Ricerca di limiti superiori di funzioni tramite ottimizzatori euristici con componenti stocastiche

Marcello Protopapa

Luglio 2019

Sommario

Lo scopo di questa esperienza è cercare una buona approssimazione del massimo di una funzione con un numero arbitrario di parametri in maniera consistente tramite algoritmi euristici con componenti stocastiche, in particolare verranno trattati due algoritmi evolutivi e confrontati i risultati.

1 Introduzione

La ricerca di massimi e minimi è generalmente fatta per problemi di ottimizzazione. Questa esperienza mostra come è possibile effettuare la ricerca di punti di massimo di una funzione applicando algoritmi evolutivi. Mentre normalmente la ricerca di massimi di una funzione sfrutta le caratteristiche del problema per scegliere i punti da cui far partire un'iterazione della ricerca (es. gradiente, linearità e continuità), la tecnica stocastica utilizza soltanto il valore della funzione in un punto. Gli algoritmi evolutivi sono utilizzati per risolvere problemi che hanno funzioni obbiettivo "brutte" in termini di continuità, derivabilità, etc.
In questa esperienza sono confrontati due particolari algoritmi evolutivi, l' algoritmo genetico e l'evoluzione differenziale, insieme all'accoppiamento di evoluzione differenziale con un classico algoritmo di ottimizzazione euristica con componenti stocastiche, l'Hill Climbing.
Nella sezione 2 verranno discussi gli algoritmi evolutivi in generale, mentre

nella sezione 3 viene trattata l'implementazione specifica, includendo un'analisi dei risultati trovati. Il codice dell'esperienza è disponibile in appendice. Ispirazione per l'esperienza è stata presa da un articolo di Dezdemona Gjylapi e Vladimir Kasëmi [2] e da un libro di Vasan Arunachalam [1].

2 Cenni Teorici

2.1 Algoritmi Evolutivi

Gli algoritmi evolutivi sono un modello di intelligenza artificiale che esegue una ricerca nello spazio di soluzioni combinando ad ogni iterazione i risultati più promettenti ed applicando mutazioni casuali alle ipotesi generali. Mitchell definisce gli algoritmi genetici, un particolare tipo di algoritmi evolutivi, nel seguente modo [4, p. 249–250]:

> Gli algoritmi genetici forniscono un metodo di apprendimento basato su un'analogia all'evoluzione biologica. Anziché cercare ipotesi dalla più generale alla più specifica, o dalla più semplice alla più complessa, gli algoritmi genetici generano l'ipotesi successiva mutando e combinando ripetutamente parti delle migliori ipotesi conosciute al momento. Ad ogni passo, un insieme di ipotesi, detta popolazione, è aggiornata scambiando una parte della popolazione con i "figli" delle migliori ipotesi correnti.

Gli algoritmi evolutivi sono dunque composti da più fasi per ogni iterazione:

- Selezione: Si valutano le ipotesi con qualche funzione di valutazione. Migliore è un'ipotesi, maggiore sarà il suo ranking. Vengono scelte le ipotesi che verranno utilizzate nella prossima fase, solitamente in base al ranking.

- Crossover: Si selezionano un certo numero di ipotesi e si mischiano per produrre una nuova ipotesi. La selezione è casuale, ma ipotesi con un ranking più alto hanno di solito probabilità maggiore di essere selezionate.

- Mutazione: Tutte le ipotesi ottenute variano con una certa probabilità.

Sono di seguito riportate alcune tecniche usate dagli algoritmi evolutivi per la rappresentazione dei dati e le varie fasi dell'esecuzione.

2.2 Rappresentazione dei dati

La rappresentazione delle ipotesi di un algoritmo evolutivo è detta cromosoma, composto da geni. Il cromosoma è solitamente rappresentato da una stringa o un vettore di bit (in cui ogni carattere o ogni bit è un gene), o da un vettore di numeri, una matrice, etc. Dipende comunque dal problema che si sta cercando di risolvere.

2.3 Crossover

Il crossover consiste nel prendere due cromosomi e scambiare parte dei loro geni. Si può fare con diverse tecniche:

- Singolo punto (Single point crossover): Si tagliano i due cromosomi in un punto specifico (predeterminato o casuale) e si scambiano i due sottovettori tagliati.
- Due punti: Come il single point crossover, ma il vettore che si taglia ha una testa e una coda scelte casualmente o predeterminate.
- Uniforme: Si scambiano un certo numero di punti del vettore, decisi secondo una distribuzione di probabilità o predeterminati.
- Analitico: Si utilizza una funzione per combinare i due geni (ad esempio and, or, punto medio di un segmento)

2.4 Mutazione

La mutazione è fatta cambiando il cromosoma in maniera casuale e viene fatta per assicurarsi che la soluzione non converga a minimi o massimi locali della funzione di ranking. Spesso la mutazione viene fatta dopo il crossover, specie quando la selezione è fatta in base al fitness. La mutazione può avere una probabilità assegnata per gene, o per tutto il cromosoma. È importante inoltre che la probabilità non sia né troppo bassa, perchè si corre il rischio di fermarsi su minimi o massimi locali, né troppo alta, altrimenti l'algoritmo si trasforma in una ricerca casuale.

2.5 Selezione

La selezione è il momento in cui vengono scelti i candidati per la generazione successiva, sia direttamente (elitismo) che indirettamente (elementi genitori per il crossover). La selezione può essere:

- A Roulette (o Ranking Lineare): Si assegna una probabilità direttamente proporzionale ai valori di fitness.
- A Torneo: Si sceglie un sottoinsieme dei cromosomi e si fa un torneo. Il vincitore si decide in base alla funzione di fitness. Si assegnano le probabilità di selezione in base alla posizione nel torneo. Si ripete fino ad aver generato tutta la nuova popolazione.
- Selezione di Boltzmann, che bilancia esplorazione e qualità dei risultati in base al numero di iterazioni calcolando entropia ed energia in modo che le selezione non dipenda solo dalla funzione di fitness [3].

Vi è inoltre il concetto di elitismo: Se un valore della generazione precedente ha fitness maggiore di tutti i valori della generazione successiva, allora il valore viene mantenuto nella generazione successiva.

2.6 Evoluzione Differenziale

L'evoluzione differenziale [5] è un particolare algoritmo evolutivo spesso usato per la ricerca in spazi di ipotesi continui, in particolare funzioni con un numero elevato di parametri. Le iterazioni nell'algoritmo sono fatte nel seguente modo:
Si inizializza il programma con valori casuali. Poi, per ogni elemento della popolazione x_i

- Mutazione: Si scelgono 3 vettori x_a, x_b, x_c dalla popolazione e si combinano secondo l'equazione

  $$x'_c = x_c + F(x_a - x_b)$$

  F è una costante detta peso o scala, solitamente in valori tra 0.5 e 1.
- Crossover: Si combinano i vettori x'_c e x_i con uno dei tipi di crossover visti in precedenza. Il vettore risultato x'_i è il vettore candidato.

- Selezione: Se $f(x_i') > f(x_i)$ (per problemi di massimo, $<$ per problemi di minimo) allora x_i' verrà aggiunto alla prossima generazione al posto di x_i.

Una caratteristica interessante dell'algoritmo è che la selezione è naturalmente bilanciata in termini della cosiddetta temperatura dei dati: Nella fase iniziale, è desiderabile che la ricerca esplori nuove soluzioni e visto che i vettori della popolazione sono molto diversi tra di loro, il vettore x_c sarà molto casuale e rumoroso. Con l'aumento delle iterazioni, la popolazione tenderà a convergere verso un massimo locale. A questo punto x_c sarà quindi molto più vicino al resto della popolazione, consentendo variazioni più piccole una volta che l'algoritmo si è avvicinato all'estremo.

2.7 Differenze tra Algoritmi Genetici ed Evoluzione Differenziale

Gli algoritmi genetici e l'evoluzione differenziale sono due esempi di algoritmi evolutivi, ed in quanto tali entrambi consistono nelle fasi di Selezione, Crossover e Mutazione. Gli algoritmi genetici si prestano meglio per ricerche in spazi di ipotesi discreti e facilmente esprimibili come stringhe o insiemi di valori binari. In contrasto, l'evoluzione differenziale è nata per la ricerca in spazi di ipotesi continui, spesso funzioni con molti parametri e in cui i cromosomi sono vettori di numeri. La differenza principale è che mentre gli algoritmi genetici generano la popolazione principalmente tramite il crossover, nell'evoluzione differenziale il ruolo della mutazione è molto più importante.

2.8 Hill Climbing

L'Hill Climbing è un algoritmo ricerca euristica con elementi stocastici molto semplice: Si genera una soluzione di partenza x, poi si valuta il valore della funzione obbiettivo in un punto $x + \delta$. Se è migliore $x = x + \delta$, altrimenti si cambia δ e si ripete lo step in x.
L'algoritmo tende a bloccarsi in estremi locali. Per questo di solito si applica la tempra simulata, per cui si fa variare la "temperatura" in base al numero di iterazioni. In pratica ciò vuol dire che durante le prime iterazioni vengono considerati anche valori peggiori del corrente, ma più si va avanti, più si considerano solo i valori migliori.

In generale l'algoritmo è utilizzato perché molto rapido e semplice da implementare, quindi è un buon punto di partenza quando si sta cercando la soluzione ad un problema.

3 Implementazione

L'implementazione è stata fatta in Matlab. Sono stati implementati un algoritmo genetico e due versioni di evoluzione differenziale, una normale e una che esegue qualche passo di Hill Climbing alla fine dell'esecuzione.

3.1 Rappresentazione dei dati

Algoritmo Genetico: Per la rappresentazione dei dati dell'algoritmo genetico è stata usata una matrice $DIM \times NDIGITS \times NP$, dove DIM è la dimensione del problema, NDIGITS la dimensione della rappresentazione binaria e NP la dimensione della popolazione. Sono state implementate le funzioni per convertire un numero da float a binario e viceversa, nonché tutte le funzioni che si occupano di manipolare i dati (metodi Mutate, Crossover). La rappresentazione binaria di un numero è stata fatta con 3 cifre prima della virgola e 10 dopo la virgola, più una per il segno. Le cifre prima della virgola dipendono dai vincoli del problema, mentre le cifre dopo la virgola sono variabili a seconda della precisione desiderata.

Evoluzione Differenziale: La rappresentazione dei dati dell'evoluzione differenziale è stata fatta con una matrice $N \times M$ di floating point in cui ogni colonna è un cromosoma, gli elementi delle righe sono i geni.

3.2 Tecniche utilizzate

3.2.1 Algoritmo Genetico

La funzione GeneticAlgorithm (Appendice A.1) è l'implementazione dell'algoritmo genetico. La funzione prende in ingresso una funzione di valutazione (di cui si vuole sapere il massimo), il numero di dimensioni dell'input, il numero di cromosomi in ogni generazione, il numero di iterazioni (generazioni) da eseguire e l'intervallo in cui cercare il massimo.
I dati di partenza sono generati casualmente e poi ordinati in base alla funzione di fitness (che è la funzione di cui si vuole trovare il massimo). Viene

poi iniziato il loop principale del programma, che esegue la selezione e il crossover fino a generare tutta la nuova popolazione, poi effettua la mutazione e il calcolo del fitness.
Il crossover è stato fatto in un singolo punto, scegliendo casualmente un vettore per ogni dimensione in base alla probabilità. Per la selezione è stato usato il Ranking Lineare con probabilità

$$p(i) = \frac{1}{n}[\beta - 2(\beta - 1)\frac{i-1}{n-1}]$$

con $\beta = 2$. La mutazione è stata fatta su ogni bit con probabilità 1/10.

3.2.2 Evoluzione Differenziale

L'evoluzione differenziale è stata implementata come descritto in 2.6, con F che dipende inversamente dal numero di iterazioni, partendo da 2 e scendendo fino a 1, crossover uniforme e probabilità di crossover 1/2. L'implementazione è nell'Appendice A.2

3.2.3 Hill Climbing

L'implementazione di Hill Climbing è riportata in Appendice A.3. Lo spostamento è in una direzione casuale per ogni dimensione, con step che diminuisce con il numero di iterazioni e condizione di arresto che dipende dal numero di iterazioni e la differenza tra due iterazioni successive in cui è variato il valore finale. Lo step è molto piccolo già in partenza e viene eseguito un numero molto limitato di iterazioni per dare i ritocchi finali alla soluzione.

3.3 Testing

Il testing è stato fatto sulla funzione di Rastrigin [6] (Figura 1) in 10 dimensioni. La funzione è definita come

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

La funzione ha un minimo in $f(0) = 0$ (è stata valutata $-f(x)$ perché l'algoritmo implementato cerca un massimo). Sono state confrontate le esecuzioni dell'Algoritmo Genetico implementato, della funzione ga di Matlab (Algoritmo Genetico), dell'Evoluzione Differenziale e dell'Evoluzione Differenziale

con Hill Climbing alla fine dell'esecuzione. Sono stati considerati come parametri separatamente la dimensione della popolazione e il numero di iterazioni dell'algoritmo. Sono state eseguite 50 iterazioni per ogni valore testato e sono stati valutati i risultati, di cui si è salvata la media e la deviazione standard.

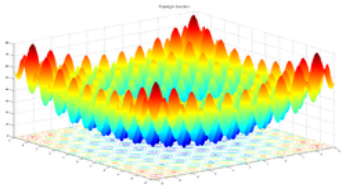

Figura 1: Plot 3D della funzione di Rastrigin.

4 Risultati

I risultati sono mostrati nei due grafici a fine sezione. L'asse delle ordinate è una media dei valori restituiti dagli algoritmi al variare del parametro testato, l'errore è la deviazione standard dei risultati dopo 50 iterazioni.

4.1 Numero di iterazioni

Per il numero di iterazioni è stata fissata la popolazione a 5 volte il numero di dimensioni. I risultati sono mostrati in Figura 2. Come si vede dal grafico, gli algoritmi genetici trovano un'approssimazione migliore con un numero di iterazioni molto basso, ma dato un numero di iterazioni sufficientemente elevato, in questo caso circa 1000, l'evoluzione differenziale dà un'approssimazione migliore e più consistente.

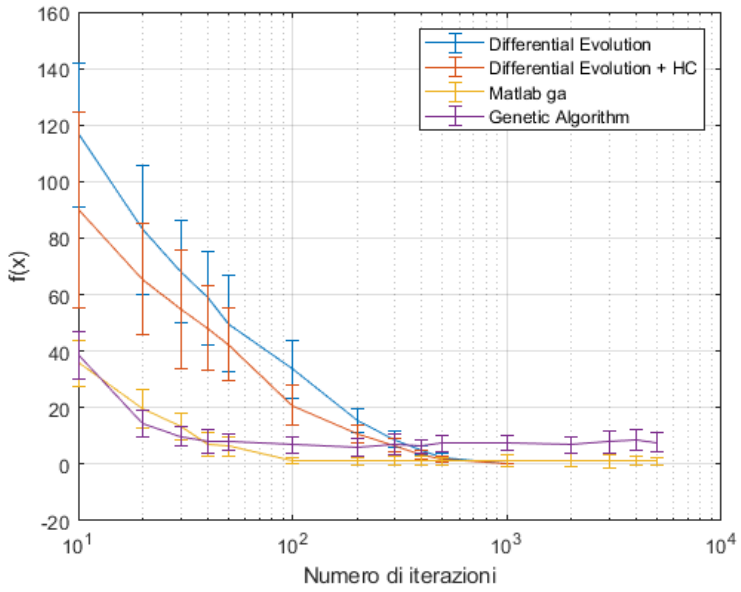

Figura 2: Media dei valori al variare del numero delle iterazioni

4.2 Popolazione

Per quanto riguarda la popolazione (Figura 3), con numero di iterazioni fisse a 2000, l'evoluzione differenziale è risultato l'algoritmo migliore, producendo risultati in un intorno piccolo di 0 (rispetto agli altri algoritmi) già con una popolazione pari a 3 volte il numero di dimensioni. Il risultato è migliore di quello che ci si aspetterebbe secondo la bibliografia, suggerisce 10 volte la dimensionalità del problema [5] contro le 3 riscontrate.

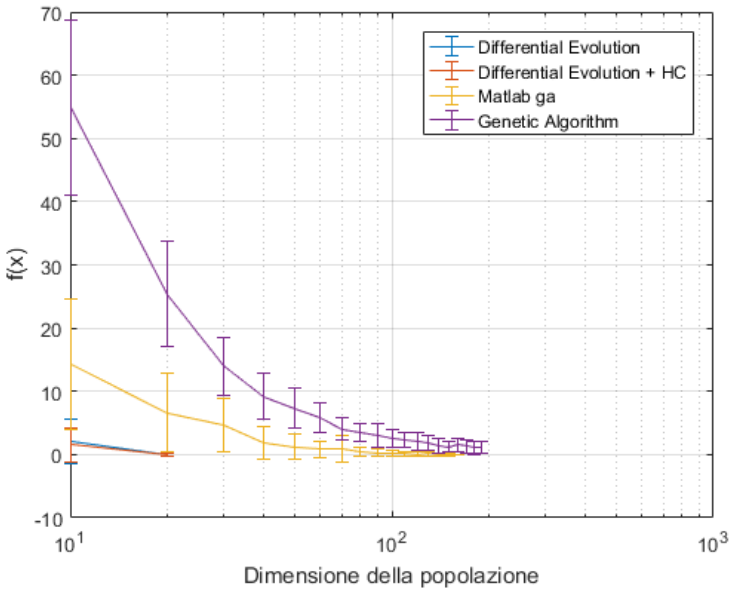

Figura 3: Media dei valori al variare della popolazione

5 Conclusioni

I risultati mostrano che, nonostante l'algoritmo non fosse particolarmente ottimizzato, l'evoluzione differenziale ha bisogno di un numero di generazioni minore dell'algoritmo genetico per trovare una buona approssimazione del massimo in maniera consistente entro il limite fissato e ha inoltre bisogno di una popolazione molto più ristretta. Ciò non toglie che con poche iterazioni Matlab dia risultati migliori, probabilmente dovuto ad un'ottimizzazione migliore dell'algoritmo e/o a una parte iniziale di preprocessing non implementata negli algoritmi presentati, ma comunque lontani dall'ottimo globale del problema. L'algoritmo genetico implementato non ha dato ottimi risultati, durante i test per il numero di iterazioni, ma il secondo grafico mostra che è molto dipendente dalla dimensione della popolazione ed è possibile che cambiando i parametri e il metodo di selezione con qualcosa di più ottimizzato per questo tipo di problemi, i risultati possano migliorare.

A Sorgente

A.1 GeneticAlgorithm.m

```
function [y] = GeneticAlgorithm(f, ndim, populationSize,
    niter, a, b)

%Initialization
digitsBefComma = 3;
digitsAftComma = 10;
nbits = 1 + digitsBefComma + digitsAftComma; %sign +
    integer digits + decimal digits

%generate the population matrix with linspace
x = zeros(ndim, nbits, populationSize);
for k = 1:populationSize
    v = rand(1, ndim).*(b-a)+a;
    x(:, :, k) = float2bin(v(:));
end

% %initialize the results
y = zeros(1, populationSize);
for i = 1:populationSize
    y(i) = f(bin2float(x(:, :, i)));
end

%sort according to ranking
[~, order] = sort(y, 'descend');
x = x(:, :, order);
y = y(order);

%calculate probabilities
beta = 2;
probabilities = (1/(populationSize))*(beta-2*(beta-1)*(
    linspace(1,populationSize, populationSize)-1)/(
    populationSize-1));
%calculate PMF(Probability Mass Function)
probabilities = cumsum(probabilities);

for i = 1:niter
```

```
33      %Initialize the new generation array and result
34      nextGenX = zeros(ndim, nbits, populationSize);
35      nextGenY = zeros(1, populationSize);
36
37      for j = 1:2:populationSize
38          for h = 1:ndim
39              %choose randomly 2 vectors according to the
                    ranking
40              v = [find(probabilities>rand(1),1) find(
                    probabilities>rand(1),1)];
41              %Crossover for every dimention
42              [nextGenX(h,:,j),nextGenX(h,:,j+1)] = crossover
                    (x(h, :, v(1)), x(h, :, v(2)));
43          end
44
45          %Mutate both vectors
46          nextGenX(:,:,j) = mutate(nextGenX(:,:,j));
47          %Ensure the value is within the problem bounds
48          val = bin2float(nextGenX(:,:,j));
49          if(val < a)
50              nextGenX(:,:,j) = float2bin(a);
51          elseif(val > b)
52              nextGenX(:,:,j) = float2bin(b);
53          end
54
55          nextGenX(:,:,j+1) = mutate(nextGenX(:,:,j+1));
56          val = bin2float(nextGenX(:,:,j+1));
57          if(val < a)
58              nextGenX(:,:,j+1) = float2bin(a);
59          elseif(val > b)
60              nextGenX(:,:,j+1) = float2bin(b);
61          end
62
63          nextGenY(j) = f(bin2float(nextGenX(:,:,j)));
64          nextGenY(j+1) = f(bin2float(nextGenX(:,:,j+1)));
65      end
66
67      maxnew = max(nextGenY);
68      %initialize an empty temporary matrix
69      newy = zeros(1, populationSize);
```

```
70      newx = zeros(ndim, nbits, populationSize);
71
72      [~, order] = sort(nextGenY, 'descend');
73      nextGenX = nextGenX(:, :, order);
74      nextGenY = nextGenY(order);
75
76      %Elitism
77      val = 0;
78      for k = 1:populationSize
79          if(y(k) > maxnew)
80              newy(k) = y(k);
81              newx(:, :, k) = x(:, :, k);
82              val = val + 1;
83          else
84              newy(k) = nextGenY(k - val);
85              newx(:, :, k) = nextGenX(:, :, k - val);
86          end
87      end
88
89      x = newx;
90      y = newy;
91  end
92
93  y = f(bin2float(x(:, :, 1)));
94  end
95
96  function res = float2bin(val)
97      %Float2bin returns the binary representation of a float
            number in base 10
98      n = 3; %digits before comma
99      m = 10; %digits after comma
100
101     res = [val<0 fix(rem(abs(val).*pow2(-(n-1):m),2))];
102 end
103
104 function res = bin2float(val)
105     %Bin2float returns the base 10 float representation of
            a binary number.
106     %n = 3;
107     %m = 10;
```

```
108     %pows = pow2(n-1:-1:-m)
109     pows = [4.0000    2.0000    1.0000    0.5000    0.2500
                0.1250    0.0625    0.0313    0.0156    0.0078
                0.0039    0.0020    0.0010];
110     s = size(val);
111     pows = repmat(pows,s(1),1);
112
113     mult = (val(:, 2:s(2)).*pows);
114     res = ((-1).^val(:,1)).*sum(mult, 2);
115 end
116
117 function [num1, num2] = crossover(val1, val2)
118     %Crossover crosses 2 binary numbers at a random point
            and returns the result (without changing the inputs)
119     digitBefComma = 3;
120     digitAftComma = 10;
121
122     %Generate random number between 2 and  1 + BinaryNumber
            .digitBefComma + BinaryNumber.digitAftComma
123     uplim = 1 + digitBefComma + digitAftComma;
124     digit = ceil(rand(1)*(uplim-2))+1;
125
126     %save the 4 subvectors
127     tmp11 = val1(1:digit-1);
128     tmp12 = val1(digit:uplim);
129     tmp21 = val2(1:digit-1);
130     tmp22 = val2(digit:uplim);
131
132     %swap the subvectors
133     num1 = [tmp11 tmp22];
134     num2 = [tmp21 tmp12];
135 end
136
137 function obj = mutate(obj)
138     %GetDigit returns the digit at position n. Sign is
            mutable too
139
140     mutationProb = 0.1;
141     digitBefComma = 3;
142     digitAftComma = 10;
```

```
143
144         %Generate a random number for each digit
145         rands = rand(digitBefComma + digitAftComma + 1, 1);
146
147         %mutate every digit where the random number generated
                is
148         %smaller then BinaryNumber.mutationProb
149         idxs = find(rands <= mutationProb);
150         obj(idxs) = mod(obj(idxs)+1,2);
151     end
```

A.2 DifferentialEvolutionHC.Cm

```
function [y] = DifferentialEvolutionHC(f, ndim,
    populationSize, niter, a, b)
%Initialization
x=linspace(a,b,populationSize);
x=repmat(x,ndim,1);

for i = 1:niter
    %calculate F
    F = 2-(i/niter)*2;

    %initialize the next generation vector
    nextGen = zeros(ndim, populationSize);

    for j = 1:populationSize
        n = populationSize;
        l = 3;
        %Choose 3 random members of the population
        Xs = randperm(n,l);

        %calculate Xc
        Xc = transpose(x(:, Xs(1)) + F*(x(:, Xs(2))-x(:, Xs
            (3))));

        %calculate X'i
        nextGen(:, j) = crossover(x(:, j), Xc, ndim);

        %Make sure it's within the problem bounds
        for k = 1:ndim
            if(nextGen(k, j) < a)
                nextGen(k, j) = a;
            elseif(nextGen(k, j) > b)
                nextGen(k, j) = b;
            end
        end
    end

    %Only keep chromosomes bigger then their old generation
        counterpart
```

```
36      for j = 1:populationSize
37          y1 = f(x(:, j));
38          y2 = f(nextGen(:, j));
39          if(y1 > y2)
40              x(:, j) = x(:, j);
41          else
42              x(:, j) = nextGen(:, j);
43          end
44      end
45  end
46
47  %find the best value
48  [~, i] = max(f(x));
49  %apply HillClimb
50  x = HillClimb(f, x(:, i), ndim);
51  %Check for problem bounds
52  for i = 1:ndim
53      if(x(i) < a)
54          x(i) = a;
55      elseif(x(i) > b)
56          x(i) = b;
57      end
58  end
59  %Return the best value
60  y = f(x(:));
61  end
62
63  function x = crossover(x1, x2, ndim)
64      %Crossover probability
65      CR = 0.5;
66
67      x = x1;
68      %Find all indexes for which a random number is smaller
            then the
69      %crossover probability
70      idxs = find(rand(1, ndim) < CR);
71      %swap values at said indexes
72      if(~isempty(idxs))
73          x(idxs) = x2(idxs);
74      end
```

```
75  end
```

A.3 HillClimb.m

```
function x = HillClimb(f, x0, ndim)
   %Initialization
    stepSize = 0.1;
    acceleration = 1.2;
    niters = 10*ndim; %10 iteration x number of dimentions
    xn = x0;
    stopcond = 0.0001; %stop if diff between 2 iteration is
       less then this

    diff = 1;
    i=1;

    while (i <= niters && diff > stopcond)
        %Move in a random direction for each dimension
        xn1 = xn + stepSize*(-1).^round(rand(ndim, 1));

        %if the new position is better then the old then
           keep it
        if(f(xn1) > f(xn))
            diff = mean(abs(xn1-xn));
            xn = xn1;
        end

        %decrease step size
        stepSize = stepSize/acceleration;
        i = i + 1;
    end

    x = xn;
end
```

Riferimenti bibliografici

[1] Vasan Arunachalam. *Machine Learning*. Department of Civil and Environmental Engineering, The University of Western Ontario, 2008.

[2] Vladimir Kasëmi Dezdemona Gjylapi. The genetic algorithm for finding the maxima of single-variable functions. *Research Inventy: International Journal Of Engineering And Science*, 4(3):46–54, Marzo 2014.

[3] Chang-Yong Lee. Entropy-boltzmann selection in the genetic algorithms. *IEEE transactions on cybernetics*, 33(1):138–142, Febbraio 2003.

[4] Tom M Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[5] Kenneth Price Rainer Storn. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, Dicembre 1997.

[6] Wikipedia. Rastrigin function, 2019.