Nombre De La Práctica	**ALGORITMOS GENÉTICOS**			**No.**	**6**
Asignatura:	**Programación lógica y funcional**	**Carrera:**	**Ingeniería En Sistemas Computacionales**	**Duración De La Práctica (Hrs)**	**2 hr**
Nombre	**Alexander Martínez Cisneros**	**Grupo**	**3701**		

I. Competencia(s) específica(s):

II. Lugar de realización de la práctica (laboratorio, taller, aula u otro):

- Aula
- Casa

III. Material empleado:

- Laptop
- Language Python
- Editor de Código Como Visual Studio Code
- Libreria de Flask

IV. Desarrollo de la práctica:

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

ALGORITMOS GENÉTICOS

Los **algoritmos genéticos** o **algoritmos evolutivos** se inspiran en las teorías evolutivas de Darwin sobre la selección natural. En grandes poblaciones de seres vivos, los individuos tienden a entrar en competencia por los recursos disponibles como alimento, agua o refugio. Los mejor adaptados y los más fuertes tenderán a vivir más tiempo y a dejar una mayor descendencia mientras, que aquellos menos adaptados tendrán una descendencia menor. Con el tiempo, los genes de aquellos individuos mejor adaptados serán más abundantes y, por lo tanto, es de esperar una mejora global de la especie. Cuando dos individuos tienen descendencia, los genes de ambos se combinan logrando dar a lugar a un individuo con características genéticas superiores a los de sus procreadores. Evidentemente, esto no siempre será así, ya que se podría encontrar descendientes menos adaptados; sin embargo, es de esperar que, a mejor calidad genética de los procreadores, mayor será la probabilidad de que las características genéticas del descendiente sean buenas.

Por lo tanto, el mecanismo permite la mejora de la especie gracias al intercambio de genes entre individuos; sin embargo, no es lo único. De vez en cuando surgen mutaciones accidentales en los genes. Estas mutaciones suelen ser negativas en la mayoría de los casos, pero de vez en cuando, se da una mutación que mejora las características genéticas del individuo. Por lo tanto, se tiene dos mecanismos básicos de mejora de la especie: el entrecruzamiento de genes y la mutación de genes.

Los algoritmos genéticos, están inspirados en los mecanismos de evolución natural, que parten de una gran población de individuos (soluciones) y va generando descendencia basándose en las características de los mejores individuos, siempre según la función de avaluación. Las nuevas soluciones van distribuyéndose, según las antiguas de forma que, con el tiempo, la población tienda a converger hacia la solución global del problema.

Al igual que ocurre con los algoritmos de búsqueda local examinados anteriormente, garantiza que se acaba encontrando la solución óptima, aunque si el algoritmo está bien diseñado y parametrizado, se puede tener cierto grado de confianza en la que la solución alcanzada será, al menos, suficientemente buena.

Es necesario definir algunos conceptos antes de continuar.

- **Individuo:** Es una solución válida para el problema, independientemente de lo buena que sea. Debe cumplir con las restricciones impuestas por el problema.
- **Población:** Conjunto de individuos(soluciones).
- **Función de Adaptación:** Indica con que grado se adapta el individuo a la solución. Es sinónimo de la función de evaluación que se ha venido usando hasta ahora.
- **Genes:** Conjunto de parámetros o características que describen al individuo (solución). Por ejemplo, en el problema TSP, una ruta valida se correspondería con un gen. En el caso del problema SAT, el conjunto de valores asignados a las variables serían un gen. Por ejemplo (1, 0, 0, 1, 1).
- **Cromosoma:** Cada uno de los parámetros o características que conforman un gen. Por ejemplo, si se tiene un gen que representa una ruta del problema TSP, cualquiera de las ciudades que conforman esa ruta sería un cromosoma.
- **Convergencia de la población:** No se trata de conseguir que una de las soluciones sea muy buena, sino que la población, en general, sea buena. Dicho de otro modo, si la población en general converge hacia

características similares, cuya función de adaptación sea buena, esto servirá de piedra de toque para poder afirmar que se converge a una solución óptima. Se dice que una población ha convergido si hay un gran porcentaje de individuos cuya función de adaptación es igual o muy similar. A ese porcentaje se le llamará **factor de convergencia** y se denotará como la letra griega γ. Un valor habitual que permite afirmar que la población ha convergido suele ser γ = 95%. Es decir, que el 95% de población tiene genes similares.

En general, un algoritmo genético está compuesto por cuatro fases bien diferenciadas.

- **Selección:** Se eligen las mejores soluciones(individuos) de toda la población. Dichas soluciones tendrán más probabilidad de dejar descendencia.
- **Cruce:** En esta fase se combinan los genes de los mejores individuos, seleccionados en la fase anterior.
- **Mutación:** Con cierta probabilidad, se mutará un cromosoma de algunos de los individuos. Hay que hacer notar que la probabilidad de que se dé una mutación, ha de mantenerse lo suficientemente baja, para que no se degrade el potencial de mejora del cruce genético.
- **Eliminación:** Si no se elimina individuos, la población crecerá indefinidamente hasta hacerse inmanejable. Es por ello que hay que eliminar a aquellos individuos (soluciones) peor adaptados.

En general, el pseudocódigo para un algoritmo genético es el siguiente:

Función algoritmo_genético( ):

Para tamaño(población) / 2

Seleccionar dos individuos

Cruzar los dos individuos

Mutar con cierta probabilidad los nuevos individuos

Añadir los los dos nuevos individuos a la población

Eliminar los dos peores individuos de la población

Si la población converge:

Fin = verdadero

Salir con la solución

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

Alternativamente, puede usarse un límite en el número de iteraciones en vez de una condición de convergencia. O pueden usarse ambos criterios a la vez: el que se cumpla primero.

Se han descrito las cuatro fases del algoritmo genético. Ahora se profundizará un poco más en cada una de ellas.

La fase de selección, es la que permite escoger aquellas soluciones más cercanas a la solución óptima. El criterio que se sigue para puntuar las soluciones es la función de adaptación. Básicamente, lo que se hace es dar una puntuación a cada individuo, basándose en el valor devuelto por la función de adaptación y según sea dicho valor, el individuo tendrá más o menos probabilidades de ser seleccionado.

Una estrategia alternativa, podría ser directamente la de tomar a los dos mejores individuos para cruzarlos, pero esta estrategia ha demostrado ser poco eficiente, ya que la falta de diversidad genética puede hacer que la evolución sea lenta e incluso que se detenga.

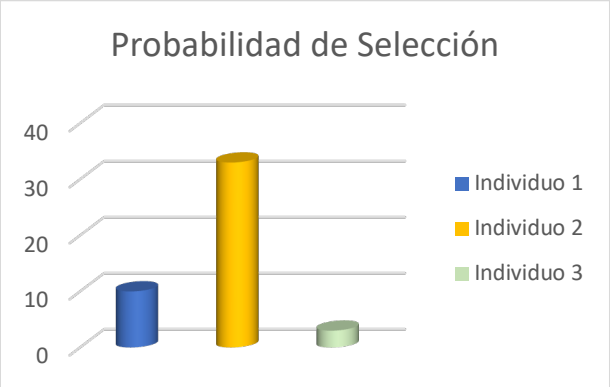
Hay varias técnicas que permiten seleccionar un solo individuo de un conjunto con una probabilidad dada. En este caso se usará una técnica sencilla y muy intuitiva. Por ejemplo, Supóngase que se tiene los siguientes individuos con sus respectivas puntuaciones:

Individuo	Puntuación
Individuo 1	10
Individuo 2	33
Individuo 3	5

Evidentemente, el individuo 2 tendra mayor probabilidad de se seleccionado que el individuo 1, y a su vez, este tendrá

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

mayor probabilidad al elgir un individuo respecto a otro, puede verse de forma más clara en el siguinete gráfico.



Para hacer la selección según la probabilidad de cada individuo, se calcula la suma total de las puntuaciones según su función de adaptación. En este caso, la suma total de las puntuaciones es 48. Supóngase que se obtiene el 25. Empezando por el primer individuo, se irá sumando todas las puntuaciones hasta que se iguale o sobrepase el valor 25. En este caso, primero se suma 10, que corresponde a la puntuación del segundo individuo. Como $10 < 25$ se sigue con el siguiente. Se suma 33, correspondiente a la puntuación del segundo individuo. Como $10 + 33 > 25$, entonces ya se puede seleccionar al individuo 2.

En cada vuelta del algoritmo se selecciona dos individuos para el cruce. Hay que hacer notar que si se selecciona al primer individuo, este puede seguir siendo seleccionado de nuevo como segundo individuo; es decir, se puede dar el caso de que se seleccione un gen para cruzarse consigo mismo.

Una vez seleccionados dos individuos se proceden al cruce genético. Sin embargo, aún no se ha hablado de cómo se puede representar o codificar la información dentro de los genes para poder hacer el cruce. Es evidente que la representación de cada individuo esta múy relacionada con el tipo de problema a solucionar. Se toma como ejemplo el problema SAT. Si se tiene n variables booleanas que deben satisfacer una función, parece lógico, una buena representación puede ser un vector de ceros o unos. Por ejemplo, si se quisiera satisfacer una función de 5 variables booleanas, un gen que podría representar una solución es el siguiente.

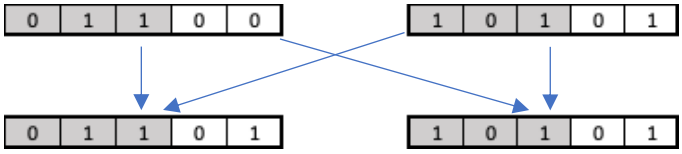
0	1	1	0	0

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

Supóngase que en la fase de selección se ha tomado el gen anterior y el siguiente.

1	0	1	0	1

El proceso de cruce consiste en elegir un punto de corte al azar. Por ejemplo, supóngase que aleatoriamente se elige el tercer cromosoma como punto de corte. Los dos nuevos genes resultantes del cruce serían los siguientes:



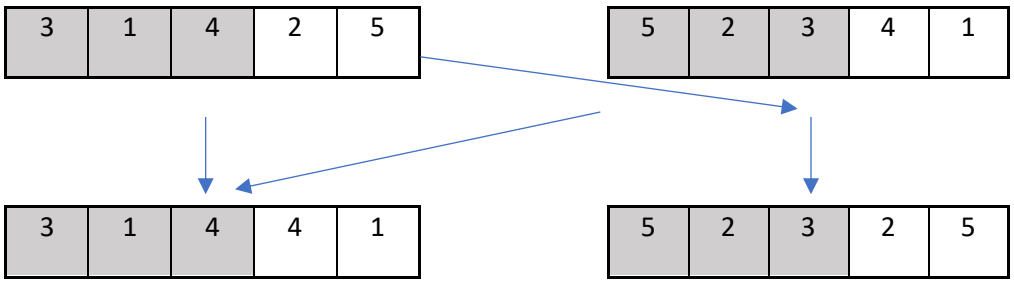
Es decir, el primer gen hijo tomaría sus primeros cromosomas del primer gen padre y el resto del segundo gen padre. En el caso del segundo gen hijo, sus primeros cromosomas los tomaría heredados del segundo gen padre y los restantes del primer gen padre.

Esta representación en forma de vector de ceros y unos es muy usada en los algoritmos genéticos. Cualquier problema basado en sí, están presentes o no ciertas características de los individuos puede representarse de esta forma. También se puede afrontar problemas de optimización de funciones simplemente representando las variables con su valor binario en vez de una forma decimal. Esta es, por lo tanto, una codificación muy interesante, siempre que se pueda representar la información del problema de esta manera.

En cualquier caso, existen muchas más formas de codificar la información. Tantas como se puedan imaginar. Nada impide usar vectores con valores enteros o de valores alfanuméricos. Póngase el problema del TSP. Una posible representación de una ruta de cinco ciudades puede ser la siguiente.

3	1	4	2	5

Donde cada valor entero le hace corresponder una ciudad. Es decir, que este gen representa la ruta 3 – 1 – 4 – 2 – 5. Sin embargo, hay que tener muy en cuenta las restricciones propias del problema. Por ejemplo. Supóngase el siguiente cruce de genes correspondientes al problema del TSP.



Los genes resultantes no son rutas validas, ya que se repiten algunas ciudades y otras ni siquiera son visitadas. Es decir, no cumplen las restricciones del problema.

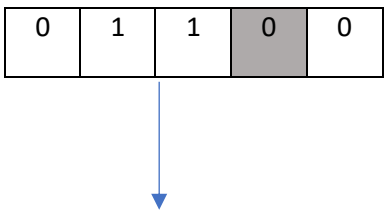
Diferentes operadores de cruce han sido descritos por diversos autores para este caso en concreto. Uno de los más sencillos es el cruce basado en la alternancia de las posiciones. Este operador va tomando alternativamente cromosomas de ambos genes y colocándolos según su orden en el padre para crear el gen hijo. Se omiten los cromosomas que ya se encuentran en el gen hijo. Con los genes de arriba se obtendría el hijo.

3	5	1	2	4

Y cambiando el orden de los padres se obtendría el segundo hijo.

5	3	2	1	4

El cruce genético es un poderoso operador; sin embargo, puede ocurrir que la configuración de la población inicial o haga muy difícil que se llegue a explorar ciertos espacios de estados. Es por ello que se introduce el operador de mutación. Este operador introduce una mutación en un cromosoma cierta probabilidad.

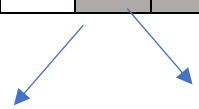


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

0	1	1	1	0

De nuevo es importante tener en cuenta las restricciones del problema. En el caso del TSP, por ejemplo, un posible operador de mutación podría ser aquel que intercambia dos ciudades consecutivas. Aunque hay otras posibilidades.

4	1	3	5	2



4	1	5	3	2

En cualquier caso, no conviene darle mucho peso a este operador. Debe actuar con una baja probabilidad para no quitar protagonismo al operador de cruce, que es el que debe conducir la mejora y la evolución de la población.

Finalmente, la última fase del algoritmo, es la eliminación de individuos, que tiene dos finalidades: por un lado, evita que la población crezca y hace que se mantenga estable; por otro lado, elimina a los individuos peor adaptados evitando que sus cromosomas se perpetúen. Hay que tener en cuenta que los genes recién creados también entran en competencia a la hora de elegir que individuos son eliminados; es decir, si se genera un hijo de muy baja calidad puede ser seleccionado para ser eliminado en la misma iteración.

El siguiente algoritmo resuelve el problema SAT(en concreto 3-SAT) usando un algoritmo genético. Por simplicidad no se usa un criterio de convergencia para detener el algoritmo, sino que se indica un límite en el número de iteraciones en la variable max_iter. Como función de adaptación, en vez de evaluar una función concreta según los valores de las variables, se genera una solución aleatoria al principio del programa y la función adaptación_sat( ) se encarga de comprobar el grado de adaptación de un gen a la solución generada aleatoriamente. Concretamente, la función de adaptación hace recuento del número de cláusulas (grupos de 3 variables) que toman el valor verdadero.

```
import math
import random


def poblacion_inicial(max_poblacion, num_vars):
    # Crear población Inicial Aleatoria
    poblacion = []
    for i in range(max_poblacion):
        gen = []
        for j in range(num_vars):
            if random.random() > 0.5:
                gen.append(1)
            else:
                gen.append(0)
        poblacion.append(gen[:])
    return poblacion


def adaptacion_3sat(gen, solucion):
    # Contar las claúsulas correctas
    n = 3
    cont = 0
    clausula_ok = True
    for i in range(len(gen)):
        n = n - 1
        if (gen[i] != solucion[i]):
            clausula_ok = False
            if n == 0:
                if clausula_ok:
```


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

```
                    cont = cont + 1

                n = 3

                clausula_ok = True

        if n > 0:

            if clausula_ok:

                cont = cont + 1

        return cont


def evalua_poblacion(poblacion, solucion):

    # Evalua todos los genes de la población.

    adaptacion = []

    for i in range(len(poblacion)):

        adaptacion.append(adaptacion_3sat(poblacion[i], solucion))

    return adaptacion


def seleccion(poblacion, solucion):

    adaptacion = evalua_poblacion(poblacion, solucion)

    # Suma de todas las puntuaciones

    total = 0

    for i in range(len(adaptacion)):

        total = total + adaptacion[i]

    # Tomar dos elementos

    val1 = random.randint(0,total)

    val2 = random.randint(0,total)

    sum_sel = 0

    #gen1 = 0

    #gen2 = 0
```


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

```
    for i in range (len(adaptacion)):
        sum_sel = sum_sel + adaptacion[i]
        if sum_sel >= val1:
            gen1 = poblacion[i]
            break
    sum_sel = 0
    for i in range(len(adaptacion)):
        sum_sel = sum_sel + adaptacion[i]
        if sum_sel >= val2:
            gen2 = poblacion[i]
            break
    return gen1, gen2


def cruce(gen1, gen2):
    # Cruza 2 genes y obtiene 2 descendientes
    nuevo_gen1 = []
    nuevo_gen2 = []
    corte = random.randint(0,len(gen1))
    nuevo_gen1[0:corte] = gen1[0:corte]
    nuevo_gen1[corte:] = gen2[corte:]
    nuevo_gen2[0:corte] = gen2[0:corte]
    nuevo_gen2[corte:] = gen1[corte:]
    return nuevo_gen1, nuevo_gen2


def mutacion(prob, gen):
    # Muta Gen con una probabilidad prob.
    if random.random() < prob:
```


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

```
        cromosoma = random.randint(0, len(gen))
        if gen[cromosoma] == 0:
            gen[cromosoma]=1
        else:
            gen[cromosoma]=0
    return gen


def elimina_peores_genes(poblacion, solucion):
    # Elimina los dos peores genes
    adaptacion = evalua_poblacion(poblacion, solucion)
    i = adaptacion.index(min(adaptacion))
    del poblacion[i]
    del adaptacion[i]
    i = adaptacion.index(min(adaptacion))
    del poblacion[i]
    del adaptacion[i]


def mejor_gen(poblacion, solucion):
    #Devuelve el mejor gen de la población
    adaptacion = evalua_poblacion(poblacion, solucion)
    return poblacion[adaptacion.index(max(adaptacion))]


def algoritmo_genetico():
    max_iter = 10
    max_poblacion = 50
    num_vars = 10
    fin = False
```


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

```
    solucion = poblacion_inicial(1, num_vars)[0]
    poblacion = poblacion_inicial(max_poblacion, num_vars)


    iteraciones = 0
    while not fin:
        iteraciones = iteraciones + 1
        for i in range((len(poblacion))//2):
            gen1, gen2 = seleccion(poblacion, solucion)
            nuevo_gen1, nuevo_gen2 = cruce(gen1, gen2)
            nuevo_gen1 = mutacion(0.1, nuevo_gen1)
            nuevo_gen2 = mutacion(0.1, nuevo_gen2)
            poblacion.append(nuevo_gen1)
            poblacion.append(nuevo_gen2)
            elimina_peores_genes(poblacion, solucion)


        if (max_iter < iteraciones):
            fin = True
    print("Solución: " + str(solucion))


    mejor = mejor_gen(poblacion, solucion)
    return mejor, adaptacion_3sat(mejor, solucion)


if __name__ == "__main__":
    random.seed()
    mejor_gen = algoritmo_genetico()
    print("Mejor gen Encontrado: " + str(mejor_gen[0]))
    print("Función de adaptación: " + str(mejor_gen[1]))
```


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

Cuando se busca una solución a un problema usando un algoritmo genético, en conveniente realizar varias ejecuciones, ya que como pasa en el algoritmo de templado simulado, se está frente a un algoritmo **no determinista** (no siempre se comporta igual) y alguna de las ejecuciones podría no obtener un buen resultado. Por otro lado, es importante ajustar bien los parámetros del algoritmo a cada problema, en concreto el número de iteraciones y población máxima. En el ejemplo, vendría mejor un ajuste u otro.

Ejemplo:

Iteraciones = 10; Max_población = 50, Num_Vars = 10

Iteraciones = 30; Max_población = 150; Num_vars =100

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

Codificación:

1. Importamos las librerías necesarias, como math para operaciones matemáticas y random para generar números aleatorios y tomar decisiones aleatorias en el código.

```
import math
import random
```

2. Creamos la función "población_inicial" para generar una población inicial de genes aleatorios para el algoritmo genético. Cada gen es una lista de valores binarios (0 o 1), y el tamaño de la población está determinado por "max_poblacion" y "num_vars" representa la longitud de cada gen.

```
def poblacion_inicial(max_poblacion, num_vars):
    # Crear población Inicial Aleatoria
    poblacion = []
    for i in range(max_poblacion):
        gen = []
        for j in range(num_vars):
            if random.random() > 0.5:
                gen.append(1)
            else:
                gen.append(0)
        poblacion.append(gen[:])
    return poblacion
```

3. Definimos una función que evalúa la adecuación de una solución gen en el contexto de un problema 3SAT, que involucra cláusulas con tres variables booleanas.

```
def adaptacion_3sat(gen, solucion):
    # Contar las claúsulas correctas
    n = 3
    cont = 0
    clausula_ok = True
    for i in range(len(gen)):
        n = n - 1
        if (gen[i] != solucion[i]):
            clausula_ok = False
            if n == 0:
                if clausula_ok:
                    cont = cont + 1
                n = 3
                clausula_ok = True
        if n > 0:
            if clausula_ok:
                cont = cont + 1
        return cont
```

4. Creamos una función para evaluar los genes de la población, calculando la adaptación de cada

gen en relación con una solución objetivo. Se utiliza un bucle for para iterar sobre la población y se almacenan las adaptaciones en una lista llamada "adaptación".

```
def evalua_poblacion(poblacion, solucion):
    # Evalua todos los genes de la población.
    adaptacion = []
    for i in range(len(poblacion)):
        adaptacion.append(adaptacion_3sat(poblacion[i], solucion))
    return adaptacion
```

5. Implementamos una función para la selección de dos genes de la población con base en su adaptación. Esta función utiliza la evaluación de la población y genera dos números aleatorios para seleccionar los genes más aptos.

```
def seleccion(poblacion, solucion):
    adaptacion = evalua_poblacion(poblacion, solucion)
    # Suma de todas las puntuaciones
    total = 0
    for i in range(len(adaptacion)):
        total = total + adaptacion[i]
    # Tomar dos elementos
    val1 = random.randint(0,total)
    val2 = random.randint(0,total)
    sum_sel = 0
    #gen1 = 0
    #gen2 = 0
    for i in range (len(adaptacion)):
        sum_sel = sum_sel + adaptacion[i]
        if sum_sel >= val1:
            gen1 = poblacion[i]
            break
    sum_sel = 0
    for i in range(len(adaptacion)):
        sum_sel = sum_sel + adaptacion[i]
        if sum_sel >= val2:
            gen2 = poblacion[i]
            break
    return gen1, gen2
```

6. Creamos una función para realizar la operación de cruce entre dos genes, generando dos descendientes. Se elige aleatoriamente un punto de cruce y se intercambian las secciones de los genes para diversificar la población.

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

```python
def cruce(gen1, gen2):
    # Cruza 2 genes y obtiene 2 descendientes
    nuevo_gen1 = []
    nuevo_gen2 = []
    corte = random.randint(0, len(gen1))
    nuevo_gen1[:corte] = gen1[:corte]
    nuevo_gen1[corte:] = gen2[corte:]
    nuevo_gen2[:corte] = gen2[:corte]
    nuevo_gen2[corte:] = gen1[corte:]
    return nuevo_gen1, nuevo_gen2
```

7. Implementamos una función para la operación de mutación en un gen con una probabilidad dada. Se elige aleatoriamente un índice y se cambia el valor binario en ese índice si se cumple la probabilidad de mutación.

```python
def mutacion(prob, gen):
    # Muta Gen con una probabilidad prob.
    if random.random() < prob:
        cromosoma = random.randint(0, len(gen) - 1)
        if gen[cromosoma] == 0:
            gen[cromosoma] = 1
        else:
            gen[cromosoma] = 0
    return gen
```

8. Creamos una función para eliminar los dos genes menos aptos de la población, basándonos en su adaptación con respecto a una solución objetivo.

```python
def elimina_peores_genes(poblacion, solucion):
    # Elimina los dos peores genes
    adaptacion = evalua_poblacion(poblacion, solucion)
    i = adaptacion.index(min(adaptacion))
    del poblacion[i]
    del adaptacion[i]
    i = adaptacion.index(min(adaptacion))
    del poblacion[i]
    del adaptacion[i]

def mejor_gen(poblacion, solucion):
```

9. Definimos una función para obtener el gen con la mejor adaptación en relación con una solución objetivo. Se utiliza la evaluación de la población y se retorna el gen con la máxima adaptación.

```python
def mejor_gen(poblacion, solucion):
    #Devuelve el mejor gen de la población
    adaptacion = evalua_poblacion(poblacion, solucion)
    return poblacion[adaptacion.index(max(adaptacion))]
```

10. Implementamos la función principal que ejecuta el algoritmo genético. Utiliza bucles para la reproducción, cruce, mutación y eliminación de genes menos aptos. Se repite hasta alcanzar el número máximo de iteraciones.

```
def algoritmo_genetico():
    max_iter = 10
    max_poblacion = 50
    num_vars = 10
    fin = False
    solucion = poblacion_inicial(1, num_vars)[0]
    poblacion = poblacion_inicial(max_poblacion, num_vars)

    iteraciones = 0
    while not fin:
        iteraciones = iteraciones + 1
        for i in range((len(poblacion))//2):
            gen1, gen2 = seleccion(poblacion, solucion)
            nuevo_gen1, nuevo_gen2 = cruce(gen1, gen2)
            nuevo_gen1 = mutacion(0.1, nuevo_gen1)
            nuevo_gen2 = mutacion(0.1, nuevo_gen2)
            poblacion.append(nuevo_gen1)
            poblacion.append(nuevo_gen2)
            elimina_peores_genes(poblacion, solucion)

        if (max_iter < iteraciones):
            fin = True
    print("Solución: " + str(solucion))

    mejor = mejor_gen(poblacion, solucion)
    return mejor, adaptacion_3sat(mejor, solucion)
```

11. Definimos el método de ejecución del programa, generando genes aleatorios y llamando a la función del algoritmo genético.

```
if __name__ == "__main__":
    random.seed()
    mejor_gen = algoritmo_genetico()
    print("Mejor gen Encontrado: " + str(mejor_gen[0]))
    print("Función de adaptación: " + str(mejor_gen[1]))
```

12. Ahora ejecutamos el programa:

```
C:\Users\juan\Downloads\Programaci-n-Logica-Funcional-PLF-main>python algoritmos_geneticos.py
Solución: [1, 0, 0, 0, 0, 1, 1, 0, 0, 0]
Mejor gen Encontrado: [1, 1, 1, 0, 0, 1, 1, 1, 0, 1]
Función de adaptación: 1
```


*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

Codificación de la aplicación en flask

1. En el contexto de una aplicación web con Flask, importamos los módulos necesarios y definimos funciones para la creación de rutas y manejo de plantillas HTML.

```
from flask import Flask, render_template
import math
import random
```

2. Usamos una extensión de flask para inicializar lo que será nuestra aplicación Flask

```
app = Flask(__name__)
```

3. Definimos cada función que hicimos en el código anterior

4. Creamos rutas para la página principal y de resultados, cada una llamando a funciones específicas. Se definen plantillas HTML para la entrada de datos y la visualización de resultados.

```
@app.route('/')
def principal():
    return render_template("index.html")
```

5. Define una ruta para la página de resultados. Llama a la función resultado () que genera una nueva ruta y renderiza otra plantilla HTML, más aparte que es el que contiene la función del algoritmo genético.

*Cualquier documento no identificado como **Controlado** se considera **COPIA NO CONTROLADA** y no es auditable.*

```python
def algoritmo_genetico():
    max_iter = 10
    max_poblacion = 50
    num_vars = 10
    fin = False
    solucion = poblacion_inicial(1, num_vars)[0]
    poblacion = poblacion_inicial(max_poblacion, num_vars)

    iteraciones = 0
    while not fin:
        iteraciones = iteraciones + 1
        for i in range((len(poblacion))//2):
            gen1, gen2 = seleccion(poblacion, solucion)
            nuevo_gen1, nuevo_gen2 = cruce(gen1, gen2)
            nuevo_gen1 = mutacion(0.1, nuevo_gen1)
            nuevo_gen2 = mutacion(0.1, nuevo_gen2)
            poblacion.append(nuevo_gen1)
            poblacion.append(nuevo_gen2)
            elimina_peores_genes(poblacion, solucion)

        if (max_iter < iteraciones):
            fin = True
    print("Solución: " + str(solucion))

    mejor = mejor_gen(poblacion, solucion)
    return mejor, adaptacion_3sat(mejor, solucion)
```

6. Inicializamos el servidor web para mostrar y renderizar resultados. Se incluye la línea de código que generará una ruta inicial aleatoria y mostrará los resultados esperados.

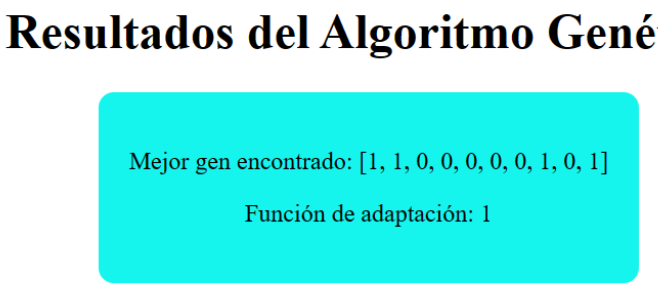
```python
print("Solución: " + str(solucion))

mejor = mejor_gen(poblacion, solucion)
return mejor, adaptacion_3sat(mejor, solucion)
```

7. Ejecutamos la aplicación localmente para verificar su funcionamiento y visualizar la página principal, presionando el botón para mostrar los resultados.

```
 * Serving Flask app 'flask_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 983-960-531
127.0.0.1 - - [21/Jan/2024 14:23:12] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jan/2024 14:23:13] "GET /static/Chiyo_Mihama_With_Python_Homework.png HTTP/1.1" 304 -
127.0.0.1 - - [21/Jan/2024 14:23:13] "GET /static/python.png HTTP/1.1" 304 -
127.0.0.1 - - [21/Jan/2024 14:23:13] "GET /favicon.ico HTTP/1.1" 404 -
```

8. Vemos que funciona nos renderiza la página principal, presionemos el botón para que nos muestre los resultados:



URL: http://genetica.pythonanywhere.com

V. Conclusión

En síntesis, los algoritmos genéticos se presentan como simuladores evolutivos que replican el proceso de selección natural para abordar problemas complejos. Inspirados en el principio de la supervivencia de los más aptos en la naturaleza, estos algoritmos emplean conceptos como selección, cruzamiento y mutación para iterativamente perfeccionar soluciones a lo largo del tiempo.

Cuando se aplican específicamente a desafíos como el SAT, los algoritmos genéticos ofrecen un enfoque innovador para la búsqueda de soluciones efectivas. Mediante la selección, combinación y ocasional modificación de características en posibles soluciones, estos algoritmos exploran de manera eficaz diversas opciones hasta dar con respuestas satisfactorias.

La relevancia de los algoritmos genéticos reside en su capacidad para afrontar problemas complicados y encontrar soluciones útiles, incluso en situaciones donde otros métodos podrían encontrar dificultades. Se erigen como herramientas versátiles aplicables en una variedad de contextos del mundo real, facilitando la toma de decisiones y la resolución de problemas prácticos.