Zbornik seminarjev iz hevristik

Izbrana poglavja iz optimizacijskih metod (2010-11)

2. marec 2012

Ljubljana, 2011

Zbornik seminarskih nalog sta po knjigi [3] izbrala in uredila R. Škrekovski (FMF) in Vida Vukašinovič (IJS)

naslov: Zbornik seminarjev iz hevristik

avtorske pravice: dr. Riste Škrekovski in Vida Vukašinovič

izdaja: prva izdaja

Založnik: dr. Riste Škrekovski, Jadranska 21, Fakulteta za matematiko in fiziko, Univerza v Ljubljani, samozaložba

avtor: Riste Škrekovski, Vida Vukašinovič

leto izida: 2012 (v Ljubljani)

natis: elektronsko gradivo
http://www.fmf.uni-lj.si/ skreko/Gradiva/Zbornik_Hevristike.pdf

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

004.023(082)(0.034.2)
004.832.2(082)(0.034.2)
303.733.4(082)(0.034.2)

ŠKREKOVSKI, Riste
Zbornik seminarjev iz hevristik [Elektronski vir]: izbrana poglavja iz optimizacijskih metod (2010-11) / Riste Škrekovski, Vida Vukašinovič. - El. knjiga. - Ljubljana : samozal. R. Škrekovski, 2012

Način dostopa (URL): http://www.fmf.uni-lj.si/ skreko/Gradiva/Zbornik_Hevristike.pdf

ISBN 978-961-276-377-0

1. Vukašinovič, Vida
260626432

Kazalo

1 GRASP **9**

1.1 Problem trgovskega potnika . . . 9
1.2 GRASP . . . 10
1.3 Algoritem . . . 11
1.4 Psevdokoda . . . 12
1.5 Zaključek . . . 14

2 Iskanje po variabilnih okolicah **15**

2.1 Spust po variabilni okolici . . . 16
2.2 Posplošitev iskanja po variabilnih okolicah . . . 17

3 Iskanje s Tabuji **21**

3.1 Iskanje s tabuji . . . 21
3.2 Kratkoročni spomin . . . 22
3.3 Srednjeročni spomin . . . 25
3.4 Dolgoročni spomin . . . 25
3.5 Zaključek . . . 27

4 Simulirano ohlajanje **29**

4.1 Uvod v simulirano ohlajanje . . . 29
4.2 Parametri algoritma . . . 30
4.2.1 Začetni približek rešitve . . . 31
4.2.2 Začetna temperatura . . . 31
4.2.3 Shema ohlajanja . . . 31
4.2.4 Ravnovesni pogoj . . . 32
4.2.5 Zaustavitveni pogoj . . . 32
4.3 Primer: problem trgovskega potnika . . . 32
4.4 Konvergenca algoritma . . . 33
4.5 Podobni algoritmi . . . 35

5 Genetsko programiranje **37**
5.1 Pregled genetskega programiranja . . . 37
5.1.1 Predstavitev programov . . . 39
5.1.2 Inicializacija začetne populacije . . . 39
5.1.3 Selekcija . . . 40
5.1.4 Križanje in mutacija . . . 40
5.2 Podrobnosti . . . 41
5.2.1 Jezik . . . 41
5.2.2 Funkcija uspešnosti . . . 42
5.2.3 Parametri . . . 42
5.3 Primer uporabe . . . 42
5.4 Implementacija . . . 43

6 Optimizacija s kolonijami mravelj **47**
6.1 Uvod . . . 47
6.2 Algoritem . . . 48
6.2.1 Grajenje rešitev . . . 49
6.2.2 Posodobitev feromonov . . . 50
6.3 Razredi . . . 51
6.4 Konvergenca . . . 54
6.5 Problem trgovskega potnika . . . 55
6.5.1 Reševanje TSP s pomočjo ACO . . . 56
6.6 Implementacija . . . 57
6.7 Posplošitev . . . 58
6.8 Zaključek . . . 58

7 Optimizacija z roji **59**
7.1 Uvod . . . 59
7.2 Okolica delcev . . . 60
7.3 PSO za diskretne probleme . . . 66

8 Razpršeno preiskovanje **69**
8.1 Uvod . . . 69
8.2 Model in sestava algoritma . . . 70
8.2.1 Primer: SS algoritem za problem p-mediane. . . . 71
8.3 Povezovanje poti (Path Relinking) . . . 73
8.3.1 Primer PR: Optimalno povezovanje v binarnem iskalnem prostoru. . . 76

9 Umetni Imunski Sistemi **77**
9.1 Naravni imunski sistem . . . 77
9.2 Umetni imunski sistemi . . . 80
9.2.1 Populacijski AIS algoritmi . . . 80
9.2.2 Omrežni AIS algoritmi . . . 83
9.2.3 Primer: Preprečevanje neželjene pošte . . . 84

10 Kolonije Čebel **87**
10.1 Kolonije čebel . . . 87
10.2 Splošno o čebelah . . . 88
10.3 Iskanje novega gnezda . . . 88
10.4 Iskanje hrane . . . 89
10.5 Svatbeni let . . . 94
10.6 Zaključek . . . 96

11 PAES **97**
11.1 Uvod . . . 97
11.1.1 Večkriterijska optimizacija . . . 97
11.1.2 Evolucijski algoritmi . . . 98
11.2 Usmerjanje prometa v komutiranih omrežjih . . . 99
11.2.1 Opredelitev problema . . . 99
11.3 PAES . . . 100
11.3.1 Pregled . . . 100
11.3.2 Delovanje algoritma . . . 101
11.3.3 Prihodnji razvoj . . . 102
11.4 Zaključek . . . 102

12 Evolucijski algoritem s pareto močjo **105**
12.1 Evolucijski algoritem s pareto močjo . . . 105
12.2 Funkcija uspešnosti . . . 106
12.3 Zmanjševanje Pareto množice z grupiranjem . . . 108
12.4 Implementacija algoritma . . . 109

13 Niched Pareto Genetski algoritem za večkriterijsko optimizacijo **113**
13.1 Uvod . . . 113
13.2 Dosedanje delo . . . 113
13.3 Niched Pareto GA . . . 114
13.3.1 Pareto dominirajoči turnirji . . . 114
13.3.2 Deljenje na nedominirajoči fronti . . . 115
13.4 Aplikacija za tri probleme . . . 116

13.4.1 Preprosta testna funkcija 116
13.4.2 Drugi problem: Schafferjev F_2 118
13.4.3 Odprti problemi v hidrosistemih 120
13.5 Diskusija 121

Uvod

Metahevristični algoritmi spadajo med nenatančne algoritme, ki temeljijo na različnih hevrističnih pristopih. Tu žrtvujemo zagotovilo, da bomo našli optimalno rešitev v zameno za relativno dobro rešitev, ki jo najdemo v mnogo krajšem času. Cilj metahevristik je učinkovito preiskovanje iskalnega prostora v želji po (skoraj) optimalni rešitvi. Tehnike, ki jih uporabljajo segajo od preprostih postopkov lokalnega iskanja do komplesnih učnih procesov. Navadno so to verjetnostni algoritmi, kar pomeni, da za usmerjanje iskanja uporabljajo verjetnostno pravilo.

Seminarske so nastale v okviru predmeta Izbrana poglavja iz optimizacijskih metod (IPOM) leta 2011-12 na Fakulteti za matematiko in fiziko. Zahvaljujemo se študentom Aleš Bizjak, Katja Ciglar, Gregor Cigüt, Julia Cafnik, Branka Janežič, Jernej Zupančič, Jelena Marčuk, Lea Letnar, Jaka Kranjc, Leon Lampret, Gregor Sušelj, Jerica Valjavec, Petra Janžekovič.

Poglavje 1

GRASP

LEON LAMPRET

GRASP (”greedy randomized adaptive search procedure”) je družina algoritmov, ki aproksimativno rešuje optimizacijske probleme, ki so časovno prezahtevni za eksaktno reševanje.

V seminarski nalogi je opisana njena konkretna uporaba za problem trgovskega potnika. Dodana je tudi datoteka z dejanskim algoritmom v programskem jeziku Java. Prav tako je dodana datoteka v Mathematici 7, v kateri sta Javin algoritem iz te seminarske in v Mathematico 7 vgrajen algoritem preizkušena na naključno uteženih polnih grafih s poljubno izbranim številom vozlišč.

GRASP metahevristiko sta leta 1989 odkrila T.A. Feo in M.G.C. Resende ([3, p. 164 & 553], [35, p. 2]).

1.1 Problem trgovskega potnika

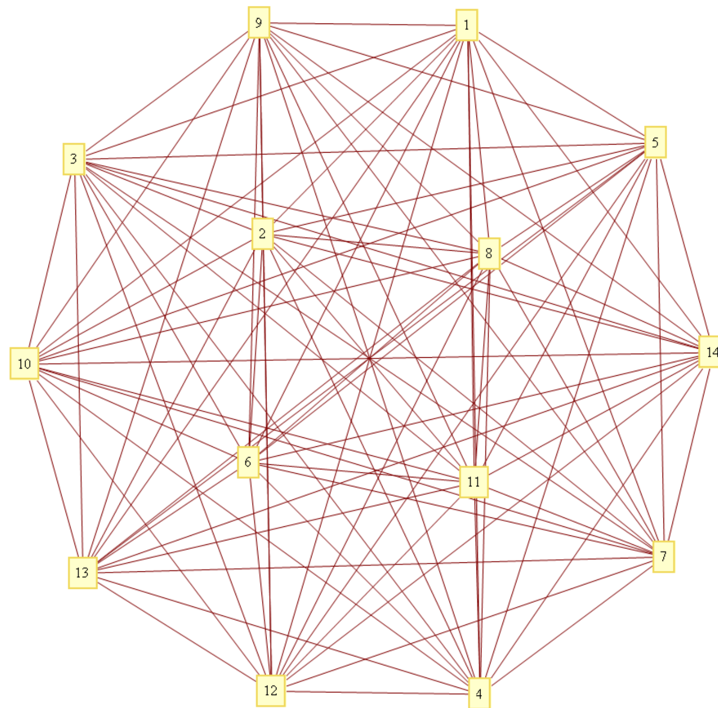
Problem trgovskega potnika (angl. ”travelling salesman problem” oz. okrajšano TSP) se v vsakdanjem jeziku glasi: danih je n mest in razdalja med poljubnim parom mest (od mesta do mesta lahko potujemo po zgolj eni poti). Najdi najkrajšo pot, ki se začne in konča v istem mestu ter obišče vsako mesto natanko enkrat!

TSP se v matematičnem jeziku teorije grafov glasi: v (neusmerjenem enostavnem) polnem grafu K_n z uteženimi povezavami (pozitivne vrednosti) najdi najkrajši cikel, ki vsebuje vsa vozlišča! Ciklom, ki vsebujejo vsa vozlišča grafa, pravimo *Hamiltonovi cikli*.

Če so vozlišča (mesta) označena s števili $1, \ldots, n$, lahko vsak Hamiltonov cikel v K_n predstavimo z zaporedjem $(1, *, \ldots, *)$ števil $\{2, \ldots, n\}$. Pri tem vsako zaporedje oblike $(1, v_2, v_3, \ldots, v_{n-1}, v_n)$ in $(1, v_n, v_{n-1}, \ldots, v_2, v_1)$ predstavljata isti cikel. Zato je število vseh možnih Hamiltonovih ciklov v K_n enako

$$h_n = \frac{(n-1)!}{2}.$$

Ko $n \in \mathbb{N}$ raste, postaja h_n neobvladljivo velik, kar pomeni, da eksaktne metode iskanja optimalnega cikla niso primerne.



Da bralec dobi občutek o računski kompleksnosti problema trgovskega potnika, omenimo dva zgleda. Graf na zgornji sliki vsebuje 14 vozlišč, 91 povezav in preko 3 miljarde Hamiltonovih ciklov. Kot drugi zgled omenimo, da če bi želeli obiskati prestolnice vseh 203 držav na svetu (vsako natanko enkrat), bi to lahko storili na več kot 10^{379} načinov (toliko je torej vseh možnih Hamiltonovih ciklov na polnem grafu vseh svetovnih prestolnic). Za primerjavo, število vseh atomov našega planeta je približno $9 \cdot 10^{49}$, število vseh zvezd v vidnem vesolju (groba ocena) pa 10^{22}.

1.2 GRASP

Metahevristika je algoritemski način reševanja kombinatoričnega optimizacijskega problema, pri katerem na začetku izberemo množico kandidatov za rešitev, in jo iterativno izboljšujemo (glede na neko vnaprej izbrano funkcijo zaželenosti), ter po dovolj korakih vrnemo najboljši element iz te množice. Metahevristike torej vrnejo približne rešitve, a veliko hitreje kot eksaktni postopki.

GRASP je metahevristika, ki sestoji iz dveh faz; tj. iz požrešne slučajne konstrukcije in lokalnega preiskovanja.

V prvi fazi na pameten način (odvisno od problema) izberemo izmed vseh možnih rešitev CL (seznam kandidatov) množico začetnih približkov RCL (omejen seznam kandidatov). To storimo deloma deterministično in deloma stohastično (vpeljemo naključne odločitve), da zagotovimo, da so začetni približki obetavni, a dovolj razpršeni po celotni množici CL, da bo druga faza pregledala čimvečji del vseh rešitev CL.

V drugi fazi za vsako izmed teh rešitev $s \in$ RCL pogledamo elemente $s' \in$ CL v njeni okolici (kaj je okolica je od problema in načina reševanja odvisno), ter v primeru da najdemo boljšo rešitev s', jo dodamo v RCL ter s odstranimo. To ponavljamo dokler zaustavitveni pogoj ni izpolnjen. Ta je lahko število iteracij, zahtevana natančnost, itd.

Psevdokoda za GRASP metahevristiko v vsej splošnosti se glasi ([3, p.164&165, 2.17&2.18], [34, p.2&3, Fig. 1&2&3], [35, p.2], [36, p.277, Fig. 5]):

Vhod: $\mathcal{O}$; objekt, ki ga preučujemo, npr. graf, matroid, itd.

Algoritem:
`ugotovi, kako izgleda množica vseh možnih rešitev CL` $\subseteq 2^{\mathcal{O}}$;
ta množica ni eksplicitno podana v programu, ker je prevelika
`eksplicitno določi, kako funkcija zaželjenosti` f`:CL` $\to \mathbb{R}$ `slika`
`eksplicitno določi množico začetnih približkov RCL` $\subseteq$ `CL`
`eksplicitno določi kaj pomeni okolica elementa s` $\in$ `RCL v množici CL`
ponavljaj dokler ni zadoščeno zaustavitvenemu pogoju (npr. predpisano število iteracij)**:**
  `naključno izberi s` $\in$ `RCL`
  `preglej vse elemente s'` $\in$ `CL v okolici s`
  `če najdeš element s', ki je bolj zaželjen, tj.` $f($`s'`$) > f($`s`$)$,
  `ga zamenjaj z s, tj. RCL:=(RCL-{s})`$\cup$`{s'}`

Vrni: `najboljši element (glede na` f`) v množici RCL.`

1.3 Algoritem

V tem poglavju je opisana konkretna uporaba GRASP metahevristike za TSP problem. Oznake so vzete iz drugega poglavja, kjer je postopek opisan v splošnem okolju.

Vhodni podatki: g, $iter$, k. Tu je g graf z vozlišči $\{1, \ldots, n\}$. Podan je v obliki incidenčne matrike, torej je $g_{i,j}$ vrednost povezave med vozliščema i in j, tj. razdalja med mestoma i in j. Podatek $iter$ označuje število dovoljenih iteracij algoritma, k pa število začetnih približkov, torej velikost množice RCL.

Množica vseh možnih rešitev CL je v našem primeru množica vseh možnih Hamiltonovih poti v g. Za te se odločimo, da jih bomo predstavili kot zaporedja števil $(l, 1, v_2, \ldots, v_n)$, $v_i \in \{2, \ldots, n\}$, kjer je l dolžina cikla $(1, v_2, \ldots, v_n)$ v grafu g. Torej sproti hranimo vrednost (dolžino) cikla, da ni treba vsakič znova računati, in tudi, da lahko cikli postanejo med seboj

primerljiva podatkovna struktura. Seveda med algoritmom množica CL nikoli ni zapisana v računalniku, saj je prevelika.

Algoritem:
Požrešna slučajna konstrukcija: Vsak začetni približek $t=(l,1,v_2,\ldots,v_n)\in$RCL ($t$..."tour") konstruiramo tako, da določimo $v_1:=1$; nato iterativno ($i=2,\ldots,n$) za v_i izberemo naključno izmed $\lfloor\frac{n}{5}\rfloor$ najbližjih vozlišč do v_{i-1}, ki še niso v t. Ko je $(1,v_2,\ldots,v_n)$ izbran, določimo še l, kot dolžino tega cikla. Take cikle t konstruiramo toliko časa, da RCL napolnimo (dokler $|\text{RCL}|<k$). Ker moramo elemente v RCL znati primerjati po dolžini, mora biti RCL urejena podatkovna struktura (v našem primeru se odločimo za **sorted set** v Javi).
Lokalno preiskovanje: Elementi v RCL so urejeni po dolžini (primerjamo jih po 0-ti koordinati l). Naključno izberemo $t\in$ RCL, toda z linearno padajočo verjetnostjo: najverjetneje izberemo cikel na vrhu RCL (najkrajši), malo manj verjetneje drugega (drugi najkrajši), še manj verjetneje tretjega, itd. Okolico cikla t definiramo kot množico vseh ciklov $t'\in$CL, ki jih dobimo iz t, tako da mu zamenjamo 2 vozlišči: $v_i\leftrightarrow v_j$. V algoritmu torej izberemo t' tako, da naključno zamenjamo dve vozlišči cikla t. Nato preverimo, če smo s tem dobili krajši cikel. V primeru da ja, t odstranimo iz RCL, ter t' dodamo v RCL. Ta postopek ponavljamo tolikokrat, kolikor je predpisano število ponovitev ($iter$).

Izhodni podatek: Na koncu vrnemo zgornji (najkrajši) element v RCL (tj. približek najkrajšega Hamiltonovega cikla v g).

1.4 Psevdokoda

V zadnjem poglavju je opisana psevdokoda za implementacijo GRASP za problem TSP. Psevdokoda je dobljena iz dejanske kode v Javi, toda poenostavljena, tako da so vsi nebistveni tehnični detajli izpuščeni, preostale pa so glavne ideje (ogrodje).

Vhod: `int[][] g, int iter, int k` g je celoštevilska simetrična matrika
Algoritem:

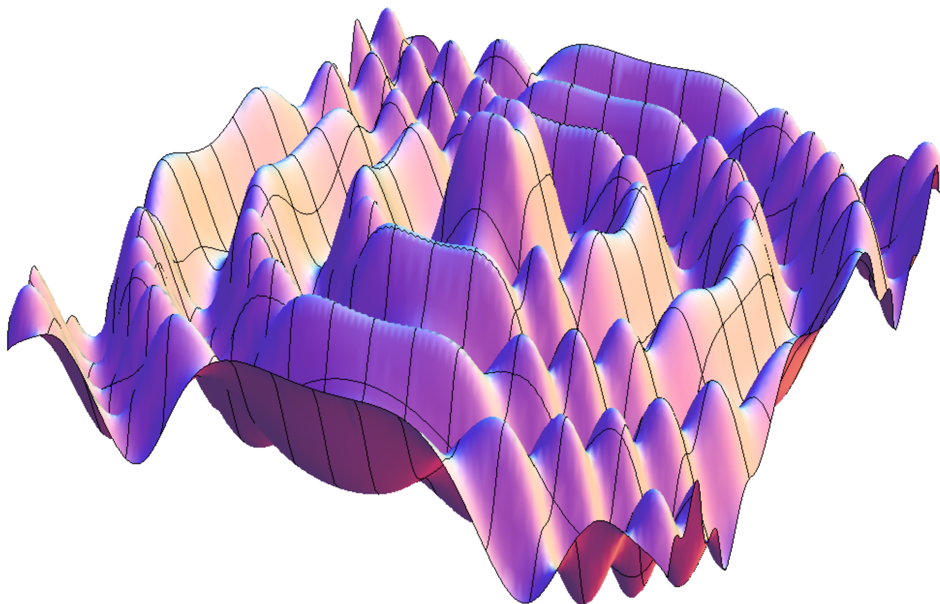
```
int n:=g.length;   število vozlišč (mest)
SortedSet<int[]> RCL:=new TreeSet<int[]>;   množica začetnih približkov

while(0<k)   'GREEDY RANDOMIZED CONSTRUCTION' (RCL polnimo)
  {int[] t:=new int[n+1];   nov cikel t[0],...,t[n], ki ga bomo dodali RCL
   t[1]:=1;   konstruiramo cikel t
   Set<int> vv:=new TreeSet<int>();   vozlišča, ki jih bomo dodajali v t;
   vrstni red dodajanja določa cikel t; sproti vv praznimo
```

```
    for (int i=2; i<=n; i++) vv.add(i);   prvo vv napolnimo
    for (int i=2; i<=n; i++)   z i se sprehajamo po t in ga polnimo; vv praznimo
      {int v=t[i-1];   prejšnje vozlišče
       SortedSet<int[]> koraki:=new TreeSet<int[]>;   urejena tabela,
       ki bo vsebovala vse sosede od v, in njihovovo oddaljenost, po katerih jih primerja
       for (j=vv.iterator(); j.hasNext(); )
         {int tmp=j.next();    tmp=vozlišče, ki je sosed od v
          koraki.add({g[v][tmp],tmp});} dodamo vozlišče in njegovo oddalje-
nost
       int tmp=random(1,min(max(n/5,10),koraki.size()));
       tmp=naključno celo število med 1 in n/5, razen ko je n majhen
       int[] korak={};   to bo tmp-ti korak v tabeli koraki
       for(j=koraki.iterator(); j.hasNext()&&0<tmp; tmp--)
          {korak=j.next();}   med n/5 najboljšimi koraki iz v izberemo naključno
       t[i]=korak[1];
       vv.remove(t[i]);
      }
    t[0]:=dolzina(t);
    RCL.add(t);   dodajamo element v urejeno drevo, torej O(log(k)) operacij
    k--;
   }
k:=RCL.size();
while(0<iter)   'LOCAL SEARCH' (RCL izboljšujemo)
  {int tmp:=getLinearRandomNumber(k);   naključno število med 1 in k
    int[] t:=new int[];
    for (i=RCL.iterator(); i.hasNext()&&0<tmp; tmp--) {t=i.next();}
    v RCL naključno izberemo cikel t; izbor krajšega cikla je bolj verjeten
    RCL.remove(t);
    int i0=random(1,n);   t[i0] in t[j0] bomo zamenjali znotraj t
    int j0=random(1,n);
    int e1 = g[t[i0-1]][t[i0]] + g[t[i0]][t[i0+1]];
    int e2 = g[t[j0-1]][t[j0]] + g[t[j0]][t[j0+1]];
    int e11= g[t[i0-1]][t[j0]] + g[t[j0]][t[i0+1]];
    int e22= g[t[j0-1]][t[i0]] + g[t[i0]][t[j0+1]];
    if (e11+e22<e1+e2)   če se dolžina cikla zmanjša, zamenjamo vozlišči
      {tmp=t[i0];
       t[i0]=t[j0];
       t[j0]=tmp;}
```

```
    RCL.add(t);
    iter--;
   }
```
Output: `RCL.first();`

OPOMBA: Demonstrirajmo z zgledom, do kakšnih težav pogosto pride pri uporabi zgornjega algoritma. Če si predstavljamo vse možne Hamiltonove cikle v našem grafu (torej množico CL) kot (diskretno) podmnožico ravnine $\mathbb{R}^2$, in definiramo funkcijo $f : CL \to \mathbb{R}$ kot preslikavo ki ciklu priredi minus njegovo dolžino, potem se naš optimizacijski problem glasi "najdi maksimum funkcije f". Množica RCL je podmnožica od CL in algoritem bo v vsaki iteraciji neko točko iz RCL zamenjal s kako v njeni neposredni okolici. Torej bodo točke iz RCL skonvergirale k njihovim **lokalnim maksimumom**, in tam ostale ujete. Pomembno je torej, da je množica RCL čimbolj razpršena, do problemov pa bo vseeno prihajalo vsakič, ko bo f imela veliko lokalnih maksimumov. Večja kot je RCL, večja bo verjetnost, da kak element pristane v bližini **globalnega maksimuma** in zato k njemu skonvergira.



1.5 Zaključek

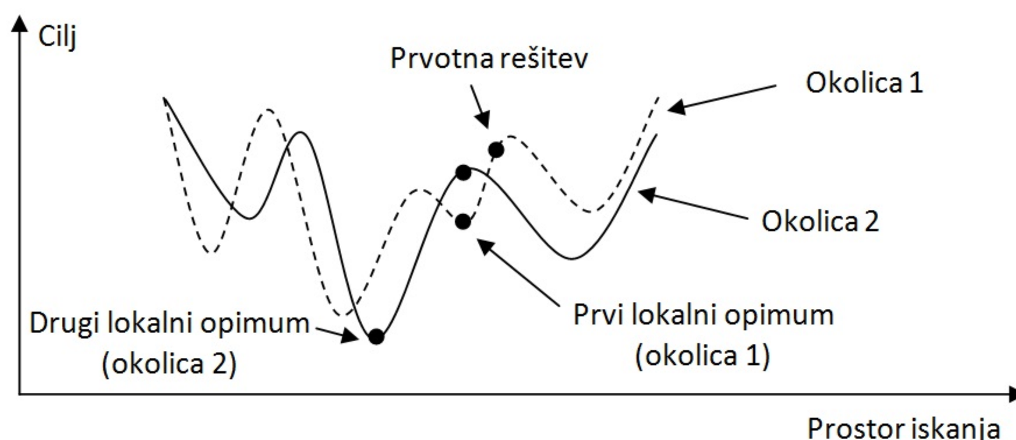
GRASP ima mnogo aplikacij, tako v akademskih krogih, kot tudi v industriji. Njegove prednosti pred drugimi algoritmi so, da ga lahko hitro implementiramo, saj sta požrešna konstrukcija in lokalno preiskovanje algoritma običajno že implementirana v okolje oz. program, s katerim delamo. Ponavadi je tudi nastavljanje parametrov minimalno. Ni pa primeren za probleme, pri katerih težje zgeneriramo množico začetnih rešitev, torej probleme, kjer je težko sploh najti kako/nekaj rešitev. [35, p.16].

Poglavje 2

Iskanje po variabilnih okolicah

KATJA CIGLAR

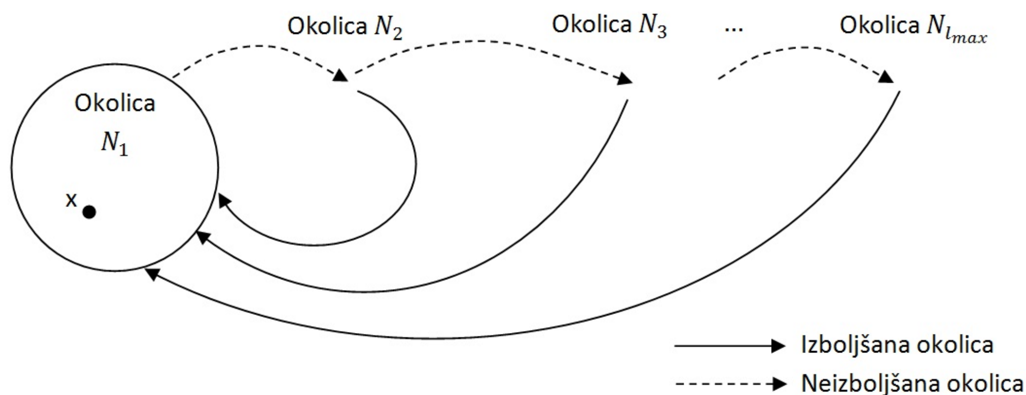
Osnovna ideja iskanja po variabilnih okolicah (ang. *variable neighborhood search*, v nadaljevanju VNS) je zaporedno raziskovanje množice v naprej določenih okolic s ciljem najti boljšo optimalno rešitev. Raziskovanje množice okolic lahko poteka slučajno ali pa sistematično, da pridemo do različnih lokalnih optimumov ter da zapustimo lokalni optimum. VNS izkorišča dejstvo, da uporaba različnih okolic pri lokalnem iskanju lahko generira različne lokalne optimume ter da je globalni optimum lokalni optimum za dano okolico (Slika 1).



SLIKA 1: Iskanje po variabilnih okolicah z uporabo dveh okolic. Prvi lokalni optimum dobimo glede na okolico 1. Glede na okolico 2 dobimo drugi lokalni optimum iz prvega lokalnega optimuma.

2.1 Spust po variabilni okolici

Algoritem VNS temelji na spustu po variabilni okolici (ang. *variable neighborhood descent*, v nadaljevanju VND), ki je deterministična verzija VNS. VND algoritem uporablja zaporedne okolice pri spustu v lokalni optimum. V prvem koraku definiramo množico okolic N_l ($l = 1, \ldots, l_{max}$). Začnemo v okolici N_1 in naj bo x prvotna rešitev. V primeru, da izboljšanje rešitve x v njeni trenutni okolici $N_l(x)$ ne obstaja, se okolica spremeni iz N_l v N_{l+1}. Če v postopku najdemo izboljšanje rešitve x, se okolica vrne v prvo N_1 in ponovno začnemo postopek (Slika 2). Strategija bo uspešna, če bodo različne okolice komplementarne, kar pomeni, da lokalni optimum za okolico N_i ne bo enak lokalnemu optimumu za okolico N_j. Algoritem VND je zapisan v nadaljevanju.



SLIKA 2: Princip delovanja algoritma spust po variabilni okolici.

ALGORITEM 1: Predloga algoritma spust po variabilni okolici.

Vnos: množica okolic N_l za $l = 1, \ldots, l_{max}$.

$x = x_0$; /*Generiramo prvotno rešitev*/

$l = 1$;

Dokler $l \leq l_{max}$ **Delaj**

Poišči najboljšega soseda x' od x v $N_l(x)$;

Če $f(x') \leq f(x)$ **Potem** $x = x'$; $l = 1$;

Sicer $l = l + 1$;

Izhod: Najboljša najdena rešitev.

Algoritem VND je povezan z izbiro okolic za namen njihove nadaljne uporabe. Pri tem je potrebno upoštevati kompleksnost okolic, saj večje kot so okolice, večja je časovna zahtevnost algoritma. Glede na vrstni red nadaljne uporabe, je najbolj popularna strategija naraščajoča

razvrstitev okolic glede na njihovo kompleksnost, to pomeni glede na njihovo moč $|N_l(x)|$.

2.2 Posplošitev iskanja po variabilnih okolicah

VNS je stohastični algoritem pri katerem najprej definiramo množico okolic N_k ($k = 1, \ldots, n$). Vsaka iteracija algoritma je sestavljena iz treh korakov: pretresanje, lokalno iskanje in premik. Pri vsaki iteraciji je rešitev x' naključno generirana v trenutni okolici $N_k(x)$, kar imenujemo pretresanje. Postopek lokalnega iskanja apliciramo na rešitev x', da generiramo rešitev x''. Trenutno rešitev nadomestimo z novim lokalnim optimumom x'', če in samo če smo našli boljšo rešitev (to pomeni $f(x'') \leq f(x)$). Enak proces ponovno začnemo z rešitvijo x'' v prvi okolici N_1. V primeru, da ne najdemo boljše rešitve (to pomeni $f(x'') \geq f(x)$), se algoritem premakne v naslednjo okolico N_{k+1}, slučajno generira novo rešitev v tej okolici in jo poskuša izboljšati. Potrebno je poudariti, da se proces lahko zacikla (na primer $x = x''$). Algoritem VNS je zapisan v nadaljevanju.

ALGORITEM 2: Predloga osnovnega algoritma iskanje po variabilni okolici.

Vnos: množica okolic N_k za $k = 1, \ldots, k_{max}$.
  $x = x_0$; /*Generiramo prvotno rešitev*/
  Ponavljaj
    $k = 1$;
    Ponavljaj
      Pretresi: izberi slučajno rešitev x' iz k-te okolice $N_k(x)$ od x;
      $x'' =$ lokalno iskanje(x');
      Če $f(x'') \leq f(x)$ **Potem** $x = x''$;
        Nadaljuj iskanje z N_1; $k = 1$;
      Sicer $k = k + 1$;
    Dokler $k = k_{max}$;
  Dokler Kriterij končanja
Izhod: Najboljša najdena rešitev.

Bolj splošni algoritem VNS dobimo, če enostavno lokalno iskanje nadomestimo z algoritmom VND. Algoritem je zapisan v nadaljevanju.

ALGORITEM 3: Predloga splošnega algoritma iskanje po variabilni okolici.

Vnos: množica okolic N_k za $k = 1, \ldots, k_{max}$ za pretresanje;
  množica okolic N_l za $l = 1, \ldots, l_{max}$ za lokalno iskanje.

$x = x_0$; /*Generiramo prvotno rešitev*/
Ponavljaj
  Od $k = 1$ **Do** k_{max} **Delaj**;
  Pretresi: izberi slučajno rešitev x' iz k-te okolice $N_k(x)$ od x;
  Lokalno iskanje z algoritmom VND:
  Od $l = 1$ **Do** l_{max} **Delaj**;
    Poišči najboljšega soseda x'' od x' v $N_l(x')$;
    Če $f(x'') \leq f(x)$ **Potem** $x' = x''$, $l = 1$;
    Sicer $l = l + 1$;
    Če je lokalni optimum je boljši od x **Potem**
      $x = x''$;
      Nadaljuj iskanje z N_1 ($k = 1$);
    Sicer $k = k + 1$;
Dokler Kriterij končanja
Izhod: Najboljša najdena rešitev.

Namen uporane algoritma VNS je povezano z izbiro okolic v koraku pretresanja. Običajno vzamemo vgnezdene okolice, kjer vsaka naslednja okolica $N_k(x)$ vsebuje prejšno $N_{k-1}(x)$:

$$N_1(x) \subset N_2(x) \subset \ldots \subset N_k(x), \forall x \in S.$$

Potrebno je poiskati kompromis med intenzivnostjo iskanja in njegovo diverzifikacijo preko porazdelitve dela med fazama lokalnega iskanja in pretresanja. Več dela v fazi lokalnega iskanja (več intenzivnosti) nam generira boljši lokalni optimum, medtem ko več dela v fazi pretresanja vrne potencialno boljše regije prostora iskanja (več diverzifikacije).

Primer 1: Vgnezdene okolice za fazo pretresanja. Za več kombinatoričnih optimizacijskih problemov je definicija vgnezndenih okolic povsem naravna. Na primer, za usmerjevalni problem kot je problem potovalnega prodajnega agenta in problem usmerjevanja vozila, kjer k-ti optimalni operater lahko uporabimo za različne vrednosti k ($k = 2, 3, \ldots$).

Tako kot lokalno iskanje, tudi VNS potrebuje majhno število parametrov. V fazi pretresanja je edini parameter število okoličnih struktur k_{max}. V primeru velikih vrednosti k_{max} (zelo velike okolice), bo algoritem VNS podoben večkrat začetemu lokalnemu iskanju. V primeru majhnih vrednosti k_{max}, bo algoritem podoben navadnemu lokalnemu iskanju.

Primer 2: Uporaba algoritma na primeru iz resničnega življenja. Primer prikazuje uporabo več okolic ($k = 9$) za rešitev problema razporejanja. Veliko nahajališč nafte na kopnskih

področjih se zanaša na umetne metode dviganja. Vzdrževanje, ki vključuje čiščenje, stimuliranje in drugo, je ključnega pomena za delovanje nahajališča. Vzdrževanje opravljajo s pomočjo ploščadi. Število razpoložljivih ploščadi je omejeno glede na potrebe po vzdrževanju nahajališč. Odličitev katero od ploščadi bodo poslali za izvedbo vzdrževanja je odvisno od več faktorjev, kot so produktivnost nahajališča, trenutno nahajanje ploščadi glede na zahtevano nahajališče in vrsta potrebnega vzdrževanja. Problem razporeditve ploščadi je poiskati najboljšo razporeditev S^* za razpoložljivih m ploščadi, tako da minimiziramo izgubo proizvodnje nahajališč, ki čakajo na vzdrževanje. Razpored je opredeljen z urejeno množico vzdrževanja nahajališč s ploščadmi. Algoritem iskanja po variabilnih okolicah je bil predlagan za rešitev tega problema z uporabo naslednjih okolic v koraku pretresanja:

1. Zamenjava poti, kjer nahajališča povezana z dvema ploščadma zamenjamo. (SS)
2. Zamenjava nahajališč z isto ploščadjo, kjer zamenjamo vzdrževanje dveh nahajališč z isto ploščadjo. (SWSW)
3. Zamenjava nahajališč z različnima ploščadma, kjer dve nahajališči vzdrževani z dvema različnima ploščadma zamenjamo. (SWDW)
4. Dodaj/odvzemi (AD), kjer je nahajališče prerazporejeno na razpored druge ploščadi.
5. Dve uporabi transformacije SWSW.
6. Dve uporabi transformacije SWDW.
7. Tri uporabe transformacije SWDW.
8. Zaporedna uporaba dveh AD transformacij.
9. Zaporedna uporaba treh AD transformacij.

Predlaganih je bilo že več razširitev algoritma GVNS, toda te razširitve niso povezane z glavnim konceptom VNS algoritma, to je zamenjava okolice.

Poglavje 3

Iskanje s Tabuji

Jerica Valjavec

3.1 Iskanje s tabuji

Iskanje s tabuji je algoritem za reševanje optimizacijskih problemov. Je ena najbolj razširjenih metaheuristik za reševanje teh problemov. Posebnost iskanja s tabuji je, da si zapomni podatke o zgodovini postopka iskanja najboljše rešitve. Pri vsaki ponovitvi trenutno najboljšo rešitev zamenja z novo najboljšo, v tej ponovitvi najdeno rešitvijo.

Možne rešitve, ki so dosegljive iz trenutne rešitve imenujemo sosede te rešitve oz. soseščina te rešitve. Novo rešitev postopek torej išče v soseščini. Najboljša rešitev v soseščini je tako izbrana za novo rešitev. Lahko pa se zgodi, da najboljša sosednja rešitev ni boljša od trenutne, to pomeni, da smo našli lokalno najboljšo rešitev. V tem primeru postopek za novo rešitev vzame slabšo rešitev (ki jo izbere med sosedami), saj s tem zbeži stran od lokalnega optimuma (saj ta ni nujno globalni optimum).

Uporaba le teh pravil lahko privede do ciklov, to pomeni, da lahko že obiskano rešitev obiščemo še enkrat (rešitev se še enkrat pojavi kot najboljša sosednja rešitev). Da se izognemo ciklom, iskanje s tabuji loči že obiskane rešitve od še ne obiskanih rešitev. Torej si zapomni trenutno pot iskanja, za kar smo že na začetku rekli, da je posebnost tega algoritma.

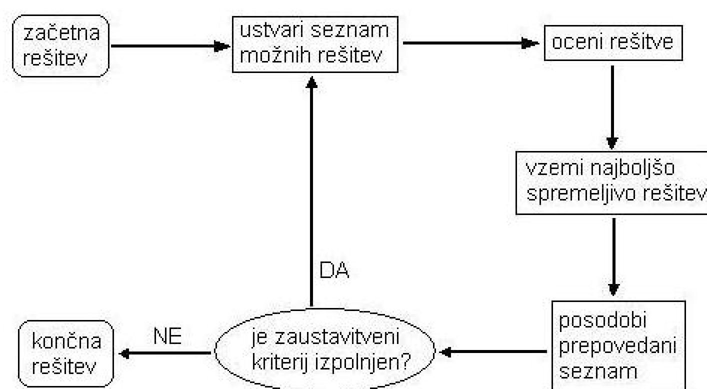
Že obiskane (nedavne) rešitve postopek hrani na posebnem seznamu, ki se imenuje *prepovedani seznam* (ali seznam tabujev). Namesto celotnih obiskanih rešitev pa seznam lahko hrani tudi le nekatere atribute obiskanih rešitev ali pa prehode (poteze) med obiskanimi rešitvami. Prepovedani seznam predstavlja kratkoročni spomin algoritma. Pri vsaki ponovitvi se prepovedani seznam, glede na novo rešitev, posodobi oz. spremeni.

Vendar pa je shranjevanje vseh že obiskanih rešitev časovno in prostorsko zahtevno, kajti pri vsaki ponovitvi moramo preveriti ali nova rešitev že pripada seznamu obiskanih rešitev. Zato je prepovedani seznam ponavadi sestavljen iz konstantnega števila prepovedanih rešitev, saj s

tem zmanjšamo omenjeni zahtevnosti. Ko na seznam dodamo novo rešitev, najstarejšo rešitev enostavno izbrišemo (odstranimo) iz seznama.

Lahko se zgodi, da neka poteza dobro obeta (generira rešitev, ki je boljša od trenutne rešitve), vendar pa je na prepovedanem seznamu. Po, do sedaj, opisanih pravilih, te poteze ne bi smeli uporabiti. Vendar jo pod določenimi pogoji, ki jih imenujemo *aspiracijski kriterij*, lahko uporabimo. Torej, potezo, ki je na prepovedanem seznamu lahko uporabimo, če zadošča aspiracijskemu kriteriju. Drugače povedano, dosegljive sosednje rešitve so tiste, ki niso na prepovedanem seznamu ali pa zadoščajo aspiracijskemu kriteriju.

Osnovna varianta algoritma iskanja s tabuji je sledeča. Vzamemo začetno rešitev, nato pa s pomočjo prepovedanega seznama in aspiracijskega kriterija določimo seznam novih možnih rešitev. Izmed teh izberemo najboljšo, posodobimo prepovedani seznam in nato nadaljujemo s postopkom. Ta opis je prikazan v spodnji skici.



Slika 3.1: Skica osnovne variante algoritma.

Kot smo spoznali, sta glavni značilnosti iskanja s tabuji, prepovedani seznam in aspiracijski kriterij. Uporabljata pa se tudi srednjeročni spomin, ki služi za usmerjeno preiskovanje in dolgoročni spomin, ki skrbi za razpršeno preiskovanje.

Srednjeročni spomin hrani elitne (npr. najboljše) rešitve, najdene med postopkom. Takim rešitvam bi radi dali prednost, običajno se to naredi z uteženimi verjetnostmi. Dolgoročni spomin pa hrani informacije o obiskanih rešitvah. Njegova naloga je, da spodbudi atribute elitnih rešitev, da raziščejo še druga (neraziskana) področja rešitev.

3.2 Kratkoročni spomin

Naloga kratkoročnega spomina je shranjevanje zgodovine postopka iskanja najboljše rešitve, s čimer preprečimo nastanek ciklov in s tem ciklanja postopka. Osnovna varianta kratkoročnega

Algorithm 1 Iskanje s tabuji

s = s_0; /začetna rešitev
prepovedani seznam;
srednjeročni spomin;
dolgoročni spomin;
while (zaustavitveni kriterij)
    poišči najboljšo dosegljivo sosednjo rešitev s';
    s = s';
    posodobi prepovedani seznam;
    posodobi srednjeročni spomin;
    posodobi dolgoročni spomin;
    if (pogoj za usmerjeno preiskovanje) usmerjeno preiskovanje;
    if (pogoj za razpršeno preiskovanje) razpršeno preiskovanje;

spomina hrani vse (do tedaj) obiskane rešitve tekom procesa. Ta varianta sicer prepreči cikle, vendar je malo uporabna, saj ustvari zapleteno hranjenje podatkov ter je časovno potratna. Npr. preverjanje ali so sosednje rešitve na prepovedanem seznamu ali ne, je časovno predrago.

Ena od izboljšav kratkoročnega spomina je, da mu omejimo dolžino prepovedanega seznama. Če je na prepovedanem seznamu shranjenih zadnjih k obiskanih rešitev, to pri iskanju s tabuji prepreči cikle dolžine največ k. Na ta način porabimo manj prostora ter izboljšamo časovno zahtevnost, vendar pa se cikli v postopku lahko še vedno pojavijo.

Najbolj pogost način predstavitve prepovedanega seznama je z atributi potez. Na prepovedanem seznamu hranimo prepovedane poteze ali pa njihove obrate. Npr. če rešitev s_j dobimo iz rešitve s_i s potezo m ($s_i \oplus m = s_j$), potem na prepovedani seznam shranimo potezo m ali pa njen obrat m^{-1} (m^{-1} je poteza za katero velja $(s_i \oplus m) \oplus m^{-1} = s_i$). Ta poteza je nato prepovedana za določeno število ponovitev postopka, to število pa imenujemo *prepovedani čas* poteze.

Kljub temu da prepovedani seznam vsebuje zadnjih k potez, se lahko pojavijo cikli dolžine manjše od k.

Primer prepovedanega seznama predstavljenega z atributi potez: Vzemimo razred permutacij. Recimo, da uporabljamo le transpozicije. Transpozicijo Π označimo z indeksoma (i, j), to je s tistima indeksoma, ki ju zamenjamo. Na prepovedani seznam pa shranimo obrat transpozicije (i, j), to pa je transpozicija (j, i). Ta transpozicija, ker je na prepovedanem seznamu, je prepovedana, saj bi z njeno uporabo dobili rešitev, ki smo jo že imeli. Prepovedani seznam bi lahko definirali tudi na strožji način, npr. tako, da bi nanj shranjevali indekse, ki nastopajo v transpozicijah. To pomeni, da bi v našem primeru na prepovedani seznam shranili indeksa i in j. S tem bi preprečili uporabo vseh transpozicij, ki vsebujejo vsaj enega od indeksov i in j.

V splošnem lahko uporabimo več prepovedanih seznamov. Nedavne rešitve ali poteze so lahko shranjene na večih seznamih hkrati. Potem je rešitev ali poteza dopustna, če ni na nobenem prepovedanem seznamu.

Primer uporabe več prepovedanih seznamov: Iščemo največjo neodvisno množico v grafu. Graf označimo z $G = (V, E)$, kjer n označuje število vozlišč ($|V| = n$). Iščemo torej množico X, ki je podmnožica vseh vozlišč ($X \subseteq V$), za katero velja, da nobena povezava nima obeh krajišč v X in je po moči največja taka množica. Iskanje s tabuji uporabimo za iskanje največje neodvisne množice velikosti k. Uporabimo funkcijo, ki minimizira število povezav, ki imajo obe krajišči v množici X. Sosednje rešitve dobimo z zamenjavo vozlišča $v \in X$ z vozliščem $w \in E \setminus X$. Število sosednjih rešitev dane rešitve je $k(n - k)$, kar je velikostnega reda $O(nk)$. Uporabimo tri različne prepovedane sezname:

- Prepovedani seznam P_1 hrani zadnje generirane rešitve.
- Prepovedani seznam P_2 hrani zadnja vozlišča, ki so bila dodana v X.
- Prepovedani seznam P_3 hrani zadnja vozlišča, ki so bila odstranjena iz X.

Dolžina prepovedanega seznama (število shranjenih rešitev ali potez) močno vpliva na potek algoritma. V vsaki ponovitvi na seznam dodamo dobljeno rešitev, najstarejšo rešitev (rešitev, ki je najdlje na prepovedanem seznamu) pa odstranimo iz seznama. Krajši kot je seznam, večja je verjetnost ponovitve rešitve oz. cikla. Daljši seznam pa nam da več omejitev (glede na to koliko potez je prepovedanih) in spodbudi razpršeno preiskovanje. Dolžino prepovedanega seznama določimo glede na strukturo pokrajine problema in njej sorodnih lastnosti.

Poznamo različne oblike prepovedanega seznama:

- *statična oblika*: Običajno se uporablja statična (konstantna) dolžina prepovedanega seznama. Dolžina (statična vrednost) je lahko odvisna od velikosti problema, še večkrat pa je odvisna od števila sosedov oz. od velikosti soseščine. Optimalna dolžina prepovedanega seznama za vse probleme ne obstaja, niti ne obstaja enolična dolžina za vse primere nekega problema. Optimalna dolžina se lahko skozi proces spreminja, zato lahko uporabimo dolžino seznama, ki se tekom postopka spreminja.
- *dinamična oblika*: Dolžina prepovedanega seznama se spreminja tekom postopka brez uporabe informacij o zgodovini postopka.
- *prilagodljiva oblika*: Dolžina prepovedanega seznama se spreminja glede na zgodovino postopka. Npr. dolžina se spremeni na podlagi obnašanja postopka v zadnjih nekaj ponovitvah.

Primer iskanja s tabuji, ki uporablja prilagodljivo obliko prepovedanega seznama je odzivno iskanje s tabuji. Tu se dolžina prepovedanega seznama poveča, če kratkoročni spomin kaže, da bo postopek obiskal že obiskano rešitev oz. da bo na poti iskanja nastal cikel. Zato kratkoročni spomin hrani obiskane rešitve in ne atributov rešitev ali potez.

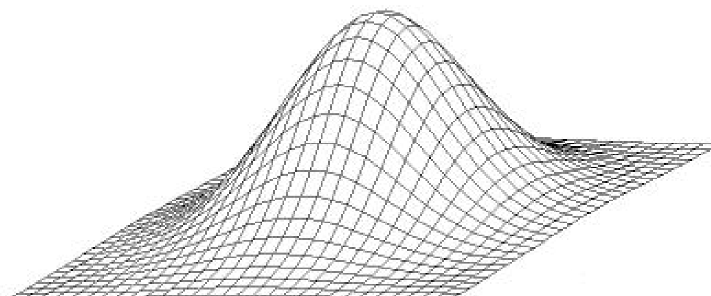
3.3 Srednjeročni spomin

Srednjeročni spomin skrbi za usmerjeno preiskovanje, hrani informacije najboljših (elitnih) najdenih rešitev z namenom, da usmeri postopek v obetavno smer (v obetavna področja rešitev). To si zagotovi s tem, da prepozna (pogoste) skupne lastnosti najboljših rešitev, nato pa postopek usmeri v iskanje med rešitvami, ki imajo prav take lastnosti. Popularen pristop kar ponovno zažene postopek z najboljšo najdeno rešitvijo ter fiksira tiste komponente rešitve, ki so najbolj obetajoče, te pa razbere iz elitnih rešitev.

Srednjeročni spomin se najpogosteje predstavi kot nedavni spomin. V ta namen morajo biti definirane komponente, povezane z rešitvami. Način definiranja teh komponent je od problema do problema različen. Nedavni spomin za vsako komponento beleži število uspešnih ponovitev postopka v katerih je ta komponenta prisotna v obiskanih rešitvah. Najpogosteje uporabljeni dogodek, ki sproži začetek usmerjenega preiskovanja je dana perioda, lahko pa začnemo tudi po določenem številu ponovitev brez napredka.

Usmerjeno preiskovanje v danem področju celotnega prostora preiskovanja, ni vedno uporabno. Učinkovitost usmerjenega preiskovanja je odvisna od pokrajinske strukture danega problema. Npr. če ima pokrajina veliko lukenj (vrzeli) in je iskanje s tabuji brez usmerjenega preiskovanja učinkovito v vsaki luknji, potem je uporabljati usmerjeno preiskovanje v vsaki luknji, neuporabno.

V nekaterih primerih, kot je npr. na sliki 3.2, je usmerjeno preiskovanje uporabno. Kjer koli začnemo, bo usmerjeno preiskovanje zelo hitro poiskalo edini maksimum.



Slika 3.2: Primer pokrajine, kjer usmerjeno preiskovanje hitro pripelje k rešitvi.

3.4 Dolgoročni spomin

Dolgoročni spomin pri iskanju s tabuji spodbuja razpršenost preiskovanja, to pomeni, da spodbuja postopek, da raziskuje še neraziskana področja rešitev. Dolgoročni spomin se najpogosteje predstavi kot frekvenčni (pogostnostni) spomin. Kot pri nedavnem spominu morajo tudi tu biti definirane komponente, povezane z rešitvami. Frekvenčni spomin si za vsako komponento

beleži število kolikokrat je komponenta prisotna v vseh že obiskanih rešitvah. Razpršeno preiskovanje se uporabi periodično ali pa po danem številu ponovitev postopka brez napredka.

Poznamo tri popularne metode (strategije) razpršenega periskovanja:

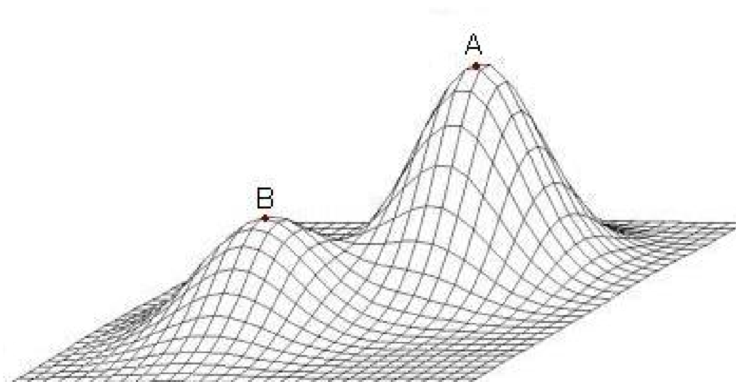
- *ponovni zagon razpršenega preiskovanja*: Ta strategija pogleda najmanjkrat obiskano komponento in vzame trenutno ali pa najboljšo rešitev s te komponente, nato pa postopek ponovno zažene na tej rešitvi.
- *nenehno razpršeno preiskovanje*: Ta strategija skrbi za nenehno nagnjenost k spodbujanju razpršenega periskovanja. Npr. nepristranska funkcija uporabi frekvenco komponent rešitev v oceni trenutnih rešitev. Pri tem so pogosto obiskane rešitve slabše ocenjene.
- *strateško nihanje*: Strateško nihanje dovoljuje uporabo vmesnih rešitev, t.j. rešitev, ki v resnici niso možne (sploh niso v množici možnih rešitev). Strategija postopek preiskovanja vodi preko teh ne možnih rešitev in nato pride nazaj po možno (obstoječo) rešitev.

Primer iskanja s tabuji za problem trgovskega potnika: Definirati moramo prepovedani seznam, srednjeročni in dolgoročni spomin ter aspiracijski kriterij.

- Prepovedani seznam sestavlja t vozlišč in prepreči, da bi bila ta vozlišča izbrana naslednjih l ponovitev postopka. Število l predstavlja prepovedani čas vozlišča, po tem času (po l ponovitvah postopka) je vozlišče odstranjeno iz prepovedanega seznama. Ali je vozlišče prepovadano ali ne, lahko predstavimo z matriko, katere elementi zavzamejo le dve vrednosti (npr. 'true' in 'false').
- Tako srednjeročni kot dolgoročni spomin sta sestavljena iz seznama t vozlišč, katera so izbrana iz zadnjih k najboljših (najslabših) rešitev. Postopek vzpodbuja (ali zavira) izbor teh vozlišč za rešitve na podlagi njihove pogostnosti pojavljanja v najboljših rešitvah in na podlagi kvalitete rešitev v katerih se pojavljajo.
- Aspiracijski kriterij je kar običajni aspiracijski kriterij, ki sprejme tiste prepovedane poteze, ki generirajo boljše rešitve (boljše od najboljših najdenih z uporabo prepovedanega seznama).

Tako kot usmerjeno preiskovanje tudi razpršeno preiskovanje ni vedno uporabno. To je odvisno od pokrajinske strukture danega problema. Npr. če so vse dobre rešitve skoncentrirane na enem mestu (in so tesno skupaj), potem razpršeno preiskovanje, ki vodi v druga področja (stran od najboljših rešitev), ni uporabno.

Na sliki 3 vidimo primer pokrajine, za katero je razpršeno preiskovanje lahko uporabno. Če postopek iskanja s tabuji začne bližje točki B, bo osnovna varianata algoritma pripeljala k rešitvi B, kar pa ni pravilna rešitev. Z uporabo razpršenega preiskovanja pa bo postopek šel pogledat še v druga področja in tako prišel do točke A, ki je rešitev.



Slika 3.3: Primer pokrajine, kjer je razpršeno preiskovanje uporabno, če gre postopek proti rešitvi B.

Čas, ki je dodeljen usmerjenemu in razpršenemu preiskovanju (kot del iskanja s tabuji), mora biti previdno razdeljen glede na lastnosti pokrajine danega problema.

3.5 Zaključek

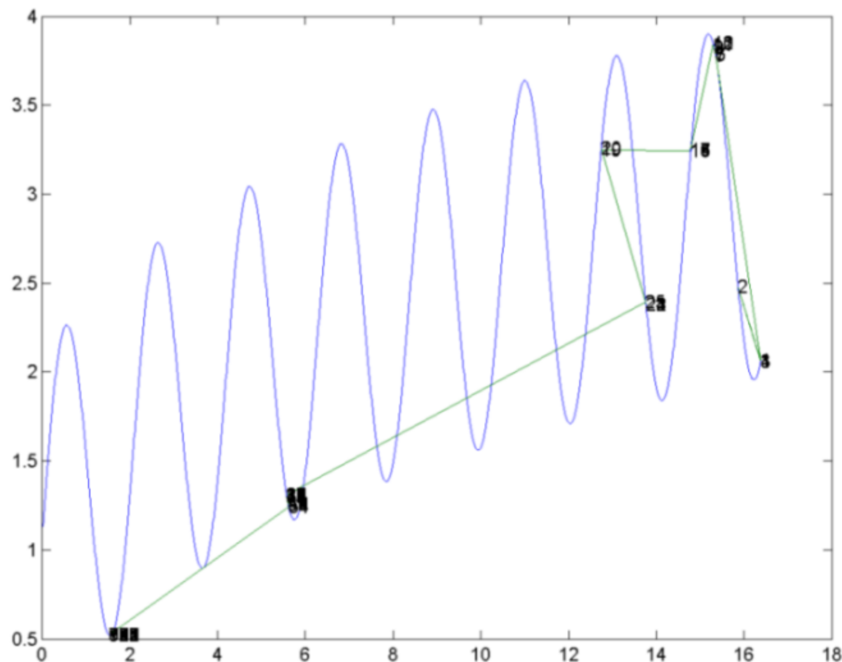
Iskanje s tabuji se uspešno uporablja za reševanje mnogo optimizacijskih problemov. V primerjavi z lokalnim iskanjem (local search) ima iskanje s tabuji več specifičnih komponent, ki morajo biti definirane. Iskanje s tabuji ima tudi večje (širše) območje preiskovanja kot lokalno iskanje. Pomembna je tudi svoboda oblikovanja posebnih komponent. Kako predstavimo prepovedani seznam, srednjeročni in dolgoročni spomin oz. kaj v njih shranjujemo je predvsem odvisno od lastnosti danega optimizacijskega problema. Iskanje s tabuji je lahko zelo občutljivo na nekatere parametre, npr. na dolžino prepovedanega seznama.

Poglavje 4

Simulirano ohlajanje

GREGOR SUŠELJ

4.1 Uvod v simulirano ohlajanje



Simulirano ohlajanje(SA) je stohastična metoda za reševanje optimizacijskih problemov. Razvila sta jo S. Kirkpatrick ter V. Cerny in jo sprva uporabila za reševanje problemov s področja kombinatorične optimizacije, kasneje pa so jo posplošili za reševanje zveznih optimizacijskih problemov. Metoda izvira iz mehanike, kjer lahko s segrevanjem in ohlajanjem spojine dosežemo določeno strukturo njenih kristalov. Ravnovesno stanje spojine po zaključku

tega postopka je odvisno od temperature, na katero smo spojino segreli in hitrosti ohlajanja. Če smo jo segreli premalo ali pa jo ohlajali prehitro, se pri vsaki temperaturi ne bo vzpostavila ustrezna kristalna struktura in spojina ne bo končala v najnižjem energijskem stanju. Zato jo moramo ohlajati počasi. Nekaj podobnega se dogaja pri reševanju optimizacijskega problema z metodo SA.

Za prvi vtis o delovanju algoritma si poglejmo njegovo skico v psevdokodi.

$s = s_0$ /začetni približek rešitve
$T = T_0$ /začetna temperatura
$while($zaustavitveni pogoj$)\{$
    $while($ravnovesni pogoj$)\{$
    naključno generiraj sosedno rešitev s'
    $if(f(s') \leq f(s))\{s = s'\}$ /vedno sprejmi novo rešitev
    $else\{$z verjetnostjo $\exp(\frac{f(s)-f(s')}{T})$ priredi $s = s'\}\}$
    $T = g(T)$ $\}$ /g je običajno padajoča funkcija
vrni rešitev s

Pri tem obstaja analogija med spremenljivkami in vrednostmi iz optimizacijskega algoritma ter količinami v fizikalnem modelu. Te povezave so prikazane v spodnji tabeli.

stanje snovi	trenutna rešitev
lega molekul	spremenljivke
energija snovi	funkcija, ki jo minimiziramo
stanje minimalne energije	globalni minimum
druga stabilna stanja	lokalni minimumi
temperatura	parameter T

Tabela 4.1: fizikalni model in optimizacijski algoritem

4.2 Parametri algoritma

Če želimo algoritem iz prejšnjega poglavja predstaviti v nekem programskem jeziku, bomo morali določiti še vse neznane parametre. Vrednosti parametrov, ki zagotavljajo hitro in uspešno konvergenco algoritma k najboljši rešitvi so močno odvisne od konkretnega optimizacijskega problema, njegove velikosti, ipd. Zaradi tega je o parametrih težko govoriti v splošnem, vseeno pa obstaja nekaj nasvetov, ki običajno vrnejo spodobne rezultate.

4.2.1 Začetni približek rešitve

Algoritem ne predpostavlja, da bi moral uporabnik znati ločiti boljše in slabše začetne približke, zato ta vrednost ni bistvena za pravilno delovanje. Določimo jo lahko naključno ali pa si izberemo nam najljubši približek.

4.2.2 Začetna temperatura

Gre za pomemben parameter. Če je temperatura zelo visoka, bo verjetnost sprejetja slabše rešitve $\exp(\frac{f(s)-f(s')}{T})$ vselej zelo blizu 1 in bomo sprejeli vsako rešitev, skoraj neodvisno od vrednosti $f(s) - f(s')$. Lahko se zgodi, da se bo algoritem dolgo naključno sprehajal po definicijskem območju funkcije in po nepotrebnem porabljal čas. Če je zelo nizka, bo $\exp(\frac{f(s)-f(s')}{T})$ približno enako 0 za vsako slabšo $f(s) - f(s') < 0$ rešitev s'. To pomeni, da bo algoritem sprejel le boljše rešitve in bo zelo verjetno skonvergiral v najbližji lokalni minimum. Prava začetna temperatura je nekje med tema dvema opcijama, kasneje pa se bo temperatura zmanjševala (glej shema ohlajanja). Navedimo dva možna postopka za določitev začetne temperature:

- Začetno temperaturo nastavimo tako visoko, da algoritem še sprejme vse sosede.
- Naključno izberemo nekaj vrednosti iz definicijskega območja $x_1, x_2, \ldots, x_{2n}$ in izračunamo standardni odklon σ na vrednostih $f(x_1) - f(x_2), f(x_3) - f(x_4), \ldots, f(x_{2n-1}) - f(x_{2n})$. Začetno temperaturo dobimo kot $T_0 = k\sigma$, kjer je k nek večkratnik. Priporočena vrednost je $k = -3/\log(p)$ in $p > 3\sigma$.

4.2.3 Shema ohlajanja

Shema ohlajanja je določena s funkcijo g, ki povezuje prejšnjo in naslednjo temperaturo $T_{i+1} = g(T_i)$. Zagotoviti mora, da temperatura pada proti 0 ($\lim\limits_{i\to\infty} T_i = 0$) in da temperatura pada dovolj počasi. Primeri shem ohlajanja so:

- $T_{i+1} = \alpha T_i$: Najpogosteje uporabljana shema. Običajno je $\alpha \in [0.5, 0.99]$.
- $T_i = T_0 - i\beta$: Algoritem se ustavi, ko je $T_i = 0$ ali pa že prej.
- Nemonotona: običajno temperatura monotono pada proti 0, obstajajo pa tudi optimizacijski problemi, pri katerih so pokazali, da je optimalna shema ohlajanja nemonotona.
- Prilagodljiva: praviloma je shema ohlajanja določena že pred začetkom izvajanja algoritma, v nekaterih primerih pa uporabimo dodatne informacije, ki jih dobimo med izvajanjem kot je vrednost $f(s)$, s, ipd. Takšno shemo imenujemo prilagodljiva shema ohlajanja.

4.2.4 Ravnovesni pogoj

Običajno se zahteva določeno število iteracij, npr. takšno, da algoritem sprejme določen delež sosednih rešitev. Uspešnost in hitrost algoritma se utegne izboljšati, če v ravnovesnem pogoju upoštevamo tudi vrednosti $f(s')$ in zanko zaključimo že prej, če npr. najdemo boljšo rešitev.

4.2.5 Zaustavitveni pogoj

Določa kdaj se algoritem ustavi. Primeri takšnih pogojev so:

- $T < T_{min}$,
- n korakov se rešitev ne spremeni.

4.3 Primer: problem trgovskega potnika

Uporabnost stohastičnih algoritmov pride do izraza pri računsko zahtevnih problemih. Eden najslavnejših izmed njih problemov je problem trgovskega potnika. Podan je poln neusmerjen utežen graf z n točkami, na katerem moramo poiskati najlažji hamiltoski cikel t.j. cikel, ki obišče vsako točko natanko enkrat. Obstaja še nekaj variacij problema kot so

- Ne-poln graf: problem je kvečjemu lažji, saj lahko dodamo manjkajoče povezave in nanje napišemo ∞ ali kako dovolj visoko število. Je pa mogoče, da rešitev ne obstaja.
- Usmerjen graf: uteži na povezavah so odvisne od smeri. V tem primeru je možnih rešitev dvakrat več.
- Uteži zadoščajo trikotniški enakosti: t.j. $U_{AB} \leq U_{AC}U_{CB}$. Izkaže se, da kljub tej dodatni zahtevi problem ostane NP-poln.

Znano je, da problema trgovskega potnika ni mogoče rešiti v polinomskem času (razen če P=NP). Če bi želeli preveriti vsako izmed možnih rešitev, bi za to potrebovali najmanj $O((n-1)!/2)$ časa, obstajajo pa tudi boljši algoritmi. Že dlje časa je znan Held–Karpov algoritem, ki s pomočjo dinamičnega programiranja problem reši v $O(n^2 2^n)$ časa.

Na http://www.tsp.gatech.edu/concorde je mogoče iz interneta prenesti za akademsko rabo prosto dostopni program Concorde, s katerim so reševali probleme z več deset tisoč točkami in je običajno zmožen tudi na navadnem računalniku v nekaj deset sekundah rešiti problem velikosti 500. Vseeno pa algoritem, ki bi rešil TSP v najslabšem primeru v času $O(1.999^n)$ ni znan.

Oglejmo si uporabo simuliranega ohlajanja za reševanje problema trgovskega potnika. Zaradi preglednosti bomo analizirali majhen primer z le 7 točkami. Označimo točke s števili $1, 2, 3, 4, 5, 6, 7$. V spodnji matriki je nekaj naključno in enakomerno na $[0, 1]$ generiranih števil. Naj število na mestu (i, j) za $i < j$ predstavlja utež na povezavi med točkama i in j.

$$\begin{bmatrix} 0 & 0.1890 & 0.4454 & 0.6846 & 0.3571 & 0.5201 & 0.5226 \\ 0 & 0 & 0.7133 & 0.1675 & 0.0921 & 0.2702 & 0.1748 \\ 0 & 0 & 0 & 0.1491 & 0.3322 & 0.7238 & 0.4157 \\ 0 & 0 & 0 & 0 & 0.0675 & 0.8688 & 0.0046 \\ 0 & 0 & 0 & 0 & 0 & 0.1767 & 0.9154 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.4317 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Na tako generiranem grafu želimo rešiti problem trgovskega potnika s pomočjo simuliranega ohlajanja. Parametri algoritma so nastavljeni:

- Začetna rešitev: permutacija 1234567
- Začetna temperatura: 1
- Shema ohlajanja: $T_{i+1} = 0.9 * T_i$
- Ravnovesni pogoj: preveri vsaj 3 sosednje rešitve ali najdi boljšo rešitev
- Zaustavitveni pogoj: $T < 0.05$
- Generiranje sosedne rešitve: naključno izberi 2 točki in ju zamenjaj

Potrebno je opomniti, da so nekateri parametri izbrani tako, da bo število iteracij zmerno in pregledno. Če bi želeli reševati večje probleme ali izboljšati natančnost, bi morali predvsem upočasniti shemo ohlajanja in pa nekoliko povečati razliko med začetno in končno temperaturo. Vmesni rezultati, izpisani po vsaki spremembi temperature so prikazani v spodnji tabeli.

Vidi se, da se je algoritem sprva dokaj naključno sprehajal med rešitvami in sprejemal tudi slabše vrednosti. Kasneje se je temperatura znižala, zato je algoritem težje sprejemal slabše vrednosti in je skonvergiral proti končni rešitvi 4356127 z vrednostjo 1.5465. Izkaže se, da to kljub temu ni najboljša rešitev(algoritem tega ne zagotavlja). Optimalna rešitev je 7431256 z vrednostjo 1.4884, ji je pa dobljena rešitev spodobno blizu.

4.4 Konvergenca algoritma

Opazimo, da je naslednji korak algoritma odvisen le od trenutnega stanja in trenutne temperature. Temu pravimo lastnost Markova ali včasih lastnost nepomnenja, saj si algoritem "ne zapomni"v katerih stanjih se je v preteklosti nahajal oziroma bolj natančno:

Naj $P(.|.)$ označuje pogojno verjetnost dogodka, $A(n)$ rešitev algoritma po n korakih in $i_0, i_2, \dots, i_n$ poljubno zaporednje rešitev. Potem velja:

$$P(A(n) = i_n | A(n-1) = i_{n-1}) = P(A(n) = i_n | A(n-1) = i_{n-1}, \dots, A(0) = i_0).$$

temperatura	vrednost funkcije	permutacija
1.0000	2.2497	1234567
0.9000	1.7907	1734562
0.8100	1.8062	7134562
0.7290	2.8234	7132564
0.6561	1.8909	7162534
0.5905	1.8909	7162534
0.5314	2.6756	7152643
0.4783	2.3781	7153624
0.4305	2.5335	3756124
0.3874	2.7050	3576124
0.3487	2.4136	3276154
0.3138	2.3241	3271654
0.2824	1.8062	3172654
0.2542	1.8062	3172654
0.2288	1.8062	3172654
0.2059	1.8062	3172654
0.1853	1.8062	3172654
0.1668	1.8062	3172654
0.1501	1.8062	3172654
0.1351	1.8062	3172654
0.1216	1.8062	3172654
0.1094	1.8062	3172654
0.0985	1.8155	7316254
0.0886	1.5627	4316527
0.0798	1.5627	4316527
0.0718	1.5627	4316527
0.0646	1.5465	4356127
0.0581	1.5465	4356127
0.0523	1.5465	4356127

Tabela 4.2: Vmesne rešitve algoritma

To pomeni, da lahko algoritem analiziramo kot nehomogeno markovsko verigo in uporabimo že znane lastnosti o limitnih porazdelitvah markovskih verig. Velja naslednji izrek:

Izrek 4.1 *Naj bo f funkcija, ki jo želimo minimizirati na končni množici stanj S, pri čemer uporabljamo funkcijo sosedov N(.) in nenaraščajočo shemo ohlajanja $T_1, T_2, \ldots$, ki limitira proti* 0. *Naj za vsaki stanji $i, j \in S$ velja, da lahko algoritem pride iz i v j v manj kot d korakih ter $i \in N(j) \Leftrightarrow j \in N(i)$. Naj bo $\Delta = \max\{f(j) - f(i); i \in S$ in $j \in N(i)\}$. Če je*

$$T_k \geq \frac{d\Delta}{\log k}$$

potem bo algoritem v limiti ko gre $k \to \infty$ z verjetnostjo 1 skonvergiral k enemu izmed globalnih minimumov neodvisno od začetnega približka.

Dokaz zgornjega izreka je dokaj obširen in ga je mogoče najti v [40].

4.5 Podobni algoritmi

Algoritem simuliranega ohlajanja se je razvil v sredini osemdesetih let, v naslednjih letih pa mu je sledilo še nekaj podobnih algoritmov. Ti algoritmi se razlikujejo predvsem po načinu sprejemanja slabše rešitve. Navedimo nekaj opcij:

1. Sprejmi sosednjo rešitev s', če je od prejšnje rešitve s slabša za največ Q. Pri tem vrednost Q posodabljamo (znižujemo) iz koraka v korak.

2. Sprejmi sosednjo rešitev, če je $f(s') <$ RECORD $+ D$, kjer je RECORD najnižja vrednost, funkcije f, ki jo je algoritem kdaj med izvajanjem izračunal in D neko število.

3. Sprejmi sosednjo rešitev, če je $f(s') <$ LEVEL, kjer se LEVEL posodablja kot LEVEL_{i+1} $=$ LEVEL_i-UP.

4. Sprejmi sosednjo rešitev, če $f(s') - f(s) < D$, kjer se D posodablja kot $D = D - (f(s') - f(s))$.

Poglavje 5

Genetsko programiranje

ALEŠ BIZJAK

5.1 Pregled genetskega programiranja

Genetsko programiranje je splošna metoda za avtomatsko iskanje računalniških programov s principi evolucije [1, 2]. To pomeni, da v genetskem programiranju, generacijo za generacijo, populacijo programov stohastično pretvarjamo v novo populacijo programov, ki upamo, da bolje rešijo dani problem.

Osnovna shema genetskega programiranja je podana na naslednji strani. Določiti moramo funkcije za izbiro posameznika, inicializacijo, mutacijo, križanje in funkcijo, ki ovrednoti dan program ter parametre kot so velikost populacije pop_size, verjetnosti križanja p_k in mutacije p_m, ter zaustavitveni kriterij oz. pogoj, ki nam pove, kdaj smo zadovoljni z rezultatom.

V osnovi je algoritem enak kot splošni genetski algoritmi, bistvena razlika pa je, da so pri genetskem programiranju osebki v populaciji programi poljubnih velikosti, ki niso v naprej predpisane. Najprej program poženemo in ga ovrednotimo. Ta del je zelo odvisen od problema, ki ga rešujemo. Mera uspešnosti je lahko npr. absolutna napaka pri iskanju funkcije, ki se najbolj prilega danim meritvam ali pa npr. količina porabljenih surovin in porabljen čas pri vodenju robotov. Programi, ki so boljši glede na dano funkcijo uspešnosti, bodo bolj verjetno izbrani kot vhod za genetske operatorje, torej bodo bolj verjetno imeli potomce in ob dobri nastavitvi parametrov algoritma bomo po nekaj generacijah, z upanjem dobili rešitev, ki je dovolj blizu optimalni in jo lahko uporabimo.

Osnovna genetska operatorja sta križanje in mutacija. Uporabljamo jih na predstavitvah programov in zato potrebujemo predstavitev, ki omogoča učinkovito izvedbo genetskih operatorjev.

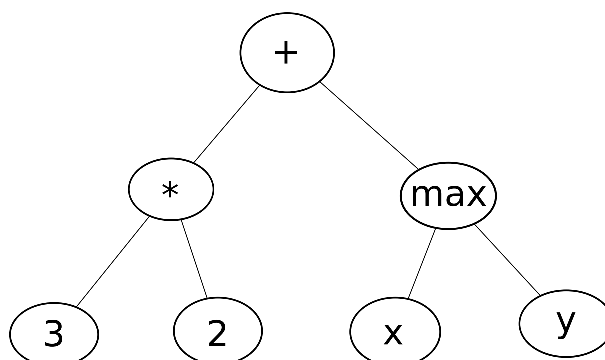
Algoritem 2 Osnovni algoritem genetskega programiranja.

$t \leftarrow 0$
$P(t) \leftarrow$ inicializiraj(pop_size)
$N \leftarrow$ najboljši$(P(t))$
while $t \leq G$ in rešitev ni dovolj dobra **do**
  $F(t) \leftarrow$ ovrednoti$(P(t))$
  $i \leftarrow 0$
  $P' \leftarrow \{\}$
  while $i < pop_size$ **do**
    $r \leftarrow$ rand$(0, 1)$
    if $r \leq p_m$ **then** {mutacija}
      $P'[i] \leftarrow$ mutacija(izberi_posameznika$(F(t), P(t))$)
    else if $r \leq p_m + p_k$ **then** {križanje}
      $s_1 \leftarrow$ izberi_posameznika$(F(t), P(t))$
      $s_2 \leftarrow$ izberi_posameznika$(F(t), P(t))$
      $P'[i], P'[i+1] \leftarrow$ križanje(s_1, s_2)
      $i \leftarrow i + 1$
    else if ... **then** {drugi genetski operatorji}
      ...
    else {reprodukcija}
      $P'[i] \leftarrow$ izberi_posameznika$(F(t), P(t))$
    end if
    $i \leftarrow i + 1$
  end while
  $t \leftarrow t + 1$
  $P(t) \leftarrow P'$
  $N' \leftarrow$ najboljši(P')
  if N' boljši od N **then**
    $N \leftarrow N'$
  end if
end while
return N

5.1.1 Predstavitev programov

Programe v genetskem programiranju ponavadi predstavimo kot abstraktna sintaktična drevesa, ali v kakšni drugi podobni obliki, npr. kot S-izraze ali kot izraze v poljski ali obratni poljski obliki. Takšna predstavitev je v primerjavi s predstavitvijo z izvorno kodo v programskem jeziku boljša, saj prihrani število korakov potrebnih za ovrednotenje programa. Prav tako pa je pri genetskih operatorjih lažje zagotoviti sintaktično pravilnost tako predstavljenih programov.

Pri taki predstavitvi so spremenljivke, konstante in drugi *atomi* v listih drevesa. Na notranjih vozliščih pa so *funkcije*. V primeru na sliki 5.1 so atomi $3, 2, x, y$, funkcije pa $*, +$ in $\max$.



Slika 5.1: Primer sintaktičnega drevesa za izraz $3 * 2 + \max(x, y)$.

Programi so lahko sestavljeni iz večih komponent, npr. programskih modulov, razredov ali funkcij. V tem primeru je predstavitev sestavljena iz množice dreves, ki jih skupaj povežemo tako, da dodamo skupen koren.

5.1.2 Inicializacija začetne populacije

Kot v drugih genetskih algoritmih moramo na začetku ustvariti začetno populacijo. Tipično to naredimo naključno, torej generiramo naključne programe, ki pa so seveda sintaktično pravilni in jih lahko ovrednotimo. Metod generiranja začetnih programov je veliko. Prvi dve, ki sta se pojavili in sta najenostavnejši, sta *polna* in *rastoča*. Pogosto se uporablja kombinacija teh dveh metod, kjer polovico programov generiramo po eni metodi, polovico pa po drugi metodi in generiramo sintaktična drevesa vseh možnih globin med izbranima mejama.

Tako v polni kot v rastoči metodi generiramo posamezne programe, ki ne presežejo neke vnaprej določene maksimalne globine. Pri polni metodi enostavno generiramo polna drevesa dane globine. Kljub temu, da so drevesa polna, nimajo vsa enakega števila vozlišč. To je posledica dejstva, da imajo lahko funkcije s katerimi jih gradimo, različno število argumentov (npr. funkciji $\sin$ in $+$). Kljub temu ta metoda ne generira dovolj raznolikih programov. V nasprotju s polno metodo, v rastoči metodi samo na začetku vedno izberemo funkcijo, kasneje pa gradimo stohastično in le ko pridemo do maksimalne globine drevesa, izberemo atom, da ne presežemo globine.

Ti metodi sta enostavni za implementacijo in uporabo, a imata veliko nezaželenih lastnosti. Ena izmed njih je, da je težko kontrolirati statistično distribucijo programov in velikosti generiranih programov ter njihove oblike. Zato obstajajo boljše, a tudi zahtevnejše metode.

Ni nujno, da začetne programe generiramo naključno. Če npr. že imamo neko rešitev problema, ki pa jo hočemo izboljšati, jo lahko uporabimo za seme in iz nje generiramo začetne programe. Ti bodo bolj verjetno zadovoljivo rešili dani problem. Tu je potrebno paziti, da ne izgubimo raznolikosti populacije. Že znane ročne rešitve so lahko veliko boljše od naključno spremenjenih in hitro preplavijo populacijo, posledično pa se začetne rešitve ne izboljšajo.

5.1.3 Selekcija

S selekcijo izberemo iz trenutne populacije osebke na katerih bomo uporabili genetske operatorje. Pri genetskem programiranju se najpogosteje uporablja *turnirska selekcija*, sledi pa *selekcija z rangiranjem*. Odvisno od primera uporabe se uporabljajo tudi druge metode selekcije.

Pri turnirski selekciji iz populacije naključno izberemo nekaj osebkov (koliko, je odvisno od velikost turnirja). Potem jih med sabo primerjamo glede na funkcijo uspešnosti in izberemo najboljšega. To ponovimo tolikokrat, kolikor osebkov potrebujemo za dani genetski operator. Npr. za mutacijo potrebujemo enega starša, za križanje pa dva. Pomembna prednost turnirske selekcije je, da gledamo samo kateri osebek je boljši, ne pa tudi koliko boljši. Tako nek osebek, ki je trenutno veliko boljši od ostalih nima velike prednosti pred ostalimi in težje preplavi populacijo s svojimi potomci. S tem poskrbimo, da ne ostanemo v lokalnem ekstremu. Z velikostjo turnirja vplivamo na evolucijski pritisk. Če je turnir majhen, bodo slabši osebki imeli večje možnosti proizvesti potomce, če pa je zelo velik, bodo zelo dobri osebki s svojimi potomci hitro preplavili celotno populacijo.

5.1.4 Križanje in mutacija

Najpogostejša verzija križanja je križanje poddreves. V vsakem od obeh staršev izberemo vozlišče v drevesu in zamenjamo poddrevesi s korenom v danem vozlišču. S tem dobimo dva nova sintaktično pravilna programa. Tipično vozlišč za križanje ne izbiramo enakomerno med vsemi vozlišči, saj je večina vozlišč listov in bi s tem večino časa zamenjevali samo po dva lista v drevesih, česar nočemo, saj ne dobimo dovolj raznolikosti v programih. Zato izbiramo notranja vozlišča z verjetnostjo (tipično $p = 0.9$) in liste v preostalih primerih.

Najpogostejša oblika mutacije je mutacija poddreves, ki sestoji enostavno iz tega, da naključno izbrano poddrevo v staršu zamenjamo z naključno generiranim drevesom. Pogosta oblika mutacije je točkasta mutacija, ki je v sorodu s točkasto mutacijo pri genetskih algoritmih. Za vsako vozlišče v danem drevesu se glede na verjetnost mutacije odločimo ali ga bomo zamenjali z drugim ali ne. Pri tem moramo paziti da vozlišče zamenjamo z vozliščem istega tipa. Npr. atome lahko zamenjamo samo z atomi in enomestne funkcije le z enomestnimi.

S tem zagotovimo, da vedno dobimo sintaktično pravilen program, ki ga lahko poženemo in ovrednotimo.

Izbira genetskega operatorja je stohastična. Tipično vzamemo verjetnost izbire križanja okoli 0.9, verjetnost izbire mutacije pa veliko manj, okoli 0.01. če ni izbran noben izmed genetskih operatorjev, torej če skupna verjetnost izbire genetskih operatorjev ni 1, uporabimo še reprodukcijo, kjer izbrani osebek enostavno kopiramo v novo populacijo.

5.2 Podrobnosti

Preden lahko uporabimo genetsko programiranje, moramo določiti nekaj parametrov in se odločiti, katere funkcije in atome bomo uporabili, kako bomo rezultate vrednotili in kdaj bomo zadovoljni z rezultatom.

5.2.1 Jezik

Z genetskim programiranjem iščemo programe. Programi so lahko podani v splošnih, Turingovo polnih programskih jezikih, bolj pogosto pa iščemo programe za reševanje neke specifične naloge in so zato jeziki specifični domeni, ki jo rešujemo. Programe sestavimo iz funkcij in terminalnih simbolov, t.j. atomov. Pred reševanjem pa se moramo odločiti, iz katerih funkcij in atomov bomo sestavljali naše programe.

Atomi so lahko spremenljivke, ki predstavljajo vhodne podatke programa, funkcije brez argumentov, tipično so to funkcije s stranskimi učinki, npr. če iščemo programe za vodenje robotov, bi lahko imeli funkcijo `TURN_LEFT()` in konstante, ki so lahko numerične vrednosti ali pa simboli drugih tipov, ki imajo konkreten pomen v dani domeni.

Množica funkcij je odvisna od problema, ki ga rešujemo. Pri simbolični regresiji so to lahko aritmetične funkcije $+$, $-$, $*$, $/$, $\sin$, $\cos$, pri evoluciji programov za vodenje robotov so to lahko pogojni stavki, premiki, zanke in podobno.

Za učinkovito delovanje genetskega programiranja sta pomembni *zaprtost* in *zadostnost* množice funkcij in atomov. Zadostnost pomeni, da lahko rešitev danega problema izrazimo z danimi funkcijami in atomi. Zaprtost pomeni, da imajo vse funkcije enako, dovolj veliko domeno, da križanje in mutacija vedno rodita sintaktično pravilne programe, ki jih lahko poženemo in ovrednotimo. To je v praksi včasih težko doseči. Pri enostavnih problemih, kot je recimo simbolična regresija, to lahko enostavno zagotovimo. Funkcije, ki v nekaterih točkah niso definirane, razširimo. Npr. definiramo $x/0 = 1$ za vse x. S tem zagotovimo, da pri izračunu vrednosti ne pride do nedefiniranih rezultatov in lahko vedno ocenimo kako dober je program. Dostikrat težav ne moremo tako enostavno rešiti. V teh primerih moramo ali spremeniti genetske operatorje, da spoštujejo tipe funkcij in ohranjajo pravilnost programov, ali pa zavrniti programe oz. jim prirediti zelo nizko oceno, da hitro izpadejo iz populacije.

Z genetskim programiranjem lahko iščemo tudi rešitve, ki niso programi, npr. načrte za

antene. V takih primerih predstavimo osebke kot programe, ki sestavijo rešitev. Funkcije so potem načini združevanja komponent, atomi pa osnovne komponente.

5.2.2 Funkcija uspešnosti

Uspešnost programa lahko merimo na več načinov, odvisno od problema. Pri simbolični regresiji je tipično uspešnost programa povezana z odstopanjem najdene funkcije od podanih podatkov oz. meritev. Posebnost meritve uspešnosti programov pri genetskem programiranju je, da moramo tipično izvršiti program, včasih večkrat. Če so programi majhni in jih lahko hitro izvršimo, uporabimo tolmača, če pa je program zelo zahteven in je izvržba časovno zelo zahtevna, lahko program prevedemo v strojno kodo in ga potem poženemo.

Pri nekaterih problemih je rezultat izvržbe programa neka izhodna vrednost in tedaj lahko vzamemo kar to vrednost za mero uspešnosti programa. V drugih primerih ima program samo stranske učinke in neposredno ne vrne nobene vrednosti, npr. program za vodenje robota. V takih primerih moramo posebej ovrednotiti stranske učinke.

5.2.3 Parametri

Določiti moramo še parametre, ki določajo velikost populacije, največje število generacij, verjetnosti izbire genetskih operatorjev, maksimalno velikost programov in podobne podrobnosti.

Parametri so zelo odvisni od problema, ki ga rešujemo. V nekaterih primerih deluje ena izbira bolje, v nekaterih druga. Velikost populacije je tipično taka, da v razumnem času lahko ovrednotimo vse programe in generiramo novo generacijo. To je lahko nekaj deset ali pa nekaj milijonov programov. Tipično je ravno izvajanje in vrednotenje programov časovno najzahtevnejši del algoritma. Ostalo, genetski operatorji in selekcija, tipično porabi veliko manj časa.

5.3 Primer uporabe

Oglejmo si enostaven primer genetskega programiranja na primeru simbolične regresije. Iščemo izraz, katerega vrednosti so enake vrednostim kvadratne funkcije $x \mapsto x^2 + x + 1$ na intervalu $[-1, 1]$.

Najprej moramo določiti jezik. Ker delamo simbolično regresijo bodo funkcije navadne matematične funkcije in ker iščemo funkcijo ene spremenljivke, bo med atomi natanko ena neodvisna spremenljivka, x. Med atome vključimo tudi numerične vrednosti na nekem intervalu, recimo $[-5, 5]$. Množica atomov, T, je torej $T = \{x, \mathcal{R}\}$, kjer $\mathcal{R}$ označuje konstante iz intervala $[-5, 5]$. Za množico funkcij izberimo $\{+, -, *, /\}$, kjer je / deljenje, ki vrne 0, kadarkoli je delitelj enak 0. S tako izbranim jezikom je vsak program v populaciji neka racionalna funkcija definirana za vse vrednosti x.

Za mero uspešnosti programa vzemimo integral absolutne napake, t.j. naj bo f najden

program. Potem je uspešnost programa definirana kot

$$\int_{-1}^{1} |f(x) - x^2 - x - 1| \,\mathrm{dx}.$$

V praksi nimamo podanih vrednosti na celotnem intervalu in zato integrala ne moremo natančno izračunati, lahko pa uporabimo numerične metode in ga dobro aproksimiramo.

Da bo primer znosen, naj bo velikost populacije enaka 4. Za križanje bomo izbrali dva programa, za reprodukcijo in mutacijo pa po enega. V realnih primerih bi križanje izvedli z verjetnostjo okoli 0.9, reprodukcijo z 0.08, mutacijo z 0.01 in ostale genetske operatorje v preostalih primerih.

Z rezultatom bomo zadovoljni, ko bo napaka nekega najdenega programa manjša od 0.01.

Sedaj začnemo z genetskim programiranjem. Denimo, da smo na začetku ustvarili programe $(x+1)-0$, $1+(x*x)$, $2+0$ in $x*(-1-(-2))$. Seveda smo ustvarili drevesa, ki predstavljajo te izraze. Npr. prvi izraz smo ustvarili z rastočo metodo. Za koren smo izbrali funkcijo $-$. Potem smo v levem poddrevesu najprej izbrali za koren poddrevesa funkcijo $+$. Potem smo za vsako od poddreves tega drevesa izbrali atom. Za levo poddrevo smo izbrali x, za desno 1. Nato smo za desnega sina korena $-$ izbrali atom 0. Podobno smo z rastočo metodo dobili tudi ostale programe. Seveda naključno generirani programi tipično niso zelo dobri pri reševanju problemov, a nekateri so malo boljši od drugih in to neenakost evolucijski proces izkoristi. V tem primeru je napaka prvega programa 0.67, drugega 1.0, tretjega 1.67 in zadnjega 2.67.

Sedaj nastopijo genetski operatorji. Najprej izberimo program za reprodukcijo. Ker je prvi program najboljši, bo najverjetneje izbran, zato predpostavimo, da res je in ga skopirajmo v novo populacijo. Zatem naredimo mutacijo in denimo da je bil za mutacijo izbran tretji program. Denimo, da smo za mutacijo izbrali vozlišče 2 in da smo to poddrevo (ki sestoji samo iz enega vozlišča) zamenjali z naključno generiranim drevesom x/x, ki je dejansko ekvivalentno 1, a tega algoritem ne ve. Dobimo program $(x/x)+0$.

Sedaj naredimo še križanje in denimo, da smo za to izbrali prvi in drugi program. To je tudi najbolj verjetno, saj sta ta dva programa najboljša. Denimo, da smo za točko križanja v prvem programu izbrali vozlišče $+$, v drugem pa najbolj levo vozlišče, ki vsebuje x. Potem z zamenjavo izbranih poddreves dobimo programa $x-0$ in $1+((x+1)*x)$. Drugi je ekvivalenten x^2+x+1. Napaka drugega programa je enaka 0 in zato se algoritem ustavi. V tem primeru se je celo zgodilo, da je najden program popoln, kar se v praksi seveda skoraj nikoli ne zgodi. Križanje je združilo dve dobri lastnosti posameznih programov v enem programu. Konkretno drugi program je imel kvadratni člen in člen $+1$, prvi pa linearni člen $x+1$. Nastali program je tako podedoval dobre lastnosti obeh staršev.

5.4 Implementacija

Implementiral sem algoritem za simbolično regresijo. Za podano tabelo $n{+}1$-teric $(x_1,\ldots,x_n,y)$, kjer program razume, da je so $x_1,x_2,\ldots,x_n$ neodvisne spremenljivke, y pa odvisna, program

išče izraz, sestavljen iz danih funkcij, ki se najbolje prilega danim podatkom.

Implementacija v splošnem ne odstopa veliko od osnovnega algoritma. Programi so predstavljeni v poljski (prefiksni) obliki, ne pa kot eksplicitna drevesa, saj je tako mogoče genetske operatorje izvesti učinkoviteje.

Ocena vrednosti programov je velikost odstopanja od podanih točk. Poleg križanja program uporablja še mutacijo po vozliščih in mutacijo z zamenjavo poddrevesa. Verjetnosti, kateri od teh operatorjev bo izbran je možno nastavljati. Vse možne nastavitve lahko dobimo z zastavico `--help`.

Spodaj je primer uporabe. Želimo čim bolj aproksimirati funkcijo $x \mapsto \cos(2x)$, ki je tabelirana na intervalu $[-1, 1]$. Za funkcije smo vzeli $+, -$ in $\sin$, terminalni simboli pa so spremenljivka x_0 in konstante $-2, -1, 0, 1, 2$. Algoritem na koncu najde program, ki sicer ne izgleda preveč lepo, a če ga poenostavimo, vidimo da je ekvivalenten izrazu $\sin(1.5708 - 2x_0)$, kar je skoraj enako $\sin\left(\frac{\pi}{2} - x_0\right)$, kar pa je znana identiteta. Torej, kljub temu, da konstante $\frac{\pi}{2}$ nismo podali, jo je algoritem uspel samostojno najti in uporabiti.

```
$ ./genetic --seed 0 cos2x.txt -fsin,+,-
INPUT:
-1 -0.416147
-0.9 -0.227202
-0.8 -0.0291995
-0.7 0.169967
-0.6 0.362358
-0.5 0.540302
-0.4 0.696707
-0.3 0.825336
-0.2 0.921061
-0.1 0.980067
-2.22045e-16 1
0.1 0.980067
0.2 0.921061
0.3 0.825336
0.4 0.696707
0.5 0.540302
0.6 0.362358
0.7 0.169967
0.8 -0.0291995
0.9 -0.227202
1 -0.416147

CONFIGURATION

Population size:                 30000
Maximum number of generations:   100
Tournament size                  20
Crossover probability:           0.9
Crossover mutation probability:  0.05
Point mutation probability:      0.05
Target precision:                1e-14
Program size when pen. applies:  200
Using unary functions:sin
Using binary functions: +  -
```

```
EVOLUTION

GENERATION: 0
Best fit: -1.66453 Worst fit: -621.364 Average fit: -33.1247
Average size: 27.4149 Maximum size: 333 Minimum size: 3

GENERATION: 1
Best fit: -0.244987 Worst fit: -228.764 Average fit: -15.5537
Average size: 19.9678 Maximum size: 213 Minimum size: 2
Best found so far: -0.244987

...

GENERATION: 28
Best fit: -3.26128e-15 Worst fit: -176.523 Average fit: -1.20317
Average size: 154.827 Maximum size: 393 Minimum size: 2
Best found so far: -3.26128e-15

FOUND A SOLUTION

MATHEMATICA:
sin [((sin [1 ] - (x0  + x0 )) - (sin [0 ] - sin [sin [sin [(2  - sin
[sin [sin [((1  - (-1) ) - sin [((1  - (-2) ) - sin [sin [sin [(2  -
sin [sin [sin [(2  - sin [sin [((1  - (-1) ) - sin [(sin [sin [sin [sin
[sin [(2  - sin [sin [sin [(2  - sin [sin [((1  - (-1) ) - sin [(2  - sin
[(2  - sin [sin [sin [sin [sin [(2  - sin [sin [((1  - (-1) ) - sin [(sin
[sin [sin [sin [sin [(2  - sin [sin [sin [(2  - sin [sin [((1  - (-2) ) -
sin [(2  - sin [(sin [sin [sin [sin [sin [(2  - sin [sin [sin [(2  - sin
[sin [((1  - (-1) ) - sin [(2  - sin [(2  - sin [sin [sin [sin [sin [(2
- sin [sin [sin [(2  - sin [sin [((1  - (-1) ) - sin [sin [(((1  - (-1) )
- sin [(2  - sin [sin [sin [sin [sin [(2  - sin [sin [((1  - (-1) ) - sin
[(sin [sin [(0  - (-1) )]] - sin [sin [(-1) ]])])]])]]]]])]) - (x0  - x0
))]])]])]]])]]]]])])])]])]]])]]]]] - sin [sin [(-1) ]])])])]])]]])]]]]] -
sin [(1  - (-1) )])])]])]]]]]])])])]])]]])]]]]] - sin [sin [(-1) ]])])]])]
]])]]]])])]]])]]]]))]
SIZE 196
```

Poglavje 6

Optimizacija s kolonijami mravelj

JAKA KRANJC

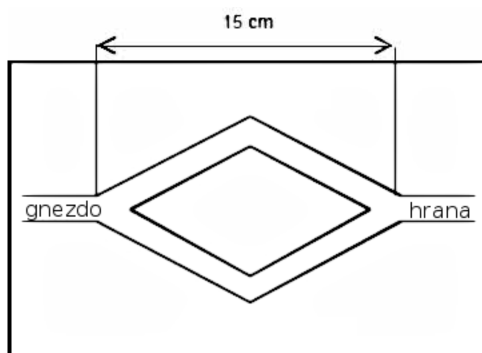
6.1 Uvod

Pri preučevanju narave so ljudje ugotovili, da so živali in rastline razvile iznajdljive tehnike za preživetje. Sedaj lahko te tehnike uporabimo za reševanje problemov, ki bi jih sicer reševali dlje časa. Konec koncev je imela narava več milijonov let, da je izoblikovala to, kar ima sedaj. Če se samo omejimo in analiziramo tehniko, s katero mravlje iščejo hrano, dobimo optimizacijo temelječo na koloniji mravelj.

Slika 6.1: Sledenje mravelj v naravi

Čeprav nam pogled na mravlje ne da slutiti, da gre za naprednejša bitja, nam podrobnosti razkrijejo, da so zelo učinkovite za vsaj eno nalogo – iskanje najkrajše poti med danima dvema točkama, gnezdom in virom hrane, kar je eksperimentalno dokazano v [30]. Če se osredotočimo na mravljo, ki je pravkar prišla iz gnezda, vidimo, da si, vse dokler ne pride do vira hrane, izbira smeri gibanja pretežnoma naključno. Ob vračanju v gnezdo pa za sabo spušča sled privlačnih kemičnih snovi – feromonov, katerih količina je odvisna od količine in kakovosti hrane, ki jo je našla in s katerimi si kasneje pomaga poustvariti pot z namenom hitrejšega dostopa do predhodnega vira.

Opazili so, da se ob prisotnosti feromonov začnejo mravlje gibati predvidljivejše – poti z močnejšo koncentracijo feromonov si izberejo z večjo verjetnostjo. To so potrdili z eksperimentom, ki je povezoval izolirano gnezdo z virom hrane preko mostu, ki je bil sestavljen iz dveh enako dolgih krakov, glej sliko 6.2. Čeprav so imele mravlje svobodno izbiro med levim ali desnim krakom, se je hitro videlo, da so skoraj vse mravlje začele uporabljati samo en krak, kar je potrdilo tudi prisotnost feromonov.



Slika 6.2: Most z dvema krakoma

V primeru, da mravlja naleti do križišča z dvemi ali več sledmi, se lahko glede na koncentracijo snovi odloči, ali bo zapustila trenutno pot in začela slediti neki novi ali bo vstrajala na sedanji. Če jih je izbrana pot vodila do hrane, nanjo spustijo še feromone in jih tako še dodatno okrepijo.

Feromoni sčasoma enakomerno pojenjajo – izhlapevajo, vendar se intenzivnost feromonov na krajših poteh kljub temu povečuje. Mravlje, ki izberejo krajšo pot, pridejo prej do hrane (in hitreje tudi nazaj v gnezno), zato lahko v istem časovnem intervalu večkrat pustijo feromone. Po nekem času, se bo večina mravelj preusmerila na krajše poti, saj je tam sled feromonov najmočnejša. Poleg tega se mravlje zelo dobro prilagodijo na spremembe v okolju. Če samo odstranimo ali spremenimo kakšno pot, bodo zelo hitro našle novo.

6.2 Algoritem

Marco Dorigo, Alberto Colorni in Vittorio Maniezzo [31, 32] so bili prvi, ki so v devetdesetih letih zasnovali osnove metahevristike za simuliranje obnašanja mravelj, poimenovano algoritem kolonije mravelj oziroma optimizacija s kolonijami mravelj, krajše ACO, ko ga uporabimo v smislu kombinatorične optimizacije. Temelji na obnašanju resničnih mravelj, natančneje na njihovi tehniki iskanja hrane in tehniki uporabe feromonov.

ACO vsebuje splošne metode reševanja različnih problemov, ki jih lahko razumemo kot nabor hevrističnih metod za uporabo na številnih problemih, in spada v družino algoritmov o inteligenci rojev, kjer se uporabi večje število neodvisnih enot (npr. mravlje, čebele, ose, ribe, ptice ...), ki nato rešujejo problem tako, kot če bi bila to živa bitja. Enota (tudi agent) je sama

po sebi precej neučinkovita in enostavna, vendar jim je dovoljeno posredno sodelovanje, to je skupno ocenjevanje dobrih rešitev, kar je poglavitna prednost tega razreda, zato je skupek enot primeren za reševanje kompleksnejših problemov in tudi takšnih, ki se spreminjajo s časom.

Ker lahko kombinatorični optimizacijski problem predstavimo z odločitvenim grafom (tudi konstrukcijski graf), v katerem so rešitve poti, postane njihova naloga iskanje poti, ki jih določijo tako z naključnimi premiki kot tudi z upoštevanjem zaželjenost – težimo k boljši rešitvi. V primeru algoritma ACO enote komunicirajo preko sledi feromonov, ki izražajo dobre poti iz predhodnih rešitev in služijo za osnovno vodilo pri grajenju novih.

Splošen algoritem 3 je sestavljen iz dveh delov, ki se ponavljata – iterirata. Prvi del je namenjen grajenju m rešitev in drugi posodobitvi feromonov, kjer m označuje vnaprej predpisano število enot. Ponavadi se zaključni pogoj določi glede na število iteracij.

Algoritem 3 Optimizacija s kolonijo mravelj, povzeta po [3]

```
inicializacija
repeat
   {korak}
   for i := 1 → m (število mravelj) do
      i-ta mravlja je postavljena na začetno točko
      i-ta mravlja upošteva feromone in zgradi rešitev
   end for
   posodobitev feromonov
until zaključni pogoj
print najboljša rešitev
```

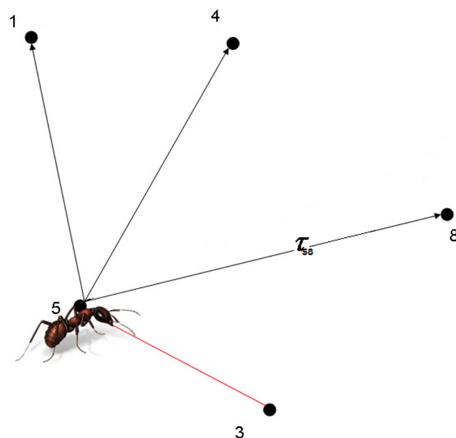
6.2.1 Grajenje rešitev

Pri grajenju posamezne rešitve se pri izbiri naslednjega vozlišča odločamo tako, da zadostimo danim omejitvam in upoštevamo hevristične zaželenosti (npr. čim nižja cena povezave). Slednje je znano tudi kot pravilo izkoriščanja in pravilo raziskovanja.

- Izkoriščanje: Verjetnost izbire naslednjega vozlišča je sorazmerna s koncentracijo feromonov na povezavi, ki vodi do tega vozlišča.
- Raziskovanje: Verjetnost izbire naslednjega vozlišča je sorazmerna z verjetnostjo, da bo povezava do vozlišča tvorila boljšo rešitev.

Dane omejitve porodijo množico dovoljenih premikov D_i, zato je verjetnost prehoda iz i-tega na j-to vozlišče, ki jo poimenujemo tudi pravilo prehoda stanj (*angl.* state transition rule), enaka

$$P_{ij} = \begin{cases} \frac{\eta_{ij}^{\alpha}\xi_{ij}^{\beta}}{\sum_{k\in D_i}\eta_{ik}^{\alpha}\xi_{ik}^{\beta}} & j \in D_i \\ 0 & j \notin D_i, \end{cases} \tag{6.1}$$



Slika 6.3: Enota (mravlja) je med 4 kandidati za naslednje vozlišče izbrala vozlišče 3

kjer je η_{ij} verjetnost izkoriščanja in ξ_{ij} verjetnost raziskovanja glede na povezavo ij ter α in β deleža vpliva prvega in drugega pravila.

Za parametra α in β se odločimo glede na problem. Če zanemarimo prvo pravilo ($\alpha \ll \beta$), potem mravlje izbirajo boljše povezave oziroma se nagibajo k požrešni metodi. Če pa zanemarimo drugo pravilo, mravlje prekomerno upoševajo feromone in premalo raziskujejo, kar vodi do nenujno optimalne rešitve, zato je pomembna uravnotežena izbira teh dveh parametrov.

vozlišče	η_{ij}	η_{ij}^2	ξ_{ij}	P_{ij}
1	0,2	0,04	2	0,021
2	0,7	0,49	1,5	0,197
3	0,1	0,01	6	0,016
4	0,6	0,36	3,5	0,337
5	0,8	0,64	2,5	0,428

Tabela 6.1: Primer izračuna verjetnosti za 5 vozlišč pri $\alpha = 2$ in $\beta = 1$

6.2.2 Posodobitev feromonov

Feromoni porazdeljeni po povezavah predstavljajo edini komunikacijski medij med enotami, zato jih je potrebno redno posodabljati, kar se stori po vsaki iteraciji, ko že vse mravlje zgradijo svojo rešitev. Najprej se na vseh povezavah zniža nivo feromona za konstanten, vnaprej določen, faktor, kar lahko zapišemo

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t),$$

kjer je $\tau_{ij}(t)$ nivo feromonov na povezavi iz vozlišča i v vozlišče j ob iteraciji t in $\rho \in [0, 1]$ delež razpada. Namen izhlapevanja je preprečevanje prekomernega kopičenja feromonov in omogočanje skokov iz lokalnega minimuma.

Nato pa se okrepijo feromoni na povezavah, ki so jih enote uporabile v svoji rešitvi. Obstaja več strategij, katere mravlje spustijo feromone in koliko jih spustijo. Naštejmo najpogostejše globalne posodobitve.

- V vsaki iteraciji imamo samo eno mravljo ($m = 1$):
  - Za vsako rešitev se okrepijo feromoni.
  - Feromoni se okrepijo, če najdemo boljšo rešitev.
- V vsaki iteraciji imamo m mravelj:
  - Za najboljših k rešitev trenutne iteracije se okrepi feromone, kjer za k velja $1 < k \leq m$.
  - Za l do sedaj globalno najboljših rešitev se okrepi feromone.
  - Najslabša rešitev zmanjša količino feromonov na tej rešitvi.

Obstajajo še lokalne posodobitve, kjer se feromon posodobi takoj po izboru povezave že pri samem grajenju rešitve.

Ker gre za aproksimativno metodo, se izbere strategijo in delež razpada eksperimentalno glede na problem. Količino okrepitve feromonov ponavadi določimo sorazmerno s kakovostjo rešitve. Pomembna je tudi pravilna izbira sorazmernega faktorja okrepitve. V primeru, da je prevelik, bo rešitev ene mravlje prekomerno vpivala na rešitev drugih mravelj; v primeru, da je premajhen, se lahko zgodi, da bodo sledi enakomerno porazdeljene. V prvem primeru se bodo rešitve mravelj kmalu po začetku poenotile, medtem ko se mravlje v drugem primeru vedejo pretežno po modelu naključnega sprehoda, če seveda zanemarimo raziskovanje.

Po potrebi lahko po vsaki iteraciji najboljšo rešitev še dodatno okrepimo.

6.3 Razredi

Glede na uporabljeno strategijo posodobitve in načinom grajenja rešitev se deli ACO še na razrede.

- **Ant System (AS)** – 1991

  Sistem mravelj je najenostavnejši in prvi algoritem v razredu ACO ter predstavlja prototip za vse algoritme, ki temeljijo na koloniji mravelj. Zasnoval ga je Marco Dorigo leta 1991 in ga opisal v svojem doktorskem delu [32]. Obstajajo tri različice: *ant-density*, *ant-quantity* in *ant-cycle*. Pri prvih dveh različicah se feromoni na povezavi ojačajo takoj, ko jo mravlja uporabi v svoji rešitvi, vendar se različici nista izkazali za učinkovite, zato se v večini primerov enači sistem mravelj z *ant-cycle*.

Pri *ant-cycle* si vsaka od m mravelj ustvari klasično rešitev, kot je opisano v razdelku 6.2.1. Nakar se za vsako rešitev odložijo feromoni. Količina feromonov, ki jih odloži mravlja, je funkcijsko odvisna od kvalitete njene rešitve. Izkazalo se je, da je sistem mravelj uporaben kvečjemu za manjše probleme (do 30 vozlišč). Zaradi tega so se razvili še drugi razredi.

- **Elitist Ant System (EAS)**

  EAS je prva izboljšava sistema mravelj. Po vsakem koraku iteracije, ko se zgradi m rešitev, se okrepi feromone bodisi le za globalno najboljšo rešitev bodisi za najboljšo rešitev v iteraciji (elitist strategy). Feromoni se spremenijo po

  $$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}^{\text{best}},$$

  kjer je $\Delta\tau_{ij}^{\text{best}} = 1/L^{\text{best}}$ kakovost globalno najboljše rešitve ali najboljše rešitve v iteraciji.

  Ostale rešitve iteracije ne spremenijo nivoja feromonov. Rešitve se zgradi klasično. Izkazalo se je, da EAS hitreje najde boljše rešitve od AS, vendar relativno hitro pride do nasičenja feromonov in do konvergence k nenujno optimalni rešitvi.

- **Rank-Based Ant System (ASrank)**

  Pri ASRANK se feromoni vedno odložijo za globalno najboljšo rešitev kot pri EAS. Poleg tega se pri tej strategiji najboljših k rešitev trenutne iteracije priredi količina feromonov glede na njihovo kakovost. Ponavadi se izbere število (utež) w, ki jo priredimo r-ti najboljši rešitvi v vrednosti $k - r$. Tedaj se feromoni spremenijo po

  $$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum r = 1^{w-1}(w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{\text{gb}},$$

  kjer je $\Delta\tau_{ij}^{\text{gb}}$ kakovost globalno najboljše rešitve in $\Delta\tau_{ij}^{r}$ kakovost r-te rešitve.

  Izkazalo se je, da je ASRANK uspešnejši od EAS in sistema mravelj.

- **Max-Min Ant System (MMAS)** – 1996

  Glavna ideja MMAS je količinska omejitev feromonov. To pomeni, da si pred reševanjem predpišemo najnižjo $\tau_{\min}$ in najvišjo $\tau_{\max}$ koncentracijo feromonov na vsaki povezavi oziroma krajše

  $$\forall i, j, t : \quad \tau_{ij}(t) \in [\tau_{\min}, \tau_{\max}].$$

  Na začetku so sledi feromona nastavljene na maksimalno vrednost $\tau_{\max}$, da razširimo raziskovanje tudi na povezave, ki imajo nižjo verjetnost izbire. Raziskave so pokazale, da je nastavitev najnižje koncentracije pomembnejša od nastavitve najvišje, saj je ta omejena že zaradi izhlapevanja.

  Feromoni so okrepljeni tako kot v razredu EAS in rešitve so zgrajene tako kot pri sistemu mravelj.

- **Ant-Q** – 1995

  ANT-Q je nadgraditev sistema mravelj z idejami iz metode Q-learning. ANT-Q se kasneje poenostavi v ACS.

- **Ant Colony System (ACS)** – 1997

  Posebnost sistema kolonije mravelj je spremenjeno pravilo prehoda stanj (6.1), ki izvira iz metode Q-learning. Na začetku določimo vrednost $q_0 \in [0, 1]$, ki predstavlja mejo med popolnim izkoriščanjem in raziskovanjem in je analogija temperature pri simuliranem ohlajanju.

  Pri grajenju rešitve se pred vsako izbiro naslednjega vozlišča naključno izbere število $q \in [0, 1]$. Če je $q < q_0$ (popolno izkoriščanje sledi in zaželjenosti), potem se za naslednje vozlišče vzame vozlišče, katerega povezava ima največjo verjetnost oziroma je najboljša, sicer se klasično nadaljuje po (6.1) (raziskovanje). Novo pravilo (6.2) se imenuje psevdonaključno sorazmerno pravilo (*angl.* pseudo random proportional rule). Ponavadi za q vzamemo 0.9, $\alpha = 1, \beta = 2$.

  $$P_{ij} = \begin{cases} \max_{j \in D_i}\{j; \eta_{ij}^{\alpha}\xi_{ij}^{\beta}\} & q < q_0 \\ \frac{\eta_{ij}^{\alpha}\xi_{ij}^{\beta}}{\sum_{k \in D_i} \eta_{ik}^{\alpha}\xi_{ik}^{\beta}} & q \geq q_0 \text{ in } j \in D_i \\ 0 & \text{sicer.} \end{cases} \tag{6.2}$$

  Če je q_0 blizu 1, se izbira le lokalne optimalne rešitve. Če je q_0 blizu 0, se razišče vse rešitve, vendar še vedno malenkostno težimo k optimalnim lokalnim rešitvam.

  Poleg tega se feromoni posodobijo tako lokalno kot tudi globalno. Lokalna posodobitev se izvaja že med grajenjem rešitve. Njen namen je ta, da se zaželjenost povezave (nivo feromonov na povezavi) zniža takoj, ko mravlja uporabi povezavo v svoji rešitvi. Posledično ta povezava postane manj zaželjena pri grajenju rešitve naslednje mravlje, zato raje posegajo po preostalih. S tem dosežemo razgibano raziskovanje po množici vseh rešitev. Pri tej posodobitvi se nivo feromonov na povezavi iz i-tega vozlišča v j-to vozlišče spremenijo po formuli

  $$\tau_{ij}(t + 1) = \sigma\tau_{ij}(t) + (1 - \sigma)\tau_0,$$

  kjer je $\sigma \in [0, 1]$ parameter ohranjanja feromonov (ponavadi vzamemo $0,9$) in τ_0 začetna vrednost feromona, ki je definirana kot $(nL)^{-1}$, kjer je L kakovost rešitve, ki jo dobimo po metodi najbližjega soseda, in n velikost problema (število vozlišč).

  Globalna posodobitev okrepi feromone le za globalno najboljšo rešitev, v redkih primerih le za najboljšo rešitev iteracije. Pri manjših problemih je razlika med uporabo globalno najboljše ali najboljše v iteraciji minimalna, pri večjih problemih pa uporaba globalno najboljše rešitve prinaša boljše rešitve. Globalna posodobitev poskrbi, da se raziskovanje usmeri v področje najboljše rešitve, kar pospeši hitrost konvergence.

- **MMAS–ACS–Hybrid**

  Gre za različico MMAS, ki uporablja psevdonaključno sorazmerno pravilo (6.2), in zelo hitro najde dobre rešitve, vendar se ne približuje k optimalni rešitvi.

- **Approximate Non-deterministic Tree Search (ANTS)** – 1999

  Razred je vpeljal Vittorio Maniezzo v [33]. Glavna lastnost je uporaba spodnje meje LB, ki se določi iz algoritmov prilagojenih za problem, in povprečja ν. Pri posodobitvi feromonov se vsako novo rešitev r primerja z zadnjimi k rešitvami. Takoj, ko je na voljo k rešitev, se izračuna povprečna kvaliteta teh rešitev ν. Če je mravlja našla boljšo rešitev od povprečne kvalitete rešitev, se feromoni na povezavah njene rešitve okrepijo, sicer se zmanjšajo, oziroma

  $$\tau(t+1) - \tau(t) = \tau_0 \left(1 - \frac{r - LB}{\nu - LB}\right),$$

  kar zagotovi preziranje manjših sprememb v zadnjih korakih, hkrati pa preprečuje iskanje le v okolici dobrih rešitev v začetnih korakih algoritma. Nato se odstrani najstarejšo rešitev, doda novo rešitev in ponovno izračuna povprečje.

ACO lahko izboljšamo z uvedbo lokalnih optimizacijskih hevristik. Vsakič, ko zgradimo rešitev, jo poskušamo optimizirati z iterativnimi algoritmi, ki temeljijo na zamenjavi povezav, npr. 2-opt, 3-opt. Ugotovili so, da zamenjava štirih ali več povezav ni več smislena, saj je kakovost izboljšav prenizka glede na potrebno računsko zahtevnost.

Raziskave kažejo, da se pri uporabi lokalnih optimizacij zniža čas izračuna za več kot 97 % in do 13 % izboljša kvaliteta rešitve. Trenutno najuporabnejša razreda sta MMAS oziroma ACS, kjer s količinsko omejitvijo feromonov oziroma lokalno posodobitvijo poskrbimo, da vse mravlje ne sledijo isti poti, in vzdržujemo raziskovanje.

6.4 Konvergenca

Prvi dokazi o konvergenci ACO so se pojavili po letu 2000. Gutjahr je za različico Graph-Based Ant System (GBAS) ob predpostavkah, da obstaja enolična optimalna rešitev, ki ima na povezavah pozitivne vrednosti izkoriščanja η_{ij}, ter da uporablja posodobitev feromonov iz EAS, pokazal, da rešitve vsake iteracije konvergirajo proti optimalni rešitvi z verjetnostjo $1 - \varepsilon$ za dovolj majhen ε.

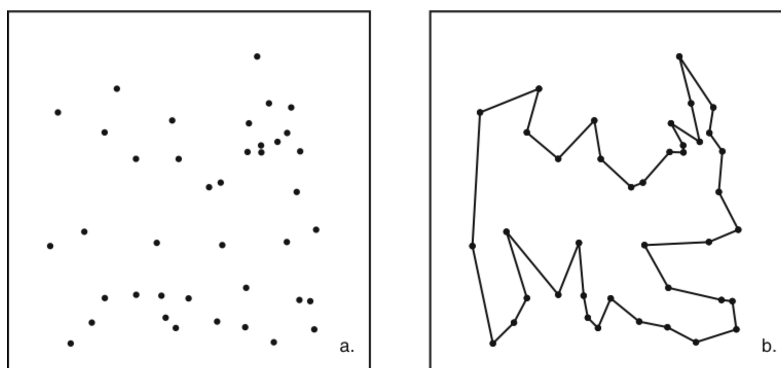
Stützle in Dorigo sta se omejila na MMAS in ACS in leta 2002 pokazala, da je mogoče dokazati, da iz iteracije v iteracijo najboljša rešitev konvergira z verjetnostjo 1 k optimalni rešitvi, in da je mogoče izračunati spodnjo mejo verjetnosti, da je trenutna najboljša rešitev optimalna. Istega leta je Gutjahr pokazal, da rešitve spremenjene različice GBAS konvergirajo k optimalni rešitvi z verjetnostjo točno 1.

Problemov, kjer lahko uporabimo ACO je veliko: usmerjanje vozil, problemi razvrščanja, izdelovanje urnikov, prirejanja, itd. V nadaljevanju bomo spoznali klasični problem za testiranje točnosti algoritmov.

6.5 Problem trgovskega potnika

Problem trgovskega potnika (*angl.* traveling salesman problem, krajše tudi TSP) je eden izmed znamenitejših predstavnikov razreda NP-težkih problemov. Poleg TSP najdemo v tem razredu še probleme barvanja grafov, celoštevilskega linearnega programiranja, celoštevilskega nahrbtnika itd. Doslej še nihče ni uspel najti učinkovitega postopka reševanja, to pomeni algoritma polinomske zahtevnosti poljubnega problema iz tega razreda.

Za začetek spoznajmo formulacijo TSP, ki jo bomo uporabili v nadaljevanju. Naj bo dan poln usmerjen graf, kjer poznamo vrednosti oziroma cene vseh povezav. Naša naloga je najti tak vpet cikel (zaporedje vseh vozlišč, glej sliko 6.4 b.), da bo vsota vrednosti vmesnih povezav najmanjša oziroma cikel najcenejši.



Slika 6.4: a. vozlišča (za lažjo preglednost povezave niso prikazane), b. rešitev – najcenejši cikel

Ideja reševanja naloge je preprosta. Med vsemi možnimi cikli izberemo najcenejšega. Ker pri ciklih ni odlikovanega začetnega vozlišča (vsa vozlišča so lahko začetna), si izberemo vozlišče, iz katerega bomo nadaljevali cikel. Na vsakem koraku, ko imamo določenih že i vozlišč, lahko izberemo naslednje vozlišče iz množice moči $n - i$, kjer je n število vseh vozlišč. Ko cikel vsebuje vsa vozlišča, končamo. Torej je število vseh ciklov, ki jih moramo preveriti, enako

$$(n - 1)!$$

Za občutek nanizajmo v tabelo 6.2 število vseh ciklov za nekaj vozlišč.

Čeprav nam tako reševanje vrne optimalno rešitev, zaradi izredno visoke časovne zahtevnosti potrebne za izračun vseh ciklov ni priporočljivo. Za lažjo predstavo, če predpostavimo, da lahko izračunamo en cikel na nanosekundo (10^{-9} s), bi na 23 vozliščih našli najkrajši cikel šele po približno 35642 letih. Seveda obstajajo še hitrejši algoritmi, ki temeljijo na metodi razveji in

n	$(n-1)!$
3	2
5	24
7	720
11	3628800
13	479001600
17	20922789888000
23	1124000727777607680000
29	304888344611713860501504000000

Tabela 6.2: Število ciklov glede na število vozlišč

omeji in bi potrebovali le nekaj dni, vendar če smo pripravljeni žrtvovati nekaj točnosti na račun hitrosti, lahko pridobimo relativno dobro rešitev že po nekaj minutah. Tedaj lahko uporabimo aproksimativne algoritme in eden od teh temelji na koloniji mravelj.

6.5.1 Reševanje TSP s pomočjo ACO

V prejšnjem poglavju smo spoznali okvirno strukturo za algoritme temelječe na koloniji mravelj, ki jo bomo sedaj priredili za reševanje TSP. Pri tem problemu je zelo pomembno pravilo zadostitve omejitev, saj nočemo eno vozlišče obiskati dvakrat. Omejitev je predstavljena s seznamom vozlišč S, ki jih bodisi še nismo obiskali (seznam dovoljenih premikov) bodisi smo jih že (seznam prepovedanih premikov – tabu seznam). S to omejitvijo mravlje ustvarijo pot, ki gre skozi vsa vozlišča natanko enkrat, še preden se vrnejo v izhodiščno vozlišče. Tedaj dobimo algoritem 4.

Na vsakem koraku iteracije za vsako mravljo naključno izberemo izhodiščno vozlišče, od koder bo zgradila svojo rešitev – najkrajši cikel. Ker izbira naslednjega vozlišča temelji na verjetnosti, se lahko mravlja kje zmoti, zato končna najboljša rešitev ni nujno točna, je pa dovolj dobra.

Naprednejši algoritmi si pomagajo s podatkovno strukturo konstantne velikosti c poimenovano seznam kandidatov, ki za dano vozlišče i vsebuje c preferiranih vozlišč. V primeru ACS enote najprej verjetnostno izbirajo med vozlišči tega seznama. Če seznam ne vsebuje primernega vozlišča, npr. vsa vozlišča so že obiskana ali pa je seznam prazen, se izbere najbližje neobiskano vozlišče.

V tabeli 6.3 so prikazane kvalitete rešitev dveh TSP iz TSPLIB glede na različne metahevristike: ACS, GA – genetski algoritem in SA – simulirano ohlajanje. Vidimo, da se je ACS odrezal najbolje in v obeh primerih našel optimum (minimum).

Algoritem 4 Algoritem kolonije mravelj za reševanje TSP, povzet po [3]

inicializacija
repeat
  for $i := 1 \to m$ (število mravelj) **do**
    {grajenje rešitev}
    i-ti mravlji se določi izhodišče j
    $S = \{1, 2, \ldots, \hat{j}, \ldots, n\}$
    repeat
      izbira naslednjega vozilišča j iz množice S z verjetnostjo (6.1)
      $S = S \setminus j$
    until $S = \emptyset$
  end for
  {posodobitev feromonov}
  for $i, j := 1 \to n$ **do**
    $\tau_{ij} = (1 - \rho)\tau_{ij}$ {izhlapevanje}
  end for
  for $i := 1 \to m$ **do**
    $\tau_{i\pi(i)} = \tau_{i\pi(i)} + \delta$ {π najboljša rešitev, δ vrednost okrepitve}
  end for
until zaključni pogoj
print najboljša rešitev

problem	ACS	GA	SA	točno
Eil50	425	428	443	425
(50 mest)	1830	25000	68512	/
KroA	21282	21761	/	21282
(100 mest)	4820	103000	/	/

Tabela 6.3: Primerjava različnih metahevristik za reševanje TSP (spodnje število je število iteracij)

6.6 Implementacija

V okviru članka je bil implementiran algoritem, ki rešuje TSP na polnem grafu. Implementacija `aco-tsp.py` je napisana v tolmaču Python in je uporabna zgolj za prikaz (demostracijo) delovanja, saj zaradi sijajne počasnosti programskega jezika ni primerna za praktično uporabo.

V programu se začetni polni graf generira naključno in je tako kot feromoni predstavljen z $n \times n$ matriko. Pri grajenju cikla se mravlja za naslednje vozlišče odloča glede na vrednost, ki jo ji vrne generator naključnih števil. Skladno s praviloma izkoriščanja in raziskovanja se povezavam, ki so krajše oziroma cenejše in nosijo več feromonov, dodeli večja verjetnost izbire. Izbrano vozlišče se nato doda v njeno rešitev, ki ji obenem služi tudi kot tabu seznam. Na koncu vsake iteracije se posodobijo feromoni, ki pripadajo petini najboljšim ciklom. Če se v iteraciji najde nova najboljša rešitev, se nivo feromonov še dodatno okrepi.

Parametre algoritma in velikost problema se določi neposredno v kodi programa. Program izpiše najboljšo rešitev v obliki poti (cikel dobimo, ko povežemo prvo in zadnje vozlišče) in pripadajočo dolžino cikla.

```
./aco-tsp.py
path:  [1, 10, 0, 7, 5, 11, 9, 4, 8, 6, 2, 3]
length:  191.55853290486112
```

Za dovolj majhne grafe (do 8 vozlišč) se pred algoritmom požene še eksaktni algoritem, ki vrne podatke v enaki obliki ter služi za primerjavo med točno in približno rešitvijo.

6.7 Posplošitev

Optimizacijo s kolonijo mravelj lahko razširimo, če dodamo mravlje s posebno nalogo, ki jih poimenujemo kraljice. Naloga kraljic je ustvarjanje novih klasičnih ACO kolonij mravelj bodisi novih kraljic. Poleg tega kraljica koordinira celotno kolonijo v smislu uporabe različnih strategij, npr. vsaki mravlji lahko določa faktor izhlapevanja njenega feromona, njena deleža izkoriščanja in raziskovanja ipd., že med samim reševanjem problema. Kraljica lahko za vsako svojo kolonijo odloča, ali lahko mravlje upoštevajo feromone drugih kolonij ali ne, določa število mravelj in iteracij in na koncu svojega življenja vrne najboljšo rešitev, ki jo je našla njena kolonija oz. njene kolonije mravelj. Za lažjo predstavo si predstavljajmo drevo, kjer je v vsakem vozlišču bodisi kraljica bodisi kolonija, in se vsaka rešitev sinov vrača k očetu. Tako dobimo adaptivne metahevristike.

6.8 Zaključek

Optimizacija s kolonijami mravelj je in bo ostala temeljna osnova za načrtovanje učinkovitih algoritmov za kombinatorično optimizacijo. Po več kot dvajsetih letih preučevanja je bilo izdanih že nekaj knjig in več kot 100 člankov, ki predstavljajo tako uspehe v praktični uporabi kot tudi raznovrstne izboljšave ali pa se ukvarjajo s teoretičnim ozadjem.

Poglavje 7

Optimizacija z roji

GREGOR CIGÜT

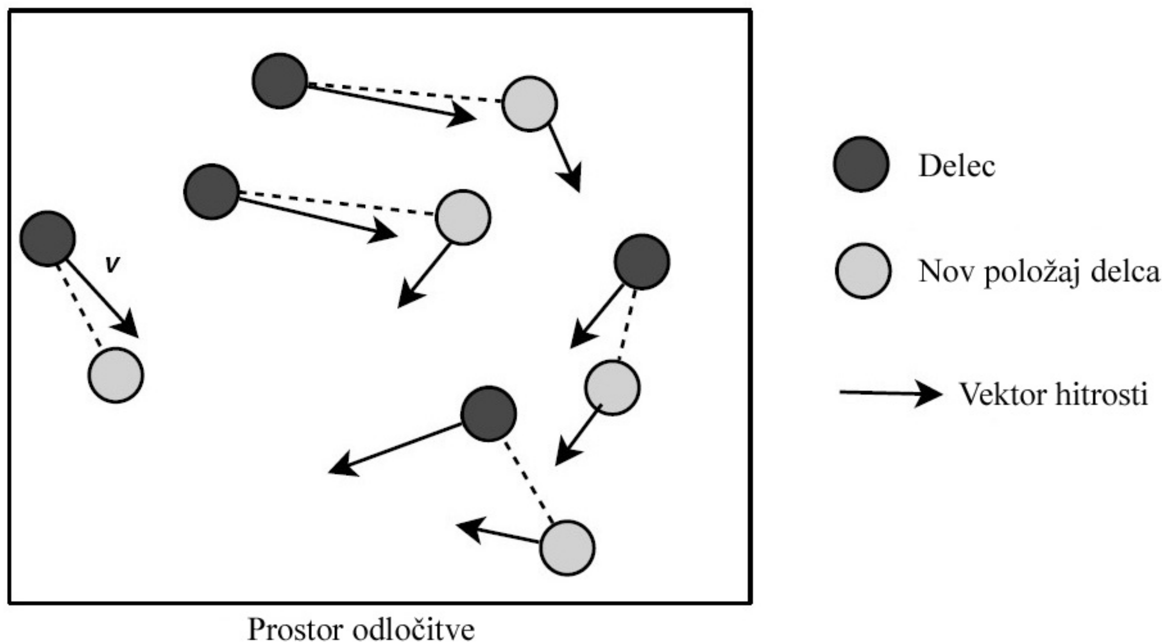
7.1 Uvod

Optimizacija z roji je na slučajni populaciji osnovana metahevristika, idejno motivirana z inteligenco rojev. Gre za imitiranje naravnega družbenega vedenja organizmov, kot je na primer združevanje ptic ali rib v jate, ko iščejo mesto z zadostno količino hrane. Dejstvo je, da se z lokalnimi premiki koordinirano vedenje pri teh jatah pojavi brez nadrejenega vodenja. Optimizacija z roji (angl. *Particle Swarm Optimization*, *PSO*) je bila izvorno skonstruirana za reševanje zveznih problemov. Prvo aplikativno uporabo tega koncepta sta na mednarodni konferenci v avstralskem Perthu leta 1995 predstavila J. Kennedy in R. C. Eberhart.

V osnovnem modelu roj sestoji iz N delcev (angl. *particles*), ki se gibljejo v D-dimenzionalnem prostoru iskanja (angl. *search space*). Vsak delec i je kandidat za rešitev problema (angl. *candidate solution*), predstavljen pa je z vektorjem položaja x_i v prostoru odločitve (angl. *decision space*). Vsak delec ima lasten položaj ter smer in hitrost gibanja. Optimizacija izkorišča sodelovanje med delci. Uspešnost posameznih delcev bo namreč vplivala na vedenje drugih delcev. Dejavnika, skladno s katerima delec i prilagaja svoj položaj x_i v smeri globalnega optimuma, sta sledeča:

- najboljši s strani delca i opažen položaj ($pbest_i$), označen s $p_i = (p_{i1}, \ldots, p_{iD})$,
- najboljši s strani celotnega roja opažen položaj ($gbest$) (ali najboljši s strani dela roja opažen položaj ($lbest$)), označen s $p_g = (p_{g1}, \ldots, p_{gD})$.

Vektor $p_g - x_i$ predstavlja razliko med trenutnim položajem delca i in najboljšim položajem, ki so ga opazili delci v njegovi okolici.



Slika 7.1: Roj s pripadajočimi položaji in vektorji hitrosti. V vsaki iteraciji se delec premakne z obstoječega položaja na nov položaj.

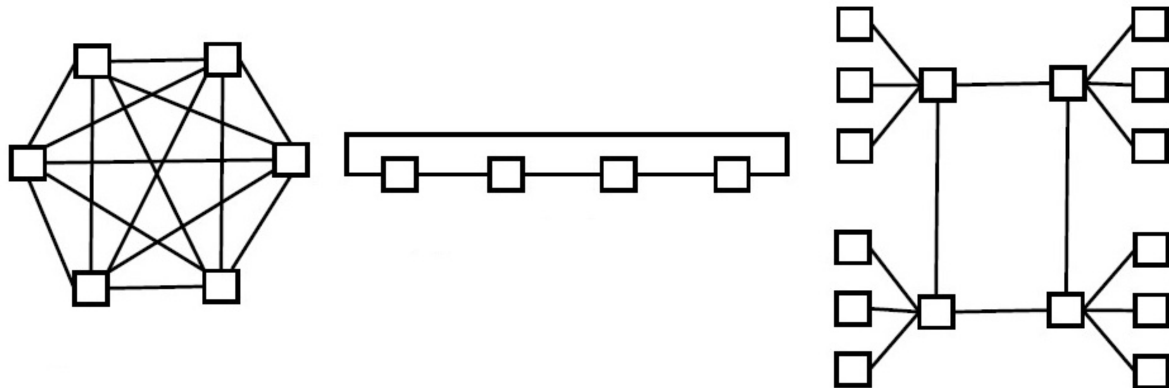
7.2 Okolica delcev

Vsakemu delcu i moramo pripisati okolico (angl. *neighborhood*), to je množico tistih delcev, ki s svojimi odločitvami predstavljajo družbeni vpliv na vedenje delca i, kar pomeni, da sooblikujejo njegovo odločitev o premikanju. Obstaja mnogo različnih načinov za definiranje okolice, tradicionalno pa se uporabljata sledeča dva:

- **metoda gbest**: Okolica delca i je definirana kot celotna populacija delcev.
- **metoda lbest**: Okolica delca i je definirana kot množica z njim neposredno povezanih delcev.

V primeru, ko so delci izolirani, je okolica prazna množica. V primeru, ko lahko roj predstavimo s polnim grafom, okolico vsakega delca predstavlja množica vseh preostalih (metoda *gbest*). Če lahko delce predstavimo na ciklu, okolico vsakega delca predstavljata njegov levi in desni sosed (metoda *lbest*). Grafovska struktura, kjer je nekaj delcev povezanih na ciklu, izključno z enim od teh pa nato še določena podmnožica delcev, ponazarja družbena omrežja, kjer je za vedenje članov neke podmožice značilna medsebojna imitacija, ki vodi v stabilno strukturo homogenih podmnožic. Vsaka podmnožica bo namreč ustvarila okolje, v katerem se bo oblikovala tamkajšnjim članom skupna vedenjska »kultura«. Na podlagi tega razmisleka sledi zaključek, da si bodo posamezniki znotraj neke podmnožice sčasoma vse bolj vedenjsko

podobni, primerjava posameznikov iz različnih podmnožic pa bo sčasoma odražala vse večje vedenjske razlike med njimi.



Slika 7.2: Roj s pripadajočimi okolicami. Poln graf (na sliki levo) predstavlja metodo $gbest$, v kateri okolice delcev predstavlja celotna populacija. Cikel (na sliki na sredini) predstavlja primer metode $lbest$, v kateri okolica posameznega delca sestoji iz dveh drugih delcev. Homogene podskupine (na sliki desno) so razvejene iz oboda kroga, na katerem je nanizanih nekaj delcev.

Skladno z izbrano okolico obstaja vodja (angl. *leader*), tj. delec, ki usmerja iskanje okolice proti boljšim območjem v prostoru odločitve. Delec ponazorimo s tremi vektorji:

- Vektor x_i predstavlja trenutni položaj delca v prostoru iskanja.
- Vektor p_i predstavlja opazovanemu delcu zaenkrat najboljši znan položaj.
- Vektor v_i predstavlja smer, v kateri se bo delec premaknil, če pri tem ne bo oviran.

Na roj lahko pogledamo tudi kot na celični avtomat (lat. *cellular automata*), kjer so posodobitve vrednosti posameznih celic (delcev) izvedene sočasno, po istem postopku, pri tem pa je nova vrednost celice odvisna le od njene prejšnje vrednosti in od celic v njeni okolici. V vsaki iteraciji, tj. ponovitvi posodobitve, bo delec izvedel sledeč postopek:

- **Posodobitev vektorja hitrosti**: Vektor hitrosti, ki ponazarja spremembo smeri in hitrosti gibanja delca, je definiran kot

  $$v_i(t) = v_i(t-1) + \rho_1 C_1 \cdot (p_i - x_i(t-1)) + \rho_2 C_2 \cdot (p_g - x_i(t-1)),$$

  kjer sta ρ_1 in ρ_2 slučajni spremenljivki, ki zavzameta vrednosti na intervalu $[0, 1]$. Konstanti C_1 in C_2 predstavljata učna faktorja, kar pomeni težo, ki jo posamezen delec namenja iskalni uspešnosti sebe oziroma svoje okolice. Parameter C_1 je kognitivni učni faktor,

ki ponazarja privlačnost, ki jo ima za posameznika njegov lastni uspeh. Parameter C_2 je družbeni učni faktor, ki ponazarja privlačnost, ki jo ima za posameznika uspeh njegove okolice. Vektor hitrosti tako skladno z izbranimi zgoraj navedenimi parametri določa smer in hitrost premika, ki ga bo delec opravil v dani iteraciji. Vrednosti vektorja v_i so omejene na vrednosti znotraj intervala $[-V_{\max}, V_{\max}]$, ki je takšen, da preprečuje morebitno eksplozijo sistema zavoljo njegove slučajnosti. Če z iteracijo dobljena vrednost v_i preseže $V_{\max}$ (oz. $-V_{\max}$), jo postavimo na $V_{\max}$ (oz. $-V_{\max}$).

V postopku posodobitve vektorja hitrosti v predpisano iteracijsko formulo predhodnemu vektorju hitrosti $v_i(t-1)$ navadno dodamo inercijsko utež w, ki zajema vpliv predhodnega vektorja hitrosti na novega:

$$v_i(t) = w\cdot v_i(t-1) + \rho_1 C_1\cdot (p_i - x_i(t-1)) + \rho_2 C_2\cdot (p_g - x_i(t-1)).$$

Večje vrednosti inercijske uteži pomenijo, da je vpliv predhodnega vektorja hitrosti na novega večji in obratno. Torej inercijska utež predstavlja odločitveno tehtanje med globalnim raziskovanjem in lokalnim osredotočanjem. Velika inercijska utež spodbuja globalno raziskovanje, kar pomeni, da premika iskanje po celotnem prostoru iskanja, majhna inercijska utež pa spodbuja lokalno osredotočanje, kar pomeni, da poglobi iskanje v okolici trenutnega položaja.

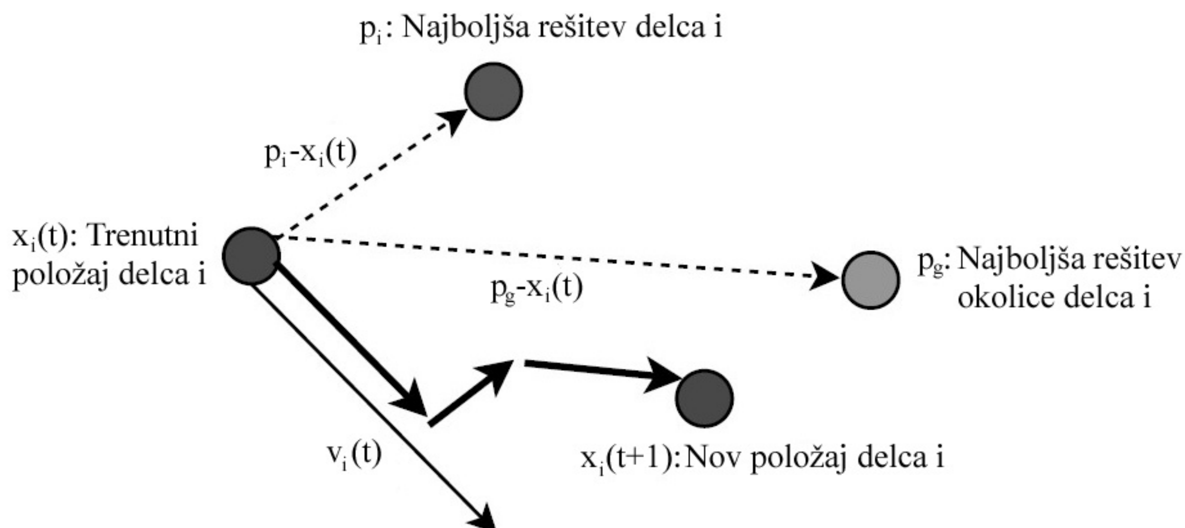
- **Posodobitev položaja**: Vsak delec posodobi svoje koordinate v odločitvenem prostoru skladno s formulo

$$x_i(t) = x_i(t-1) + v_i(t)$$

  in se nato pa se premakne na ta novi položaj.

- **Posodobitev znanja o najboljših položajih**: Vsak delec (lahko potencialno) posodobi najboljšo lokalno rešitev: če je $f(x_i) < f(p_i)$, posodobimo p_i na vrednost $p_i = x_i$. Nadalje se (lahko potencialno) posodobi tudi najboljša globalna rešitev celotnega roja: če je $f(x_i) < f(p_g)$, posodobimo p_g na vrednost $p_g = x_i$.

V vsaki posodobitveni iteraciji bo torej vsak delec spremenil svoj položaj skladno z lastno izkušnjo in doprinosom njegove okolice. Kot v vsakem konceptu inteligence rojev delci tudi v PSO izmenjujejo informacije o izkušnjah, ki so jih pridobili pri izvedenem iskanju. Vedenje celotnega sistema torej izhaja iz interakcij med posameznimi delci. V PSO so izmenjane informacije predstavljene z najboljšo globalno rešitvijo p_g.



Slika 7.3: Premik delca in posodobitev vektorja hitrosti. Delec i je v trenutku t na položaju $x_i(t)$, smer in hitrost premikanja pa podaja vektor $v_i(t)$. Delec je doslej kot najboljšo rešitev odkril položaj p_i, njegova okolica pa položaj p_g. Skladno z inercijsko utežjo delec nekaj poti naredi v smeri vektorja $v_i(t)$, skladno s kognitivnim in družbenim učnim faktorjem pa tudi v smeri položajev p_i in p_g. Rezultat je nov položaj $x_i(t+1)$.

Psevdokoda algoritma za optimizacijo z roji. Opisane korake, segajoče od začetne razporeditve delcev do njihovih premikov in ugotavljanja najboljše rešitve, lahko jedrnato zapišemo v algoritemski psevdokodi.

Slučajna inicializacija položajev celotnega roja.
Ponavljaj (do izpolnitve zaustavitvenega kriterija):
    Izračunaj $f(x_i)$.
    Za (vse delce i):
        Posodobi vektor hitrosti:
        $v_i(t) = v_i(t-1) + \rho_1 C_1 \cdot (p_i - x_i(t-1)) + \rho_2 C_2 \cdot (p_g - x_i(t-1))$.
        Premakni se na nov položaj:
        $x_i(t) = x_i(t-1) + v_i(t)$.
        Če je $f(x_i) < f(p_i)$, posodobi:
            $p_i = x_i$.
        Če je $f(x_i) < f(p_g)$, posodobi:
            $p_g = x_i$.
        Posodobi (x_i, v_i).
    Končaj zanko.
Končaj zanko.

Primer: PSO za optimizacijo zvezne funkcije. Ilustrirajmo, kako z uporabo predstavljenega algoritma za PSO rešimo problem optimizacije (minimizacije) zvezne dvodimenzionalne fukn-

Parameter	Pomen	Tipične vrednosti
N	Velikost populacije	$[20, 60]$
k	Velikost okolice	$[2, \frac{N(N-1)}{2}]$
w	Inercijska utež	$[0.4, 0.9]$
C_1, C_2	Pospešitveni konstanti	≤ 2

Tabela 7.1: Tipične vrednosti parametrov PSO algoritma

cije f. Označimo spodnjo mejo položaja v j–ti dimenziji z l_j, zgornjo pa z u_j. Položaj delca i predstavimo z njegovima koordinatama v dvodimenzionalnem prostoru $x_i = (x_{i1}, x_{i2})$.

Zagon algoritma pomeni, da se najprej skladno z zakonom enakomerne porazdelitve izvede slučajna inicializacija začetnega položaja $x_i(0) = (x_{i1}(0), x_{i2}(0))$ delca i, pri čemer sta abscisna in ordinatna koordinati položaja vsakega delca znotraj predpisanih mej. Nato vsak delec izračuna vrednost kriterijske funkcije f v svojem začetnem položaju. Privzemimo metodo *gbest*, kar pomeni, da za okolico, ki vpliva na vedenje posameznega delca, vzamemo celotno populacijo. Ker poznamo funkcijske vrednosti vseh začetnih položajev, lahko določimo vodjo, to je delec na položaju p_g. Nato vsak delec posodobi vektor hitrosti $v_i(1) = (v_{i1}(1), v_{i2}(1))$, ki ponazarja hitrosti gibanja v horizontalni in vertikalni smeri, skladno s predpisom $v_i(1) = v_i(0) + \rho_1 C_1 \cdot (p_i - x_i(0)) + \rho_2 C_2 \cdot (p_g - x_i(0))$, ki zajema vplive trenutnega vektorja hitrosti $v_i(0)$ ter najboljše lokalne in globalne rešitve. Skladno s formulo $x_i(1) = x_i(0) + v_i(1)$ se delec premakne na nov izračunan položaj $x_i(1) = (x_{i1}(1), x_{i2}(1))$, če pa ta v kateri od koordinat $x_{i1}(1)$ oz. $x_{i2}(1)$ izstopi iz predpisanega območja $[l_i, u_i]$, ga postavimo na ustrezen rob.

Vsak delec nato izračuna novo najboljšo lokalno rešitev p_i. Če za njegov novi položaj $x_i(1)$ velja $f(x_i(1)) < f(p_i)$, potem je nova najboljša lokalna rešitev enaka $p_i = x_i(1)$. Skladno s sodelujočo okolico (za katero smo privzeli celotno populacijo), vsak delec izračuna še novo najboljšo globalno rešitev p_g. Če velja $f(x_i(1)) < f(p_g)$, potem je nova najboljša globalna rešitev enaka $p_g = x_i(1)$.

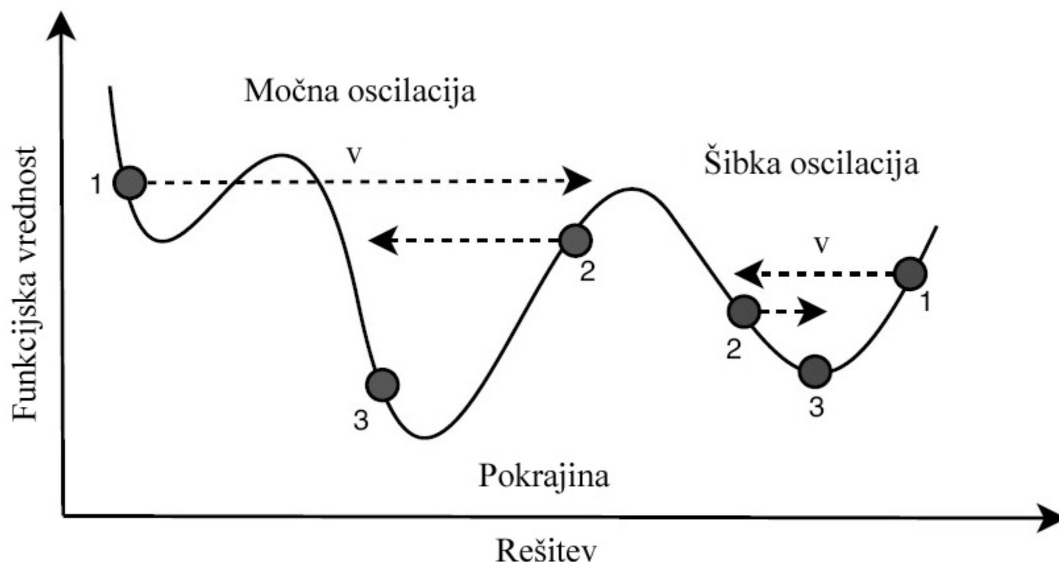
Opisani koraki predstavljajo eno izvedbo iteracije, ki jo ponavljamo, dokler ne zadostimo ustavitvenemu kriteriju. Takrat kot rezultat optimizacije z roji preberemo rešitev p_g.

Kalibracija parametrov PSO algoritma.

Zgornja tabela povzema tipične vrednosti, ki jih zavzemajo parametri PSO algoritma. Število delcev N je obratnosorazmerno s številom priporočenih iteracij algoritma. Večanje populacije izboljša rezultate, a poveča časovno zahtevnost. Potrebno je pretehtati med kvaliteto rezultatov in časom, ki ga bomo za dosego le–teh porabili. Večina PSO implementacij uporablja velikost populacije N na intervalu $[20, 60]$.

Ekstremna primera za številčnost k okolice, s katero delec vzpostavi interakcijo, sta celotna populacija ($k = \frac{N(N-1)}{2}$) in najožja okolica, osnovana na predstavitvi delcev na ciklu ($k = 2$). Seveda pa so za izbiro okolice na voljo številne vmesne poti, denimo razporeditev delcev na torusu ali hiperkocki. Uporaba večjih okolic pospeši konvergenco h globalni rešitvi, toda hkrati poveča verjetnost, da konvergenca ne poteka k resničnemu globalnemu optimumu. Uporaba manjših okolic konvergenco upočasni, lahko pa izboljša kvaliteto rešitve, saj je prostor iskanja le–te bolj diverzificiran. Optimalni odgovor na tehtanje med hitrostjo konvergence in kvaliteto rezultatov je tako potrebno poiskati skladno s pokrajinsko strukturo (angl. *landscape*) optimizacijskega problema.

Parametroma ρ_1 in ρ_2 dodeljeni pospešitveni konstanti C_1 in C_2 predstavljata intenzivnost gibanja proti najboljši lokalni oz. najboljši globalni rešitvi. Kadar sta parametra blizu 0, bodo hitrosti in smeri gibanja spreminjane počasi, zato bodo premiki gladki, kar bo pomenilo raziskovanje v bližini. Kadar pa parametra ρ_1 in ρ_2 zavzameta večje vrednosti, bodo premiki opazneje oscilirali od predhodnega položaja, kar sovpada s širokim preučevanim področjem. Večina implementacij za obe pospešitveni konstanti uporablja vrednost 2.



Slika 7.4: Močne in šibke oscilacije v roju. Za lokalnost oz. globalnost raziskovanja je pomembno, kakšni sta pospešitveni konstanti C_1 in C_2.

Za zgornjo mejo $V_{\max}$ je priporočeno, da jo izberemo v odvisnosti od obsega problema. Lahko pa določanje tega parametra opustimo, če vektorje hitrosti posodabljamo skladno s spodnjo formulo:

$$v_i(t) = \chi \cdot (v_i(t-1) + \rho_1 C_1 \cdot (p_i - x_i(t-1)) + \rho_2 C_2 \cdot (p_g - x_i(t-1))),$$

kjer je χ zožitveni koeficient, določen kot $\chi = \frac{\kappa}{|\frac{1-\frac{\rho}{2}-\sqrt{|\rho^2-4\rho|}}{2}|}$, pri čemer je $\kappa \in [0,1]$ (tipično je $\kappa = 1$) in $\rho = \rho_1 C_1 + \rho_2 C_2$ (tipično bi naj bil $\rho > 4$, torej navzdol omejimo $\rho_1 C_1, \rho_2 C_2 > 2.05$).

Uporaba zožitvenega koeficienta bo zmanjšala amplitudo oscilacije delcev.

Zgornji nabor parametrov je lahko inicializiran tudi dinamično ali adaptivno, kar pomeni, da se tekom iteracij vrednosti parametrov spreminjajo. Dinamika parametrov služi spremenljivemu tehtanju med intenzivnostjo in razpršenostjo raziskovanja. Tipično je začetna vrednost inercijske uteži w postavljena na $w = 0.9$, nato pa sčasoma pada proti $w = 0.4$.

7.3 PSO za diskretne probleme

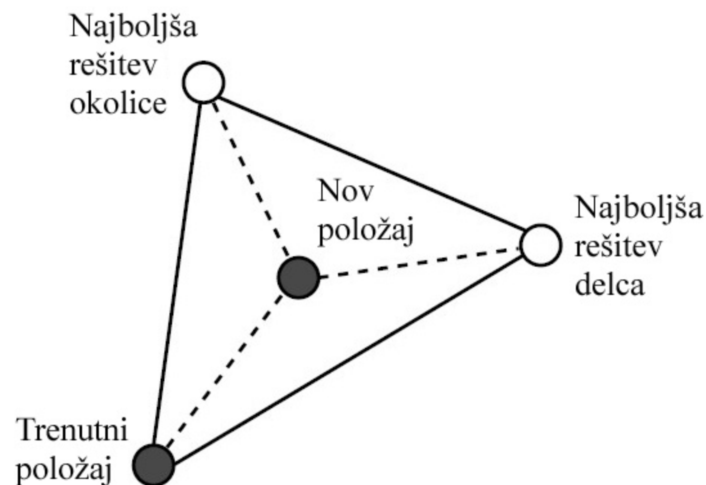
Algoritmi za optimizacijo z roji so tradicionalno namenjeni reševanju zveznih optimizacijskih problemov (zveznost v smislu premikov v prostoru odločitve). Da bi lahko s PSO algoritmi reševali diskretne optimizacijske probleme (diskretnost v smislu premikov v prostoru odločitve), moramo narediti nekaj prlagoditev v modelu. Diskretni model se od zveznega razlikuje v dveh bistvenih lastnostih:

- **Preslikava položajev delcev v diskretne rešitve**: Položaj delca lahko diskretno predstavimo na različne načine, najpogosteje gre za binarno kodo ali permutacijo. Pri binarni preslikavi položaj delca predstavimo z n–dimenzionalnim binarnim vektorjem.

- **Modeli vektorjev hitrosti**: Modeli premikanja lahko temeljijo na realnih vrednostih, so slučajni ali pa premik izberemo z vnaprej podanega seznama možnih premikov. V slučajnih modelih vektorjev hitrosti z binarno preslikavo povežemo smer in hitrost premika z verjetnostjo, da vsaka od binarnih komponent zavzame vrednost 1. V binarnem PSO algoritmu za pretvorbo hitrosti v_i na interval $[0, 1]$ uporabimo sigmoidno funkcijo
$$S(v_{id}) = \frac{1}{1 + e^{-v_{id}}}.$$
Nato generiramo še eno število na intervalu $[0, 1]$ Če je to število manjše od preslikane hitrosti $S(v_{id})$, postavimo vrednost komponente x_{id} v vektorju položaja x_i na 1, sicer pa na 0. Hitrost narašča, kadar sta člena $p_{id} - x_{id}$ oz. $p_{gd} - x_{id}$ pozitivna (kadar sta p_{id} oz. p_{gd} enaka 1), in pada, kadar je obratno. Iz sigmoidne funkcije sledi, da z naraščanjem v_{id} imenovalec v ulomku pada, zato $S(v_{id})$ narašča. S tem, ko se $S(v_{id})$ bliža vrednosti 1, pa je tudi verjetnost, da bo naključno generirano število na intervalu $[0, 1]$ manjše od $S(v_{id})$, večja. To pa je ravno verjetnost, da bomo x_{id} postavili na 1. Večja hitrost v_{id} torej pomeni večjo verjetnost, da bo $x_{id} = 1$.

Primer: Geometrijska optimizacija z roji. Geometrijska PSO (GPSO) predstavlja inovativno diskretizacijo sicer zveznega PSO algoritma, osnovano na geometrijskem okvirju rekombinacijskih operatorjev. Položaj delca i je predstavljen z binarnim vektorjem $x_i = (x_{i1}, \ldots, x_{in})$,

katerega komponente x_{ij} za $j \in 1, \ldots, n$ zavzamejo bodisi vrednost 1 bodisi vrednost 0. Ključna vsebina GPSO algoritmov je koncept premikanja delcev. Namesto vektorja hitrosti vsakemu delcu priredimo ti. »three-parent mask-based crossover«operator (3PMBCX), ki premika delec. 3PMBCX operator vzame tri »starše«, tj. binarne vektorje a, b in c dolžine n, in nato tvori »potomca«, tj. nov binarni vektor dolžine n, tako da za vsako komponento slučajno določi, iz katerega starša bo preslikana v novi vektor. Za delec i so trije starši, ki sodelujejo v 3PMBCX operatorju trenutni položaj x_i, najboljši globalni položaj p_g in najboljši kadarkoli (zgodovinsko) zavzet položaj opazovanega delca p_i.



Slika 7.5: Geometrijsko križanje v GPSO algoritmu. Delec se pomakne v območje znotraj trikotnika, katerega oglišča predstavljajo trenutni položaj delca, najboljša rešitev, ki jo je delec doslej odkril, in najboljša rešitev, ki jo je odkril katerikoli delec.

Uteži w_1, w_2 in w_3 predstavljajo verjetnosti, da bo neka komponenta v novem binarnem vektorju položaja zavzela vrednost, ki jo ima istoležna komponenta v enem od staršev x_i, p_g oz. p_i. Skladno s pomenom staršev uteži predstavljajo inercijsko vrednost trenutnega položaja (w_1), vpliv družbeno najboljšega položaja (w_2) in pomen, ki ga posameznik daje najboljšemu položaju, ki ga je kadarkoli sam zavzel (w_3). Konstrukcija geometrijskega križanja zahteva, da so uteži w_1, w_2 in w_3 nenegativne in da se seštejejo v 1.

Poglavje 8

Razpršeno preiskovanje

PETRA JANŽEKOVIČ

8.1 Uvod

Razpršeno preiskovanje (v nadaljevanju SS algoritem) je prvi opisal F. Glover. SS algoritem je deterministična strategija, ki se jo uspešno uporablja za reševanje nekaterih kombinatoričnih in optimizacijskih problemov. Čeprav so principe te metode definirali že leta 1977, je uporaba razpršenega preiskovanja šele v svojih začetkih.
SS je evolucijska in populacijska metahevristika, ki povezuje rešitve izbrane iz referenčne množice v novo zgrajene rešitve. Metoda se začne z generiranjem začetne populacije, ki zadošča kriterijem raznolikosti in kvalitete. Začetni nabor rešitev, ki je ponavadi sestavljen iz približno desetih rešitev, se potem konstruira z izbiro dobrih reprezentativnih rešitev iz populacije. Z izbranimi rešitvami pridobimo začetne rešitve z metodo izboljšanja, ki temelji na S-metahevristiki. Glede na rezultat takega postopka, je referenčna množica in celo populacija rešitev izboljšana tako, da vključuje tako visoko kvaliteto kot raznolike rešitve. Postopek se nadaljuje dokler ni zadoščeno zaustavitvenim pogojem.
SS algoritem vsebuje različne postopke, ki ustvarijo začetno populacijo, s katero zgradijo in izboljšajo referenčno množico in s katero izboljšajo konstruirane rešitve, itd. SS uporablja izrecno strategije, ki iščejo okrepitev in raznolikost. Vsebuje iskanje komponent iz P-metahevristike in S-metahevristike. Algoritem začne na naboru raznolikih rešitev, ki predstavljajo referenčno množico (glej model algoritma). Ta množica rešitev se razvija s pomočjo novih kombinacij rešitev kot tudi z uporabo lokalnega iskanja (ali drugih S-metahevristik).

8.2 Model in sestava algoritma

Model algoritma SS:

Algoritem 5 Scatter Search

```
{Začetna faza}
Izberi populacijo z uporabo diverzifikacijske metode;
Uporabi metodo za izboljšanje populacije;
Posodobitev referenčne množice;
{SS iteracija}
while kriteriji za ustavitev do
    Metoda na podmnožicah;
    while kriteriji za ustavitev 1 do
        Kombinacijska metoda;
        Metoda za izboljšanje;
    end while
    Posodobitev referenčne množice;
end while
```

Algoritem razpršenega preiskovanja je v splošnem sestavljen iz naslednjih petih metod (glej sliko 1):

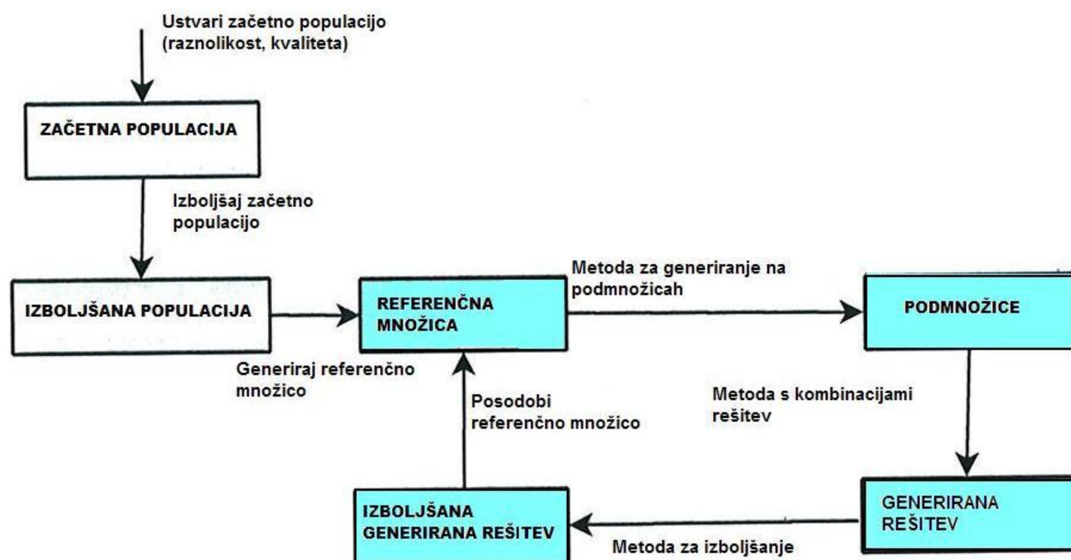
- *Diverzifikacijska splošna metoda:* Ta metoda generira množico raznolikih začetnih rešitev. V splošnem so vgrajeni požrešni postopki, da ustvarijo raznolikost iskanja, tako da izberejo visoko kvalitetne rešitve.

- *Metoda izboljšanja:* Ta metoda ustvari iz poskusne rešitve eno ali več izboljšanih začetnih rešitev z uporabo nekaterih S-metahevristik. V splošnem se uporabi algoritem za lokalno iskanje, ki generira lokalni optimum.

- *Posodobitev referenčne množice:* V tej iskalni komponenti se konstruira in ohranja referenčna množica. Namen te metode je ohraniti raznolikost kot tudi ohraniti visoko kvaliteto rešitev. Na primer, lahko izberemo rešitve z najboljšo uporabnostno funkcijo iz množice RefSet1 in nato dodamo rešitve z najboljšo raznolikostjo iz množice RefSet2 (RefSet = RefSet1 + RefSet2).

- *Posplošena metoda na podmnožicah:* Ta metoda deluje na referenčni množici RefSet, da ustvari podmnožico rešitev kot bazo za ustvarjanje kombiniranih rešitev. Ta metoda ponavadi izbere vse podmnožice določene velikosti r (najpogosteje je $r = 2$). Ta postopek je podoben kot izbirni mehanizem pri evolucijskem algoritmu (EA). Kakorkoli, pri SS gre za determinističen operator, medtem ko imamo pri EA v splošnem opravka s stohastičnimi operatorji. Še več, velikost referenčne množice pri SS algoritmu je veliko manjša kot velikost populacije pri EA algoritmu. To je razlog zakaj lahko uporabimo več mehanizmov številčnejše selekcije.

- *Metoda kombiniranih rešitev:* Na novo združimo dano podmnožico rešitev pridobljeno s posplošeno metodo na podmnožicah. V splošnem uporabimo utežene strukturirane kombinacije preko linearnih kombinacij in splošnih zaokroževanj v diskretne spremenljivke. Na ta operator lahko gledamo tudi kot na posplošitev križnega operatorja v EA, kjer združimo več kot dva osebka.



Slika 8.1: Iskalne komponente SS algoritma.

Tri komponente iskanja (diverzifikacijska splošna metoda, metoda izboljšanja in metoda kombinacij) so specifične za problem, medtem ko sta drugi dve iskalni komponenti (metoda posodobitve referenčne množice, posplošena metoda na podmnožicah) splošni. Te metode se prepletajo kot kaže slika in algoritem.

8.2.1 Primer: SS algoritem za problem p-mediane.

Problem p-mediane je problem razporeditve p-stavb s širokim razredom logističnih in razporeditvenih aplikacij kot so lokacija skladišč in javnih zgradb.
Podana je množica $L = \{v_1, v_2, \ldots, v_m\}$, ki predstavlja m možnih lokacij za p zgradb. Podana je množica $U = \{u_1, u_2, \ldots, u_n\}$ n strank. Podana je matrika velikosti $n \times m$, $D = (d_{ij})_{n \times m}$, ki predstavlja razdaljo (ali dane stroške), ki zadovolji zahteve strank, ki se nahajajo na u_i od zgradb, ki se nahajajo na v_j. Problem p-mediane vsebuje določitev p zgradb na lokacije L in razporeditev vseh strank u_i v zgradbe, tako da se minimizira objektivna funkcija, ki predstavlja skupno razdaljo med strankami in njim pripadajočo najbližjo zgradbo:

$$f(X) = \min \sum \min_{v_j \in S} d_{u_i v_j},$$

kjer S predstavlja množico zgradb ($S \subseteq L$ in $|S| = p$). Poglejmo SS algoritem za rešitev tega problema. Brez izgube splošnosti lahko predpostavimo, da je $L = U$ in $m = n$. Problem vsebuje izbiro množice S s p lokacijami iz U, ki jih priredimo danim zgradbam. V nadaljevanju so predstavljene nekatere rešitve drugačnih problemov, kjer uporabljamo algoritem SS.

- *Predstavitev:* Rešitev problema p-mediane lahko predstavimo kot diskretni vektor $v = \{v_1, v_2, \ldots, v_n\}$ velikosti n, ki predstavlja srečanja vseh strank U. Spremenljivka v_i definira zgradbo v S za $i \leq p$ in je točka brez zgradbe za $i > p$.

- *Okolica:* Operator premika je lahko operator izmenjave. Okolica rešitve je definirana z množico izmenjav $I = (i, j) : 1 \leq i \leq p, p < j \leq n$. Za zamenjavo $(i, j) \in I$ je oklica rešitve x enaka $x - v_i + v_j$.

- *Začetna populacija:* Množica točk L je razdeljena na disjunktne podmnožice. Na vsaki podmnožici uporabimo požrešni algoritem takole: začetno točko u iz množice L_i izberemo naključno, nato se izvede $p - 1$ iteracij za izbiro najbolj oddaljene točke od teh, ki so že izbrane. Ta požrešni postopek se uporabi na različnih začetnih točkah, da ohranimo različne rešitve za vsako podmnožico L_i. Na vsaki dobljeni rešitvi se nato izvede še metoda izboljšanja. Da ohranimo raznolikost, definiramo funkcijo, ki predstavlja razdaljo med dvema rešitvama. Razdaljo definiramo takole:

$$\text{dist}(X, Y) = f_Y(X) + f_X(Y),$$

  kjer je $f_Y(X) = \sum_{v \in Y} \min_{u \in X} d_{uv}$. S tem je velikost populacije Pop imenovana Popsize, parameter α pa določa razmerje rešitev, izbranih z objektivno funkcijo. Preostalih $(1-\alpha)$ Popsize rešitev je izbranih z uporabo diverzifikacijske dosegajoče funkcije h:

$$h(X) = f(X) - \beta \text{dist}(X, \text{Pop}),$$

  kjer je $\text{dist}(x, \text{Pop}) = \min_{Y \in \text{Pop}} \text{dist}(X, Y)$ in β konstanten parameter.

- *Posplošitev referenčne množice:* RefSet1 rešitve so izbrane glede na kvaliteto objektivne funkcije. RefSet2 rešitve so izbrane s pomočjo kriterija za raznolikost (RefSet = RefSet1 + RefSet2). Ko je enkrat v referenčni množici vsebovana tudi najboljša rešitev iz RefSet1, je najbolj oddaljena rešitev, ki je v referenčni množici iterativno dodana. Ta postopek raznolikosti ponovimo tolikokrat, kot je velikost množice RefSet2.

- *Posplošena metoda na podmnožicah:* Izberemo vse podmnožice velikosti $r = 2$. Da se izognemo podvajanju, hranimo informacije o že izbranih kombinacijah.

- *Metoda kombinacij rešitev:* Uporabimo lahko križni operator, ki na novo združi dve rešitvi. Najprej izberemo točko X, ki je skupna obema rešitvama. Izvede se konstrukcija

množice

$$L(u) = \{v \in L : d_{un} \leq \beta d_{\max}\},$$

kjer je $u \in L \backslash X$ in $d_{\max} = \max_{u,v \in L} d_{uv}$. Potem izberemo točko u^*, tako da je $d_{xu^*} = \max_{u \in L} d_{Xu}$ in naključno točko $v \in L(u^*)$, ki je dodana v X. Ta postopek se ponavlja dokler ni $|X| = p$.

- *Metoda izboljšanja:* Uporabi se neposredno lokalno iskanje s predstavitvijo in okolico kot sta predstavljeni zgoraj. Metodo izboljšanja lahko razširimo na katerokoli S-metahevristiko kot so naprimer tabu iskanje ali namišljena priključitev. Ta postopek nam vrne množico ImpSolSet izboljšanih rešitev.
- *Posodobitev referenčne množice:* Uporabi se postopek generiranja referenčne množice na množici sestavljeni iz unije množic RefSet in ImpSolSet.

8.3 Povezovanje poti (Path Relinking)

Algoritem povezovanja poti (v nadaljevanju PR algoritem) je bil predstavljen s strani Gloverja v delu o SS algoritmu. PR algoritem dovoljuje odkrivanje poti, ki povezujejo elitne rešitve najdene z algoritmom SS. To strategijo se lahko posploši in uporabi na katerikoli metodi, ki generira bazen dobrih rešitev, kot so npr. evolucijski algoritmi, požrešni prilagodljivi algoritmi ($GRASP$), algoritem kolonije mravelj in iterativni algoritmi lokalnega iskanja.
Glavna ideja PR je generirati in raziskati trajekorijo v prostoru rešitev, ki povezuje začetno rešitev s z iskano rešitvijo t. Ideja je ta, da interpretiramo linearne kombinacije točk v Evklidskem prostoru kot poti med in nad rešitvami v nekem sosednjem prostoru. Pot med dvema rešitvama v iskalnem prostoru (sosednji prostor/prostor okolice) bo v splošnem podala rešitve, ki imajo skupne lastnosti z vhodnimi rešitvami. Zaporedje sosednjih rešitev v odločitvenem prostoru je generirano iz začetnih rešitev v končne rešitve. Postopek nam vrne najboljšo najdeno rešitev iz zaporedja. Na vsaki iteraciji izberemo najboljšo možnost glede na objektivno funkcijo in glede na zmanjšanje razdalje d med dvema rešitvama. To se ponavlja dokler ni razdalja enaka 0. Algoritem vrne najboljšo rešitev najdeno na trajektoriji.

Osnovni model PR algoritma:
Glavne težave na katere lahko naletimo pri PR, so naslednje:

- *Izbira poti:* Vprašati se moramo, na kaj vse moramo biti pozorni pri generiranju poti. Vodilna hevristika za generiranje poti mora znati minimizirati razdaljo med vodilno rešitvijo. Zato mora biti razdalja d definirana v iskalnem prostoru, ki je povezan s problemom. Tukaj moramo upoštevati računsko kompleksnost tega postopka. Medtem ko je za nekatere probleme zahtevnost polinomska, je za nekatere druge optimizacijske probleme lahko veliko bolj kompleksna. Pravtako lahko obravnavamo in vzamemo več poti vzporedno.

Algoritem 6 Path Relinking

Require: Začetna rešitev s in končna rešitev t.
  $x = s$;
  while dist$(x, t) \neq 0$ **do**
    Najdi najboljšo potezo, ki zmanjša razdaljo dist$(x \oplus m, t)$;
    $x = x \oplus m$; /*Dodaj potezo m k rešitvi x */;
  end while
Ensure: Najboljša rešitev najdena na trajektoriji med s in t.

Naj bosta s in t začetna in vodilna rešitev. Generirajmo sedaj zaporedje $(s, x_1, x_2, \ldots, t)$. Na primer, množico kandidatov $S(x_i)$ za x_{i+1} lahko generiramo s pomočjo x_i tako, da izberemo v okolici rešitve x_i tiste z minimalno razdaljo do vodilne rešitve x':

$$S(x_i) = x_{i+1}/x_{1+1} \in N(x_i) \quad \text{in} \quad \min(d(x_{i+1}, t)).$$

Kardinalnost te množice je lahko večja kot ena. Zato moramo uporabiti še dodatne izbirne kriterije, da določimo rešitev iz množice kandidatov $S(x_i)$. Mehanizem izbire je lahko osnovan na naslednjih različnih kriterijih:

– *Kvaliteta:* Izbira rešitve glede na objektivno funkcijo problema. Izbira najboljše ali najslabše rešitve bo imela različen vpliv na iskanje (intenzivnost, raznolikost).

– *Zgodovina iskanja:* Spomin iskanja lahko uporabimo v fazi izbire. Na primer, uporaba pomnilnika v tabu iskanju lahko izloči nekatere rešitve.

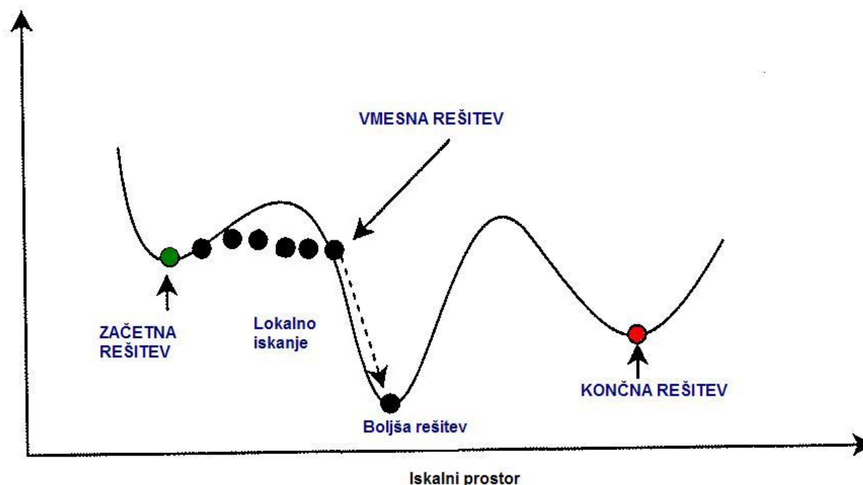
Če pravila izbire ne prepovedujejo ciklov v generiranju poti, potem se tabu seznam ohranja. Izbiro poti lahko vodimo tudi s pomočjo množice $\Delta(s, t)$, ki predstavlja množico različnih komponent (na primer, povezave v problemih iskanja poti) začetnih rešitev s in željenih rešitev t.

- *Vmesni operatorji:* Tukaj se vprašamo, kako se spopasti z uporabo operatorjev na vsakem koraku pri konstrukciji poti. Pretehtati moramo nekatere izbrane rešitve na poti. Na primer, lahko uporabimo S-metavristiko na vsaki vmesni rešitvi. Slika 2 prikazuje primer, kjer uporabimo postopek lokalnega iskanja na vmesnih rešitvah, da izboljšamo kvaliteto iskanja.

Obstaja več različnih alternativ, kako izbrati par začetnih in željenih rešitev (s, t) (glej sliko 3):

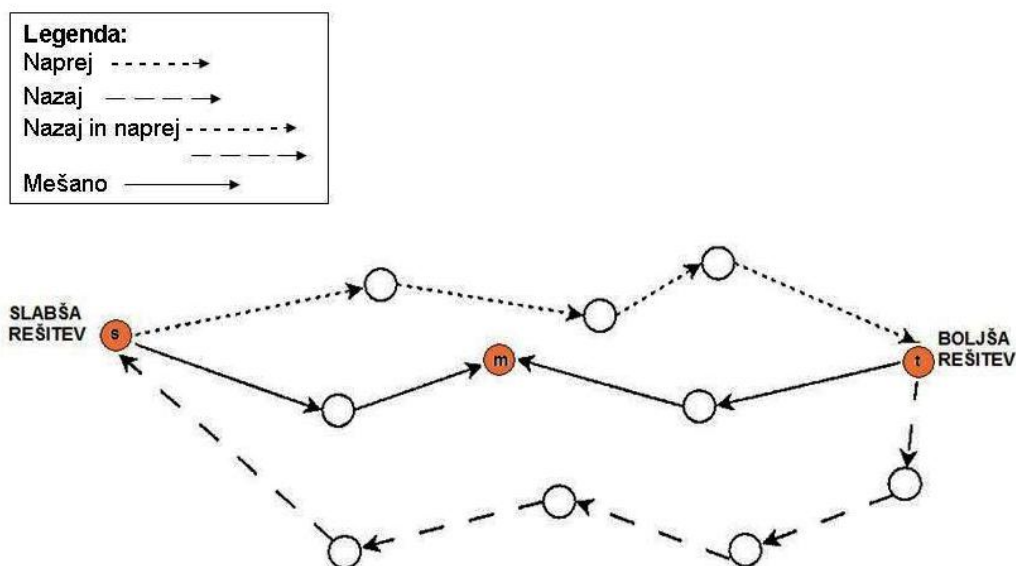
- *Strategija naprej:* Najslabšo rešitev med s in t uporabimo kot začetno rešitev.

- *Strategija nazaj:* Vloga vsake rešitve se zamenja. Boljšo rešitev med s in t uporabimo za začetno rešitev. Ker bolj raziščemo okolico začetne rešitve kot okolico željene rešitve, je strategija nazaj v splošnem boljša kot strategija naprej.



Slika 8.2: Uporaba postopka lokalnega iskanja na vmesni rešitvi v PR algoritmu. V strukturi 'velikih dolin' je lahko vmesna rešitev med dvema lokalnima optimuma premišljeno vodena do tretje doline privlačnosti. Ta dolina lahko vsebuje boljši lokalni optimum.

- *Strategija nazaj in naprej:* Dve poti konstruiramo paralelno, tako da uporabimo s kot začetno in končno rešitev. Ta strategija doda veliko k časovni zahtevnosti, kar ni nujno uravnoteženo s kvaliteto dobljene rešitve.
- *Mešano povezovanje:* Podobno kot v strategiji nazaj in naprej sočasno konstruiramo dve poti iz s in t, vendar je vodilna rešitev na isti razdalji od s in t. Ta vzporedna strategija lahko zmanjša časovno zahtevnost.



Slika 8.3: Različne PR strategije glede na začetno in trenutno rešitev.

8.3.1 Primer PR: Optimalno povezovanje v binarnem iskalnem prostoru.

V smislu binarnih optimizacijskih problemov lahko pot najdemo s pomočjo iskanja najboljše poteze na vsakem koraku posebaj (flip operator). To nas privede od trenutne rešitve korak bliže končni rešitvi. Razdaljo, ki jo uporabljamo je Hammingova razdalja in jo je preprosto izračunati. Proces izbire poti lahko vidimo kot omejeno lokalno iskanje, kjer je množica možnih rešitev omejena in je strategija izbire sosedov posplošena. Za nebinarne predstavitve je lahko izračun razdalje med dvema rešitvama precej težji.

Poglavje 9

Umetni Imunski Sistemi

Branka Janežič

9.1 Naravni imunski sistem

Organizmi imajo različne mehanizme za obrambo pred škodljivimi tujki iz okolja. Mikroorganizmi, kot so bakterije, virusi, glivice in mnoge praživali, lahko bolezen povzročijo posredno ali neposredno. Neposredno delujejo tako, da poškodujejo in uničijo telesne celice, posredno pa škodujejo s tvorjenjem strupov, ki zavirajo delovanje telesnih celic. Obrambni sistem se skuša upreti škodljivim učinkom mikroorganizmov ter se znebiti tudi lastnih celic, ki so se na kakšenkoli način spremenile in jih ne prepozna več kot lastne. Sposobnost telesa, da se brani pred tujimi snovmi imenujemo **odpornost**.

Obrambo proti tujkom delimo v dve skupini: nespecifična odpornost in specifična odpornost. Nespecifična odpornost deluje proti različnim tujkom na splošno, medtem ko specifična deluje le proti določenemu tujku. V mehanizme nespecifične odpornosti so vključeni tako pasivni kot aktivni načini bojevanja proti tujkom. Glavni pasivni način obrambe so pregrade, ki razmejujo notranjost telesa od zunanjosti (koža, sluznične membrane), aktivni pa je na primer povečanje telesne temperature.

Ker je nespecifična obramba zelo splošna in se ne prilagaja vsakemu napadalcu posebej, se velikokrat zgodi, da ni dovolj učinkovita. Takrat pride na vrsto specifična obramba, pri kateri je odziv obrambnih mehanizmov prilagodljiv, saj priredijo orožje za vsakega vsiljivca posebej. Poleg tega ima ta obrambni mehanizem elemente, ki si zapomnijo strategijo in vrsto vsakega napadalca, tako da je obramba pozneje zelo hitra in učinkovita. Tak način obrambe omogoča telesu, da postane odporno proti specifičnemu mikrobu. Z izrazom imunost torej označujemo vse tiste lastnosti telesa, ki mu omogočajo, da postane odporno proti točno določenemu mikrobu oz. bolezni.

Imunost proti različnim tujkom je lahko naravna oz. prirojena, ali pridobljena. Naravno

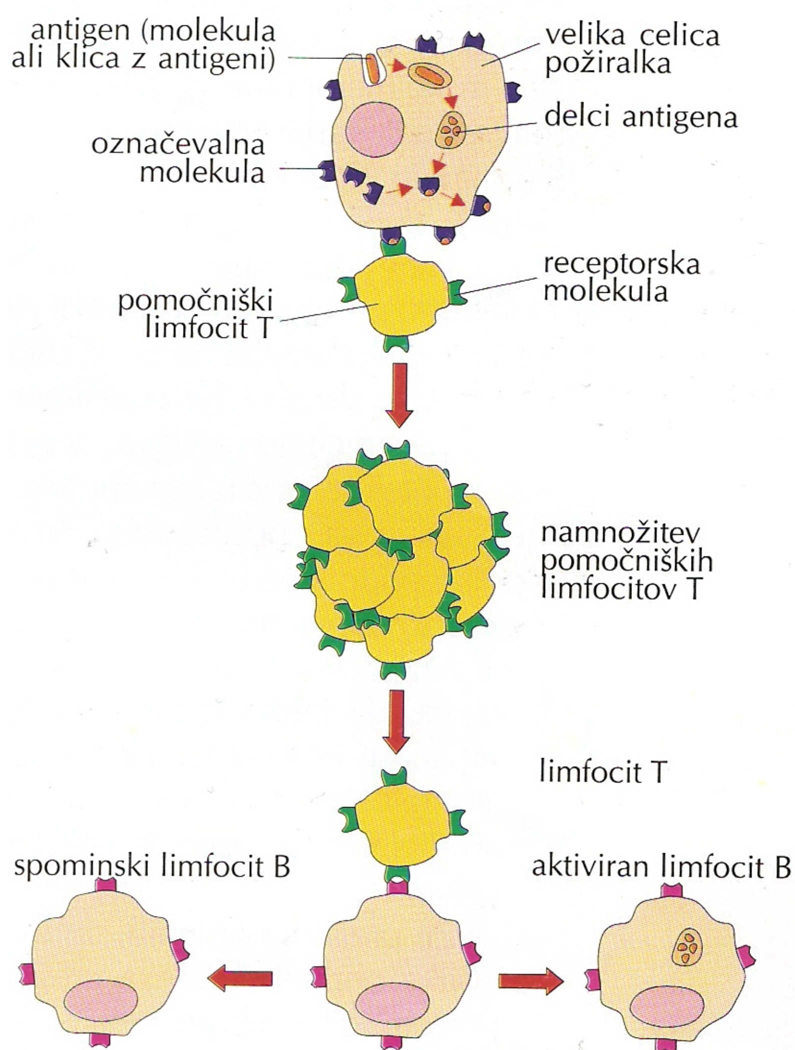
imunost jo imamo prirojeno, pri pridobljeni imunosti pa mora imunski sistem priti najprej v stik s tujkom, nato pa prilagodi obrambo, specifično proti temu tujku. Tega pa mora telo prepoznati kot tujega in ga razločevati od lastnih delov telesa. Zato imajo vse telesne celice oziroma deli celic na svojih površinah oznake, po katerih jih telo prepozna kot svoje. Tudi tujki imajo take oznake, vendar so drugačne kot lastne, zato jih telesna obramba lahko loči. Te onake so posebne kemične snovi, ki so za vsako vrsto celic oziroma za tujke drugačne. Te snovi imenujemo **antigene**. Poznamo torej lastne in tuje antigene.

Antigeni tujkov so značilno drugačni od telesnih antigenov. Navadno se nahajajo na površini mikrobov, kjer so del njihove zunanje ovojnice, lahko pa se znajdejo tudi prosto v tkivnih tekočinah. Imunost specifičnega sistema torej temelji na prepoznavanju tujih antigenov ter na dveh različnih načinih njihovega uničevanja v telesu. Zato poznamo dva mehanizma specifične imunske obrambe. V prvega so vključene celice, ki spodbujajo nastajanje posebnih snovi, imenovanih **protitelesa**. Te snovi delujejo uničujoče na vsak antigen, proti kateremu so razvite. V drug mehanizem specifične obrambe pa so vključene lastne telesne celice, ki so jih napadli virusi, bakterije ali druge tuje celice in so se zaradi tega spremenile.

V specifičnem imunskem sistemu izzovejo nastajanje protiteles antigeni, zato lahko rečemo, da je antigen pravzaprav vsaka molekula, ki spodbudi tvorbo protiteles. Protitelesa se vežejo na specifične anigene po načelu ključa in ključavnice. Napadalce, oziroma celice, ki imajo na površini antigene, protitelesa tudi uničijo. Specifičnih protiteles ponavadi v telesnih tekočinah ni veliko, zato je treba takoj, ko tujki množično vdrejo v telo, njihovo proizvodnjo pospešiti.

Antigen je potrebno najprej prepoznati. To nalogo opravijo velike celice požiralke (makrofagi). Te ne samo, da antigen prepoznajo, temveč ga tudi zaobjamejo in požrejo. Od tuje celice ostanejo samo delci antigenov, ki se vgradijo v plazemsko membrano makrofagov in štrlijo navzven, tako da jih lahko prepoznajo receptorske molekule na pomočniškem limfocitu T. Ko te pridejo v stik z delci antigenov se aktivirajo in namnožijo. Po aktivaciji limfocitov T se tudi v njihove plazemske membrane vgradijo posebne receptorske molekule. S temi molekulami prepoznajo tretjo vrsto belih krvnih celic B, in se nanje vežejo. Takoj po vezavi na celice B se aktivirajo tudi te in se začnejo deliti v dve vrsti hčerinskih celic. Iz prvih se tvorijo specifična protitelesa (2000 do 20000 protiteles na sekundo), ki se lotijo preostalih tujkov z enakimi antigeni, druge pa postanejo spominski limfociti B (glej sliko 9.1). Protitelesa se s telesnimi tekočinami prenašajo po telesu in če naletijo na antigen, se nanj vežejo in ga onesposobijo.

Druga oblika boja imunskega sistema proti tujkom je obramba preko prepoznavanja lastnih telesnih celic, ki so se spremenile. Na plazemski membrani celic, ki so se okužile z virusom, se namreč pojavijo virusni antigeni. Te prepoznajo ubijalske celice T in plazemsko membrano napadejo. Ubijalske celice T začnejo pod vplivom antigenov izločati beljakovine, ki luknjajo plazemsko membrano napadenih celic. Membrana celic začne puščati in celice odmrejo, nazadnje pa jih pospravijo še celice požiralke.



Slika 9.1: Delovanje B in T limfocitov v naravnem imunskem sistemu.
Vir: [15]

9.2 Umetni imunski sistemi

Pri razvoju novih optimizacijskih algoritmov, se čedalje bolj posnema naravne imunske sisteme, saj so ti robustni, kompleksi in prilagodljivi. Področje, ki uporablja načela imunskega sistema za optimizacijo in probleme učenja strojev, je znano kot **umetni imunski sistemi (artificial immune systems - AIS)**. AIS so torej prilagodljivi sistemi, osnovani na teoretični imunologiji, v glavnem pa se uporabljajo pri problemih rudarjenja podatkov.

AIS algoritme lahko razdelimo v dva razreda: populacijske AIS algoritme, ki izhajajo iz teorije selekcije klonov in negativne selekcije ter omrežne AIS algoritme, ki izhajajo iz toerije omrežij za imunske sisteme, na primer teorije nevarnosti.

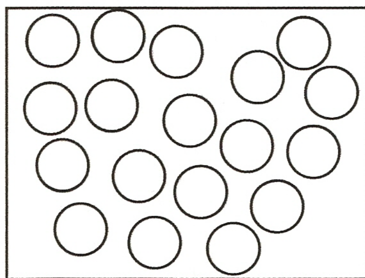
Pri oblikovanju AIS algoritmov moramo najprej določiti način predstavitve komponent (antigenov, protiteles, celic, molekul) sistema, oziroma način kodiranja. Najpogosteje se uporablja binarno kodiranje ali zapis z n-terico realnih števil. Potrebno je tudi ovrednotiti povezave med komponentami sistema (med sabo in z okoljem). Mera ujemanja je lahko predstavljena kot razdalja, ki bo odvisna od načina kodiranja; za binarno predstavitev uporabimo Hammingovo razdaljo (število neenakih komponent v dveh nizih enake dolžine), pri predstavitvi z realnimi števili pa uporabimo Evklidsko ali pa Manhattan razdaljo (razdalja med dvema točkama, če lahko potujemo samo po mreži).

9.2.1 Populacijski AIS algoritmi

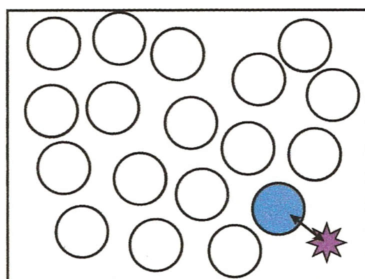
Teorija selekcije klonov

Selekcija klonov se uporablja pri modeliranju reakcije imunskega sistema na infekcijo. Teorijo selekcije klonov je predstavil M. Burnet leta 1959, ki je za svoje delo na AIS prejel tudi Nobelovo nagrado. Njegova teorija je osnovana na konceptu kloniranja in ujemanja med komponentami sistema. B in T limfociti so izbrani za uničenje specifičnih antigenov, ki napadajo telo (Slika 9.2). Ko je telo izpostavljeno tujemu antigenu, se tiste B celice, ki se najbolje vežejo z antigenom (Slika 9.3) pričnejo razmoževati s kloniranjem. B celice so programirane, da klonirajo specifične tipe protiteles, ki se bodo vezala s specifičnim antigenom (Slika 9.4). Vez med antigenom in protitelesom je odvisna od tega, kako dobro se receptorska molekula na protitelesu ujema z označevalno molekulo na antigenu. Boljše kot je to ujemanje, močnejša je vez. To lastnost poimenujemo **afiniteta**. Nekatere klonirane celice se diferencirajo v aktivirane limfocite B (antibody secretors), ostale klonirane celice pa postanejo spominske celice. Velik delež kloniranih celic nato še mutira, kar spodbuja genetsko raznolikost (Slika 9.5). Na koncu se izvede še selekcija, ki zagotovi preživetje celic z višjo afiniteto (Slika 9.6).

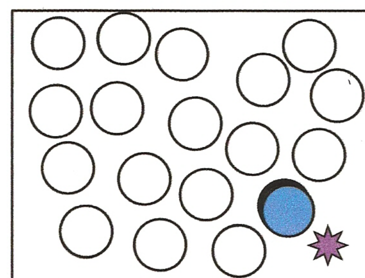
Eden glavnih algoritmov, ki so osnovani na teoriji selekcije klonov je algoritem CLONALG. Protitelo predstavlja rešitev, antigen pa predstavlja vrednost funkcije, ki jo optimiziramo. Proti-



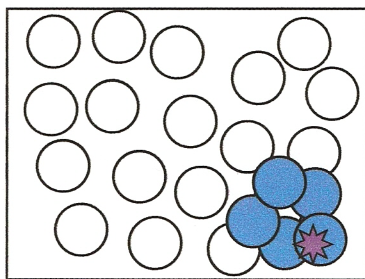
Slika 9.2: **1. korak.** Imamo naključno zgenerirano začetno populacijo protiteles. Vsak krogec predstavlja eno protitelesce. Vir: [16]



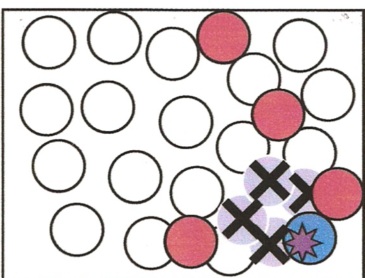
Slika 9.3: **2. korak.** Antigen (obarvan vijolično) vstopi v telo. Z uporabo dane metrike mu določimo najbližji klon (obarvan modro). Vir: [16]



Slika 9.4: **3. korak.** Kloniramo najbližje protitelesce sorazmerno z ujemanjem med protitelesom in antigenom. Večje ujemanje pomeni večje število klonov. Vir: [16]



Slika 9.5: **4. korak.** Mutiramo klone s stopnjo inverzno ujemanju z atnigenom. Večje kot je ujemanje, manjše je število mutiranih klonov. Vir: [16]



Slika 9.6: **5. korak.** Najdemo klon, ki se najbolje ujema z antigenom. Ostale klone izbrišemo in vsakega nadomestimo z novim, naključno zgeneriranim protitelesom. Vir: [16]

telesa in antigeni so večinoma predstavljeni s standardnim kodiranjem, torej nizi števil, kjer so lahko števila binarna, cela ali realna. Ujemanje je odvisno od uporabljene metrike (Evklidska, Manhattan, Hamming). Afiniteta med protitelesom in antigenom je torej povezana z razdaljo; način predstavitve bo delno določal metriko, ki jo bomo uporabili za izračun stopnje interakcije med protitelesom in antigenom.

Algoritem CLONALG. Najprej naključno zgeneriramo populacijo N protiteles. Vsako protitelo predstavlja rešitev za naš optimizacijski problem. Potem izberemo n protiteles glede na selekcijsko shemo. Izbrana protitelesa kloniramo in mutiramo ter tako konstruiramo novo populacijo protiteles. n protiteles generira N_c klonov, proporcionalno njihovim afinitetam (ta je v CLONALG algoritmu za optimizacijski problem definirana kot namenska funkcija). Višja kot je afiniteta, večje je število klonov. Število kloniranih protiteles N_c se izračuna z naslednjo formulo:

$$N_c = \sum_{i=1}^{n} \lfloor \frac{\beta N}{i} \rfloor,$$

kjer je β faktor množenja. Vsak člen v tej vsoti pripada številu klonov izbranega protitelesa. Na primer, za $N = 100$ in $\beta = 1$, protitelesce z najvišjo afiniteto ($i = 1$) proizvede 100 klonov, medtem ko jih telesce z drugo največjo afiniteto proizvede 50. Potem so kloni podvrženi mutaciji in dodajanju receptorskih molekul. Dodajanje receptorskih molekul ustvari možnost, da se iz lokalnega območja prestavimo nekam drugam in tako povečamo verjetnost da dosežemo globalni optimum (diverzifikacija). Stopnja mutiranja klonov je inverzno proporcionalna z afiniteno antigenov. Večja kot je afiniteta, manjša je stopnja mutiranja. Novo populacijo ovrednotimo, izberemo nekaj njenih elementov in jih dodamo originalni populaciji. Določen odstotek najslabših elementov prejšnje populacije protiteles zamenjamo z naključno generiranimi rešitvami. Ta korak prinese nekaj diverzitete v populacijo.

Naslednji algoritem prikaže splošno shemo CLONALG algoritmov:

Vnos: Začetna populacija P_0
$P = P_0$; generacija začetne populacije naključnih protiteles
Ponavljaj
  Ovrednoti vsa obstoječa protitelesa in izračunaj njihovo afiniteto
  Izberi $N\%$ protiteles z najvišjo afiniteto
  Kloniraj izbrana protitelesa
  Mutiraj klonirana protitelesa
  Ovrednosti vsa klonirana protitelesa
  Dodaj $R\%$ najboljših kloniranih protiteles v množico protiteles
  Odstrani najslabše elemente iz množice protiteles

Dodaj nova naključna protitelesa v populacijo
Dokler ni zadoščeno zaustavitvenemu kriteriju
Izhod: Najboljša najdena populacija

Glede na teorijo selekcije klonov, naslednja tabela prikaže analogijo med naravnimi imunskimi sistemi in reševanjem optimizacijskega problema.

Naravni imunski sistem	Optimizacijski problem
Protitelesce	Rešitev
Afiniteta	Namenska funkcija (objective function)
Antigen	Optimizacijski problem
Kloniranje	Reprodukcija rešitev
Mutacija	Večkratne mutacije rešitev
Zorenje afinitete	Mutacija in izbira najboljših rešitev
Dodajanje receptorskih molekul	Diverzifikacija

Princip negativne selekcije

Imunski sistem se lahko sooči z disfunkcionalnostjo, ko zameša tuje celice s lastnimi. V izogib tej disfunkcionalnosti, naše telo eliminira limfocite T, ki reagirajo na lastne celice, skozi proces imenovan negativna selekcija. Preživeli bodo samo limfociti, ki se ne odzivajo na lastne celice. Take limfocite imenujemo tolerantne.

Algoritmi negativne selekcije (NSA) so osnovani na principu zorenja T celic in diskriminacije v biološkem imunskem sistemu. Nezrele T celice so neke vrste detektorji anomalij. NSA je razvil Forres leta 1994, da bi rešili problem varnosti računalnikov. Ker so NSA generirani kot množica detektorjev anomalij, se večinoma uporabljajo pri zaščiti računalnika, problemih za zaznavanje vdora v omrežje, v zadnjem času pa se algoritmi z negativno selekcijo uporabljajo tudi za reševanje optimizacijskih problemov.

9.2.2 Omrežni AIS algoritmi

Teoretski razvoj teorije imunskih omrežij je začel Jerne leta 1974. Imunsko omrežje je definirano kot dinamičen in samoregulativen sistem, sestavljen iz imunskih celic, ki imajo sposobnost da komunicirajo in prepoznajo ne samo antigene, ampak tudi druge imunske celice. V teoriji imunskih omrežij je imunski sistem sestavljen iz celic in molekul, ki komunicirajo med sabo, zato ni potrebne nobene zunanje stimulacije antigenov.

Algoritem aiNET (artificial immune network). To je eden najbolj znanih algoritmov, osnovanih na teoriji imunskih omrežij.

Najprej imamo populacijo antigenov in naključno zgenerirano populacijo protiteles (B celic). Za predstavitev antigenov in protiteles uporabimo vektor realnih števil, afiniteta med antigenom in protitelescem pa je definirana z Evklidsko razdaljo. Potem izberemo dano število protiteles in jih kloniramo. Selekcija je osnovana na njihovi afiniteti z antigeni. Večja kot je

afiniteta protitelesa, večja je verjetnost, da bo izbran in večje bo število njegovih klonov. Zgenerirani kloni potem še mutirajo, pri čemer je stopnja mutacije inverzno proporcionalna ujemanju z antigeni.

Nato izberemo dano število klonov in jih vključimo v imunsko omrežje (clonal memory). Protitelesa z afiniteto z ostalimi protitelelesi nižjo od danega praga odstranimo iz imunskega omrežja (clonal suppresion). Tudi protitelesa z afiniteto z antigeni manjšo od danega praga odstranimo iz imunskega omrežja (natural death rate). Dano število novih slučajno zgeneriranih protiteles vgradimo v imunsko omrežje in jih ovrednotimo glede na njihovo afiniteto z obstoječimi protitelesi (metadynamics). Protitelesa z afiniteto manjšo od danega praga ponovno odstranimo.

Teorija nevarnosti (danger theory)
Teorijo nevarnosti je predstavil P. Matzinger leta 1994. Tradicionalne teorije imunskih sistemov so osnovane na razlikovanju med lastnim in nelastnim, teorija nevarnosti pa predpostavi, da imunskega sistema mogoče ne izzovejo tuje celice. Lahko se tudi zgodi, da ni nobene imunske reakcije na tujo bakterijo, ali pa da imunski sistem reagira na lastne celice, na primer prepozna lastne molekule, ki vsebujejo poškodovane celice. AIS algoritmi, osnovani na teoriji nevarnosti, se v zadnjem času uporabljajo za reševanje varnosti računalnikov in problemov učenja strojev.

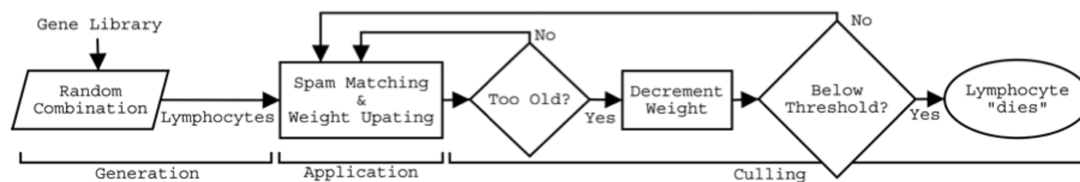
9.2.3 Primer: Preprečevanje neželjene pošte

Umetne imunske sisteme se uporablja tudi za preprečevanje prejemanja neželjene pošte. Tako kot imunski sistem loči med lastnimi in tujimi celicami, naj bi tudi program ločil med željeno in neželjeno pošto. Problem, ki se pojavi, pa je, da se v naravi antigen, značilen za določen tujek, tekom časa ne spreminja, medtem ko se predmeti zanimanja človeka in informacije, ki jih želi dobiti, venomer spreminjajo. Sistem mora biti zato sposoben ne samo učenja, ampak tudi pozabljanja - predvsem pa mora biti zelo prilagodljiv.

Neželjena pošta (spam) bo predstavljala tuje antigene, željeno sporočilo pa bo uporabljeno kot lastni antigen oziroma protitelo. Limfociti so digitalni biti informacije in vsak vsebuje vzorec, ki ga uporabimo kot protitelo.

Sistem zazna tujek, ko se nanj veže. Povezava ni nujno popolna, zato se lahko eno protitelo veže na več različnih antigenov, vendar se z nekaterimi veže bolje kot z drugimi. Moč povezave je torej odvisna od tega, kako dobro se ujemata obliki na antigenu in protitelesu. Obstaja več kot tisoč različnih antigenov, ki jih mora imunski sistem prepoznati, čeprav sam vsebuje le nekaj več kot 100 različnih receptorjev. Zato je tudi slabo povezovanje nujno, da lahko majhno število protiteles zazna veliko število možnih tujkov, ki so si podobnih oblik.

Da se sistemu ni potrebno vedno znova učiti, kateri antigeni so tuji, so v sistemu tudi spominske celice. Te si, potem, ko so enkrat že zaznale tuj antigen, zapomnijo, kako se braniti pred njim. Nekatere spominske celice lahko ostanejo v telesu za vedno, kar nam povzroči



Slika 9.7: Najprej imamo knjižnico podatkov. Na njej določimo elemente, ki označujejo spam in tiste, ki ga ne. Ko dobimo sporočilo, ga preverimo z uteženimi elementi iz knjižnice. V kolikor je element že zelo star, mu zmanjšamo utež. Vsakemu sporočilu tako priredimo število in če je nad danim pragom, ga označimo kot spam.
Vir:[18]

doživljenjsko odpornost. Primer, ki ga poznamo, so ošpice ali pa norice, ki jih ima večina ljudi že v otroštvu, potem pa telo postane nanje imuno in tako za njimi ponavadi zbolimo največ enkrat.

Tudi spam ponavadi vsebuje velike količine podobnih besed, tako da lahko sistem prepozna več podobnih ali enakih sporočil skozi daljše obdobje kot neželjene. Vendar ker želimo, da je naš sistem sposoben tudi pozabljanja, nočemo da je to obdobje trajno. Zato vsakemu antigenu dodamo utrež, ki nam pove, kako neželjen je ta del informacije. S starostjo informacije, se njena utež zmanjšuje in tako se lahko zgodi, da bomo vsake toliko še vedno dobili kakšen spam.

Tako kot imunski sistem tudi zaščita proti spamu deluje na več nivojih. Osnovna nespecifična obramba so zakoni, ki prepovedujejo določena sporočila, oziroma kaznujejo njihove pošiljatelje. Specifična obramba pa je kombinacija več različnih sistemov za preprečevanje spama.

Delovanje preprečevanja spama:
Najprej moramo imeti ‘knjižnico’elementov, za katere vemo, ali so željeni ali ne. To zbirko mailov uporabimo za učenje. V vsakem sporočilu preverimo delež željenih in neželjenih elementov ter vsakemu sporočilu priredimo število. Glede na mejo, ki smo jo določili, razvrščamo sporočila med spam ali pa navadno pošto. Ko je sistem enkrat naučen, se knjižnica z vsakim novim sporočilom posodablja, elementi knjižnice pa se lahko tudi starajo, odmirajo ali pa ‘rojevajo’na novo. Življenjski cikel elementov knjižnice je prikazan v sliki [9.7].

Na ta način se sistem, medtem ko razlikuje med lastnim in tujim, hkrati uči in pozablja informacije ter se tako prilagaja spreminjajoči naravi sporočil.

Poglavje 10

Kolonije Čebel

JULIA CAFNIK

V seminarski nalogi se bomo spraševali o optimizacijskem obnašanju čebeljih kolonij in posledično o tem, kako se njihovo obnašanje aplicira v optimizacijskih algoritmih za reševanje kompleksnih kombinatoričnih in optimizacijskih problemov. Videli bomo, da se lahko iz obnašanja čebel veliko naučimo in da nam lahko pomaga pri reševanju kompleksnih problemov.

10.1 Kolonije čebel

Čebelje optimizacijske algoritme uvrščamo med optimizacijo s pomočjo inteligence roja (ang. swarm intelligence algorithms).

Roj si lahko predstavljamo kot množico mobilnih agentov, ki lahko med seboj direktno oz. indirektno komunicirajo. V kolektivu lahko dosežejo rešitev problema preko decentralne in nenadzorovane koordinacije (samoorganizacija). Pri čemer se upošteva tudi robustnost. To pomeni, da se skupinski cilj lahko doseže tudi, če niso vsi v skupini zmožni sodelovati pri doseganju le tega in kjer se z dinamičnimi reakcijami prilagodijo na spremembe okolja - adaptacija. Primeri rojev so socialno živeče žuželke, jate ptic, jate rib ter človeško telo.

Komunikacija, samoorganizacija in semantična interakcija so sestavine inteligentnega obnašanja v kolektivu in tudi glavni akterji pri reševanju kompleksnih optimizacijskih problemov.

V zadnjem desetletju je bilo objavljenih veliko člankov na to temo. Optimizacijski algoritmi s pomočjo čebeljih kolonij temeljijo na obnašanju čebel v kolonijah. Mnogo značilnosti se da uporabiti v modelih inteligentnega in kolektivnega obnašanja. Med ta obnašanja se uvrščajo tudi iskanje nektarja, parjenje med letom, iskanje hrane, zibajoči ples in delitev dela.

Algoritmi na podlagi čebeljih kolonij večinoma temeljijo na treh različnih modelih. Vsakega izmed teh bomo predstavili v seminarski nalogi. To so: iskanje hrane, iskanje gnezda ter svatbeni let.

10.2 Splošno o čebelah

Čebela je socialna žival - živi v kolonijah oz. skupinah, ki štejejo na višku razvoj od 20000 pa vse do 80000 osebkov. Čebele živijo v divjini v naravnih bivališčih. V družini je navadno ena matica, v času čebelje paše pa od nekaj 100 do nekaj 1000 trotov. Vse drugo so čebele delavke. Na kratko: Čebela matica in čebele delavke so ženski osebki, troti pa so moški osebki.

Matica je edina spolno zrela samica v družini. Njena naloga je ustvarjanje potomstva. V čebelji družini je ena sama matica. Matica je večja in težja od čebel delavk in živi do 5 let.

Čebele delavke so neplodne samice. V družini so najmanjše. Imajo dobro razvite organe za zbiranje hrane, za gradnjo satja, za hranjenje legla in za obrambo cele družine. Živijo od 5 tednov do 8 mesecev.

Glede na delo, ki ga opravljajo, jih delimo na hišne (panjske) in pašne čebele.

Hišne čebele so sprva čistilke, stare do 6 dni, ki počistijo celico in okolico, nato postanejo stražarke, ki stražijo ob vhodu v panj. Čebele graditeljice so stare od 12 do 18 dni in proizvajajo vosek ter gradijo gnezdo - satje (šesterokotne celice).

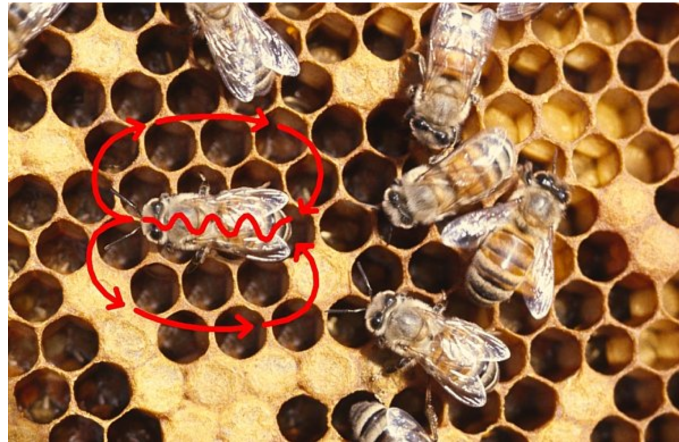
Pašne čebele so nabiralke stare 19 do 60 dni. Poletijo v naravo in nabirajo potrebne snovi za obstoj družine, kot so nektar - medičina, cvetni prah - pelod,drevesno smolo - propolis ter vodo.

Trot je samec, ki oplodi matico in pomaga s svojo telesno toploto ogrevati čebeljo zalego. Živi od 3 do 6 mesecev - od spomladi do poletja, ko je v naravi dovolj paše. Konec poletja jih delavke izrinejo iz panja pred čebelnjak, kjer poginejo. Ob nenormalnih razmerah izjemoma ostanejo v družini tudi pozimi, če ni matice ali je ostala le ta neoplojena.

10.3 Iskanje novega gnezda

Pozno pomladi oz. zgodaj poleti se čebele v koloniji razdelijo v dve skupini. Matica in polovica čebel delavk tvorijo prvo skupino, hčer matice in druga polovica čebel delavk pa tvorijo drugo skupino. Prva skupina namerava ustvariti novo kolonijo, druga skupina pa bo nato uničila prvo skupino. Za iskanje novega prostora za gnezdenje se nekaj sto čebel odpravi na pot, ostale pa ostanejo v gnezdu. Najverjetneje zato, da ohranijo čim več energije v roju, dokler ne pade odločitev o tem, kam se bodo selili oz. kje bo novo gnezdo in svojo energijo porabijo za selitev k novemu gnezdu. Čebele iskalke predstavijo različne lokacije gnezda s pomočjo zibajočega plesa. Hitrost in in orientacija plesa sta povezani s kakovostjo lokacije gnezda. Različni atributi so uporabljeni za karakterizacijo lokacije gnezda, kot so npr. vhodno območje, smer vhoda, višina vhoda, velikost gnezda itd. Čebele ne odnehajo, dokler se čebelji ples ne nagne k eni strani. Z drugimi besedami, čebele iskalke glasujejo za svoje najljubšo lokacijo preko plesa in se nato preko glasovanja skupinsko odločijo, katera lokacija bo zmagala. Zibajoči ples služi za

informiranje ostalih čebel o lokaciji gnezda ter celo o viru hrane. Kvaliteta lokacije gnezda je povezana z velikostjo gnezda, oddaljenosti od starega gnezda itd.



Slika 10.1: **Zibajoči ples** je vedenjski vzorec, ki ga izvajajo delavke nekaterih vrst čebel v koloniji. Ko delavka, ki ima nalogo iskanja hrane (t.i. skavtka), odkrije zadovoljiv vir hrane, se ob povratku v panj postavi na sat in se prične premikati v vzorcu osmice z ravnim srednjim delom. V ravnem delu pozibava z zadkom in pobrenčava s krili, na koncu pa se vsakič v drugo smer obrne in vrne na izhodišče. To gibanje lahko ponovi tudi do več desetkrat, delavke, ki so okrog nje, pa ga aktivno spremljajo. Poleg paše je lahko ples tudi sporočilo o legi vodnega vira ali primernega kraja za ustvarjenje nove kolonije pri rojenju.

10.4 Iskanje hrane

Ena izmed vlog, ki jih ima čebelja kolonija, je iskanje novih, bogatih virov hrane. Glede na vlogo posameznih čebel v koloniji, jih delimo na:

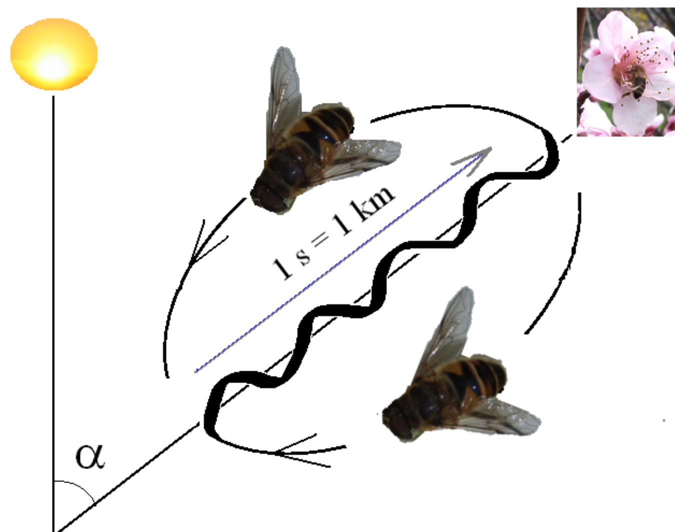
- **Nezaposlene iskalke.** Obstajajo čebele, ki nimajo znanja o tem, kje ležijo bogati viri hrane. Ločimo med dvema vrstama nezaposlenih čebel: skavtke, ki iščejo hrano v okolju, ki obdaja gnezdo ter čebele opazovalke, ki čakajo v gnezdu in opazujejo zibajoči ples ostlih čebel in se nato odločijo, ali bodo sledile priporočenemu viru hrane.

- **Zaposlene iskalke.** Zaposlene čebele so povezane z določenim izvirom nektarja. Imajo potrebno znanje o izvoru hrane. Najdejo vir za iskoriščanje, si zapomnijo njegovo lokacijo, natovorijo nekaj nektarja in se vrnejo v gnezdo. Lahko omenimo tri možna stanja glede na količino nektarja: če je nektarja zelo malo, ker je vir že preveč izkoriščen, zaposlene iskalke zapustijo ta vir hrane in postanejo nezaposlene iskalke. Če je dovolj nektarja le zanjo, čebela ne deli vira z ostalimi čebelami in nadaljuje z izkoriščanjem tega vira. In tretja možnost je, da če najde čebela pomeben vir hrane, se vrne v gnezdo, kjer na plesišču izvede zibajoči ples, da bi o najdišču obvestila še ostale čebele.

Glede vira hrane imajo čebele dve strategiji:

- **Iskanje novega vira hrane.** Skavtka raziskuje okolico gnezda, da bi našla nov vir hrane. Če najde vir hrane, se vrne v gnezdo in na plesišču izvede zibajoči ples, da bi o najdišču obvestila še ostale čebele. Opazovalke postanejo zaposlene iskalke.

- **Izkoriščanje vira hrane.** Zaposlene iskalke preračunajo količino in kakovost vira hrane. Lahko nadaljuje z izkoriščanjem tega vira, ali pa se odloči, da ga zapusti. V tem primeru zaposlena iskalka postane nezaposlena iskalka; postane skavtka ali opazovalka.



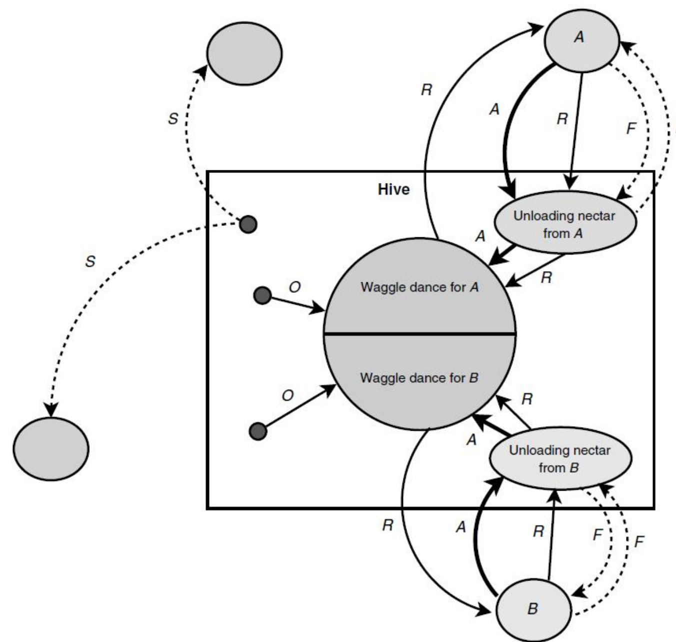
Slika 10.2: Zibajoči ples vsebuje informacijo o oddaljenosti in smeri vira hrane. Informacijo o oddaljenosti daje dolžina ravnega dela - za vsake 100 m oddaljenosti se trajanje ravnega dela podaljša za okrog desetinko sekunde. Ker je satje v panju navpično, se informacija o smeri prevede na navpičnico - kot ravnega dela osmice glede na smer navpično navzgor odgovarja kotu vira hrane glede na položaj Sonca. Za primer, če skavtka izvaja osmico z ravnim delom navpično navzgor, pomeni to, da je paša v smeri naravnost proti Soncu. Zibajoči ples je torej abstraktno sporočilo o razdalji do paše in njeni legi.

Naravna kolonija čebel	Umetna kolonija čebel
Vir hrane	Rešitev
Kvaliteta nektarja	Kriterijska funkcija
Opazovalke	Izkoriščanje iskanja
Skavtka	Raziskovanje iskanja

Tabela 10.1: Analogija med naravnimi in umetnimi čebeljimi sistemi

Kolonija čebel se lahko razširi (do 14km) v različne smeri, tako da čimbolj učinkovito išče različne vire hrane. Kolonija se razvija in krepi s pomočjo razporejanja svojih iskalcev k virom hrane, ki so dobre kvalitete. Mesta, kjer so cvetlične paše z veliko količino nektarja oz. veliko cvetnega prahu, naj bodo obiskovana od velikega števila čebel, medtem ko cvetlične paše z manj nektarjem oz. cvetnim prahom, pa naj bodo obiskovana od manjšega števila čebel. Proces

iskanja hrane v koloniji se začne s skavtkami, ki so poslane naokrog, da bi našle obetavne cvetlice. Skavtke se premikajo naključno od enega mesta k drugemu. Med žetvenim obdobjem, kolonija nadaljuje z iskanjem hrane, tako da ohrani en del populacije kot skavtke. Skavtke, ki so našle pašo, ki je nad določeno mejo, kar se tiče kvalitete (merjeno kot kombinacija različnih meril, kot je vsebovanost sladkorja itd.), se vrnejo v gnezdo in odložijo nektar in gredo na ”plesišče”, kjer odplešejo zibajoči ples. Ta ples služi kot komunikacija med ostalimi čebelami in vsebuje tri vrste informacij o cvetlici: smer, v kateri se nahaja cvetlica, oddaljenost od gnezda ter njena uspešnost oz. kvaliteta (angl. fitness). Kolonija, ki ima to informacijo, je natančno obveščena o tem, kam poslati svoje čebele.



Slika 10.3: Obnašanje kolonije čebel pri odkrivanju vira hrane. Recimo, da imamo dva na novo odkrita vira hrane, vir A in B. Nezaposlena iskalka nima znanja o tem, kje se nahaja vir hrane. Lahko je čebela skavtka S, ki začne odkrivati okolico gnezda, ali pa opazovalka O, ki opazuje zibajoči ples. Po opredelitvi lokacije vira hrane, čebela postane zaposlena iskalka. Po tem, ko zaposlena čebela odloži nektar, ima tri možnosti: zapustiti vir hrane (A), pridobiti nove člane za najdeni vir hrane ali nadaljevati z izkoriščanjem vira hrane, brez novih čebel.

Znanje posamezne čebele o pašah temelji le na učenju iz zibajočega plesa. Ta ples omogoči koloniji, da primerja koristnost različnih paš med seboj glede na kvaliteto hrane in oddaljenost oz. energijo, ki jo bodo porabile, če bodo tam nabirale. Po zibajočem plesu se plesalke (tj. skavtka) vrne k cvetlici skupaj s sledilkami, ki so čakale znotraj gnezda. Več sledilk kot gre, bolj je najdišče obetavno. To omogoči koloniji, da pridobiva hrano hitro in učinkovito. Medtem ko nabirajo, opazujejo zalogo paše. To je potrebno zato, da se odločijo, ali bodo tudi pri naslednji vrnitvi izvedle zibajoči ples in s tem promovirale to najdišče in si prizadevale za še večje število čebel, ki bo tam nabiralo, ali pa se bodo odločile, da je to nabirališče sedaj že preveč izčrpano in ni več potrebno novih čebel privabit v to najdišče.

Psevdokoda algoritma, poimenovan čebelji algoritem (angl. Bee algorithm - BA) je nastal na podlagi vedenja čebelje kolonije pri iskanju hrane. Algoritem se začne z n skavtkami, ki so naključno razporejene po prostoru iskanja (angl. search space). Oceni se kvaliteto najdišča, ki ga je našla skavtka (tj. rešitev). Nato se izbere tiste čebele, ki so bile pri tem najbolj uspešne. Najbolje ocenjenih paš naj bo e. Te paše se nato uporabi kot izhodišče za raziskovanja njihove okolice. Nato algoritem poskrbi za to, da se v okolici najboljših e lokacij naredi čimveč novih iskanj hrane, tako da se tja nameni čimveč čebel opazovalk. Čebele so lahko izbrane neposredno, glede na posamezno uspešnost lokacije, ki jo obiskujejo ali pa je uspešnost lokacije, kjer nabirajo, pogojena z verjetnostjo, s katero bodo tja poslane. Bolj ko je neka lokacija uspešna, več čebel se bo tja napotilo. Skupaj z iskanjem paše sta ti dve operaciji ključnega pomena v algoritmu kolonije čebel. Za vsako pašo se nato izbere le najbolj uspešno čebelo, ki bo tvorila naslednjo generacijo čebel. Ta restrikcija je vpeljana umetno in se v naravi ne zgodi. Vpeljana je zato, da se zmanjša število potencialnih rešitev. Nato preostale čebele zopet naključno razporedimo po prostoru in začne se novo iskanje hrane. Proces se nadaljuje, dokler ne doseže nek določen ustavljalni kriterij. Na koncu vsake iteracije, ima kolonija dve skupini, ki bodo tvorile novo populacijo; predstavnice posameznih najdišč ter skavtke, ki so naključno razporejene za iskanje novih virov hrane.

Psevdokoda čebeljega algoritma:

Naključna inicializacija kolonije čebel
Ocenjevanje uspešnosti populacije čebel
repeat
  /*Ustvarjanje nove populacije*/
  Določanje poti za raziskovanje soseščine
  Določiti velikost najdišča
  Določiti število čebel za posamezno najdišče ter oceniti njihovo uspešnost
  Izbrati predstavnika iz vsakega nabirališča
  Preostale čebel preusmeriti k naključnemu letenju in iskanju novih najdišč ter oceniti njihovo uspešnost
until Zaustavitveni kriterij

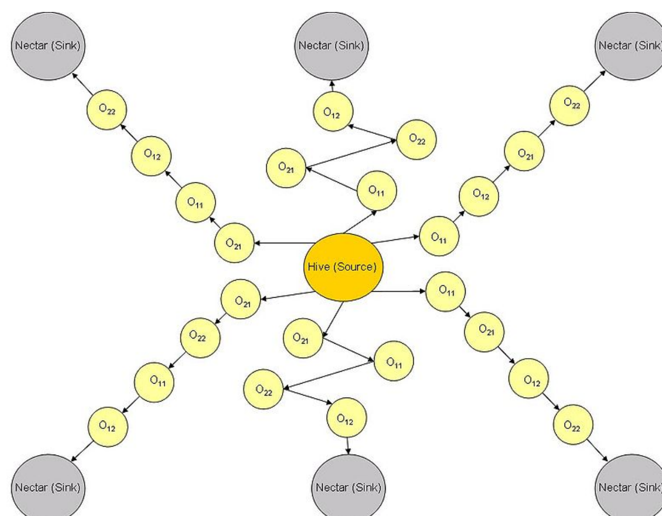
Ta algoritem zahteva nekaj parametrov, ki jih je potrebno na začetku določiti. To so število skavtk (n), število izbranih mest (m) za nadaljnje nabiranje izmed vseh obiskovanih mest, število najboljših najdišč izmed izbranih najdišč (e), število dodeljenih čebel za najboljša nabirališča (nep), število čebel, določenih za ostala ($m - e$) najdišča (nsp), začetna velikost najdišč (ngh), ki vključuje tudi iskalni teren, soseščino ter zaustavljalni kriterij.

Primer uporabe optimizacija pri čebeljih kolonijah. Sedaj bomo predstavili primer uporabe optimizacije pri čebeljih kolonijah. S to vrsto optimizacije si bomo pomagali pri problemu razporejanja opravil (angl. job-shop scheduling problem).

Problem razporejanja opravil (JSP) je optimizacijski problem, kjer imamo j opravil (angl. jobs) različnih velikosti. To pomeni, da se razlikujejo po času, ki je potreben za posamezno opravilo. Vsako opravilo je sestavljeno iz M operacij, ki se jih realizira na M strojih. Vsaka operacija se izvede na posameznem stroju. Vsaka opravilo je sestavljeno iz vseh operacij. Urnik (angl. schedule) nam pove, kdaj se kaj na posameznem stroju izvaja. Kriterijska funkcija, ki jo želimo minimizirati je celotno trajanje opravila.

Dopustna rešitev JSP-ja je dovršen urnik operacij. Vsako rešitev si lahko predstavljamo kot pot od gnezda do vira hrane. Trajanje opravila, ki ga dobimo kot rešitev, je enako koristnosti vira hrane, torej v smislu oddaljenosti in kvalitete nektarja. Krajše kot je trajanje, večja je koristnost poti. Torej lahko opazujemo kolonijo čebel, kjer bo vsaka čebela potovala proti potencialni poti (imamo disjunktni graf). Preskrbovalci se premikajo po povezavah od enega vozlišča k drugemu v disjunktnem grafu. Preskrbovalec mora obiskati vsako vozlišče enkrat, s tem da prične pri izvornem vozlišču (tj. gnezdo) in konča pri končnem vozlišču (tj. vir hrane oz. cvetlica). Ko je enkrat odkrita dopustna rešitev, se bodo čebele vrnile v gnezdo, da bi izvedle zibajoči ples. Zibajoči ples bo predstavljal vrsto ustreznih rešitev, izmed katerih bodo ostale

čebele lahko odločale, kateri poti bodo sledile. Čebele z boljšo kriterijsko funkcijo bodo imele višjo verjetnost, da se bo njihovo pot uvrstila med dobre rešitve. S tem se bo spodbudilo, da bodo dobre rešitve skonvergirale k najboljši rešitvi.



Slika 10.4: Uporaba optimizacije čebljih kolonij pri reševanju job-shop scheduling problema.

10.5 Svatbeni let

Razmnoževanje čebel poteka preko matice, ki se pari s približno 7-20 trotov. Temu procesu pravimo svatbeni let. Svatbeni let poteka v zraku, daleč stran od gnezda. Sprva matica izvede poseben ples. Troti jo opazujejo in se nato parijo z njo. Svoje spermije položijo v njeno spermateko. Ko je spermateka polna ali ko energija matice pade pod določen prag, se vrnejo v gnezdo. Tri dni po zadnjem svatbenem letu matica izleže svoja jajčeca. Iz oplojenih jajčes se razvijejo mlade čebele, ki so bodisi matice bodisi čebele delavke. Iz neopljenih jačec pa se razvijejo troti. MBO (angl. Marriage in honeybees optimization) algoritem se je razvil na podlagi svatbenega leta čebel. Osnovan je na svatbenem letu, ki si ga lahko predstavljamo kot množica tranzicij v prostoru stanj (v okolju), kjer se matica sprehaja med temi različnimi stanji z neko hitrostjo in se z neko verjetnostjo pari s trotom, na katerega naleti. Na začetku leta je matica inicialzirana z neko energijo in se vrne v svoje gnezdo, ko njena energija pade pod določeno mejo oz. ko je njena spermateka polna. Trot se pari s čebelo z neko verjetnostjo prob(Q, D):

$$\text{prob}(Q, D) = e^{\frac{-\Delta(f)}{S(t)}},$$

kjer je prob(Q, D) verjetnost, da bo trot D položil svojo spermo v spermateko matice Q. To je torej verjetnost, da bo parjenje uspešno. $\Delta(f)$ je absolutna vrednost razlike med uspešno položitvijo spermija v spermateko (tj. $f(D)$) ter uspešnostjo matice Q (tj. $f(Q)$). $S(t)$ je hitrost matice za časa t. Verjetnost za parjenje je visoka, ko je matica še v začetku svojega paritvenega

obdobja in je njena hitrost visoka, ali ko je uspešnost trota enaka uspešnosti matice. Po vsaki tranziciji, se hitrost in energija matice spremenita sledeče:

$$S(t+1) = \alpha \cdot S(t)$$
$$E(t+1) = E(t) - \gamma,$$

kjer je $\alpha \in (0,1)$ in γ je delež enregije, ki se zmanjša po vsaki tranziciji.

Psevdokoda za MBO algoritem:

```
Naključna inicijalizacija matice
Izboljšanje matice s pomočjo trotov
for Vnaprej definirano število svatbenih letov do

    Inicijalizacija energije in hitrosti
    while Energija matice > 0 do

        Matica se giblje med stanji in z neko verjetnostjo najde trota
        if trot izbran then
            Doda svojo spermo v matično spermateko
            Posodobimo energijo in hitrost matice
        end if
    end while
    Generiranje mladih čebel preko križanja in mutacij
    Uporaba delavcev za izboljšanje mladih čebel
    Posodobitev uspešnosti delavcev
    if najboljša mlada čebela je boljša od matice then
        Nadomestitev kromosoma matice z najboljšim kromosomom mlade čebele
        Uboj vseh mladih čebel
    end if
end for
```

Ta psevdokoda je osnova MBO algoritmu. Algoritem se začne z naključno inicijalizacijo matičnega genotipa. Nato se s pomočjo delavcev izboljša matični genotip. Privzamemo, da je ponavadi matica dobra čebela. Nato pride do množičnih svatbenih letov. Naključno se inicializira matična energija in hitrost. Nato se matica giblje med različnimi stanji (tj. rešitvami) v prostoru, glede na njeno hitrost in se pari s troti. Če je parjenje uspešno, trot doda svojo spermo v matično spermateko (tj. množica delnih rešitev). Nato se hitrost in energija matice zmanjšata. Ko matica zaključi s svojim svatbenim letom, se vrne v gnezdo, kjer pride do križanja naključno izbrane sperme iz spemateke z genomom matice. Križanje se konča z nastankom mlade čebele.

Pride do mutacije mladih čebel. Delavci nato izboljšajo uspešnost mladih čebel in oceni se uspešnost vsakega delavca glede na obseg izboljšav h katerim je pripomogel. V primeru, da je mlada čebele bolj uspešna od matice, jo ta nadomesti. Preostale mlade čebele se ubije in nadaljuje se z naslednjim svatbenim letom.

Parametri, ki jih mora uporabnik sam določiti so število matic, velikost matične spermateke, ki predstavlja maksimalno število parjenj na matico v enem svatbenem letu, število in tip delavcev, in število mladih čebel, ki jih bodo ustvarile matice.

10.6 Zaključek

Razvijanje optimizacijskih algoritmov na podlagi obnašanja čebeljih kolonij je razmeroma novo področje v raziskovanju. Veliko je še neodkritega. Ena izmed glavnih prednosti, ki se ne uporabi samo pri iskanju hrane, ampak tudi pri iskanju novega gnezda, je ta, da imajo čebele sposobnost, da iz različnih virov hrane izberejo najboljši vir. Zanimivo pri tem je, da se to zgodi brez centralnega odločanja in brez, da bi morale čebele obiskovati vsako najdišče posebej in jih nato primerjati med seboj. Odločitev o najboljšem najdišču je končni produkt neke vrste konkurenčnega boja na plesišču med čebelami, kjer ples za najboljšo najdišče po določenem času izpodrine ostale plese za ostala najdišča. Iz obnašanja čebel se lahko veliko naučimo in to uporabimo pri reševanju najrazličnejših optimizacisjkih problemov.

Poglavje 11

PAES

JELENA MARČUK

11.1 Uvod

11.1.1 Večkriterijska optimizacija

V uvodnem delu seminarske naloge bomo predstavili osnovne koncepte večkriterijske optimizacije in razloge za uporabo evolucijskih algoritmov pri reševanju tovrstnih problemov. Večkriterijski problemi so problemi, ki imajo dve ali več, ponavadi nasprotujočih si kriterijskih funkcij. Osnovna razlika med optimizacijo z eno kriterijsko funkcijo in večimi je ta, da večkriterijski problem nima ene optimalne rešitve ampak množico rešitev, med katerimi ponavadi velja negativna povratna zveza. To pomeni, da izboljšanje rešitve po enem kriteriju povzroči njeno poslabšanje po drugih kriterijih. Eden izmed načinov kako se temu izognemo je, da večkriterijski problem prevedemo na enokriterijskega in sicer tako, da vsaki izmed prvotnih kriterijskih funkcij priredimo utež. Na ta način za vsak nabor uteži dobimo eno optimalno rešitev problema. To je široko uporabljena in enostavna metoda, ki pa ima veliko pomanjkljivosti. Glavni problem, ki se pojavi pri metodi utežene vsote je subjektivnost odločevalca, ki v končni fazi pripisuje uteži glede na svoja lastna prepričanja o pomembnosti kriterijskih funkcij. Objektiven pristop zato sloni na Pareto optimalnem razvrščanju rešitev. Rešitev je Pareto optimalna, če ne obstaja nobena druga rešitev, ki bi bila boljša od te rešitve po vseh kriterijih [12].

Problem večkriterijske optimizacije je definiran kot problem iskanja dopustnega vektorja spremenljivk, ki optimizira vektorsko funkcijo, katere elementi so kriterijske funkcije. Pri enokriterijski optimizaciji je prostor kriterijev množica realnih števil, ki je popolno urejena, kar pomeni, da poljubni dve rešitvi brez problema lahko razrstimo med sabo. Pri večkriterijskem optimizacijskem problemu pa je prostor kriterijev večdimenzionalen in nam dopušča le delno

ureditev rešitev. Dve rešitvi sta tako pogosto neprimerljivi. Koncept Pareto dominantnosti reši ta problem in nam omogoči primerjavo rešitev večkriterijskega optimizacijskega problema.

Rešitev večkriterijskega problema x dominira rešitev y, če sta izpolnjeni naslednji dve zahtevi:

1. Rešitev x ni slabša od rešitve y po vseh kriterijih,

2. Rešitev x je boljša od rešitve y po vsaj enem kriteriju.

Iz definicije dominantnosti izhaja več drugih pomembnih definicij. Pri optimizaciji namreč iščemo množico nedominiranih rešitev v neki množici rešitev. Množica nedominiranih rešitev v množici vseh dopustnih rešitev je Pareto optimalna množica, njeni elementi pa Pareto optimalne rešitve. Slika Pareto optimalne množice v prostoru kriterijev se imenuje Pareto čelo. Cilj večkriterijske optimizacije je poiskati aproksimacijsko množico, ki bo čim bližje Pareto čelu in katere elementi bodo čimbolj enakomerno razporejeni vzdolž čela.

11.1.2 Evolucijski algoritmi

Evolucijski algoritmi temeljijo na Darwinovi teoriji o evoluciji: s selekcijo in kombinacijo genetskega zapisa osebkov skozi generacije ustvarjajo vedno boljše in boljše rešitve. Ti algoritmi se uspešno uporabljajo v enokriterijski optimizaciji. Ker so populacijski, so primerni tudi za naloge večkriterijske optimizacije, kjer želimo v enem zagonu algoritma dobiti več nedominiranih rešitev. Predstavljeni so bili številni evolucijski pristopi k reševanju večkriterijskih problemov, metode pa se med seboj v glavnem razlikujejo po tem, na kakšen način opravljajo selekcijo in s kakšnimi prijemi dosegajo enakomerno razporejenost rešitev [14].

V zadnjih letih je bilo veliko različnih raziskav na področju večkriterijske optimizacije z genetskimi algoritmi. Vendar pa so pri optimizaciji praktičnih problemov metode lokalnega iskanja zasenčile genetske algoritme tako pri enokriterijski, kot pri večkriterijski optimizaciji, kjer so kriteriji združeni s pomočjo metode utežene vsote. Malo raziskav se je ukvarjalo s tem, kako je metode lokalnega iskanja možno uporabiti pri reševanju resnično večkriterijskih problemov, da bi našli čimbolj raznolike rešitve v Pareto optimalni množici. V seminarski nalogi bomo predstavili evolucijski algoritem PAES (*Pareto Archived Evolution Strategy*), ki sta ga razvila Knowels in Corne [13]. Algoritem uporablja metodo lokalnega iskanja za generacijo novih kandidatov za rešitev, pri postopku selekcije pa si pomaga s shranjenimi informacijami o populaciji.

11.2 Usmerjanje prometa v komutiranih omrežjih

V tem razdelku bomo predstavili širši okvir, v katerem je bil razvit PAES algoritem. V ta namen bomo opredelili problem oziroma kriterijske funkcije, ki jih je potrebno optimizirati.

11.2.1 Opredelitev problema

Problem usmerjanja prometnih tokov zahteva izbiro množice poti za dano omrežje, tako da so pri danem prenosu komunikacijski stroški in zastoji minimizirani. Dano je telekomunikacijsko omrežje z omejeno pasovno širino, čez katerega moramo usmeriti več prometnih zahtev v skladu s tremi kriteriji. Primarni kriterij, ki mu želimo zadostiti je, da je usmerjanje dopustno, oziroma da nobena povezava ni preobremenjena. Drugi kriterij zahteva, da so stroški komunikacije, povezani z uporabo vsake povezave, čim manjši. Nazadnje pa želimo, da je izkoriščenost povezav pod določeno ciljno mejo.

Podano je omrežje $G = (N, E)$, kjer je N množica vozlišč velikosti n in E množica dvosmernih povezav velikosti m. Vsaka povezava $e \in E$ ima kapaciteto $b(e)$ in ceno $c(e)$. Kapacitete povezav v omrežju $b(e),\ e \in E$ ležijo v množici $\{16, 64\}$, tj. obstajata dve vrsti povezav; povezave hrbteničnega omrežja s kapaciteto 64 in lokalne povezave s kapaciteto 16. Promet poteka v časovnem okviru, ki ga označimo z množico T, ki vsebuje l diskretnih časovnih intervalov $t \in T$.

Podana je še množica R z j komunikacijami, ki morajo biti usmerjene čez G. Vsaka komunikacija $r \in R$ določa izvirno vozlišče $v(r)$ in destinacijo $w(r)$ tako, da je $v(r), w(r) \in N$, $\forall r$. Za vsako komunikacijo r so dani tudi: konekcijski čas $\tau_\alpha(r) \in T$, diskonekcijski čas $\tau_\beta(r) \in T$ in pasovna širina $h(r)$.

Naloga je najti pot $P(r)$ v G za vsako komunikacijo r, ki povezuje $v(r)$ in $w(r)$ v časovnem intervalu $\tau_\alpha(r) \leq t < \tau_\beta(r)$ tako, da minimiziramo spodaj naštete kriterijske funkcije. Omeniti je potrebno, da je za vsaki klic samo ena pot, ki obstaja skozi celoten čas trajanja povezave. To pomeni, da se klici, ko so enkrat povezani, ne preusmerjajo.

Sedaj lahko zapišemo skupni promet po povezavi e v časovnem intervalu t

$$f(e,t) = \sum_{r \in R} \{h(r) \,|\, e \in P(r),\ \tau_\alpha(r) \leq t < \tau_\beta(r)\}\,. \tag{11.1}$$

Dopustno usmerjanje je tisto, v katerem skupni promet po vsaki povezavi v vsakem ločenem časovnem intervalu ne presega kapacitete povezave

$$f(e,t) \leq b(e),\ \forall e \in E,\ \forall t \in T. \tag{11.2}$$

To lahko dosežemo z minimizacijo razlike $f(e,t) - b(e)$, dokler je $f(e,t) > b(e)$

$$\min \sum_{t \in T} \sum_{e \in E} \max \left\{ f(e,t) - b(e), 0 \right\}. \tag{11.3}$$

Enačba (11.3) predstavlja prvo kriterijsko funkcijo.

Za zadostitev drugega kriterija je potrebno najti tako usmerjanje prometa čez omrežje, da so stroški minimalni, pri čemer moramo upoštevati omejitev (11.3). Strošek usmerjanja ene komunikacije $r \in R$ v enem časovnem intervalu $t \in T$ po poti $P(v,w)$ med v in w je dan z naslednjim izrazom

$$g(r,t) = \{h(r) \mid \tau_\alpha(r) \le t < \tau_\beta(r)\} \times \sum_{e \in P(r)} c(e). \tag{11.4}$$

To pomeni, da je skupni strošek usmerjanja vseh komunikacij v celotnem časovnem okviru T enak

$$\sum_{t \in T} \sum_{r \in R} g(r,t). \tag{11.5}$$

Če izraz (11.4) vstavimo v (11.5), dobimo drugo kriterijsko funkcijo, ki jo želimo minimizirati

$$\min \sum_{t \in T} \sum_{r \in R} \left\{ \{h(r) \mid \tau_\alpha(r) \le t < \tau_\beta(r)\} \times \sum_{e \in P(r)} c(e) \right\}. \tag{11.6}$$

Tretji in zadnji kriterij je minimizacija odklonov od ciljne izkoriščenosti u za vsako povezavo v omrežju po vseh časovnih korakih

$$\min \sum_{t \in T} \sum_{e \in E} \max \left\{ f(e,t) - \frac{u \times b(e)}{100}, 0 \right\}. \tag{11.7}$$

Vse dokler je $f(e,t) > u \times b(e)/100$ za vsaj eno povezavo $e \in E$ in vsaj en časovni interval $t \in T$, obstaja pritisk na optimizacijski proces, da najde bolj uravnotežene rešitve. V poskusih, ki sta jih izvedla Knowles in Corne, je bil $u = 50$. Prvotno so bile tri zgoraj navedene kriterijske funkcije združene v eno s pomočjo metode utežene vsote. Uteži, ki so jih uporabljali, so se že v prejšnjih poskusih drugih avtorjev izkazale za učinkovite, saj dajejo dopustna in dobro uravnotežena usmerjanja.

11.3 PAES

11.3.1 Pregled

PAES (Pareto Archived Evolution Strategy) algoritem je bil razvit z dvema glavnima ciljema v mislih. Prvi je ta, da mora biti algoritem striktno omejen na lokalno iskanje, to pomeni, da mora uporabljati mutacijske operatorje z majhnimi spremembami in se premikati od trenutne rešitve

do bližnjega soseda. Zaradi tega je PAES precej drugačen od najbolj znanih genetskih algoritmov za večkriterijsko optimizacijo, ki ohranjajo populacijo rešitev, iz katere se potem izvaja selekcija. Drugi cilj je, da mora algoritem obravnavati vse nedominirane rešitve na isti način in jim prirediti enako vrednost. Doseganje obeh ciljev istočasno je tako zelo problematično. To je zato, ker v večini primerov primerjava dveh rešitev ni izvedljiva, saj nobena ne dominira druge. PAES premaga ta problem tako, da shranjuje že najdene nedominirane rešitve v arhiv, ki se potem uporablja kot orodje za določanje dominantnosti med rešitvami.

Na PAES algoritem lahko gledamo kot da je sestavljen iz treh osnovnih delov: generatorja kandidatov za rešitev, funkcije, ki sprejema rešitve in arhiva nedominiranih rešitev. Tako gledano, PAES predstavlja najenostavnejši netrivialni pristop k postopku večkriterijskega lokalnega iskanja. Generator kandidatov za rešitev ohranja eno trenutno rešitev in v vsaki iteraciji s pomočjo naključne mutacije proizvede enega novega kandidata. Način delovanja funkcije, ki sprejema kandidate za rešitev, je očiten v primeru, ko kandidat dominira trenutno rešitev ali obratno, vendar problematičen v primeru nedominiranosti. V slednjem primeru PAES uporablja primerjalno množico, da lahko izbere med kandidatom in trenutno rešitvijo. Arhiv nedominiranih rešitev predstavlja naraven in priročen vir, iz katerega dobimo primerjalno množico.

11.3.2 Delovanje algoritma

Struktura PAES algoritma je prikazana na sliki 11.1. Algoritem se začne z inicializacijo enega samega kromosoma, ki mu rečemo trenutna rešitev. Na drugem koraku algoritem s pomočjo večkriterijske stroškovne funkcije oceni dano trenutno rešitev. Potem se naredi kopija trenutne rešitve, ki jo spremenimo s pomočjo mutacijskega operatorja. Mutirana kopija, ki jo spet ocenimo, predstavlja novega kandidata za rešitev. Na naslednjem koraku primerjamo staro trenutno rešitev z novim kandidatom. V primeru ko ena rešitev dominira drugo, se enostavno odločimo katero rešitev bomo obdržali. Problem nastane, ko sta rešitvi neprimerljivi oziroma ko nobena ni dominirana s strani druge. Takrat novega kandidata primerjamo z referenčno populacijo nedominiranih rešitev, ki so shranjene v arhivu. Če primerjava s populacijo v arhivu ne da prednosti eni rešitvi pred drugo, potem obdržimo tisto rešitev, ki leži v najmanj zgoščenem delu kriterijskega prostora.

Arhiv ima dve različni vlogi. Prva je da shranjuje in posodablja vse nedominirane rešitve. Druga vloga arhiva je, da predstavlja aproksimacijo za trenutno nedominirano čelo in zato omogoča pravilno izbiro med trenutno rešitvijo in kandidatom. Slednje je tisto, kar nam zagotavlja, da bo proces na vsakem koraku našel boljšo rešitev. Brez te vloge arhiva algoritem ne loči med dobro in slabo rešitvijo in se zato brezciljno sprehaja po kriterijskem prostoru.

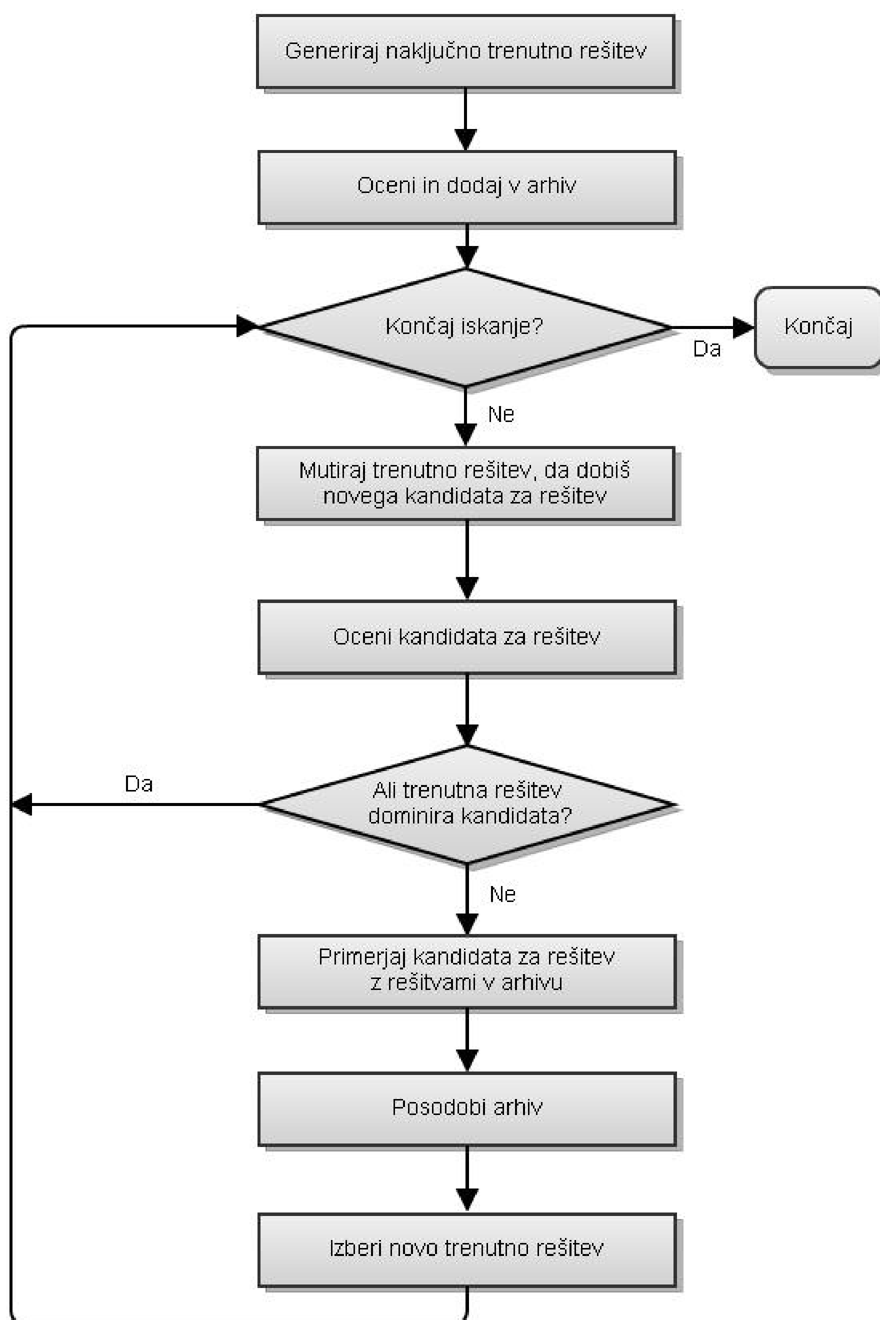
Velikost arhiva je omejena v odvisnosti od tega, koliko končnih rešitev želimo imeti. Vsaki generirani kandidat, ki ni dominiran s strani svojega očeta oziroma trenutne rešitve, se primerja z vsako rešitvijo v arhivu. Kandidati, ki dominirajo referenčno populacijo, so vedno sprejeti v arhiv, medtem ko so dominirani kandidati vedno zavrnjeni. Nedominirani kandidati so arhivirani v odvisnosti od tega, koliko rešitev se že nahaja na njihovi lokaciji v mreži. Kriterijski prostor namreč razdelimo na d-dimenzionalno mrežo, kjer je d število kriterijskih funkcij. Za vsako novo rešitev algoritem poišče in zabeleži njeno lokacijo v mreži. Tako v vsakem trenutku vemo, kje se v mreži nahajajo rešitve iz arhiva. Nedominirani kandidat se pridruži že polnem arhivu samo v primeru, ko si ne deli pozicije v mreži z velikim številom rešitev, in sicer tako, da zamenja eno izmed rešitev, ki se nahajajo v najbolj zgoščenem delu prostora. Isti sistem se uporablja pri izbiri med trenutno rešitvijo in kandidatom, ko kandidat ni dominiran in ne dominira nobene rešitve v arhivu. Procesa arhiviranja in sprejemanja sta prikazana na sliki 11.2.

11.3.3 Prihodnji razvoj

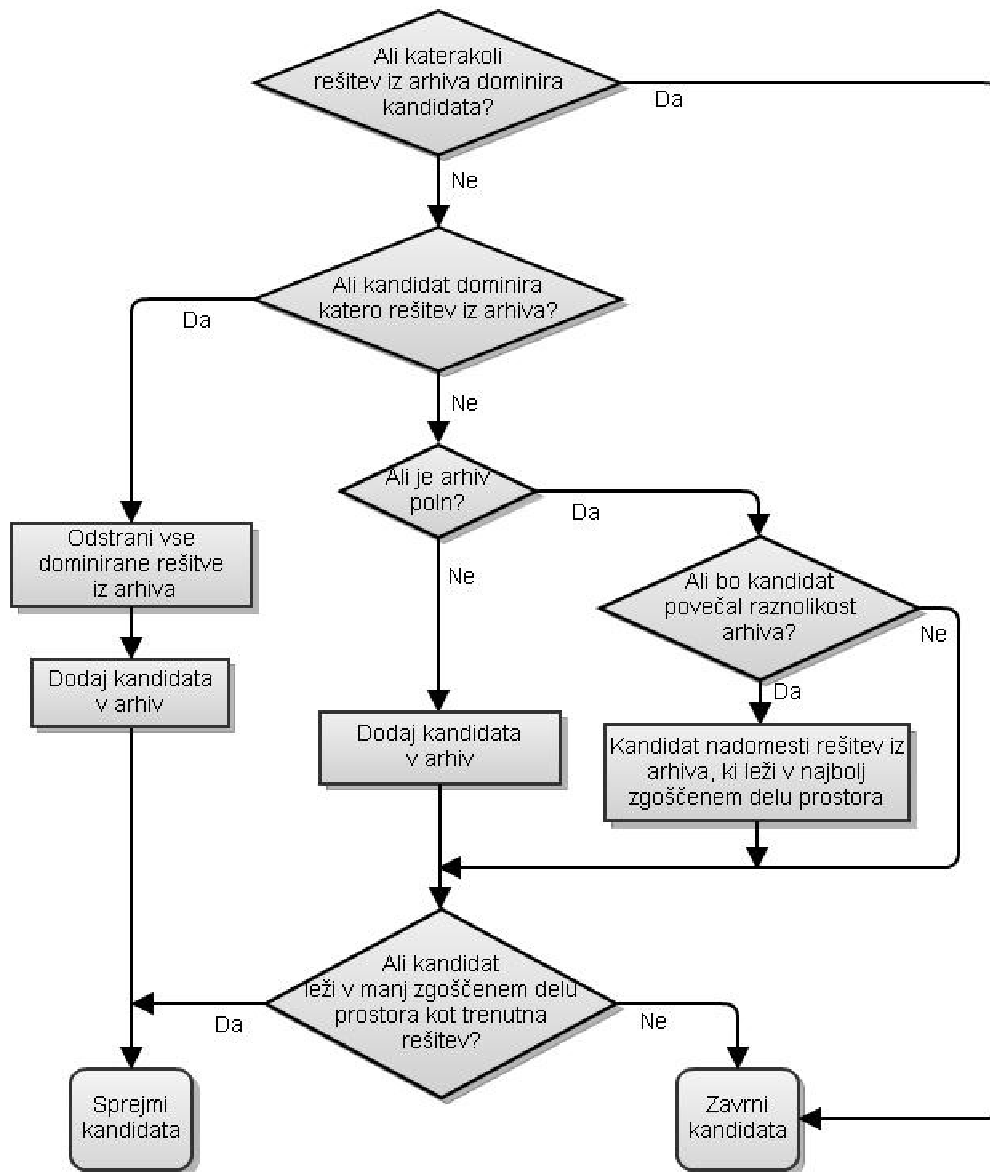
PAES je mogoče identificirati kot $(1 + 1)$ evolucijsko strategijo in zato predstavlja izhodiščni algoritem za večkriterijsko optimizacijo. Avtorja članka [13] sta raziskovala tudi učinkovitost $(1 + \lambda)$ in $(\mu + \lambda)$ različic algoritma. Prvi je identičen PAES algoritmu $(1 + 1)$, s to razliko da iz trenutne rešitve mutira ne enega ampak λ kandidatov. Vsaki izmed λ kandidatov se primerja s trenutno rešitvijo na isti način kot v PAES $(1+1)$ in najboljši od teh zamenja trenutno rešitev. $(\mu+\lambda)$ različica ohranja populacijo velikosti μ, iz katere naredi λ kopij in s pomočjo tekmovalne selekcije izbere najboljše rešitve glede na arhiv, ki jih potem mutira. Mutirane rešitve se spet primerjajo z rešitvami v arhivu in najboljših μ nadomesti trenutno rešitev.

11.4 Zaključek

Delovanje PAES algoritma je na telekomunikacijskem problemu iz drugega razdelka bilo primerjano z delovanjem alternativnega evolucijskega algoritma za večkriterijsko optimizacijo NPGA (*Niched Pareto Genetic Algorithm*). Rezultati kažejo, da je PAES sposoben najti raznolik nabor rešitev primerljivih z rešitvami NPGA in to počne v bistveno krajšem času. Kljub temu, da je v bistvu zelo enostaven algoritem, PAES dobro in konsistentno opravlja svojo nalogo. Zato predstavlja odskočno desko za razvoj bolj zapletenih alternativnih evolucijskih algoritmov za večkriterijsko optimizacijo.



Slika 11.1: Shema PAES algoritma



Slika 11.2: Shematični prikaz arhiviranja in sprejemanja

Poglavje 12

Evolucijski algoritem s pareto močjo

Jernej Zupančič

Evolucijski algoritem s pareto močjo sta razvila E. Zitzler in L. Thiele ter ga leta 1998 opisala v sestavku *An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach*. Hotela sta razviti metodo, ki bo vračala čimbolj raznolike Pareto rešitve, ki bi bile približno enakomerno razporejene po pareto čelu problema.

12.1 Evolucijski algoritem s pareto močjo

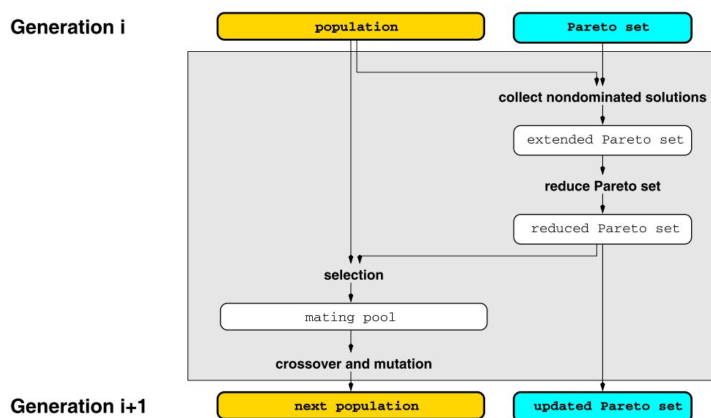
Evolucijski algoritem s pareto močjo oz. Strength Pareto Evloutionary Algorithm (SPEA) je metoda za večkriterijsko optimizacijo. Algoritem uporablja nekaj standardnih metod pa tudi nekaj novih tehnik, da vzporedno poišče več Pareto-optimalnih rešitev. Drugim evolucijskim algoritmom je podoben v tem, da

- posebej shranjuje do sedaj najdene Pareto-optimalne rešitve,
- s pomočjo Pareto dominance določa uspešnost osebkov,
- z grupiranjem zmanjšuje število nedominiranih rešitev in pri tem ohranja karakteristike Pareto-optimalnega čela.

Po drugi strani pa se od vseh drugih algoritmov razlikuje po tem, da

- kombinira vse zgoraj opisane tehnike,
- določa uspešnost osebkov le v odvisnosti od posebej shranjene Pareto množice, dominanca med osebki v populaciji ni ključna,
- vsi osebki v zunanji Pareto množici sodelujejo pri selekciji,

- uporablja novo klasifikacijsko metodo, ki ohranja raznolikost populacije; ta metoda bazira na Pareto dominanci in ni povezana z razdaljami.



Slika 12.1: Prikaz delovanja algoritma.

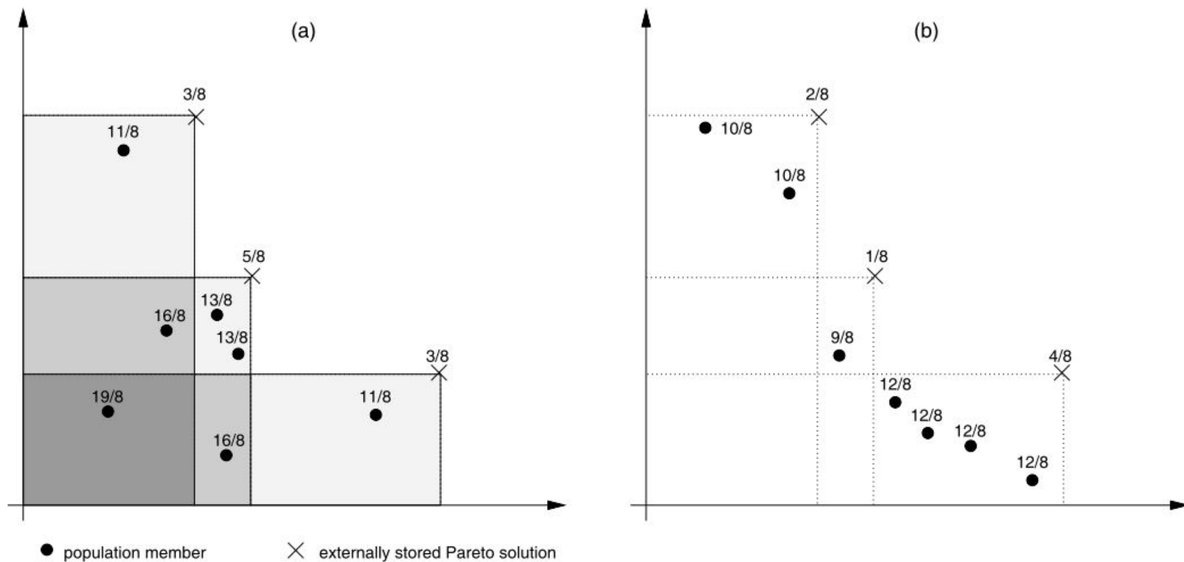
V grobem algoritem deluje na naslednji način. V prvem koraku posodobimo zunanjo Pareto množico - iz populacije poberemo nedominirane osebke, jih prestavimo v Pareto množico in iz le-te pobrišemo dominirane rešitve. Če je osebkov v Pareto množici preveč, z grupiranjem izberemo še dovoljeno število najboljših predstavnikov. Potem ko vsem osebkom določimo uspešnost, iz unije populacije in Pareto množice naključno izbiramo po dva tekmovalca, ki se bojujeta (binarni turnir). Zmagovalec preživi in nastopi pri reprodukciji, poraženec pa izpade iz tekmovanja. Po selekciji na osebkih poženemo še običajno križanje in mutacijo in tako dobimo novo generacijo populacije.

12.2 Funkcija uspešnosti

Kot smo že omenili, SPEA posebej hrani Pareto optimalne rešitve. To pomeni, da imamo v vsakem trenutku in v vsaki generaciji tam shranjene rešitve, ki trenutno dominirajo celotno populacijo. Ker jo hranimo izven populacije, se rešitve pri selekciji ne morejo porazgubiti in velikost populacije ne omejuje števila optimalnih osebkov oz. moči Pareto-množice.

Ta zunanja Pareto množica pa se uporablja še za izračun uspešnosti. Metode, pri katerih je ena populacija osnova za izračun uspešnosti druge populacije, so se razvile iz raziskovanj na področju imunskih sistemov za adaptivno reševanje problemov. Osnovna populacija je ponavadi dosti manjša od druge. Naša naloga je poiskati nove nedominirane rešitve, kjer so osebki ocenjeni glede na število Pareto-točk, ki jih pokrivajo. Za uspešnost osebka bi torej lahko določili kar število Pareto-točk, ki ta osebek pokrivajo. Toda s tem načinom bi izgubili raznolikost optimalnih rešitev, začele bi se zgoščevati.

V nasprotju z drugimi evolucijskimi algoritmi, ki problem raznolikosti rešujejo s pomočjo metrike na genotipih ali fenotipih, SPEA sloni na Pareto-dominanci in deluje tako, da porazdeli



Slika 12.2: Primer izračuna fitnesa osebkov.

osebke populacije čimbolj enakomerno okrog Pareto čela, tako da vsaka Pareto-reštev pokriva približno enako število osebkov iz populacije. Izračun je sledeč. Uspešnost osebkov iz Pareto množice je določen z obratno vrednostjo števila osebkov iz populacije, čemur prištejemo 1, da v imenovalcu nikoli ne dobimo ničle. Temu rečemo tudi *moč* Pareto rešitve. Uspešnost osebka iz dominirane populacije pa je vsota moči Pareto rešitev, ki ta osebek pokrivajo, čemur prištejemo še 1. Tako zagotovimo, da se bodo Pareto rešitve verjetneje reproducirale.

```
def FitnesFunkcija(populacija,paretoMnozica):
  (*posodobi zunanjo pareto mnozico*)
  A=IzberiNedominiraneRešitve(populacija)
  B=ZdružiParetoMnožici(paretoMnožica,A)
  IF |B|>maxParetoTock THEN:
    paretoMnožica=ReducirajParetoMnožico(B)
  ELSE:
    paretoMnožica=B
  (*računaj Pareto moč*)
  FOR paretoOsebek IN paretoMnožica:
    števec=0
    FOR popOsebek IN populacija:
      IF Pokriva(paretoOsebek,popOsebek):
        števec=števec+1
    moč=števec/(|populacija|+1)
    NastaviFitnes(paretoOsebek,moč)
  (*določi fitnes*)
  FOR popOsebek IN populacija:
```

```
vsota:=0
FOR paretoOsebek IN paretoMnožica:
  IF Pokriva(paretoOsebek,popOsebek):
    vsota=vsota + paretoOsebek.fitnes
NastaviFitnes(popOsebek,vsota+1)
```

Potem ko določimo uspešnost osebkov, je na vrsti selekcija, ki poteka po metodi binarnih turnirjev. V vsakem koraku naključno izberemo osebka iz unije populacije in Pareto množice. Osebek z manjšo uspešnostjo izpade iz tekmovanj (ga brišemo iz unije), kopijo zmagovalca pa si shranimo v posebni množici (*mating pool*). Po enem turnirju se torej moč unije zmanjša za ena. To ponavljamo toliko časa, dokler v uniji ne ostane samo še en osebek, tega na koncu premaknemo v *mating pool*. S tem smo dobili selekcijo osebkov, skupaj jih je toliko, kot je bila moč začetne unije populacije in Pareto-množice, vendar se lahko uspešnejši in srečnejši osebki pojavijo v več kopijah. Sedaj v selekciji samo še izvedemo križanje in mutacijo, da dobimo novo generacijo.

12.3 Zmanjševanje Pareto množice z grupiranjem

Pri nekaterih problemih je lahko Pareto-optimalna množica zelo velika, lahko celo neskončna. Rezanje nedominiranih rešitev je včasih nujno zaradi naslednjih razlogov.

- Ko hoče nekdo izbrati neko optimalno rešitev, je naštevanje vseh Pareto rešitev lahko nesmiselno, če njihovo število presega neke razumne meje.
- Lahko nam zmanjkuje računalniškega spomina, kamor bi hranili vse optimalne rešitve (pri zveznih Pareto čelih).
- Velikost zunanje Pareto množice lahko nekajkrat preseže velikost populacije. V tem primeru bo veliko Pareto rešitev enakovrednih (pokrivale bodo točno iste osebke) pa še računanje se lahko močno upočasni.
- Če osebki zunanje Pareto množice niso enakomerno razporejeni po Pareto čelu, lahko pride do velikih odstopanj pri tem, koliko osebkov iz populacije pokriva posamezna Pareto rešitev. Spomnimo se, hočemo tako Pareto množico, ki bo enakomerno razporejena po celotnem Pareto čelu in vsak osebek naj bi pokrival približno enako število osebkov iz populacije.

Obstaja več poskusov za zmanjšanje Pareto množice; toda mi hočemo tak postopek, ki nam bo vrnil reprezentativen vzorec želene velikosti, torej bo čimbolj ohranjal lastnosti velike Pareto množice. Tu nam na pomoč priskoči analiza grupiranja. V splošnem grupiranje razbije

p-elementno množico na q množic, ki pa so dokaj homogene, elementi v teh manjših množicah so si podobni. SPEA uporablja *povprečno povezovalno metodo* predstavljeno v naslednjem algoritmu.

```
def ReducirajParetoMnožico(paretoMnožica):
  (*inicilizacija: vsaka pareto množica tvori svojo gručo*)
  množicaGruč=[]
  FOR paretoOsebek IN paretoMnožica:
    množicaGruč=množicaGruč + [[paretoOsebek]]
  (*združuj gruče dokler ne prideš do sprejemljivega števila gruč*)
  WHILE |množicaGruč|>maxParetoTočk:
    (*izberi najbližji gruči*)
    minRazdalja=Infinity
    FOR [X,Y] PODSEZNAM V množicaGruč:
      IF RazdaljaMedGručami(X,Y)<minRazdalja:
        gruča1=X
        gruča2=Y
        minRazdalja=RazdaljaMedGručami(X,Y)
    novaGruča=gruča1 + gruča2
    množicaGruč=množicaGruč - [gruča1,gruča2]
    množicaGruč=množicaGruč + novaGruča
  (*izberimo reprezentativno rešitev iz vsake gruče*)
  paretoMnožica=[]
  FOR gruča IN množicaGruč:
    paretoOsebek=IščiCentroid(gruča)
    paretoMnožica=paretoMnožica + [paretoOsebek]
```

V začetku vsak osebek Pareto množice tvori eno gručo. V vsakem koraku se gruči, kjer je povprečna razdalja med osebki najmanjša, združita. Število gruč se tako zmanjšuje, dokler jih ne ostane le želeno število. Nato v vsaki gruči izberemo reprezentativen osebek, centroid, pri katerem je povprečna razdalja do vseh drugih osebkov v gruči najmanjša, in tvorimo novo Pareto-množico.

12.4 Implementacija algoritma

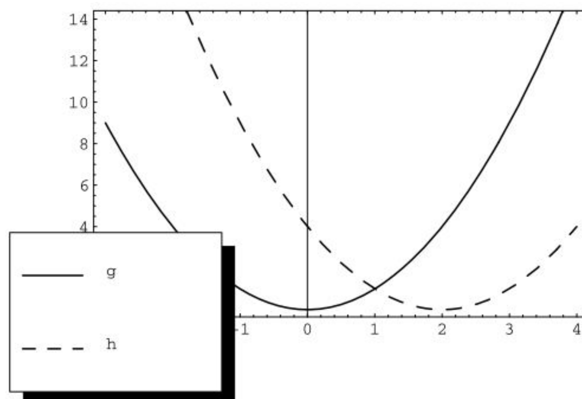
V želji, da se bolje spoznam z algoritmom, sem se ga odločil tudi sprogramirati v programskem jeziku Python. Zavedam se, da ta jezik najbrž ni najboljša izbira glede hitrosti izvajanja, je pa verjetno najbolj enostaven za uporabo. Vse glavne funkcije so že dobro opisane v samem članku. Algoritem sem implementiral tako, da sprejme več funkcij ene spremenljivke in na nekem območju vrne nekaj osebkov, ki dobro aproksimirajo pareto čelo. Potem ko določimo

natančnost, lahko števila na območju zakodiramo v bitni zapis. Vsako število lahko tako predstavimo kot nekaj ničel in enic. Tako kodiranje je v evolucijskih algoritmih zelo uporabno, saj številne operacije, kot npr. mutacijo in križanje, na bitnih besedah enostavno izvajamo.

Za testiranje sem si izbral Schafferjevo f_2 funkcijo, ki jo je Schaffer uporabil v svoji disertaciji. Skoraj vsako metodo za večkriterijsko optimizacijo preizskusijo na f_2, ker je enostavna in pokaže, kako metoda porazdeli osebke po Pareto čelu. Schafferjeva funkcija je definirana sledeče:

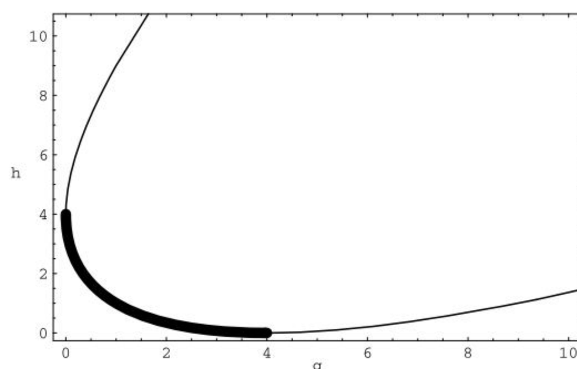
$$f_2(x) = (g(x), h(x)) = (x^2, (x-2)^2).$$

Naš cilj je hkrati minimizirati tako prvo, kot drugo komponento funkcije. Očitno Pareto čelo sestavljajo $x \in [0, 2]$. Zunaj tega intervala obe funkciji naraščata, znotraj pa ena narašča, medtem ko druga pada.



Slika 12.3: Funkciji g in h.

Iz parametrične slike je razvidno tudi iskano Pareto čelo.

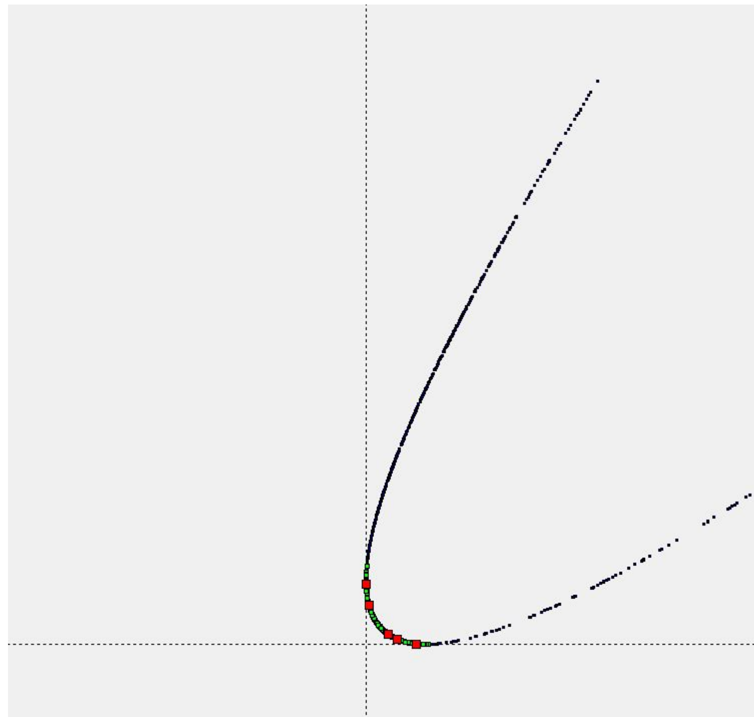


Slika 12.4: Graf f_2 in odebeljeno Pareto čelo.

V algoritmu smo uporabili naslednje parametre.

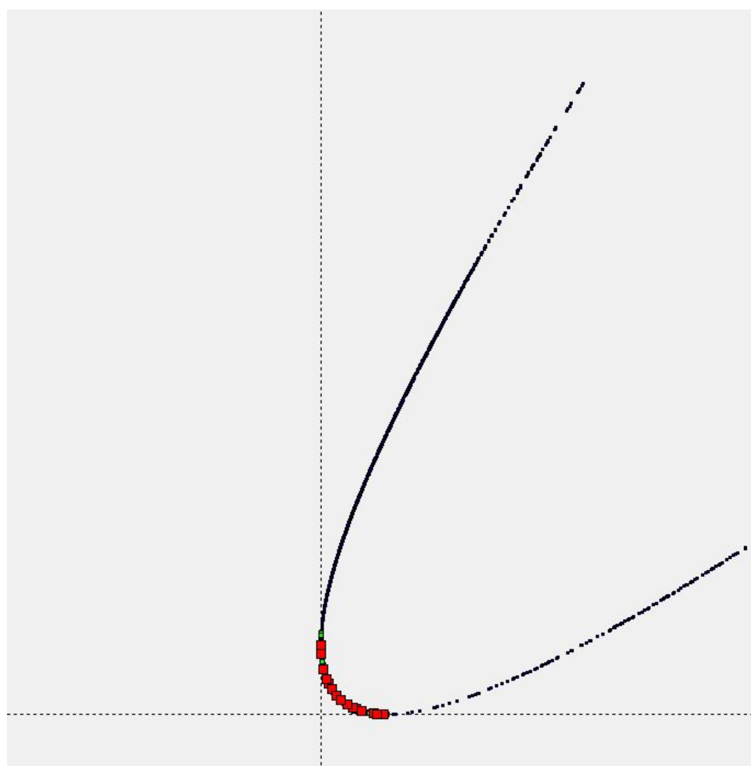
- Natančnost: 0.001 na intervalu $[-6, 6]$, kar je pomenilo 14-bitno kodiranje.
- Velikost populacije: 45/35/25.

- Velikost Pareto množice: 5/15/5.
- Verjetnost križanja: 0.01.
- Verjetnost mutacije: $\frac{1}{14}$.
- Število generacij: 50.

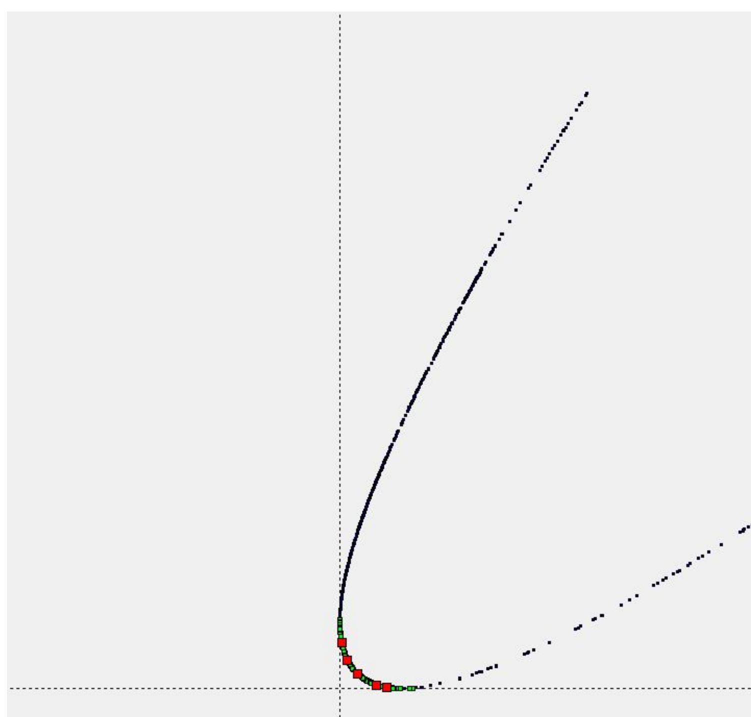


Slika 12.5: Primer za 45 osebkov v populaciji in 5 v Pareto množici.

Seveda sem uporabljal tudi druge parametre. Ugotovil sem, da SPEA sicer dobro deluje, poišče Pareto čelo in nekaj osebkov, ki jo enakomerno aproksimirajo, vendar se čas izvajanja zelo podaljša, ko večamo število generacij in dovoljeno moč Pareto množice. Po natančni preiskavi kode sem ugotovil, da največ časa porabi grupiranje, torej zmanjševanje prevelikih Pareto množic. Razlog za to je verjetno primerjanje vseh parov osebkov iz te množice, kar je računsko zelo zahtevno. Z uporabo drugačnega načina grupiranja bi verjetno lahko zmanjšal čas delovanja algoritma, vendar tako ne bi uporabljal funkcije opisane v članku.



Slika 12.6: Primer za 35 osebkov v populaciji in 15 v Pareto množici.



Slika 12.7: Primer za 25 osebkov v populaciji in 5 v Pareto množici.

Poglavje 13

Niched Pareto Genetski algoritem za večkriterijsko optimizacijo

LEA LETNAR

13.1 Uvod

Genetski algoritem GA je uporaben skoraj le za enokriterijske probleme. Toda v resničnem svetu GA uporablja ciljne funkcije z večimi kriteriji. GA uporabnik najde funkcijo večih kriterijev, da dobi skalarno funkcijo uspešnosti. Pogosta orodja, kako sestaviti več kriterijev, so omejitve z vhodnimi in kazenskimi funkcijami ter uteženimi linearnimi kombinacijami kriterijskih vrednosti. Toda uteži in penalties so problematični. GA rešitev je zelo občutljiva za majhne spremembe koeficientov kazenske fumcije in uteži.

Nekaj študij se je lotilo drugačnih pristopov večkriterijske optimizacije z GA: uporabimo GA, da najdemo vse možne kompromisne rešitve med konfliktnimi kriteriji. Take rešitve so nedominirane, kar pomeni, da ni nobene druge boljše rešitve. Ta množica rešitev leži na ploskvi, ki jo imenujemo Pareto optimalna fronta. Cilj Pareto GA je najti reprezentativni vzorec rešitev povsod na Pareto fronti.

13.2 Dosedanje delo

Schaffer je v razpravi leta 1984 [28] in v kasnejšem delu [29] za iskanje rešitev večkriterijskih problemov predlagal Vector Evaluated GA (VEGA) . VEGA je ustvaril, da bi našel in ohranil večkratna klasifikacijska pravila v množici, ki zajema problem. VEGA je poskušal doseči ta cilj z zbiranjem deležev naslednje generacije le z uporabo enega od vseh kriterijev. Čeprav je Schaffer poročal o uspehu, so domnevali, da je VEGA sposoben najti le ekstremne točke Paretove fronte, kjer je en kriterij maksimalen, saj nikoli ne izbere med kompromisnimi rešitvami

glede na kriterije. Goldberg je v poročilu zgodovine GA [21], ki vsebuje tudi VEGA, da bi v večkriterijskem problemu premaknili populacijo k Pareto fronti, predlagal nedominirano razvrstitev in selekcijo. Predlagal je tudi neke vrste niching, da GA ne bi skonvergiral k eni točki v fronti. Niching mehanizem, kot razdelitev [23], bi dovolil GA ohraniti posameznike povsod na nedominirajoči fronti.

Fonseca in Fleming [20] ter Horn in Nafpliotis [25] so neodvisno od Schafferjevih dveh predlogov uspešno označili dobljen algoritem kot pretežek. Fonseca in Fleming sta sklenila veliko dobrih kompromisov v štiri-kriterijskem modelu plinske turbine. Horn in Nafpliotis sta se osredotočila na dvo-kriterijski problem, ki bo opisan kasneje.

13.3 Niched Pareto GA

Posebnosti Niched Pareto GA so omejene na izvedbo selekcije za genetski algoritem. Tehnika, ki jo največkrat izvajajo je GA turnirski izbor. V turnirskem izboru je množica posameznikov naključno izbrana iz sedanje populacije in najboljši posameznik te podmnožice je premaknjen v naslednjo populacijo. S prilagoditvijo velikosti turnirja imamo nekaj kontrole nad pritiskom izbora in pri hitrosti konvergence. Najmanjši turnir, turnir velikosti dva, predstavlja počasnejšo kovergenco kot katerikoli drugi večji turnir. Turnirski izbor predvideva, da želimo en odgovor za problem. Po nekaj številu generacij bo populacija skonvegirala k enoti. Da se izognemo kovergenci in ohranimo večkratno optimalno rešitev po Paretu smo spremenili turnirski izbor v dveh smereh. Prvič: dodamo Pareto dominiran turnir. Drugič: ko imamo nedominiran turnir je izvedena delitev in tako določimo zmagovalca.

13.3.1 Pareto dominirajoči turnirji

Dvojna relacija dominacije privede do dvojnega turnirja, v katerem primerjamo dva naključno izbrana posameznika. Če eden dominira drugega, zmaga. Na začetku uporabimo majhen lokalno dominirajoč kriterij, vendar kmalu ugotovimo, da doseže nezadosten dominirajoč pritisk. Preveč dominiranih posameznikov je bilo v naslednji generaciji. Zdi se, da je velikost vzorca dveh premajhna, da ocenjuje posameznikovo dominirajočo razvrstitev.

Ker želimo večji dominirajoči pritisk in večjo kontrolo, izvedemo naslednjo vzorčno shemo. Dva kandidata za izbor in primerjalna množica posameznikov so izbrani naključno iz populacije. Vsakega kandidata primerajamo glede na posameznika iz primerjalne množice. Če je eden izmed kandidatov dominiran s primerjalno množico in drugi ni, je drugi izbran za reprodukcijo. Če ni noben dominiran s primerjalno množico, moramo uporabiti delitev, da izberemo zmagovalca, kot bomo razložili kasneje. Velikost vzorca t_{dom}, velikost primerjalne množice, nam daje kontrolo nad pritiskom izbora, oz. kar imenujemo dominirajoči pritisk. Izvršitev Niched Pareto GA je nekoliko občutljiva na količino dominacije proti delitvi uporabljenega pritiska.

Problem se poveča, če sta oba kandidata na tekoči nedominirajoči fronti, saj ne bo noben

dominiran. Majhen t_{dom} bi lahko pomenil, da celo izven fronte noben ne bo dominiran, in seveda oba bi lahko bila dominirana. Kako je potem izbran zmagovalec v nedoločeni tekmi? Če izberemo zmagovalca naključno, bo genetsko kopičenje povzročilo, da populacija skonvergira k eni regiji Paretove fronte. Da to preprečimo, izvedemo obliko deljenja, ko ni prednosti med posamezniki.

13.3.2 Deljenje na nedominirajoči fronti

Uspešnost deljenja sta predstavila Goldberg in Richardson [23], to je natančno analiziral Deb[19], in uspešno uporabil pri težkih problemih v resničnem svetu. Cilj uspešne delitve je razdelitev populacije med številne maksimume v iskanem prostoru, tako da vsak maksimum dobi delež populacije sorazmerno višini tega maksimuma.

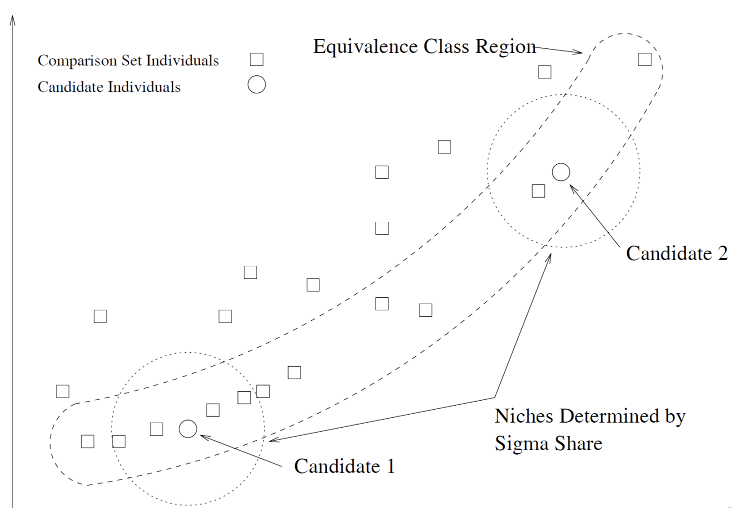
Da dosežemo to razporeditev, delimo sklice za nepravilnost pozameznikove objektivne fitness f_i z niche štetjem m_i izračunano za tega posameznika. Ta degredacija je dobljena z deljenjem objektivne fitness z niche štetjem, da najdemo delitveno fitness $\frac{f_i}{m_i}$. Niche štetje m_i je ocena, kako množična je okolica posameznika i. Izračunana je med vsemimi posamezniki v sedanji populaciji: $m_i = \sum_{j \in Pop} Sh[d[i,j]]$, kjer je $d[i,j]$ razdalja med posameznikoma i in j ter $Sh[d]$ delitvena funkcija. $Sh[d]$ je padajoča funkcija $d[i,j]$, tako da je $Sh[0] = 1$ in $Sh[d \geq \sigma_{share}] = 0$. Običajno uporabimo trikotno delitveno funkcijo, kjer je $Sh[d] = 1 - \frac{d}{\sigma_{share}}$ za $d \leq \sigma_{share}$ in $Sh[d] = 0$ za $d > \sigma_{share}$. Tukaj je σ_{share} niche območje, fiksiran preko uporabnika z neko oceno minimalne delitve, ki je zaželjena ali pričakovana med končnimi rešitvami. Posamezniki znotraj σ_{share} razdalje dokler niso v istem niche. Torej je konvergenca znotraj nicha, toda izognemo se kovergenci celotne populacije. Ko se napolni en niche, njegov niche štetje naraste do točke, kjer je njegova deljena fitness nižja kot pri ostalih niche.

Uspešnost delitve je bila originalno kombinirana s fitnes proporcionalno delitvijo. Ko je delitev kombinirana s turnirsko delitvijo, kakorkoli, niche GA pokaže bolj kaotično obnašanje. Divjim padcem in vzponom v niche podpopulaciji povzročenim z naivno kombinacijo deljenja in turnirske selekcije se lahko izognemo [26]. Oei, Goldberg in Chang [26] so predlagali uporabo turnirske selekcije, v kateri so niche štetja izračunana ne z uporabo sedanje populacije, ampak z nekaj dodane naslednje populacije. To metodo so uspešno uporabili Goldberg, Deb in Horn [22] v niching-težkem problemu. Empirično so tudi našli [26], da je bil vzorec populacije zadosten za oceno niche štetja in izognili so se tudi $\mathcal{O}(n)$ primerjanjem, ki so jih rabili za izračun natanko m_i. V Niched Pareto GA vključujemo obe tehniki.

V katerikoli aplikaciji deljenja lahko izpeljemo genotipsko deljenje, odkar imamo vedno genotip. Toda Debovo delo [19] je pokazalo, da je v splošnem phenotipic deljenje boljše od genotipskega. Intuitivno, želimo predstaviti deljenje v svetu, ki ga imamo raje, to je v nekem phenotipic svetu. Zainteresirani smo v ohranitvi raznolikosti v phenotipic Pareto optimalni fronti, ki obstaja samo v kriterijskem svetu, torej predstavimo delitev v kriterijskem svetu.

Ko sta kandidata bodisi oba dominirana bodisi oba nedominirana, je verjetno da sta oba

v istem ekvivalenčnem razredu. Ker želimo ohraniti raznolikosti znotraj fronte in večina posameznikov v tem ekvivalenčnem razredu, ne izvršimo nobene oblike fitnes degradacije glede na niche štetje. Namesto tega je najboljši fit kandidat določen za kandidata, ki ima najmanjše število posameznikov v njegovem niche, torej najmanjše niche štetje. Tako vrsto deljenja imenujemo delitev ekvivalenčnih razredov.



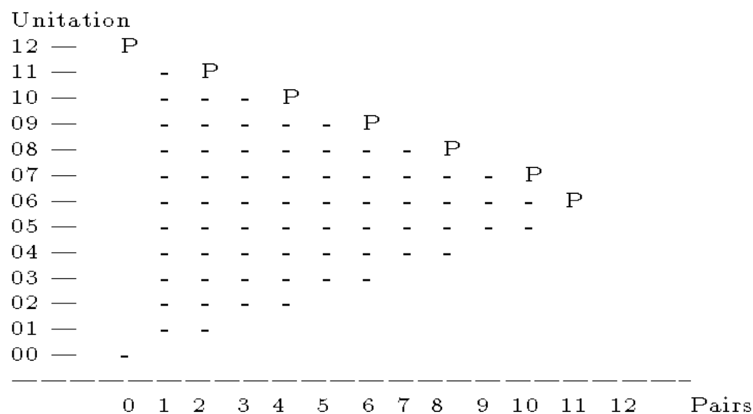
Slika 13.1: Ekvivalenčni razredi delitve

Na sliki 1 je prikazano, kako bi te vrste deljenje delovalo med dvema nedominiramima posameznikoma. Tukaj je maksimum na $x-$osi in minimum na $y-$osi. V tem primeru dva kandidata za delitev nista dominirana s primerjalno množico. Torej ta dva kandidata sta v Pareto optimalni podmnožici unije primerjalne in kandidatov. S stališča Pareta ni noben kandidat bolj zaželjen. Toda, če želimo ohraniti uporabno raznolikost, t.j. reprezentativen vzorec Pareto fronte, je očitno, da bi bilo najbolje izbrati kandidata, ki ima najmanjše niche štetje. V našem primeru je to kandidat 2.

13.4 Aplikacija za tri probleme

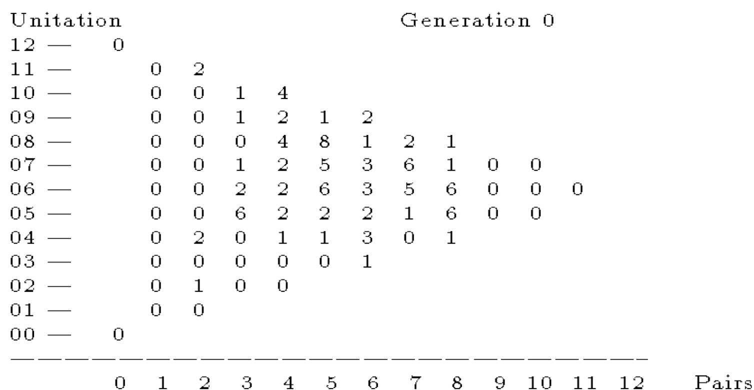
13.4.1 Preprosta testna funkcija

Začnimo testirati naš nov algoritem s konstrukcijo preprostega problema z lahko izračunljivo Pareto optimalno fronto. Problem 1 imenujemo unitacija proti parom, za dva kriterija unitacije in komplementarnih sosednih parov. Unitacija $Unit[s]$ je število enic v fiksirani dolžini bitov stringa s, torej $Unit[01110010] = 4$. Pari $Prs[s]$ je število parov sosednih komplementarnih bitov, torej $Prs[01110010] = 4$. Problem je maksimizirati obe lastnosti. Med stringi posamezne unitacije, tisti stringi z največjim mešanjem enic in ničel določa tiste, ki imajo enke na kupu; 01010 dominira 01100, čeprav imata oba enako število enic.



Slika 13.2: Prvi problem je diskreten z dvo-dimenzionalni, kriterijskim prostorom in z označenimi možnimi$-$ in Pareto P točkami

Na sliki 2 je graf možne regije $2-$dim kriterijskega prostora $Unit$ proti Prs za $12-$bitni problem. P označuje točko na Pareto fronti, medtem ko - označuje možno točko, ki je dominirana z nekaj člani Pareto množice.



Slika 13.3: Distribucija z naključno generirano začetno populacijo

Na sliki 3 smo narisali začetno populacijo, generacijo 0, 100 naključno ustvarjenih $12-$bitnih posameznikov. Označene številke v kriterijskem prostoru so številke posameznikov s kriterijskimi vrednostmi ujemajočih se s koordinatami v kriterijskega prostora.

Na sliki 4 je prikazana populacija po 100 generacijah. Vidimo, da je GA našel vse razen enega člana Pareto množice in zdi se, da ohrani veliko podpopulacijo v vsaki točki. Razen tega, je tukaj nekaj dominiranih posameznikov v sedanji populaciji. Čeprav ni prikazano, smo narisali populacijo razporeditve za veliko generacij in opazili, da GA algoritem ohrani približno enako velikost podpopulacije v vsaki Pareto točki za veliko generacij. Dominirane rešitve se običajno pojavijo, kljub križanju in mutaciji, vendar se ne ohranijo.

Opazovali smo podobna obnašanja veliko zagonov GA na različnih začetnih populacijskih razporeditvah. Uspešno smo se lotili večjih problemov, z ujemajočo se večjo populacijo: 400 posameznikov $28-$bitni problem.

Generation 100

Unitation	0	1	2	3	4	5	6	7	8	9	10	11	12
12 —	26												
11 —		0	11										
10 —		0	0	0	14								
09 —		0	0	0	0	0	12						
08 —		0	0	0	0	0	0	0	16				
07 —		0	0	0	0	0	0	0	0	0	21		
06 —		0	0	0	0	0	0	0	0	0	0	0	
05 —		0	0	0	0	0	0	0	0	0	0		
04 —		0	0	0	0	0	0	0	0				
03 —		0	0	0	0	0	0						
02 —		0	0	0	0								
01 —		0	0										
00 —	0												

Pairs

Slika 13.4: Stabilna subpopulacija na Pareto fronti

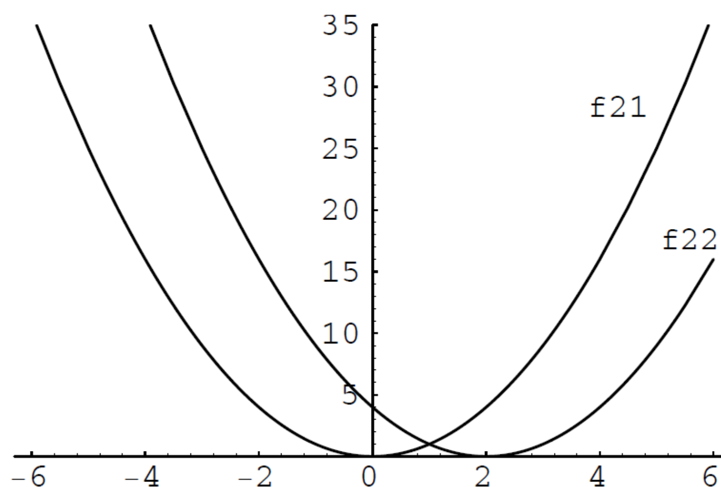
Opazimo, da je ta problem $GA-$lahek, da je lahko najti točke na fronti, toda ni nujno lahek za Niched Pareto GA. Odkar je pravzaprav veliko več rešitev med srednjimi točkami fronte in edino ena ali dve v končnih točkah fronte, naj bi bilo težje ohraniti enako velikost subpopulacije v ekstremnih točkah.

13.4.2 Drugi problem: Schafferjev F_2

Sedaj bomo primerjali naš algoritem z Schafferjevim VEGA z zagonom na testni funkciji iz Schafferjeve razprave [19]. To je preprosta funkcija F_2 z enim parametrom, realno vrednostjo x, in dvema kriterijema, f_{21} in f_{22}, ki ju minimiziramo:

$$f_{21}(x) = x^2,$$

$$f_{22}(x) = (x-2)^2 .$$

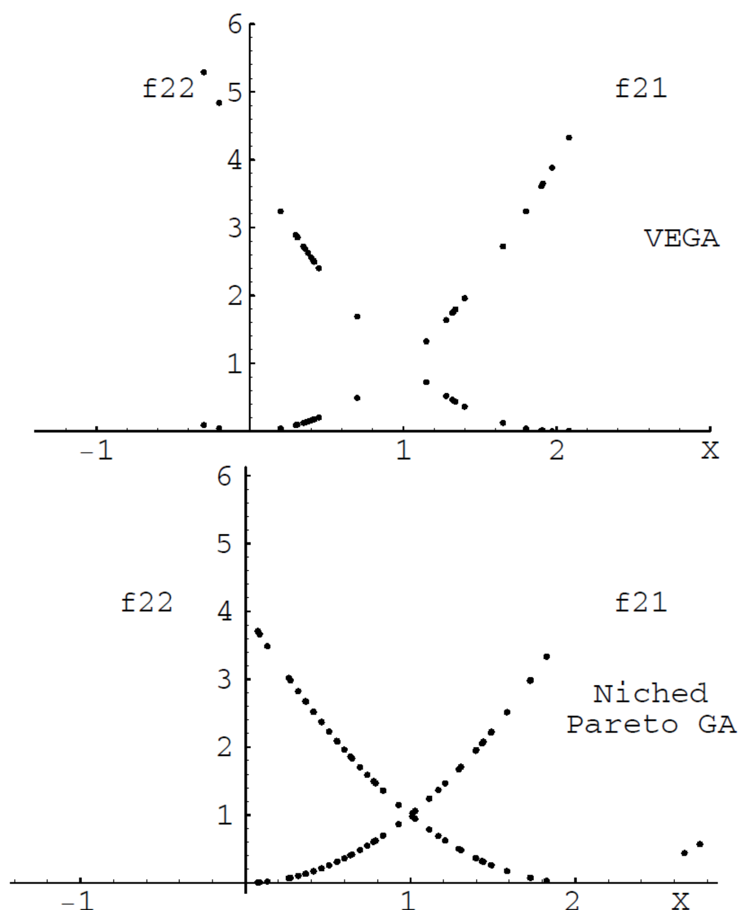


Slika 13.5: Schafferjeva funkcija F_2, $P = \{x;\ 0 \leq x \leq 2\}$

Odločitvena spremenljivka je shranjena na $14-$bitni string kot dvojiško zakodirano število. Torej $000000000000 = x_{min} = -0,6$ in $111111111111 = x_{max} = 6,00$. Na sliki 5 narišemo $f21$ in $f22$. Jasno je, da je Pareto fronta med funkcijama, kjer kompromisna rešitev obstaja. To je za $0 \leq x \leq 2,00$, ena funkcija narašča k najboljšemu, druga pa pada od najboljšega.

Kot Schaffer bomo uporabili majhno populacijo velikosti $N = 30$. Naša niche velikost bo $\sigma_{share} = 0,1$ in turnir velikosti $t_{dom} = 4$. Kot je prikazano na sliki 6 je Niched Pareto GA sposoben ohraniti precej rešitev kljub širjenju na Pareto fronti. Kot teče VEGA zgoraj je tukaj nekaj dominiranih posameznikov populacije, desno od $x = 2$, toda večina posameznikov je fronti. Čeprav ima naša populacija nekaj lukenj v razporeditvi na fronti, so bolj enakomerno razporejeni kot v generaciji 3 VEGA zagona. Najbolj pomembno je to, da Niched Pareto GA zagotavlja stabilnost v razporeditvi populacije za veliko več generacij kot smo pokazali pri VEGA.

F_2 je lahek problem za GA: začetna populacija že vsebuje veliko posameznikov na fronti. Kakorkoli, ta fronta je bolj zgoščena kot v problemu 1 zgoraj, izziv Niched Pareto GA je ohraniti N subpopulacij na fronti velikosti 1.



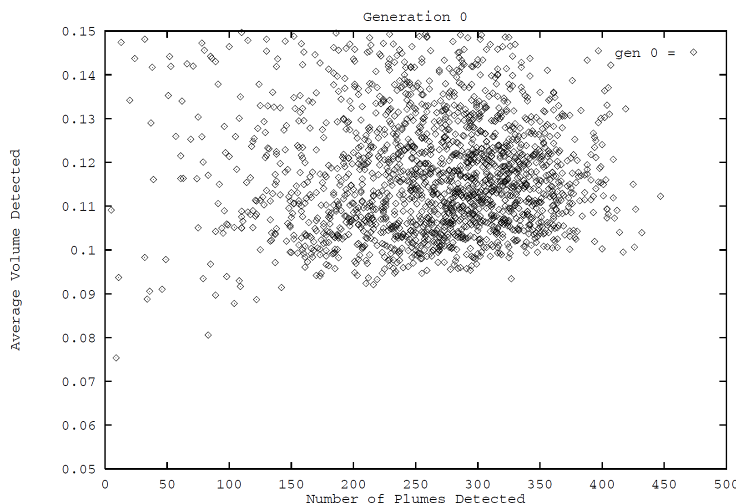
Slika 13.6: VEGA proti Niched Pareto GA. Zgoraj: VEGA na F_2 Pareto fronti, tretja generacija. Spodaj: Niched Pareto GA distribucija, generacija 200.

13.4.3 Odprti problemi v hidrosistemih

Da bi izvali sposobnosti Niched Pareto GA za iskanje raznolikih kriterijskih rešitev, smo za naš tretji testni problem izbrali večjo, resnično, t.j. nerešeno, aplikacijo: optimalno postavitev za zaslonski nadzor podtalne vode. Problem je postaviti k izhodnih od možnih w vodnjakov, tako da maksimiziramo število odkritih puščanj? od polnjenja iz zemlje v obkrožajočo podtalno vodo in miminizirati volumen čiščenja. Ta dva objekta sta v konfliktu. Preprosto optimiranje za volumen čiščenje nam bo dal odgovor s kriterijema $(0, 0)$, kjer ne odkrijemo peres in nimamo volumna onasnaževanja za čiščenje. Če maksimiziramo število odkritih plumes naš volumen za čiščenje zelo naraste.

Pomembno je, da je ta problem neukrotljiv. Iskani prostor je velikosti $\binom{w}{k}$. V našem primeru je $w = 396$ in $k = 20$. Iskani prostor je torej $20269 * 10^{33}$. To je razlog, da ni mogoče vedeti prave Pareto optimalne fronte iz štetja.

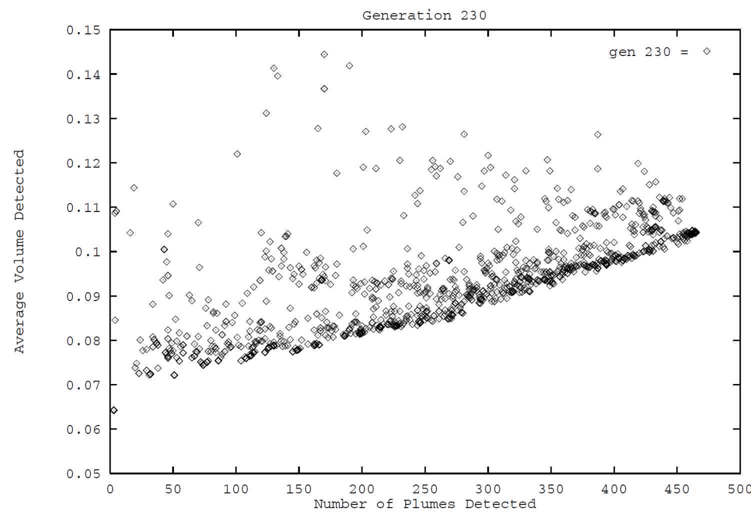
Monte-Carlo simulacija je bila uporabljena za prikaz množice možnih puščanj plumes, množica vodnjakov, ki odkrijejo vsako puščanje in izpuščen volumen, vsakič ko najden vodnjak onesnažuje ??. Ko uporabimo te podatke konstruiramo vektor vrednosti prilagajoče se funkcije, ki vrne število plumes in povprečni najden volumen s katerikoli dano množico vodnjakov.



Slika 13.7: Začetna poulacija distribucije, problem 3

V prvem zagonu je $N = 2000$, $\sigma_{share} = 40$, $t_{dom} = 40$, $p_c = 0,8$ in mutacij ni. Na sliki 7 lahko vidimo, da naključna začetna populacija je zmotena vseskozi iskani prostor. Slika 8 prikazuje, da je Niched Pareto GA čez 230 generacij našel očitno fronto, ki se je izboljšala od začetne populacije. Obljublja, da se bo raznolikost na določeni fronti ohranila tudi čez več generacij. Tukaj je gotovo izboljšanje lokacije fronte in padanja števila dominiranih posameznikov v populaciji.

Še vedno ne vemo, če je to dejansko Pareto optimalna fronta ali sub-optimalna fronta. Toda zagoni pokažejo, da ekvivalenčni razredi delitve in domirajočih turnirjev delujejo skupaj.



Slika 13.8: Končna distribucija, problem 3

Pokazali smo, da kompromisno rešitveno krivuljo lahko z Niched Pareto GA bolje kot naključni vzorec razvijemo do odprtega problema.

13.5 Diskusija

Uvodni rezultati tehnike Niched Pareto so spodbudni. Fonseca in Fleming [20] sta tudi poročala o začetnem uspehu s podobnim algoritmom. Toda ugotovili smo, da je izvedba Niched Pareto GA občutljiva na nastavitev nekaj parametrov. V splošnem je pomembno, da imamo dovolj veliko populacijo, da učinkovito raziskujemo in poskusimo širino Pareto fronte.

Zdi se, da je obnašanje Niched Pareto GA najbolj prizadeto s stopnjo nanesenega pritiska selekcije. Kot velikost turnirja t_{size} vpilva na pritisk selekcije in prezgodnjo konvergenco v GA turnirske selekcije, tako t_{dom} direktno vpliva na konvergenco Niched Pareto GA. Horn in Naflpiotis [25] sta ponazorila prevelik in premajhen vpliv pritiska dominacije. Tukaj povzemimo njune empirično dobljene smernice:

- $t_{dom} \approx 1\%$ od N; rezultati v preveč dominiranih rešitvah
- $t_{dom} \approx 10\%$ od N; majhen odstop in kompletna distribucija
- $t_{dom} \gg 20\%$ od N; povzroči prezgodnjo konvergenco k majhnemu delu fronte. Niso našli kompromisne rešitve.

Nismo se še lotili kritičnega problema iskanja, toda imamo nekaj intuicije. V primeru Pareto optimizacije nam raznolikost tekoče nedominirane fronte v bistvu pomaga pri iskanju novih kompromisnih rešitev, to je razširitev fronte. Posamezniki z različnih delov fronte so mogoče križani za proizvod potomcev, ki dominirajo del fronte, ki leži med starši. To je, informacijo od zelo različnih tipov kriterijskih rešitev lahko kombiniramo, da

Del dokaza za to vidimo v primeru 3. Ker od ekvivalenčnega razreda delitve ne moremo pričakovati, da ohrani več kot eno kopijo posameznika in ker smo uporabili visoko križanje, je bila ohranitev fronte čez sto generacij velika glede na konstantno generacijo in regeneracijo posameznikov na fronti iz križanj dveh različnih staršev. Večina križanj staršev na ali zraven fronte odstopi potomce tudi na ali zraven fronte. To obnašanje je dokaz, da Pareto raznolikost pomaga Pareto iskanju.

Končno izpostavimo, da dominirani turnirji ne zanašajo na dominirano relacijo, toda na asimetrično, tranzitivno relacijo. Ekvivalenčni razredi delitve niso uporabni samo v Pareto optimalni fronti, toda v kateremkoli ekvivalenčnem razredu delne ureditve. Niched Pareto GA je torej uporaben za iskanje kateregakoli delno urejenega prostora, ne samo takega, ki je induciran s Pareto pristopom k več-objektnim problemom.

Literatura

[1] J. R. Koza and R. Poli, *A genetic programming tutorial*, 2003.

[2] R. Poli and W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*, 2008.

[3] El–G. Talbi, *Metaheuristics: From Design to Implementation*, Wiley 2009.

[4] J. Kennedy, R. Eberhart, *Particle Swarm Optimization*, Proceedings of IEEE International Conference on Neural Networks 1995 1942–1948.

[5] http://www.youtube.com/watch?v=xGGtfoC97dc

[6] http://en.wikipedia.org/wiki/Beecolonyoptimization

[7] http://www.swarmintelligence.org/

[8] http://www.ipvs.uni-stuttgart.de/abteilungen/bv/lehre/...termine/.../Schwarm.pdf

[9] http://web.mysites.ntu.edu.sg/yhlow/public/.../tsp-indin08.pdf

[10] http://www.cleveralgorithms.com/natureinspired/swarm/beesalgorithm.html

[11] E. Zitzler, L. Thiele, *An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach* 1998 10–16.

[12] P. D. Justesen, *Multi-objective Optimization using Evolutionary Algorithms*, Department of Computer Science, University of Aarhus, Denmark 2009.

[13] J. D. Knowles, D. W. Corne, *The Pareto Archived Evolution Strategy: A new Baseline Algorithm for Pareto Multiobjective Optimisation*, Proceedings of the 1999 Congress on Evolutionary Computation 1999, 98–105.

[14] T. Robič, B. Filipič, *Večkriterijska optimizacija z evolucijskimi algoritmi*, Inštitut Jožef Stefan, Odsek za inteligentne sisteme.

[15] P. Stušek, *Biologija človeka za gimnazije*, DZS 2002.

[16] http://ima.ac.uk/papers/greensmith2010.pdf

[17] http://eprints.nottingham.ac.uk/621/1/03intros_ais_tutorial.pdf

[18] http://terri.zone12.com/doc/academic/crossroads/

[19] K. Deb, *Genetic algoritms in multimodal function optimization*, MS thesis TCGA Report No.(89002). University of Alabama.

[20] C. M. Fonseca in P. J. Fleming, *Genetic algoritms for multiobjective optimization: formulation, discussion in generalization*, Proceedings of the Fifth International Conference on Genetic Algoritms. Morgan-Kauffman, 416-–423.

[21] D. E. Goldberg, *Genetic Algoritms in Search, Optimization, and Machine Learning.* Rading, MA: Addison-Weasly, 1989.

[22] D. E. Goldberg, K. Deb in J. Horn, *Massive multimodality, deception, and genetic algoritms*, Parallel Problem Solving From Nature, 3, North- Holland, 1992 37–46.

[23] D. E. Goldberg, J. J. Richardson, *Genetic Algoritms with sharing for multimodal fuction optimization.* Genetic Algoritms and Their Applications: Proceedings of the Second ICGA. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987 41-–49.

[24] J. Horn, *Finite Markov chain analysis of genetic algoritms with niching.* Genetic Algoritms and Their Applications: Proceedings of the Fifth International Conference on Genetic Algoritms. Morgan-Kauffman, 1993 110-–117.

[25] J. Horn, N. Nafpliotis, *Multiobjective optimization using the Niched Pareto genetic algoritm.* IlliGAL Report No.(93005). Illinois Genetic Algoritms Laboratory. University of Illinois at Urbana- Champaing, 1993.

[26] C. K. Oei, D. E. Goldberg in S. J. Chang, *Tournament selection, niching and the preservation of diversity.* IlliGAL Report No.(91011). Illinois Genetic Algoritms Laboratory. University of Illinois at Urbana-Champaing, 1991.

[27] J. P. Richardson, M. R. Palmer, G. Liepinis in M. Hiliard, *Some guidelines for genetic algorithms with penalty functions.* Proceedings of the Third International Conference on Genetic Algoritms. Morgan-Kauffman, 1989 191-–197.

[28] J. D. Schaffer, *Some experiments in machine learining using vector evaluated genetic algorithms.* Unpublished doctoral dissertation, Vanderbilt University, 1984.

[29] J. D. Schaffer, *Multiobjective optimization using with vector evaluated genetic algorithms.* In J. Grefenstette, ed., Proceedings of an International Conference on Genetic Algoritms and their Applications, 1985 93–100.

[30] J. L. Deneubourg, S. Goss, Collective patterns and decision-making, *Ethology, Ecology & Evolution 1* 1989, 295–311.

[31] M. Dorigo, V. Maniezzo, A. Colorni, The Ant System: Optimization by a Colony of Cooperating Agents, *IEEE Transactions on Systems, Man and Cybernetics – Part B* 1996, 26, 29–42.

[32] M. Dorigo, *Optimization, Learning, and Natural Algorithms*, Dip. Elettronica, Politecnico di Milano, Milano, 1992.

[33] V. Maniezzo, Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem, *INFORMS Journal on Computing* 1999, 11(4):358–369.

[34] M. G .C. Resende, C. C. Ribeiro, *Greedy Randomized Adaptive Search Procedures*, To appear in 'State of the Art Handbook in Metaheuristics', Kluwer, 2002.

[35] L. S. Pitsoulis, M. G. C. Resende, *Greedy Randomized Adaptive Search Procedures*, AT&T Labs Research Technical Report, 2001.

[36] C. Blum, A. Roli, *Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison*, ACM Computing Surveys, Vol. 35, No. 3, 2003, pp. 277–279.

[37] F. Glover, G. A. Kochenberger, *Handbook of metaheuristics*, Springer, 2003.

[38] P. J. M. Laarhoven, E. H. L. Aarts, *Simulated annealing: theory and applications*, Springer, 1987.

[39] http://en.wikipedia.org/wiki/Travelling_salesman_problem

[40] W. Michiels, E. H. L. Aarts, J. Korst, *Theoretical aspects of local search*, Springer, 2007.

[41] F. Glover, *Tabu search: A tutorial*, Universitiy of Colorado, 1990.

[42] http://en.wikipedia.org/wiki/Mathematical_optimization

[43] F. Glover, M. Laguna, R. Martí, *Advances in Evolutionary Computation: Theory and Applications*, Springer-Verlag, New York, 2003 519–537.

[44] P. Korošec, J. Šilc, B. Robič(, *Populacijske metode kot oblika metahevristične kombinatorične optimizacije*, Elektrotehniški vestnik, 2005.