SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5898

Radna okruženja za evolucijsko računanje

Hrvoje Spajić

Zagreb, lipanj 2019.

Sadržaj

1. Uvod **1**

2. Evolucijsko računarstvo **2**
2.1. Evolucijski algoritmi . . . 2
2.1.1. Operatori mutacije . . . 2
2.1.2. Operatori selekcije . . . 3
2.1.3. Operatori križanja . . . 4
2.2. Algoritmi rojeva . . . 5
2.2.1. Algoritam kolonije mrava . . . 5

3. Radna okruženja za evolucijsko računarstvo **6**
3.1. Radna okruženja za evolucijske algoritme . . . 6
3.1.1. Radna okruženja za genetske algoritme . . . 6
3.1.2. Radna okruženja za genetsko programiranje . . . 8
3.2. Radna okruženja za algoritme roja . . . 10
3.2.1. Radna okruženja za optimizaciju kolonije mrava . . . 10
3.3. Memoizacija . . . 12
3.4. Paralelizacija . . . 12
3.4.1. Paralelizacija na razini algoritma . . . 12
3.4.2. Paralelizacija na razini jedne iteracije algoritma . . . 13
3.5. Nadgledanje međurezultata . . . 13

4. Optimizacijski problemi **14**
4.1. Jednokriterijske i višekriterijske optimizacije . . . 14
4.2. Diskretni i kontinuirani optimizacijski problemi . . . 14
4.3. Lokalni i globalni optimum . . . 14
4.4. Meka i tvrda ograničenja . . . 15
4.5. Primjeri optimizacijskih problema . . . 15
4.5.1. Schwefelova funkcija . . . 15
4.5.2. Problem trgovačkog putnika . . . 15
4.5.3. Problem pronalaska odgovarajućeg regularnog izraza . . . 16
4.5.4. Problem naprtnjače . . . 16
4.5.5. Problem kompozicije ciljne slike pravokutnicima u tonovima sive boje . . . 17

5. Praktični rad **18**
5.1. Radno okruženje *GA* 18
5.1.1. Rješavanje problema kompozicije ciljne slike pravokutnicima u tonovima sive boje 18
5.2. Radno okruženje *pyevolve* 24
5.2.1. Rješavanje problema naprtnjače 24

6. Učinkovitost radnih okruženja **30**
6.1. Utjecaj paralelizacije 30
6.2. Utjecaj memoizacije 32
6.3. Analiza rezultata 34

7. Zaključak **35**

Literatura **36**

1. Uvod

Evolucijsko računarstvo pokušava riješiti probleme koje egzaktni algoritmi ne mogu zbog prevelike računske složenosti.

Radna okruženja za evolucijsko računarstvo pokušavaju ubrzati proces rješavanja problema pružajući korisniku već napisane općenite funkcionalnosti evolucijskog računarstva. Prednost razvijanja radnog okruženja u odnosu na pisanje algoritama "od nule" je, osim uštede vremena, i istestiranost radnog okruženja. Što veći broj ljudi koristi isti softver, veća je šansa pronalaženja grešaka.

Kako nije moguće pokriti sve probleme, radna okružena moraju biti dovoljno prilagodljiva da korisnik može sam definirati nove operatore prilagođene njegovom tipu problema.

Na korisnost radnih okruženja utječu i tehnike ubrzanja koje pruža te tehnike detaljnije analize problema.

U ovom radu opisani su neki optimizacijski problemi, algoritmi evolucijskog računarstva te je prikazana uporaba i usporedba više radnih okruženja za evolucijsko računarstvo. Prikazan je i utjecaj paralelizacije i memoizacije na brzinu izvođenje algoritama.

2. Evolucijsko računarstvo

Evolucijsko računarstvo je grana umjetne inteligencije, a proučava prirodom inspirirane algoritme za rješavanje optimizacijskih problema. Svoju primjenu nalaze u rješavanju problema visoke složenosti kod kojih "klasični" deterministički algoritmi nisu dovoljno dobri[13].

Kako ne pretražuju cijeli prostor mogućih rješenja, već se heuristički usmjeravaju, ne garantiraju pronalazak optimalnog rješenja.

Evolucijsko računarstvo dijeli se na:

1. evolucijske algoritme,

2. algoritme rojeva,

3. ostale (nesvrstane) algoritme.

Neki nesvrstani algoritmi su diferencijska evolucija (engl., *Differential Evolution*), umjetni imunološki algoritmi (engl., *Artificial immune system*) i algoritam kolonije pčela (engl., *Particle swarm optimization*). Nisu korišteni u ovom radu te nisu detaljnije opisani.

2.1. Evolucijski algoritmi

Skupina algoritama koja se temelji na Darwinovoj teoriji evolucije, a kretanje kroz prostor rješenja na operacijama mutacije, selekcije i križanja nad jednom populacijom rješenja.

Evolucijski algoritmi dijele se na:

- genetski algoritme,
- genetsko programiranje,
- evolucijske strategije,
- evolucijsko programiranje,
- klasifikatorske sustave.

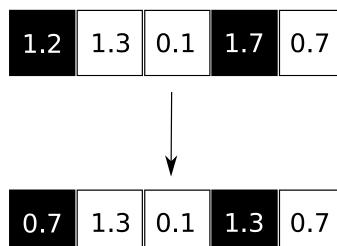
2.1.1. Operatori mutacije

Operatori mutacije mijenjaju pojedinu jedinku populacije nekom vjerojatnošću mutacije p_m. Operatorom se osigurava genetska raznolikost, odnosno dobivanje jedinka čija svojstva ne moraju biti već prisutna u populaciji.

Jednolika mutacija

Ako je jedinka zadana kao niz cijelih ili realnih brojeva, jednolika mutacija mijenja vrijednost nasumično odabranom vrijednošću iz mogućeg prostora pretraživanja. Potrebno je zadati i vjerojatnost mutacije pojedinog broja u nizu.

Slikom 2.1 prikazan je primjer mutacije jedinke zadane kao niz od pet realnih brojeva, kod kojih je svaki broj u rasponu od 0 do 2, te su mutirani prvi i treći broj.

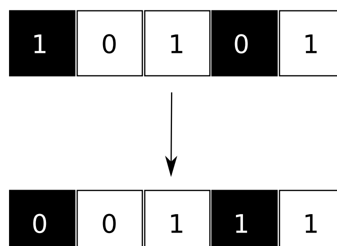


Slika 2.1: Mutiranje prvog i trećeg broja u nizu jednolikom mutacijom.

Mutacija izmjenom bitova

Ako je jedinka zadana kao niz bitova, mutacija izmjenom bitova (engl., *Bit flipping mutation*) nekom vjerojatnošću p_i mijenja bit na određenom položaju.

Slikom 2.2 prikazan je primjer mutacije prvog i trećeg bita u nizu.



Slika 2.2: Mutiranje prvog i trećeg bita u nizu mutacijom izmjenom bitova.

2.1.2. Operatori selekcije

Operatorom selekcije odabiru se jedinke za primjenu operatora križanja. Svojstvo operatora selekcije koje osigurava da se određen broj najboljih jedinki sigurno nalazi u sljedećoj generaciji naziva se elitizam.

Neke moguće tehnike selekcije su turnirska, eliminacijska, te elitizam.

Turnirska selekcija

Turnirska selekcija simulira niz turnira u kojima se pobjednici dodaju u skup za razmnožavanje. Broj sudionika turnira određuje i brzinu konvergencije jer kod velikog broja sudionika veća je vjerojatnost da će se u sljedeću populaciju dodavati samo najbolje jedinke prethodne populacije [7].

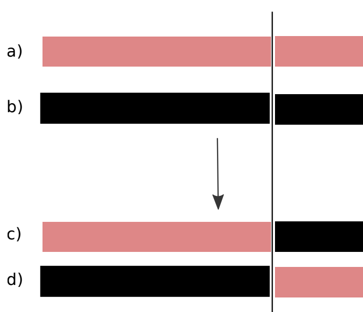
2.1.3. Operatori križanja

Operatorom križanja kombiniraju se dvije ili više jedinki, te se stvara jedna ili više novih jedinki.

Neke moguće tehnike križanja su križanje s jednom točkom presjeka, križanje s više točaka presjeka te jednoliko križanje.

Križanje s jednom točkom presjeka

Križanje s jednom točkom presjeka (engl., *Single point crossover*) je križanje kod kojeg se nasumično izabere jedno mjesto na kromosomu te se dijelovi lijevi dio prve jedinke spoji s desnim dijelom druge i obratno. Ideja je prikazana slikom 2.3.



Slika 2.3: Križanje s jednom točkom presjeka. a) i b) su roditelji, a c) i d) djeca.

Genetski algoritmi i genetsko programiranje

Genetski algoritmi temeljeni su na teoriji evolucije, a opisao ih je John Holland. Djeluju nad populacijom rješenja te kroz križanje, mutaciju i selekciju pokušavaju usmjeriti populaciju prema optimalnom rješenju. Proces križanja mutacije i selekcije ponavlja se dok nije pronađeno zadovoljavajuće rješenje ili dok nije napravljen najveći broj ponavljanja. Jedinke genetskog programiranja nepromjenjive su strukture.

Za razliku od genetskih algoritama, jedinke genetskog programiranja promjenjive su duljine i čest prikaz jedinki je pomoću podatkovne strukture stablo.

Osnovni princip rada genetskih algoritama opisan je pseudokodom 2.1.

Izvorni kod programa 2.1: Pseudokod genetskog algoritma

```
1   Stvaranje inicijalne populacije
2   Evaluacija
3   Ponavljaj:
4       Selekcija
5       Krizanje
6       Mutacija
7   Dok nije zadovoljen uvjet zaustavljanja
```

2.2. Algoritmi rojeva

Populacijski algoritmi temeljeni na proučavanju ponašanja rojeva u prirodi. Populacija se sastoji od jedinki koje pretražuju prostor rješenja temeljem svog prethodnog iskustva te iskustva okolnih jedinki. Pri određivanju smjera pretrage težina osobnog je osobni faktor (engl., *Individual factor*), a težina iskustva okolnih jedinki socijalni faktor (engl., *Social factor*).

Neki algoritmi rojeva su:

- algoritam roja čestica (engl., *Particle swarm optimization*)
- algoritam kolonije mrava (engl., *Ant colony optimization*)
- algoritam kolonije pčela (engl., *Bee colony optimization*)

2.2.1. Algoritam kolonije mrava

Algoritam kolonije mrava temeljen je na ponašanju kolonije mrava u prirodi. Uočeno je da mravi uspješno pronalaze najkraći put do izvora hrane jer prilikom kretanja za sobom ostavljaju feromonski trag koji im pomaže u odabiru puta, odnosno veća je vjerojatnost da će pratiti jači feromonski trag.

Feromonski tragovi su "hlapljivi", odnosno intenzitet im opada svakom iteracijom, ali i rastu onim putovima kojim su prošli mravi u iteraciji.

Pseudokod je dan kodom 2.2.

Izvorni kod programa 2.2: Pseudokod algoritma kolonije mrava

```
1   Dok nije kraj:
2       ponovi za svakog mrava:
3           stvori rjesenje
4           vrednuj rjesenje
5       kraj ponovi
6       odaberi podskup mrava
7       ponovi za izabrane mrave
8           azuriraj feromonske tragove
9       kraj ponovi
10      ispari feromonske tragove
11  kraj ponavljanja
```

3. Radna okruženja za evolucijsko računarstvo

Glavni motivi za razvoj radnog okruženja za evolucijsko računarstvo su omogućiti bržu i sigurniju implementaciju algoritama te uštedu vremena razvoja.

Svako radno okruženje omogućuje kako i korištenje predefiniranih tipova rješenja, operacija tako i njihovu prilagodbu specifičnu za problem koji pokušava riješiti.

Dodatna svojstva koju nude su memoizacija, paralelizacija i nadgledanje međurezultata (engl., *monitoring*).

3.1. Radna okruženja za evolucijske algoritme

3.1.1. Radna okruženja za genetske algoritme

Primjeri radnih okruženja koji podržavaju genetske algoritme su:

- *DEAP*,
- *pyevolve*,
- *GA - R*
- *Jenetics*

Za demonstriranje osnovnog principa rada sličnih radnih okruženja, korišten je paket *GA* za programski jezik R [9]. Na primjeru problema definiranog u poglavlju 4.5.1.

Priprema potrebnih paketa

Paket *GA* pruža osnovu za realizaciju genetskih algoritama. Uvoz paketa dan je izvornim kodom 3.1.

Izvorni kod programa 3.1: Priprema paketa

```
1 install.packages("GA")
2 library(GA)
```

Definiranje funkcije dobrote

Implementacije funkcije dobrote dana je izvornim kodom 3.2.

Izvorni kod programa 3.2: Definiranje funkcije dobrote

```
1
2 schwefel2d <- function(x1, x2) {
3  return (837.9658 -x1*sin(sqrt(abs(x1))) - x2*sin(sqrt(abs(x2))))
4 }
```

Pokretanje algoritma

Uz opis parametara, izvedba je trivijalna. Detaljniji opis izveden je u praktičnom dijelu. Naprednije korisnike upućuje se na dokumentaciju [10]. Pokretanje algoritma ostvareno je izvornim kodom 3.3.

- **fitness** - deklariranje funkcije čiji se optimum traži
- **lower** - minimalne vrijednosti rješenja
- **upper** - maksimalne vrijednosti rješenja
- **popSize** - veličina populacije
- **maxiter** - najveći broj generacija
- **crossover** - operator križanja
- **mutation** - operator mutacije
- **selection** - operator selekcije
- **monitor** - funkcija nadgledanja rješenja
- **run** - najveći broj generacija bez promjene dobrote

Izvorni kod programa 3.3: Pokretanje algoritma

```
1
2  > GA <- ga(type = "real-valued",
3  +          fitness = function(x) -schwefel2d(x[1], x[2]),
4  +          lower = c(-500,-500),
5  +          upper = c(500,500),
6  +          popSize = 50,
7  +          maxiter = 1000,
8  +          crossover = gareal_laCrossover,
9  +          mutation = gareal_raMutation,
10 +          selection = gareal_lsSelection,
11 +          monitor = NULL,
12 +          run = 100
13 +          )
```

Provjera rješenja

Rješenje se nalazi u varijabli *GA* koja je objekt razreda tipa *S4*, a sadrži informacije o parametrima provedenog algoritma, te o pronađenom rješenju. Ispisano rješenje dano je izvornim kodom 3.4.

Izvorni kod programa 3.4: Provjera rješenja

```
GA settings:
Type                  =  real-valued
Population size       =  50
Number of generations =  1000
Elitism               =  2
Crossover probability =  0.8
Mutation probability  =  0.1
Search domain =
        x1   x2
lower -500 -500
upper  500  500

GA results:
Iterations             = 392
Fitness function value = -0.0001307039
Solution =
           x1       x2
[1,] 420.9459 420.9865
```

Iz kojeg iščitavamo rješenje kao točku (420.9459, 420.9865), pronađeno u 392 iteracije, uz vrijednost funkcije gubitka -0.0001307039.

3.1.2. Radna okruženja za genetsko programiranje

Primjeri radnih okruženja koji podržavaju genetsko programiranje su:

- *DEAP*,
- *gramEvol*,
- *Pyvolution*,
- *Jenetics*

Za demonstriranje osnovnog principa rada sličnih radnih okruženja, poslužit će paket *gramEvol* za programski jezik R [8]. Na primjeru problema definiranog u poglavlju 4.5.3. Implementacija primjera preuzeta je sa službene *web* stranice[6].

Priprema potrebnih paketa

Za početak dohvaćeni su i uvezeni potrebni paketi. Paket *gramEvol* pruža osnovu za genetsko programiranje, dok paket *rex* pruža mogućnost generiranja gramatike. Priprema paketa implementirana je izvornim kodom 3.5.

Izvorni kod programa 3.5: Priprema potrebnih paketa

```
install.packages("gramEvol")
install.packages("rex")
library(gramEvol)
library(rex)
```

Definiranje cilja traženog regularnog izraza

Postavljen je skup nizova koji pripadaju regularnom jeziku i skup nizova koji ne pripadaju regularnom jeziku. Definiranje cilja ostvareno je izvornim kodom 3.6.

Izvorni kod programa 3.6: Definiranje cilja

```
1 matching <- c("1", "11.1", "1.11", "+11", "-11", "-11.1")
2 non.matching <- c("a", "1.", "1..1", "-.1", "-", "1-", "1.-1",
3 ".-1", "1.-", "1.1.1", "", ".", "1.1-", "11-11")
```

Definiranje funkcije dobrote

Nakon toga definirana je funkcija dobrote tako da zbraja broj nizova u kojima se podudaranje događa, a treba se dogoditi, i broj nizova u kojima se podudaranje ne događa, a ni ne treba se dogoditi. Implementacija je dana izvornim kodom 3.7.

Izvorni kod programa 3.7: Definiranje funkcije dobrote

```
1 re.score <- function(re) {
2     score <- sum(sapply(matching, function(x) grepl(re, x))) +
3     sum(sapply(non.matching, function(x) !grepl(re, x)))
4     return (length(matching) + length(non.matching) - score)
5 }
6
7 fitfunc <- function(expr) re.score(eval(expr))
```

Definiranje gramatike koja stvara rješenja

Implementacija dana izvornim kodom 3.8, a provjera izvornim kodom 3.9.

Izvorni kod programa 3.8: Definiranje gramatike koja stvara rješenja

```
1  grammarDef <- CreateGrammar(list(
2    re = grule(rex(start, rules, end)),
3    rules = grule(rule, .(rule, rules)),
4    rule = grule(numbers, ".", or("+", "-"), maybe(rules)))
5  )
```

Izvorni kod programa 3.9: Provjera zadane gramatike

```
1 > grammarDef
2
3 <re> ::= rex(start, <rules>, end)
4 <rules> ::= <rule> | <rule>, <rules>
5 <rule> ::= numbers | "." | or("+", "-") | maybe(<rules>)
```

Pokretanje algoritma

Pokretanje algoritma dano je izvornim kodom 3.10.

Izvorni kod programa 3.10: Pokretanje algoritma

```
1  > GrammaticalExhaustiveSearch(grammarDef, fitfunc, max.depth = 7,
2  terminationCost = 0)
```

Provjera rješenja

Ispis rješenja pretrage dan je izvornim kodom 3.11.

Izvorni kod programa 3.11: Rezultati pretrage

```
1  GE Search Results:
2    Expressions Tested: 6577
3    Best Chromosome:    0 1 3 0 2 1 3 1 0 0 1 1 0 0 3 0 0
4    Best Expression:    rex(start, maybe(or("+", "-")),
5      maybe(numbers, "."), numbers, maybe(numbers), end)
6    Best Cost:          0
```

Što zapravo znači da je dobiveno rješenje u izvornom kodu 3.12.

Izvorni kod programa 3.12: Ispis rješenja

```
1  > rex(start, maybe(or("+", "-")), maybe(numbers, "."), numbers, maybe(numbers), end)
2  ^(?:(?:\+|-))?(?:[[:digit:]]+\.)?[[:digit:]]+(?:[[:digit:]]+)?$
```

3.2. Radna okruženja za algoritme roja

Neki od pronađenih radnih okruženja su:

- *inspyred*
- *DEAP*

3.2.1. Radna okruženja za optimizaciju kolonije mrava

Za demonstriranje osnovnog principa rada korištena je knjižnica *inspyred* [3]. Riješen je problem trgovačkog putnika opisan u poglavlju 4.5.2. Knjižnica je napisana za programski jezik *Python*. Implementacija rješenja je u potpunosti preuzeta s [3].

Postavljanje knjižnice

Knjižnicu je potrebno instalirati i uvesti u program. Kod 3.13 prikazuje instalaciju preko linux ljuske. Kod 3.14 prikazuje uvoz knjižnice u program.

Izvorni kod programa 3.13: Instalacija knjižnice preko linux ljuske

```
1  $ pip install inspyred
```

Izvorni kod programa 3.14: Uvoz knjižnice

```
from random import Random
from time import time
import math
import inspyred
```

Definiranje točaka

U izvornom kodu 3.15 definirano je 10 točaka te udaljenosti između njih.

Izvorni kod programa 3.15: Postavljanje knjižnice

```
points = [(110.0, 225.0), (161.0, 280.0), (325.0, 554.0), (490.0, 285.0),
        (157.0, 443.0), (283.0, 379.0), (397.0, 566.0), (306.0, 360.0),
        (343.0, 110.0), (552.0, 199.0)]

weights = [[0 for _ in range(len(points))] for _ in range(len(points))]

for i, p in enumerate(points):
    for j, q in enumerate(points):
        weights[i][j] = math.sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)
```

Definiranje problema i postavljanje parametara

Knjižnica nudi već predefiniran problem, odnosno funkciju gubitka i parametre. Implementacija dana je izvornim kodom 3.16. Za detalje pogledati dokumentaciju [3].

Izvorni kod programa 3.16: Definiranje problema i postavljanje parametara

```
problem = inspyred.benchmarks.TSP(weights)
ac = inspyred.swarm.ACS(prng, problem.components)
ac.terminator = inspyred.ec.terminators.generation_termination
```

Pokretanje algoritma

Pokretanje algoritma s veličinom populacije 10 i najvećim brojem generacija 50. Algoritam pokrenut izvornim kodom 3.17.

Izvorni kod programa 3.17: Pokretanje algoritma

```
final_pop = ac.evolve(generator=problem.constructor,
    evaluator=problem.evaluator,
    bounder=problem.bounder,
    maximize=problem.maximize,
    pop_size=10,
    max_generations=50)
```

Pregled rezultata

Programski kod 3.18 ispisuje rezultat 3.19.

Izvorni kod programa 3.18: Ispis rezultata

```
1 best = max(ac.archive)
2 print('Best Solution:')
3 for b in best.candidate:
4     print(points[b.element[0]])
5 print(points[best.candidate[-1].element[1]])
6 print('Distance: {0}'.format(1/best.fitness))
```

Izvorni kod programa 3.19: rezultat

```
1         Best Solution:
2         (306.0, 360.0)
3         (283.0, 379.0)
4         (157.0, 443.0)
5         (161.0, 280.0)
6         (110.0, 225.0)
7         (343.0, 110.0)
8         (552.0, 199.0)
9         (490.0, 285.0)
10        (397.0, 566.0)
11        (325.0, 554.0)
12        Distance: 1566.13630514
```

3.3. Memoizacija

Memoizacija je tehnika spremanja međurezultata vremenski zahtjevnih funkcija. Potrebno je naglasiti da spremanje međurezultata povećava memorijske zahtjeve programa te da u kontekstu evolucijskog računarstva često međurezultati postanu nevažni nakon većeg broja stvorenih generacija pa je potrebno izbrisati spremljene međurezultate.

Ponekad nije svojstvo radnog okruženja, već je svojstvo programskog jezika ili vanjske knjižnice.

3.4. Paralelizacija

Svrha paralelizacije je brže izvođenje programa i/ili dijeljenje memorijskih resursa između više računala. Paralelizaciju je moguće ostvariti na razini algoritma, populacije te jedne iteracije[13].

Dalje je razmatrana samo paralelizacija na jednom računalu s višedretvenim procesorom.

3.4.1. Paralelizacija na razini algoritma

Paralelizacija na razini algoritma je paralelizacija kod koje je više primjeraka istog algoritma pokrenuto istovremeno. Takva paralelizacija može biti suradna (engl., *co-*

operative) ili nesuradna (engl., *non-cooperative*). Kod suradne paralelizacije instance algoritama izmjenjuju podatke o svojim trenutnim stanjima te se prema tim podacima prilagođavaju, dok kod nesuradne svaka instanca radi nezavisno neznajući za ostale[13].

3.4.2. Paralelizacija na razini jedne iteracije algoritma

Ovisno o problemu te odabranim genetskim operatorima, potrebno je paralelizirati samo "usko grlo" algoritma. Po potrebi, moguće je paralelizirati operatore križanja, mutacije ili selekcije te funkciju gubitka[13].

3.5. Nadgledanje međurezultata

Nadgledanjem međurezultata uočava se ponašanje algoritma za zadane parametre, sprema međurezultate, te ih iscrtava na graf, ali i pruža mogućnost dinamičkog mijenjanja parametara. U pravilu, nadgledanje rezultata ostvareno je oblikovnim obrascem strategija, u kojem algoritmu predajemo funkciju koja prima trenutno stanje algoritma (tzv. *kontekst*).

4. Optimizacijski problemi

Skup problema kod kojih je potrebno pronaći najbolje rješenje u prostoru mogućih rješenja nazivamo optimizacijskim problemima. Rješenje se evaluira funkcijom cilja, odnosno dobrote, ovisno o tome govori li se traženju minimuma ili maksimuma funkcije.

Dijele se prema broju kriterija po kojima se evaluira rješenje (na jednokriterijske i višekriterijske) te domeni prostora (na kontinuirane i diskretne).

4.1. Jednokriterijske i višekriterijske optimizacije

Rješenje je moguće evaluirati po jednom ili više nezavisnih kriterija. U slučaju jednokriterijske optimizacije, funkcija cilja preslikava rješenje u skalar koji opisuje kvalitetu rješenja. Kod višekriterijske optimizacije rješenje se evaluira preko više nezavisnih kriterija pa je i kvaliteta rješenja opisana vektorom, gdje je vrijednost svakog elementa vektora zapravo kvaliteta rješenja po jednom kriteriju. Rješavanje problema višekriterijskih optimizacija svodi se na svođenje jednokriterijskih optimizacija ili uvođenjem prioriteta.

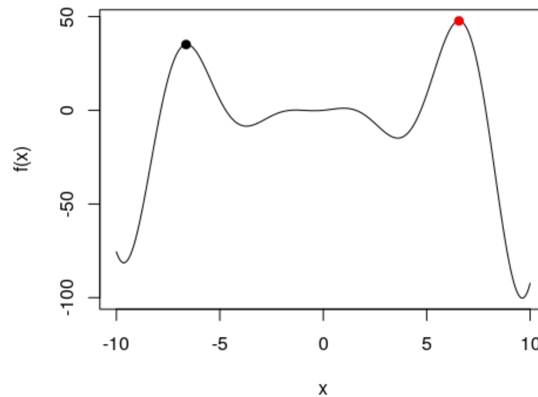
4.2. Diskretni i kontinuirani optimizacijski problemi

Ovisno o tome je li prostor rješenja definiran nad realnim ili cijelim brojevima, problem je kontinuiran ili diskretan. Diskretni optimizacijski problemi često se bave pronalaženjem zadovoljavajuće kombinacije ili permutacije pa ih tada zovemo kombinatornim problemima.

4.3. Lokalni i globalni optimum

Optimizacijski problemi često imaju više lokalnih optimuma koji se čine kao zadovoljavajuće rješenje. Na primjeru funkcije 4.1, točka $(-6.63, 35.1)$ očito nije globalni maksimum na intervalu $-10 < x < 10$ te je lokalni maksimum, dok točka $(6.56, 47.71)$ je globalni maksimum na promatranom intervalu [10].

$$f(x) = (x^2 + x)cos(x) \tag{4.1}$$



Slika 4.1: Funkcija 4.1 s označenim lokalnim i globalnim maksimumima.

4.4. Meka i tvrda ograničenja

Pri definiranju problema i obliku rješenja ponekad je potrebno reći kakva rješenja nisu poželjna ili uopće prihvatljiva. Nepoželjna rješenja opisana su mekim ograničenjima, a uobičajeno se "kažnjavaju" smanjivanjem dobrote rješenja. Neprihvatljiva rješenja su ona koja ne zadovoljavaju tvrda ograničenja, te se takva rješenja odmah odbacuju.

4.5. Primjeri optimizacijskih problema

4.5.1. Schwefelova funkcija

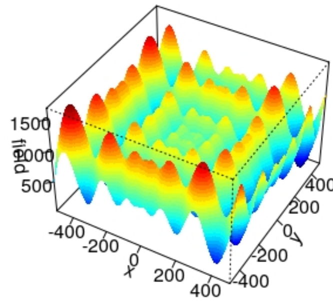
Problem je pronaći točku minimuma d-dimenzionalne Schwefelove funkcije. Optimizacijski problem je jednokriterijski, a prostor rješenja definiran je nad skupom realnih brojeva. Schwefelova d-dimenzionalna funkcija je oblika 4.2, ima globalni minimum u $f(\mathbf{x}^*) = 0$, za $\mathbf{x}^* = (420.9687, ..., 420.9687)$[11]. Funkciju odlikuje velik broj lokalnih minimuma.

$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|}) \tag{4.2}$$

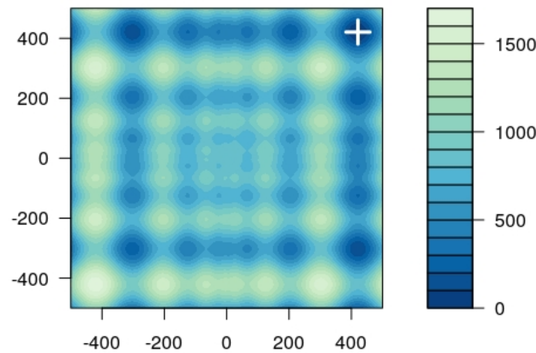
Primjer Schwefelove funkcije u dvije dimenzije ($d = 2$) dan je slikom 4.2, a konture funkcije s označenim minimumom slikom 4.3.

4.5.2. Problem trgovačkog putnika

Ako je poznat popis gradova i udaljenost između svakog od njih, problem trgovačkog putnika (engl. *Traveling Salesman Problem*) je problem pronalaska najkraćeg puta obilaska svih gradova. Problem je jednokriterijski, kombinatorni s tvrdim ograničenjem da rješenje mora biti permutacija svih gradova. Klasa složenosti problema je $\mathcal{NP}$-težak pa je neprigodan za rješavanje neheurističkim algoritmima[2].



Slika 4.2: Dvodimenzionalna Schwefelova funkcija.



Slika 4.3: Konture dvodimenzionalne Schwefelove funkcije s označenom točkom minimuma.

4.5.3. Problem pronalaska odgovarajućeg regularnog izraza

Ako je poznat skup nizova znakova koje regularni izraz mora zadovoljavati i skup niza znakova koje ne smije zadovoljavati, optimizacijski problem je pronalazak takvog regularnog izraza[6]. Problem je jednokriterijski i kombinacijski uz tvrdo ograničenje da ne smije zadovoljavati niti jedan izraz iz skupa niza znakova koje ne smije zadovoljavati.

4.5.4. Problem naprtnjače

Problem naprtnjače je kombinatorni problem popunjavanja naprtnjače ograničenog kapaciteta elementima za koje je definirana vrijednost i težina. Tvrdo ograničenje može biti (ali i ne mora) da suma težina svih elemenata ne prelazi ukupni kapacitet naprtnjače[13].

4.5.5. Problem kompozicije ciljne slike pravokutnicima u tonovima sive boje

Ciljnu sliku potrebno je dobiti "slaganjem" više pravokutnika ispunjenih tonovima sive boje. Cilj je minimizirati razliku između ciljne slike i rješenja. Postoje najmanje dvije varijante definicije problema, a razlikuju se po svojstvima pravokutnika koji tvore sliku. U jednoj verziji pravokutnici su neprozirni pa preklapanjem jedan pravokutnik u potpunosti zaklanja drugi, dok u drugoj verziji pravokutnici imaju prozirnost pa se njihovim preklapanjem boje "miješaju".

Na slici 4.4 prikazan je presjek pravokutnika s faktorom prozirnosti $\alpha = 0.5$, te presjek pravokutnika bez prozirnosti.

Slika 4.4: a) presjek pravokutnika s faktorom prozirnosti $\alpha = 0.5$, b) bez faktora prozirnosti

5. Praktični rad

U ovom poglavlju pobliže su prikazana svojstva radnih okruženja za genetske algoritme, na primjeru radnog okruženja *GA*, te radnog okruženja *pyevolve*.

Prikazano je kako definirati nove genetske operatore, paralelizacija, memoizacija te nadgledanje rješenja. Također je prikazano vremensko trajanje izvođenja problema trgovačkog putnika 4.5.2 oba radna okruženja.

5.1. Radno okruženje *GA*

GA je radno okruženje pisano za programski jezik *R*. Tipovi rješenja mogu biti permutacije, cijeli brojevi ili realni brojevi. Sadrži više predefiniranih genetskih operatora za svaki od prethodnih tipova podataka. Također ima podršku za algoritme lokalne pretrage te otočnu evoluciju (engl., *Island evolution*).

5.1.1. Rješavanje problema kompozicije ciljne slike pravokutnicima u tonovima sive boje

U ovom poglavlju opisano je rješenje problema 4.5.5, varijanta u kojoj pravokutnici imaju prozirnost.

Učitavanje potrebnih paketa

Pri rješavanju ovog problema korišteni su paketi:

- *GA* - radno okruženje
- *imager* - paket za manipulaciju nad slikama
- *memoise* - paket za memoizaciju [12]
- *dplyr* - paket sadrži "pipe" operator

A učitani su izvornim kodom 5.1.

Izvorni kod programa 5.1: Učitavanje paketa

```
1 library(GA)
2 library(imager)
3 library(memoise)
4 library(dplyr)
```

Učitavanje ciljne slike i osnovnih parametara

Ciljna slika prikazana je na slici 5.1, a učitana je izvornim kodom 5.2. U istom izvornom kodu učitane su i dimenzije slike.

Slika 5.1: Ciljna slika.

Izvorni kod programa 5.2: Učitavanje slike

```
1 target_image_path <- "11-kuca-200-133.png"
2 width <- width(target_image)
3 height <- height(target_image)
```

Definiranje jedinke

Jedinka je definirana kao jednodimenzionalan niz u kojem uzastopne šestorke označavaju sljedeće vrijednosti:

- *x0* - x koordinatu točke,
- *y0* - y koordinatu točke,
- *width* - širinu pravokutnika,
- *height* - visinu pravokutnika,
- *grayscale* - ton sive u rasponu od 0 do 1,
- *alpha* - koeficijent prozirnosti u rasponu od 0 do 1.

Lako je zaključiti da je duljina niza zapravo umnožak broja pravokutnika i broja gore navedenih atributa. Struktura jedinke definirana je kodom 5.3.

Izvorni kod programa 5.3: Definiranje jedinke

```
1 structure_elements =
2    c("x0", "y0", "width", "height", "grayscale", "alpha")
3 size_of_structure <- length(structure_elements)
4 number_of_rectangles <- 55
5 duljina_jedinke <- size_of_structure * number_of_rectangles
6 rect_iter <- 0:(number_of_rectangles-1) * size_of_structure + 1
7 #rect_iter sadrzi indekse na kojima pocinju sestorke koje
8 # definiraju pojedini pravokutnik
```

Definiranje prostora pretraživanja

Definirano je da pojedini pravokutnik ne može biti opisan točkom van slike, te da mu visina i širina neće prelaziti petinu visine i širine slike. Također, ton sive i prozirnost su koeficijenti između 0 i 1. Takav prostor pretraživanja opisan je 5.4. Jedinka sadrži onoliko šestorki koliko sadrži pravokutnika, tako da vektor koji ograničava prostor pretraživanja ponavljamo toliko puta. Proširenje je prikazano kodom 5.5

Izvorni kod programa 5.4: Definiranje ograničenja pojedinog pravokutnika

```
1 searchSpace.lower <- c(0, 0, -width/5, -height/5, 0, 0)
2 searchSpace.upper <- c(width, height, width/5, height/5, 1, 1)
```

Izvorni kod programa 5.5: Definiranje ograničenja jedinke

```
1 searchSpace.lower <- rep(searchSpace.lower, number_of_rectangles)
2 searchSpace.upper <- rep(searchSpace.upper, number_of_rectangles)
```

Definiranje funkcije gubitka

Funkciju gubitka definirana je kao zbroj apsolutne razlike piksela ciljne slike i slike jedinke. Definirana je pomoćna funkcija koja od jedinke stvori sliku, a implementacija je izvornim kodom 5.6

Izvorni kod programa 5.6: Definiranje funkcije gubitka

```
child_to_image <- function(child) {

  cimg(array(1, c(width,height,1,3))) -> tmp_img

  for (i in rect_iter){
    x0 <- child[i]
    y0 <- child[i+1]
    x1 <- child[i+2] + x0
    y1 <- child[i+3] + y0
    grayscale <- child[i+4]
    alpha <- child[i+5]
    draw_rect(tmp_img, x0, y0, x1, y1,
    opacity = alpha, color=
      rgb(red = grayscale, green = 0,blue = 0,alpha = 1),filled = TRUE)
      -> tmp_img
  }

  return(tmp_img)

}

absolute_1d_loss <- function(child) {
  child_img <- child_to_image(child)
  sum(abs(child_img[,,1,1] - target_image.grayscale)) -> absLoss
  return (width*height - absLoss)
}
```

Memoizacija

Funkcija gubitka je skupa zbog pretvaranja jedinke u sliku pa ćemo ju memoizirati. Memoizacija je pojašnjena u poglavlju 3.3, a ostvarena je izvornim kodom 5.7.

Izvorni kod programa 5.7: Definiranje memoizacije

```
m_absolute_1d_loss <- memoise(absolute_1d_loss)
```

Definiranje operatora križanja

U rješavanju ovog problema koristi se križanje s jednom točkom križanja, ali na način da pravokutnici ostanu očuvani.

Funkcija koja opisuje operator križanja prima trenutno stanje algoritma (argument *object*), parametre predane *ga* funkciji (argument ..., tzv., *ellipsis*), te listu pozicija roditelja u trenutnoj populaciji. Funkcija vraća imenovanu listu s dva imenovana člana, *children* koji sadrži djecu i *fitness* koji sadrži vrijednosti funkcije dobrote ako su poznate (*NA* inače). Implementacija je dana izvornim kodom 5.8.

Izvorni kod programa 5.8: Definiranje operatora križanja

```
point_structure_crossover <- function(object, parents) {
  p1index <- parents[1]
  p2index <- parents[2]

  p1 <- object@population[p1index,]
  p2 <- object@population[p2index,]

  break_point <- sample(rect_iter, 1) #tocka presjeka

  #rubni slucaj kad je tocka presjeka zapravo prva tocka
  if(break_point == 1) break_point <- rect_iter[2]

  #izracun jedinki
  v1 <- c(p1[1:(break_point-1)], p2[break_point:duljina_jedinke])
  v2 <- c(p2[1:(break_point-1)], p1[break_point:duljina_jedinke])

  #definiranje rezultata
  rezultat <- list()
  rezultat$children <- matrix(data = c(v1, v2), nrow = 2, byrow = T)
  rezultat$fitness <- c(NA, NA)

  return (rezultat)
}
```

Funkcija nadgledanja rezultata

Funkcija kao parametar prima trenutno stanje algoritma. Implementacija 5.9 briše memoizirane vrijednosti svakih 1000 generacija.

Izvorni kod programa 5.9: Funkcija nadgledanja rezultata

```
monitor <- function(obj) {
    if(obj@iter %% 1000 == 0) {
      forget(m_absolute_1d_loss)
    }
}
```

Pokretanje algoritma

Algoritam se pokreće funkcijom *ga*. Funkciji se predaje niz imenovanih parametara:

- **type** - tip podataka jedinke
- **fitness** - funkcija dobrote
- **lower** - najniže vrijednosti prostora realnih rješenja
- **upper** - najviše vrijednosti prostora realnih rješenja
- **pmutation** - vjerojatnost mutacije

- **selection** - funkcija selekcije
- **mutation** - funkcija mutacije
- **crossover** - funkcija križanja
- **popSize** - veličina populacije
- **elitism** - broj najboljih jedinki koje se prenose u sljedeću generaciju
- **maxiter** - najveći broj iteracija
- **parallel** - postavke paralelizacije
- **monitor** - funkcija nadgledanja rezultata
- **run** - najveći broj iteracija s istom vrijednosti dobrote

Imenovani parametri ovise o odabranom tipu podataka, genetskim operatorima te funkciji nadgledanja rezultata.

Izvorni tekst programa 5.10 prikazuje poziv funkcije. Algoritam je paraleliziran, ali nigdje nije navedeno na kojoj razini, odnosno koji dijelovi algoritma su paralelizirani. Paralelizacija je algoritmu zadana parametrom *parallel*. Kako je vrijednost parametra u ovom slučaju zastavica *true*, algoritam interno dohvaća broj jezgri preko funkcije *detect cores* paketa *parallel*.

Izvorni kod programa 5.10: Pokretanje algoritma

```
1  GA <- ga(type = "real-valued",
2           fitness = m_absolute_1d_loss,
3           lower = searchSpace.lower,
4           upper = searchSpace.upper,
5           pmutation = 0.2,
6           selection = gareal_tourSelection,
7           mutation = gareal_raMutation,
8           crossover = point_structure_crossover,
9           popSize = 55,
10          elitism = 5,
11          maxiter = 10000,
12          parallel = TRUE,
13          monitor = monitor,
14          run = 100
15 )
```

Rezultati

Uzima se najbolja jedinka i pretvara u sliku. Pretvorba je implementirana kodom 5.11, a 5.2 rezultantna slika.

Izvorni kod programa 5.11: Pregled rezultata

```
1  > child_to_image(GA@solution[1,]) %>%
2  +     plot(., colourscale=function(r,g,b) rgb(r,r,r), rescale = F)
```

Slika 5.2: Rezultat

5.2. Radno okruženje *pyevolve*

pyevolve je radno okruženje pisano za programski jezik *python*. Sadrži niz predefiniranih genetskih operatora, skaliranje, metode za statističku analizu i grafičko prikazivanje statistike i rezultata, te interaktivni način (engl., *interactive mode*).

5.2.1. Rješavanje problema naprtnjače

U ovom poglavlju opisano je rješenje problema 4.5.4 korištenjem *pyevolve* radnog okruženja. Predefinirani operatori dovoljni su za analizu i rješavanje problema.

Jedinka je definirana kao niz zastavica koje označavaju sadrži li jedinka u svom ruksaku element na indeksu zastavice.

Učitavanje potrebnih modula

Učitavanje potrebnih modula izvedeno je u izvornom kodu 5.12.

Izvorni kod programa 5.12: Učitavanje potrebnih modula

```
1 from pyevolve import G1DBinaryString
2 from pyevolve import GSimpleGA
3 from pyevolve import Selectors
4 from pyevolve import Mutators
5 from pyevolve import Scaling
6 from pyevolve import Consts
7
8 from itertools import izip
```

Učitavanje podataka

Izvor podataka [5]. Podaci su podijeljeni u četiri datoteke (*Napomena: "x" je oznaka broja problema*):

- *p0x_c* - kapacitet naprtnjače,

- *p0x_w* - težine elemenata,
- *p0x_p* - vrijednosti elemenata,
- *p0x_s* - optimalan rezultat

Funkcije za učitavanje podataka dana je kodom 5.13. U glavnom programu u izvornom kodu 5.14 prikazano je učitavanje podataka.

Dalje u tekstu obrađuje se primjer *p08*.

Izvorni kod programa 5.13: Funkcije za učitavanje podataka

```
1  def get_variables_from_file(file):
2     result = [int(r) for r in file.readlines()]
3     return result
4
5  def read_problem_variables(problem_id = "p08"):
6     profits_file = open("data/" + problem_id + "_p.txt")
7     profits = get_variables_from_file(profits_file)
8     profits_file.close()
9
10    weights_file = open("data/" + problem_id + "_w.txt")
11    weights = get_variables_from_file(weights_file)
12    weights_file.close()
13
14    capacity_file = open("data/" + problem_id + "_c.txt")
15    capacity = get_variables_from_file(capacity_file)[0]
16    capacity_file.close()
17
18    optimal_file = open("data/" + problem_id + "_s.txt")
19    optimal = get_variables_from_file(optimal_file)
20    optimal_file.close()
21
22    return (profits, weights, capacity, optimal)
```

Izvorni kod programa 5.14: Funkcije za učitavanje podataka

```
1  global profits
2  global weights
3  global capacity
4  global n
5  global total_weight
6
7  profits, weights, capacity, optimal = read_problem_variables("p08")
8
9  n = len(weights) #broj elemenata
10 total_weight = sum(weights)
```

Definiranje funkcije gubitka

Funkcija gubitka definirana je izrazom 5.1 gdje je:

– **x** - vektor zastavica. Postavljena zastavica na i-tom mjestu označava da jedinka sadrži element na i-tom mjestu.

– W - zbroj težina svih elemenata,

– p_i - vrijednost i-tog elementa,

– w_i - težina i-tog elementa.

Implementacija je dana izvornim kodom 5.15.

$$loss(\mathbf{x}) = 2W - \sum_i x_i p_i + \sum_i (w_i) |\sum_i (x_i w_i) - W| \quad (5.1)$$

Izvorni kod programa 5.15: Definiranje funkcije gubitka

```
1
2 def evaluation_function(chromosome):
3    calc_profit = 0
4    calc_weight = 0
5
6    for profit, weight, is_contained in izip(profits, weights, chromosome):
7       if(is_contained == 0):
8          continue
9       calc_profit += profit
10      calc_weight += weight
11
12   score = total_weight*total_weight - calc_profit + total_weight
13               * abs(calc_weight - capacity)
14
15   return score
```

Definiranje genoma

Nad genomom definira se tip, funkcija inicijalizacije, operator mutacije, križanja te funkciju evaluacije. Ako neki od operatora nije eksplicitno naveden, postavljen je predefinirani operator ovisno o tipu genoma. U ovom slučaju, operatori inicijalizacije i križanja nisu navedeni te su oni postavljeni na nasumični inicijalizator(u dokumentaciji *G1DBinaryStringInitializator*), te na križanje s jednom točkom presjeka (u dokumentaciji *G1DBinaryStringXSinglePoint*).

Izvorni kod programa 5.16: Definiranje genoma

```
1 genome = G1DBinaryString.G1DBinaryString(n)
2 genome.evaluator.set(evaluation_function)
3 genome.mutator.set(Mutators.G1DBinaryStringMutatorFlip)
```

Definiranje genetskog algoritma

Kroz definiranje genetskog algoritma definira se način selekcije, tip optimizacije (traženje minimuma ili maksimuma) te broj generacija. Implementacija je dana izvornim kodom 5.17.

Izvorni kod programa 5.17: Definiranje genetskog algoritma

```
ga = GSimpleGA.GSimpleGA(genome)
ga.selector.set(Selectors.GTournamentSelector)
ga.setMinimax(Consts.minimaxType["minimize"])

ga.setGenerations(2700)
```

Definiranje populacije

Kroz definiranje populacije definira se veličina, skaliranje te broj najboljih jedinki koje se prenose u sljedeću generaciju. Implementacija je dana kodom 5.18

Izvorni kod programa 5.18: Definiranje populacije

```
pop = ga.getPopulation()
pop.setPopulationSize(750)
ga.setElitism(True)
ga.setElitismReplacement(30)
pop.scaleMethod.set(Scaling.LinearScaling)
```

Pokretanje algoritma

Metodi *evolve* predan je parametar *freq_stats* kojim se određuje učestalost pozivanja funkcije nadgledanja rezultata. Kako nije navedena funkcija nadgledanja rezultata, koristi se predefinirana koja ispisuje najmanju, prosječnu i najveću dobrotu jedinke te broj generacije.

Program je pokrenut programskim kodom 5.19, a ispisani međurezultati su prikazani u 5.20

Izvorni kod programa 5.19: Pokretanje algoritma

```
ga.evolve(freq_stats=300)
```

Izvorni kod programa 5.20: Ispis međurezultata

```
1
2        Gen. 0 (0.00%): Max/Min/Avg Fitness(Raw)
3        [215217960389923.72(211166405075268.00)/
4        162176787888048.31(164116347179893.00)/
5        179348300324936.41(179348300324936.53)]
6
7        Gen. 1000 (20.00%): Max/Min/Avg Fitness(Raw)
8        [209877428323336.75(206579524815866.00)/
9        162925458631830.34(164054226446720.00)/
10       174897856936113.91(174897856936113.94)]
11
12       Gen. 2000 (40.00%): Max/Min/Avg Fitness(Raw)
13       [207185720502690.50(200103619216371.00)/
14       161835193766182.44(164054226446720.00)/
15       172654767085575.47(172654767085575.38)]
16
17       Gen. 3000 (60.00%): Max/Min/Avg Fitness(Raw)
18       [206160489865482.12(198770600337576.00)/
19       161931592864060.31(164054098160058.00)/
20       171800408221235.09(171800408221235.16)]
21
22       Gen. 4000 (80.00%): Max/Min/Avg Fitness(Raw)
23       [205856334478235.00(199260585906940.00)/
24       162270826978017.62(164054098160058.00)/
25       171546945398529.19(171546945398529.12)]
26
27       Gen. 5000 (100.00%): Max/Min/Avg Fitness(Raw)
28       [205747141731581.88(204686859630121.00)/
29       163817930899466.31(164054098160058.00)/
30       171455951442984.84(171455951442984.97)]
```

Pregled rješenja

Objekt rješenja se dohvaća preko metode *bestIndividual()*, a objekt sadrži informacije o provedenom algoritmu i dobroti rješenja.

Ispisano rješenje dano je u 5.21. Dobiveno rješenje je *1101110000111011100010001*, s vrijednošću funkcije gubitka $1.6405409816e + 14$, odnosno ukupnim kapacitetom *13346262*, te težinom *6402560*.

Izvorni kod programa 5.21: Pregled rješenja

```


      Total time elapsed: 36.249 seconds. - GenomeBase Score:
164054098160058.000000 Fitness:                 163817930899466.312500

        Params:          {}

        Slot [Evaluator] (Count: 1)
                Name: evaluation_function - Weight: 0.50
        Slot [Initializator] (Count: 1)
                Name: G1DBinaryStringInitializator - Weight: 0.50
                Doc:  1D Binary String initializator
        Slot [Mutator] (Count: 1)
                Name: G1DBinaryStringMutatorFlip - Weight: 0.50
                Doc:  The classical flip mutator for binary strings
        Slot [Crossover] (Count: 1)
                Name: G1DBinaryStringXSinglePoint - Weight: 0.50
                Doc:  The crossover of 1D Binary String, Single Point

- G1DBinaryString
String length:   24
String:          110111000011101110010001
```

6. Učinkovitost radnih okruženja

Glavni razlozi za stvaranje radnih okruženja su ušteda vremena korisniku zbog kraćeg vremena pisanja koda, efikasnijeg izvođenja programa zbog kvalitete napisanog okruženja, manja vjerojatnost greške u programu, te veći broj već razvijenih mogućnosti.

U ovom poglavlju ispitan je utjecaj paralelizacije i memoizacije na prostorne i vremenske zahtjeve, te su uspoređena dva radna okruženja.

Ispitivanje je provedeno na četverojezgrenom *Intel Core i5-5200U* procesoru, takta 2.20 Hz.

6.1. Utjecaj paralelizacije

U ovom poglavlju prikazan je utjecaj paralelizacije pri rješavanju problema trgovačkog putnika na vremensko i prostorno izvođenje programa. Korišteno je radno okruženje *GA*.

Korišten je primjer i kod iz poglavlja 5.1, uz nadodanu funkciju nadgledanja rezultata te izmijenjene parametre algoritma.

Funkcija nadgledanja međurezultata

Definirana je funkcija koja za prvu i svaku stotu generaciju zabilježi vremenski trenutak u kojem je generacija generirana. Vrijeme je mjereno od trenutka početka izvođenja programa. Implementacija funkcije dana je izvornim kodom 6.1.

Izvorni kod programa 6.1: Definiranje funkcije nadgledanja međurezultata

```
1  monitor <- function(x) {
2          if(x@iter %% 100 == 0 || x@iter == 1) {
3                  time <- as.numeric(Sys.time() - start_time)
4                  iter <- x@iter
5                  runDataFrame <<- rbind(runDataFrame, c(iter, time))
6          }
7  }
8
9  runDataFrame <-  data.frame(Gen = integer(), time = double())
10 start_time <- Sys.time()
```

Pokretanje algoritma

Algoritam generira najviše 1000 generacija, s veličinom populacije 55. U ovom trenutku naglasak nije na efikasnosti generiranja rješenja, već samo na vremenskoj analizi. Slijedno su pokrenuta dva algoritma, prvi nije paraleliziran te je njegova implementacija dana izvornim kodom 6.2, dok drugi je te je njegova implementacija dana izvornim kodom 6.3.

Izvorni kod programa 6.2: Pokretanje algoritma

```
1
2  GA <- ga(type = "real-valued",
3           fitness = absolute_1d_loss,
4           lower = searchSpace.lower,
5           upper = searchSpace.upper,
6           pmutation = 0.2,
7           selection = gareal_tourSelection,
8           mutation = gareal_raMutation,
9           crossover = point_structure_crossover,
10          popSize = 55,
11          elitism = 5,
12          maxiter = 1000,
13          parallel = F,
14          monitor = monitor,
15          run = 100
16 )
```

Izvorni kod programa 6.3: Pokretanje algoritma

```
1
2  GA <- ga(type = "real-valued",
3           fitness = absolute_1d_loss,
4           lower = searchSpace.lower,
5           upper = searchSpace.upper,
6           pmutation = 0.2,
7           selection = gareal_tourSelection,
8           mutation = gareal_raMutation,
9           crossover = point_structure_crossover,
10          popSize = 55,
11          elitism = 5,
12          maxiter = 1000,
13          parallel = T,
14          monitor = monitor,
15          run = 100
16 )
```

Tablica 6.1: Rezultati bez paralelizacije

Broj generacije	trenutak generiranja (u minutama)
1	0.054337
100	3.950987
200	7.929653
300	11.956440
400	15.923583
500	19.975753
600	24.137899
700	28.271459
800	32.359452
900	36.303572
1000	40.283065

Tablica 6.2: Rezultati s paralelizacijom

Broj generacije	trenutak generiranja (u minutama)
1	0.044153
100	3.323507
200	6.708185
300	10.046442
400	13.399328
500	16.759842
600	20.130740
700	23.494662
800	26.858687
900	30.223891
1000	33.589826

Prikaz rezultata

U tablici 6.2 prikazan je trenutak generiranja određene generacije paraleliziranog algoritma.

6.2. Utjecaj memoizacije

U ovom poglavlju prikazan je utjecaj memoizacije pri rješavanju problema trgovačkog putnika na vremensko i prostorno izvođenje programa. Korišteno je radno okruženje *GA*. Korišten je primjer i kod iz poglavlja 5.1, uz ostvarenu memoizaciju, a rješenje je uspoređeno s pokretanjem algoritma bez paralelizacije i memoizacije s rezultatom 6.1.

Tablica 6.3: Rezultati s memoizacijom

Broj generacije	trenutak generiranja (u minutama)
1	0.049523
100	2.971360
200	3.584964
300	4.943158
400	6.187493
500	7.569821
600	8.763606
700	9.969527
800	11.336784
900	12.703022
1000	13.925927

Ostvarenje memoizacije

Memoizacija je ostvarena *memoise* funkcijom iz paketa *memoise*. Implementacija je dana izvornim kodom 6.4.

Izvorni kod programa 6.4: Ostvarenje memoizacije

```
1 m_absolute_1d_loss <- memoise(absolute_1d_loss)
```

Pokretanje algoritma

Pokretanje algoritma ostvareno je izvornim kodom 6.5.

Izvorni kod programa 6.5: Pokretanje algoritma

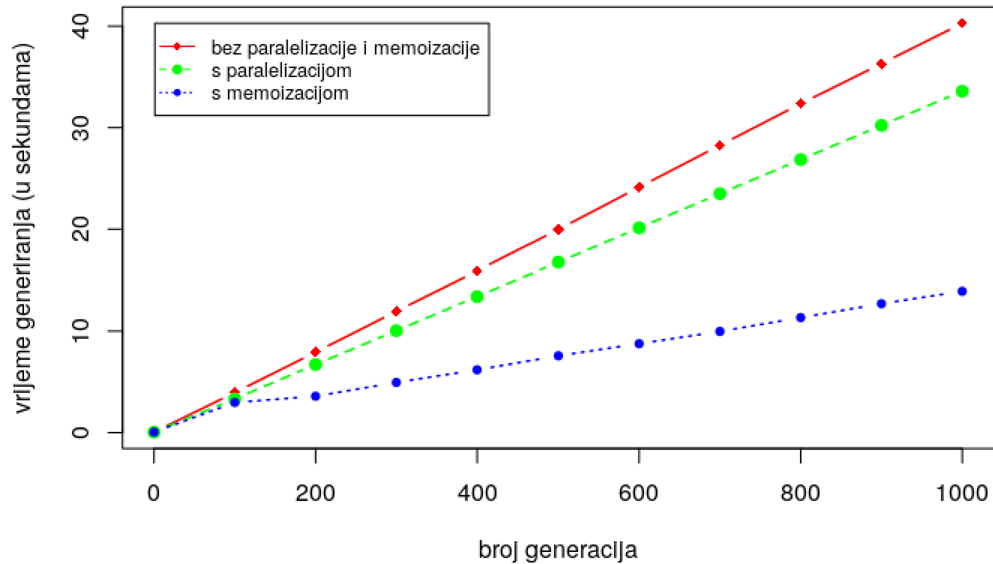
```
1  GA <- ga(type = "real-valued",
2           fitness = m_absolute_1d_loss,
3           lower = searchSpace.lower,
4           upper = searchSpace.upper,
5           pmutation = 0.2,
6           selection = gareal_tourSelection,
7           mutation = gareal_raMutation,
8           crossover = point_structure_crossover,
9           popSize = 55,
10          elitism = 5,
11          maxiter = 1000,
12          parallel = F,
13          monitor = monitor,
14          run = 100
15 )
```

Prikaz rezultata

U tablici 6.3 prikazani su rezultati izvođenja algoritma s memoizacijom.

6.3. Analiza rezultata

Vidljivo je da je paralelizacija ubrzala proces pretrage, ali ne proporcionalno broju dretvi. Također je vidljivo da memoizacija ima najveći utjecaj na poboljšanje brzine izvođenja programa, ali da je značajna razlika vidljiva nakon sto generacija. Razlog tomu je što memoizacija treba "zapamtiti" podatke i učinkovita je tek nakon što se jedinke u populaciji počnu ponavljati.



Slika 6.1: Vremenski trenutak od početka pokretanja algoritma u kojem je generirana određena generacija.

7. Zaključak

U okviru ovog rada prikazano je rješavanje više optimizacijskih problema različitim radnim okruženjima, prikazano je i definiranje prilagođenih jedinki i genetskih operatora, te učinak memoizacije i paralelizacije na brzinu izvođenja rada.

Korisnost radnog okruženja ovisi o svojstvima jezika za koji je napisana, kvaliteti dokumentacije, predefiniranim operatorima i složenosti sučelja za definiranje prilagođenih operatora.

Dodatne mogućnosti za ubrzanje izvođenja algoritma i analizu problema često nisu nativno podržane samim okruženjem, ali ih uglavnom nije teško izvesti dodatnim paketima ili knjižnicama.

Promatrana radna okruženja imaju manjkavu dokumentaciju što usporava proces rješavanja problema, ali podržavaju velik broj predefiniranih operatora.

Što se tehnika ubrzanja algoritama tiče, oba podržavaju paralelizaciju, ali zbog loše dokumentacije teško je razlučiti koji dio algoritma je paraleliziran što može utjecati na oblikovanje funkcije gubitka i prilagođenih operatora.

Niti jedno okruženje ne podržava memoizaciju iako je po provedenom ispitivanju dala puno veće ubrzanje od paralelizacije.

Literatura

[1] *DEAP documentation.* URL https://deap.readthedocs.io/en/master/.

[2] Travelling salesman problem. URL https://en.wikipedia.org/wiki/Travelling_salesman_problem. Accessed: 2019-05-26.

[3] inspyred ant colony optimization. https://pythonhosted.org/inspyred/examples.html#ant-colony-optimization. Accessed: 2019-05-26.

[4] *Pyevolve documentation.* URL http://pyevolve.sourceforge.net/.

[5] John Burkardt. *Data for the 01 Knapsack Problem.* URL http://people.sc.fsu.edu/~jburkardt%20/datasets/knapsack_01/knapsack_01.html.

[6] Philip H.W. Leong Farzad Noorian, Anthony M. de Silva. Grammatical evolution: A tutorial using gramevol. 2016. URL https://cran.r-project.org/web/packages/gramEvol/vignettes/ge-intro.pdf.

[7] Branko Spasojević Nikolina Mišljenčević. *Genetski algoritmi.* URL http://www.zemris.fer.hr/~golub/ga/studenti/projekt2007/ga.html#3.3.3.

[8] Farzad Noorian, Anthony M. de Silva, i Philip H. W. Leong. gramEvol: Grammatical evolution in R. *Journal of Statistical Software*, 71(1):1–26, 2016. doi: 10.18637/jss.v071.i01.

[9] Luca Scrucca. GA: A package for genetic algorithms in R. *Journal of Statistical Software*, 53(4):1–37, 2013. URL http://www.jstatsoft.org/v53/i04/.

[10] Luca Scrucca. *A quick tour of GA*, 2019. URL https://cran.r-project.org/web/packages/GA/vignettes/GA.html.

[11] Derek Bingham Sonja Surjanovic. *Virtual Library of Simulation Experiments.* URL https://www.sfu.ca/~ssurjano/schwef.html.

[12] Hadley Wickham, Jim Hester, Kirill Müller, i Daniel Cook. *memoise: Memoisation of Functions*, 2017. URL https://CRAN.R-project.org/package=memoise. R package version 1.1.0.

[13] Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi. Metaheuristike.*, 2013.

Radna okruženja za evolucijsko računanje

Sažetak

U radu je prikazano rješavanje nekih optimizacijskih problema u različitim radnim okruženjima za različite algoritme evolucijskog računarstva. Pri tome je prikazano i definiranje prilagođenih jedinki i genetskih operatora te implementacija memoizacije i paralelizacije. Napravljena je analiza utjecaja paralelizacije i memoizacije na vremensko izvođenje algoritma.

Ključne riječi: radno okruženje, evolucijsko računarstvo, problem trgovačkog putnika, knapsack problem, paralelizacija, memoizacija

Evolutionary Computation Frameworks

Abstract

In this paper it is described and shown how to solve some optimization problems in different frameworks for different Evolutionary Computation algorithms. It's shown how to define custom units, and genetic operators. It is also shown how to implement memoisation and parallelization. A simple analysis is made to show an impact of memoisation and parallelization on time performance of a algorithm.

Keywords: framework, evolutionary computing, traveling salesman problem, knapsack problem, parallelization, memoisation