Minimizări de funcții.
Hill Climbing. Simulated Annealing.

Rotariu Vlad-Anton

October 27, 2022

1 Introducere

Optimizarea matematică înseamnă selecția celui mai bun element dintr-un set de alternative valabile. Problemele de optimizare apar în discipline precum informatică, inginerie, economie.Optimizarea constă in maximizarea sau minimizarea unei funcții reale, prin alegerea sistematică a parametrilor și calcularea funcției.

In acest articol vor fi prezentați doi algoritmi de optimizare: Hill Climbing si Simulated Annealing impreuna cu implementarea lor in Python.

2 Algoritmi

2.1 Hill Climbing

Acest algoritm încearca să minimizeze sau să maximizeze o funcție $f(x)$, unde x este un vector de valori. La fiecare iterație, Hill Climbing va adapta un singur element din x și va determina dacă noul vector x' îmbunătățește valoarea lui $f(x)$, caz în care $x := x'$. Algoritmul se încheie în cazul in care nu se găsește niciun x'. $f(x)$ se va numi **minim local**.

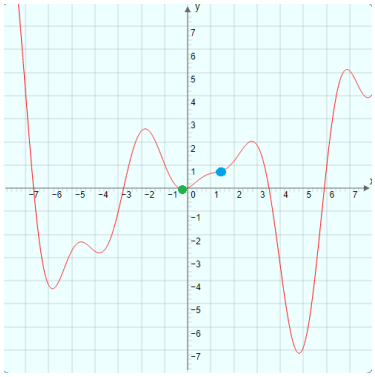
Pseudocod

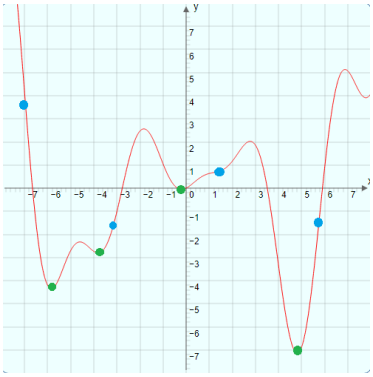
```
t := 0
initialize best
repeat
        local := FALSE
        select a candidate solution vc at random
        evaluate vc
        repeat
                vn := Improve(Neighborhood(vc))
                if eval(vn) is better than eval(vc)
                then vc := vn
```

```
            else local := TRUE
        until local
        t := t + 1
        if vc is better than best
        then best := vc
until t = MAX
```

Se poate observa că pentru a mări spațiul de căutare, algoritmul Hill Climbing este iterat de t ori. Aceste iterații ajuta algoritmul să nu rămână blocat într-un minim local.

In imaginile de mai jos putem observa acest fenomen. Punctele albastre sunt valorile alese de algoritm iar punctele verzi sunt rezultatele unei iteratii. Se observa ca in imaginea din stanga obtinem un minim local, pe cand in a doua imagine algoritmul a facut mai multe iteratii si a reusit sa ajunga chiar in punctul de minim global.





2.2 First improvement și best improvement

În funcție de metoda în care este implementată funcția *Improve*()
există 2 abordări ale algoritmului.

First improvement

Această abordare este o abordare greedy. Primu vecin bun găsit este ales ca un nou candidat.

Best improvement

Această abordare presupune faptul ca toți vecinii sunt testați, iar cel mai bun este ales ca un nou candidat.

2.3 Simulated Annealing

Numele algoritmului reprezintă o metodă folosită in metalurgie
(annealing = călire) ce presupune încălzirea și răcirea controlată a materialelor pentru a mări dimensiunea cristalelor și pentru a reduce defectele. Noțiunea de răcire treptată din algoritm este interpretată ca o scădere treptată a probabilității de a accepta soluții mai slabe. Acceptarea acestor soluții permite o căutare mai largă a optimului global.

Iterația

O iterație a algoritmului presupune:

1. Alegerea unui vecin s' al unei stări s
2. Deciderea probabilistică intre a accepta vecinul sau a păstra starea curentă

Vom alege o funcție $f(x)$. Probabilitatea de acceptare va fi calculată astfel:

$$P = e^{-\frac{|f(s)-f(s')|}{T}}, T > 0$$

unde T este temperatura.

Observăm că P este direct proporțională cu T, deci temperatura mare inseamnă probabilitate mare.

Pseudocod

```
t := 0
initialize the temperature T
select a current candidate solution vc at random
evaluate vc
repeat
        repeat
                select at random vn: a neighbor of vc
                if eval(vn) is better than eval(vc)
                        then vc := vn
                        elseif random[0,1) < exp(-abs(eval(vn) - eval(vc)) / T)
                            then vc := vn
        until (termination-condition)
        T := g(T; t)
        t := t + 1
until (halting-criterion)
```

3 Implementare

3.1 Introducere

Pentru implementarea algoritmilor a fost folosit Python 3.8. Pentru fiecare algoritm a fost creată o clasă aferentă: `HillClimbing` și `SimulatedAnnealing`. În constructorul fiecărei clase sunt inițializați următorii membri: `dimension`(dimensiunea funcției), `limits`(lista ce contine capetele intervalului funcției), `function`(funcția de minimizat), `n`(numărul de iterații), `precision`(precizia), `bits_number`(lungimea necesară reprezentării unei valori ca bitstring).

în plus, `HillClimbing` conține membrul `mode` ce semnifică tipul de algoritm(first sau best).

```
class HillClimbing:

    def __init__(self, dimension, limits, mode, function, n=10000, precision=3):
        self.dimension = dimension
        self.limits = limits
        self.mode = mode
        self.function = function
        self.n = n
        self.precision = precision

        self.bits_number = (limits[1] - limits[0]) * 10 ** precision
        self.bits_number = int(np.ceil(np.log2(self.bits_number)))
```

```
class SimulatedAnnealing:

    def __init__(self, dimension, limits, function, n=200, precision=3):
        self.dimension = dimension
        self.limits = limits
        self.function = function
        self.n = n
        self.precision = precision

        self.bits_number = (limits[1] - limits[0]) * 10 ** precision
        self.bits_number = int(np.ceil(np.log2(self.bits_number)))
```

3.2 Reprezentarea datelor

Pentru a reprezenta candidații pentru optimizări, aceștia vor fi transformați in șiruri de biti(bitstring). Intervalul de căutare [a,b] se va discretiza la o precizie 10^{-d}. Acest interval va fi împărțit în $N = (b-a) \cdot 10^d$ subintervale, deci sunt necesari $n = [\log_2 N]$ biti pentru a reprezenta o valoare din acest interval.

Formula folosită pentru decodificarea bitstringului este:

$$x = a + decimal(bitstring) \cdot \frac{b-a}{2^n - 1}$$

În programul nostru Python vom folosi funcția `generate_candidate()` pentru a genera aleatoriu un bitstring de lungime `dimension` · `bits_number` ce reprezintă vectorul de parametri a funcției de minimizat.

```
def generate_candidate(self):
        result = ''.join([random.choice(['0', '1']) for _ in range(
            0, self.bits_number * self.dimension)])

        return result
```

3.3 Căutarea candidaților

Pentru căutarea noilor candidați(vecinilor), folosim funcția `mutation()`.Funcția presupune schimbarea bitilor cu ajutorul funcției `swap_bit()` pâna când un candidat mai bun este găsit. `mutation()` este implementată in funcție de algoritmul ales.

```
def swap_bit(self, bitstring, bit_pos):
        result = ''
        for i in range(0, len(bitstring)):
            if i == bit_pos:
                result += str(-1 * (int(bitstring[i]) - 1))
            else:
                result += bitstring[i]

        return result
```

De asemenea pentru a compara candidații, este folosită funcția `evaluate()` ce decodifică bitstringul in care se afla parametrii si calculează valoarea funcției din `self.function`.

```
def evaluate(self, bitstring):
        parameters = []
        a = self.limits[0]
        b = self.limits[1]
        n = self.bits_number

        for i in range(0, self.dimension):
            parameter = int(bitstring[n * i: n * (i + 1)], 2)
            parameter = a + (parameter * (b - a) / (2 ** n - 1))
            parameters.append(parameter)

        return self.function(parameters)
```

Hill Climbing

Funcția tratează două cazuri, deoarece `mode` poate fi *first* sau *best*. În primul caz, funcția `swap_bit()` este apelată pană este găsit un rezultat mai bun. În al doilea caz, funcția `swap_bit()` este apelatț pentru fiecare bit din string, iar cel mai bun candidat este returnat.

```
def mutation(self, bitstring):
        initial_score = self.evaluate(bitstring)
        bitstring_length = len(bitstring)

        if self.mode == 'first':
            for i in range(0, bitstring_length):
                bitstring_copy = bitstring
                bitstring_copy = self.swap_bit(bitstring_copy, i)

                if self.evaluate(bitstring_copy) < initial_score:
                    return bitstring_copy

            return bitstring

        elif self.mode == 'best':
            best_position = -1
            best_score = initial_score

            for i in range(0, bitstring_length):
                bitstring_copy = bitstring
                bitstring_copy = self.swap_bit(bitstring_copy, i)

                if self.evaluate(bitstring_copy) < best_score:
                    best_position = i
                    best_score = self.evaluate(bitstring_copy)

            if best_position == -1:
                return bitstring
            else:
                return self.swap_bit(bitstring, best_position)
```

Se observă ca funcția returnează bitstringul primit ca parametru daca nu găsește un ”vecin” mai bun.

Simulated Annealing

În acest caz, `mutation()` schimbă aleatoriu un singur singur bit din bitstring si returnează noul string.

```
def mutation(self, bitstring):
        mutation_bit = random.randint(0, len(bitstring) - 1)
        return self.swap_bit(bitstring, mutation_bit)
```

3.4 Iterația

Hill Climbing

În funcția `HC()`, `search_minima()` este apelată de `self.n` ori. `search_minima()` generează un prim candidat aleatoriu iar apoi caută "vecini" cu ajutorul funcției `mutation()`. Dacă "vecinul" este identic cu candidatul, funcția se oprește. `HC()` returnează cel mai bun rezultat returnat de `search_minima()`.

```
def search_minima(self):
    candidate = self.generate_candidate()
    is_local = False

    while not is_local:
        new_candidate = self.mutation(candidate)

        if new_candidate == candidate:
            return self.evaluate(candidate)

        candidate = new_candidate

def HC(self):
    best = self.search_minima()

    for _ in range(0, self.n):
        minima = self.search_minima()

        if best > minima:
            best = minima

    return best
```

Simulated Annealing

Funcția `SA()`, inițializează temperatura si generează "vecini" ai candidatului cu ajutorul funcției `mutation()`. Candidatul curent va fi schimbat dacă `evaluate()` returnează o valoare mai buna. De asemenea schimbarea se poate realiza și probabilistic. Funcția se oprește când temperatura atinge un anumit minim.

```
def SA(self):
        candidate = self.generate_candidate()
        is_local = False
        T = 1000

        while T > 0.05:
            for i in range(0, self.n):
                new_candidate = self.mutation(candidate)
                score = self.evaluate(new_candidate) - self.evaluate(candidate)

                if score < 0:
                    candidate = new_candidate
                elif random.random() < np.exp(-1 * np.abs(score) / T):
                    candidate = new_candidate

            T *= 0.95

        return self.evaluate(candidate)
```

4 Experiment

Pentru acest experiment vom folosi 4 funcții de test. În matematica aplicată, funcțiile de test sunt folositoare pentru a evalua rata convergenței, precizia si performanța generală a algoritmilor de optimizare.

Funcțiile folosite sunt: *De Jong, Schwefel's, Rastrigin's, Michalewicz's.* Pentru fiecare din aceste funcții vor rula 3 algoritmi: Simulated Annealing, Hill Climbing first si Hill Climbing best.

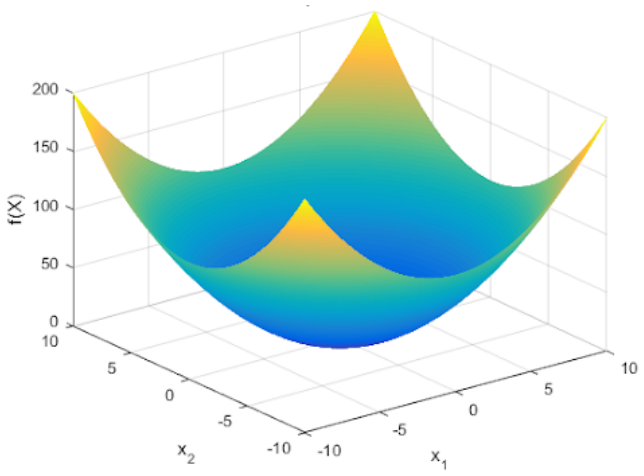
De asemenea, algoritmii vor fi rulați pentru mai multe dimensiuni ale funcțiilor: 5, 10, 30.

(*Algoritmul Hill Climbing a fost rulat pe 200 de iterații iar Simulated Annealing pe 1000 de iteratii cu un T initial de 1000*)

De Jong

Funcția De Jong:

$$f(x) = \sum_{i=1}^{n} x_i^2, x_i \in [-5.12, 5.12]$$



Implementare în Python:

```
def de_jongs(parameters):
    np_parameters = np.array(parameters)
    np_parameters = np_parameters ** 2

    return np_parameters.sum()
```

Rezultate:

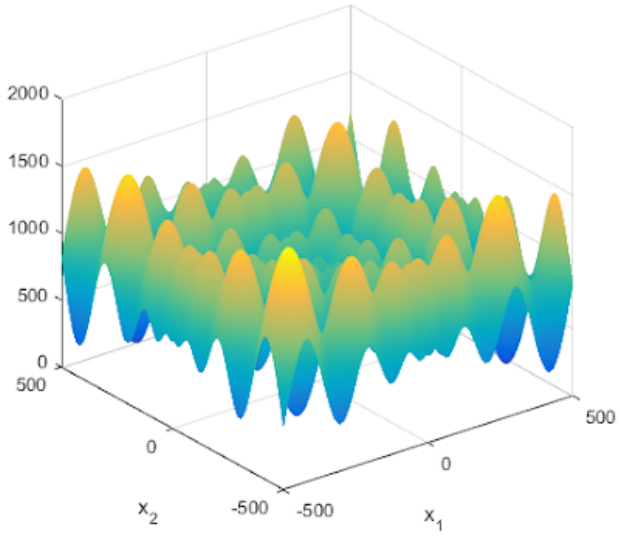
dimensiune	media	σ	timp mediu de execuție(secunde)
5	0, 0, 0.12	0, 0, 0.08	14.65, 10.93, 12.51
10	0, 0, 0.28	0, 0, 0.09	100,88, 63,68, 14.52
30	0, 0, 0.71	0, 0, 0.17	3131, 1772, 39

Fiecare valoare din setul de valori obținute corespunde in această ordine unui algoritm: Hill Climbing best, Hill Climbing first, Simulated Annealing.

Schwefel's

Funcția Schwefel's:

$$f(x) = -\sum_{i=1}^{n} -x_i \cdot sin(\sqrt{|x_i|}), x_i \in [-500, 500]$$



Implementare în Python:

```
def schwefel(parameters):
    np_parameters = np.array(parameters)
    np_parameters = (-1 * np_parameters) * \
        (np.sin(np.sqrt(np.abs(np_parameters))))

    return np_parameters.sum()
```

Rezultate:

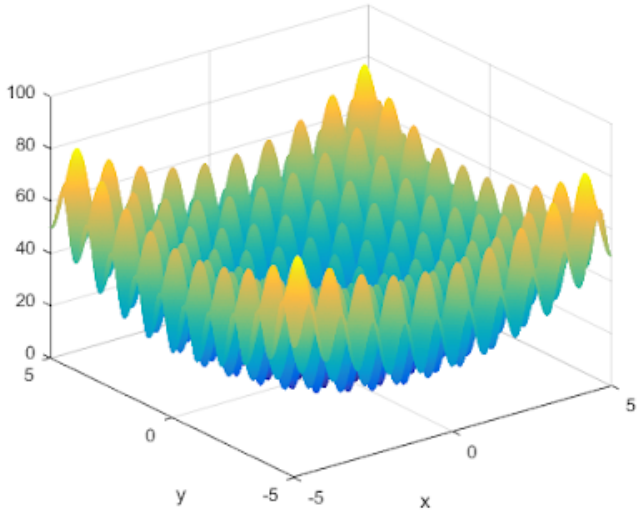
dimensiune	media	σ	timp mediu de execuție(secunde)
5	-2088.10, -2012.42, -2078.70	13.47, 42.67, 35.61	37.05, 23.74, 15.97
10	-3972.47, -3791.62, -4088.96	81.50, 95.21, 99.62	226.10, 149.29, 18.52
30	-11136.87, -10600.64, -11842.15	175.85, 223.98, 275.26	8487.57, 5153.03, 55.05

Fiecare valoare din setul de valori obținute corespunde in această ordine unui algoritm: Hill Climbing best, Hill Climbing first, Simulated Annealing.

Rastrigin's

Funcția Rastrigin's:

$$f(x) = A \cdot n + \sum_{i=1}^{n} \left[x_i^2 - A \cdot cos(2\pi x_i) \right], A = 10, x_i \in [-5.12, 5.15]$$



Implementare în Python:

```
def rastrigins(parameters):
    np_parameters = np.array(parameters)
    np_parameters = (np_parameters ** 2) - \
        (10 * np.cos(2 * np.pi * np_parameters))

    return 10 * len(parameters) + np_parameters.sum()
```

Rezultate:

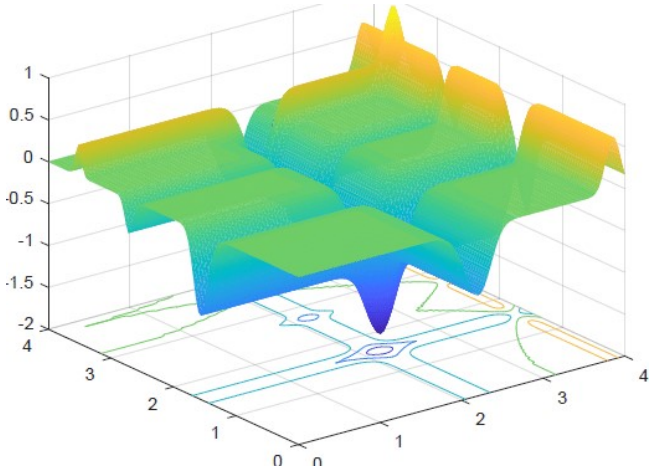
dimensiune	media	σ	timp mediu de execuție(secunde)
5	1.19, 1.66, 3.65	0.60, 0.70, 2.25	15.81, 10.66, 15.60
10	5.80, 7.14, 9.00	1.31, 1.73, 3.52	89.07, 59.86, 17.12
30	32.37, 39.96, 31.89	3.31, 3.11, 5.66	2563.94, 1570.71, 42.26

Fiecare valoare din setul de valori obținute corespunde in această ordine unui algoritm: Hill Climbing best, Hill Climbing first, Simulated Annealing.

Michalewicz's

Funcția Michalewicz's:

$$f(x) = -\sum_{i=1}^{n} sin(x_i) \cdot \left(sin\left(\frac{i \cdot x_i^2}{\pi} \right) \right)^{2 \cdot m}, m = 10, x_i \in [0, \pi]$$



Implementare în Python:

```
def michalewicz(parameters):
    np_parameters = np.array(parameters)

    for i in range(0, len(np_parameters)):
        parameter = np_parameters[i]
        np_parameters[i] = np.sin(parameter) * \
            ((np.sin(i * (parameter ** 2) / np.pi)) ** 20)

    return -1 * np_parameters.sum()
```

Rezultate:

dimensiune	media	σ	timp mediu de execuție(secunde)
5	-3.69, -3.69, -3.52	0, 0, 0.13	18.20, 13.40, 24.13
10	-8.32, -8.24, -8.00	0.08, 0.13, 0.23	120.57, 78.45, 34.61
30	-25.75, -24.93, -25.86	0.30, 0.32, 0.46	3975.52, 1903.29, 90.58

Fiecare valoare din setul de valori obținute corespunde în această ordine unui algoritm: Hill Climbing best, Hill Climbing first, Simulated Annealing.

Optimizări

Pentru a reduce timpul de rulare algoritmii au folosit procese multiple.

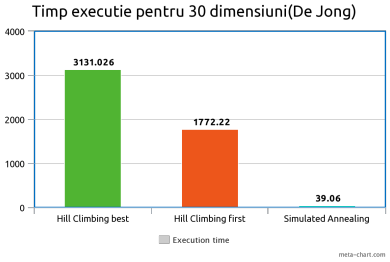
```
aux = 1
    for i in range(0, int(30/processes_number)):
        for j in range(0, int(processes_number)):
            p = multiprocessing.Process(target=execute, args=(aux,
                                                              dimension, interval, function, mode, algorithm))
            aux += 1
            processes.append(p)
            p.start()

        for process in processes:
            process.join()
```

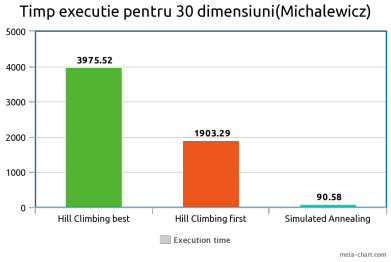
De asemenea pentru a grăbi operațiile ce presupun liste a fost folosită libraria `numpy`(vezi implementarea funcțiilor).

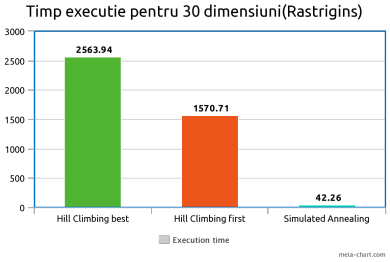
5 Concluzii

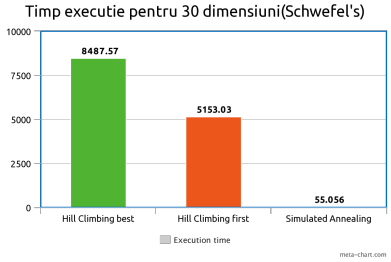
Concluziile rezultate din compararea tabelelor de valori sunt următoarele:

1. Hill Climbing best este cel mai încet algoritm. Pe de altă parte acesta oferă cele mai bune rezultate și cele mai mici deviații standard.
2. Simulated Annealing este cel mai rapid algoritm, dar nu oferă întotdeauna cele mai bune soluții.

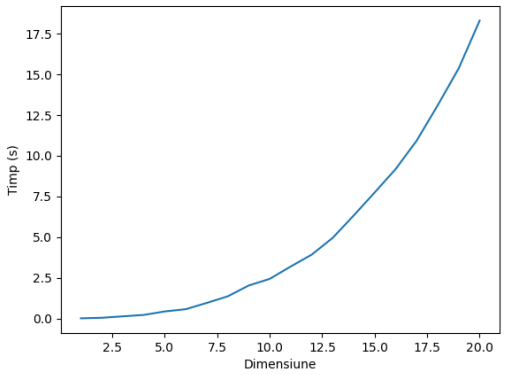








3. Timpul de execuție crește exponențial cu numărul de dimensiuni. Putem vizualiza acest lucru pe datele obținute pe algoritmul Hill Climbing best pentru funcția Schwefel cu câte 3 iterații.



References

[1] Wikipedia
Hill Climbing. `https://en.wikipedia.org/wiki/Hill_climbing`
Simulated Annealing. `https://en.wikipedia.org/wiki/Simulated_annealing`

[2] BenchmarkFcns
Sphere function. `http://benchmarkfcns.xyz/benchmarkfcns/spherefcn.html`
Schwefel function. `http://benchmarkfcns.xyz/benchmarkfcns/schwefelfcn.html`
Rastrigin function. `http://benchmarkfcns.xyz/benchmarkfcns/rastriginfcn.html`

[3] ResearchGate
Michalewicz function. `https://www.researchgate.net/figure/The-landscape-of-the-Michalewicz-Function_fig4_339814374`

[4] GEATbx `http://www.geatbx.com/docu/fcnindex-01.html#P150_6749`

[5] Meta-Chart `https://www.meta-chart.com/bar`

[6] Pandas `https://pandas.pydata.org/`

[7] Jupyter `https://jupyter.org/`