CENTRO DE CIENCIAS BÁSICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

METAHEURÍSTICAS I

7° "A"

ACTIVIDAD 2_04. SIMULE LA PLANIFICACIÓN DE CARGA CON UN AG

Profesor: Francisco Javier Luna Rosas

Alumnos:

Almeida Ortega Andrea Melissa

Espinoza Sánchez Joel Alejandro

Flores Fernández Óscar Alonso

Gómez Garza Dariana

González Arenas Fernando Francisco

Orocio García Hiram Efraín

Fecha de Entrega: Aguascalientes, Ags.,**11** de octubre de 2021

Actividad 2_04. Simule la Planificación de Carga con un Algoritmo Genético

Antecedentes

Los Algoritmos Genéticos (AGs) son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Están basados en el proceso genético de los organismos vivos. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza de acorde con los principios de la selección natural y la supervivencia de los más fuertes, postulados por Darwin (1859).

Por imitación de este proceso, los Algoritmos Genéticos son capaces de ir creando soluciones para problemas del mundo real. La evolución de dichas soluciones hacia valores óptimos del problema depende en buena medida de una adecuada codificación de estas.

Los principios básicos de los Algoritmos Genéticos fueron establecidos por Holland (1975), y se encuentran bien descritos en varios textos – Goldberg (1989), Davis (1991), Michalewicz (1992), Reeves (1993) –. En la naturaleza los individuos de una población compiten entre sí en la búsqueda de recursos tales como comida, agua y refugio. Incluso los miembros de una misma especie compiten a menudo en la búsqueda de un compañero. Aquellos individuos que tienen más éxito en sobrevivir y en atraer compañeros tienen mayor probabilidad de generar un gran número de descendientes. Por el contrario, individuos poco dotados producirán un menor número de descendientes.

Esto significa que los genes de los individuos mejor adaptados se propagarán en sucesivas generaciones hacia un número de individuos creciente. La combinación de buenas características provenientes de diferentes ancestros puede a veces producir descendientes “super individuos”, cuya adaptación es mucho mayor que la de cualquiera de sus ancestros. De esta manera, las especies evolucionan logrando unas características cada vez mejor adaptadas al entorno en el que viven.

Los Algoritmos Genéticos usan una analogía directa con el comportamiento natural. Trabajan con una población de individuos, cada uno de los cuales representa una solución factible a un problema dado. A cada individuo se le asigna un valor o puntuación, relacionado con la bondad de dicha solución. En la naturaleza esto 3 equivaldría al grado de efectividad de un organismo para competir por unos determinados recursos. Cuanto mayor sea la adaptación de un individuo al problema, mayor será la probabilidad de que el mismo sea seleccionado para reproducirse, cruzando su material genético con otro individuo seleccionado de igual forma.

Este cruce producirá nuevos individuos – descendientes de los anteriores – los cuales comparten algunas de las características de sus padres. Cuanto menor sea la adaptación de un individuo, menor será la probabilidad de que dicho individuo sea seleccionado para la reproducción, y por tanto de que su material genético se propague en sucesivas generaciones. De esta manera se produce una nueva población de posibles soluciones, la cual reemplaza a la anterior y verifica la interesante propiedad de que contiene una mayor proporción de buenas características en comparación con la población anterior.

Así a lo largo de las generaciones las buenas características se propagan a través de la población. Favoreciendo el cruce de los individuos mejor adaptados, van siendo exploradas las áreas más prometedoras del espacio de búsqueda. Si el Algoritmo Genético ha sido bien diseñado, la población convergerá hacia una solución óptima del problema.

Los algoritmos de balanceo de carga están diseñados en esencia para distribuir igualmente la carga en los procesadores y maximizar su utilización minimizando la ejecución total de la tarea y del tiempo. Para lograr estos objetivos, el mecanismo de equilibrio de carga debe ser "justo" en la distribución de carga a través de los procesadores. Esto implica que la diferencia entre el más pesado y el más ligero los procesadores deben minimizarse.

Centralizado – descentralizado

En un algoritmo de equilibrio de carga centralizado, la carga global de información se recopila en un único procesador, llamado planificador central. Este planificador tomará todas las decisiones de equilibrio de carga en función de la información que se envíe de otros procesadores.

En el equilibrio de carga descentralizado, cada procesador en el sistema transmitirá su información de carga al resto de los procesadores para que las tablas de carga puedan actualizarse.

Estático – dinámico

Para problemas de equilibrio de carga estático, toda la información que rige las decisiones de equilibrio de carga se conoce en el avance. Las tareas se asignarán durante el tiempo de compilación según su conocimiento a priori y no se verá afectado por el estado del sistema en ese momento. Por otro lado, un mecanismo dinámico de equilibrio de carga tiene que asignar tareas a los procesadores dinámicamente a medida que llegan.

Umbrales

Si una cola de procesador es aceptable o no se determina por los umbrales ligeros y pesados utilizados. Si el tiempo de finalización de la tarea de un procesador (agregando la actual carga del sistema y aquellos aportados por las nuevas tareas) es dentro de los umbrales ligero y pesado que esta cola de procesador será aceptable. Si está por encima del umbral pesado o por debajo del umbral de ligero, entonces es inaceptable.

Al nosotros hacer uso de un solo procesador podemos decir que hacemos uso de un equilibrio de carga centralizado para de poder llevar a cabo nuestro algoritmo genético, ya que este será el que decidirá el equilibrio descarga para la distribución de toda la información que se pueda enviar a los demás procesadores.

De igual manera nuestro equilibrio de carga es estático, los diversos datos que vamos recabando dictan a lo largo de la ejecución del programa como se irán

asignando las tareas tomando en cuenta los diversos datos que se vayan registrando.

No hacemos uso de umbral debido a la enorme cantidad de dados que estamos analizando, ya que al recopilar tanta información y tener la necesidad de trabajar y analizar todos estos datos hacer uso de umbrales no es la mejor opción que se tiene por las limitantes que ofrece.

Complejidad computacional

La complejidad computacional es el campo de la teoría de la computación que estudia teóricamente la complejidad inseparable a la resolución de un problema. Los recursos que normalmente se estudian son:

- El tiempo: que por medio de una aproximación al número y patrón de pasos de ejecución de un algoritmo para solucionar algún problema.
- El espacio: que por medio de una aproximación a la cantidad de memoria que utilizamos para solucionar algún problema.

Podemos estudiar al igual otros parámetros, así como el número de procesadores que necesitamos para solucionar el problema en equivalente. La teoría de la complejidad se distingue de la teoría de la computabilidad en que ésta se necesita de la posibilidad de manifestar problemas como algoritmos más efectivos sin tomar en cuenta los recursos necesarios para esto.

Los problemas que tienen una solución con orden de complejidad lineal son los problemas que se resuelven en un tiempo que se vincula linealmente con su tamaño. Los problemas de decisión son clasificados en conjuntos de complejidad comparable llamados clases de complejidad.

Notación O-grande

Al tratar de caracterizar la eficiencia de un algoritmo en términos del tiempo de ejecución, independientemente de cualquier programa o computadora en particular, es importante cuantificar el número de operaciones o pasos que el algoritmo

requerirá. Si se considera que cada uno de estos pasos es una unidad básica de cálculo, entonces el tiempo de ejecución de un algoritmo puede expresarse como el número de pasos necesarios para resolver el problema.

Una buena unidad básica de cálculo para comparar los algoritmos de sumatoria mostrados anteriormente podría ser contar el número de instrucciones de asignación realizadas para calcular la suma. En una función, el número de instrucciones de asignación es 1 (Suma=0) más el valor de n (el número de veces que ejecutamos Suma= Suma + i). Podemos denotar esto por una función, digamos T, donde $T(n) = 1 + n$. El parámetro n a menudo se denomina el "tamaño del problema", y podemos interpretar la función como "$T(n)$ es el tiempo que se necesita para resolver un problema de tamaño n, a saber, $1 + n$ pasos".

En las funciones de sumatoria mencionadas anteriormente, tiene sentido utilizar el número de términos en la sumatoria para indicar el tamaño del problema.

Los científicos de la computación prefieren llevar esta técnica de análisis un poco más allá. Resulta que el número exacto de operaciones no es tan importante como determinar la parte más dominante de la función $T(n)$. En otras palabras, a medida que el problema se hace más grande, una parte de la función $T(n)$ tiende a dominar la parte restante. Este término dominante es lo que, al final, se utiliza para la comparación. La función orden de magnitud describe la parte de $T(n)$ que más rápido crece a medida que aumenta el valor de n. El orden de magnitud es a menudo llamado notación O-grande (por "orden") y se escribe como $O(f(n))$. Esta notación proporciona una aproximación útil al número real de pasos en el cálculo. La función $f(n)$ brinda una representación sencilla de la parte dominante de la función $T(n)$ original.

Usamos la notación Θ grande para acotar de manera asintótica el crecimiento de un tiempo de ejecución a que esté dentro de factores constantes por arriba y por abajo.

En el ejemplo anterior, $T(n) = 1 + n$. A medida que n se hace grande, la constante 1 será cada vez menos significativa para el resultado final. Si estamos buscando una aproximación para $T(n)$, entonces podemos despreciar el 1 y simplemente decir que el tiempo de ejecución es $O(n)$. Es importante notar que el 1 es ciertamente significativo para $T(n)$. No obstante, a medida que n se hace grande, nuestra aproximación será igualmente exacta sin él.

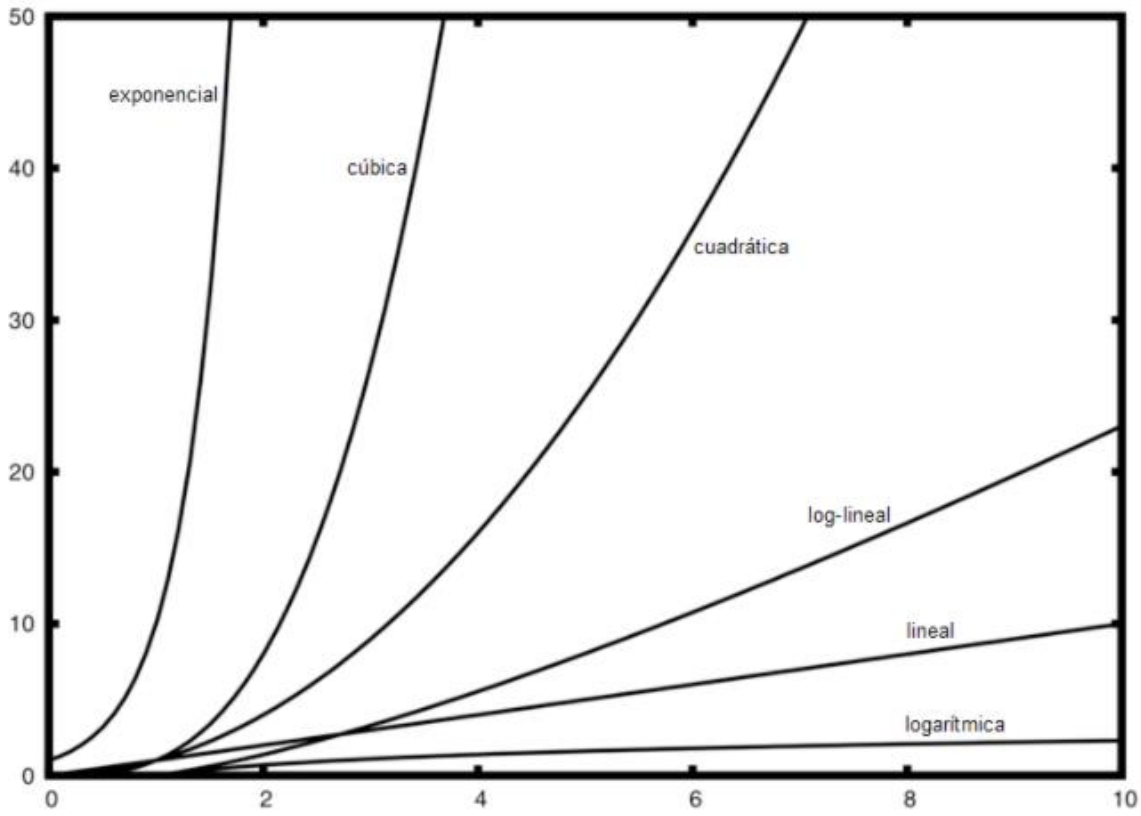
Supongamos que, para algún algoritmo, el número exacto de pasos es $T(n) = 5n^2 + 27n + 1005$. Cuando n es pequeño, digamos 1 ó 2, la constante 1005 parece ser la parte dominante de la función. Sin embargo, a medida que n se hace más grande, el término n^2 se convierte en el más importante. De hecho, cuando n es realmente grande, los otros dos términos se vuelven insignificantes en el papel que desempeñan para la determinación del resultado final. Una vez más, para aproximar $T(n)$ a medida que n se hace grande, podemos ignorar los otros términos y concentrarnos en $5n^2$. Además, el coeficiente 5 se vuelve insignificante cuando n se hace grande. Podemos decir entonces que la función T(n) tiene un orden de magnitud $f(n) = n^2$, o simplemente que es $O(n^2)$.

Una serie de funciones de orden de magnitud muy comunes aparecerán una y otra vez a medida en que se estudian algoritmos. Éstas se muestran en la Tabla 1. Para decidir cuál de estas funciones es la parte dominante de cualquier función $T(n)$, debemos compararlas entre sí a medida que n se hace grande.

Tabla 1: Funciones comunes para la notación O-grande

f(n)	Nombre
1	Constante
$\log n$	Logarítmica
n	Lineal
$n \log n$	Log-lineal
n^2	Cuadrática
n^3	Cúbica
2^n	Exponencial

La Figura 1 muestra las gráficas de las funciones comunes de la Tabla 1. Note que cuando n es pequeño, las funciones no están muy bien definidas una con respecto a otra. Es difícil saber cuál es la dominante. Sin embargo, a medida que n crece, existe una relación definida y es fácil compararlas entre sí.



En el caso de nuestro proyecto, la complejidad computacional nos afecta a un grado mayor del que pensábamos, dado que, cada uno de los integrantes contamos con procesadores diferentes en los que algunos trabajan a un menor tiempo y tienen mucho más espacio de memoria. Teniendo esto en cuenta, realizamos esta actividad valorando el procesador que tiene cada integrante del equipo para que, el algoritmo genético a base de nuestro registro reparta la cantidad de tweets adecuada para cada computadora.

Simulación de Distribución de Carga con Algoritmo Genético

La propuesta de nuestro equipo recae en un algoritmo genético centralizado con la siguiente denotación:

- PC de Melissa
- PC de Joel
- PC de Óscar
- PC de Dariana
- PC de Fernando
- PC de Hiram

Donde la computadora central que actuará como servidor será la PC de Joel. El algoritmo no trabajará bajo uso de umbrales, tampoco con carga dinámica.

Se basó mucho en el artículo proporcionado en clases *"Observations on Using Genetic Algorithms for Dynamic Load-Balancing"* sin embargo, propone una arquitectura muy atractiva para la distribución de carga tanto dinámica como variable, pues en él se expone una idea con tareas de distinto peso.

En el caso propio, el equipo pensó en principio que no existiría esto en la presentación del algoritmo, ya que las tareas a procesar son el análisis de sentimientos de los tweets. Cada tweets deberá pasar por el mismo procedimiento y por ello se consideraba que se analizaría con el mismo peso, ya que se cree que las diferencias de tiempo pueden ser discriminadas. Esto debido a que primero se planteó una asignación de pesos a cada tweet bajo la fórmula:

$$Peso = \ln(Palabras)$$

Pero para la velocidad de los tiempos de ejecución, el comportamiento logarítmico era despreciable, ya que la cantidad máxima de palabras era de 200 en cada tweet, lo que hizo considerar más en despreciar estas diferencias.

Entonces frente a un algoritmo de balanceo de carga de tareas donde cada tarea es un tweet y por lo tanto, cada tarea pesa lo mismo, la pregunta del equipo recaía en cómo realizar este procedimiento interesante, ya que se designó como valor 1 a cada tweet pero sin dar aún una propuesta atractiva en el proceso, lo más sencillo que podía hacerse, incluso sin un algoritmo genético, era repartir todas las tareas entre seis y que el procedimiento siga su curso.

Esta propuesta, aunque cierta, no realizaba ninguna tarea de evaluación heurística, por lo que el equipo discutió que este proyecto consiste en la distribución de carga de modo que un equipo no esté trabajando más de la cuenta mientras que otro equipo carece de trabajo. Para ello se propuso realizar el balance de carga con todas las tareas teniendo el mismo peso pero variando la potencia del procesador de cada computadora, por lo que primeramente se midió el poder de procesamiento de cada equipo.

Primeramente se diseñó un problema computacionalmente fácil pero que requiriera de mucho tiempo para completarse (véase el anexo 1) y se reportaron los tiempos de ejecución de cada equipo con una serie de ejecuciones.

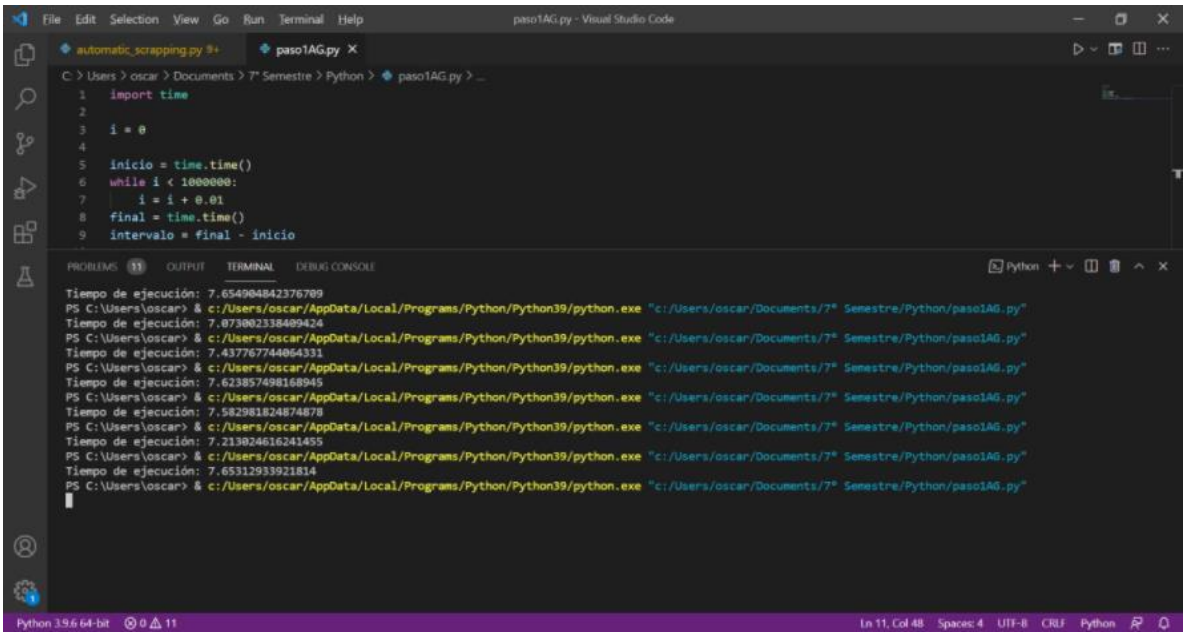
La PC de Melissa reportó los siguientes resultados

```
Tiempo de ejecución: 7.725000858306885

In [2]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.286704778671265

In [3]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.389048099517822

In [4]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.981545448303223

In [5]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.123982906341553

In [6]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.439027309417725

In [7]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 7.987653017044067

In [8]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.185094594955444

In [9]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 9.21445369720459

In [10]: runfile('C:/Users/mel_a/Documents/I.C.I/Python/untitled0.py',
wdir='C:/Users/mel_a/Documents/I.C.I/Python')
Tiempo de ejecución: 8.52103066444397
```

Con una media de 9.311 segundos para la PC de Melissa.

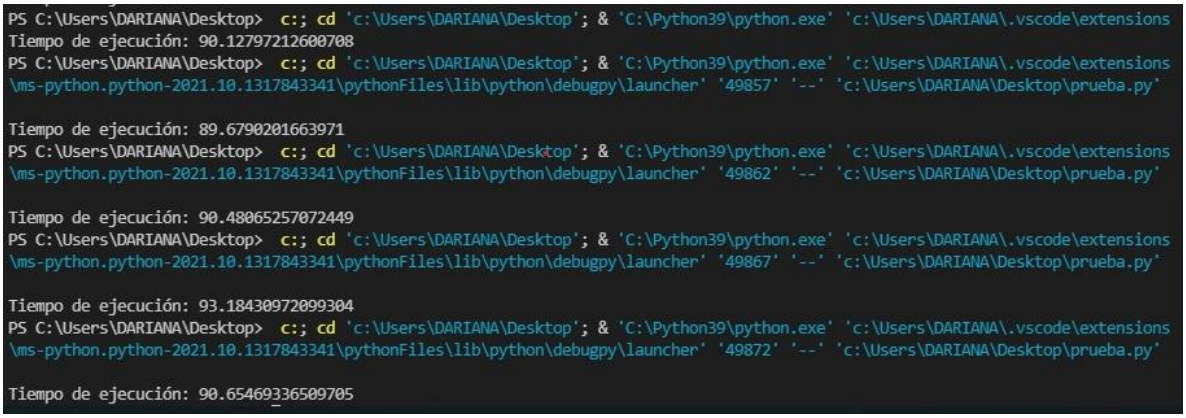
La PC de Joel realizó diez ejecuciones para un reporte promedio de 14.8685 segundos.

La PC de Óscar reportó los siguientes resultados:



Con una media de 7.458 segundos para la PC de Óscar.

La PC de Dariana reportó los siguientes resultados:



Con una media de 90.8248 segundos para la PC de Dariana.

La PC de Fernando realizó diez ejecuciones para un reporte promedio de 16.1 segundos.

La PC de Hiram reportó los resultados siguientes:

Con una media de 23.006 segundos para la PC de Hiram.

Los resultados fueron los siguientes:

PC de Melissa	9.311
PC de Joel	14.8685
PC de Óscar	7.458
PC de Dariana	90.8248
PC de Fernando	16.1
PC de Hiram	23.006

Con estos datos conocidos, la idea era asignar la cantidad adecuada de tweets para que todos los procesadores, con su debida potencia, procesaran los tweets que les permitieran sus capacidades.

El algoritmo genético propuesto está planteado bajo los términos siguientes:

Los cromosomas serán tuplas de seis números, cada uno representando la cantidad de tweets que cada persona deberá procesar, es decir, un cromosoma tiene la siguiente apariencia:

$$[TMelissa][TJoel]\left[T\acute{O}scar\right][TDariana][TFernando][THiram]$$

Se genera una población aleatoriamente, tomando como base la cantidad total de tweets extraídos, en este caso, 6,942,021 tweets en total. Una restricción a tomar en cuenta es que la suma de todos los valores de la tupla debe dar este valor:

$$TMelissa\ + TJoel + T\acute{O}scar + TDariana + TFernando + THiram = 6{,}942{,}021$$

Una vez que toda la población se ha generado, el algoritmo genético procede de forma común: se aplica elitismo, selección, cruzamiento y mutación.

Para el elitismo no existe mucho problema, pues sólo se seleccionarán los n mejores cromosomas para la siguiente generación. El problema es cómo determinar qué cromosoma es mejor que otro.

Para plantear la función objetivo primeramente se pensó que la cantidad de tweets dividido entre el tiempo registrado de una persona deberá ser muy similar a la cantidad de tweets dividido entre el tiempo registrado de otra persona, por ejemplo:

$$\frac{TMelissa}{9.311} = \frac{TJoel}{14.8685}$$

Esto debe ocurrir para todos los integrantes del equipo:

$$\frac{TMelissa}{9.311} = \frac{TJoel}{14.8685} = \frac{T\acute{O}scar}{7.458} = \frac{TDariana}{90.8248} = \frac{TFernando}{16.1} = \frac{THiram}{23.006}$$

La mejor forma para ajustar todos estos valores a uno similar podrá ser la herramienta estadística conocida como la desviación estándar. Para ello primero se calculó la media de todos estos valores:

$$Media = \frac{\left(\frac{TMelissa}{9.311} + \frac{TJoel}{14.8685} + \frac{T\acute{O}scar}{7.458} + \frac{TDariana}{90.8248} + \frac{TFernando}{16.1} + \frac{THiram}{23.006}\right)}{6}$$

Posteriormente se calculará un valor muy similar a la definición formal de desviación estándar en Estadística Descriptiva:

$$\sigma = \left|\frac{TMelissa}{9.311} - Media\right| + \left|\frac{TJoel}{14.8685} - Media\right| + \left|\frac{T\acute{O}scar}{7.458} - Media\right| + \left|\frac{TDariana}{90.8248} - Media\right| + \left|\frac{TFernando}{16.1} - Media\right| + \left|\frac{THiram}{23.006} - Media\right|$$

Ahora, el valor más pequeño de σ será el que describa mejor a un cromosoma, pero por fines prácticos del grupo, se conoce mejor la forma de evaluarlos cuando una función expresa a un cromosoma como “mejor” cuando ésta le asigna un valor mayor, para ello se invirtió la función:

$$Función\ objetivo = \frac{1}{\sigma}$$

Finalmente, se tuvieron pequeños inconvenientes en el desarrollo del código que llevaron a corregir la función debido a sus asíntotas, de modo que la función final será la siguiente:

$$Función\ objetivo = \frac{1}{(\sigma \times 1{,}000) + 1} \times 100$$

Esta misma función se usa para la selección por ruleta en el siguiente paso.

Como siguiente operador se tiene el cruzamiento, que es difícil debido al modelo que se planteó, sin embargo se encontró una buena forma de realizar el cruzamiento con este modelo. Sean las tuplas $T1$ y $T2$ las dos seleccionadas para realizar cruzamiento:

$$T1 = [T1Melissa][T1Joel][T1Óscar][T1Dariana][T1Fernando][T1Hiram]$$

$$T2 = [T2Melissa][T2Joel][T2Óscar][T2Dariana][T2Fernando][T2Hiram]$$

A diferencia de un punto de corte en cadenas binarias, en este caso se eligen dos números de cada tupla y se buscará el menor de ellos; se generará un número aleatorio entre 0 y el menor de los cuatro seleccionados y se rotará este número $addition$. Supongamos que se elige aleatoriamente $TDariana$ y $THiram$ de ambas tuplas. Los nuevos valores después del cruzamiento serán los siguientes:

$$T1 = [T1Melissa][T1Joel][T1Óscar][T1Dariana - addition][T1Fernando][T1Hiram + addition]$$

$$T2 = [T2Melissa][T2Joel][T2Óscar][T2Dariana + addition][T2Fernando][T2Hiram - addition]$$

Modelo de cruzamiento inspirado en una técnica Simplex de Investigación de Operaciones para balanceo de cargas en el método mencionado anteriormente. Todos los números de las tuplas se combinarán para crear dos hijos.

La mutación realiza un procedimiento similar. En este caso se toma el número menor de la tupla y se generará un número aleatorio entre 0 y el menor de toda la tupla y se rotará este número $addition$. Supongamos la siguiente tupla:

$$T1 = [T1Melissa][T1Joel][T1Óscar][T1Dariana][T1Fernando][T1Hiram]$$

Después de la mutación, la tupla tendrá la siguiente forma:

$$T1 = [T1Melissa - addition][T1Joel + addition][T1Óscar - addition][T1Dariana + addition][T1Fernando - addition][T1Hiram + addition]$$

Se plantearon estos dos métodos de cruzamiento y mutación respectivamente para no perder la restricción de que cada tupla debe cumplir con la suma de todos sus números equivalente al número total de tweets.

Posteriormente se repite el procedimiento para cada generación.

Ahora se llevó lo anteriormente explicado a código Python (véase anexo 2) y lo primero que se realizó fue calibrar el algoritmo genético:

```
#%% Calibrar variables
total_tweets = 6942021
qe = 6 # Inamovible
qi = 60 # Movible
pc = 75 # Movible
pm = 40 # Movible
qelitism = 5 # Movible
qg = 10
sample = [[], [], []]
power_process = [9.311, 14.8685, 7.458, 90.8248, 16.1, 23.006] # Orden alfabético: Melissa, Joel, Óscar, Dariana,
ig = 0
```

Posteriormente se crea la población inicial y se realizan las evaluaciones con respecto a la función objetivo:

```
#%% AG: Creación de la población aleatoria
sample[0] = FirstSample(qe,qi,total_tweets)
PrintSample(qi, sample)
```

```
#%% AG: Procedimiento de cada generación
while ig < qg:
    print("Población " + str(ig))
    print("")

    # Evaluamos a todas las muestras
    sample[1] = evaluate(qi,sample,power_process)

    print("Población " + str(ig) + " al ser evaluada")
    PrintSample(qi,sample)
    print("")

    # Evaluamos la suma acumulada
    sample[2] = evaluateA(qi,sample)

    print("Población " + str(ig) + " al ser evaluada con acumulación")
    PrintSample(qi,sample)
    print("")

    # Ordenamos por valor de f(x) es decir, por sample[1]
    sample = Sorting(sample, qi)

    print("Población " + str(ig) + " al ser ordenada")
    PrintSample(qi,sample)
    print("")

    # Reevaluamos la suma acumulada
    sample[2] = evaluateA(qi,sample)

    print("Población " + str(ig) + " al ser reevaluada")
    PrintSample(qi,sample)
    print("")
```

Luego se crea una nueva estructura para la nueva generación y se aprovecha para pasar por elitismo a los mejores individuos:

```
# Creamos una nueva muestra vacía
newSample = newSampleCreator(qi)

print("Nueva muestra " + str(ig + 1) + " vacía")
print(newSample)
print("")

print("Se aplicará elitismo")
x = input()

# Se aplica elitismo
ii = 0
while ii < qelitism:
    newSample[0][ii] = sample[0][ii]
    #newSample[1][ii] = sample[1][ii]
    #newSample[2][ii] = sample[2][ii]
    ii = ii + 1

print("Nueva muestra " + str(ig + 1) + " después de elitismo")
for i in range(qi):
    try:
        print(str(newSample[0][i]) + " = " + str(sum(newSample[0][i])))
    except:
        pass
print("")
print("Se aplicará cruzamiento")
x = input()
```

Después se realiza el cruzamiento primeramente realizando la selección de dos padres:

```
# Se aplicará cruzamiento
while ii < qi:
    # Seleccionamos por ruleta
    # Padre 1
    random_pick = random.uniform(0, sample[2][qi - 1])

    for ic in range(qi):
        if random_pick < sample[2][ic]:
            c1 = sample[0][ic]
            print("Se elige como padre 1 el individuo " + str(ic) + "(random_pick = " + str(random_pick) + ")")
            break

    # Padre 2
    random_pick = random.uniform(0, sample[2][qi - 1])

    for ic in range(qi):
        if random_pick < sample[2][ic]:
            c2 = sample[0][ic]
            print("Se elige como padre 2 el individuo " + str(ic) + "(random_pick = " + str(random_pick) + ")")
            break
```

Lo siguiente es evaluar si los padres elegidos realizarán cruzamiento o no:

```
ii = 0
while ii < qi:
    # Se evalúa si se aplicará mutación
    random_num = random.randint(0,99)
    if random_num < pm:
        print("El cromosoma " + str(ii) + " ha activado mutación (Probabilidad: " + str(random_num) +
        print("")

        processors = [0, 1, 2, 3, 4, 5]
        random.shuffle(processors) # Se mezclarán y se hará una sumas balanceadas cíclica

        print("Tupla mezclada: " + str(processors))
        print("")

        print("Valores antes de la mutación: ")
        print(newSample[0][ii])
        print("")

        min_value = min(newSample[0][ii])
        operation = random.randint(0, 1) # Elige la operación con la que empezará la suma equilibrada
        addition = random.randint(0, min_value)

        print("Tupla " + str(processors))
        print("min_value = " + str(min_value))
        print("addition = " + str(addition))
        print("")
```

Posteriormente se harán lo que en el equipo se le denominó “sumas equilibradas”, que es el concepto de equilibrio por medio de una técnica Simplex:

```
for i in [0, 2, 4]:
    # Sumas equilibradas
    min_value = min(c1[processors[i]], c1[processors[i + 1]], c2[processors[i]], c2[processors[i + 1]])
    operation = random.randint(0, 1) # Elige la operación con la que empezará la suma equilibrada
    addition = random.randint(0, min_value)

    print("Tuplas " + str(processors[i] + 1) + " y " + str(processors[i + 1] + 1))
    print("min_value = " + str(min_value))
    print("addition = " + str(addition))
    print("")

    if operation == 0:
        c2[processors[i + 1]] = c2[processors[i + 1]] - addition
        c1[processors[i + 1]] = c1[processors[i + 1]] + addition
        c1[processors[i]] = c1[processors[i]] - addition
        c2[processors[i]] = c2[processors[i]] + addition
    else:
        c2[processors[i + 1]] = c2[processors[i + 1]] + addition
        c1[processors[i + 1]] = c1[processors[i + 1]] - addition
        c1[processors[i]] = c1[processors[i]] + addition
        c2[processors[i]] = c2[processors[i]] - addition

print("Valores después del cruzamiento: ")
print(c1)
print(c2)
print("")
```

Si no se cruzan, el procedimiento contrario es heredarlos de igual manera.

Después se realiza la mutación:

```
# Se aplica mutación
ii = 0
while ii < qi:
    # Se evalúa si se aplicará mutación
    random_num = random.randint(0,99)
    if random_num < pm:
        print("El cromosoma " + str(ii) + " ha activado mutación (Probabilidad: " + str(random_num) + " < "
        print("")

        processors = [0, 1, 2, 3, 4, 5]
        random.shuffle(processors) # Se mezclarán y se hará una sumas balanceadas cíclica

        print("Tupla mezclada: " + str(processors))
        print("")

        print("Valores antes de la mutación: ")
        print(newSample[0][ii])
        print("")

        min_value = min(newSample[0][ii])
        operation = random.randint(0, 1) # Elige la operación con la que empezará la suma equilibrada
        addition = random.randint(0, min_value)

        print("Tupla " + str(processors))
        print("min_value = " + str(min_value))
        print("addition = " + str(addition))
        print("")

        if operation == 0:
            newSample[0][ii][processors[0]] = newSample[0][ii][processors[0]] - addition
            newSample[0][ii][processors[1]] = newSample[0][ii][processors[1]] + addition
            newSample[0][ii][processors[2]] = newSample[0][ii][processors[2]] - addition
            newSample[0][ii][processors[3]] = newSample[0][ii][processors[3]] + addition
            newSample[0][ii][processors[4]] = newSample[0][ii][processors[4]] - addition
            newSample[0][ii][processors[5]] = newSample[0][ii][processors[5]] + addition
```

El procedimiento se repetirá según la cantidad de generaciones deseado.

Un ejemplo de ejecución de código es el siguiente. A continuación se muestra la población inicial al ser evaluada y ordenada:

```
Población 0 al ser reevaluada
[2969361, 1539170, 1815651, 18523, 412987, 186329] = 6942021        31622037    31622037
[1865628, 1340774, 3321758, 89961, 257220, 66680] = 6942021         31562903    63184940
[2199470, 2560613, 1966710, 93170, 32732, 89326] = 6942021      31546451    94731391
[758906, 3125166, 1812820, 347267, 419893, 477969] = 6942021        30975301    125706692
[668383, 1823891, 2580073, 1260065, 74070, 535539] = 6942021        30969025    156675717
[2547676, 1679639, 1001566, 977186, 302243, 433711] = 6942021       30870443    187546160
[875519, 2158420, 3016210, 586654, 191857, 113361] = 6942021        30774515    218320675
[1845586, 3211955, 444884, 202954, 1088166, 148476] = 6942021       30756221    249076896
[1680857, 1036057, 3977078, 218624, 2748, 26657] = 6942021      30399821    279476717
[2650913, 2797244, 147921, 126160, 717987, 501796] = 6942021        30170303    309647020
[227888, 2168853, 2913690, 981996, 641130, 8464] = 6942021      30168005    339815025
[2948873, 2369384, 816626, 341442, 283551, 182145] = 6942021        30106627    369921652
[3583567, 791277, 1486148, 492479, 398246, 190304] = 6942021        29768075    399689727
[568776, 723038, 3902812, 558178, 421514, 767703] = 6942021         29752731    429442458
[3441356, 688951, 2009969, 652667, 136069, 13009] = 6942021         29732121    459174579
[3793563, 313321, 2046431, 372089, 361754, 54863] = 6942021         29588863    488763442
[2551365, 3115234, 757508, 258417, 197677, 61820] = 6942021         29575993    518339435
[2482156, 3272432, 934092, 193415, 44168, 15758] = 6942021      29530019    547869454
[985930, 972146, 1833844, 2523668, 351808, 274625] = 6942021        29366973    577236427
[2650989, 2815668, 873007, 598450, 2346, 1561] = 6942021        29295811    606532238
[4078958, 1395460, 1084937, 340042, 507, 42117] = 6942021       29082167    635614405
[223062, 664049, 2854577, 1731647, 371117, 1097569] = 6942021       28842531    664456936
[4110636, 864439, 1453124, 461013, 42878, 9931] = 6942021       28776869    693233805
[4405407, 1578050, 554754, 168294, 221645, 13871] = 6942021         28749945    721983750
[3165453, 1296076, 764776, 1425381, 279518, 10817] = 6942021        28738499    750722249
[3678398, 28387, 1106706, 736053, 1260842, 131635] = 6942021        28597099    779319348
[4490396, 1031536, 1239910, 73570, 99624, 6985] = 6942021       28579967    807899315
[2606706, 1447308, 488702, 2090652, 26763, 281890] = 6942021        28525451    836424766
[4574472, 1273115, 925421, 9432, 104218, 55363] = 6942021       28411815    864836581
[880266, 4004944, 904777, 792508, 357802, 1724] = 6942021       28325263    893161844
[4617886, 656977, 1571440, 23067, 52409, 20242] = 6942021       28324987    921486831
[445283, 939017, 4049558, 1292680, 176507, 38976] = 6942021         28106749    949593580
[202769, 3178837, 1264093, 1763174, 124357, 408791] = 6942021       28036145    977629725
[3758510, 2760947, 317397, 22715, 40287, 42165] = 6942021       28027703    1005657428
[4841075, 1390739, 295531, 202586, 140523, 71567] = 6942021         27832845    1033490273
[4935908, 540620, 825644, 165224, 459982, 14643] = 6942021      27688943    1061179216
[3679508, 3014407, 108490, 35119, 12794, 91703] = 6942021       27678787    1088858003
[4331043, 1039278, 725498, 792389, 49251, 4562] = 6942021       27673303    1116531306
[88008, 5093751, 1741794, 1174, 7228, 10066] = 6942021      27373257    1143904563
[5349748, 144004, 1408908, 28851, 670, 9840] = 6942021      26861263    1170765826
[463477, 4867792, 253975, 698397, 257604, 400776] = 6942021         26787789    1197553615
[1322455, 5121094, 563, 480577, 3847, 13485] = 6942021      26716825    1224270440
```

Observemos que todas las tuplas suman 6,942,021. El número siguiente es su valor como función objetivo y el siguiente se trata del valor acumulado para realizar elitismo, que de hecho, al aplicar elitismo en esta población se obtiene lo siguiente:

```
Nueva muestra 1 vacía
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

Se aplicará elitismo


Nueva muestra 1 después de elitismo
[2969361, 1539170, 1815651, 18523, 412987, 186329] = 6942021
[1865628, 1340774, 3321758, 89961, 257220, 66680] = 6942021
[2199470, 2560613, 1966710, 93170, 32732, 89326] = 6942021
[758906, 3125166, 1812820, 347267, 419893, 477969] = 6942021
[668383, 1823891, 2580073, 1260065, 74070, 535539] = 6942021
```

Posteriormente en el cruzamiento, observemos cómo el algoritmo avisa que no se ha activado el cruzamiento:

```
Se elige como padre 1 el individuo 3(random_pick = 110888477.71224487)
Se elige como padre 2 el individuo 17(random_pick = 542848757.1963074)
Los cromosomas 49 y 50 no han activado cruzamiento (Probabilidad: !82 < 75!)
```

Asimismo, se aprecia a continuación la activación del cruzamiento:

```
Se elige como padre 1 el individuo 53(random_pick = 1515027406.0065634)
Se elige como padre 2 el individuo 43(random_pick = 1270469611.342023)
Los cromosomas 53 y 54 han activado cruzamiento (Probabilidad: 54 < 75)
```

Cuando esto sucede, se reparten en pares los valores de las tuplas para mezclarse como se muestra a continuación:

```
Tupla mezclada: [4, 2, 3, 5, 0, 1]

Valores antes del cruzamiento:
[445283, 939017, 4049558, 1292680, 176507, 38976]
[4394701, 1559395, 680532, 186949, 95867, 24577]

Tuplas 5 y 3
min_value = 95867
addition = 10059

Tuplas 4 y 6
min_value = 24577
addition = 13195

Tuplas 1 y 2
min_value = 445283
addition = 344965

Valores después del cruzamiento:
[100318, 1283982, 4039499, 1279485, 186566, 52171]
[4739666, 1214430, 690591, 200144, 85808, 11382]
```

Generando una población de individuos ya cruzados:

```
[2910289, 2355836, 817043, 345955, 322135, 190763] = 6942021
[1321644, 5129616, 50, 481090, 4658, 4963] = 6942021
[668843, 1826377, 2574342, 1270469, 71584, 530406] = 6942021
[668843, 1826377, 2574342, 1270469, 71584, 530406] = 6942021
[907762, 3985833, 877281, 793891, 376913, 341] = 6942021
[2446418, 3264221, 961149, 219354, 25496, 25383] = 6942021
[668843, 1826377, 2574342, 1270469, 71584, 530406] = 6942021
[6311967, 610862, 2571, 2131, 1055, 13435] = 6942021
[2456600, 3354140, 760853, 19511, 292442, 58475] = 6942021
[3536121, 450045, 2006624, 891573, 41304, 16354] = 6942021
[5559159, 725333, 625816, 3467, 11910, 16336] = 6942021
[5422971, 70781, 1408387, 28441, 1191, 10250] = 6942021
[6295358, 66474, 359857, 192163, 27896, 273] = 6942021
[3733768, 373116, 2115273, 303247, 361854, 54763] = 6942021
[2910289, 2355836, 817043, 345955, 322135, 190763] = 6942021
[3839643, 2758954, 275755, 14468, 44311, 8890] = 6942021
[3532215, 2699599, 297181, 225772, 138873, 48381] = 6942021
[4987467, 1698043, 97250, 19437, 15345, 124479] = 6942021
[5801053, 302214, 475984, 353544, 9219, 7] = 6942021
[2456600, 3354140, 760853, 19511, 292442, 58475] = 6942021
[3472949, 1287102, 457280, 1414743, 288492, 21455] = 6942021
[5280471, 121154, 731806, 368189, 128, 440273] = 6942021
[4739666, 1214430, 690591, 200144, 85808, 11382] = 6942021
[3689104, 47042, 980928, 717398, 1386620, 120929] = 6942021
[4987467, 1698043, 97250, 19437, 15345, 124479] = 6942021
[5280471, 121154, 731806, 368189, 128, 440273] = 6942021
[5752105, 1131588, 35261, 17545, 910, 4612] = 6942021
[2446418, 3264221, 961149, 219354, 25496, 25383] = 6942021
[6295358, 66474, 359857, 192163, 27896, 273] = 6942021
[784261, 2289257, 2885373, 631150, 213997, 137983] = 6942021
[6533076, 329256, 7074, 36297, 36281, 37] = 6942021
[1844933, 3214720, 442119, 187687, 1103433, 149129] = 6942021
[3839643, 2758954, 275755, 14468, 44311, 8890] = 6942021
[5801053, 302214, 475984, 353544, 9219, 7] = 6942021
[6295358, 66474, 359857, 192163, 27896, 273] = 6942021
[3733768, 373116, 2115273, 303247, 361854, 54763] = 6942021
[755218, 659469, 3578184, 581861, 408772, 958517] = 6942021
[6047208, 22261, 844701, 24559, 3287, 5] = 6942021
[758906, 3125166, 1812820, 347267, 419893, 477969] = 6942021
[2446418, 3264221, 961149, 219354, 25496, 25383] = 6942021
[5926199, 527918, 274824, 53085, 158711, 1284] = 6942021
[755218, 659469, 3578184, 581861, 408772, 958517] = 6942021
```

Para la mutación pasa algo similar, pues se puede apreciar el caso de cuando no se activa la mutación como cuando sí se activa a continuación:

Cuando se activa la mutación, el cromosoma se desordena y un valor será intercambiado entre todos los números de la tupla:

```
Tupla mezclada: [1, 3, 5, 0, 4, 2]

Valores antes de la mutación:
[947291, 467396, 3386111, 389788, 600845, 1150590]

Tupla [1, 3, 5, 0, 4, 2]
min_value = 389788
addition = 33411

Valores después de la mutación:
[980702, 433985, 3419522, 423199, 567434, 1117179]
```

Una vez que terminan estos procedimientos de ejecutarse, la nueva generación cambia de estructura para repetir y volver a heredar las veces que se hayan indicado.

El programa es muy inestable en sus ejecuciones, sin embargo en los mejores retornos se realizaron promedios y cálculos de estos resultados, obteniendo como valores los siguientes para realizar el balanceo de carga:

- PC de Melissa: 1,752,929.
- PC de Joel: 1,097,725.
- PC de Óscar: 2,188,458.
- PC de Dariana: 179,704.
- PC de Fernando: 1,013,759.
- PC de Hiram: 709,446.

Observemos que tiene sentido, ya que la PC de Óscar es la más veloz, por ello se le asigna mayor carga a él, caso contrario a la PC de Dariana.

Conclusiones

Andrea Melissa Almeida Ortega:

Como se pudo observar ya en la práctica de algoritmo genético simple enfocada en nuestro proyecto, nos damos cuenta que la optimización es algo esencial para el software y no es simplemente priorizar que realice las acciones más rápido, si no de entender lo que está haciendo el programa para poder repartir las distintas tareas y acciones para que se puedan ejecutar de la manera más eficaz posible.

Joel Alejandro Espinoza Sánchez:

La implementación y uso del algoritmo genético será muy importante para el proyecto de la materia y más importante con un planificador de carga que evite los fallos a lo largo de su ejecución. Con este programa se puede planificar que el conjunto de computadoras que estamos utilizando obtenga ventajas al hacer la carga de tweets dependiendo el procesador.

Óscar Alonso Flores Fernández:

Al poder elaborar un algoritmo genético basándonos en lo que hemos estado elaborando podemos tener un esquema de trabajo mucho más directo y eficiente para poder llevar a cabo las distintas tareas, buscando siempre el poder cumplir con lo que el AG nos dicta y así optimizar el trabajo, la división de tareas, etc.

Dariana Gómez Garza:

En la elaboración de esta práctica implementamos para otro fin el algoritmo genético y otras herramientas útiles que nos ayudaron a observar las funciones de nuestro proyecto que tenemos en marcha de esta materia y lo útil que es para mejorar otros algoritmos. Las clases que tuvimos nos dieron una idea mejor de cómo realizar esta investigación. También nos dimos cuenta de cómo podríamos aplicar esto y que fuera funcional al mismo tiempo para nuestro proyecto en curso.

Fernando Francisco González Arenas:

Esta distribución de tweets permite utilidades en las que podemos agilizar el trabajo de las computadoras fácilmente. En este trabajo podemos observar cómo nuestras computadoras a partir de un programa en Python y un planificador de carga funcional podemos tener nuestras computadoras trabajando de manera acorde a su comportamiento.

Hiram Efraín Orocio García:

Un algoritmo genético es útil al momento de buscar soluciones por medio de exploración, así que aprender a implementar uno así sea en una aplicación básica, en un futuro lo podríamos implementar en problemas mayores que serán de gran ayuda.

Referencias bibliográficas

- C. Darwin (1859). *On the Origin of Species by Means of Natural Selection*, Murray, London.
- C. Reeves (1993). *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications.
- D. E. Goldberg, Jr. R. Lingle (1985). *Alleles, loci and the traveling salesman problem*, en Proceedings of the First International Conference on Genetic Algorithms and Their Applications,154-159.
- Z. Michalewicz (1992). *Genetic Algorithms + Data Structures = Evolution Programs*, SpringerVerlag, Berlin Heidelberg.
- Brad Miller, David Ranum. (2006). *Problem solving with algorithms and data structures using python*, Fraklin, Beedle & Associates.
- Albert Y. Zomaya, Senior Member, IEEE, and Yee-Hwei Teh. (SEPTEMBER 2001). *Observations on Using Genetic Algorithms for Dynamic Load-Balancing*. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 12, 13. Octubre 10, 2021.

Anexos

Anexo 1: Código con complejidad computacional lineal estándar en Python:

```
import time

i = 0

inicio = time.time()
while i < 1000000:
    i = i + 0.01
final = time.time()
intervalo = final - inicio

print("Tiempo de ejecución: " + str(intervalo))
```

Anexo 2: Código del algoritmo genético en Python:

```
#%% Portada
'''
                    Centro de Ciencias Básicas
            Departamento de Ciencias de la Computación
                        Metaheurísticas I
                             7° “A”

        Actividad 2.04: Simulación de Algoritmo Genético

             Profesor: Francisco Javier Luna Rosas

            Alumnos:
                     Almeida Ortega Andrea Melissa
                     Espinoza Sánchez Joel Alejandro
                     Flores Fernández Óscar Alonso
                     Gómez Garza Dariana
                     González Arenas Fernando Francisco
                     Orocio García Hiram Efraín

Fecha de Entrega: Aguascalientes, Ags., 11 de octubre de 2021
'''

#%% Descripción

#%% Importación de librerías
import numpy as np
import random

#%% Funciones de apoyo
# Función FirstSample
def FirstSample(qe,qi,total_tweets):
    sample = []
    for i in range(qi):
        sample.append([])
        remaining_tweets = total_tweets
        for j in range(qe):
            sample[i].append(random.randint(0, remaining_tweets))
            remaining_tweets = remaining_tweets - sample[i][j]
        if remaining_tweets != 0:
            assignment = random.randint(1,6)
            sample[i][assignment - 1] = sample[i][assignment - 1] + remaining_tweets
    return sample

# Función PrintSample
def PrintSample(qi,sample):
    for i in range(qi):
        try:
            print(str(sample[0][i]) + " = " + str(sum(sample[0][i])) + " \t\t " +
str(sample[1][i]) + " \t " + str(sample[2][i]))
        except:
            try:
                print(str(sample[0][i]) + " = " + str(sum(sample[0][i])) + " \t\t " +
str(sample[1][i]) + " \t " + str(sample[2]))
            except:
                print(str(sample[0][i]) + " = " + str(sum(sample[0][i])) + " \t\t " +
str(sample[1]) + " \t " + str(sample[2]))
    return
```

```
# Función evaluate
def evaluate(qi,sample,power_process):
    evaluation = []
    for i in range(qi):
        function = -(np.abs(sample[0][i][0] - 1752929) + np.abs(sample[0][i][1] - 1752929)
+   np.abs(sample[0][i][2]   -   2188458)   +   np.abs(sample[0][i][3]   -   179704)   +
np.abs(sample[0][i][4] - 1013759) + np.abs(sample[0][i][5] - 709446)) + 34710105

        print("Función objetivo " + str(i))
        print(function)

        evaluation.append(function)
    return evaluation

# Función evaluateA
def evaluateA(qi,sample):
    evaluation = []
    evaluation.append(sample[1][0])
    i = 1
    while i < qi:
        evaluation.append(evaluation[i - 1] + sample[1][i])
        i = i + 1
    return evaluation

# Función Sorting
def Sorting(sample, qi):
    i = 0
    while i < qi - 1:
        if sample[1][i] < sample[1][i + 1]:
            for j in range(3):
                aux = sample[j][i]
                sample[j][i] = sample[j][i + 1]
                sample[j][i + 1] = aux
            i = 0
            continue
        i = i + 1
    return sample

# Función nullNewSample
def newSampleCreator(qi):
    nullCreator = []
    for i in range(qi):
        nullCreator.append(0)
    nullNewSample = []
    nullNewSample.append(nullCreator[:])
    nullNewSample.append(nullCreator[:])
    nullNewSample.append(nullCreator[:])
    return nullNewSample

#%% Calibrar variables
total_tweets = 6942021
qe = 6 # Inamovible
qi = 60 # Movible
pc = 75 # Movible
pm = 40 # Movible
qelitism = 5 # Movible
qg = 10
sample = [[], [], []]
power_process = [9.311, 14.8685, 7.458, 90.8248, 16.1, 23.006] # Orden alfabético: Melissa,
Joel, Óscar, Dariana, Fernando, Hiram
ig = 0
```

```
#%% AG: Creación de la población aleatoria
sample[0] = FirstSample(qe,qi,total_tweets)
PrintSample(qi, sample)

#%% AG: Procedimiento de cada generación
while ig < qg:
    print("Población " + str(ig))
    print("")

    # Evaluamos a todas las muestras
    sample[1] = evaluate(qi,sample,power_process)

    print("Población " + str(ig) + " al ser evaluada")
    PrintSample(qi,sample)
    print("")

    # Evaluamos la suma acumulada
    sample[2] = evaluateA(qi,sample)

    print("Población " + str(ig) + " al ser evaluada con acumulación")
    PrintSample(qi,sample)
    print("")

    # Ordenamos por valor de f(x) es decir, por sample[1]
    sample = Sorting(sample, qi)

    print("Población " + str(ig) + " al ser ordenada")
    PrintSample(qi,sample)
    print("")

    # Reevaluamos la suma acumulada
    sample[2] = evaluateA(qi,sample)

    print("Población " + str(ig) + " al ser reevaluada")
    PrintSample(qi,sample)
    print("")

    # Creamos una nueva muestra vacía
    newSample = newSampleCreator(qi)

    print("Nueva muestra " + str(ig + 1) + " vacía")
    print(newSample)
    print("")

    print("Se aplicará elitismo")
    x = input()

    # Se aplica elitismo
    ii = 0
    while ii < qelitism:
        newSample[0][ii] = sample[0][ii]
        #newSample[1][ii] = sample[1][ii]
        #newSample[2][ii] = sample[2][ii]
        ii = ii + 1

    print("Nueva muestra " + str(ig + 1) + " después de elitismo")
    for i in range(qi):
        try:
            print(str(newSample[0][i]) + " = " + str(sum(newSample[0][i])))
        except:
```

```
            pass
    print("")
    print("Se aplicará cruzamiento")
    x = input()

    # Se aplicará cruzamiento
    while ii < qi:
        # Seleccionamos por ruleta
        # Padre 1
        random_pick = random.uniform(0, sample[2][qi - 1])

        for ic in range(qi):
            if random_pick < sample[2][ic]:
                c1 = sample[0][ic]
                print("Se elige como padre 1 el individuo " + str(ic) + "(random_pick = "
+ str(random_pick) + ")")
                break

        # Padre 2
        random_pick = random.uniform(0, sample[2][qi - 1])

        for ic in range(qi):
            if random_pick < sample[2][ic]:
                c2 = sample[0][ic]
                print("Se elige como padre 2 el individuo " + str(ic) + "(random_pick = "
+ str(random_pick) + ")")
                break

        # Se evalúa si se aplicará cruzamiento
        random_num = random.randint(0,99)
        if random_num < pc:
            print("Los cromosomas " + str(ii) + " y " + str(ii + 1) + " han activado
cruzamiento (Probabilidad: " + str(random_num) + " < " + str(pc) + ")")
            print("")

            # Mezclamos los 6 números de una tupla
            processors = [0, 1, 2, 3, 4, 5]
            random.shuffle(processors) # Se mezclarán y se harán sumas balanceadas de par
en par

            print("Tupla mezclada: " + str(processors))
            print("")

            print("Valores antes del cruzamiento: ")
            print(c1)
            print(c2)
            print("")

            for i in [0, 2, 4]:
                # Sumas equilibradas
                min_value    =    min(c1[processors[i]],    c1[processors[i    +    1]],
c2[processors[i]], c2[processors[i + 1]])
                operation = random.randint(0, 1) # Elige la operación con la que empezará
la suma equilibrada
                addition = random.randint(0, min_value)

                print("Tuplas " + str(processors[i] + 1) + " y " + str(processors[i + 1] +
1))
                print("min_value = " + str(min_value))
                print("addition = " + str(addition))
                print("")
```

```
                if operation == 0:
                    c2[processors[i + 1]] = c2[processors[i + 1]] - addition
                    c1[processors[i + 1]] = c1[processors[i + 1]] + addition
                    c1[processors[i]] = c1[processors[i]] - addition
                    c2[processors[i]] = c2[processors[i]] + addition
                else:
                    c2[processors[i + 1]] = c2[processors[i + 1]] + addition
                    c1[processors[i + 1]] = c1[processors[i + 1]] - addition
                    c1[processors[i]] = c1[processors[i]] + addition
                    c2[processors[i]] = c2[processors[i]] - addition

            print("Valores después del cruzamiento: ")
            print(c1)
            print(c2)
            print("")

        else:
            print("Los cromosomas " + str(ii) + " y " + str(ii + 1) + " no han activado
cruzamiento (Probabilidad: !" + str(random_num) + " < " + str(pc) + "!)")

        newSample[0][ii] = c1
        try:
            newSample[0][ii + 1] = c2
        except:
            pass

        ii = ii + 2

    print("Nueva muestra " + str(ig + 1) + " después del cruzamiento")
    for i in range(qi):
        try:
            print(str(newSample[0][i]) + " = " + str(sum(newSample[0][i])))
        except:
            pass
    print("")
    print("Se aplicará mutación")
    x = input()

    # Se aplica mutación
    ii = 0
    while ii < qi:
        # Se evalúa si se aplicará mutación
        random_num = random.randint(0,99)
        if random_num < pm:
            print("El cromosoma " + str(ii) + " ha activado mutación (Probabilidad: " +
str(random_num) + " < " + str(pm) + ")")
            print("")

            processors = [0, 1, 2, 3, 4, 5]
            random.shuffle(processors) # Se mezclarán y se hará una sumas balanceadas
cíclica

            print("Tupla mezclada: " + str(processors))
            print("")

            print("Valores antes de la mutación: ")
            print(newSample[0][ii])
            print("")

            min_value = min(newSample[0][ii])
```

```
            operation = random.randint(0, 1) # Elige la operación con la que empezará la
suma equilibrada
            addition = random.randint(0, min_value)

            print("Tupla " + str(processors))
            print("min_value = " + str(min_value))
            print("addition = " + str(addition))
            print("")

            if operation == 0:
                newSample[0][ii][processors[0]]   =   newSample[0][ii][processors[0]]   -
addition
                newSample[0][ii][processors[1]]   =   newSample[0][ii][processors[1]]   +
addition
                newSample[0][ii][processors[2]]   =   newSample[0][ii][processors[2]]   -
addition
                newSample[0][ii][processors[3]]   =   newSample[0][ii][processors[3]]   +
addition
                newSample[0][ii][processors[4]]   =   newSample[0][ii][processors[4]]   -
addition
                newSample[0][ii][processors[5]]   =   newSample[0][ii][processors[5]]   +
addition
            else:
                newSample[0][ii][processors[0]]   =   newSample[0][ii][processors[0]]   +
addition
                newSample[0][ii][processors[1]]   =   newSample[0][ii][processors[1]]   -
addition
                newSample[0][ii][processors[2]]   =   newSample[0][ii][processors[2]]   +
addition
                newSample[0][ii][processors[3]]   =   newSample[0][ii][processors[3]]   -
addition
                newSample[0][ii][processors[4]]   =   newSample[0][ii][processors[4]]   +
addition
                newSample[0][ii][processors[5]]   =   newSample[0][ii][processors[5]]   -
addition

            print("Valores después de la mutación: ")
            print(newSample[0][ii])
            print("")

        else:
            print("El cromosoma " + str(ii) + " no ha activado mutación (Probabilidad: !"
+ str(random_num) + " < " + str(pm) + "!)")
            print("")
        ii = ii + 1

    print("Nueva muestra " + str(ig + 1) + " después de la mutación")
    for i in range(qi):
        try:
            print(str(newSample[0][i]) + " = " + str(sum(newSample[0][i])))
        except:
            pass
    print("")
    print("Se tratará una nueva generación")
    x = input()

    sample = newSample
    ig = ig + 1
```