Nelder-Meadov algoritam s heuristikama

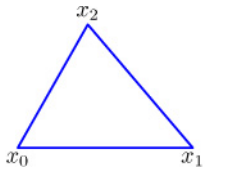
Matija Šantek i Mateo Dujić, Sveučilište u Zagrebu, PMF Matematički odsjek

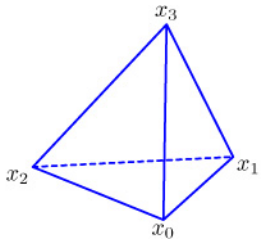
mentor: doc. dr. sc. Goranka Nogo

1 Opis Nelder-Meadova algoritma

Nelder-Meadov algoritam je dizajniran za rješavanje optimizacijskog problema minimizacije nelinearne funkcije $f\colon \mathbb{R}^n \to \mathbb{R}$. Važno je da metoda koristi samo funkcijske vrijednosti u nekim točkama iz $\mathbb{R}^n$ i ne pokušava računati približnu vrijednost gradijenta na ikojoj od tih točaka.

Cijeli algoritam baziran je na simpleksima. Simplex $S \in \mathbb{R}^n$ definiran je kao konveksna ljuska $n+1$ vrhova $x_0, \ldots, x_n \in \mathbb{R}^n$. Na primjer, simpleks u R^2 je trokut, a u R^3 je tetraedar.





Pretraživanje započinje s $n+1$ točkom koje se smatraju vrhovima radnog simpleksa i pripadajućim funkcijskim vrijednostima u tim točkama $f_j = f(x_j), j = 0, \ldots, n$. Vrhovi početnog simpleksa ne smiju pripadati istoj hiperravnini. Metoda tada vrši niz transformacija radnog simpleksa S, koje su usmjerene *smanjivanju* funkcijskih vrijednosti vrhova. U svakom koraku transformacije su određene računanjem jednog ili više novih testnih vrhova koje se uspoređuju s već izračunatim vrhovima na simpleksu. Proces se završava kada radni simplex S postane dovoljno malen u nekom smislu ili kada funkcijske vrijednosti f_j budu dovoljno blizu u nekom smislu (dano je da su f s kojima radimo neprekidne).

Pseudokod algoritma svodi se na sljedeće:

[ ]:
```
Konstruiraj početni radni simpleks S
Ponavljaj sljedeće korake dok uvjeti zaustavljanja nisu zadovoljeni:
    Izračunaj informacije za uvjete zaustavljanja
    Ako uvjeti zaustavljanja nisu zadovoljeni:
```

```
        transformiraj radni simpleks
Vrati najbolji vrh trenutnog simpleksa S i pripadajuću funkcijsku vrijednost
```

1.1 Transformacije simpleksa

Jedna iteracija Nelder-Meadova algoritma sastoji se od sljedeća tri koraka: 1. Poredak: označimo indekse h, s, l kao najgori, drugi najgori i najbolji vrh, redom. Dakle:

$$f_h = \max_j f_j, \qquad f_s = \max_{j \neq h} f_j, \qquad f_l = \min_{j \neq h} f_j.$$

2. Centroid: Izračunajmo centroid c najbolje strane - to je ona nasuprot najgorem vrhu x_h:

$$c := \frac{1}{n} \sum_{j \neq h} x_j$$

3. Trasformacije: izračunajmo novi radni simpleks iz prethodnog. Prvo, pokušajmo zamijeniti samo najgori vrh x_h boljim vrhom koristeći refleksiju, ekspanziju ili kontrakciju u odnosu na najbolju stranu. Sve testne točke leže na pravcu koji prolazi kroz točke x_h i c te najviše dvije se računaju u jednoj iteraciji. Ako ovo uspije, prihvaćena točka postaje novi vrh radnog simpleksa. Ako ne uspije, skupljamo simpleks prema najboljem vrhu x_l. Samo u ovom slučaju, računamo n vrhova odjednom.

- Transformacije su kontrolirane s 4 parametra:
  - α za refleksiju,
  - β za kontrakciju,
  - γ za ekspanziju i
  - δ za skupljanje.
- Trebaju zadovoljavati sljedeće uvjete:

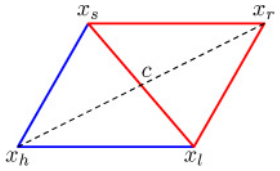
$$\alpha > 0, \quad 0 < \beta < 1, \quad \gamma > 1, \gamma > \alpha, \quad 0 < \delta < 1.$$

- Obično se koristi:

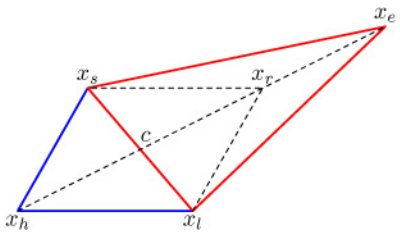
$$\alpha = 1, \quad \beta = \frac{1}{2}, \quad \gamma = 2, \quad \delta = \frac{1}{2}.$$

Transformacije koje vršimo nad simpleksom su:

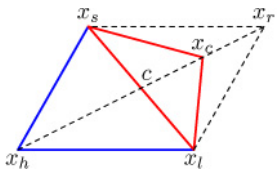
- Refleksija: izračunamo točku refleksije $x_r := x + \alpha(c - x_h)$ i $f_r := f(x_r)$. Ako je $f_l \leq f_r \leq f_s$, prihvatimo x_r i završi iteraciju.



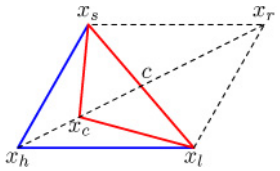
- Ekspanzija: ako $f_r < f_l$, izračunaj točku ekspanzije $x_e := c + \gamma(x_r - c)$ i $f_e := f(x_e)$. Ako $f_e < f_r$, prihvati x_e i završi iteraciju. Inače (ako je $f_e \geq f_r$), prihvati x_r i završi iteraciju.



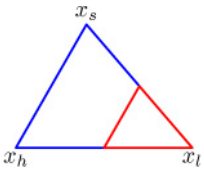
- Kontrakcija: Ako $f_r \geq f_s$, izračunaj točku kontrakcije x_c koristeći bolju od točaka x_h i x_r.
  - Vanjska: Ako $f_s \leq f_r < f_h$ izračunaj $x_c := c + \beta(x_r - c)$ i $f_c := f(x_c)$. Ako $f_c \leq f_r$, prihvati x_c i završi iteraciju. Inače, primijeni skupljanje.



  - Unutarnja: Ako $f_r \geq f_h$, izračunaj $x_c := c + \beta(x_h + c)$ i $f_c := f(x_c)$. Ako $f_c < f_h$, prihvati x_c i završi iteraciju. Inače, primijeni skupljanje.



- Skupljanje: Izračunaj n novih vrhova $x_j := x_l + \delta(x_j - x_l)$ i $f_j := f(x_j)$, za $j = 0, \ldots, n,$, pri čemu $j \neq l$.



S obzirom da mi rješavamo problem optimizacije s ograničenjima, u implementaciju algoritma ćemo pomoću odgovarajuće funkcije evidentirati koliko je zadana točka x unutar pripadajućih ograničenja.

1.2 Naša implementacija Nelder-Meadova algoritma

Isprva uključujemo potrebne Python biblioteke, granice, potrebne vrijednosti za početak algoritma te implementaciju funkcije, primjerice u ovom slučaju funkcija Rosenbrock.

[ ]:
```
import copy
import numpy as np
import math

lowerBound, upperBound = -30, 30
x = np.random.uniform(low=lowerBound, high=upperBound, size=(10,))
N = len(x)
l = [lowerBound for i in range(10)]
u = [upperBound for i in range(10)]


def f(x):
    #Rosenbrock, x_i in [-30, -30]
    Sum = 0
    for i in range(0, N-1):
        Sum = Sum + 100 * ((x[i])**2 - x[i+1])**2 + (x[i] - 1)**2

    return Sum
```

Sljedeća funkcija služi za vraćanje vrhova početnog simpleksa, ovisno o indeksu, vraća vrh po rednom broju u simpleksu.

[ ]:
```
def _e(index,size):
    arr = np.zeros(size)
    arr[index] = 1.0
    return arr
```

S obzirom da funkcije testiramo uz dane granice, funkcija Delta vraća mjeru koliko je koja koordinata van zadanih ograničenja. Uočimo da je $Delta(x) = 0$ ako i samo ako je $x \in [l_1, u_1] \times \cdots \times [l_N, u_N]$.

[ ]:
```
def Delta(x):
    Sum = 0
    for i in range(0,N):
        Sum = Sum + max(x[i]-u[i],0) + max(l[i]-x[i], 0)
    return Sum
```

Funkcija 'better' vraća True ako je 'x' bolja u simpleksu, prvi kriterij je iznos od 'Delta', a drugi kriterij je funkcijska vrijednost. Inače vraća False, ako je 'y' bolja.

[ ]:
```
def better(x,y):
    Dx = Delta(x)
    Dy = Delta(y)
    if(Dx < Dy):
        return True
    elif(Dx == Dy and f(x) < f(y)):
        return True
    else:
        return False
```

Funkcija 'findExtremes' pronalazi indeks najboljeg, najgoreg i drugog najgoreg vrha (s obzirom na funkciju 'better').

[ ]:
```
def findExtremes(s):
    Ds = np.empty([N+1])
    fs = np.empty([N+1])

    for i in range(0, N+1):
        Ds[i] = Delta(s[i])
        fs[i] = f(s[i])

    arg_b = np.argmin(Ds)
    for i in range(0, N+1):
        if Ds[i] == Ds[arg_b] and i != arg_b:
            if(fs[i] < fs[arg_b]):
                arg_b = i;

    arg_w = np.argmax(Ds)
    for i in range(0, N+1):
        if Ds[i] == Ds[arg_w] and i != arg_w:
            if(fs[i] > fs[arg_w]):
                arg_w = i;

    #mask the worst to find the second worst
    Ds[arg_w] = 0

    arg_sw = np.argmax(Ds)
    for i in range(0, N+1):
        if Ds[i] == Ds[arg_sw] and i != arg_w:
            if(fs[i] > fs[arg_sw]):
                arg_sw = i;

    return arg_b,arg_w,arg_sw
```

Centroid, kao što je objašnjeno gore, vraća odgovarajuću točku.

[ ]:
```
def centroid(s,argWorst):
    return np.mean(np.delete(s, argWorst), axis = 0)
```

Zatim, konačno, slijedi implementacija algoritma. Vraćamo najbolji vrh dobiven. Parametri:

- Lambda - parametar za veličinu stranica simpleksa
- M - maksimalan broj iteracija
- Epsilon - uvjet konvergencija za funkciju

[ ]:
```
def SimplexLocalSearch(option, Epsilon, Lambda, M, Alpha=1, Beta=0.5, Gamma=2,␣
↪Delta=0.5):
    #inicijalizacija simpleksa
    N = len(x)
```

```
    s = np.empty([N+1,N])
    s[0] = x
    for j in range(1,N+1):
        s[j] = x + Lambda*(u[j-1]-l[j-1])*_e(j-1,N);
    k = 0

    #određivanje ekstrema u simpleksu
    argBest, argWorst, argSecondWorst = findExtremes(s)
    sBest, sWorst, sSecondWorst = s[argBest], s[argWorst], s[argSecondWorst]

    while abs(f(sBest) - f(sWorst)) > Epsilon and k < M:
        #refleksija
        r = (1+Alpha)*centroid(s,argWorst) - Alpha*sWorst
        if(better(sBest,r) and better(r,sSecondWorst)):
            s[argWorst] = r
        #ako ne refleksija, onda ekspanzija
        elif(better(r,sBest)):
            e = (1+Gamma)*centroid(s,argWorst)-Gamma*sWorst
            if(better(e,r)):
                s[argWorst] = e
            else:
                s[argWorst] = r
        else:
            #ako ne ekspanzija, onda kontrakcija
            c = (1-Beta)*centroid(s,argWorst)+Beta*sWorst
            if(better(c,sWorst)):
                s[argWorst] = c
            else:
                #ovaj dio je bitan za heuristiku usmjereni bijeg
                if(option == 1):
                    return sBest, k, s, argBest, argWorst, argSecondWorst
                #konačno, ako ne ekspanzija, onda skupljanje
                for j in range(0,N+1):
                    s[j] = Delta*(s[j] + sBest)

        argBest, argWorst, argSecondWorst = findExtremes(s)
        sBest, sWorst, sSecondWorst = s[argBest], s[argWorst], s[argSecondWorst]
        k = k + 1

    #ovisno o heuristici, dodamo potrebne povratne argumente
    return sBest, k
```

2 Meta-heuristike za nelinearno programiranje

Kako Nelder-Meadov algoritam funkcionira po principu 'spuštanja niz padinu', rješenja će uglavnom biti u lokalnim optimumima, a ne u globalnim. Kako bismo izbjegli takvo 'zaglavljivanje', predstavljamo nekoliko principa bježanja iz lokalnih optimuma. To su redom:

1. Iterirani slučajni početak
2. Usmjereni bijeg
3. Ne-Tabu pretraživanje
4. Simulirano kaljenje

Ovisno o metoda svaka ima svoje parametre, ali sve metode imaju parametar "Najveći broj iteracija u Nelder-Meadovu algoritmu" kojim zaustavljamo traženje nakon dovoljnog broja iteracija. Parametar se nalazi u uvjetu glavne while petlje N.M. algoritma. Koristimo ga jer nam je iskustvo u traženju funkcija pokazalo da u većini slučajeva algoritam, sam po sebi, ne pronalazi bolja rješenja te ga nema smisla čekati.

2.1 Iterirani slučajni početak

Ova metoda sastoji se od ponovnog pokretanja algoritma od slučajnog rješenja svaki put kada simpleks konvergira po kriteriju ϵ ili kada je dostignut maksimalan broj evaluacija M. U svakoj iteraciji, na slučajan način se izabere točka i simpleks se ponovno izgradi iz te točke. Kad god se najbolje rješenje dodatno popravi, provodi se novo lokalno pretraživanje s kriterijem zaustavljanja ϵ', za profinjenje lokalnog optimuma.

[ ]:
```
def IteratedSimplex(Epsilon, EpsilonApostrophe, Lambda, M):
    k = 0
    while(k < M):
        x = np.random.uniform(low=lowerBound, high=upperBound, size=(10,))
        x, _k = SimplexLocalSearch(x, Epsilon, Lambda, M-k)
        k = k + _k

        try: xBest
        except NameError: xBest = None

        if(xBest is None or f(x) < f(xBest)):
            xBest, _k = SimplexLocalSearch(x, EpsilonApostrophe)
            k = k + _k

        return xBest
```

2.2 Usmjereni bijeg

Još jedna mogućnost za izbjegavanje lokalnih optimuma je sljedeća. Kada simpleks konvergira po kriteriju ϵ, počni širiti simpleks kroz njegov najbolji vrh (stalno ažurirajući poredak vrhova). Ekspanzija će isprva smanjiti kvalitetu točaka, ali nakon određenog broja ponavljanja, dostignut će lokalni 'pessimum' i ekspanzije koje se budu dalje vršile će voditi napretku. Dozvolit ćemo da se ekspanzija vrši dok god se ne popravi najgora točka simpleksa. U toj točki očekujemo da se nalazimo na drugoj strani brda. Dakle, ako algoritam ponovno pokrenemo iz te točke, očekujemo da ćemo doseći drugi lokalni optimum.

[ ]:
```

```

```
def DirectionalEscape(Epsilon,EpsilonApostrophe,Lambda,M, Gamma = 1.25,Beta=-0.
↪5):
    k = 0
    it = 0
    x = np.random.uniform(low=lowerBound, high=upperBound, size=(10,))
    while(k < M):
        temp, _k, s, argBest, argWorst, argSecondWorst =␣
↪SimplexLocalSearch(1,x,Epsilon,Lambda,M-k)
        x = copy.copy(temp)

        k = k + _k
        try: xBest
        except NameError: xBest = None

        #ako je prva iteracija ili smo pronašli novu najbolju točku,␣
↪profinjujemo rješenje
        if xBest is None or f(x) < f(xBest):
            temp, _k, s, argBest, argWorst, argSecondWorst =␣
↪SimplexLocalSearch(2,x,EpsilonApostrophe,Lambda,M-k)
            xBest = np.zeros(N)
            xBest = xBest + temp
            k = k + _k
            print("***NEW BEST VALUE*** iteration: ", k, "| best found value: ",␣
↪f(xBest))

        if xBest is not None and k//1000 > it:
            it = k//1000
            print("iteration:", it * 1000, "| best found value: ", f(xBest))

        argBest, argWorst, argSecondWorst = findExtremes(s)

        xWorst = None

        #dok god se najgora točka ne popravi, nismo na drugoj strani brda
        #ovdje se simpleks stalno ekspandira preko najboljeg vrha dok god se␣
↪najgora točka ne popravi
        while xWorst is None or better(x,xWorst):
            xWorst = copy.copy(x)
            x = Gamma*s[argBest] + (1-Gamma) * centroid(s,argBest)
            s[argBest] = copy.copy(x)
            argBest, argWorst, argSecondWorst = findExtremes(s)

    return xBest
```

2.3 Ne-tabu pretraživanje

Istraživanjem meta-heuristika nad Nelder-Meadovim algoritmom ustanovilo se da će algoritam pronaći dobra rješenja ako pretražujemo područja prethodnih lokalnih optimuma, umjesto da ih izbjegavamo, kao što je to slučaju kod Tabu pretraživanja. Objašnjenje ovog zaključka svodi se na to da lokalni optimumi često znaju biti blizu drugih lokalnih optimuma. Dakle, moglo bi imati smisla pretražiti područje oko prethodnih optimuma, umjesto da ih izbjegavamo. Odatle naziv Ne-tabu pretraživanje. U ovome algoritmu, područje oko lokalnog optimuma se pretražuje pogađanjem slučajnih rješenja u njegovoj blizini i ponovnim pokretanjem pretraživanja iz njih. Parametri:

- σ - zadaje udaljenost od trenutnog baznog rješenja, koristi se za dobivanje novih rješenja
- R - zadaje broj pokušaja pogađanja oko svakog baznog rješenja (nakon R koraka bazno rješenje se mijenja novim najboljim pronađenim rješenjem) Napomenimo odmah da je teško pogoditi dobre parametre.

[ ]:
```
def NonTabuSearch(Epsilon,EpsilonApostrophe,Lambda,M, Gamma = 2,Beta=-0.5, R =␣
↪10, Sigma = 10e-20):
    k = 0
    it = 0
    x = np.random.uniform(low=lowerBound, high=upperBound, size=(10,))
    x, _k = SimplexLocalSearch(x,Epsilon,Lambda,M-k)
    k = k + _k
    xBest = copy.copy(x)

    y = copy.copy(x)


    while(k < M):
        #pogađamo R rješenja u Sigma okolini baznih rješenja
        for i in range(1, R):
            for j in range(1,N):
                x[j] = y[j] + Sigma*(u[j] - l[j])

            x, _k = SimplexLocalSearch(x,Epsilon,Lambda,M-k)
            k = k + _k

            try: xBest
            except NameError: xBest = None

            if xBest is None or f(x) < f(xBest):
                print("***NEW BEST VALUE*** iteration: ", k, "| best found value:
↪ ", f(x))
                xBest, _k = SimplexLocalSearch(x,EpsilonApostrophe,Lambda,M-k)
                k = k + _k

            try: xWorst
            except NameError: xWorst = None
```

```
            if xWorst is None or f(x) < f(xWorst):
                xWorst = copy.copy(x)

        y = copy.copy(xWorst)
        if( xBest is not None and k//1000 > it):
            it = k//1000
            print("iteration:", it * 1000, "| best found value: ", f(xBest))

    return xBest
```

2.4 Simulirano kaljenje

Općenito je poznat princip simuliranog kaljenja, u ovome slučaju svodi se na to da prilikom početne veće temperature, s većom vjerojatnosti prihvaćamo lošija rješenja u odnosu na slučaj kada je temperatura niža pa je manja vjerojatnost da prihvatimo lošija rješenja. Svakako, ako u koraku pronađemo rješenje koje je bolje nego korak prije, na njemu vršimo Nelder-Meadov algoritam. Rješenja pronalazimo u okolini radijusa z kojeg smanjujemo ako pronađemo lošije rješenje, a povećavamo ako pronađemo bolje rješenje.

[ ]:
```
def SimulatedAnnealing(Epsilon, Lambda, T_max = N, T_min=0, beta=1, mu=3, nu=1,␣
↪tau = 1.5,
                       alpha = 0.5):
    #generiramo početnu točku
    x_0 = np.random.uniform(low=lowerBound, high=upperBound, size=(10,))

    scores = list()

    x = copy.copy(x_0)
    scores.append(x_0)
    T = T_max

    #postavljamo radijus
    z_max = (upperBound - lowerBound)/2
    z_min = (upperBound - lowerBound)/50
    z_0 = (z_max + z_min)/2
    z = z_0

    #postavljamo maksimalan broj iteracija za nelder-meadov algoritam
    N_max = 500 * N



    while(T > T_min):
        k = 0
        print(T, f(scores[-1]))
        while(k <= mu):
```

```
            #generiranje susjednih rješenja
            l = list()
            for i in range(0,N):
                start = random.randint(0, N/nu-1)
                y = copy.copy(x)
                for j in range(0,nu):
                    y[start*nu+j] = np.random.uniform(low = y[start*nu+j]-z,
                                                      high = y[start*nu+j]+z,
                                                      size = (1,) )
                l.append(y)

            #pronađi najbolji y
            x_apostrophe = copy.copy(l[0])
            for i in range(1,N):
                if(f(l[i]) < f(x_apostrophe)):
                    x_apostrophe = copy.copy(l[i])


            delta_E = f(x_apostrophe) - f(scores[-1])

            #ako je rješenje bolje, tada je delta_E < 0 pa to rješenje odmah␣
↪prihvaćamo
            if delta_E < 0:
                x = copy.
↪copy(SimplexLocalSearch(500,x_apostrophe,Epsilon,Lambda))
                scores.append(x)
                z = tau * z
            else:
                z = alpha * z
                if(rand() < exp(-delta_E / T)):
                    #gore rješenje će vjerojatnije biti izabrano što je veća␣
↪temperatura
                    x = copy.
↪copy(SimplexLocalSearch(500,x_apostrophe,Epsilon,Lambda))
                    scores.append(x)
                    x = copy.copy(x)
            k = k + 1
        T = T - beta

    #još na kraju primijenimo nelder-meada na najbolje rješenje
    minimum = copy.copy(x)
    for i in range(0,len(scores)):
        if(f(scores[i]) < f(minimum)):
            minimum = copy.copy(scores[i])

    x = SimplexLocalSearch(N_max,minimum,Epsilon,Lambda)
    scores.append(x)
```

```
    evaluations = list()
    for i in range(0,len(scores)):
        evaluations.append(f(scores[i]))
        if(f(scores[i]) < f(minimum)):
            minimum = copy.copy(scores[i])

    return [minimum, f(minimum), evaluations]
```

3 Funkcije

Funkcije nad kojima vršimo pretraživanje, s pripadajućim ograničenjima su:

1. Sferna funkcija
   - najbolje rješenje: $f^* = 0$, $N = 10$

$$f(x) = \sum_{1}^{N} x_i^2, \quad x_i \in [-30, 30], \quad i = 1, \ldots, N$$

2. Rosenbrock
   - najbolje rješenje: $f^* = 0$, $N = 10$

$$f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{N} x_i^2}\right) - \exp\left(\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi x_i)\right) + 20 + e, \quad x_i \in [-30, 30], \quad i = 1, \ldots, N$$

3. Ackley
   - najbolje rješenje: $f^* = 0$, $N = 10$

$$f(x) = \sum_{i=1}^{N-1} \left[100(x_i^2 - x_{i+1}^2)^2 - (x_i - 1)^2\right], \quad x_i \in [-5, 10], \quad i = 1, \ldots, N$$

4. Griewank
   - $d = 4000$
   - najbolje rješenje: $f^* = 0$, $N = 10$

$$f(x) = \frac{1}{d}\sum_{i=1}^{N}(x_i - 100)^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i - 100}{\sqrt{i}}\right) + 1, \quad x_i \in [-600, 600], \quad i = 1, \ldots, N$$

5. Michalewicz
   - $m = 10$
   - najbolje rješenje: $f^* = -9.6601517\ldots$, $N = 10$

$$f(x) = -\sum_{i=1}^{N} \sin(x_i) \cdot \sin^{2m}\left(\frac{i \cdot x_i^2}{\pi}\right), \quad x_i \in [0, \pi], \quad i = 1, \ldots, N$$

6. Shekel

- $m = 5, 7, 10$
- najbolja rješenja redom po m: $f^* = -10.1532$, $f^* = -10.4029$, $f^* = -10.5364$, $N = 4$

$$f(x) = -\sum_{i=1}^{m} \left(\frac{1}{\sum_{j=1}^{N} (x_j - C_{ji})^2 + \beta_i} \right), \quad x_i \in [0, 10], \quad i = 1, \ldots, N$$

7. Langermann
   - $m = 5$
   - najbolje rješenje: $f^* = -5.1621259$, $N = 2$

$$f(x) = -\sum_{i=1}^{m} \frac{c_i \cos\left(\pi(x_1 - a_{i1})^2 + (x_2 - a_{i2})^2\right)}{e^{\frac{(x_1 - a_{i1})^2 + (x_2 - a_{i2})^2}{\pi}}}, \quad x_i \in [0, 10], \quad i = 1, \ldots, N$$

4 Rezultati

4.1 Obični Nelder-Meadov algoritam

Funkcija	Prosječna nađena vrijednost	Najbolja nađena vrijednost	Lambda	Najviše it. u N-M
Sferna	9.3772e-48	0.0	1	100000
Rosenbrock	2972616.825944823	0.1100	0.1	100000
Ackley	1.4103	3.5527e-15	10	100000
Griewank	2.1931	0.6378	1	100000
Michalewicz	-3.5838	-5.3661	10	100000
Shekel	-6.4879	-10.5364	10	100000
Langermann	-1.6914	-4.0614	10	100000

4.2 Iterirani slučajni početak

Funkcija	Prosječna nađena vrijednost	Najbolja nađena vrijednost	Lambda	Najviše it. u N-M
Sferna	2.5e-323	0.0	1	100000
Rosenbrock	781414.9711	1.737e-17	1	100000
Ackley	0.2718	3.5527e-15	10	100000
Griewank	1.4187	2.8755e-14	1	100000
Michalewicz	-4.5594	-6.3828	10	100000
Shekel	-9.9228	-10.5364	10	100000
Langermann	-1.9287	-4.1276	10	100000

4.3 Usmjereni bijeg

Funkcija	Prosječna nađena vrijednost	Najbolja nađena vrijednost	Lambda	Najviše it. u N-M
Sferna	9.3402e-42	0.0	0.1	100000
Rosenbrock	0.0	0.0	10	1000
Ackley	2.7711e-15	0.0	10	1000
Griewank	0.0683	0.0	0.1	5000
Michalewicz	-5.8197	-7.4364	10	1000
Shekel	-10.1498	-10.5364	10	1000
Langermann	-2.6621	-4.1223	10	1000

4.4 Ne-tabu pretraživanje

Funkcija	Prosječna nađena vrijednost	Najbolja nađena vrijednost	Lambda	Sigma	R	Najviše it. u N-M
Sferna	7.6601e-11	4.2881e-20	10	1	2	1000
Rosenbrock	508.6913	6.7596e-20	10	1	2	1000
Ackley	2.4825e-11	3.5527e-15	10	1	2	1000
Griewank	1.7222e-05	0.0	0.1	1	2	1000
Michalewicz	-6.1145	-7.4298	10	0.1	10	1000
Shekel	-10.5362	-10.5364	10	1	2	1000
Langermann	-3.8754	-4.1556	10	0.1	2	1000

4.5 Simulirano kaljenje

Funkcija	Prosječna nađena vrijednost	Najbolja nađena vrijednost	Lambda	Temp	Najviše it. u N-M
Sferna	5.0515e-75	5.4294e-192	10	100	100000
Rosenbrock	4.0122e-17	0.0	10	100	1000
Ackley	2.0322e-14	3.5527e-15	10	100	1000
Griewank	0.0273	0.0	0.1	10	5000
Michalewicz	-8.5811	-9.5785	10	100	1000
Shekel	-7.9302	-10.5364	10	10	5000
Langermann	-3.1604	-4.1557	10	10	5000

5 Analiza

Rezultate ne prikazujemo grafički, jer svakako ne možemo vidjeti golim okom razliku između reda veličine 10^{-10} i recimo 10^{-75}. Analizirajmo metode redom po funkcijama:

1. Sferna

Sve osim ne-tabu i simuliranog kaljenja našle su u nekom trenutku najbolje rješenje, ali prosječno najbolje su bile iterirani slučajni početak i simulirano kaljenje. Za ovu funkciju je zato najbolje rezultate pokazao iterirani slučajni početak. Iskustvo testiranja je pokazalo da je najbolje za ovu

funkciju pustiti da se Nelder Meadov algoritam izvrši do kraja jer ona nema lokalnih minimuma (osim globalnog u nuli koji je rješenje).

2. Rosenbrock

Otprilike pola funkcija je pronašlo najbolje rješenje, međutim pogotovo uspješnim pokazao se usmjereni bijeg koji je u svakom mogućem pokretanju pronašao najbolje rješenje. S druge strane, većina ostalih funkcija imali su jako veliku prosječnu vrijednost. Najgorim se pokazao obični Nelder-Meadov algoritam i po ovoj funkciji vidimo korist heuristika u odnosu na obični algoritam.

3. Ackley

Sve metode, osim usmjerenog bijega pronašle su isto najbolje rješenje, $3.5527 \cdot 10^{-15}$. Funkcija Ackley pokazala se kao tvrd orah, koju smo jedino uspjeli riješiti usmjerenim bijegom. Najgorim se opet pokazao obični Nelder-Meadov algoritam.

4. Griewank

Sve metode, osim običnog N.M. i iteriranog slučajnog početka uspjele su u nekom trenutku pronaći najbolje rješenje. Za ovu funkciju, prosječno najboljom pokazalo se ne-tabu pretraživanje, najgorom opet Nelder-Meadov algoritam.

5. Michalewicz

Ova funkcija pokazala se također tvrdim orahom i nijedna metoda nije uspjela pronaći njezin minimum. Najbliže najboljem rješenju bilo je simulirano kaljenje, a ona se pokazala i prosječno najboljom. Obični Nelder-Meadov algoritam opet je bio najgori.

6. Shekel

Ova funkcija bila je u manje dimenzija nego funkcije prije pa su sve metode uspjele pronaći najbolje rješenje. Prosječno najboljim pokazalo se ne-tabu pretraživanje, a najgorim ponovno obični N.M. algoritam.

7. Langermann

Ova funkcija, unatoč što je samo u 2 dimenzije, nije bila lako rješiva u metodama. Nijedna metoda nije pronašla najbolje rješenje. Najbolje rješenje (u odnosu na druge metode) pronašlo je simulirano kaljenje, a prosječno najbolje ne-tabu pretraživanje.

Iz analize možemo zaključiti da su sve heuristike pomogle običnom Nelder-Meadovu algoritmu kod funkcija u kojima je on zapinjao u lokalnim optimumima. Heuristike usmjereni bijeg, simulirano kaljenje i ne-tabu pretraživanje pokazale su se svaka zasebno dobra za neku funkciju, niti jedna nije bila potpuno bolja od svih drugih. Dakle, odlično su se nadopunjavale i pronašle najbolja rješenja kod svih funkcija osim Langermanna.

6 Literatura

[1] Saša Singer and John Nelder *Nelder-Mead algorithm*, (2009)

[2] João Pedro Pedroso, *Simple meta-heuristics using the simplex algorithm for non-linear programming*, (2007)

[3] Ahmed Fouad Ali, *Hybrid Simulated Annealing and Nelder-Mead algorithm for solving large-scale global optimization problems*, (2014)