Secuencia de números

Marcos Esteve Casdemunt

Octubre 2019

Contents

1 Descripción del problema

El problema de la secuencia de números consiste en encontrar un conjunto de operaciones que aplicadas a un conjunto de números se aproxime al resultado buscado.

Para nuestro problema se han propuesto las siguientes restricciones:

- El resultado de la operación división solo puede ser entero, por lo que si la división entre dos números a y b es decimal no se podrá realizar
- Se buscará aproximar la solución del problema sin pasarse, es decir, nunca superando el valor buscado
- Es obligatorio utilizar todos los números y además cada número solo podrá ser utilizado una vez
- El orden de aplicación de las operaciones es de izquierda a derecha

Por ejemplo dado los números 4, 10 ,7, 9, 2, 25 queremos encontrar un conjunto de operaciones que permita aproximar el valor de 134. Una posible solución podría ser:

$$4 + 10 * 7 + 9 + 2 + 25 = 134$$

2 Diseño del algoritmo

Para resolver el problema planteamos dos técnicas, por una parte el uso de algoritmos genéticos y, por otra parte el uso de enfriamiento simulado. Pasamos a comentar algunos aspectos interesantes del diseño de ambas soluciones.

2.1 Diseño del algoritmo genético

2.1.1 Codificación del individuo

En nuestro caso, un individuo se codifica como un vector donde para cada posición del vector se indica la operación que se va a realizar. Una posible codificación del individuo propuesto anteriormente podría ser:

$$[0, 2, 0, 0, 0]$$

Donde el número 0 indicaría la operación suma, 1 la resta, 2 la multiplicación y 3 la división.

2.1.2 Función *fitness*

La idea principal de la función *fitness* consiste en evaluar la calidad de un individuo, por ello se computa el resultado de las operaciones y se obtiene la diferencia con el valor objetivo. La técnica propuesta consiste en darle un mal valor de *fitness* a un individuo en caso de no cumplir las restricciones propuestas

Un individuo será optimo si se consigue obtener el valor objetivo, es decir, la diferencia entre el valor objetivo y el valor obtenido es 0.

2.1.3 Selección

El objetivo de la función selección consiste en seleccionar dos padres que se van a cruzar. Se han propuesto dos técnicas:

- Selección totalmente elitista donde solo aquellos individuos que tienen mejor fitness se cruzan.
- Selección proporcional donde se introduce una determinada probabilidad de cruce. De esta forma se consigue que los individuos que tienen mejor *fitness* tengan más probabilidades de reproducirse pero no la certeza.

2.1.4 Cruce

La operación cruce consiste en dados dos padres seleccionados previamente cruzarlos para obtener los descendientes. De nuevo se han propuesto dos técnicas:

Cruce de un punto, donde se corta la cadena genética de los padres en dos mitades y se intercambian para formar la cadena genética de los hijos. Un posible ejemplo lo podemos apreciar en la figura 1

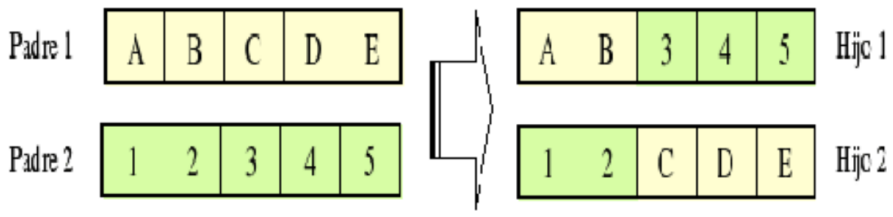

Figure 1: Cruce por un punto

Cruce uniforme, donde cada gen se elige de forma aleatoria del padre que viene. Un posible ejemplo lo podemos apreciar en la figura 2

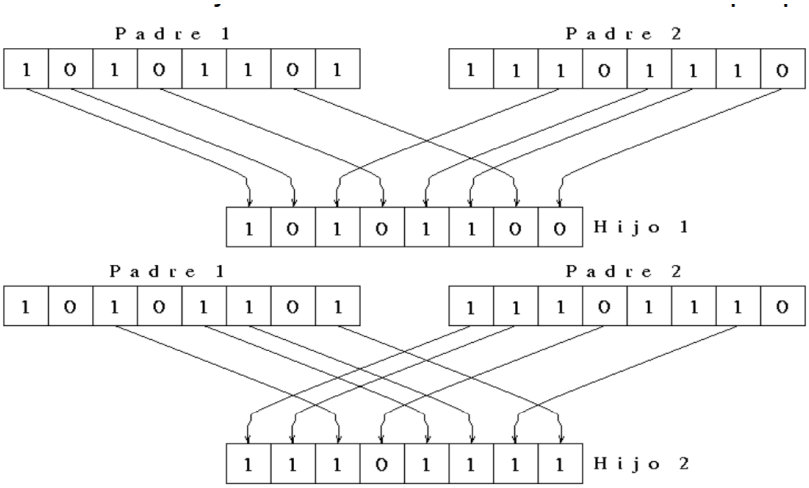

Figure 2: Cruce uniforme

2.1.5 Mutación

La operación mutación trata de modificar la carga genética de un hijo de acuerdo a una determinada probabilidad de mutación. En nuestro caso se ha implementado una mutación bit a bit, es decir, se aplica una probabilidad de mutacion p_{mut} a cada gen para decidir si este debe mutar. Un posible ejemplo lo podemos apreciar en la figura 3

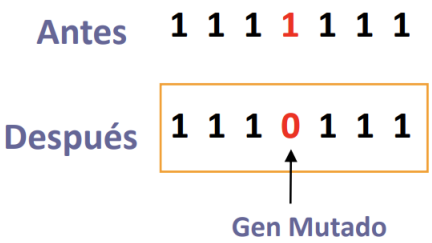

Figure 3: Mutación bit a bit

2.1.6 Reemplazo

La operación de reemplazo permite establecer el criterio de supervivencia en cada iteración o generación. En nuestro caso se han implementado distintas técnicas.

- Reemplazo por juicio final, donde solo aquel individuo más avanzado sobrevive y se mezcla con una nueva población generada aleatoriamente.

- Reemplazo por estado estacionario, donde en cada generación se generan 2 hijos y estos reemplazan a los peores individuos de la población.

2.1.7 Convergencia

Garantizar la convergencia del algoritmo es necesario para poder encontrar una solución en un tiempo específico. Para ello se han propuesto los siguientes mecanismos:

- Limitar el número máximo de generaciones que realizará el algoritmo.
- Finalizar si un individuo es óptimo, es decir, se han encontrado un conjunto de operaciones que permite obtener el valor objetivo buscado.
- Implementar un mecanismo de convergencia rápida que permite finalizar rápido en el caso de encontrar una solución factible y no encontrar otra con un *fitness* mejor en un número finito de iteraciones.

2.2 Diseño del enfriamiento simulado

En este caso, el algoritmo de enfriamiento simulado se trata de una metaheuristica de mejora. Este tipo de técnicas permite, dada una solución inicial buscar soluciones que mejoren una determinada función objetivo. Además al tratarse de una metaheuristica de búsqueda no monótona, es decir, una metaheuristca que permite movimientos que no mejoren a la solución actual, se consigue escapar de posibles óptimos locales de baja calidad.

Para el diseño de esta solución se ha optado por reutilizar la misma codificación del individuo y función fitness que en el caso del genético. Además, la función de convergencia es muy similar a la comentada en el algoritmo genético estableciendo un criterio de parada por superar un número máximo de iteraciones y otro al encontrar la solución óptima.

2.2.1 Obtener vecinos

A la hora de obtener los posibles vecinos de un determinado estado se ha tratado de obtener todas las posibles permutaciones factibles de ese estado. De esta forma restringimos posibles estados inválidos con respecto a las restricciones propuestas anteriormente

2.2.2 Operador aleatorio

Esta función se encarga de obtener una posible solución del conjunto de vecinos de un estado. Para ello se ha diseñado un proceso que selecciona a un individuo del conjunto de vecinos de forma aleatoria.

3 Implementación de la solución

El objetivo principal de este apartado es comentar algunos detalles de implementación del algoritmo genético y el enfriamiento simulado.

Para la implementación del algoritmo se ha optado por realizar una implementacion desde cero en Python gracias a la versatilidad que ofrece para el desarrollo.

3.1 Esquema algoritmo genético

```
    iteracion = 0
    terminar = False
    pueblo = iniciarPueblo()
    while not terminar:
        seleccionados = seleccion(pueblo, probCruce )
        nuevaGeneracion = cruce(seleccionados)
        nuevaGeneracion = mutacion(nuevaGeneracion,
    probMutacion)
        pueblo = reemplazo(pueblo, nuevaGeneracion)
        iteracion += 1
        converge = convergencia(pueblo)
        contadorEstancado = estancado()
        terminar = iteracion > maxIter or converge or (
    contadorEstancado > maximoMeseta and solucionFactible)
```

Listing 1: Esquema del algorítmo genético

Como se ha comentado anteriormente un aspecto interesante del algoritmo es la función de convergencia. Como podemos observar la función está formada por tres atributos: número máximo de iteraciones, encontrar la solución óptima o encontrar un posible mínimo local y no conseguir mejorar el *fitness* en un número de iteraciones.

3.1.1 Selección

Como se ha comentado en el apartado 2.1.3 se han implementado dos modelos de selección: por una parte una selección elitista y por otra, una selección elitista con una probabilidad de selección que permite que individuos que no sean los mejores se puedan reproducir. Los esquemas algorítmicos se pueden apreciar a continuación:

```
def seleccion_elitista(pueblo):
    preseleccion = ordenarPuebloPorFitness(pueblo)
    seleccionados = [list(preseleccion[0][0]), list(
    preseleccion[1][0])]
    return seleccionados
```

Listing 2: Selección elitista

Como se puede observar en el esquema anterior, solo los dos mejores individuos son seleccionados para cruzarse.

```
def seleccion_elitista_probCruce(pueblo, probCruce):
    preseleccion = ordenarPuebloPorFitness(pueblo)
  ix = 0
    seleccionados = []
    while True:
        if random.random() > 1-probCruce:
            seleccionados.append(preseleccion[ix][0])
        if len(seleccionados) == 2:
            return seleccionados
        if ix >= len(preseleccion)-1:
            ix = 0
        ix += 1
    return seleccionados
\label{elitistaProbCruce}
```

Listing 3: Selección elitista con probabilidad de cruce

Como se detalla en el esquema anterior se seleccionan los dos padres de acuerdo al orden del pueblo por fitness y una probabilidad que decide quien se va a cruzar.

3.1.2 Cruce

De acuerdo a lo comentado en 2.1.4 se han implementado dos funciones de cruce: por una parte el cruce de un punto y por otra el cruce uniforme. Los esquemas algorítmicos se detallan a continuación:

```
def crucePor1Punto(padre1, padre2):
    hijos = []
    longitudIndividuo = len(padre1)
    hijos.append(padre1[0:longitudIndividuo/2]+padre2[
    longitudIndividuo/2:-1])
    hijos.append(padre2[0:longitudIndividuo/2]+padre1[
    longitudIndividuo/2:-1])
    return hijos
```

Listing 4: Cruce por un punto

Como se está detallando en el anterior esquema el material genético de los padres se divide por la mitad y se combina para formar el material genético de los hijos.

```
def cruceUniforme(padre1, padre2):
    hijos = []
    longitudIndividuo = len(padre1)
    j = 0
    while j < 2:
        hijo = []
        for i in range(longitudIndividuo):
            if random.random() > 0.5:
                hijo.append(padre1[i])
            else:
                hijo.append(padre2[i])
        hijos.append(hijo)
```

```
            j += 1
        return hijos
```

Listing 5: Cruce uniforme

En el caso de cruce uniforme, el material genético de los hijos se decide a partir de una selección aleatoria del material de los padres.

3.1.3 Mutación

Como se comenta en el apartado 2.1.5 se ha implementado un único mecanismo de de mutación. El esquema algorítmico se puede apreciar a continuación:

```
def mutacion(nuevaGeneracion, probMutacion):
    generacionMutada = []
    for individuo in nuevaGeneracion:
        ix = 0
        individuoLista = list(individuo)
        while ix < len(individuoLista):
            if random.random() > 1-probMutacion:
                individuoLista[ix] = random.randint(0, 3)
            ix += 1
        generacionMutada.append(tuple(individuoLista))
    return generacionMutada
```

Listing 6: Mutación

Como se puede observar en el caso de superar la probabilidad de mutación se escoge una operación aleatoria entre la suma, resta multiplicación o división

3.1.4 Reemplazo

Tal y como se ha comentado en el apartado 2.1.6 se han implementado dos mecanismos de reemplazo: juicio final y estado estacionario. El esquema algoritmico de cada uno se puede apreciar a continuación.

```
def juicioFinal(pueblo, nuevaGeneracion):
    mejorIndividuo = calcularMejorIndividuo(pueblo,
    nuevaGeneracion)
    pueblo = iniciarPueblo()
    pueblo.add(mejorIndividuo)
    return pueblo
```

Listing 7: Juicio final

Como se observa se calcula el mejor individuo de una determinada población y se añade a una nueva población generada aleatoriamente.

```
def estadoEstacionario(pueblo, nuevaGeneracion):
    nuevoPueblo = pueblo + nuevaGeneracion
    puebloOrdenado = ordenarPuebloPorFitness(nuevoPueblo)
    return puebloOrdenado[0:-len(nuevaGeneracion)] #Cogemos
    todos menos los n peores individuos
```

Listing 8: Estado estacionario

Como se observa en el esquema algorítmico, se añade la nueva generación de n individuos al pueblo reemplazando los n peores individuos.

3.1.5 Inicio de la población

En primer lugar comentar que la población inicial se ha generado aleatoriamente entre las posibles operaciones disponibles.

Además un aspecto interesante del inicio de la población es como se ha escogido el tamaño de la misma. Como se ha comentado en las clases de teoría el tamaño de la población debe crecer exponencialmente con el tamaño del individuo o bien estar entre el tamaño del individuo y dos veces él. En nuestro caso se ha escogido está ultima propuesta. Generando una población de un tamaño de dos veces el tamaño del individuo.

3.2 Esquema enfriamiento simulado

```
1  sucesores = obtenerSucesores(solucionInicial)
2    iteraciones = 0
3    solucionActual = solucionInicial
4    mejorSolucion = solucionActual.
5    temperatura = temperaturaInicial
6    while len(sucesores) > 0 and not convergencia(iteraciones,
     solucionActual):
7
8        solucionNueva = obtenerSolucion(sucesores)
9        incrementoFitness = fitness(
10           solucionActual) - fitness(solucionNueva)
11       if incrementoFitness > 0:
12           solucionActual = solucionNueva.
13           sucesores = obtenerSucesores(solucionActual)
14           if fitness(solucionActual) < fitness(mejorSolucion)
     :
15               mejorSolucion = solucionActual
16       else:
17           if random.random() < math.e ** (incrementoFitness /
      temperatura):
18               solucionActual = solucionNueva
19               sucesores = obtenerSucesores(solucionActual)
20       iteraciones += 1
21       temperatura = actualizarTemperatura(iteraciones, k,
     temperatura)
22   return mejorSolucion
```

Listing 9: Esquema enfriamiento simulado

En el esquema algorítmico anterior podemos apreciar el funcionamiento del enfriamiento simulado. Basicamente se puede resumir de la siguiente forma: En el caso de encontrar una solución que mejore la mejor solución se acepta. En caso de obtener una solución peor se acepta de acuerdo a una probabilidad que depende de una determinada temperatura. Inicialmente para

valores altos de la temperatura se aceptarán casi todos los estados pero con el paso del tiempo la temperatura disminuye disminuyendo la probabilidad de aceptar un estado que no mejore la solución actual.

3.2.1 Actualizar temperatura

Para la actualización de la temperatura se han implementado dos métodos:

$$\alpha(i,T) = \frac{T_i}{1+kT_i} \quad k \approx 0$$

$$\alpha(i,T) = kT_i \quad k \in \{0.8, 0.99\}$$

Estos métodos permiten obtener una variación de la temperatura como la que podemos apreciar en la siguiente gráfica

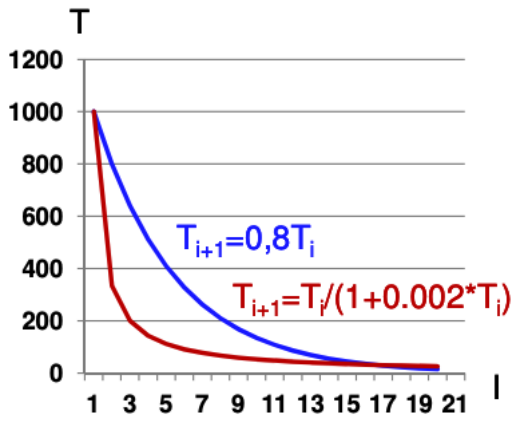

Figure 4: Evolución de la temperatura con las distintas formulas propuestas

Destacar, que en nuestras pruebas ambos métodos han demostrado obtener resultados similares, por lo que se ha optado por utilizar el primer método.

4 Evaluación

A la hora de evaluar las prestaciones de los algoritmos se ha optado por realizar una primera evaluación con los ejemplos del boletín.

Table 1: Ejemplos propuestos en el boletin

Números	**Objetivo**
[4, 10, 7, 9, 2, 25]	232
[10, 2, 9, 5, 7, 100]	298
[2, 75, 9, 6, 100, 8]	474
[3, 10, 7, 6, 75, 10]	381
[7, 3, 50, 25, 6, 100]	741
[2, 4, 75, 4, 50, 2]	502
[9, 6, 75, 7, 2, 50]	264

Tras esto se ha implementado un script que genera secuencias de números de distintos tamaños así como un valor objetivo siguiendo la siguientes restricciones:

- los n números se eligen al azar entre 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75 y 100 sabiendo que la probabilidad del 1 al 10 debe ser el doble a la probabilidad del resto
- el número objetivo se elige al azar entre $\frac{101*n}{6}$ y $\frac{999*n}{6}$

Table 2: Ejemplos generados con el script

Números	**Objetivo**
[50, 10, 9, 10, 5, 1]	764
[2, 3, 6, 9, 1, 25]	541
[10, 9, 100, 25, 5, 9]	153
[100, 50, 75, 5, 1, 10]	699
[4, 5, 3, 7, 5, 10]	415
[100, 1, 2, 3, 50, 6]	262
[8, 7, 1, 100, 7, 2]	813
[2, 10, 2, 4, 10, 6]	658
[75, 10, 1, 2, 7, 4]	364
[8, 4, 3, 8, 6, 25]	798
[1, 7, 10, 1, 6, 10, 9, 8, 4, 9, 3, 3]	1431
[25, 50, 6, 100, 75, 3, 5, 4, 3, 5, 9, 3]	815
[5, 5, 50, 9, 75, 4, 10, 75, 2, 1, 8, 3]	899
[2, 3, 2, 2, 25, 1, 6, 10, 1, 3, 6, 7]	1776
[1, 7, 5, 2, 50, 6, 10, 75, 8, 7, 10, 2]	1394
[5, 7, 75, 1, 75, 9, 3, 2, 75, 3, 25, 7]	247
[6, 100, 100, 10, 25, 4, 25, 8, 2, 9, 25, 9]	359
[5, 3, 7, 25, 10, 1, 9, 9, 3, 6, 7, 25]	1578
[5, 6, 10, 8, 8, 75, 8, 9, 6, 4, 1, 75]	1603
[25, 100, 50, 50, 10, 1, 6, 10, 100, 5, 100, 8]	834

4.1 Algoritmo genético

Para la evaluación del algoritmo genético se ha realizado una exploración exhaustiva por los distintos parámetros, estos parámetros son:

- selección elitista o elitista con probabilidad
- cruce por un punto o uniforme
- reemplazo por estado estacionario o juicio final
- probabiliad de cruce entre los valores [0.4, 0.6, 0.8]
- probabilidad de mutación entre los valores [0.025, 0.05, 0.075, 0.1]

En nuestro caso, el uso del reemplazo por estado estacionario con selección elitista con probabilidad y cruce uniforme produce buenos resultados en un tiempo razonable. En cambio,se ha observado en las pruebas, que en algunos de los casos en los que el individuo tiene una longitud elevada es interesante realizar reemplazo por juicio final con selección elitista o elitista con probabilidad y cruce uniforme. Se pueden apreciar los detalles de ejecución en la tabla inferior.

Table 3: Mediana de los tiempos dependiendo del reemplazo, selección o cruce elegidos

Tipo reemplazo	**Tipo seleccion**	**Tipo cruce**	**Tiempo**
estadoEstacionario	ElististaConProb	1punto	2.508342
		uniforme	1.861349
	elitista	1punto	3.001092
		uniforme	2.184276
juicioFinal	ElististaConProb	1punto	1.923871
		uniforme	1.882169
	elitista	1punto	2.188569
		uniforme	1.978196

Por otra parte, se ha observado en nuestros experimentos que la probabilidad de cruce 0.8 con la probabilidad de mutación 0.1 obtiene mejores convergencias.

Al analizar algunos casos de convergencia se ha observado que el algoritmo es capaz de encontrar soluciones aceptables en unas pocas iteraciones. Un posible ejemplo lo podemos observa en la imagen inferior donde el valor objetivo era 415 y tras apenas 20 iteraciones el sistema es capaz de dar una solución aproximada mientras que con apenas 100 iteraciones el sistema consigue obtener la combinación que da la solución optima.

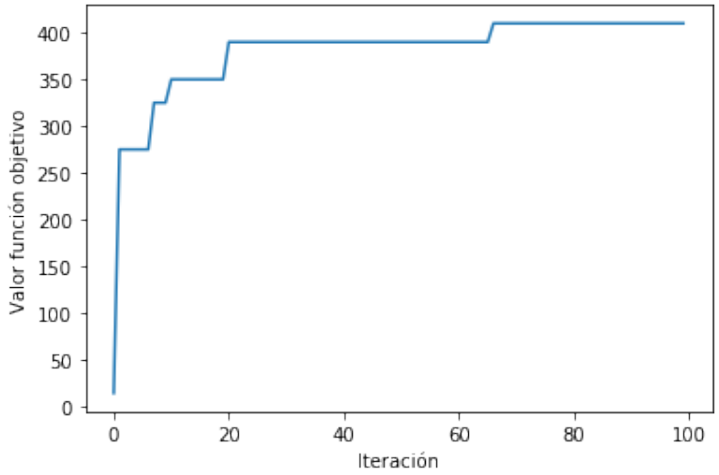

Figure 5: Convergencia para los números [4, 5, 3, 7, 5, 10] y valor objetivo 415

Por último a continuación se muestran algunos de los valores obtenidos al realizar una ejecución con reemplazo por juicio final selección elitista, cruce uniforme probabilidad de mutación 0.1 y probabilidad de cruce 0.8.

Mejor valor	**Objetivo**	**Iteraciones**	**Tiempo (seg)**
696	764	10017	1.868134
415	415	3	0.000745
262	262	53	0.010191
364	364	9	0.002667
1394	1394	6329	2.836094
247	247	1952	0.841080
359	359	406	0.181289
1578	1578	10042	4.316110
1603	1603	504	0.221076
833	834	10832	4.718243

4.2 Enfriamiento simulado

Por otra parte en la evaluación del enfriamiento simulado se ha realizado una exploración exhaustiva sobre un ejemplo complejo como pueden ser la combinación [5, 6, 10, 8, 8, 75, 8, 9, 6, 4, 1, 75] con el valor objetivo 1603. Los parámetros explorados se comentan a continuación

- temperatura entre los valores [1, 10, 100, 1000, 10000]
- k entre los valores [0.001, 0.01, 0.1, 1, 10]

Se ha iniciado el vector de operaciones todo a suma y para cada una de las posibilidades se han realizado 100 experimentos con el fin de obtener resultados mas estables y veraces.

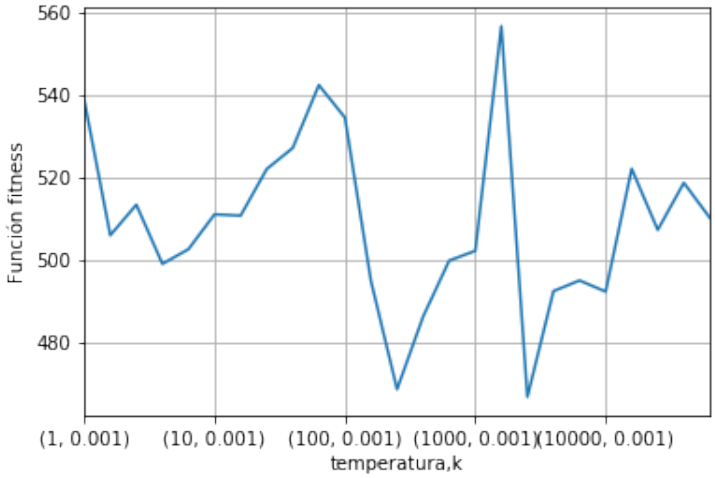

Figure 6: Evolución del valor de la función fitness al variar T y k

En la gráfica 6 podemos observar como evoluciona el valor de la función *fitness* medio al variar los parámetros de temperatura y k. A la vista de los resultados se observa que hay un mínimo con temperatura 1000 y k 0.1.

Por otra parte, utilizando los valores calculados anteriormente, se han explorado dos aproximaciones:

- Buscar una solución a partir de una solución inicial como puede ser la suma de todos los números
- Buscar una solución a partir de la mejor solución obtenida por el genético.

En el primer caso se observa que los resultados obtenidos son buenos aunque no comparables a los obtenidos con el algoritmo genético. Esto se puede deber a que enfriamiento simulado se queda estancado en mínimos locales mientras que el algoritmo genético es capaz de realizar una mejor exploración del espacio de busqueda. Algunos ejemplos los podemos apreciar en la siguiente tabla.

Table 4: Resultados obtenidos con la técnica de enfriamiento simulado

Números	**Objetivo**	**Mejor Valor**
[50, 10, 9, 10, 5, 1]	764	396
[2, 3, 6, 9, 1, 25]	541	350
[10, 9, 100, 25, 5, 9]	153	148
[100, 1, 2, 3, 50, 6]	262	261
[1, 7, 10, 1, 6, 10, 9, 8, 4, 9, 3, 3]	1431	532
[25, 50, 6, 100, 75, 3, 5, 4, 3, 5, 9, 3]	815	657
[5, 5, 50, 9, 75, 4, 10, 75, 2, 1, 8, 3]	899	735
[2, 3, 2, 2, 25, 1, 6, 10, 1, 3, 6, 7]	1776	1633
[5, 7, 75, 1, 75, 9, 3, 2, 75, 3, 25, 7]	247	232
[6, 100, 100, 10, 25, 4, 25, 8, 2, 9, 25, 9]	359	323
[5, 3, 7, 25, 10, 1, 9, 9, 3, 6, 7, 25]	1578	1391
[25, 100, 50, 50, 10, 1, 6, 10, 100, 5, 100, 8]	834	465

Por último, al intentar mejor las soluciones obtenidas por el algoritmo genético con enfriamiento simulado no se han obtenido resultado. Esto es debido a que el algoritmo genético es capaz de obtener soluciones óptimas en la mayoría de las ocasiones.

5 Conclusión

A la vista de lo expuesto anteriormente podemos extraer distintas conclusiones.

En primer lugar, se han implementado dos scripts en Python para solucionar el problema de la secuencia de números mediante las técnicas metaheuristicas de enfriamiento simulado y algoritmos genéticos. Durante el desarrollo y evaluación de las soluciones se ha observado que la implementación de los algoritmos es sencilla mientras que el ajuste de los parámetros es bastante tedioso.

Por otra parte se ha observado que en nuestro caso particular los algoritmos genéticos han sido capaces de obtener mejores resultados y, en muchos de los casos, soluciones óptimas.

Además un aspecto interesante a destacar es la velocidad con la que ambos algoritmos son capaces de obtener una solución factible con un buen resultado en la función objetivo.

Por último, destacar que este trabajo ha permitido obtener una idea más amplia de las técnicas metaheuristicas y como aplicarla a posibles problemas del mundo real.