razreda problema „p2“ te su rezultati prikazani na isti način – tablicom 6.2 i grafom 6.2 u nastavku.

Tablica 6.2: Rezultati algoritama iz ECF-a nad problemom "p2"

funkcija	SST	RW	EvolStrat	Elimination	Najbolji
1	83,5100	77,1330	76,8241	75,4763	88,9509 (P3)
2	85,8358	78,3698	76,6288	76,4634	90,5231 (CMA-VNS)
3	83,4326	79,5972	75,2107	76,4543	89,4817 (CMA-VNS)
4	84,1144	80,0683	77,1583	75,7136	90,5490 (CMA-VNS)
5	84,1239	78,7836	77,1724	75,6285	90,0503 (CMA-VNS)

Kod razreda problema „p2“ maksimalan broj dozvoljenih evaluacija iznosi 4624, te zbog toga i rezultati kod algoritama natjecatelja dosta variraju (u usporedbi s problemima iz razreda „p1“) što se vidi na grafu 6.2. Ostali parametri razreda problema su broj varijabli (N) – 136, zatim K = 3, evaluacijska konstanta je 4, a funkcije koje određuju vrijednosti tablice dobrota su *NearestNeighbor*, *NoChange*, *Normal*, *AllUnique*.



Slika 6.2: Rezultati algoritama na problemu "p2"

6.2 Ispitivanje s FloatingPoint genotipom

Kako bi se problemi mogli ispitati i s algoritmima koji se izvršavaju samo nad *FloatingPoint* genotipom dodano je dekodiranje kontinuiranih vrijednosti u niz bitova prilikom same evaluacije rješenja. Način dekodiranja opisan je u radu [15] te se ovdje neće dodatno opisivati, no spomenut će se bitni parametri te način na koji ih pravilno postaviti.

Parametar *dimension* samo *FloatingPoint* genotipa treba biti djeljiv parametrom *truth_table.size* koji se zadaje u *Registry* bloku konfiguracijske datoteke. Parametar *truth_table.size* predstavlja veličinu vektora *bool* vrijednosti koja će se evaluirati, iz tog razloga taj parametar treba postaviti na broj varijabli zadan razredom, tj. instancom problema. Veličinom *FloatingPoint* genotipa može se odrediti koliko bitova će predstavljati jednu realnu vrijednost. Broj bitova koji predstavlja jednu vrijednost računa se kao $\frac{truth\_table.size}{dimension}$, pa zato trebaju biti djeljivi. Postoje još parametri koji označavaju način binarnog dekodiranja – uobičajeno ili *grayevo*, te način na koji će se bitovi postaviti unutar niza bitova – uobičajeno, slijedno ili će se bitovi jednog realnog broja razdvojiti.

U ovom ispitivanju koriste se pretpostavljene vrijednosti za navedene dodatne parametre – binarno slijedno dekodiranje.

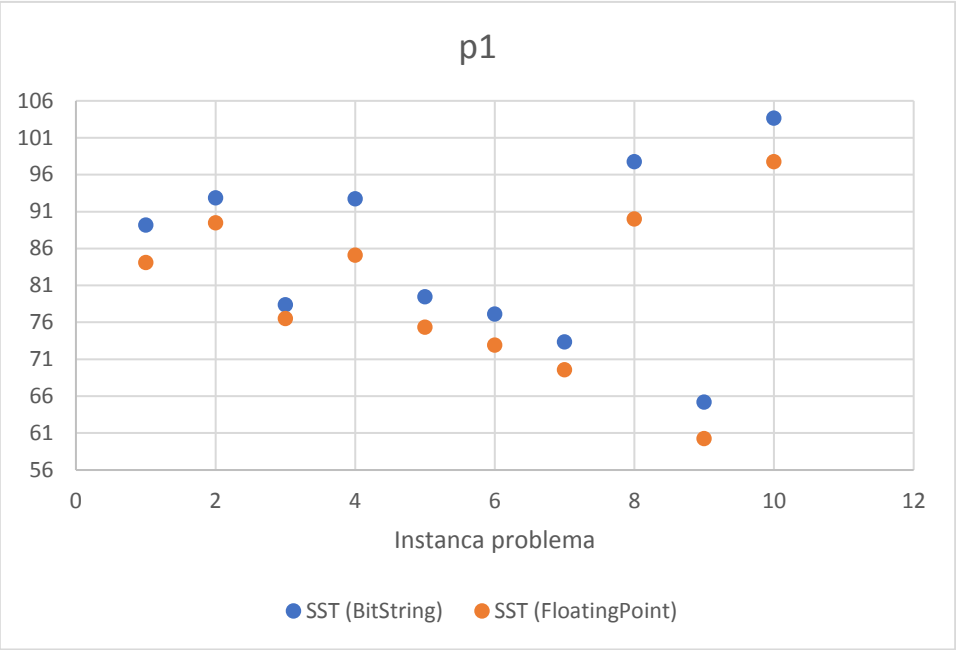
Sada se mogu pokrenuti algoritmi implementirani u ECF-u koji koriste *FloatingPoint* genotip jer će se prilikom evaluacije taj vektor dekodirati u niz bitova na parametrima određen način, a korištenje *BitString* genotipa kod evaluacije nije uklonjeno te se dva načina mogu uspoređivati za algoritme koji podržavaju oba genotipa.

S obzirom da algoritmi koji se mogu pokrenuti s *BitString* genotipom podržavaju i genotip *FloatingPoint* (neovise o genotipu), mogu se usporediti njihova rješenja dobivena različitim genotipima. U ovom ispitivanju pokrenuta je optimizacija na prvih trideset primjeraka razreda problema „p1“, no u grafovima je prikazano samo prvih deset radi preglednosti (gubitka informacije nema jer je situacija vrlo slična). Broj varijabli za taj problem je 126, pa se u *BitString* genotipu veličina postavila na 126, a za *FloatingPoint* genotip *truth_table.size* postavljen je na 126, dok je veličina *FloatingPoint* genotipa na 63, pa su dva bita pri dekodiranju predstavljala jednu vrijednost realnog broja. Dopušteni interval za vrijednosti realnih brojeva bio je [0, 4]. Razred problema „p1“ već je opisan u prošlom ispitivanju. Ispitivanje se izvršilo na jednak način kao i prethodno ispitivanje, osim opisane razlike.

Rezultati svih algoritama neovisnih o genotipu mogu se vidjeti u tablici 6.3. U nastavku su prikazani grafovi s rezultatima *Steady State Tournament* i *Roulette Wheel* algoritma s oba genotipa.

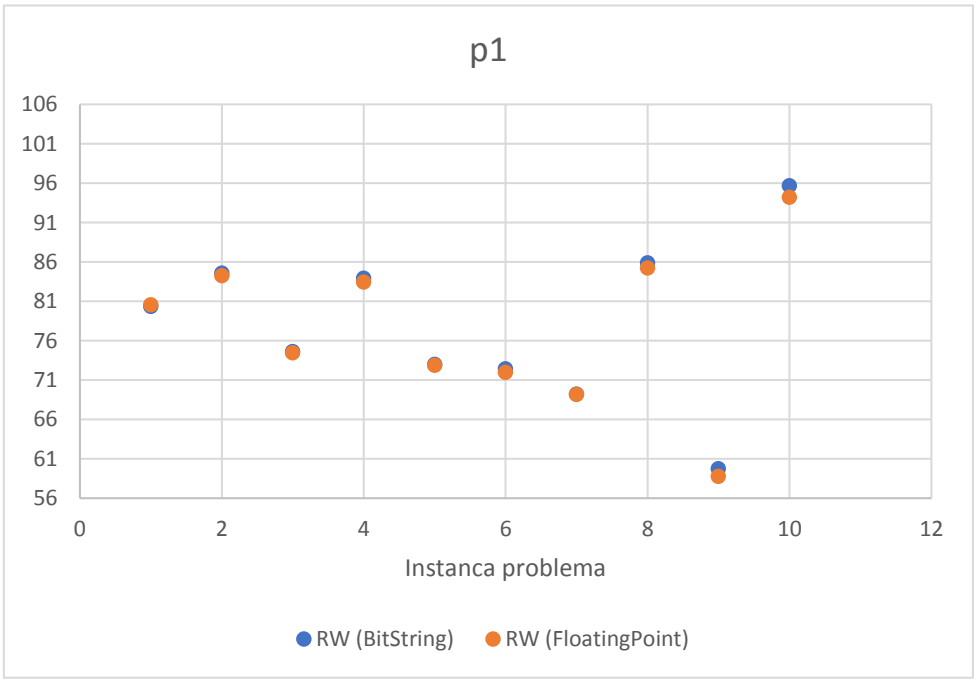
Tablica 6.3: Rezultati s BitString i FloatingPoint genotipom

funkcija	**SST (BS)**	**SST (FP)**	**RW (BS)**	**RW (FP)**	**ES (BS)**	**ES (FP)**	**Elim (BS)**	**Elim (FP)**
1	89,2141	84,1356	80,3400	80,5596	78,7825	77,7174	77,1692	77,8932
2	92,8845	89,5195	84,5884	84,2413	84,0503	82,9663	82,7453	83,1559
3	78,4226	76,5627	74,6252	74,4482	73,7886	73,5649	72,7668	72,2534
4	92,7764	85,1239	83,9338	83,4337	78,9107	79,0933	77,2565	76,8958
5	79,4910	75,3735	73,0110	72,8683	72,1931	70,7453	69,8422	70,3672
6	77,1354	72,9292	72,4435	71,9932	70,3599	69,8692	70,0053	69,6100
7	73,3928	69,6115	69,2382	69,1624	68,1692	68,6945	67,4065	66,8306
8	97,7768	90,0385	85,8705	85,2125	82,9696	83,2230	81,4356	81,0622
9	65,2163	60,2730	59,7489	58,7608	58,0319	56,5726	56,3402	56,4884
10	103,6630	97,7864	95,6924	94,2155	91,6076	89,8440	89,0601	88,9491



Slika 6.3: Usporedba genotipa za SST algoritam

Kod *Steady State Tournament* algoritma na slici 6.3 postoji veća razlika između rezultata dobivenih različitim genotipima, dok je kod *Roulette Wheel* algoritma (*Elimination* i ES algoritma) razlika vrlo mala. Za *Steady State Tournament* algoritam *BitString* genotip pronalazio je bolja rješenja.



Slika 6.4: Usporedba genotipa za RW algoritam

Ukoliko se uspoređuju rezultati između dva algoritma *Steady State Tournament* je bolji, također, bolji je i od druga dva algoritma.

Na slici 6.4 prikazani su rezultati *Roulette Wheel* algoritma kod kojeg je razlika između rezultata s *BitString* i *FloatingPoint* genotipom mala; grafovi za druga dva algoritma vrlo su slična ovome.

Ispitani su algoritmi iz ECF-a koji ovise o genotipu i mogu se izvoditi samo s *FloatingPoint* genotipom, ti algoritmi su:

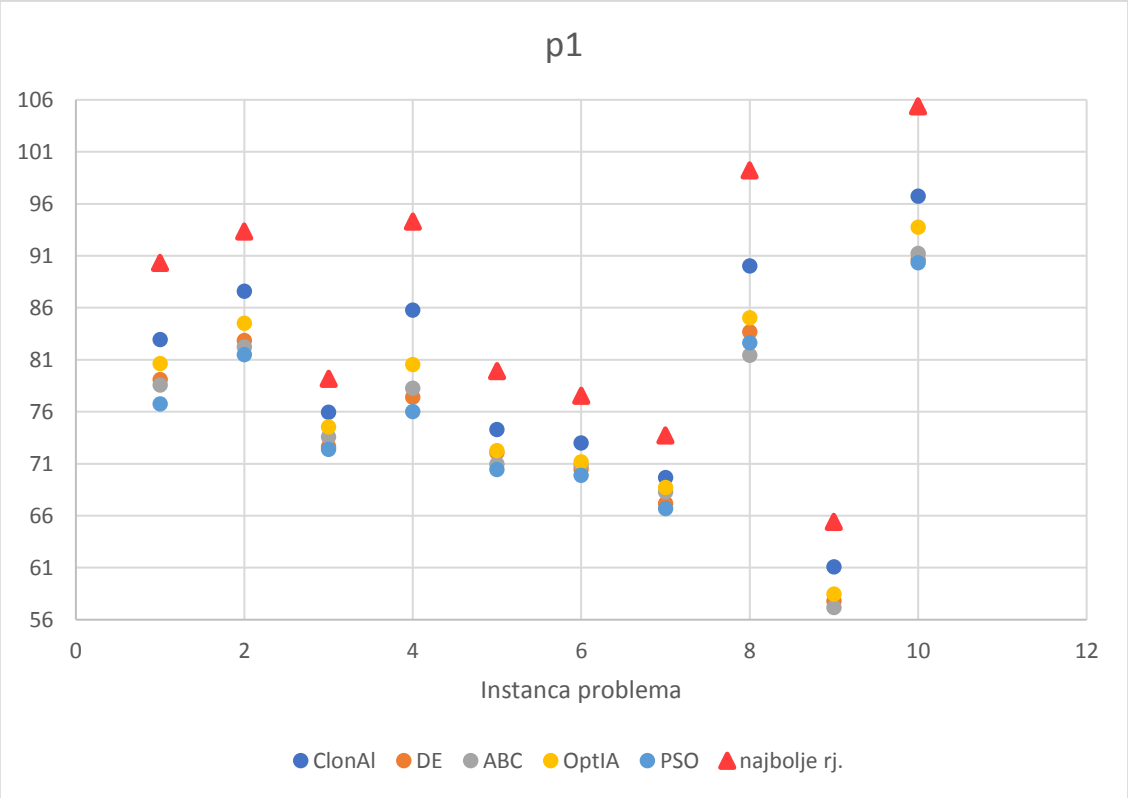
- *Clonal Selection* (CLONALG),
- *Differential Evolution* (DE),
- *Artificial Bee Colony* (ABC),
- *Immune Optimization* (optIA),
- *Particle Swarm Optimization* (PSO).

Ispitani su nad razredom problema „p1“ s jednako parametriziranim *FloatingPoint* genotipom kao za usporedbu *BitString* genotipa s *FloatingPointom*. Na slici 6.5 je prikazan graf i dobiveni rezultati te su trokutom označene najbolje vrijednosti dobivene s natjecanja kako bi se mogla usporediti uspješnost algoritama.

Rješenja dosta odstupaju od najboljih s natjecanja, a i *Steady State Tournament* algoritam s *BitString* genotipom dao je bolja rješenja. Točne vrijednosti prikazane su u tablici (Tablica 6.4).

Tablica 6.4: Algoritmi s FloatingPoint genotipom

funkcija	ClonAl	DE	ABC	OptIA	PSO	Najbolji
1	82,9488	79,1011	78,5517	80,6172	76,761	90,3235
2	87,5973	82,831	82,2175	84,516	81,4891	93,3318
3	75,9466	72,6456	73,5744	74,5429	72,3714	79,1587
4	85,7519	77,4063	78,2576	80,5298	76,0161	94,2709
5	74,282	72,0707	70,9570	72,2633	70,4269	79,8947
6	72,9753	70,4709	70,8803	71,1666	69,8751	77,5407
7	69,6507	67,1724	68,2940	68,7050	66,6767	73,7165
8	90,0069	83,6621	81,4117	85,0335	82,6291	99,2097
9	61,0859	57,7922	57,1557	58,4606	55,8999	65,4092
10	96,7381	90,6171	91,2040	93,7464	90,3183	105,3594



Slika 6.5: Algoritmi koji se pokreću samo s FloatingPoint genotipom

7. Zaključak

Problemi kombinatoričke optimizacije teški su problemi i sveprisutni u stvarnom životu. Problemi su različiti te činjenica da za svaki pojedini i postoji neki algoritam koji daje zaista odlična rješenja nije zadovoljavajuća. Problemi oko ugođavanja parametara ili korištenja različitih algoritama za pojedine zadatke nisu prihvatljivi te se pokušavaju riješiti. Ideja je pronaći algoritam koji će pronalaziti rješenja raznih problema jednako dobro, bez gubitka učinkovitosti, na općenitim problemima. Takav algoritam ne bi trebao ovisiti o karakteristikama pojedinog zadatka gdje bi se za svaki problem ovisno o svojstvima, parametar algoritma mijenjao. Također, u stvarnom svijetu problem je i vrijeme – rješenja trebaju biti dostupna što prije, što također ne smije utjecati na učinkovitost.

Natjecanje CBBOC bavi se upravo ovim problemima u optimizaciji. Iz tih su razloga stvorili generator različitih problema kako bi se svi algoritmi ovisni o svojstvima pojedinog zadatka kažnjavali. Način rangiranja algoritama na natjecanju zato uzima prosjek rezultata algoritma nad svim različitim problemima. Uz to svi su algoritmi imali samo određen konstantan broj evaluacija koji su mogli iskoristiti za pronalazak najboljeg rješenja.

U ovom radu isprobani su algoritmi iz ECF okruženja te su se pokazali većinom lošiji, s time da je potrebno još ispitivanja napraviti s obzirom na broj različitih razreda problema. Iz provedenih ispitivanja, *Steady State Tournament* algoritam pokazao se najboljim prilikom korištenja *BitString* genotipa.

S obzirom da se algoritmi iz ECF-a inače pokreću s velikim brojem evaluacija, činjenica da su sada bili ograničeni relativno vrlo malim brojem evaluacija, pokazala se na dobivenim rezultatima. No, to ograničenje postavljeno je upravo zbog imitiranja stvarnih situacija, što znači da se ti algoritmi možda ne bi koristili u takvim situacijama. Ukoliko se poveća broj evaluacija daju bolje rezultate.

Algoritmi P3 i CMA-VNS ostvarili su odlične rezultate na natjecanju i zato je ideja bila implementirati ih unutar ECF-a. Oba algoritma trenutno ovise o genotipu. Algoritam P3 zahtijeva *BitString* genotip, a CMA-VNS *FloatingPoint*. S obzirom da su za CMA-VNS algoritam ostavljena predefinirana pravila s natjecanja, dakle pravila određena nad problemima generiranim pomoću NK modela, potrebno je ispitati taj algoritam nad drugim te uvesti promjene u predefiniranim pravilima kako bi bila još općenitija. Algoritam nije usko vezan za probleme, osim što je trenutno implementiran za *BitString* genotip. Algoritam je inače osmišljen za diskretan prostor pretraživanja tako da se može proširiti, no ovisno o intervalu, vrijeme izvođenja moglo bi se povećati.

8. Dodatak

U radu je korišteno okruženje za evolucijsko računanje te algoritmi implementirani unutar okruženja. S obzirom da nisu dio rada, ali se koriste i često spominju kroz rad, kratki opis okruženja i postojećih algoritama stavljen je u dodatak.

8.1 ECF

ECF (*Evolutionary Computation Framework*) je okruženje za primjenu evolucijskog računanja. Evolucijsko računanje obuhvaća algoritme za globalnu optimizaciju inspirirane evolucijom u biologiji te predstavlja područje umjetne inteligencije i „mekog“ računarstva (engl. *soft computing*) koje se bavi proučavanjem takvih algoritama.

Okruženje je napisano C++ programskim jezikom te koristi biblioteku *boost C++*. Upute za preuzimanje, instalaciju te korištenje ECF-a nalaze se na stranicama ECF-a [14]. U nastavku su objašnjeni i opisani dijelovi ECF-a koji su korišteni u radu, više o mogućnostima ECF-a piše na stranicama.

Unutar okruženja postoje već implementirani algoritmi od kojih su neki spominjani i korišteni u radu. U nastavku je cijeli popis algoritama, a u idućem poglavlju algoritmi korišteni u radu su ukratko opisani.

Algoritmi dostupni za korištenje unutar ECF-a su:

- *Steady State Tournament* (SST),
- *Roulette Wheel* (RW),
- *Elimination*,
- *Particle Swarm Optimization* (PSO),
- *Differential Evolution* (DE),
- *Genetic Annealing* (GA),
- *Artificial Bee Colony* (ABC),
- *Clonal Selection* (CLONALG),
- *Immune Optimization* (optIA),
- *Evolution Strategy* (ES),
- *Random Search*.

ECF koristi genotipe kao strukture podataka koje predstavljaju jedinke populacije u algoritmu. Postoji *BitString*, *Binary*, *FloatingPoint*, *Permutation* i *Tree* genotip.

BitString je upravo to – niz bitova. Implementiran je kao vektor *bool* vrijednosti. *Binary* genotip prima parametre kao realni vektor dok interno dekodira vrijednosti u odgovarajuće binarne vrijednosti te nad njima vrši križanja i mutacije. *FloatingPoint* predstavlja vektor realnih brojeva. *Permutation* genotip predstavlja vektor indeksa (permutaciju indeksa). *Tree* predstavlja stablo čvorova gdje čvor maskira neki od primitiva – *Double*, *Int*, *Bool*, *Char* ili *String*.

Algoritmi koji se mogu koristiti neovisno o genotipu su *Steady State Tournament*, *Roulette Wheel*, *Evolution Strategy*, *Elimination*, *Genetic Annealing*, *Random Search*.

Ostali algoritmi mogu se koristiti samo uz *FloatingPoint* genotip (*Clonal Selection Algorithm*, *Immune Algorithm*, *Particle Swarm Optimization*, *Differential Evolution*, *Artificial Bee Colony*).

Odabir algoritma, genotipa te promjene njihovih parametara zadaje se kroz parametarsku (konfiguracijsku) datoteku koja se šalje kao argument programu. Datoteka je pisana XML formatom. Početni 'tag' je ECF, a unutar njega postoje tri bloka: *Algorithm*, *Genotype* i *Registry*.

U bloku *Algorithm* zadaje se algoritam koji će se izvršiti, ukoliko blok ne postoji, koristit će se *Steady State Tournament*. Ukoliko je zapisano više algoritama, koristi se prvi navedeni.

U *Genotype* bloku odabire se tip genotipa te se zadaju njemu karakteristični parametri – veličina, granice vrijednosti, broj decimala i slično.

U *Registry* bloku zapisuju se parametri koji nisu vezani za algoritam ni genotip, kao primjerice maksimalan broj evaluacija, dobrota koja ukoliko se dostigne može se optimizacija završiti, maksimalan broj generacija, te neki dodatni parametri eventualno vezani za problem. U ovom radu tako se koristi parametar datoteke koja sadrži instancu problema te broj instanci za optimizaciju ukoliko se želi provesti optimizacija nekoliko instanci unutar istog direktorija.

U nastavku je prikazan sadržaj cijele parametarske datoteke u kojoj se vide i parametri algoritama korištenih za ispitivanje algoritama ECF-a nad problemima generiranim za natjecanje s *BitString* genotipom.

```
<ECF>
     <Algorithm>
          <SteadyStateTournament>
               <Entry key="tsize">3</Entry>
          </SteadyStateTournament>
          <Elimination>
```

```
                <Entry key="gengap">0.6</Entry>
                <Entry key="selpressure">10</Entry>
          </Elimination>
          <EvolutionStrategy>
                <Entry key="lambda">4</Entry>
                <Entry key="mu">1</Entry>
                <Entry key="rho">1</Entry>
                <Entry key="selection">plus</Entry>
          </EvolutionStrategy>
          <RouletteWheel>
                <Entry key="crxprob">0.7</Entry>
                <Entry key="selpressure">10</Entry>
          </RouletteWheel>
          <RandomSearch/>
     </Algorithm>
     <Genotype>
                <BitString>
                     <Entry key="size">136</Entry>
                </BitString>
     </Genotype>
     <Registry>
          <Entry key="problem.file">p2/00000.txt</Entry>
          <Entry key="population.size">500</Entry>
          <Entry key="mutation.indprob">0.5</Entry>
          <Entry key="term.eval">4624</Entry>
          <Entry key="term.stagnation">50000</Entry>
          <Entry key="term.fitnessval">1000</Entry>
     </Registry>
</ECF>
```

Parametrizacija 1: Primjer konfiguracijske datoteke

Algoritmi koji koriste samo *FloatingPoint* i korišteni su u ispitivanju također su opisani, a parametri algoritama korišteni prilikom ispitivanja prikazani su u nastavku. Za algoritme neovisne o genotipu, prilikom pokretanja s *FloatingPoint* genotipom, korišteni su isti parametri kao kod pokretanja s *BitString* genotipom.

```
<ECF>
```

```
    <Algorithm>
        <Clonalg>
            <Entry key="beta">1</Entry>
            <Entry key="c">0.2</Entry>
            <Entry key="d">0</Entry>
            <Entry key="n">100</Entry>
        </Clonalg>
        <OptIA>
            <Entry key="c">0.2</Entry>
            <Entry key="dup">5</Entry>
            <Entry key="elitism">0</Entry>
            <Entry key="tauB">100</Entry>
        </OptIA>
        <ParticleSwarmOptimization>
            <Entry key="weightType">0</Entry>
            <Entry key="weight">0.8</Entry>
            <Entry key="maxVelocity">100</Entry>
        </ParticleSwarmOptimization>
        <DifferentialEvolution>
            <Entry key="F">1</Entry>
            <Entry key="CR">0.9</Entry>
        </DifferentialEvolution>
        <ArtificialBeeColony>
            <Entry key="limit">300</Entry>
            <Entry key="elitism">1</Entry>
        </ArtificialBeeColony>
    </Algorithm>
    …
</ECF>
```

8.2 Algoritmi

Steady State Tournament

Algoritam je *k*-turnirski po vrsti selekcije, također i eliminacijski. Parametar *k* zadaje se u parametrizaciji (*tsize*), u ovom radu korištena je 3-turnirska selekcija.

Ukoliko nije zadan algoritam u parametarskoj datoteci, koristi se upravo *Steady State Tournament*.

```
jedna generacija
     ponavljaj veličina_populacije puta
          turnir = k slučajnih jedinki iz populacije
          odaberi najgoru iz turnira
          roditelji = 2 slučajne jedinke iz turnira
          dijete = križaj
          mutiraj(dijete)
```

Pseudokod 8.1: Steady State Tournament

Roulette Wheel

Uobičajeni genetski generacijski algoritam kod kojeg se selekcija jedinki određuje proporcionalno dobroti jedinke. Parametri algoritma su vjerojatnost križanja (*crxprob*) te selekcijski pritisak – koliko je najbolja jedinka bolja od najgore.

```
jedna generacija
     jedinke za novu generaciju proporcionalno dobroti
     stvaranje nove generacije(kopije);
     noCrx = (velicina_populacije) * crxprob / 2;
     ponavljaj (noCrx puta)
          slučajno odaberi dva roditelja
          križaj
          zamijeni roditelje djecom
     mutiraj novu generaciju
```

Pseudokod 8.2: Roulette Wheel

Evolution Strategy

Implementacija u ECF-u je $(\mu/\rho +/, \lambda)$ tip evolucijske strategije. Podržana je i *plus* i *zarez* strategija. *Plus* strategije su strategije kod kojih se selekcija za novu populaciju vrši i nad skupinom roditelja (μ) i nad skupinom djece (λ). Imaju svojstvo elitizma pa će se najbolja jedinka roditelj pojaviti i u novoj populaciji. Kod ovog specifičnog tipa u slučaju strategije *plus* od μ roditelja stvara se λ djece koja se ubacuju u skup roditelja te se bira najboljih μ jedinki za roditelje sljedeće generacije, s time da se ρ parametar koristi prilikom izbora roditelja, te može primjerice označavati broj roditelja koji se biraju prilikom stvaranja djece. U ECF-u ρ parametar može imati vrijednosti jedan ili dva, gdje jedan označava da će se nad roditeljem provesti mutacija, a dva da će se izvršiti križanje roditelja.

Kod *zarez* strategija selekcija za novu populaciju vrši se samo nad skupinom djece (λ). U ovom je slučaju moguće izgubiti do sad najbolje jedinke, ali iz tog razloga manje zapinje u lokalnim optimumima. Za navedeni tip s tri parametra selekcija se vrši tako da se od μ roditelja stvara λ djece, a od djece se bira μ jedinki za roditelje sljedeće generacije gdje ρ ima isto značenje kao i za *plus* strategiju.

U ECF-u s obzirom da se veličina populacije definira neovisno o algoritmu, implementacija algoritma pokreće više ES roditeljskih populacija, pa tako parametar *population.size* treba biti višekratnik parametra μ. Broj populacija bit će $\frac{population.size}{mu}$.

```
jedna generacija
     ponavljaj za svaku subpopulaciju
          dodaj μ jedinki u roditelje
          stvori λ djece od slučajnih ρ roditelja za svako
          ako zarez strategija
               roditelji = μ najbolje djece
          inače // plus strategija
               roditelji=μ najboljih od djece i roditelja
```

Pseudokod 8.3: Evolution Strategy

Elimination

Implemetiran je eliminacijski algoritam koji koristi obrnutu proporcionalnost dobrote prilikom selekcije. Kroz konfiguracijsku datoteku zadaje se parametar *gengap* koji označava postotak populacije koji će se eliminirati (ukloniti). Drugi parametar je selekcijski pritisak – koliko je najgora jedinka lošija od najbolje.

```
jedna generacija
     ukloni (gengap * pop_size) jedinki
          pomoću obrnute proporcionalne dobrote
     stvori nove jedinke križanjem(slučajnim odabirom)
     provedi mutaciju nad generacijom
```

Pseudokod 8.4: Elimination

Clonal Selection (CLONALG)

Algoritam inspiriran teorijom klonirajuće selekcije koja objašnjava rad limfocita u imunološkom sustavu potaknut određenim antigenima koji napadaju tijelo. Parametar *beta* postotak je koji određuje koliko će se puta klonirati određeno

antitijelo, *c* označava postotak mutacije, *d* označava postotak populacije koji će se u svakoj generaciji slučajno generirati kako bi zamijenio najlošije jedinke, a *n* je broj antitijela koji se generira u svakoj generaciji. Kod proporcionalne verzije kloniranja broj klonova pojedinog antitijela ovisi o njegovoj dobroti, a parametar *selectionScheme* ako je *CLONALG1* ostavlja po jedan (najbolji) klon pojedinog antitijela. Pretpostavljena vrijednost varijante kloniranja je *proportional*, a selekcijske *CLONALG1*, te su navedene korištene u ispitivanju, s obzirom da nisu navedene posebno u konfiguracijskoj datoteci.

```
jedna generacija
     kloniraj najboljih n antitijela
     mutiraj klonirane(c)
     ostavi najboljih (1-d)*veličina populacije
     generiraj novih d*veličina populacije
     zamijeni staru novom populacijom
```

Pseudokod 8.5: Clonal selection algorithm

Immune Optimization (optIA)

Također inspiriran imunološkim sustavom. Parametar *c* je postotak mutacije, *dup* je broj klonova za svako antitijelo, a *tauB* je najveći broj generacija u kojima jedinka nije postigla napredak. Također, može se uključiti elitizam parametrom *elitism*.

```
jedna generacija
     kloniraj svako antitijelo dup puta
     mutiraj antitijela (c)
     ažuriraj starost antitijela
     ako je elitizam
          sačuvaj antitijelo bez obzira na starost
     inače
          ako je starije od tauB
               zamijeni novim antitijelom
     zadrži pop.size najboljih klonova
     ako ima manje od pop.size
          dodaj nova antitijela
     zamijeni staru populaciju novom
```

Pseudokod 8.6: Imune Optimization

Particle Swarm Optimization (PSO)

Algoritam koristi populaciju čestica koje se kreću u prostoru pretraživanja prema jednostavnoj matematičkoj formuli za brzinu i poziciju čestice. Na kretanje čestice utječu lokalni optimumi, ali i najbolje poznato rješenje pronađeno od strane drugih čestica. Parametar *maxVelocity* određuje najveću dozvoljenu brzinu čestice, a *weightType* određuje je li jačina inercije promjenjiva ili konstantna, a *weight* njenu inicijalnu vrijednost.

```
jedna generacija
     ažuriraj najbolju jedinku u povijesti
     za svaku česticu
          nađi najbolju česticu u susjedstvu
          izračunaj brzinu čestice uz ograničenja
          ažuriraj poziciju čestice uz ograničenja
          evaluiraj česticu
```

Pseudokod 8.7: Partice Swarm Optimization

Differential Evolution (DE)

Algoritam koristi agente koji se kreću prostorom pretraživanja matematičkom formulom kojom se kombiniraju pozicije postojećih agenata iz populacije. Ukoliko nova pozicija agenta predstavlja napredak, pohranjuje se u populaciju. Parametar *CR* je postotak križanja, a *F* konstanta skaliranja prilikom generiranja *donora*.

```
jedna generacija
     stvori donor za svaku jedinku
     križaj donora s odgovarajućom jedinkom
     za svaku jedinku
          ako je donor bolji
               zamijeni jedinku donorom
```

Pseudokod 8.8: Differential Evolution

Artificial Bee Colony (ABC)

Algoritam je inspiriran ponašanjem pčela u potrazi za hranom. Kolonija pčela u potrazi za hranom (rješenja) dijeli se u tri grupe pčela: zaposlene pčele, pčele koje promatraju, te pčele izviđači. Parametri algoritma su *limit* i *elitism*; *limit* označava najveći broj generacija za svaku jedinku bez napretka, a elitizam se može isključiti ili uključiti.

```
jedna generacija
     zaposlene pčele
          stvori novi izvor hrane
     pčele koje gledaju
          odabere izvor hrane ovisno o dobroti
          za svaki odabrani izvor hrane
               stvara novi izvor hrane
     pčele izviđači
          za svaki izvor hrane
               ako je prešao ograničenje
                    zamijeni izvor hrane novim
```

Pseudokod 8.9: Artificial Bee Colony

9. Literatura

[1] Wikipedia, Combinatorial optimization, 2004., https://en.wikipedia.org/wiki/Combinatorial_optimization

[2] Skorin-Kapov, N., Uvod u optimizacije, Prezentacija s kolegija „Heurističke metode optimizacija“, 2014.

[3] Wikipedia, Approximation algorithm, 2004., https://en.wikipedia.org/wiki/Approximation_algorithm

[4] Tauritz, D.R., Swan J., Goldman, B.W., 1st Combinatorial Black-Box Optimization Competition (CBBOC), 2015., http://web.mst.edu/~tauritzd/CBBOC/GECCO2015/

[5] Wikipedia, Epistasis, 2016., https://en.wikipedia.org/wiki/Epistasis

[6] Goldman, B.W., Utilities - Problem Class Generator, Rank Entries, 2016., https://github.com/cbboc/Utilities

[7] Goldman, B.W., C++ CBBOC API, 2016., https://github.com/cbboc/cpp

[8] Tauritz, D.R., Swan J., Goldman, B.W., Combinatorial Black Box Optimization Competition (CBBOC) @ GECCO 2015 - Using the API, 2015., http://web.mst.edu/~tauritzd/CBBOC/GECCO2015/cbboc-readme.txt

[9] Goldman, B.W., Punch, W.F., Parameter-less Population Pyramid, CBBOC, GECCO, 2015.

[10] Xue, F., Shen, G.Q.P., CMA-VNS: A Short Description, CBBOC, GECCO, 2015.

[11] Auger, A., Hansen, N., Tutorial CMA-ES — Evolution Strategies and Covariance Matrix Adaptation, GECCO, Amsterdam, Netherlands, 2013.

[12] Wikipedia, CMA-ES, 2006., https://en.wikipedia.org/wiki/CMA-ES

[13] Ćosić, M., Stohastička numerička optimizacija u okruženju za evolucijsko računanje, Završni rad, FER, 2012.

[14] Jakobović, D., ECF - Evolutionary Computation Framework, http://ecf.zemris.fer.hr/

[15] Krček, M., Optimizacija Booleovih funkcija za kriptografske postupke, Završni rad, FER, 2015.

Problemi i algoritmi kombinatoričke optimizacije

Sažetak

Rad opisuje kombinatoričke optimizacijske probleme i razloge korištenja heurističkih metoda u njihovom rješavanju. Natjecanje CBBOC ponudilo je nekolicinu algoritama koji su se pokazali vrlo uspješnima. S natjecanja su preuzeti generirani problemi i objašnjen je NK model korišten za generiranje problema. Postojeći algoritmi iz ECF okruženja ispitani su nad problemima s natjecanja, u radu su opisani rezultati te je provedena usporedba s rezultatima natjecanja. S obzirom na rezultate na natjecanju, unutar ECF okruženja implementirana su dva algoritma pobjednika – algoritam P3 i CMA-VNS. Oba algoritma su detaljno objašnjena te su napomenute određene prilagodbe za ECF okruženje.

Ključne riječi: kombinatorički optimizacijski problemi; heurističke metode; NK model; ECF; algoritam P3; algoritam CMA-VNS

Combinatorial optimization problems and algorithms

Abstract

This thesis describes combinatorial optimization problems and explains why heuristics are often used for solving them. On a competition named CBBOC there were several algorithms that proved to be very successful in solving this problems. For the competition, combinatorial problems were generated using NK model that is explained in the thesis. Existing algorithms from ECF framework were tested on generated combinatorial problems from the competition. The obtained results are shown in graphs, analyzed and compared to results from the competition. Considering the results of the competition, two winner algorithms are implemented in ECF environment – algorithm P3 and CMA-VNS. Both of the algorithms are explained in the thesis and specific adjustments for the ECF environment are mentioned.

Keywords: combinatorial optimization problems; heuristics; NK model; ECF; algorithm P3; algorithm CMA-VNS