Primjena genetskih algoritama u generiranju računalnog koda

Horvat, Luka

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:926805

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Luka Horvat

PRIMJENA GENETSKIH ALGORITAMA U GENERIRANJU RAčUNALNOG KODA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Luka Grubišić

Zagreb, 2018.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod **1**

1 Genetski algoritmi **2**
1.1 Opis 2
1.2 Mutacija i križanje 2
1.3 Selekcija 3
1.4 Genetsko programiranje 3

2 Implementacija **5**
2.1 Osnovni algoritam 5
2.1.1 Inicijalizacija i selekcija 5
2.1.2 Mutacija i križanje 7
2.1.3 Iteracija 7
2.2 Specijacija 8
2.3 Distribuirani sustav 10
2.4 Jednostavni primjer 11

3 Primjer: Regularni izrazi **13**
3.1 Definicija 13
3.2 Proširenje 14
3.3 Sintaksna stabla 15
3.3.1 Prihvaćanje nizova znakova 15
3.3.2 Inicijalna populacija 17
3.3.3 Mutacija i križanje 17
3.4 Problem: Prepoznavanje jezika 20
3.4.1 Opis 20
3.4.2 Implementacija 20
3.4.3 Rezultati 22
3.5 Primjene 24

3.5.1 Prepoznavanje jezika u rečenicama 24
3.5.2 Prepoznavanje izmišljenih riječi 26

Bibliografija **28**

Uvod

Genetski algoritmi dobro su poznati po svojoj širokoj primjenjivosti. Ipak, klasične primjene uglavnom se svode na optimizaciju fiksnog broja parametara. U analogiji s biološkom evolucijom, skup parametara predstavlja jedinku/genom. Mutacija je pomak jednog ili više parametara, a križanje je primjena neke funkcije usrednjavanja na odgovarajuće parametre dvije jedinke.

U ovom radu promatra se nešto direktnija interpretacija. Genom će predstavljati računalni kod u obliku sintaksnog stabla. *Fitness* jedinke bit će određen interpretiranjem tog koda u nekom okruženju i vrednovanjem njegovog uspjeha u rješavanju danog zadatka.

Često je funkciju daleko lakše opisati primjerima kako ona djeluje nad nekim skupom ulaznih podataka i eventualnim ograničenjima na samu definiciju. Konačni cilj je dobiti sustav koji sam razvija strategiju za rješavanje danog problema bez intervencije programera.

Glavna primjena obrađena u ovom radu je automatsko generiranje regularnih izraza koji prihvaćaju jedan, a ne prihvaćaju drugi skup riječi. Konkretno, evoluiran je regularni izraz koji prepoznaje engleske, a odbija njemačke riječi.

Konačni rezultat je izraz dug približno sto znakova koji razaznaje engleske od njemačkih riječi točnošću od približno 75%. Takav izraz može zamijeniti korištenje rječnika veličine nekoliko megabajta u primjenama u kojima nije potrebna maksimalna preciznost, ali je važna brzina prepoznavanja i veličina koda. Razmotrene su još neke potencijalne primjene, kao što je prepoznavanje jezika teksta koji ima greške ili koristi sleng.

Poglavlje 1

Genetski algoritmi

1.1 Opis

Za dani optimizacijski problem definiramo funkciju cilja $f : U \to V$ gdje su U prostor rješenja i V neki skup s definiranim totalnim uređajem. Funkcija cilja svakom potencijalnom rješenju pridružuje rezultat tako da "boljim" rješenjima daje veći rezultat.

Genetski algoritam je metaheuristika inspirirana prirodom[8]. Elemente iz skupa U nazivamo jedinke, a skup jedinaka zovemo populacija. Populacija obično kreće kao skup nasumično generiranih jedinki. Te jedinke se ocjenjuju funkcijom f pa se, analogno s prirodnom selekcijom, eliminiraju lošija rješenja dok se bolja međusobno križaju i mutiraju čime nastaje nova generacija. Proces se iterira sve dok nismo zadovoljni rješenjem.

Važno je da jedna iteracija algoritma ne rezultira samo jednim rješenjem već cijelom populacijom kandidata. Iako neka jedinka postiže gore rezultate, njezino mjesto u prostoru rješenja možda može rezultirati boljim rezultatima u kasnijim generacijama.

Genetski algoritmi mogu se primijeniti na vrlo složene probleme u kojima funkcija cilja vrednuje različite aspekte koji zajedno čine rješenje. U takvim primjenama lako je moguće da neka jedinka postiže dobre rezultate u jednom aspektu, ali gore u nekom drugom. Tada očekujemo da će križanje s komplementarnom jedinkom proizvesti rješenja koja su bolja od oba roditelja.

1.2 Mutacija i križanje

Uloga mutacije je uvođenje novog genetskog materijala u populaciju. Način mutiranja jedinke ovisi o njenoj reprezentaciji. Ako su genomi reprezentirani kao vektori bitova, mutacija bi mogla biti definirana kao okretanje nasumičnog podskupa tih bitova. Kod

složenijih struktura moguće su razne implementacije te je to jedan od parametara s kojim možemo poboljšati djelovanje algoritma.

Slično, križanje također ovisi o reprezentaciji. Uloga križanja je kombiniranje prednosti različitih rješenja. Na primjeru bit-vektora, križanje može biti implementirano tako da se odabere podskup bitova drugog roditelja te se oni presade u kopiju prvog roditelja na istim pozicijama. Zbog simetrije, može se definirati da križanje proizvede dva potomka, u kojem slučaju drugi potomak se generira analogno s zamijenjenim ulogama roditelja.

Ukoliko je to moguće svaka jedinka iz populacije križa se sa svakom drugom jedinkom. Budući da to rezultira kvadratnom eksplozijom novih kandidata koje treba evaluirati, često je takav pristup preskup. Umjesto tog, roditelji mogu biti birani po svom uspjehu u trenutnoj generaciji. Više rangirana jedinka imat će veću šansu da bude izabrana za razmnožavanje.

1.3 Selekcija

Način na koji odabiremo jedinke koje preživljavaju je također važan. Najjednostavnija metoda rangira sve jedinke u trenutnoj generaciji te uzima najboljih n za sljedeću generaciju. Ukoliko se želi dati šansa gorim rješenjima, jedinke se mogu nasumično selektirati s većom šansom selekcije za bolje jedinke. Takav pristup često se kombinira s *elitizmom*, što znači da neki fiksni broj najboljih jedinki ima garantirano preživljavanje u sljedeću generaciju.

1.4 Genetsko programiranje

U ovom radu genom je reprezentiran kao sintaksno stablo programa. Implementacija mutacije i križanja opisana je u sekciji Sintaksna Stabla. Korištenje genetskih algoritama za razvoj programa zove se genetsko programiranje. Reprezentacija koda može biti linearna lista instrukcija nalik assembleru, ili imati strukturu grafa, najčešće stabla. Kod linearne reprezentacije mutacija može biti implementirana kao ubacivanje ili izbacivanje neke instrukcije u kod dok križanje možemo implementirati kao presađivanje djela koda iz jednog roditlja u drugog.

Stablaste strukture uglavnom mutiraju na sličan način (dodavanjem i brisanjem podstabla), no križanje može biti dosta složenije. Budući da stabla imaju više "strukture" od liste instrukcija, potrebno je odlučiti što od te strukture se želi zadržati u novoj jedinki. Implementacija uglavnom odabire neko podstablo u oba roditelja te ih zamjenjuje. Odabir podstabla također nije trivijalan jer se mora donjeti odluka želi li se svakoj grani dati jednaka vjerojatnost odabira ili će ta vjerojatnost ovisiti

o veličini podstabla. Također treba odlučiti u kojem trenutku prestaje silazak niz stablo.

Ipak, stablaste strukture imaju i prednosti. Svako podstablo uglavnom predstavlja samostalnu cjelinu pa zamjenom istih je veća vjerojatnost da presađujemo funkcionalnost. Kod linearne reprezentacije, komad koda je vrlo vjerojatno koristan samo u danom kontekstu pa neće imati smisla presađen u novu jedinku.

Poglavlje 2

Implementacija

Algoritam je napisan u programskom jeziku Haskell koji je pogodan za baratanje stablastim strukturama. Prvo je implementirana generička verzija genetskog algoritma parametrizirana nad odabirom funkcije cilja, mutacije i križanja.

2.1 Osnovni algoritam

Osnovne funkcije kojima je implementiran algoritam su sljedeće:

```
newInitialUnit :: MonadEvolution u v m => m u
mutateUnit     :: MonadEvolution u v m => u -> m u
crossUnits     :: MonadEvolution u v m => u -> u -> m u
evaluateUnit   :: MonadEvolution u v m => u -> m v
```

Tip funkcije, primjerice `crossUnits`, govori da ona prima dvije vrijednosti tipa `u` te vraća novu vrijednost istog tipa, unutar konteksta `m`. `MonadEvolution u v m` znači da kontekst unutar kojeg se `crossUnits` izvodi mora odrediti kojeg konkretnog tipa su `u` i `v` te dati samu implementaciju te funkcije. Varijabla `u` predstavlja skup U, dok varijabla `v` predstavlja V.

Propagiranjem tog ograničenja kroz složenije funkcije, možemo izgraditi cijeli algoritam bez konkretnih implementacija osnovnih funkcija. Sljedeće funkcije su poopćenja `newInitialUnit` i `evaluateUnit` za liste jedinki.

2.1.1 Inicijalizacija i selekcija

```
initialPopulation :: MonadEvolution u v m => Int -> m [u]
initialPopulation = (`replicateM` newInitialUnit)

evaluatePopulation :: MonadEvolution u v m => [u] -> m [(u, v)]
```

```
evaluatePopulation = mapM (\u -> do
    v <- evaluateUnit u
    return (u, v))
```

`initialPopulation` generira traženi broj jedinki dok `evaluatePopulation` poziva `evaluateUnit` na svakoj jedinci u listi te vraća novu listu gdje su jedinke uparene s njihovim rezultatima. Ovo je primjer složenijih funkcija koje su i dalje parametrizirane nad implementacijom osnovnih funkcija.

Slijdeće funkcije implementiraju selekciju. Metoda koja se koristi je deterministička turnirska selekcija u kojoj se nasumično odabire podskup populacije te se uzima najbolja jedinka tog podskupa. Proces se ponavlja dok ne skupimo dovoljno pobjednika.

```
data GeneticConfiguration = GeneticConfiguration
    { tournamentPoolSize        :: Int
    , persistedPoolSize         :: Int
    , initialStalenessTolerance :: Int }
    deriving (Eq, Ord, Read, Show)

selectBest :: Ord v => [(a, v)] -> (a, v)
selectBest = maximumBy (comparing snd)

tournamentSelectUnit ::
    (MonadRandom m, Ord v) => Int -> [(a, v)] -> m (a, v)
tournamentSelectUnit poolSize evaluatedPop =
    selectBest <$> uniformRandomN poolSize evaluatedPop

tournamentSelectPool ::
    ( MonadRandom m
    , MonadEffect (ReadEnv GeneticConfiguration) m
    , Ord v )
    => [(u, v)] -> m [(u, v)]
tournamentSelectPool evaluatedPop = do
    GeneticConfiguration{..} <- readEnv
    replicateM persistedPoolSize
        (tournamentSelectUnit tournamentPoolSize evaluatedPop)
```

`selectBest` daje najbolju jedinku na bazi njenog rezultata. `tournamentSelectUnit` implementira jednu iteraciju turnirske selekcije. Vidimo da ta funkcija od svog konteksta zahtjeva mogućnosti generiranja slučajnih brojeva (`MonadRandom m`). Konačno, `tournamentSelectPool` ponavlja turnirsku selekciju traženi broj puta. Vidimo da osim zahtjeva za slučajnim brojevima, također se zahtjeva da okruženje sadrži konfiguraciju genetskog algoritma. Jedan od parametara u konfiguraciji je veličina pojedinog turnira (`tournamentPoolSize`) dok je drugi broj jedinki koje selektiramo za sljedeću generaciju (`persistedPoolSize`).

2.1.2 Mutacija i križanje

Još jedna funkcija koja se koristi bez eksplicitne implementacije je

```
evolutionActivity ::
    MonadEvolution u v m => EvolutionActivity u v -> m ()
```

Ona sliži da algoritam za vrijeme rada može prijaviti što se u nekom trenutku događa. Ta informacija je korisna kao dijagnostika i kao sredstvo kojim pratimo napredak algoritma.

Slijede funkcije mutacije i križanja populacije:

```
crossPool :: MonadEvolution u v m => [(u, v)] -> m [(u, v)]
crossPool pool = sequence $ crossAndLog <$> pool <*> pool
    where crossAndLog a@(u1, _) b@(u2, _) = do
              newUnit  <- crossUnits u1 u2
              newValue <- evaluateUnit newUnit
              evolutionActivity
                  (CrossActivity (a, b) (newUnit, newValue))
              return (newUnit, newValue)


mutatePool :: MonadEvolution u v m => [(u, v)] -> m [(u, v)]
mutatePool = mapM mutateAndLog
    where mutateAndLog a@(u, _) = do
              newUnit  <- mutateUnit u
              newValue <- evaluateUnit newUnit
              evolutionActivity
                  (MutationActivity a (newUnit, newValue))
              return (newUnit, newValue)
```

Obje funkcije pozivaju odgovarajuću osnovnu funkciju na svim jedinkama (ili svim parovima) iz populacije, nakon tog vrednuju novonastalu jedinku te prijavljuju aktivnost. Vođenje evidencije o aktivnostima unutar algoritma otvaramo mogućnost praćenja statistike o efikasnosti križanja i mutacije, a te informacije se mogu koristiti za dinamičko podešavanje parametara u konfiguraciji. Ukoliko ta funkcionalnost nije potrebna moguće je ignorirati te informacije, a zahvaljujući tome da je Haskell lazy jezik, ne plaćamo cijenu generiranja zapisa.

2.1.3 Iteracija

Sljedeće je definirana funkcija koja iz jedne generacije generira sljedeću.

```
advancePopulation :: MonadEvolution u v m => [(u, v)] -> m [(u, v)]
advancePopulation evaluatedPop = do
    let alpha = selectBest evaluatedPop
```

```
    selected <- (alpha :) <$> tournamentSelectPool evaluatedPop
    crossed <- crossPool selected
    mutated <- mutatePool selected
    return (selected <> crossed <> mutated)
```

Funkcija prvo odabire najbolju jedinku kojoj garantira opstanak (*elitizam*), zatim turnirskom selekcijom odvaja dio populacija koji preživljava te ga križa i mutira. Konačna generacija sastoji se od odabranih starih jedinki, križanih jedinki i mutiranih jedinki.

Funkcija `iterativeEvolution` implementira finalni algoritam.

```
iterativeEvolution :: MonadEvolution u v m => m ()
iterativeEvolution = do
    GeneticConfiguration{..} <- readEnv
    initPop <- initialPopulation persistedPoolSize
    evaluatedPop <- evaluatePopulation initPop
    void $ loopM evaluatedPop $ \pop -> do
        newPop <- advancePopulation pop
        evolutionActivity (NewGenerationActivity newPop)
        return newPop
```

Generira inicijalnu populaciju te iterativno poziva `advancePopulation`. Ne postoji uvjet zaustavljanja, ali funkcija u svakoj iteraciji javlja trenutnu generaciju.

2.2 Specijacija

U biologiji, specijacija je evolucijski proces kojim nastaju nove vrste. Postoji nekoliko načina na koji nova vrsta može nastati. Primjerice, dio populacije može biti geografski odvojen od ostatka pa njihova evolucija prirodno divergira. Pozitivna posljedica postojanja vrsta je raznolikost, što čini populacije otpornijima na promjene u okruženju. U slučaju optimizacije, naše okruženje se uglavnom ne mijenja, ali genetska raznolikost i dalje ima svoje prednosti.

Jedan od problema s kojima se heurističke metode moraju boriti je stagnacija. Osnovni kompromis svakog heurističkog algoritma je između što većeg pokrivanja prostora rješenja i konvergiranja prema nekom konkretnom optimumu. U genetskom algoritmu mutacija uglavnom ima ulogu pretraživanja prostora, dok križanje pomaže konvergenciji. Ukoliko je učinak mutacije preslab, evolucija može zapeti u lokalnim maksimumima. S druge strane, preveliki faktor mutacije može degenerirati algoritam u neusmjereno nasumično traženje rješenja. U složenim problemima, pogotovo onima genetskog programiranja, razlika između dva potencijalno dobra rješenja može biti vrlo velika. Ukoliko evolucija zapne na jednom od njih, postoji velika vjerojatnost da drugo neće biti otkriveno.

Jedan od načina za suprotstavljanje ovom problemu je dinamično povećanje faktora mutacije kad je detektirana stagnacija. Drugi način je ponovno pokretanje algoritma od početka. Time se na neki način generira nova vrsta koja potencijalno bolje rješava početni problem. U procesu se gubi prijašnje rješenje. Pristup koji je uzet u ovom radu je nešto drukčiji.

Iteriranjem genetskog algoritma pratimo koliko je generacija prošlo s jednakim najboljim rezultatom. Kad taj broj postane veći od nekog definiranog praga, sprema se zadnja generacija te se proglašava vrstom nulte razine. Algoritam kreće od početka, generirajući novu vrstu nulte razine. Kad su dvije vrste nulte razine generirane, one se križaju te se algoritam nastavlja na rezultatu križanja do sljedeće stagnacije. Tada se najbolji rezultat proglašava vrstom prve razine. Procedura se rekurzivno ponavlja. Svaki put kad su generirane dvije vrste n-te razine, one se križaju i evoluiraju u vrstu $(n+1)$-ve razine. Ako takvih nema, generira se nova vrsta nulte razine.

U pravilu, vrste više razine uglavnom postižu bolje rezultate, ali imaju i veći potencijal zapinjanja. Dvije vrste iste razine nisu imale nikakvih doticaja pa se očekuje da su njihova rješenja različita. Križanjem takve dvije vrste sprječava se stagnacija. Važno je napomenuti da generiranje vrste n-te razine traje 2^n vremena. Ukoliko n postane dovoljno velik, to može biti preskupo.

Funckija `evolveSpecies` generira jednu vrstu nulte razine. Implementirana je gore opisana procedura u kojoj se prati koliko je generacija prošlo s istim rezultatom.

```
evolveSpecies :: MonadEvolution u v m => [(u, v)] -> Int -> m [(u, v)]
evolveSpecies evaluatedPop stalenessTolerance = do
    (_, spec) <- whileM
        (0, evaluatedPop)
        (\(staleness, _) -> return (staleness < stalenessTolerance)) $
        \(staleness, pop) -> do
            let (_, lastBestScore) = selectBest pop
            newPop <- advancePopulation pop
            evolutionActivity (NewGenerationActivity newPop)
            let (_, newBestScore) = selectBest newPop
            if lastBestScore == newBestScore then
                return (staleness + 1, newPop)
            else return (0, newPop)
    return spec
```

Nakon toga, implementirana je funkcija za spajanje dvije vrste. Slična je funkciji za jednu iteraciju genetskog algoritma.

```
mergeSpecies :: MonadEvolution u v m
    => Int -> [(u, v)] -> [(u, v)] -> m (Int, [(u, v)])
mergeSpecies newLvl spec1 spec2 = do
    let alpha1 = selectBest spec1
    champions1 <- (alpha1 :) <$> tournamentSelectPool  spec1
```

```
    let alpha2 = selectBest spec2
    champions2 <- (alpha2 :) <$> tournamentSelectPool spec2
    let champs = champions1 <> champions2
    crossed <- crossPool champs
    mutated <- mutatePool champs
    newSpec <- evolveSpecies (champs <> crossed <> mutated) newLvl
    return (newLvl, newSpec)
```

Ostatak opisane metode implementiran je u sljedećoj funkciji:

```
speciesEvolution :: MonadEvolution u v m => m ()
speciesEvolution =
    void $ loopM [] $ \specList -> case specList of
        (lvl1, spec1) : (lvl2, spec2) : rest | lvl1 == lvl2 -> do
            let newLvl = lvl1 + 1
            newSpec :: (Int, [(u, v)]) <- mergeSpecies newLvl spec1 spec2
            evolutionActivity (SpeciesStackChange (newSpec : rest))
            return (newSpec : rest)
        rest -> do
            GeneticConfiguration{..} <- readEnv
            initPop <- initialPopulation persistedPoolSize
            evaluatedPop <- evaluatePopulation initPop
            newSpec <- evolveSpecies evaluatedPop
                initialStalenessTolerance
            evolutionActivity (SpeciesStackChange
                ((initialStalenessTolerance, newSpec) : rest))
            return ((initialStalenessTolerance, newSpec) : rest)
```

Stablo vrsta se pamti kao stog s invarijantom da je uvijek sortiran tako da su na vrhu vrste najniže razine. Kad se generira nova vrsta nulte razine ona ide na vrh stoga (što održava invarijantu). Kad se na vrhu nađu dvije vrste n-te razine, one se spajaju u vrstu $(n+1)$-ve razine koja se stavlja na vrh stoga. U tom slučaju invarijanta jedino ne vrijedi ako se na stogu nalazila još barem jedna vrsta n te razine, ali u tom slučaju su prije dolaska nove vrste n-te razine na stogu već bile dvije takve vrste pa bi se one već prije spojile.

2.3 Distribuirani sustav

Ovakav pristup također ima prednost lake paralelizacije. Svaki zadatak razvoja jedne vrste do stagnacije je potpuno neovisan o drugima pa ih se više može istovremeno izvršavati na više procesorskih jedinica, ili na više računala. U ovom radu distribuirana arhitektura implementirana je pomoću RabbitMQ[4] sustava. Glavni proces vodi stablo vrsta koje započinje kao jedan prazan čvor. Taj čvor se pretvara u zadatak koji se stavlja u red zadataka. Kad se novi klijent uključi u mrežu on uzima

taj zadatak, razvije novu vrstu te ju vraća glavnom procesu. Kad je stablo puno ono postaje lijevo podstablo ispod novog stabla te se generira prazno desno podstablo. Generiraju se zadaci za sve prazne čvorove u novom stablu te se ono na isti način popunjava.

2.4 Jednostavni primjer

Kao primjer, genetski algoritam je primijenjen na jednostavan problem traženja broja. Jedinka je predstavljena realnim brojem, a njezin fitness je obrnuto proporcionalan udaljenosti od traženog broja (100 u ovom slučaju). Mutacija je pomak za neki broj u intervalu $[-1, 1]$, a križanje je srednja vrijednost. Inicijalne jedinke su brojevi iz intervala $[0, 200]$.

```
mutateNumber :: MonadRandom m => Double -> m Double
mutateNumber x = (x +) <$> getRandomR (-1, 1)

crossNumbers :: Monad m => Double -> Double -> m Double
crossNumbers x y = return $ (x + y) / 2

evaluateNumber :: Monad m => Double -> m Double
evaluateNumber x = return $ - (abs (100 - x))

randomNumber :: MonadRandom m => m Double
randomNumber = getRandomR (0, 200)
```

Funkcija `evoHandler` kombinira te četiri funkcije.

```
evoHandler ::
    MonadRandom m
    => Effect (Evolution Double Double) method 'Msg
    -> m (Effect (Evolution Double Double) method 'Res)
evoHandler (MutateUnitMsg u) = MutateUnitRes <$> mutateNumber u
evoHandler (CrossUnitsMsg u1 u2) =
    CrossUnitsRes <$> crossNumbers u1 u2
evoHandler (EvaluateUnitMsg u) = EvaluateUnitRes <$> evaluateNumber u
evoHandler NewInitialUnitMsg = NewInitialUnitRes <$> randomNumber
```

Također definirana je funkcija koja će pratiti aktivnost algoritma.

```
printActivity :: MonadIO m => EvolutionActivity Double Double -> m ()
printActivity (NewGenerationActivity gen) = print $ selectBest gen
printActivity (SpeciesStackChange specs) =
    forM_ specs $ \(lev, pop) ->
        putStrLn $ show (lev - 10) <> "␣" <> show (selectBest pop)
printActivity _ = return ()
```

Konačno, funkcija `test` pokreće algoritam.

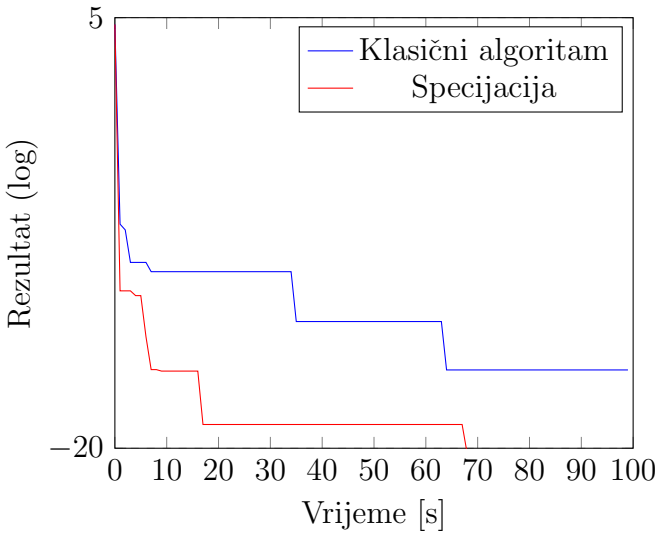
```
test :: IO ()
test =
    handleReadEnv (return (GeneticConfiguration 10 20 10)) $
    handleEvolution evoHandler printActivity
    speciesEvolution
```

Na ovako jednostavnom primjeru može se samo vidjeti da algoritam doista funkcionira. Spajanje vrsta također povremeno urodi plodom. Npr.

```
(99.99999998176624,-1.8233762943964393e-8)
...
5 (99.99999998176624,-1.8233762943964393e-8)
5 (100.00000000594719,-5.947185854893178e-9)
(99.99999999990195,-9.805489753489383e-11)
...
6 (99.99999999990195,-9.805489753489383e-11)
```

Ovdje se vidi da u stablu imamo dvije vrste pete razine, koje svaka za sebe dosta dobro rješava problem. Ipak, jedna od njih se približava broju 100 s jedne strane, a druga s druge. Njihovim spajanjem dobivamo novu vrstu koja daje još bolje rješenje. Naravno, očekuje se da će i standardni algoritam ovdje profinjavati rješenje jer postoji samo jedan lokalni maksimum koji je ujedno i globalni.

Usporedbom brzine konvergencije ta dva pristupa, dobiven je sljedeći graf.



Poglavlje 3

Primjer: Regularni izrazi

Regularni izrazi predstavljaju mali domenski programski jezik za opisivanje uzoraka u tekstu. Njihova implementacija je dana za gotovo svaki korišteni programski jezik, a u velikom broju njih je uključena i u standardne biblioteke. Imaju širok spektar primjena: od validacije unosa do pretraživanja teksta. U ovom radu, regularni izrazi se koriste kao primjer složenijeg genoma kojeg će algoritam evoluirati. Reprezentirani su kao sintaksna stabla na kojima je potrebno definirati genetske operatore. Korištenje genetskih algoritama za generiranje regularnih izraza korišteno je primjerice u detekciji neželjene pošte u [7].

3.1 Definicija

Niz znakova skupa A označava se $[A]$ i definiran je na sljedeći način

- $[]$ je niz znakova skupa A i predstavlja prazan niz
- neka je $s \in [A]$ niz znakova i $c \in A$ znak, tada je $c : s$ niz znakova i predstavlja znak c dodan na početak niza s

Neka su $n \in \mathbb{N}_0$, $a \in [A]$ i $b \in [A]$. S $n \cdot a$ označavamo niz a ponovljen n puta. S ab označavamo konkatenaciju nizova a i b.

Neka je Σ abeceda. Regularni izrazi su elementi skupa $RegEx \subset [\Sigma \cup \{\epsilon, \phi, |, ^*, (,)\}]$ Sintaksa regularnih izraza definirana je rekurzivno:

- $a \in \Sigma$ je regularni izraz
- znak ϵ je regularni izraz
- znak ϕ je regularni izrazi

- neka su R i S regularni izrazi:
  - RS je regularni izraz
  - $R|S$ je regularni izraz
  - R^* je regularni izraz
  - (R) je regularni izraz

Semantika regularnih izraza dana je funkcijom $matches : RegEx \rightarrow \mathcal{P}([\Sigma])$ definiranom po slučajevima[6]:

- $matches(\phi) = \phi$
- $matches(\epsilon) = \{[]\}$
- neka je $c \in \Sigma$, tada $matches(c) = \{c : []\}$
- $matches(RS) = \{ab \mid a \in matches(R), b \in matches(S)\}$
- $matches(R|S) = matches(R) \cup matches(S)$
- $matches(R^*) = \{n \cdot s \mid n \in \mathbb{N}_0, s \in matches(R)\}$
- $matches((R)) = matches(R)$

Kaže se da regularni izraz R prihvaća neki niz s ako $s \in matches(R)$.

Intuitivno, ϵ izraz prihvaća samo prazan niz, ϕ ne prihvaća nikakav niz, $c \in \Sigma$ kao regularni izraz prihvaća samo niz koji sadrži samo znak c. RS prihvaća niz ukoliko se on može podijeliti na dva djela takva da R prihvaća prvi, a S drugi dio. $R|S$ prihvaća niz ukoliko R ili S prihvaćaju niz. R^* prihvaća niz koji je nula ili više ponavljanja uzorka R. Zagrade nemaju neko semantičko značenje već su tu za grupiranje operacija.

3.2 Proširenje

Regularni izrazi uglavnom su implementirani s još nekim dodatnim operacijama. One često omogućuju provjeravanje i ne-regularnih jezika, no neke od njih su samo sintaktički dodaci koji ne mijenjaju ekspresivnost jezika, ali olakšavaju posao programeru. S istom idejom, i u ovom radu je proširena definicija regularnih izraza. Dodaju se sljedeće sintaktičke forme:

- . je regularni izraz koji prihvaća bilo koji znak

- $R?$ je regularni izraz koji prihvaća niz koji je nula ili jedno pojavljivanje uzorka R
- $[a - b]$ prihvaća bilo koji znak koji je u intervalu između znaka a i b uključivo

Jasno je da su $R?$ i $R|\epsilon$ ekvivalentni. Slično, $[a - b]$ se može raspisati kao operacija | primijenjena na sve znakove u tom intervalu, a . je ekvivalentno $[a_{min} - a_{max}]$ gdje su a_{min} i a_{max} najmanji i najveći znakovi u abecedi. Ta notacija jedino ima smisla ako postoji totalni uređaj na abecedi, a u implementacijama se uglavnom uzima brojčana vrijednost znakova.

3.3 Sintaksna stabla

Regularni izrazi definirani su ovako u kodu:

```
data RegEx =
      Failed
    | Empty
    | AnyChar
    | Literal Char
    | Concat RegEx RegEx
    | Alternate RegEx RegEx
    | Kleene RegEx
    | Optional RegEx
    | Range Char Char
    deriving (Eq, Ord, Read, Show)
```

3.3.1 Prihvaćanje nizova znakova

Sljedeće, implementirana je funkcija koja provjerava prihvaća li regularni izraz neki niz znakova. Promatranjem slučaja u kojem je izraz oblika RS vidimo vidi se koji je glavni problem i kojim pristupom je riješen. Naivna implementacija bi mogla uzimati prefikse niza tako dugo dok R ne prihvati jedan od njih, pa prihvatiti niz ukoliko S prihvaća ostatak, no ako je izraz primjerice $(a^*)b$ vidi se da takva implementacija neće prihvatiti niz aab. Problem je u tome da već prvi neprazni prefiks niza odgovara izrazu (a^*), ali ostatak niza (ab) ne odgovara izrazu b. Lako je vidjeti da semantička interpretacija tog izraza doista uključuje niz aab jer ga je moguće podijeliti u nizove aa i b koji su prihvaćeni izrazima a^* i b.

Vođeno time, funkcija je definirana sa sljedećim tipom

```
match :: RegEx -> [Char] -> [[Char]]
```

Ideja je da za dani niz funkcija vraća sve moguće sufikse koji mogu ostati nakon što je izraz prihvatio neki prefiks. Na primjer: `match (Kleene 'a') "aaa"` je `["aaa", "aa", "a", ""]` pokazujući da a^* prihvaća znak a nula, jedan, dva ili tri puta. S takvom definicijom izraz `r` prihvaća niz `s` ako je jedan od sufiksa u `match r s` prazan niz. Slijedi definicija funkcije `match`.

```
match Failed _ = []
match Empty s = [s]
match AnyChar (_ : ss) = [ss]
match AnyChar []       = []
match (Literal c) (s : ss) | c == s = [ss]
match (Literal _) _ = []
```

`Failed` simbolizira ϕ koji ne prihvaća nikakav niz (prazna lista sufiksa). `Empty` (ϵ) prihvaća samo prazan samo prazan niz, što znači da jedini prefiks koji ostaje je sam niz. `AnyChar` (.) prihvaća bilo koji znak što znači da ako je niz neprazan, sufiks koji ostaje je sve osim prvog znaka. Ukoliko je niz prazan, on je odbijen. Znak `c` prihvaća samo niz koji počinje s istim znakom, u suprotnom odbija niz.

```
match (Alternate r1 r2) s = match r1 s ++ match r2 s
```

Kod `Alternate r1 r2` ($R_1|R_2$), ostaju svi sufiksi koje `r1` ili `r2` mogu ostaviti.

```
match (Concat r1 r2) s = [s'' | s' <- match r1 s, s'' <- match r2 s']
```

`Concat r1 r2` (R_1R_2) je nešto zanimljiviji. Za svaki sufiks `s'` koji `r1` ostavlja, vraća sve sufikse `s''` koje `r2` ostavlja na `s'`.

```
match (Kleene r) s =
    ordNub $ s : [s'
        | x <- match r s
        , x /= s
        , s' <- match (Kleene r) x]
```

`Kleene r` (R^*) prije svega prihvaća i prazan niz pa se u skup sufiksa odmah dodaje i početni niz. Zatim, gledaju se svi sufiksi koji ostaju iza `r`, za svaki od njih opet se rekurzivno poziva `match (Kleene r)`. Dodana je i provjera `x /= s` s kojom se osigurava da je `r` doista "pojeo" dio niza, u suprotnom rekurzija nikad neće stati. `ordNub` funkcija miče duplikate i dodana je jer je eksperimentalno pokazano da to drastično ubrzava program. Uzorci kao $.^{**}$ generiraju jako velike liste pune duplikata koje ova funkcija eliminira.

Implementacija `Optional r` ($R?$) i `Range c1 c2` ($[c_1 - c_2]$) je jednostavna.

```
match (Optional r) s = s : match r s
match (Range c1 c2) (c : ss) = [ss | c1 <= c && c2 >= c]
match (Range _ _) [] = []
```

3.3.2 Inicijalna populacija

Prvo je definirana funkcija koja generira inicijalne, nasumične regularne izraze.

```
randomRegex :: MonadRandom m => Int -> m RegEx
```

Funkcija prima parametar koji govori koja je veličina stabla koje želimo generirati. Veličina u ovom slučaju odgovara broju čvorova u stablu. Slijedi implementacija

```
randomRegex 0 = getRandomR (1 :: Int, 10) >>= \case
    1 -> return AnyChar
    2 -> do
        x <- uniform alphaNums
        y <- uniform alphaNums
        return $ Range (min x y) (max x y)
    _ -> Literal <$> uniform alphaNums
randomRegex n = getRandomR (1 :: Int, 4) >>= \case
    1 -> Concat    <$> randomRegex left <*> randomRegex right
    2 -> Alternate <$> randomRegex left <*> randomRegex right
    3 -> Kleene    <$> randomRegex (n - 1)
    4 -> Optional  <$> randomRegex (n - 1)
    _ -> error "Can't␣happen"
    where left  = (n - 1) `div` 2
          right = (n - 1) - left
```

Ako je tražena veličina 0, generira se neki od listova i to tako da se s vjerojatnošću $\frac{8}{10}$ generira znak iz abecede (u ovom slučaju slova i brojevi). ϵ i ϕ se nikad ne generiraju jer ti regularni izrazi uglavnom nikad nisu direktno implementirani u programskim jezicima.

Za veličinu veću od 0, generira se neki od granajućih izraza. Za unarne čvorove rekurzivno se poziva funkcija s veličinom umanjenom za jedan, dok se za binarne čvorove veličina dijeli na dvije grane.

3.3.3 Mutacija i križanje

Za mutiranje stabla ideja je sljedeća: odabire se neko podstablo, izračuna se njegova veličina s te se zamjenjuje s slučajno generiranim stablom čija veličine s' td. $max(s-2,0) \leq s' \leq s+2$. Križanje jednostavno odabire podstablo lijevog roditelja i zamjenjuje ga slučajnim podstablom desnog.

Potrebno je definirati funkciju koja računa veličinu stabla. Nije dovoljna samo ukupna veličina jer je od interesa veličina svakog pod stabla. U tu svrhu gradimo stablo veličina:

```
data TreeSize = TreeSize Int [TreeSize]
    deriving (Eq, Ord, Read, Show)
```

```
weighRegex :: RegEx -> TreeSize
weighRegex Failed = TreeSize 1 []
weighRegex Empty  = TreeSize 1 []
weighRegex AnyChar  = TreeSize 1 []
weighRegex (Literal _) = TreeSize 1 []
weighRegex (Range _ _) = TreeSize 1 []
weighRegex (Concat l r) = let tl@(TreeSize sl _) = weighRegex l
                              tr@(TreeSize sr _) = weighRegex r
                          in  TreeSize (sl + sr + 1) [tl, tr]
weighRegex (Alternate l r) = let tl@(TreeSize sl _) = weighRegex l
                                 tr@(TreeSize sr _) = weighRegex r
                             in  TreeSize (sl + sr + 1) [tl, tr]
weighRegex (Kleene l) = let tl@(TreeSize sl _) = weighRegex l
                        in  TreeSize (sl + 1) [tl]
weighRegex (Optional l) = let tl@(TreeSize sl _) = weighRegex l
                          in  TreeSize (sl + 1) [tl]
```

Listovi imaju veličinu 1 i nemaju podstabla, dok za granajuće izraze imamo rekurzivne pozive koji generiraju stablo veličina s korijenom koji je suma sve djece plus 1 za sam čvor. U mutaciji i križanju je potrebna funkcionalnost odabira nasumičnog podstabla što upućuje na funkciju ovog tipa

```
randomSubtree :: MonadRandom m => RegEx -> m RegEx
```

Problem je da osim dohvaćanja podstabla, potrebna je i mogućnost zamjene istog s nekim drugim. Zato je `randomSubtree` definirana ovako

```
randomSubtree :: MonadRandom m => RegEx -> m (RegEx, RegEx -> RegEx)
```

Prvi član para je nasumično podstablo, dok je drugi član funkcija koja vraća originalni izraz s odabranim podstablom zamijenjenim parametrom. Intuitivno, rezultat ove funkcije je dekonstrukcija stabla tako da je iz njega izvađen jedan izraz i vraćen je ostatak koji na tom mjestu ima rupu. Vrijedi: Za regularni izraz r, neka je $(r', f) = randomSubtree(r)$, tada $r = f(r')$. Slijedi implementacija funkcije

```
randomSubtree :: MonadRandom m => RegEx -> m (RegEx, RegEx -> RegEx)
randomSubtree regex = withWeight regex (weighRegex regex)
```

Računa se veličina stabla te se paralelno prolazi kroz sam izraz i stablo veličina u funkciji `withWeight`.

```
    where withWeight Failed        _ = return (Failed, identity)
          withWeight Empty         _ = return (Empty, identity)
          withWeight AnyChar       _ = return (AnyChar, identity)
          withWeight (Literal c)   _ = return (Literal c, identity)
          withWeight (Range c1 c2) _ = return (Range c1 c2, identity)
```

Negranajući slučajevi mogu odabrati samo sebe, pritom od originalnog izraza ne ostaje ništa. To znači da kad bi taj podizraz zamijenili nekim drugim, novo stablo bilo bi jednakom tom novom podizrazu. Zato, funkcija koju vraćamo je identiteta (`identity Failed == Failed`, `identity Empty == Empty`...).

```
withWeight
    (Concat l r)
    (TreeSize cs [lts@(TreeSize ls _), rts@(TreeSize _ _)]) = do
        choice <- getRandomR (1, cs)
        if choice == 1 then return (Concat l r, identity)
        else if choice - 1 < ls then do
            (lsub, lfun) <- withWeight l lts
            return (lsub, \x -> Concat (lfun x) r)
        else do
            (rsub, rfun) <- withWeight r rts
            return (rsub, Concat l . rfun)
```

Kod konkatenacije algoritam je sljedeći. Za veličinu stabla cs odabire se cijeli broj iz $[1, cs]$. Ako je taj broj 1, odabrano je cijelo stablo. Tada je rezultat isti kao i kod listova. Za ostale odabire, smatra se da je odabrano neko podstablo iz lijeve ili desne grane (vjerojatnost je ovisna o veličini tih grana). Rekurzivnim pozivom odabire se to podstablo i funkcija za zamjenu. U konačnom rezultatu potrebno je još uračunati da dano podstablo pripada onom na kojem je funkcija originalno pozvana pa vraćamo novu funkciju zamjene koja još dodatno zamata parametar.

Procedura za alternaciju je identična

```
withWeight
    (Alternate l r)
    (TreeSize cs [lts@(TreeSize ls _), rts@(TreeSize _ _)]) = do
        choice <- getRandomR (1, cs)
        if choice == 1 then return (Alternate l r, identity)
        else if choice - 1 < ls then do
            (lsub, lfun) <- withWeight l lts
            return (lsub, \x -> Alternate (lfun x) r)
        else do
            (rsub, rfun) <- withWeight r rts
            return (rsub, Alternate l . rfun)
```

Za unarne izraze radi se skoro ista stvar uz pojednostavljenje jer postoji samo jedna grana.

```
withWeight (Kleene s) (TreeSize cs [sts@(TreeSize _ _)]) = do
    choice <- getRandomR (1, cs)
    if choice == 1 then return (Kleene s, identity)
    else do
        (sub, fun) <- withWeight s sts
```

```
        return (sub, Kleene . fun)
withWeight (Optional s) (TreeSize cs [sts@(TreeSize _ _)]) = do
    choice <- getRandomR (1, cs)
    if choice == 1 then return (Optional s, identity)
    else do
        (sub, fun) <- withWeight s sts
        return (sub, Optional . fun)
```

Tako definiranom funkcijom, implementacija mutacije i križanja je skoro trivijalna.

```
mutateRegex :: MonadRandom m => RegEx -> m RegEx
mutateRegex regex = do
    (sub, fun) <- randomSubtree regex
    let TreeSize s _ = weighRegex sub
    fun <$> (randomRegex =<< getRandomR (max (s - 2) 0, s + 2))

crossRegex :: MonadRandom m => RegEx -> RegEx -> m RegEx
crossRegex a b = do
    (sub, _) <- randomSubtree a
    (_, fun) <- randomSubtree b
    return (fun sub)
```

Kao što je ranije opisano, mutacija odabire neko podstablo, računa veličinu te ga zamjenjuje novim podstablom slične veličine. Križanje podstablo iz lijevog roditelja zamjenjuje podstablom desnog.

3.4 Problem: Prepoznavanje jezika

3.4.1 Opis

Problem na kojem je testiran algoritam je sljedeći: Možemo li generirati regularni izraz koji prihvaća riječi engleskog jezika, ali odbija riječi njemačkog?

Engleski i njemački nemaju disjunktne skupove riječi pa je savršena klasifikacija nemoguća. Također, pravila konstrukcije riječi te njihova porijekla su veoma složena pa ostaje upitno koliko dobra aproksimacija uopće može biti izražena jezikom regularnih izraza.

Algoritam "trenira" na skupovima od 300 engleskih i 300 njemačkih riječi. Cilj je prihvatiti svaku englesku, a odbiti svaku njemačku riječ.

3.4.2 Implementacija

Definirana je funkcija dobrote. Dana jedinka testirana je na svim engleskim i njemačkim riječima iz skupa za trening. Kao osnovna ocjena, broju prihvaćenih engleskih riječi

oduzet je broj prihvaćenih njemačkih riječi. Problem s tako jednostavnom evaluacijom je da je moguće konstruirati izraz koji je ekvivalentan alternaciji svih engleskih riječi iz skupa te će taj izraz dobiti savršen rezultat, iako očito ne obavlja željenu funkciju. To je klasični problem *overfittinga* poznat u strojnom učenju. U ovom slučaju ipak postoji i veći problem. Tako veliki regularni izrazi su spori za evaluaciju. Algoritam koji provjerava prihvaćanje neke riječi nije najbrži te je čak eksponencijalne složenosti u nekim slučajevima.

Uz osnovni rezultat, ocjenjuje se još i veličina izraza u kojoj se nagrađuju manji izrazi. Logično je pitati se u kojem omjeru je najbolje vrednovati osnovni rezultat i veličinu izraza. Ipak, bolji osnovni rezultat na neki način treba uvijek biti više vrednovan od goreg, ali manjeg izraza. Stoga, odabran je $\mathbb{Z} \times \mathbb{Z}$ kao skup rezultata. Prvi element para je osnovni rezultat, dok je drugi element veličina izraza pomnožena s -1. Uređaj odabran za taj skup je leksikografski pa je time dobiveno željeno svojstvo.

Time je ostvareno da će evolucijski proces smanjivati veličinu izraza ukoliko ne može poboljšati rezultat. K tome je dodan uvjet da ako je veličina izraza veća od 100, onda je razlika veličine i 100 još dodatno oduzeta od osnovnog rezultata. Time je na neki način limitirana veličina izraza na 100, osim u slučajevima kad je moguće drastičnije poboljšanje osnovnog rezultata nauštrb veličine izraza.

Zbog potencijalnih izraza za koje je potrebno eksponencijalno vrijeme za evaluaciju, dodan je i brojač vremena koji diskvalificira izraz ukoliko mu treba predugo da bude evaluiran. Slijedi implementacija funkcije evaluacije

```
evaluateRegex ::
    (MonadRandom m, MonadIO m)
    => [String] -> [String] -> RegEx -> m (Int, Int)
evaluateRegex eng ger rgx = do
    maybeScore <- liftIO $ timeout 1000 $ do
        let engMatch = length $ filter (isMatch rgx) eng
            gerMatch = length $ filter (isMatch rgx) ger
        validity <- evaluate $ engMatch - gerMatch
        sizePenalty <- evaluate $
            let TreeSize s _ = weighRegex rgx in -(max s 10)
        return (validity + min 0 (sizePenalty + 100), sizePenalty)
    case maybeScore of
        Nothing ->
            return (-100, 0)
        Just s -> return s
```

Ukoliko je potrebno više od 1000 mikro sekundi za evaluaciju izraza, vraća se rezultat $(-100, 0)$. Uz gore opisanu proceduru još je dodan slučaj u kojem se izrazi manji od 10 ne penaliziraju. Time se dodaje više materijala za križanje u prve generacije.

3.4.3 Rezultati

Evoluiranje izraza je relativno spor proces no budući da brzina poboljšanja očekivano pada s vremenom, unutar pola sata dobivena su razumno dobra rješenja. U ispisu su vidljive zanimljive situacije kao

```
Free
|
+- Free
|  |
|  +- Queued
|  |  |
|  |  +- Calculated ((69,-21))
|  |  |
|  |  `- Calculated ((190,-69))
|  |
|  `- Queued
|     |
|     +- Calculated ((170,-92))
|     |
|     `- Calculated ((179,-88))
...

=============================================

Free
|
+- Free
|  |
|  +- Calculated ((195,-76))
|  |
|  `- Queued
|     |
|     +- Calculated ((170,-92))
|     |
|     `- Calculated ((179,-88))
...
```

Gdje se vidi da su se dvije stagnirane vrste na rezultatima $(69, -21)$ i $(190, -69)$ spojile u novu vrstu s rezultatom $(195, -76)$.

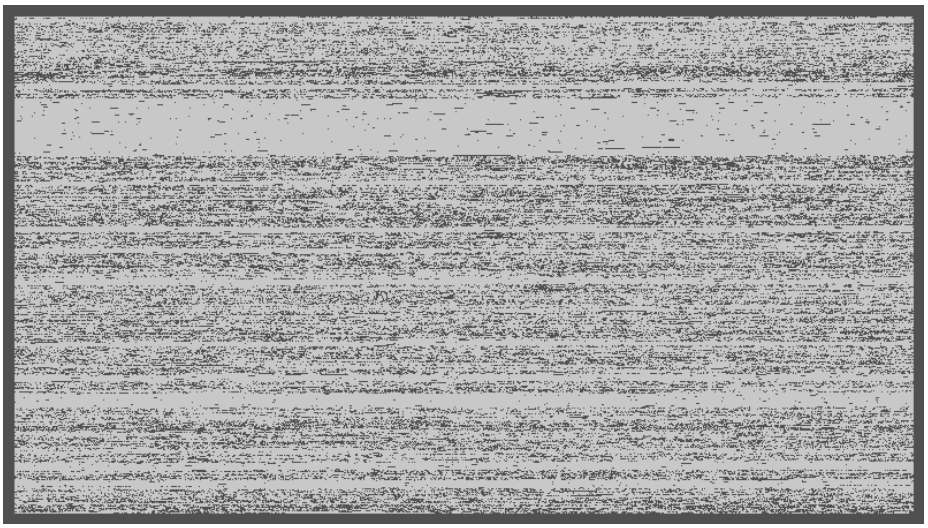
Dužim razvojem dobiven je sljedeći izraz s rezultatom $(223, -64)$.

```
((0|(0|(([ -d]|([f-w]|y)))*))|((([&-y]|y))*([Q-c](.(.s*)?)?|.((y|s.((
y|x)|s))|o(,|(([&-y]|s))*)))|(([ -d]|x*)|y)(c(([&-y]|x))*)?))
```

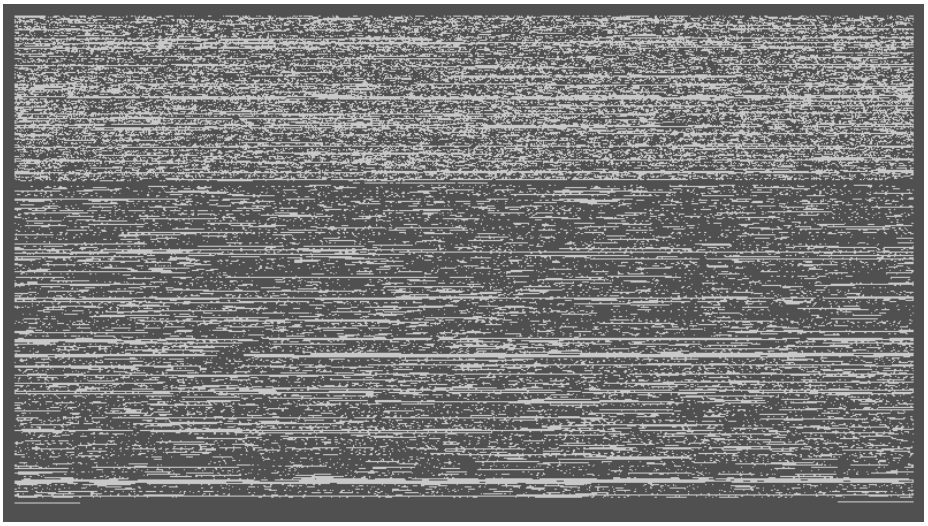
Taj rezultat predstavlja približno 75% točnost na testnim primjerima.

Izraz je testiran na korpusu od približno 400000 engleskih i njemačkih riječi[1][3].

Generirane su slike na kojima bijeli piksel predstavlja riječ koju izraz prihvaća, dok crni pikseli predstavljaju ostale riječi.



Slika 3.1: Engleski korpus (75% riječi prihvaćeno)



Slika 3.2: Njemački korpus (33% riječi prihvaćeno)

3.5 Primjene

Generiranjem regularnih izraza genetskim algoritmom dobiven je efektivan način za prepoznavanje zajedničkih uzoraka u skupu primjera. Metoda je primjenjiva na problemima koji podnose djelomično točna rješenja.

3.5.1 Prepoznavanje jezika u rečenicama

Izraz koji prepoznaje jezik u pojedinačnim riječima s točnošću od 70-75% nije jako pouzdan, ali ako se isti izraz primjeni na nekoliko riječi u rečenici očekuje se pouzdanija procjena. Izraz je testiran na djelu teksta iz članka o genetskim algoritmima na Wikipediji [2]. Engleske rečenice prepoznate kao njemačke i njemačke rečenice prepoznate kao engleske su precrtane.

in a genetic algorithm a population of candidate solutions called individuals creatures or phenotypes to an optimization problem is evolved toward better solutions. each candidate solution has a set of properties its chromosomes or genotype which can be mutated and altered; traditionally solutions are represented in binary as strings of 0s and 1s but other encodings are also possible. the evolution usually starts from a population of randomly generated individuals and is an iterative process with the population in each iteration called a generation. ~~in each generation the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved.~~ the more fit individuals are stochastically selected from the current population and each individual's genome is modified recombined and possibly randomly mutated to form a new generation. ~~the new generation of candidate solutions is then used in the next iteration of the algorithm. commonly the algorithm terminates when either a maximum number of generations has been produced or a satisfactory fitness level has been reached for the population.~~ a typical genetic algorithm requires. a genetic representation of the solution domain. a fitness function to evaluate the solution domain. a standard representation of each candidate solution is as an array of bits. arrays of other types and structures can be used in essentially the same way. the main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size which facilitates simple crossover operations. variable length representations may also be used but crossover implementation is more complex in this case. tree-like representations are explored in genetic programming and graph-form representations are explored in evolutionary programming; a mix of both linear chromosomes and trees is explored in gene expression programming. once the genetic representation and the fitness function are defined a ga proceeds to initialize a population of solutions and then to improve it through repetitive application of the mutation crossover inversion and

selection operators. the population size depends on the nature of the problem but typically contains several hundreds or thousands of possible solutions. ~~often the initial population is generated randomly allowing the entire range of possible solutions the search space.~~ occasionally the solutions may be "seeded" in areas where optimal solutions are likely to be found. during each successive generation a portion of the existing population is selected to breed a new generation. individual solutions are selected through a fitness-based process where fitter solutions as measured by a fitness function are typically more likely to be selected. certain selection methods rate the fitness of each solution and preferentially select the best solutions. other methods rate only a random sample of the population as the former process may be very time-consuming. ~~the fitness function is defined over the genetic representation and measures the quality of the represented solution.~~ ~~the fitness function is always problem dependent.~~ for instance in the knapsack problem one wants to maximize the total value of objects that can be put in a knapsack of some fixed capacity. a representation of a solution might be an array of bits where each bit represents a different object and the value of the bit 0 or 1 represents whether or not the object is in the knapsack. not every such representation is valid as the size of objects may exceed the capacity of the knapsack. the fitness of the solution is the sum of values of all objects in the knapsack if the representation is valid or 0 otherwise. in some problems it is hard or even impossible to define the fitness expression; in these cases a simulation may be used to determine the fitness function value of a phenotype e. g. computational fluid dynamics is used to determine the air resistance of a vehicle whose shape is encoded as the phenotype or even interactive genetic algorithms are used. the next step is to generate a second generation population of solutions from those selected through a combination of genetic operators: crossover also called recombination and mutation. for each new solution to be produced a pair of "parent" solutions is selected for breeding from the pool selected previously. by producing a "child" solution using the above methods of crossover and mutation a new solution is created which typically shares many of the characteristics of its "parents". new parents are selected for each new child and the process continues until a new population of solutions of appropriate size is generated. although reproduction methods that are based on the use of two parents are more "biology inspired" some research[3][4] suggests that more than two "parents" generate higher quality chromosomes. these processes ultimately result in the next generation population of chromosomes that is different from the initial generation. generally the average fitness will have increased by this procedure for the population since only the best organisms from the first generation are selected for breeding along with a small proportion of less fit solutions. ~~these less fit solutions ensure genetic diversity within the genetic pool of the parents and therefore ensure the genetic diversity of the subsequent generation of children.~~ opinion is divided over the importance of crossover versus mutation. there are many references in fogel 2006 that support the importance of mutation-based search. although crossover and mutation are known as the main genetic operators it is possible to use other operators such as regrouping colonization-extinction or migration in genetic algorithms. [5] it is worth tuning parameters such as the mutation probability crossover probability and population size to find reasonable settings for the problem class being worked on. a very small mutation rate may lead to genetic drift which is non-ergodic in nature. a recombination rate that is too high may lead to premature convergence of the genetic algorithm. a mutation rate that is too high may lead to loss of good solutions unless elitist selection is employed.

evolutionare algorithmen werden vorrangig zur optimierung oder suche eingesetzt. konkrete probleme die mit ea gelost werden sind ausserst divers: z. die entwicklung von sensornetzen aktienmarktanalyse oder rna-strukturvorhersage. auch bei problemen uber deren beschaffenheit nur wenig wissen vorliegt konnen sie zufriedenstellende losungen finden. dies ist auf die eigenschaften ihres naturlichen vorbildes zuruckzufuhren. ~~in der biologischen evolution sind die gene von organismen naturlich vorkommenden mutationen ausgesetzt wodurch genetische variabilitat entsteht.~~ ~~mutationen konnen sich positiv negativ oder gar nicht auf erben auswirken.~~ da es zwischen erfolgreichen individuen zur fortpflanzung rekombination kommt konnen sich arten uber lange zeitraume an einen vorliegenden selektionsdruck anpassen z. klimaveranderungen oder die erschliessung einer okologischen nische. diese vereinfachte vorstellung wird in der informatik idealisiert und kunstlich im computer nachgebildet. dabei wird die gute eines losungskandidaten explizit mit einer fitnessfunktion berechnet sodass verschiedene kandidaten vergleichbar sind. in der praxis konnte z. die form einer autotur so optimiert werden dass der aerodynamische widerstand minimal wird. die eigenschaften einer potenziellen losung werden dabei im rechner als genom gespeichert. haufige problemreprasentationen sind genome aus binaren oder reellen zahlen oder eine reihenfolge bekannter elemente bei kombinatorischen problemen z. travelling salesman. die starken vereinfachungen die im vergleich zur evolution getroffen werden stellen ein problem in bezug auf die erforschung evolutionsbiologischer fragestellungen mit ea dar. ergebnisse konnen nicht einfach auf die komplexere natur ubertragen werden. evolutionare algorithmen unterscheiden sich untereinander vor allem in der jeweiligen genetischen reprasentation der fitnessfunktion und den genutzten genetischen operatoren: mutation rekombination und selektion. die rastrigin-funktion ist eine multimodale funktion da sie viele lokale extrema aufweist. dies stellt einen nachteil fur den rekombinationsoperator dar. ~~mutation und rekombination sind die suchoperatoren evolutionarer algorithmen mit denen der suchraum erkundet wird.~~ ihre anwendung auf losungskandidaten kann keine verbesserung garantieren allerdings erhalt der suchprozess durch die selektion eine richtung die bei erfolgreicher konzeption zum globalen optimum fuhrt. wahrend mit dem mutationsoperator vollig neue bereiche des suchraums erschlossen werden konnen ermoglicht die rekombination vor allem die zusammenfuhrung erfolgreicher schemata building-block-hypothese. eine erfolgreiche suche basiert also auf der kombination beider eigenschaften. der erfolg eines rekombinationsoperators hangt von der beschaffenheit der fitnesslandschaft je mehr lokale optima die fitnesslandschaft aufweist desto wahrscheinlicher erzeugt die rekombination aus zwei individuen die sich auf einem lokalen optimum befinden einen nachfahren im tal dazwischen. ~~mutation ist von dieser eigenschaft der fitnesslandschaft nahezu unabhangig.~~ der entwurf der verschiedenen komponenten bestimmt wie sich der evolutionare algorithmus bei der optimierung des gegebenen problems in bezug auf konvergenzverhalten benotigte rechenzeit und die erschliessung des problemraums verhalt. insbesondere mussen die genetischen operatoren sorgfaltig auf die zugrunde liegende reprasentation abgestimmt sein sodass sowohl die bekannten guten regionen des problemraums genutzt als auch die unbekannten regionen erkundet werden konnen. dabei spielen die beziehungen zwischen such- und problemraum eine rolle. im einfachsten fall entspricht der suchraum dem problemraum direkte problemreprasentation. das no-free-lunch-theorem der optimierung besagt dass alle optimierungsstrategien gleich effektiv sind wenn die menge aller optimierungsprobleme betrachtet wird. unter der gleichen voraussetzung ist auch kein evolutionarer algorithmus grundsatzlich besser als ein anderer. dies kann nur dann der fall sein wenn die menge aller probleme eingeschrankt wird. ~~genau das wird in der praxis auch zwangslaufig getan.~~ ein ea muss also problemwissen ausnutzen z. durch die wahl einer bestimmten mutationsstarke. werden also zwei ea verglichen dann wird diese einschrankung impliziert. ~~der schemasatz von john h.~~ holland wird allgemein als erklarung des erfolgs von genetischen algorithmen gesehen. er besagt vereinfacht dass sich kurze bitmuster mit uberdurchschnittlicher fitness schnell in einer generation ausbreiten die durch einen genetischen algorithmus evolviert wird. so konnen aussagen uber den langfristigen erfolg eines genetischen algorithmus getroffen werden. mit der theorie der virtuellen alphabete zeigte david e. goldberg 1990 dass durch eine reprasentation mit reellen zahlen ein ea der klassische rekombinationsoperatoren z. uniformes oder

> n-punkt crossover nutzt bestimmte bereiche des suchraums nicht erreichen kann im gegensatz zu einer reprasentation mit binaren zahlen. ~~daraus ergibt sich dass ea mit reeller reprasentation arithmetische operatoren zur rekombination nutzen mussen z.~~ arithmetisches mittel. mit geeigneten operatoren sind reellwertige reprasentation entgegen der fruheren meinung effektiver als binare.

Približno 85% engleskih i njemačkih rečenica je točno označeno.

Alternativno rješenje ovog problema može se implementirati traženjem dane riječi u engleskom i njemačkom rječniku. Pristup s regularnim izrazima ima nekoliko prednosti:

- regularni izraz je niz od stotinjak znakova koji u potpunosti opisuju algoritam prepoznavanja dok engleski i njemački rječnici imaju nekoliko megabajta
- efikasna implementacija regularnih izraza brže prepoznaje riječ nego pretraživanje rječnika
- glagolska vremena, padeži i slični elementi jezika mijenjaju izgled riječi pa oblik u kojem je ona u tekstu možda neće postojati u rječniku
- regularni izraz bolje podnosi tipografske greške, kratice, sleng i slične transformacije izvornih riječi

Situacije u kojima je ova metoda primjenjiva su one u kojima je važna brza i fleksibilna procjena, a u kojima je točnost manje bitan faktor. Primjer takve situacije je detekcija jezika kojim korisnik piše u web pregledniku u svrhu odabira točnog rječnika za provjeru pravopisa. Implementacija je mala (u memorijskom smislu) i brza što je pogodno za izvođenje na web stranici, a ako se detektira pogrešan jezik korisnik može lako ispraviti grešku.

3.5.2 Prepoznavanje izmišljenih riječi

Jezici evoluiraju što znači da se nove riječi stalno dodaju u govor dok se neke druge prestaju koristiti. Ipak, nove riječi i dalje imaju zajedničke karakteristike s ostatkom rječnika. Provjerena je efikasnost izraza u prepoznavanju takvih riječi. U tu svrhu korišten je generator[5] da se generira 100 nepostojećih engleskih i njemačkih riječi. Metoda generiranja odabire sljedove znakova koji se često pojavljuju u danom jeziku i provjerava da slučajno generirana riječ već ne postoji u rječniku. Regularni izraz točno prepoznaje englesku riječ u 87% slučajeva, a njemačku u 63%. Kao i prije, pogrešno označene riječi su precrtane.

> achicad adjugagon adkerizoodoclis afshoid aphabinparactuaminchama apsenwom bariallogism ~~bitylge~~ braenos brochfus catore chiempla comaly cous crecubflue deitenomal dewomillabiblepdrag discrauran dism distor distroma

domast duce ~~enappemish~~ endiald ~~entite~~ ettaversimpharm euriphlogenilin ~~fer~~ fessy flatoism ~~fless~~ flos forepliwit fruclotian glyposeute granae hecron hobtlum hopaymphagul ~~hurele~~ ~~intizabboa~~ jappleywhic jic knear lebacraut lism ~~magent~~ maphood merochersal minutrowle mokie ~~nessnater~~ nortate nost noxymosce oborid octinart paginandocon pergloctive poless preconte propyke psilly ~~ptrand-enamen~~ pyershlic refick rhoracerlide salgism saxybdo scition seckedan skis spirmago starga subiscodead suctivous supark supenoft ~~supioze~~ supurea thearbed thiandecuppod thonoscula tonan trinnae unaggatah unashilophia und unidae unjecty unmutknostionoiddly unsfa unston unt untly ~~ureh~~ ~~ursective~~ vaphospenodoar warsart

~~abellag~~ ~~abge~~ ~~abgeltnate~~ anbankbelzlor ~~aufvolin~~ aupter ausbrechter ause ausee baefuehr benbraufrien ~~bliche~~ ~~boehken~~ braetetitz bungen ~~dbatt~~ ~~dereigerichs~~ dieriver dirspien duen dungent durcher eigete einwelaeuden eitte ent erandsultes erlblit ers exen fer ferkinnstmen ~~finapirdachoehs~~ ~~fischlos-ge~~ ~~flazohrenruet~~ ~~folicheig~~ ~~fraem~~ frene fuechen fuendes gemter ~~giph-nerant~~ glielnicken ~~graechfort~~ heiereischwe hem ~~hoen~~ ~~hoten~~ ichten klervinne ~~koergen~~ lebstude lichrde masslagier ~~mitaulaetio~~ ~~mobuchen~~ ~~moniger~~ ~~muebetuebe~~ ~~neber~~ offsauer ossiespier ~~pare~~ pfelleraezit pragszwoh ~~prausbrin~~ ~~prospoegen~~ ~~pulaers~~ rederes satischen ~~skergeglogreckte~~ stertestig stes ~~tadisethation~~ ~~tart~~ ~~toputogigelte~~ traehlamme tritess ~~uehnaches~~ ~~umares~~ ~~umdermort~~ unarkels unsunenen ver veren verinst verleitten vermen verstem vert ~~vorsynahe~~ wellsiminke wenges wildereit ~~wingion~~ wistette ~~zoge~~ zudeubeon zugwuckernknut zursamt zweichhoet

Izraz dobro prepoznaje i izmišljene engleske riječi, a nešto gore njemačke. Očita je prednost nad pretraživanjem rječnika koje u ovom slučaju ne možemo iskoristiti.

Bibliografija

[1] *350000 simple english words*, https://github.com/dwyl/english-words, posjećeno veljača 2018.

[2] *Genetic algorithm - Wikipedia*, https://en.wikipedia.org/wiki/Genetic_algorithm, posjećeno veljača 2018.

[3] *German words*, https://raw.githubusercontent.com/Haspaker/anagram.tips/master/words/de/dict/german.wordlist.txt, posjećeno veljača 2018.

[4] *RabbitMQ*, https://www.rabbitmq.com/, posjećeno veljača 2018.

[5] *Random name/word generator*, https://foxular.net/namegen/run/, posjećeno veljača 2018.

[6] Alfred V. Aho i Jeffrey D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972, ISBN 0-13-914556-7.

[7] Eric Conrad, *Genetic Regular Expressions: a New Way to Detect and Block Spam*, (2008), https://www.sans.edu/student-files/presentations/genetic_reg_ex_withnotes.pdf.

[8] Melanie Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA, USA, 1998, ISBN 0262631857.

Sažetak

U ovom radu prezentirana je primjena genetskog programiranja, tehnike koja koristi genetski algoritam za generiranje računalnog koda. Definirani su osnovni pojmovi i principi te je dana implementacija u programskom jeziku Haskell. Demonstriran je način na koji se postiže specijacija, odnosno stvaranje novih vrsta, tako da se konstruira stablo u kojem svaki čvor predstavlja novu vrstu koja je evoluirana odvojeno od ostalih. Zatim se te nezavisne vrste križaju pa njihov produkt nastavlja evoluirati. Takva strategija je podložna paralelizaciji te je opisana distribuirana implementacija koja se može izvršavati na više računala koristeći RabbitMQ[4] tehnologiju.

Definiran je jezik regularnih izraza kojima se opisuju uzorci u tekstu. Dana je njihova implementacija u Haskellu te je opisano kako se na sintaksnim stablima regularnih izraza provode križanja i mutiranje. Zatim je postavljen problem prepoznavanja pripada li neka riječ engleskom ili njemačkom jeziku. Na osnovu 300 primjera jednog i drugog jezika genetski algoritam evoluira izraz koji prihvaća najveći broj engleskih riječi i što manji broj njemačkih. Konačni produkt postiže 75% točnost na primjerima za trening i 71% točnost na sveukupnom korpusu od 400000 riječi[1][3]. Korisnost tako generiranog izraza demonstrirana je na prepoznavanju jezika u cijelim rečenicama gdje je postignuta približno 85% točnost, te na prepoznavanju izmišljenih riječi.

Summary

In this work we presented an application of genetic programming, a technique which uses genetics algorithms to generate computer code. We defined basic terms and principles and provided an implementation in the programming language Haskell. We demonstrated a way to achieve speciation, the creation of new species, by constructing a tree in which each nore represents a new species evolved separate form others. Then those independent species are crossed and ther product continues evolving. This strategy allows for paralelization so we describe a distributed implementation which can be ran on multiple computers via the RabbitMQ[4] technology.

We define the language of regular expressions which are used to describe patterns in text. An implementation is given in Haskell and it is described how crossing and mutation is done on regular expression syntax trees. Then we pose the problem of classifying words as either English or German. Given 300 examples of each language the algorithm evolves an expression that accepts as many English words as possible while rejecting as many German. The final product achieves the accuracy of 75% on the training sample and 71% on the whole corpus of 400000 words[1][3]. The utility of such an expression is demonstrated in recognizing the language of whole sentences where we achieve approximately 85% accuracy, and on recognizing made-up words.

Životopis

Luka Horvat, rođen 2. listopada 1993. godine u Koprivnici, živio je u Ludbregu gdje je pohađao osnovnu školu. Nakon osnovne, upisao je srednju školu te završava smjer medijski tehničar u Elektrostrojarskoj školi, Varaždin. 2012. godine upisao je preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu i završio ga 2015. godine. Nakon toga je upisao diplomski studij Računarstvo i matematika.

U osnovnoj i srednjoj školi sudjelovao je na natjecanjima iz matematike i informatike. Tokom studija radio je u tvrtki Logon na razvoju računovodstvenog softvera. Nakon toga, 2015. godine se zaposlio u tvrtki SPAN gdje je razvijao sustav za vizualizaciju kretanja ugroženih vrsta ptica. Pola godine kasnije, početkom 2016. godine zaposlio se u tvrtki Deegeetal gdje je razvijao sustav za automatizirano prikupljanje 'leadova' te je radio na raznim web projektima. Dvije godine kasnije, krajem 2017. godine, zaposlio se u tvrtki Strongly Typed gdje trenutno radi na razvoju web aplikacija.