Implementación de Random Mutation Hill Climber

Karen L. Poblete 116452
Maestría en Ciencias en Computación
Instituto Tecnológico Autónomo de México
Complejidad y Computabilidad
22 de noviembre de 2014

Un Random Mutation Hill Climber (RMHC) o Escalador de Mutación Aleatoria, es un algoritmo que intenta, de manera iterativa, alcanzar un valor óptimo para una función dada mutando la pasada mejor solución encontrada. Esté algoritmo sólo cuenta con un individuo, sobre el cual se realizan iterativamente las mutaciones. Si el individuo mejora su valor de función *fitness*, éste se vuelve el mejor hasta ese punto y es mutado uno de sus bits aleatoriamente. Éste algoritmo ha mostrado tener mejores resultados que los demás escaladores que existen.

En éste reporte se muestran los resultados de aplicar el algoritmo RMHC a 4 diferentes funciones para encontrar el valor mínimo que las satisface. Se crearon dos clases en Java: RMHC.java y RandomMHC.java. Ambas clases implementan funciones que ayudan a solucionar el problema en partes.

Funciones minimizadas

Las funciones a minimizar se presentan a continuación. Dichas funciones fueron tomadas del documento de descripción de funciones referencia para analizar diferentes algoritmos optimizadores.

1. **Función de De Jong:** Es una función continua, convexa y unimodal. A continuación se presenta su definición general:

$$f(x) = \sum_{i=1}^{n} x_i^{\,2} \tag{F.1}$$

El dominio está restringido a $-5.12 \le x_i \le 5.12, \quad i = 1,\ldots,n$; el mínimo conocido es 0. En la Fig. F.1 se muestra la gráfica para $n=2$. Se programó para n=4, con 3 bits de números enteros y 7 decimales.

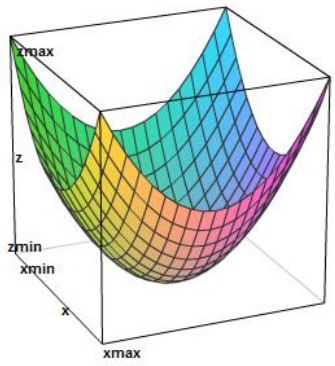

Fig. F.1 Función de De Jong

2. **Función del valle de Rosenbrock:** El óptimo global se encuentra dentro de un largo, estrecho y parabólico plano valle. Es difícil llegar al óptimo global y se la función se define así:

$$f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^{\,2})^2 + (1 - x_i)^2 \qquad \text{(F.2)}$$

El dominio se encuentra entre $-2.048 \le x_i \le 2.048, \quad i = 1,\ldots,n$; el mínimo conocido es 0. En Fig. F.2 se muestra la función para $n=2$. Se programó para n=2, con 2 bits enteros y 7 decimales.

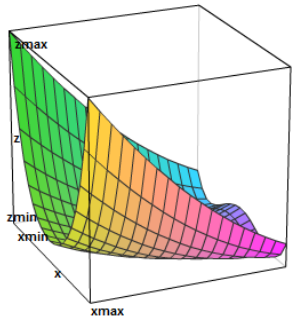

Fig. F.2. Función del Valle de Rosenbrock

3. **Función de Schwefel:** Es una función engañosa, es muy común convergir en direcciones equivocadas. La función se define como a continuación se muestra.

$$f(x) = \sum_{i=1}^{n} [-x_i \sin(\sqrt{|x_i|}] \qquad \text{(F.3)}$$

El dominio de la función se define entre $-500 \le x_i \le 500, \quad i = 1,\ldots,n$; el mínimo conocido es -418.9828n. En la Fig. F.3 se muestra la gráfica de la función para $n=2$. Se programó para n=3, con 9 bits para enteros y 7 para decimales.

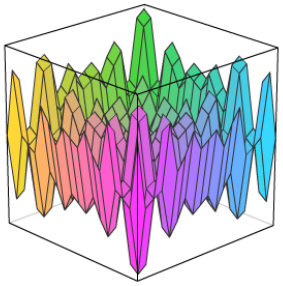

Fig. F.3. Función de Schwefel

4. **Función de Ackley:** Es una función multimodal. Se define como:

$$f(x) = a \cdot \exp(-b \cdot \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^{\,2}}) - \exp(\frac{1}{n}\sum_{i=1}^{n} \cos(cx_i)) + a + \exp(1) \qquad \text{(F.4)}$$

Utiliza los valores contantes $a = 20, \quad b = 0.2 \quad c = 2\pi$. El dominio de la función está restringido a $-32.768 \le x_i \le 32.768, \quad i = 1,\ldots,n$; El mínimo conocido es 0. En la figura Fig. F.4 se muestra una gráfica de la función para *n=2*. Aquí se programó para n=3, con 6 bits enteros y 7 decimales.

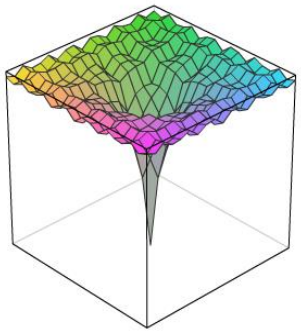

Fig. F.4. Función de Ackley

Pseudocódigo del Escalador de Mutación Aleatoria

El escalador de mutación aleatoria programado se basa en el siguiente pseudocódigo. El primer paso es generar a un individuo aleatoriamente. La longitud de dicho individuo depende de cuantos bits se utilizan para enteros E y decimales D más el bit de signo S; se hace la suma de dichos valores y se multiplica por el número de variables n:

L = (E+D+S)*n

Ese individuo se llena con 0 y 1 aleatoriamente. Después el mejor se define como ese individuo creado y el mejor *fitness* como un valor infinito. A continuación se empiezan las iteraciones hasta G, donde por cada iteración se evalúa la función *fitness* con el individuo, si éste tiene un valor menor entonces el mejor será sustituido por ese individuo que fue mejor y se sustituirá también el valor del mejor *fitness*. Por último se realiza una mutación

que consiste en tomar un número aleatorio entre 0 y L, sí era 0 se convierte en 1 y viceversa.

```
1. [Generate the individual]
for i=1 to L
      Generate a uniform random number 0 ≤ ρ < 1.
      If ρ < 0.5 make bit_i ← 0; else make bit_i ← 1.
endfor
Make best ← I(0)
          BestFit ← ∞
2. [Iterate]
for i = 1 to G
      [Evaluate the individual]
      f(i) ← fitness (x_i)
      if fitness(i)<BestFit
            best ← I(x_i)
            BestFit ← fitness(x_i)
      endif
      [Mutate]
      Generate a uniform random number 1 ≤ k ≤ L .
      Make bit_k ← bit_k (complemento)
endfor
```

La función de fitness

La función de *fitness* es aquella función que nos indica qué tan bien encaja el individuo en la función que se intenta optimizar. Para fines de este reporte, es la función que minimiza el valor de la función que se esté procesando en ese momento, es decir, puede ser una de las cuatro funciones descritas con anterioridad.

Las funciones que se utilizan aquí tienen restricciones de dominio, los cuales marcan los valores mínimos y máximos que pueden tener las n variables involucradas. Para que las soluciones cumplan con dicha restricción es necesario utilizar una función de *fitness* que castigue a los individuos que no cumplan.

Se utiliza un valor máximo de fitness que se puede asignar a una variable y se divide entre el número de variables que intervienen en la función, dicho valor es r_max. Cuando una variable esta fuera del dominio un contador r que aumenta su valor en 1. Al final dicho contador se multiplica por r_max y se suma a la función de *fitness*. Esto garantiza que los valores cumplan con la restricción. A continuación se presenta una parte del código que muestra cómo se programó esto.

```
case 1: //De Jong s Function
                max_r = max/4;
                for(int i=0;i <4;i++){
```

```
            fit = fit + Math.pow(valores[i], 2);
            if(valores[i]<-5.12 || valores[i]>5.12){
                r++;
            }
        }
        fit = fit +(r*max_r);
        break;
```

Descripción del programa

Como se mencionó con anterioridad se implementaron dos clases en java: RMHC.java y RandomMHC.java, las cuales contienen las funciones necesarias para que el RMHC funcione. La clase RMHC.java contiene las configuraciones de las 4 funciones de las cuales se requiere encontrar el valor mínimo global. Por cada una de las funciones hace un número indicado de iteraciones del algoritmo y escribe los resultados en *Resultados.txt*. En dicho archivo sólo se encuentra el valor final después de ejecutado el algoritmo.

La clase RandomMHC.java contiene todas las funciones del RMHC y también realiza una escritura en un archivo llamado *ValoresCorrida.txt*, que muestra información complementaria, es decir, escribe los valores finales de las n variables. En ambas escrituras de archivo, si el archivo no existe se crea y si existe se agregan los resultados al documento sin borrar los anteriores.

El programa se corre desde consola por medio del comando: *`java RMHC`*, en la carpeta donde se encuentran los .class, es por eso necesario la previa compilación de las clases, por medio del comando *`javac -g RMHC.java RandomMHC.java`*. Se adjunta el código comentado.

En consola sólo se puede observar en que vuelta del algoritmo y de que función se encuentra. Todos los resultados se ven reflejado en los txt antes descritos. A continuación se muestra un ejemplo de la salida en la consola.

```
Ronda 26 Funcion 2
Ronda 27 Funcion 2
Ronda 28 Funcion 2
Ronda 29 Funcion 2
Ronda 30 Funcion 2
```

Resultados

Se corrió el programa para cada una de las funciones. A continuación se muestran los resultados respectivamente.

Función 1. Ésta función se corrió para G*4 = 4,000,000, 30 veces, donde G es el número de iteraciones máximas del algoritmo.

```
Funcion: 1  var[0]: 0.0614013671875
Funcion: 1  var[1]: 0.11846923828125
Funcion: 1  var[2]: 0.11895751953125
Funcion: 1  var[3]: 0.06353759765625
Funcion: 1  var[4]: 0.05499267578125
Funcion: 1  var[5]: 0.04168701171875
Funcion: 1  var[6]: 0.02703857421875
Funcion: 1  var[7]: 0.07733154296875
Funcion: 1  var[8]: 0.049072265625
Funcion: 1  var[9]: 0.11749267578125
Funcion: 1  var[10]: 0.21490478515625
Funcion: 1  var[11]: 0.04180908203125
Funcion: 1  var[12]: 0.14251708984375
Funcion: 1  var[13]: 0.03045654296875
Funcion: 1  var[14]: 0.05120849609375
Funcion: 1  var[15]: 0.0572509765625
Funcion: 1  var[16]: 0.19598388671875
Funcion: 1  var[17]: 0.02813720703125
Funcion: 1  var[18]: 0.0145263671875
Funcion: 1  var[19]: 0.0987548828125
Funcion: 1  var[20]: 0.03594970703125
Funcion: 1  var[21]: 0.108642578125
Funcion: 1  var[22]: 0.01348876953125
Funcion: 1  var[23]: 0.09246826171875
Funcion: 1  var[24]: 0.0596923828125
Funcion: 1  var[25]: 0.0198974609375
Funcion: 1  var[26]: 0.1624755859375
Funcion: 1  var[27]: 0.04925537109375
Funcion: 1  var[28]: 0.04913330078125
Funcion: 1  var[29]: 0.0718994140625
```

Los valores se aproximan a 0, que es el mínimo conocido. En promedio **f(x) = 0.075614421**. Y la desviación estándar **ds(f(x))= 0.28032727**.

Función 2. Ésta función se corrió para G = 1,000,000, 30 veces, donde G es el número de iteraciones máximas del algoritmo.

```
Funcion: 2  var[0]: 0.0
Funcion: 2  var[1]: 0.0
Funcion: 2  var[2]: 6.140768527984619E-5
Funcion: 2  var[3]: 0.0
Funcion: 2  var[4]: 0.0
Funcion: 2  var[5]: 0.0
Funcion: 2  var[6]: 0.0
Funcion: 2  var[7]: 0.0
Funcion: 2  var[8]: 6.140768527984619E-5
Funcion: 2  var[9]: 0.0
Funcion: 2  var[10]: 0.0
Funcion: 2  var[11]: 6.140768527984619E-5
Funcion: 2  var[12]: 6.140768527984619E-5
Funcion: 2  var[13]: 0.0
Funcion: 2  var[14]: 0.0
Funcion: 2  var[15]: 2.5010108947753906E-4
Funcion: 2  var[16]: 0.0
```

```
Funcion: 2  var[17]: 0.0
Funcion: 2  var[18]: 0.0
Funcion: 2  var[19]: 0.0
Funcion: 2  var[20]: 6.140768527984619E-5
Funcion: 2  var[21]: 0.0
Funcion: 2  var[22]: 0.0
Funcion: 2  var[23]: 0.0
Funcion: 2  var[24]: 0.0
Funcion: 2  var[25]: 0.0
Funcion: 2  var[26]: 0.0
Funcion: 2  var[27]: 0.0
Funcion: 2  var[28]: 6.140768527984619E-5
Funcion: 2  var[29]: 0.0
```

Los valores se aproximan a 0, que es el mínimo conocido.
En **promedio f(x) = 2.06182E-05**. Y la desviación estándar **ds(f(x))= 0.00026911**.

Función 3. Ésta función se corrió para G*2 = 2,000,000, 10 veces, donde G es el número de iteraciones máximas del algoritmo.

```
Funcion: 3  var[0]: -1252.8010361486004
Funcion: 3  var[1]: -1254.573544745903
Funcion: 3  var[2]: -1252.33265811792
Funcion: 3  var[3]: -1255.455349875264
Funcion: 3  var[4]: -1251.3033304769338
Funcion: 3  var[5]: -1255.308971425755
Funcion: 3  var[6]: -1255.7113840258098
Funcion: 3  var[7]: -1246.7820279781986
Funcion: 3  var[8]: -1251.5010723292621
Funcion: 3  var[9]: -1254.6841459935458
```

El mínimo conocido para n = 2 es -418.9828. Pero en éste experimento se utilizó n =3. En **promedio f(x) = -1253.04535**. Y la desviación estándar **ds(f(x)) = 8.262995676**.

Función 4. Ésta función se corrió para G = 1,000,000, 10 veces, donde G es el número de iteraciones máximas del algoritmo.

```
Funcion: 4  var[0]: 41.71828182845905
Funcion: 4  var[1]: 41.71828182845905
Funcion: 4  var[2]: 41.71828182845905
Funcion: 4  var[3]: 41.71828182845905
Funcion: 4  var[4]: 41.71828182845905
Funcion: 4  var[5]: 41.71828182845905
Funcion: 4  var[6]: 41.71828182845905
Funcion: 4  var[7]: 41.71828182845905
Funcion: 4  var[8]: 41.71828182845905
Funcion: 4  var[9]: 41.71828182845905
```

El mínimo conocido es 0. En este experimento el valor mínimo resultante fue aproximado a 41. Es probable que el algoritmo se haya atorado en un mínimo local.
En **promedio f(x) = 41.7182818**. Y la desviación estándar **ds(f(x))= 0**.

Conclusiones

Existen funciones engañosas que llevan a los algoritmos de optimización a valores que no son los valores óptimos y que además es difícil salir de ahí. El RMHC en algunas ocasiones logra salir de esos picos engañosos y corregir su camino debido a la aleatoriedad del cambio de los bits del individuo o solución.

Referencias

[1] Apuntes de Complejidad y Computabilidad. Otoño 2014.
[2] Jason Brownlee. Clever Algorithms: Nature-Inspired Programming Recipes. 2014.
http://www.cleveralgorithms.com/nature-inspired/stochastic/hill_climbing_search.html