TRABAJO PRÁCTICO N°3

INTELIGENCIA ARTIFICIAL – ALGORITMOS GENÉTICOS

“Solución al Acertijo de Einstein”

Integrantes del Grupo:

Di Fresco, Lucas – LU: 1068498
Guglielmone, Lucas – LU: 1110022
Lacuesta, Gastón Axel – LU: 1117695
Orozco, Patricio – LU: 1058382
Rondan, Ignacio – LU: 1068437
Vera, Samira – LU: 1101684

Profesor:

Ponzoni, Nelson

Cuatrimestre: 02 – Año: 2022

UNIVERSIDAD ARGENTINA DE LA EMPRESA
FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

INTELIGENCIA ARTIFICIAL – ALGORITMOS GENÉTICOS

Tabla de Contenidos:

Consigna

Un algoritmo es una serie de pasos organizados que describe el proceso que se debe seguir, para dar solución a un problema específico. En este caso particular, los algoritmos genéticos son llamados así porque se inspiran en la evolución biológica y su base genético-molecular.

Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (mutaciones y recombinaciones genéticas), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.

Los campos de aplicación generalmente se extienden a resolución de problemas con restricciones y exploración de soluciones, todo ello en un amplio, casi infinito, espacio de soluciones posibles.

Objetivo

- Entender y comprender el procesamiento mediante algoritmos genéticos.
- Desarrollar los conceptos de operadores en un caso de aplicación.
- Identificar los pros y cons de la utilización de este tipo de técnicas.

Plazo

Según lo indicado en el cronograma. Webcampus.

NOTA: Se extiende el plazo, una semana adicional a la estipulada originalmente.

Problema

Tenemos 5 casas de cinco colores diferentes y en cada una de ellas vive una persona de una profesión diferente. Cada uno programa en un lenguaje diferente, utiliza una base de datos NoSQL diferente y utiliza un editor de textos diferente.

Se tienen las siguientes pistas:

1. Hay 5 casas.
2. El Matemático vive en la casa roja.
3. El hacker programa en Python.
4. El Brackets es utilizado en la casa verde.
5. El analista usa Atóm.
6. La casa verde está a la derecha de la casa blanca.
7. La persona que usa Redis programa en Java
8. Cassandra es utilizado en la casa amarilla
9. Notepad++ es usado en la casa del medio.
10. El Desarrollador vive en la primera casa.
11. La persona que usa HBase vive al lado de la que programa en JavaScript.
12. La persona que usa Cassandra es vecina de la que programa en C#.
13. La persona que usa Neo4J usa Sublime Text.
14. El Ingeniero usa MongoDB.

15. EL desarrollador vive en la casa azul.

Responda: ¿Quién usa el editor Vim?

Tareas

El equipo debe realizar las siguientes actividades. En todos los casos, cada miembro del equipo debe poder justificar las respuestas.

1. Identificar y diseñar un individuo
2. Proponga un método para la generación de individuos. Poblar el universo de individuos iniciales.
3. A partir del código/estructura provista
    a. Implementar el operador Selección
    b. Implementar el operador Cruzamiento
    c. Implementar el operador Mutación
4. Implemente 3 condiciones de corte para el problema según las restricciones enumeradas en la sección “Problema”
5. Identifique alternativas de cómo generar individuos para agregar a la población de individuos

Nota: Cuenta con la estructura en “dev_ga_base.py”. Aunque puede proponer la suya. Es a modo guía o ejemplo

Solución

Identificar y Diseñar un Individuo

La primera tarea a realizar fue el diseño del fenotipo que se utilizó como individuo de una población a evaluar. Para ello se realizó un análisis de las características que debía cumplir.

Entre ellas se encontró que:

1. El fenotipo debe contener un arreglo de genes ordenados, llamado cromosoma, que sirva como solución al problema.
2. El fenotipo debe tener alguna forma de medir su efectividad respecto a la solución del problema.
3. El fenotipo debe admitir mutaciones aleatorias de sus genes.
4. El fenotipo debe poder inicializarse a partir de cruces de dos fenotipos
5. El fenotipo debe poder inicializarse de manera aleatoria

Finalmente, este fue el resultado:

Se definieron diccionarios de datos para cada categoría posible en donde se asocia un código binario con una característica. Además, por simplicidad, se define un arreglo de todos los posibles códigos binarios.

```
genes =      ['001', '010', '011', '100', '101' ]
colors =     {'001' : 'red',          '010' : 'blue',         '011' : 'green',    '100' : 'white',    '101' : 'yellow'}
profession = {'001' : 'Mathematician','010' : 'Hacker',       '011' : 'Engineer', '100' : 'Analyst',  '101' : 'Developer'}
lenguaje =   {'001' : 'Python',       '010' : 'C#',           '011' : 'Java',     '100' : 'C++',      '101' : 'JavaScript'}
database =   {'001' : 'Cassandra',    '010' : 'MongoDB',      '011' : 'HBase',    '100' : 'Neo4j',    '101' : 'Redis'}
editor =     {'001' : 'Brackets',     '010' : 'Sublime Text', '011' : 'Vim',      '100' : 'Atom',     '101' : 'Notepad++'}
```

La clase se inicializa con un arreglo de genes, el cromosoma, y un puntaje que guardara la efectividad del individuo.

```
# Se crea un individuo
def __init__(self):
    # Chromosoma que contiene un arreglo de genes
    self.chromosome = []

    # Puntaje que determina la eficiencia del fenotipo para resolver el problema
    self.score = 0
```

El primer método de la clase que se desarrollo fue el método “encode” que lo que hace es generar un cromosoma aleatorio para el fenotipo. Inicialmente se utilizó un método en el cual se asignaban 25 genes seguidos de manera aleatoria, pero después de algunas pruebas, se encontró que si se inicializaba el cromosoma sin permitir genes duplicados en las distintas categorías se lograban mejores resultados. Se debe tener en cuenta también que el orden de los genes está dado por el número de la

casa de menor a mayor y dentro de cada casa se ubica el Color, la Profesión, el Lenguaje, la Base de Datos y el Editor.

```
# Inicializar el chromosoma con genes aleatorios sin repetirlos en las distintas categprías
def encode(self):
    # Se genera una pila de genes para cada categoría para que no se repitan los genes de la mismas categorías en las distintas casas
    col = genes.copy()
    prof = genes.copy()
    lang = genes.copy()
    db = genes.copy()
    ed = genes.copy()

    # Se mezclan al azar los genes de cada categoría
    random.shuffle(col)
    random.shuffle(prof)
    random.shuffle(lang)
    random.shuffle(db)
    random.shuffle(ed)

    # Se asignan los genes al chromosoma en el orden Color-Profesión-Lenguaje-Base de Datos-Editor
    for i in range(5):
        self.chromosome.append(col[i])
        self.chromosome.append(prof[i])
        self.chromosome.append(lang[i])
        self.chromosome.append(db[i])
        self.chromosome.append(ed[i])

    self.fitness()
    pass
```

Por otro lado, se desarrolló una clase que decodificaba el cromosoma y lo transformaba en un arreglo de datos que facilitara la comprensión del mismo.

```
# Se traduce cada gen de 3 bits en los distintos nombres de cada categoría
# según el diccionario y se devuelve un arreglo con los datos de cada casa por separado
def decode(self):
    code = [[colors[self.chromosome[i*5 + 0]],
             profession[self.chromosome[i*5 + 1]],
             lenguaje[self.chromosome[i*5 + 2]],
             database[self.chromosome[i*5 + 3]],
             editor[self.chromosome[i*5 + 4]]] for i in range(5)]

    return code
```

El método “mutate” permite introducir una mutación aleatoria en un gen aleatorio del cromosoma. Tanto el gen a mutar seleccionado ser insertado, como el gen a reemplazar son seleccionados al azar. Podría darse el caso que se remplazara un gen por otro gen igual.

```
# Se introduce una mutación aleatoria de un unico gen del Fenotipo
def mutate(self):
    self.chromosome[random.randrange(0,24)] = genes[random.randrange(0, 4)]
    pass
```

Por otro lado, el fenotipo permite la inicialización del cromosoma a partir de dos fenotipos llamados madre y padre. De esta manera, se selecciona un gen al azar de los dos padres que divida sus

cromosomas en el mismo lugar y se utiliza la primera parte del cromosoma del padre junto con la segunda parte del cromosoma de la madre para confeccionar el nuevo cromosoma del fenotipo.

```
# Se cruza dos chromosomas de Fenotipos diferentes para generar un solo chromosoma y asignarselo a este Fenotipo
def crossover(self, parent_1, parent_2):
    self.chromosome.clear()
    indiceCorte = random.randrange(0, 23);
    for gen in parent_1.chromosome[0:indiceCorte] : self.chromosome.append(gen)
    for gen in parent_2.chromosome[indiceCorte:25] : self.chromosome.append(gen)
pass
```

Finalmente, el método “fitness” se encarga de evaluar la efectividad del individuo para resolver el problema. Esta operación se centra en evaluar si la solución cumple o no con las restricciones del caso. En la primera parte, se evalúa si existen genes repetidos por cada categoría en las distintas casas. En caso de que así fuera, se le resta un punto al contador (por cada ocurrencia) porque eso conformaría una combinación invalida.

```
# Se calcula la eficacia del chromosoma para resolver el problema dado basado en las restricciones del mismo
def fitness(self):
    self.score = 0

    # Se asginan valores de premio y castigo para el puntaje
    ok_score = 1
    fail_score = -1

    # Se decodifica el chromosoma para poder tratarlo de manera mas sencilla
    # (Este paso se podría evitar pero obligaría a trabajar al programador con genes de cadenas de bits)
    solution = self.decode()

    # Se castiga al Fenotipo en caso de que existan genes repetidos en las distintas categorías
    # (Combinaciones inválidas)
    for i in range(4):
        for j in range(i + 1, 5):
            if solution[i][0] == solution[j][0]: self.score += fail_score
            if solution[i][1] == solution[j][1]: self.score += fail_score
            if solution[i][2] == solution[j][2]: self.score += fail_score
            if solution[i][3] == solution[j][3]: self.score += fail_score
            if solution[i][4] == solution[j][4]: self.score += fail_score
```

En la segunda parte, el método evalúa si se cumplen con las condiciones del problema. Por ello, se le asigna un puntaje positivo por cada condición que se cumpla. Dado que existen 14 condiciones diferentes (la primera se cumple por diseño dado que cada cromosoma admite solo 5 casas) entonces el puntaje máximo que se puede obtener es de 14 puntos.

```
# Se premia al Fenotipo en caso de que se cumplan con las restricciones que se deben cumplir dentro de una misma casa
for house in solution:
    if (house[1] == 'Mathematician' and house[0] == 'red') : self.score += ok_score
    if (house[1] == 'Hacker' and house[2] == 'Python') : self.score += ok_score
    if (house[4] == 'Brackets' and house[0] == 'green') : self.score += ok_score
    if (house[1] == 'Analyst' and house[4] == 'Atom') : self.score += ok_score
    if (house[3] == 'Redis' and house[2] == 'Java') : self.score += ok_score
    if (house[3] == 'Cassandra' and house[0] == 'yellow') : self.score += ok_score
    if (house[3] == 'Neo4j' and house[4] == 'Sublime Text') : self.score += ok_score
    if (house[1] == 'Engineer' and house[3] == 'MongoDB') : self.score += ok_score
    if (house[1] == 'Developer' and house[0] == 'blue') : self.score += ok_score

if (solution[2][4] == 'Notepad++') : self.score += ok_score
if (solution[0][1] == 'Developer') : self.score += ok_score

# Se premia al Fenotipo en caso de que se cumplan con las restricciones que se deben cumplir en relación a otras casas
for i in range(len(solution) - 1):
    if (solution[i][0] == 'white' and solution[i + 1][0] == 'green') : self.score += ok_score

for i in range(1, len(solution) - 1):
    if (solution[i][3] == 'HBase') :
        if(solution[i - 1][2] == 'JavaScript' or solution[i + 1][2] == 'JavaScript'): self.score += ok_score
    if (solution[i][3] == 'Cassandra') :
        if(solution[i - 1][2] == 'C#' or solution[i + 1][2] == 'C#'): self.score += ok_score

# En total, se deben cumplir con 14 restricciones, por lo tanto, el puntaje mas alto posible es 14
pass
```

Generación y Evolución de una Población

Para generar y manejar una población, se implementó la clase “Riddle”. La clase “Riddle” cuenta con 2 métodos principales y 3 métodos de soporte y se encarga de ofrecer una solución al problema. Para ello, deberá generar una población, hacerla evolucionar a lo largo de un número finito de generaciones y evaluar si en la población existe algún individuo que cumpla con las condiciones del problema.

El método principal de esta clase es el método “evolve”. El mismo se encarga de generar una nueva población (conjunto de individuos) y hacerla evolucionar, es decir, seleccionar a los individuos más aptos, cruzarlos, mutarlos y determinar si se debe seguir evolucionando. A este ciclo lo llamamos generación y este método permite un número finito de generaciones para evitar bucles infinitos. Por la tanto, la primer condición de corte de la evolución es en caso de que se supere la cantidad máxima de generaciones establecida.

Dentro del método, se hace llamado a otros 3 métodos secundarios. El primero de ellos el método “generate” que se encarga de generar una nueva población con individuos inicializados con el método “encode”, es decir, al azar. El tamaño de la población lo determina el usuario.

Despues de la inicialización de la población y del resto de las variables necesarias, se inicia el bucle de la evolución y lo primero que se realiza es una selección de la población. Esta selección consiste en eliminar a una proporción de la población. Esta proporción está dada por variable “crosover_prop” y la determina el usuario. Se pueden dar 3 escenarios con esta variable, si la variable es menor a 0.5, entonces la población decrecerá en cantidad a medida que evoluciona. Por otro lado, si la variable es mayor que 0.5, la población crecerá en cantidad. Finalmente, si la variable es igual a 0.5, la población se mantendrá estable y no cambiara en cantidad. Esto se debe a que cuando se cruza a la población, se

generan dos individuos nuevos por cada par de individuos. Por lo tanto, si elimino la mitad de la población y después la duplico, el tamaño total final será el mismo.
La cruza de la población en sí, es como se explicó anteriormente, se seleccionan los mejores individuos de a pares y se los cruza para que generen dos hijos. Uno con el primer individuo como padre y otro con el segundo individuo como padre. De esta manera, los individuos seleccionados sobreviven a la siguiente generación y generan la nueva generación de hijos.
Una vez que finaliza la etapa de cruza, se pasa a la etapa de mutación. En esta etapa, se llama al método “mutate” por cada individuo de la población y se decide de manera aleatoria, en base al coeficiente de mutación otorgado por el usuario, si el individuo debe mutar o no.
Finalmente, se pasa a la sección de condición de corte. En esta sección, se evalúa el fitness de cada individuo y se ordena la población de mayor a menor según el puntaje obtenido.
Una vez resuelto el puntaje, se evalúa si el primer individuo de la población obtuvo un puntaje máximo. En ese caso, se da por terminada la evolución y se devuelve la solución del problema.
Sino, se evalúa que el número de generaciones no supere al máximo de generaciones permitidas.
Y, por último, se evalúa si el tamaño de la población es demasiado chico o demasiado grande como para continuar.
En caso de que ninguna de estas condiciones se cumpla, se repetirá el ciclo hasta que alguna de ellas se cumpla.

```python
# Se inicia un nuevo proceso de evolución de una población
def evolve(self, n_population):

    # Se genera una nueva población
    self.generate(n_population)
    self.history.clear()

    break_condition = False
    currentGeneration = 0

    # Se hace iterar a la población en las diferentes generaciones
    while not(break_condition):

        # Seleccion: Se seleccionan los individuos de la población que van a poblar a la nueva generación
        self.population = self.population[0:(int)(len(self.population) * self.crossover_prop)]

        # Crossover: Se cruzan de a pares los individuos con mejores resultados de la generación anterior y se generan dos nuevos hijos
        for index in range(0, len(self.population), 2):
            if(index + 1 <= len(self.population)):
                self.crossover(index, self.population[index], self.population[index + 1])

        # Mutate: Se introducen mutaciones aleatorias a los individuos de la nueva generación
        for ind in self.population:
            self.mutate(ind, self.mutation_prop)

        # Condicion de corte: Se analiza si la evolución debe terminar

        # Se calcula el fitness de cada individuo y se lo reordena de mayor a menor (Siendo el primero el mejor)
        for ind in self.population: ind.fitness()
        self.population.sort(key = self.getScore, reverse = True)
        self.history.append(self.population[0].score)

        # Se consulta si el mejor individuo resuelve el problema
        if self.population[0].score >= 14 : break_condition = True

        # Se consulta si se supero la máxima cantidad de generaciones
        if currentGeneration >= self.maxGenerations : break_condition = True

        # Se analiza si la población crecio o decrecio mas de lo aceptado
        if len(self.population) <= 10 or len(self.population) > 50000: break_condition = True

        currentGeneration += 1


    self.status(currentGeneration - 1)
    return self.population[0].score, self.population[0]
```

Finalmente, se implementó el método “solve”. Este método, se creó para manipular a las diferentes poblaciones generadas. Dado que el algoritmo propuesto no ofrece un 100% de chances de obtener una solución posible, para mejorar esas posibilidades, se puede buscar evolucionar a muchas poblaciones seguidas hasta encontrar una que ofrezca una solución. Por eso, el método “solve” se encarga de llamar al método “evolve” y hacer un seguimiento de cada población generada. Generando una nueva población cada vez que la población anterior no haya obtenido una solución válida. Por supuesto, este bucle también tiene una vía de escape que es el número de iteraciones máximas permitidas. Esta variable también es otorgada por el usuario.

```
# Se inicia el proceso de resolución del problema
# Se determina el tamaño de la población, la cantidad de generaciónes para cada población y la cantidad de iteraciones que se deben evaluar
# Además, se asigna la proporción de mutaciones y la proporcion de cruce entre individuos
def solve(self, n_population, generations = 3, iterations = 10, mutationRatio = 1, crossoverRatio = 0.5):

    self.iterationCount = 0
    self.maxGenerations = generations
    self.mutation_prop = mutationRatio
    self.crossover_prop = crossoverRatio

    # Variables para mostrar los datos (No aportan a la resolución, se utilizan solo de manera informativa)
    fitSum = 0
    bestFit = 0
    bestIndi = Phenotype()
    bestHistory = []

    break_condition = False

    print("")
    print(f"Población creada con {len(self.population)} individuos")

    print("")
    print("Inicio del proceso iterativo")

    # Bucle de iteraciones.
    # Se hace evolucionar a n cantidad de poblaciones hasta conseguir una que cumpla al 100% con las restricciones del problema
    # o hasta que se cumpla la cantidad maxima de iteraciones
    while not(break_condition):
        self.iterationCount += 1
        # Se hace evolucionar a una nueva población
        fit, indi = self.evolve(n_population)
        fitSum += fit
        if (fit > bestFit) :
            bestFit = fit
            bestIndi.chromosome = indi.chromosome
            bestHistory = self.history.copy()
        # Se verifica que la población actual cumpla con las restricciones dadas
        if (fit >= 14) : break_condition = True
        # Se verifica que el número de iteraciones no supere al máximo
        if (self.iterationCount >= iterations) : break_condition = True
```

Conclusiones

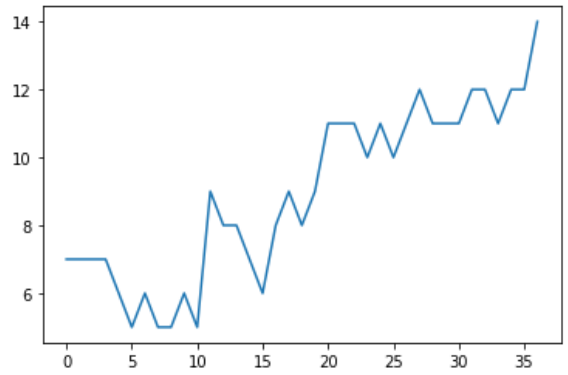
Antes que nada, es necesario aclarar que el problema a resolver, a diferencia del acertijo de Einstein, tiene más de una solución posible válida. Esto al principio genero confusión porque el algoritmo no siempre devolvía la misma solución.

El algoritmo generado se probó con varias configuraciones distintas, con poblaciones de entre 250 y 2.000 individuos, entre 100 y 2.000 generaciones, con coeficientes de mutación entre 0,05 y 1 y coeficientes de cruce entre 0 y 1.

En cuanto al tamaño de la población, se observó que a medida que el tamaño era mayor, se obtenían resultados exitosos con menos iteraciones pero cada iteración consumía más tiempo. Como para comparar, con una población de 750 individuos, se requieren menos de 25 iteraciones de 100 generaciones para conseguir una solución válida y cada iteración lleva alrededor de 1 segundo en

realizarse. Con una población de 2.000 individuos, cada iteración de 100 generaciones consume alrededor de 3 segundos pero en general, se consigue una solución con menos de 5 iteraciones. Por lo tanto, pareciera que a mayor población, mejor es la eficiencia del algoritmo pero también se evidencia que si la población crece mucho, podría demorarse por demás en pasar de generación a generación y dejar de ser eficiente.

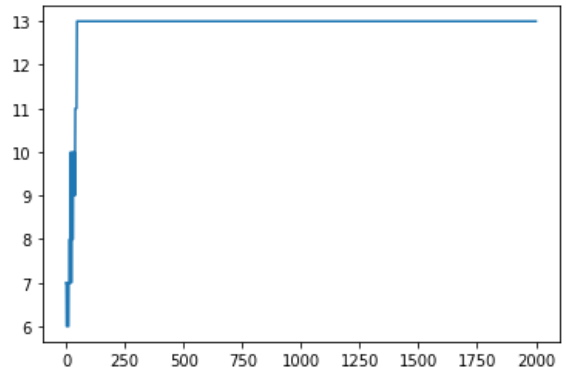
```
Población creada con 2000 individuos

Inicio del proceso iterativo
- Population: 2000 - Iteration: 1 - Generations: 36 - Fitness: 100.0 %

Fin del proceso, mejor resultado

Exito! Se encontro una solución posible

- Fitness: 100.0 %   (Avg: 100.0 %)

- Iterations: 1
```




```
- Genoma:
['010', '101', '010', '100', '010', '101', '010', '001', '001', '011', '001', '001', '011', '101', '101', '100',
'100', '100', '011', '100', '011', '011', '101', '010', '001']

- Individuo:
Casa 1:  ['blue', 'Developer', 'C#', 'Neo4j', 'Sublime Text']
Casa 2:  ['yellow', 'Hacker', 'Python', 'Cassandra', 'Vim']
Casa 3:  ['red', 'Mathematician', 'Java', 'Redis', 'Notepad++']
Casa 4:  ['white', 'Analyst', 'C++', 'HBase', 'Atom']
Casa 5:  ['green', 'Engineer', 'JavaScript', 'MongoDB', 'Brackets']

Tiempo transcurrido 00:00:01.34
```

Por otro lado, el número de generaciones se determinó en 100 dado que al evaluar los algoritmos en 2.000 generaciones y ver sus resultados, se nota que aquellas poblaciones que no logran la solución antes de las 100 generaciones, por lo general, parecerían entrar en una situación de “overfitting” dado que no terminan de solucionarse nunca.
En el grafico anterior, se muestra como con una población de 500 individuos a partir de la generación 100 la población no logra superar los 13 puntos de score. Esto se repite para otras configuraciones. Y por lo tanto, se determinó que si la población no había encontrado una solución antes de la generación número 100, entonces era más costoso seguir intentando que empezar de nuevo con otra población.

```
Población creada con 500 individuos

Inicio del proceso iterativo
- Population: 500 - Iteration: 1 - Generations: 2000 - Fitness: 92.86 %

Fin del proceso, mejor resultado

No se encontro una solución posible, intentelo de nuevo

- Fitness: 92.86 %   (Avg: 92.86 %)

- Iterations: 1
```




```
- Genoma:
['010', '101', '101', '100', '010', '100', '100', '100', '011', '100', '011', '011', '010', '010', '001', '101',
'010', '001', '001', '011', '001', '001', '011', '101', '101']

- Individuo:
Casa 1:  ['blue', 'Developer', 'JavaScript', 'Neo4j', 'Sublime Text']
Casa 2:  ['white', 'Analyst', 'C++', 'HBase', 'Atom']
Casa 3:  ['green', 'Engineer', 'C#', 'MongoDB', 'Brackets']
Casa 4:  ['yellow', 'Hacker', 'Python', 'Cassandra', 'Vim']
Casa 5:  ['red', 'Mathematician', 'Java', 'Redis', 'Notepad++']

Tiempo transcurrido 00:00:14.62
```

Solo a modo de observación, uno de los intentos que se hizo tratar de “salvar” a estas poblaciones fue introducir el concepto de “seeding” en donde si después de una n cantidad de generaciones, no hubo cambio en el puntaje, se debía eliminar a un cierto porcentaje de la población y generar nuevos individuos para reemplazarlos. Esto genero resultados prometedores pero menos eficientes, ocasionalmente generando soluciones en generaciones “trabadas”.
Adicionalmente, se experimentó con un concepto similar pero en vez de reemplazar individuos, se buscaba aumentar el coeficiente de mutación. Este modelo devolvió mejores resultados que el de “seeding” dado que permitía un “destrabe” más fino, pero aun así, no mejoraba la cantidad de generaciones.
La variable que más hizo notar la diferencia entre conseguir resultados exitosos y no, fue el coeficiente de mutación. A medida que aumentamos el coeficiente de mutación se obtuvieron resultados exitosos con menor cantidad de iteraciones, es decir, aumentaba la probabilidad de la población de conseguir un resultado exitoso.
Por eso, es que concluimos que la mejor configuración para conseguir una solución válida a este problema es con una población de 2.000 individuos, 100 generaciones, 5 iteraciones, un coeficiente de mutación de 1 (100%) y un coeficiente de cruce de 0,5.