Seminario de solucion de problemas de inteligencia artificial

Practica 03

Javier Emmanuel Astudillo Salamanca

213496188

```
Seminario de solucion de problemas de inteligencia artificial
Practica 03
Javier Emmanuel Astudillo Salamanca
213496188
   Descripción
   Marco teorico
      Heuristica
      Metaheuristica
       Algoritmos genéticos evolutivos
          <u>Descripción</u>
          Genotipo
          Fitness
          Metodo de seleccion
          <u>Apareamiento</u>
          Condicion de parada
   Mutacion
   Desarrollo
      Individuo
      Poblacion
      Método genera alelo
      Metodo de eleccion
      <u>Apareamiento</u>
       Selección de los mejores
      Ciclo principal
   Resultados
       Mutacion 0%
          Generacion 0
          Generacion 50
          Generacion 100
       Mutacion 1%
          Generacion 0
          Generacion 50
          Generacion 100
      Mutacion 2%
          Generacion 0
          Generacion 50
          Generacion 100
       Mutacion 5%
          Generacion 0
```

Generacion 50 Generacion 100

Conclusiones

<u>Nota</u>

Referencias

Descripción

Considere el problema de

$$\frac{\min}{x} f(x) = \sum_{i=1}^{n} x_i^2$$

En un dominio de búsqueda [-5, +5], para una esfera dimension 20. El límite de generaciones a 100, el tamaño de la población de 20, compare los rangos de mutación de 1%, 2%, 5%.

Marco teorico

Heuristica

Algunos problemas de optimización son difíciles de solucionar por medio de métodos tradicionales debido a la complejidad, cantidad de respuestas o simplemente por que con el poder de cómputo actual tardaríamos una eternidad en resolverlos.¹

Es ahí cuando entran en acción las metaheurísticas que nos define heurística de la siguiente manera:

"Procedimiento simple, a menudo basado en el sentido común, que se supone que ofrecerá una buena solución (aunque no necesariamente la óptima) a problemas difíciles, de un modo fácil y rápido". (Zanakins y Evans, 1981) "²

Así pues se utilizan cuando no existe un método exacto de resolución, cuando el que existe demora bastante tiempo para encontrar una respuesta óptima o cuando se tienen limitaciones de tiempo.

Metaheuristica

Las metaheurísticas como las heurísticas, tampoco garantizan la obtención de un óptimo al problema, a diferencia de las heurísticas las metaheurísticas tratan de evitar los óptimos locales, enfocados en el óptimo global.

La lógica de las técnicas metaheurísticas es similar: el punto de partida es una solución (o conjunto de soluciones) que típicamente no es óptima. A partir de ella se obtienen otras

¹ (2014). Algoritmos Evolutivos y Algoritmos Genéticos. Retrieved September 6, 2016, from http://www.it.uc3m.es/~jvillena/irc/practicas/estudios/aeaq.

² (2013). Técnicas metaheurísticas - Ingeniería de Organización y Logística. Retrieved September 6, 2016, from http://www.iol.etsii.upm.es/arch/metaheuristicas.pdf.

parecidas, de entre las cuales se elige una que satisface algún criterio, a partir de la cual comienza de nuevo el proceso. Este proceso se detiene cuando se cumple alguna condición establecida previamente.

Algoritmos genéticos evolutivos

Su raíz procede de los algoritmos bioinspirados, basados en imitar el comportamiento o algún fenómeno existente en la naturaleza para resolver el problema, emplean métodos heurísticos no deterministas de "búsqueda", "aprendizaje", evolución.

Descripción

En el caso del algoritmo en cuestión, se basa en la máxima evolutiva de "Los más aptos sobreviven y dejan descendencia", por lo tanto este algoritmo utiliza una función para evaluar a los más aptos, lo que se llama "fitness" esta función sirve como un indicador de cuáles son los individuos más aptos y cuales son los que nos interesan para "dejar descendencia", la idea de que se reproduzcan los más aptos es que con cada generación se tenga una solución más cercana al óptimo global y que cuando se alcance la condición de parada se tenga un resultado lo más cercano al óptimo global posible.

Genotipo

Cada individuo cuenta con un genotipo que consta de valores binarios "0s" y "1s" agrupándolos y representando estos grupos características o el "fenotipo" del individuo.

En este problema en particular el genotipo consta de un grupo de 4 genotipos , uno por cada variable y consta de 5 alelos en lugar de 4.

Fitness

Es una magnitud que mide qué tan apto es el individuo (solución) con respecto a sus pares, indicándonos un mayor valor que es más cercano al óptimo.

Metodo de seleccion

Existen varios métodos de selección siendo los más populares el de "ruleta" o "torneo", en el presente trabajo se utilizara el método de ruleta. Que consiste en evaluar a cada individuo para obtener su "fitness" y en base a ese valor otorgarle una probabilidad, a mayor "fitness" mayor probabilidad de ser elegido.

Una vez calculada esa probabilidad mediante una función "random" que elija dos individuos diferentes para aparearse.

Apareamiento

Una vez que se han elegido los dos individuos para aparearse el método que usaremos para hacer esto es el de "swap" que consiste en tomar el genotipo de cada individuo y mediante una función random elegir un número entre (1 y n -1), n siendo el número de genes que tiene la población.

Una vez teniendo este número, se procede a hacer un intercambio de los genes a partir de esa posición por ejemplo:

X = 2

```
Individuo 1 : [ 3.23, 2.22, -1.45, ... ]
Individuo 2 : [ 2.45, -4.56, -1.5, ... ]

Hijo 1 : [ 3.23, -4.56, -1.5, ... ]

Hijo 2 : [ 2.45, 2.22, -1.45, ... ]
```

Condicion de parada

Esta condición de parada es arbitraria y puede ser un número determinado de generaciones, determinado tiempo, cuando las generaciones no cambien mas, etc.

Para este trabajo se decidió establecer la condición de parada en 100 generaciones.

Mutacion

Para darle variedad a la población y buscar un individuo que se acerque mas a la solución ideal sin importar si estuvo o no presente en la primera generación, se implementa la mutación.

Se elige un porcentaje de mutación y a partir de este se calcula el número de alelos que tienen que cambiar en la población, es decir si se tiene una población de 20 individuos y cada uno tiene 20 alelos el total de la poblacion tendra "400 alelos".

Si se tiene un porcentaje de mutación del 1%, mutaremos 4 alelos.

```
20 * 20 = 400 % 1 = 4
```

def mutarPoblacion(self):

```
numeroMutantes = int(math.ceil(self.tamanoPoblacion * self.numAlelos * self.porcentajeMutacion))

for i in range( numeroMutantes ):
    aux = self.poblacion[ random.randint( 0, self.tamanoPoblacion - 1 ) ]
    aux.setMutacion( random.randint( 0, self.numAlelos - 1 ) , self.generaAlelo() )
```

Una vez que se calcula el número de alelos que tiene que ser mutados, se inicia un ciclo de que itere ese número de veces y en cada iteración se elige un individuo de la población y un alelo de ese individuo se asigna un valor en el rango de [- 5, +5].

Desarrollo

Primero declaramos las clases y atributos que nos ayudaran a resolver el problema:

Individuo

Que cuenta con 3 atributos: genoma, fitness.

Un constructor que toma por parámetro "genoma".

Se agregó el atributo fx para guardar el resultado de la función original y a diferencia de la práctica anterior el genoma y el fenotipo son arreglos.

Poblacion

```
class Poblacion(object):
    """docstring for Poblacion"""

def __init__(self, funcionFitness ):
```

```
super(Poblacion, self).__init__()
self.poblacion = []
self.poblacionNueva = []
self.funcionFitness = funcionFitness
self.totalFitness = 0
```

Cuenta con 2 atributos "población" y "hijos" que son listas que almacenan la población actual de la que se eligen los individuos más aptos y "hijos" que almacena los hijos de los individuos que se aparean.

```
Método genera alelo
```

```
def generaAlelo(self): return (random.random() * random.randint( self.minAlelo, self.maxAlelo ) )
```

Metodo de eleccion

Para este trabajo se eligió el método de ruleta que se implementa de la siguiente manera:

```
def ruleta( self ):
    totalProbabilidad = 0
    elegido = round( random.random() , NUM_DECIMALES )

for individuo in self.poblacion:
    aux = (individuo.fitness / self.totalFitness)
    totalProbabilidad += aux

if elegido <= totalProbabilidad:
    return individuo</pre>
```

Se inicializa la variable auxiliar "totalProbabilidad" a 0 que sirve como acumulador del rango de probabilidad que tiene cada individuo, la variable "elegido" guarda un valor aleatorio entre 0 y 1 con 4 cifras decimales y por cada individuo se asegura que el valor elegido no haya caído en el rango asignado a ese individuo, si es así regresa al individuo.

Apareamiento

Para la función aparear se eligió el método "swap".

Primero se elige el número de alelos que se van a conservar por cada individuo, con una función random que elige un número entre (1 y n - 1), siendo n el número total de alelos.

Enseguida se emplean 2 variables auxiliares que guardaran los genomas resultantes del swap e inicializará 2 objetos individuo inicializarlos con el genotipo resultante y el fenotipo que se obtiene haciendo una llamada a "diccionario Genómico" y se regresan los dos individuos resultantes.

```
def aparear(self, padre, madre):
    numAlelos = random.randint( 1, len( padre.genoma ) - 1 )

genoma1 = padre.genoma[ : numAlelos ] + madre.genoma[ numAlelos : ]
    genoma2 = madre.genoma[ : numAlelos ] + padre.genoma[ numAlelos : ]

return Individuo( genoma1 ) , Individuo( genoma2)
```

Selección de los mejores

Para la elección de los mejores para aparearse se utiliza la ruleta que regresa a los individuos más aptos es necesario que ambos individuos sean diferentes, por lo que es necesario asegurarse que esto no ocurra, una vez que tenemos a dos individuos diferentes, se aparea y se integran a la lista "hijos".

```
def seleccionMejores( self ):
    aux1 = self.ruleta()

flag = True

while( flag ):
    aux2 = self.ruleta()

if aux1 != aux2:
    flag = False

hijos = self.aparear( aux1, aux2 )

self.hijos.append( hijos[0] )
self.hijos.append( hijos[1] )
```

Ciclo principal

```
def cicloPrincipal(self):
numGeneraciones = 0
```

```
p.inicializaPoblacion()
p.calculaFitness()
p.sortPoblacion()
print("Generacion " + str(numGeneraciones) )
p.imprimirPoblacion()
while (numGeneraciones <= self.maxGeneraciones):
       #print(len(self.poblacion))
       self.hijos = []
       self.resetTotalFltness()
       self.sortPoblacion()
       self.calculaFitness()
       while len(self.hijos) < self.tamanoPoblacion:
               self.seleccionMejores()
               self.mutarPoblacion()
       self.setPoblacion( self.hijos )
       numGeneraciones += 1
       if (numGeneraciones \% 50) == 0:
               print("Generacion " + str(numGeneraciones) )
               p.calculaFitness()
               p.sortPoblacion()
               p.imprimirPoblacion()
```

Inicializamos el "numGeneraciones" a 0, así como la variable "población" pasándole como parámetro la función con la que evaluará el "fitness", inmediatamente después se inicializa la población.

Se inicia un ciclo que tiene como condición de paro que el número de generaciones sea igual al máximo.

Al inicio de cada ciclo se eliminan todos los elementos en "hijos", poner a 0 el "totalFitness" y asignar a cada individuo el fitness actual.

Así mismo se entra a un segundo ciclo que ejecuta la selección de los mejores individuos, los aparea e inserta los hijos a la nueva población, repitiendo esto hasta que "hijos" tenga el

tamaño de requerido, una vez que se tiene la población "hijos", se procede a mutar esa población.

Por último se asigna "hijos" a la "población" y se incrementa el contador de generaciones, se repite el ciclo hasta que se cumple con la condición de paro.

Resultados

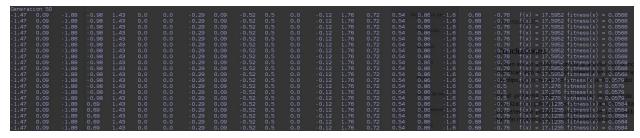
Mutacion 0%

Generacion 0



Peor individuo \rightarrow Fitness : 0.0084 f(x) : 119.7352 Mejor individuo \rightarrow Fitness : 0.0245 f(x) : 40.7922

Generacion 50



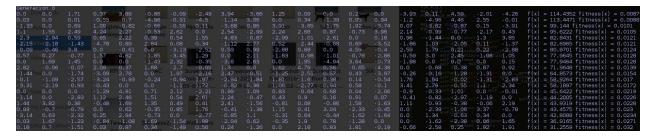
Peor individuo \rightarrow Fitness: 0.0568 f(x): 17.5952 Mejor individuo \rightarrow Fitness: 0.0584 f(x): 17.1235



Peor individuo \rightarrow Fitness : 0.0568 f(x) : 17.5952 Mejor individuo \rightarrow Fitness : 0.0568 f(x) : 17.5952

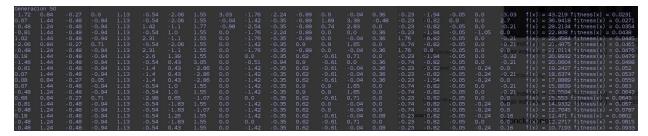
Mutacion 1%

Generacion 0

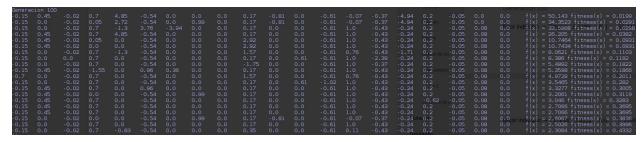


Peor individuo \rightarrow Fitness : 0.0087 f(x) : 114.4352 Mejor individuo \rightarrow Fitness : 0.032 f(x) : 31.2559

Generacion 50



Peor individuo \rightarrow Fitness : 0.0231 f(x) : 43.2190 Mejor individuo \rightarrow Fitness : 0.0933 f(x) : 10.7193



Peor individuo \rightarrow Fitness : 0.0199 f(x) : 50.143 Mejor individuo \rightarrow Fitness : 0.4332 f(x) : 2.3084

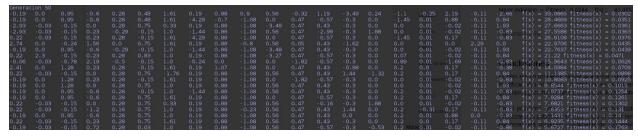
Mutacion 2%

Generacion 0

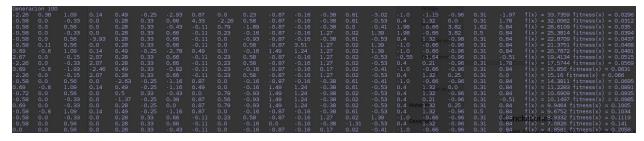


Peor individuo \rightarrow Fitness : 0.0076 f(x) : 132.4154 Mejor individuo \rightarrow Fitness : 0.0244 f(x) : 41.0653

Generacion 50



Peor individuo \rightarrow Fitness: 0.0302 f(x): 33.0865 Mejor individuo \rightarrow Fitness: 0.1763 f(x): 5.6727



Peor individuo \rightarrow Fitness: 0.0296 f(x): 33.7359 Mejor individuo \rightarrow Fitness: 0.2058 f(x): 4.8581

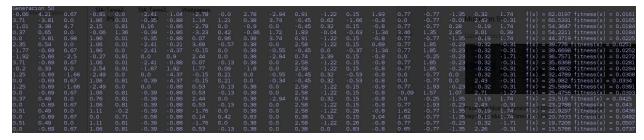
Mutacion 5%

Generacion 0

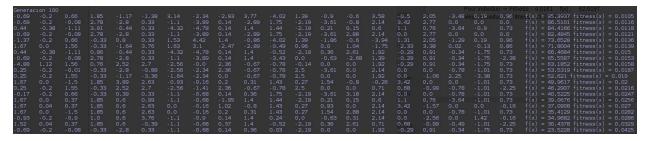


Peor individuo \rightarrow Fitness: 0.0094 f(x): 106.4150 Mejor individuo \rightarrow Fitness: 0.0322 f(x): 31.0166

Generacion 50



Peor individuo \rightarrow Fitness: 0.0161 f(x): 62.0197 Mejor individuo \rightarrow Fitness: 0.0737 f(x): 13.5766



Peor individuo \rightarrow Fitness: 0.0105 f(x): 95.3937 Mejor individuo \rightarrow Fitness: 0.0425 f(x): 23.5228

Conclusiones

Es una variación al algoritmo usado anteriormente, pues se elimina el fenotipo y se da una mayor variedad a los alelos, por lo que sería imposible darles un valor binario a tantas variables.

En esta ocasión se insertó la función mutarPoblacion() que nos sirvió para mejorar la respuesta, pues de no haberlo hecho solo se hubiera podido buscar un mínimo local, pues la muestra de individuos se limitaría a alguna combinación de la generación inicial alejándose del mínimo global.

Variando el porcentaje de mutación desde 0%, 1%, 2% y 5% las conclusiones a las que llegué fueron que cuando no hay mutación, se logra una buena combinación de genes pero se limita solamente a aquellos que fueron elegidos en la primera generación por lo que generalmente no es un buen maximo global.

Con 1% fue con el porcentaje que mejor resultados obtuve, pues permite encontrar un mínimo local entre la población inicial y le permite mantenerse a lo largo de las generaciones a los mejores individuos, con ese pequeño porcentaje de variedad permite explorar resultados que no aparecen en la primera generación, fue la que mejor resultados me dio.

2% de mutación no genera malos resultados, siguen siendo cercanos al mínimo global, sin embargo son ligeramente peores que con 1%, un valor intermedio más cercano al 1% creo que seria el optimo.

Con el 5% de mutación se genera demasiada aleatoriedad lo que repercute en la evaluación del fitness y en el resultado final.

Nota

Había un inquietud que tenia que probar y era que la función que nos mostró en clase para generar un número aleatorio entre -5 y 5 , si en verdad era uniformemente aleatoria.

Realice una prueba con el siguiente codigo python:

```
import random
```

Lo que hace es crear un arreglo de tamaño 11 conteniendo ceros e iterando 100,000 veces usando una función parecida a la que nos dio y otra usando "random.uniform(- 5 , 5)", obteniendo el número flotante, lo redondea al entero mas proximo y lo castea a entero para poder obtener un índice del arreglo y agrega un offset de +5 para que no este fuera de rango.

Es decir:

Cuando el valor sea -4.9 será redondeado a -5.0, casteado a -5 y con el offset corresponde al índice 0 del arreglo.

```
a [0] y sumándole 1 unidad cada vez que salga ese numero. a[0] += 1
```

Y así con todos los números del rango con la siguientes respuestas.

```
[936, 3007, 5587, 9402, 16162, 29839, 16071, 9378, 5658, 3062, 898] (4879, 10111, 9855, 9927, 10054, 10013, 10058, 9886, 10245, 9929, 5043]
```

Por falta de tiempo y exceso de cansancio, no formatee la salida, pero el arreglo de arriba corresponde a los valores obtenidos con la función en forma de:

```
random() * randint( -5, 5)
```

En el cual es mucho más probable que genere un valor cercano a cero, es 30 veces más probable que genere un 0 que un -5, como en este caso el mejor fitness se logra con 0's es por eso que nos da tan buenas respuestas, sin embargo está viciado desde su concepción, en un caso en el que el mejor fitness no se logre con 0's no daría una respuesta ni siquiera próxima.

El arreglo de abajo muestra los resultados de la función random uniforme, lo cual nos muestra que si es uniforme, siendo los valores menos probables -5 y 5, debido que solo los números entre -4.6 y -5 e redondeará a -5, lo cual es la mitad de por ejemplo -3 [-3.4 , -2.6].

Encontre esta funcion para generar flotantes aleatorios uniformes en c++

```
float randomFloat(float a, float b)
{
    float random = ((float) rand()) / (float) RAND_MAX;
    float diff = b - a;
    float r = random * diff;
    return a + r;
}
```

http://stackoverflow.com/questions/5289613/generate-random-float-between-two-floats

Espero pueda hacer sus propias pruebas y me pueda responder si estoy en lo correcto o cometi algún error.

Referencias

(2014). Algoritmos Evolutivos y Algoritmos Genéticos. Retrieved September 6, 2016, from http://www.it.uc3m.es/~ivillena/irc/practicas/estudios/aeaq.

(2013). Técnicas metaheurísticas - Ingeniería de Organización y Logística. Retrieved September 6, 2016, from http://www.iol.etsii.upm.es/arch/metaheuristicas.pdf.