Seminario de solucion de problemas de inteligencia artificial

Practica 02

Javier Emmanuel Astudillo Salamanca

213496188

Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Descripcion

Considere el problema de
$f(x) = a + 2b + 3c + 4d = 30 \mid x\ 0 <= x >= 30$

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]

Asi 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 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.

---

[1] (2014). Algoritmos Evolutivos y Algoritmos Genéticos. Retrieved September 6, 2016, from http://www.it.uc3m.es/~jvillena/irc/practicas/estudios/aeag.
[2] (2013). Técnicas metaheurísticas - Ingeniería de Organización y Logística. Retrieved September 6, 2016, from http://www.iol.etsii.upm.es/arch/metaheuristicas.pdf.

Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

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.

Descripcion

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 po cada variable y consta de 5 alelos en lugar de 4.

Fenotipo

Dado el genotipo del individuo se puede descifrar cuales son sus características.
En esta práctica el fenotipo se calcula por cada variable que compone el arreglo de genotipos.

```
def getFenotipo(self, genoma):
                return  self.diccionarioGenomico[ genoma[0] ] ,\
                        self.diccionarioGenomico[ genoma[1] ] ,\
                        self.diccionarioGenomico[ genoma[2] ] ,\
                        self.diccionarioGenomico[ genoma[3] ]
```

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.

Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

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 : 10 01001
Individuo 2 : 11 10110

Hijo 1 : 10 01001
Hijo 2 : 11 10110

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 30 generaciones.

Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Desarrollo

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

Individuo

```
class Individuo(object):
	"""docstring for Individuo"""
	def __init__(self, genoma, fenotipo):
		super(Individuo, self).__init__()

		self.genoma = genoma
		self.fenotipo = fenotipo
		self.fitness = -1
		self.fx = None

	def setFitness(self, fitness):		self.fitness = fitness
	def setFx(self, fx):	self.fx = fx
```

Que cuenta con 3 atributos: genoma, fenotipo, fitness.
Un constructor que toma por parametros “genoma” y “fenotipo”.
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"""

	diccionarioGenomico = \
		{
		 "00000" : 0,   "00001" : 1.,   "00010" : 2.,   "00011" : 3.,
		 "00100" : 4.,  "00101" : 5.,   "00110" : 6.,   "00111" : 7.,
		 "01000" : 8.,  "01001" : 9.,   "01010" : 10.,  "01011" : 11.,
		 "01100" : 12., "01101" : 13.,  "01110" : 14.,  "01111" : 15.,

		 "10000" : 16., "10001" : 17.,  "10010" : 18.,  "10011" : 19.,
		 "10100" : 20., "10101" : 21.,  "10110" : 22.,  "10111" : 23.,
		 "11000" : 24., "11001" : 25.,  "11010" : 26.,  "11011" : 27.,
		 "11100" : 28., "11101" : 29.,  "11110" : 30.,  "11111" : 30.
```

```
            }

    def __init__(self, funcionFitness ):
            super(Poblacion, self).__init__()

            self.poblacion = []
            self.poblacionNueva = []

            self.funcionFitness = funcionFitness
            self.totalFitness = 0
```

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

Asi mismo cuenta con una variable estática “diccionarioGenomico” que es un diccionario al que se le hacen consultas, se le pregunta por el genoma y devuelve el fenotipo.

Ej: diccionarioGenomico[“0000”] → -4.0

Se cambió el diccionario genomico para contener los números del 0 al 30, se repitió el 30 para tener números binarios cerrados y la población consta de 128 individuos.

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
```

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 asi 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, self.diccionarioGenomico[genoma1] ) , \
                Individuo( genoma2, self.diccionarioGenomico[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 “poblacionNueva”.

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

        flag = True

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

                if aux1 != aux2:
                        flag = False
```

```
        hijos = self.aparear( aux1, aux2 )

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

Ciclo principal

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

        p = Poblacion(funcionFitness)
        p.inicializaPoblacion()
        p.calcularFitness()

        while( numGeneraciones != maxGeneraciones ):
                del( p.poblacionNueva[:] )
                p.resetTotalFitness()
                p.calcularFitness()

                while( len( p.poblacionNueva) != MAX_POBLACION ):
                        p.seleccionMejores()

                p.setPoblacion( p.poblacionNueva )
                p.calcularFitness()
                p.sortPoblacion()

                numGeneraciones += 1
```

Inicializamos el “numGeneraciones” a 0, asi como la variable “poblacion” pasandole como parametro la función con la que evaluara 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 “poblacionNueva”, poner a 0 el “totalFitness” y asignar a cada individuo el fitness actual.

Asi 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 la “nuevaPoblacion” tenga el tamaño de requerido.

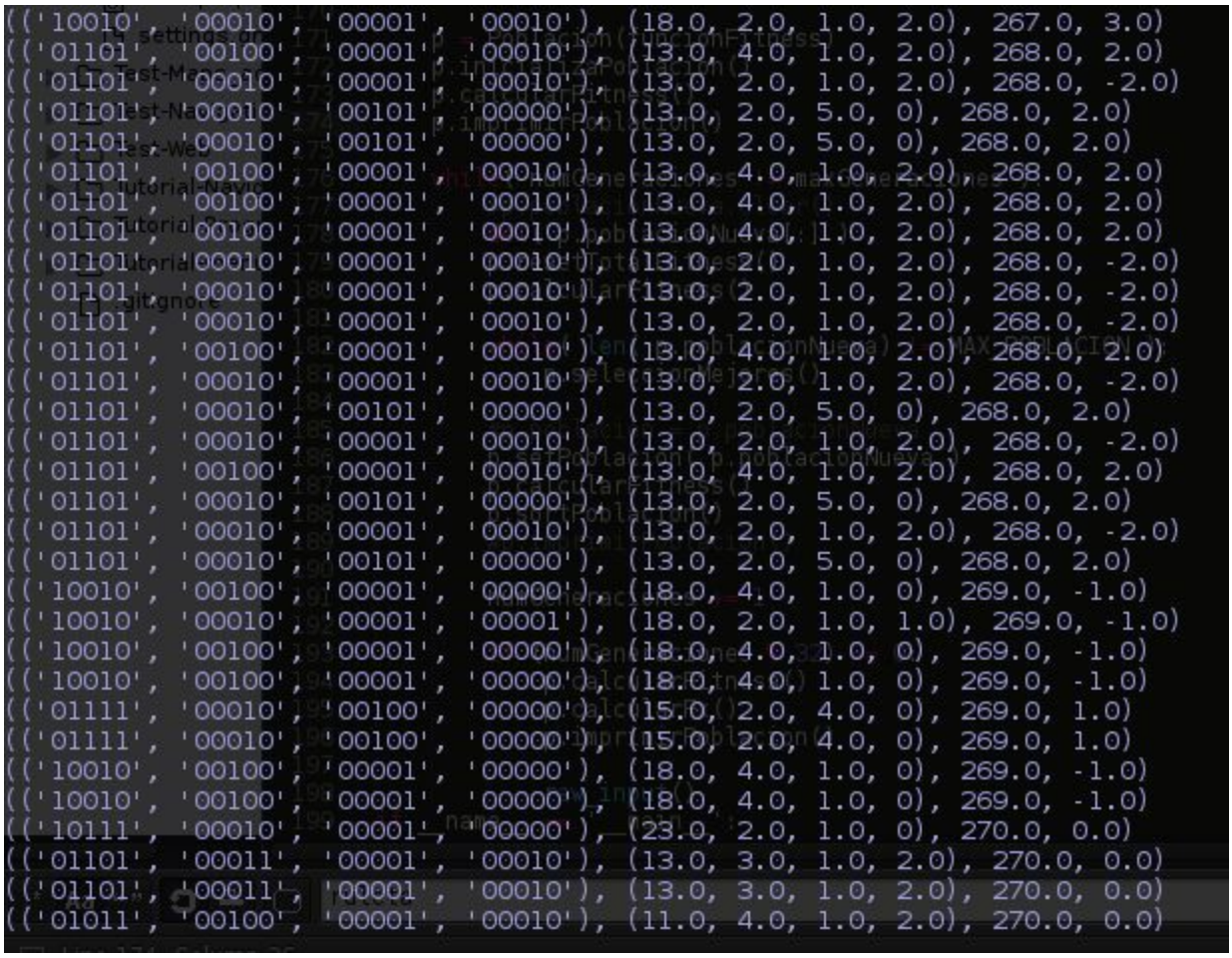
Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Por último se asigna la “nuevaPoblacion” a la “poblacion” y se incrementa el contador de generaciones, se repite el ciclo hasta que se cumple con la condición de paro.

Resultados

Generacion 32

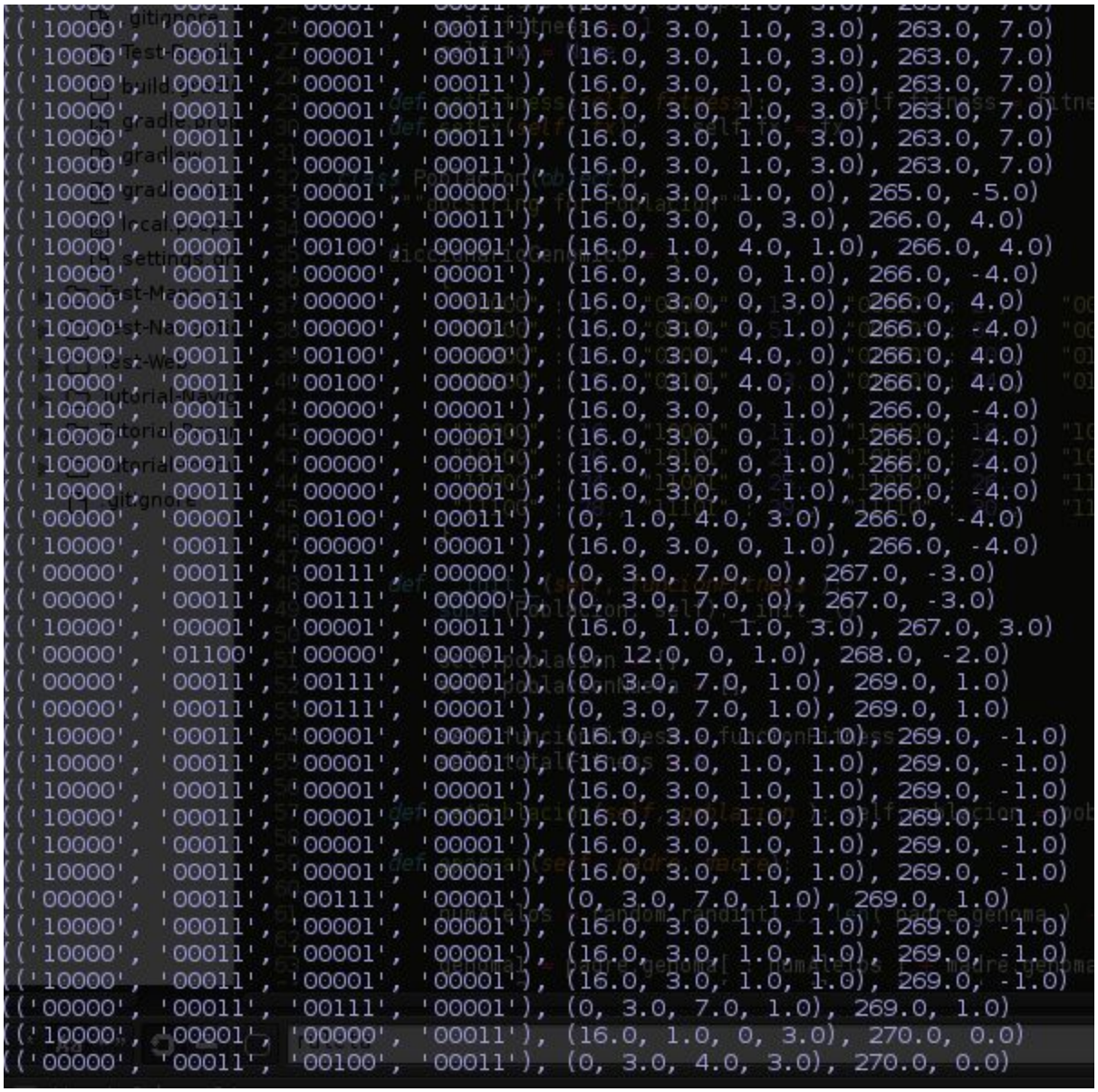
Fitness minimo: 227 (A = 2 , B = 4, C = 1 , D = 15)
Fitness maximo: 270 (A = 11 , B = 4, C = 1 , D = 2)



Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Generacion 64

Fitness minimo: 227 (A = 16 , B = 12, C = 7 , D = 3)
Fitness maximo: 270 (A = 0 , B = 3, C = 4 , D = 3)



Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Generacion 96

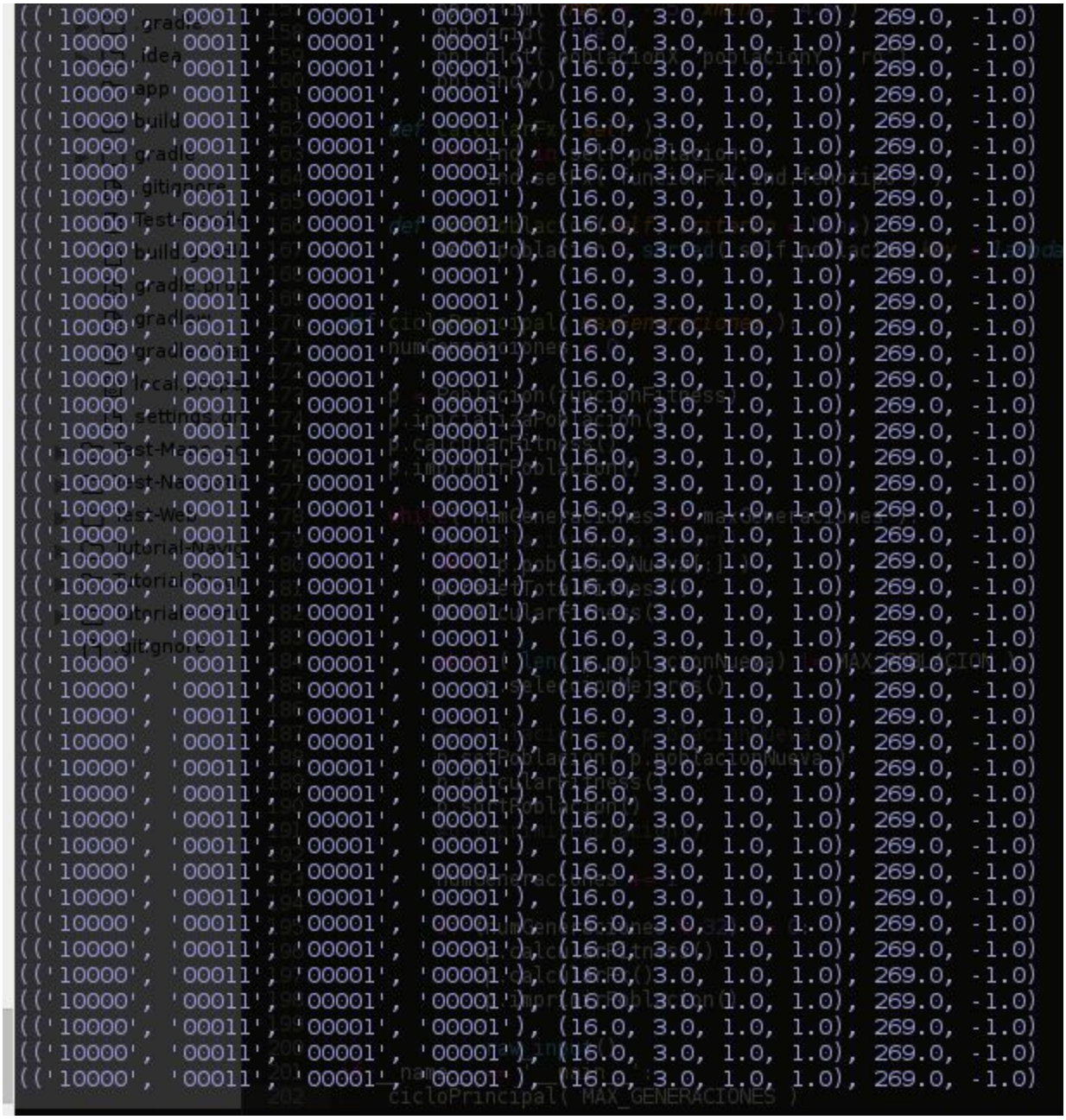
Fitness minimo: 249 (A = 16 , B = 12, C = 4 , D = 1)
Fitness maximo: 269 (A = 0 , B = 12, C = 1 , D = 1)

```
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00001', '00100', '00001'), (16.0, 1.0, 4.0, 1.0), 266.0, 4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00100', '00000'), (16.0, 3.0, 4.0, 0), 266.0, 4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00100', '00000'), (16.0, 3.0, 4.0, 0), 266.0, 4.0)
(('10000', '00011', '00000', '00011'), (16.0, 3.0, 0, 3.0), 266.0, 4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('10000', '00011', '00000', '00001'), (16.0, 3.0, 0, 1.0), 266.0, -4.0)
(('00000', '01100', '00000', '00001'), (0, 12.0, 0, 1.0), 268.0, -2.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('00000', '01100', '00001', '00001'), (0, 12.0, 1.0, 1.0), 269.0, 1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('00000', '01100', '00001', '00001'), (0, 12.0, 1.0, 1.0), 269.0, 1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('10000', '00011', '00001', '00001'), (16.0, 3.0, 1.0, 1.0), 269.0, -1.0)
(('00000', '01100', '00001', '00001'), (0, 12.0, 1.0, 1.0), 269.0, 1.0)
```

Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Generacion 128

Fitness minimo: 249 (A = 16 , B = 12, C = 7 , D = 3 , f(x) = 43)
Fitness maximo: 270 (A = 0 , B = 3, C = 4 , D = 3 , f(x) = 30)



Javier Emmanuel Astudillo Salamanca
213496188
Ing. Computación

Conclusiones

Si bien era un problema similar al anterior con la inclusión de las otras variables fue necesario modificar algo el código en especial en la manera en que se obtiene el fenotipo a partir del genotipo, asi como la manera en que se lleva a cabo el apareamiento.

En el problema anterior era posible recorrer todo el diccionario genomico y de esta forma inicializar a la población, en este caso al haber tantas opciones como 31^4 opciones se decidió por tener una población únicamente de 128 individuos, por lo que hubo que hacer uso de la función random para generar la población inicial.

La manera de generar el genotipo fue teniendo 32 genes iniciales y mediante una agrupación de 4 genes formar el genoma que representaban a las 4 variables.

Como comentario aunque había cerca de 1 millón de posibles soluciones para el problema planteado, sigue siendo una cantidad irrisoria para una computadora moderna, pues se desarrolló el método voraz para resolver este problema y resulto ser más rápido y certero que el presente algoritmo, supongo que si incrementara la cantidad de soluciones aún más probaria se haría notorio la verdadera eficiencia del presente algoritmo.

Referencias

(2014). Algoritmos Evolutivos y Algoritmos Genéticos. Retrieved September 6, 2016, from http://www.it.uc3m.es/~jvillena/irc/practicas/estudios/aeag.
(2013). Técnicas metaheurísticas - Ingeniería de Organización y Logística. Retrieved September 6, 2016, from http://www.iol.etsii.upm.es/arch/metaheuristicas.pdf.