INSTITUTO POLITÉCNICO NACIONAL

Centro de Investigación en Computación

ASIGNATURA:

Metaheurísticas

Actividad #25:

Guía Taller No.14

"Solución de problemas mediante

Algoritmos Genéticos Estacionarios"

PROFESORA:

Dra. Yenny Villuendas Rey

Centro de Investigación en Computación
Instituto Politécnico Nacional

PRESENTA:

Juan René Hernández Sánchez

Adriana Montserrat García Carrillo

1. Introducción

Los algoritmos genéticos (AG) son algoritmos inspirados en la selección natural que han logrado semejar todo el proceso evolutivo de los seres vivos para resolver problemas de optimización, búsqueda y aprendizaje en las máquinas

La teoría de los algoritmos genéticos fue desarrollada por John H. Holland, sus colegas y sus estudiantes en la Universidad de Michigan, en un proyecto que buscaba explicar el proceso de adaptación natural de los sistemas y diseñar sistemas artificiales que conservaran los más importantes mecanismos de los sistemas naturales. Casi todos los conceptos de los algoritmos genéticos están basados en conceptos de biología y genética.

Las características de los seres vivos se encuentran registradas en los cromosomas y los genes, lo que hace que cada individuo sea diferente dentro de una población. En algoritmos genéticos los cromosomas se representan mediante cadenas (strings) y cada posición de la cadena representa un gen.

El primer paso desarrollar un algoritmo genético es codificar el parámetro deseado como una cadena y usar un alfabeto. La longitud de un cromosoma está determinada por el número de genes que lo conforman.

El conjunto ordenado de genes de un cromosoma se conoce como genotipo. De acuerdo con los valores que tengan los genes de una persona podremos decir que es alta o baja, de ojos azules o verdes, de cabello liso o rizado, etc. Esto se denomina fenotipo, y en los algoritmos genéticos corresponde a la decodificación de un cromosoma.

Cada cromosoma es una solución, buena o mala, del problema. A medida que el algoritmo genético se va desarrollando las soluciones se acercan cada vez más a la óptima. Simulando los sistemas de selección natural a los que se encuentran sometidos los organismos vivos, los algoritmos genéticos utilizan tres operadores básicos para solucionar el problema: Selección, Cruzamiento y Mutación.

2. Desarrollo

Asignatura: Metaheurísticas
Actividad No.25
Guía Taller No.14
Título: Solución de problemas mediante Algoritmos Genéticos Estacionarios

Contenido:

- Métodos heurísticos de solución de problemas.
- Algoritmos Genéticos Estacionarios

Objetivo: Implementar algoritmos genéticos, en lenguajes de alto nivel, para la solución de problemas de la profesión.

1. Enuncie las ventajas y desventajas de los Algoritmos Genéticos

- **Ventajas:**

a) Paralelización: Los algoritmos evolutivos son por naturaleza paralelos. Independientemente de cómo se realice la paralelización, la idea clave es distribuir la carga computacional en varios procesadores y así acelerar la ejecución general del AG.
b) Hibridación: puede ser una forma muy eficaz de mejorar el rendimiento de los AG. La forma más común de hibridación es acoplar los AG con técnicas de búsqueda local e incorporar conocimientos específicos del dominio en el proceso de búsqueda.
c) Continuación en el tiempo: las capacidades de la mutación y la recombinación se utilizan de forma óptima para obtener una solución de la mayor calidad posible con recursos computacionales limitados. La utilización del tiempo (o la continuación) explota la compensación entre la búsqueda de soluciones con grandes poblaciones y una única época de convergencia o utilizar una pequeña población con múltiples épocas de convergencia.
d) Relajación de la evaluación: una evaluación del fitness precisa, pero computacionalmente costosa es sustituida por una estimación del fitness menos precisa, pero de bajo costo computacional.

- **Desventajas:**

a) Los AG tienen muchos parámetros a configurar.
b) Por lo general mejoran la calidad de la solución, pero se produce un aumento del tiempo de cálculo.
c) Pueden tardar mucho en converger, o no converger en absoluto, dependiendo en cierta medida de los parámetros que se utilicen, el tamaño de la población, el número de generaciones, entre otros.
d) Pueden converger prematuramente. La convergencia prematura y el utilizar poblaciones muy pequeñas podría incluso extinguir a dichas poblaciones o mermar su diversidad.

2. Diga las diferencias entre fenotipo y genotipo.

Diferencias:

a. El genotipo comprende a los genes y el fenotipo a la expresión de los componentes.

El conjunto ordenado de genes de un cromosoma se conoce como genotipo y se denomina fenotipo a la decodificación de un cromosoma.

b. El genotipo es único, el fenotipo puede ser igual.

c. El fenotipo depende del genotipo, pero no viceversa

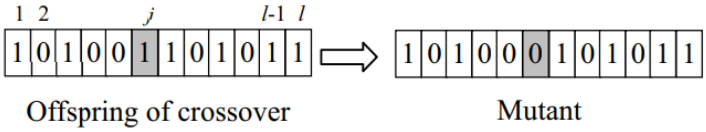
3. Mencione 3 operadores de Mutación. Explique el funcionamiento de uno de ellos.

- Mutación de cambio de bits, mutación normal y mutación uniforme.

- **Mutación de cambio de bits**

Cada gen de un individuo se muta con una probabilidad p_m, llamada tasa de mutación.

Siempre que deba mutar el gen j, se hace un cambio de bit, es decir, se cambia de 1 a 0 o de 0 a 1. A este operador se le llama mutación de cambio de bits.



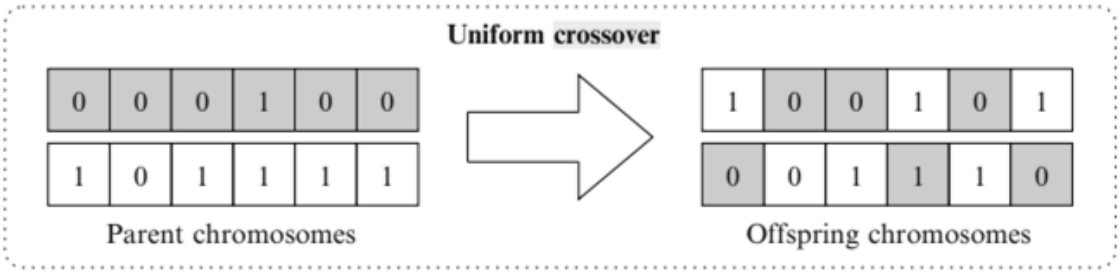
4. Mencione 3 operadores de Cruzamiento. Explique el funcionamiento de uno de ellos.

- Operador de cruzamiento de dos puntos de corte, operador de cruzamiento uniforme y operador de cruzamiento aritmético.

- **Operador de cruzamiento uniforme**

Un cruzamiento uniforme o también conocido como cruzamiento discreto intercambia información entre los padres de una manera diferente. Se suponen dos individuos, por ejemplo, A y B son seleccionados para realizar el cruzamiento uniforme. Cada gen tiene la probabilidad de 0.5 de heredar el gen de A, caso contrario se hereda el gen de B. Si se quiere generar dos descendientes a partir de dos padres, entonces cada gen del segundo descendiente puede ser seleccionado inversamente al correspondiente gen del primer descendiente. El cruzamiento uniforme puede utilizar múltiples padres para generar múltiples hijos.

Nota: El cruzamiento uniforme parametrizado consiste en tener dos padres y obtener un descendiente. Este cruzamiento maneja una probabilidad diferente de 0.5



5. Mencione 3 operadores de Selección. Explique el funcionamiento de uno de ellos.

- Operador de selección por muestreo aleatorio universal, operador de selección por torneo y operador de selección por emparejado variado inverso (NAM).

- **Operador de selección por torneo.**

Para implementar la selección de torneos, solo necesitamos elegir k individuos al azar con reemplazo y comparar los valores de aptitud de estos k individuos, que es el torneo. El mejor gana el torneo y se selecciona en el grupo de apareamiento. El k se denomina tamaño del torneo, que controla la presión selectiva. La selección de torneos más utilizada es la selección de torneos binaria, en la que k = 2.

6. Valore qué impacto tiene el tamaño de la población en la convergencia de un Algoritmo Genético.

- Existen mayores posibilidades de elección de individuos cuando el tamaño de la población crece, lo que provoca que el Algoritmo Genético mejore su desempeño. De igual manera, el tiempo de procesamiento aumenta.

7. Dados los problemas resueltos en la clase práctica 1, proponga las estructuras de datos necesarias para su implementación mediante un algoritmo genético estacionario.

- Listas y arreglos

8. Diseñe e implemente la interfaz de usuario para la solución de los problemas planteados.

- **Problema de la mochila**

```
from copy import copy
import random

tam_cromosoma = 10
tam_poblacion = 20
generacion_global = 1
k_torneo = 2  # tam de la poblacion del torneo

valor_objetos = [25, 21, 13, 19, 24, 22, 12, 21, 15, 13]
peso_objetos = [20, 16, 27, 16, 15, 30, 35, 15, 40, 48]
peso_maximo = 100
umbral = 0.1
```

```
prob_cruzamiento = 0.6
poblacion = []

class Gen:
    cromosoma = []
    aptitud = int
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.calcular_aptitud()
        self.generacion = generacion

    def calcular_aptitud(self):
        peso_aux = peso_maximo
        valmax = 0
        for j in range(tam_cromosoma):
            # Evaluando los objetos que estan dentro de la mochila
            if self.cromosoma[j] == 1:
                peso_aux = peso_aux - peso_objetos[j]
                valmax = valmax + valor_objetos[j]

        if peso_aux < 0:
            valmax = -1

        self.aptitud = int(valmax)

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < umbral:
                self.cromosoma[i] = random.randint(0, 1)

    def __str__(self):
        return "< Crom: " + str(self.cromosoma) + " Apt: " +
str(self.aptitud) + " Gen: " + str(self.generacion) + " >"

# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.randint(0, 1) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma

def seleccion_torneo(poblacion):
    torneo = []
```

```
    for i in range(k_torneo):
        seleccion = random.randint(0, len(poblacion) - 1)
        gen = poblacion[seleccion]
        torneo.append(gen)
        poblacion.pop(poblacion.index(gen))

    select = min(torneo, key=lambda x: x.aptitud)
    torneo.pop(torneo.index(select))

    for i in torneo:
        poblacion.append(i)

    return select

def uniform_crossover(gen1, gen2):
    descendienteA = []
    descendienteB = []

    for j in range(tam_cromosoma):
        if random.random() < 0.5:
            descendienteA.append(gen1.cromosoma[j])
            descendienteB.append(gen2.cromosoma[j])
        else:
            descendienteA.append(gen2.cromosoma[j])
            descendienteB.append(gen1.cromosoma[j])
    return descendienteA, descendienteB

def hamming_distance(gen1, gen2):
    result = 0
    if len(gen1.cromosoma) != len(gen2.cromosoma):
        print("Error: Los cromosomas no tienen el mismo tamaño")
    else:
        for x in range(tam_cromosoma):
            if gen1.cromosoma[x] != gen2.cromosoma[x]:
                result += 1
    return result

def crowding_replacement(poblacion, descendiente1, descendiente2, padre1,
padre2):

    dis_11 = hamming_distance(padre1, descendiente1)
    dis_12 = hamming_distance(padre1, descendiente2)

    dis_21 = hamming_distance(padre2, descendiente1)
    dis_22 = hamming_distance(padre2, descendiente2)
```

```
    if (dis_11 + dis_22) <= (dis_12 + dis_21):
        gan1 = max(padre1, descendiente1, key=lambda x: x.aptitud)
        poblacion.append(gan1)
        gan2 = max(padre2, descendiente2, key=lambda x: x.aptitud)
        poblacion.append(gan2)
    else:
        gan1 = max(padre1, descendiente2, key=lambda x: x.aptitud)
        poblacion.append(gan1)
        gan2 = max(padre2, descendiente1, key=lambda x: x.aptitud)
        poblacion.append(gan2)

#*************************************************************************

# Generacion Inicial aleatoria
for _ in range(tam_poblacion):
    poblacion.append(Gen(generacion_global, generar_cromosoma()))

print(max(poblacion, key=lambda x: x.aptitud))

# Iteracion para 500 generaciones
while generacion_global < 500:
    # Generando nueva poblacion
    generacion_global += 1
    gen1 = seleccion_torneo(poblacion)
    gen2 = seleccion_torneo(poblacion)
    # Probabilidad de Cruzamiento.
    if random.random() < prob_cruzamiento:
        # Cruzamiento de los padres
        crom_des1, crom_des2 = uniform_crossover(gen1, gen2)
        des1 = Gen(generacion_global, crom_des1)
        des2 = Gen(generacion_global, crom_des2)
        # Mutacion del descendiente
        des1.mutacion()
        des2.mutacion()
        des1.calcular_aptitud()
        des2.calcular_aptitud()
        crowding_replacement(poblacion, des1, des2, gen1, gen2)

    else:
        poblacion.append(gen1)
        poblacion.append(gen2)
    print(max(poblacion, key=lambda x: x.aptitud))
    #print(tam_poblacion == len(poblacion))
```

- **Problema del agente viajero**

```
import random
import numpy as np

tam_cromosoma = 10
tam_poblacion = 30
generacion_global = 1
k_torneo = 2  # tam de la poblacion del torneo

ciudades = np.zeros((tam_poblacion+1, tam_poblacion+1), int)
for i in range(tam_poblacion+1):
    for j in range(tam_poblacion+1):
        if i == j:
            continue
        ciudades[i, j] = random.randint(1, 100)

umbral = 0.1
prob_cruzamiento = 0.6
poblacion = []

class Gen:
    cromosoma = []
    aptitud = int
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.calcular_aptitud()
        self.generacion = generacion

    def calcular_aptitud(self):
        costo = 0
        origen = np.where(self.cromosoma == 1)
        inicio = origen
        j = 2
        n = tam_cromosoma
        while j < n:
            destino = np.where(self.cromosoma == j)
            costo += ciudades[origen[0], destino[0]]
            origen = destino
            j += 1
        costo += ciudades[origen[0], inicio[0]]
        self.aptitud = costo[0]
```

```
    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < umbral:
                indice = random.randint(0, tam_cromosoma-1)
                aux = self.cromosoma[indice]
                self.cromosoma[indice] = self.cromosoma[i]
                self.cromosoma[i] = aux

    def __str__(self):
        return "< Crom: " + str(self.cromosoma) + " Apt: " +
str(self.aptitud) + " Gen: " + str(self.generacion) + " >"

# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = np.zeros((tam_cromosoma), int)
    for i in range(tam_cromosoma):
        cromosoma[i] = i+1
    np.random.shuffle(cromosoma)
    return cromosoma

def seleccion_torneo(poblacion):
    torneo = []

    for i in range(k_torneo):
        seleccion = random.randint(0, len(poblacion) - 1)
        gen = poblacion[seleccion]
        torneo.append(gen)
        poblacion.pop(poblacion.index(gen))

    select = min(torneo, key=lambda x: x.aptitud)
    torneo.pop(torneo.index(select))

    for i in torneo:
        poblacion.append(i)

    return select

def uniform_order_based_crossover(cromosomaA, cromosomaB):
    # Inicializacion Plantilla binaria aleatoria
    template_binario = np.zeros((tam_cromosoma), int)

    # Rellenamos Plantilla binaria aleatoria
    for i in range(tam_cromosoma):
        if random.random() < 0.5:
            template_binario[i] = 1
```

```
    # Inicializacion de cromosomas Hijos
    descendienteA = np.zeros((tam_cromosoma), int)
    descendienteB = np.zeros((tam_cromosoma), int)

    # Rellenamos Hijos con Plantilla binaria aleatoria
    for i in range(tam_cromosoma):
        if template_binario[i] == 1:
            descendienteA[i] = cromosomaA[i]
            descendienteB[i] = cromosomaB[i]

    buffer_hijo_A = []
    buffer_hijo_B = []

    # creando Buffer de genes restantes del Hijo A
    for i in range(tam_cromosoma):
        if not (cromosomaB[i] in descendienteA):
            buffer_hijo_A.append(cromosomaB[i])

    # creando Buffer de genes restantes del Hijo B
    for i in range(tam_cromosoma):
        if not (cromosomaA[i] in descendienteB):
            buffer_hijo_B.append(cromosomaA[i])

    # Rellenar los espacios restantes del Hijo A con los valores en orden
del Padre B
    for i in range(tam_cromosoma):
        if template_binario[i] == 0:
            descendienteA[i] = buffer_hijo_A.pop(0)
            descendienteB[i] = buffer_hijo_B.pop(0)

    return descendienteA, descendienteB

def hamming_distance(gen1, gen2):
    result = 0
    if len(gen1.cromosoma) != len(gen2.cromosoma):
        print("Error: Los cromosomas no tienen el mismo tamaño")
    else:
        for x in range(tam_cromosoma):
            if gen1.cromosoma[x] != gen2.cromosoma[x]:
                result += 1
    return result

def crowding_replacement(poblacion, descendiente1, descendiente2, padre1,
padre2):
```

```
    dis_11 = hamming_distance(padre1, descendiente1)
    dis_12 = hamming_distance(padre1, descendiente2)

    dis_21 = hamming_distance(padre2, descendiente1)
    dis_22 = hamming_distance(padre2, descendiente2)

    if (dis_11 + dis_22) <= (dis_12 + dis_21):
        gan1 = min(padre1, descendiente1, key=lambda x: x.aptitud)
        poblacion.append(gan1)
        gan2 = min(padre2, descendiente2, key=lambda x: x.aptitud)
        poblacion.append(gan2)
    else:
        gan1 = min(padre1, descendiente2, key=lambda x: x.aptitud)
        poblacion.append(gan1)
        gan2 = min(padre2, descendiente1, key=lambda x: x.aptitud)
        poblacion.append(gan2)

#*****************************************************************************

# Generacion Inicial aleatoria
for _ in range(tam_poblacion):
    poblacion.append(Gen(generacion_global, generar_cromosoma()))

print(min(poblacion, key=lambda x: x.aptitud))

# Iteracion para 500 generaciones
while generacion_global < 500:
    # Generando nueva poblacion
    generacion_global += 1
    gen1 = seleccion_torneo(poblacion)
    gen2 = seleccion_torneo(poblacion)
    # Probabilidad de Cruzamiento.
    if random.random() < prob_cruzamiento:
        # Cruzamiento de los padres
        crom_des1, crom_des2 = uniform_order_based_crossover(
            gen1.cromosoma, gen2.cromosoma)
        descendiente1 = Gen(generacion_global, crom_des1)
        descendiente2 = Gen(generacion_global, crom_des2)
        # Mutacion del descendiente
        descendiente1.mutacion()
        descendiente2.mutacion()

        descendiente1.calcular_aptitud()
        descendiente2.calcular_aptitud()
```

```
        crowding_replacement(poblacion, descendiente1, descendiente2, gen1,
gen2)
    else:
        poblacion.append(gen1)
        poblacion.append(gen2)

    print(min(poblacion, key=lambda x: x.aptitud))
    #print(tam_poblacion == len(poblacion))
```

- $f(x) = \sum_{i=1}^{D} x_i^2 \ , \ con -10 \leq x_i \leq 10$

```
from copy import copy
import random

tam_cromosoma = 10
tam_poblacion = 20
generacion_global = 1
k_torneo = 2  # tam de la poblacion del torneo

umbral = 0.1
prob_cruzamiento = 0.6
poblacion = []

class Gen:
    cromosoma = []
    aptitud = int
    generacion = int

    def __init__(self, generacion, cromosoma):
        self.cromosoma = cromosoma
        self.calcular_aptitud()
        self.generacion = generacion

    def calcular_aptitud(self):
        valmax = 0
        for i in self.cromosoma:
            valmax += i**2
        self.aptitud = int(valmax)

    def mutacion(self):
        for i in range(tam_cromosoma):
            if random.random() < umbral:
                self.cromosoma[i] = random.uniform(-10, 10)
```

```
    def __str__(self):
        cromosoma_str = [round(x, 2) for x in self.cromosoma]
        return "< Crom: " + str(cromosoma_str) + " Apt: " +
str(self.aptitud) + " Gen: " + str(self.generacion) + " >"

# Con Valores de Representaciones de Orden
def generar_cromosoma():
    cromosoma = [random.uniform(-10, 10) for _ in range(tam_cromosoma)]
    random.shuffle(cromosoma)
    return cromosoma

def seleccion_torneo(poblacion):
    torneo = []
    for i in range(k_torneo):
        seleccion = random.randint(0, tam_poblacion - 1)
        gen = poblacion[seleccion]
        torneo.append(gen)
    return min(torneo, key=lambda x: x.aptitud)

def uniform_crossover(gen1, gen2):
    descendienteA = []

    for j in range(tam_cromosoma):
        if random.random() < 0.5:
            descendienteA.append(gen1.cromosoma[j])
        else:
            descendienteA.append(gen2.cromosoma[j])
    return descendienteA

def elitism_replacement(poblacion, descendiente):
    # Seleccion de Peor (Minimizar)
    reemplazo = max(poblacion, key=lambda x: x.aptitud)

    if descendiente.aptitud < reemplazo.aptitud:
        poblacion.pop(poblacion.index(reemplazo))
        poblacion.append(descendiente)

#*****************************************************************************

# Generacion Inicial aleatoria
for _ in range(tam_poblacion):
    poblacion.append(Gen(generacion_global, generar_cromosoma()))

print(min(poblacion, key=lambda x: x.aptitud))
```

```
# Iteracion para 500 generaciones
while generacion_global < 500:
    # Generando nueva poblacion
    generacion_global += 1
    gen1 = seleccion_torneo(poblacion)
    gen2 = seleccion_torneo(poblacion)
    # Probabilidad de Cruzamiento.
    if random.random() < prob_cruzamiento:
        # Cruzamiento de los padres
        cromosoma_descendiente = uniform_crossover(gen1, gen2)
        descendiente = Gen(generacion_global, cromosoma_descendiente)
        # Mutacion del descendiente
        descendiente.mutacion()
        descendiente.calcular_aptitud()
        # Remplazo Elitita
        elitism_replacement(poblacion, descendiente)
    else:
        poblacion.append(gen1)
        poblacion.append(gen2)

    print(min(poblacion, key=lambda x: x.aptitud))
```

3. Conclusiones

Los algoritmos genéticos presentan importantes ventajas sobre los algoritmos tradicionales, como son: su capacidad para trabajar con varios puntos simultáneamente, abarcando así un espacio mayor de búsqueda (paralelismo), además no requieren manipular directamente los parámetros del problema y poseen una gran facilidad para adaptarse a diferentes tipos de problemas.

A su vez, los algoritmos genéticos también presentan algunas desventajas como son: la dificultad para encontrar una representación apropiada de los parámetros del problema o para hallar una función objetivo adecuada.

Los AG algoritmos no aseguran encontrar la solución óptima, pero en muchos casos pueden proporcionar soluciones bastante adecuadas y que por otros métodos hubiera sido imposible encontrar. Dicho lo anterior, estos algoritmos representan, por todas sus características, una opción que debe ser tenida en cuenta cuando se busca resolver un problema con algún grado de complejidad.

4. Referencias

[1] Yu & Gen. Introduction to Evolutionary Algorithms. 2010. Capítulos 1 al 3.

[2] Burke & Kendall. Search Metodologies. 2005 Capítulo 4.

[3] Russell & Norving. Artificial Intelligence - A Modern Approach – 1995. Capítulo 4.