Análisis de Algoritmos: Similitud de Imagenes

Nicolás Feoli[1] y Sebastian Salas[2]

***Abstract*— Este artículo se describe la implementación de un algoritmo genético de creación de imágenes en el lenguaje de programación Python versión 2.7. Se utilizaron tres algoritmos de similitud de imágenes los cuales son descritos y analizados. Se incluye en el documento el resultado y el análisis de 12 experimentos con distintas imágenes tanto a color como en escala de grises y su efectividad con los distintos algoritmos implementados.**

I. INTRODUCCIÓN

Desde la creación de las imágenes digitales hasta la actualidad, se ha venido revolucionando su calidad y usos. Estas están presentes en la vida cotidiana de las personas. Una imagen o desde un punto de vista más computacional un arreglo bidimensional de pixeles pueden ser generadas desde imágenes aleatorias mediante algoritmos genéticos.

Estos algoritmos, implementados especialmente en la inteligencia artificial, son inspirados en la evolución biológica y su base genético-molecular. Estos funcionan entre el conjunto de soluciones de un problema llamado fenotipo, y el conjunto de individuos de una población natural, codificando la información de cada solución en una cadena, generalmente binaria, llamada cromosoma. Los símbolos que forman la cadena son llamados los genes. Cuando la representación de los cromosomas se hace con cadenas de dígitos binarios se le conoce como genotipo.

Los cromosomas evolucionan a través de iteraciones, llamadas generaciones. En cada generación, los cromosomas son evaluados usando alguna medida de aptitud. Las siguientes generaciones (nuevos cromosomas), son generadas aplicando los operadores genéticos repetidamente, siendo estos los operadores de selección, cruzamiento, mutación y reemplazo.

Dentro de las funciones de adaptabilidad se utilizaron los siguientes algoritmos: distancia euclideana, mean square error y la creación de uno propio. En cuanto a la distancia euclideana se usa para comparar dos vectores que en este caso serán de pixeles y el error cuadrático medio mide el promedio de los errores al cuadrado, es decir, la diferencia entre el estimador y lo que se estima. El algoritmo propio se basa en que si el pixel x es diferente al mismo pixel de la imagen meta, a ese se le suma uno para generar una nueva población.

II. COMPLEJIDAD ALGORÍTMICA

A. *Análisis Formal del Algoritmo Propio*

Se asignará una unidad de tiempo a las declaraciones, asignaciones, operaciones simples $(+, -, *, /)$. A continuación, el fragmento de código del algoritmo.

```
def calcularAdaptabilidad (individual):
    fitness = 0
    for i in range(len(individual)):
        if individual[i] == listaGenesMeta[i]:
            fitness += 1
    return fitness
```

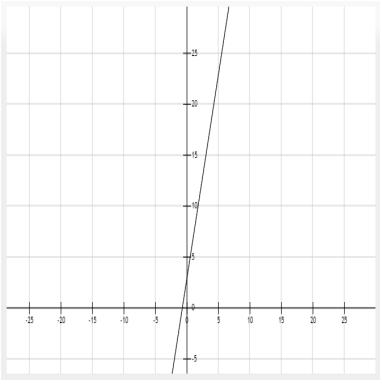
Observe que primero se tiene una inicialización, lo que nos da 1 unidad de tiempo. Luego se tiene un ciclo for que ejecuta un aumento $n + 1$ veces (n por la cantidad de elementos que tenga la lista de pixeles de la imagen (tamaño) lo cual equivale a $(n + 1)$ unidades.

Dentro del for se tienen 2 unidades de tiempo aproximadamente por la comparación que se tiene y el retorno interno del for. Esto se realiza n veces, por lo que la complejidad de esta sección ser $2 * n$.

En el mismo for tenemos una estructura if que tiene un incremento que cuenta por una unidad de tiempo 1n. Por último, un retorno, al cual se le puede asignar una unidad de tiempo.

Si unimos los datos nos queda una complejidad total aproximada del algoritmo de:

$$1 + (n + 1) + 2n + 1n + 1$$
$$= n + 1 + 3n + 2$$
$$= 4n + 3$$



Nota: Otra manera de visualizarlo sería $Cn + C$ donde C equivale a una constante de tiempo que requiere el proceso.

Dado que se desea obtener la complejidad, se pueden eliminar los valores que no crecerán ms que n y la constante que la multiplica. Lo que nos da $O(n)$.

*Estudiantes del Instituto Tecnolgico de Costa Rica

B. Análisis Formal del Algoritmo MSE

De igual manera se asignará una unidad de tiempo a las declaraciones, asignaciones, operaciones simples. A continuación, el fragmento de código del algoritmo error cuadrático medio.

```
def mse(imageA):
    for i in range(len(imagenA)):
        errorCM += (np.array(imageA) - np.array
(listaGenesMeta)) ** 2)
    errorCM /= float(tamanoPoblacion)
    return errorCM
```

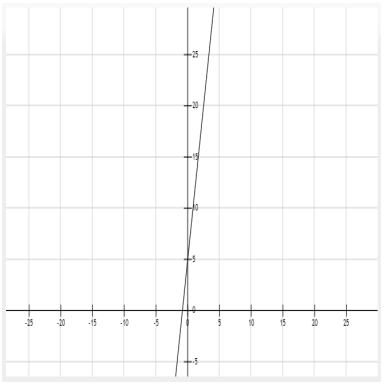
El algoritmo inicia con un ciclo for que ejecuta un aumento $n + 1$ veces (n por la cantidad de elementos que tenga la lista de pixeles de la imagen (tamaño) lo cual equivale a $n + 1$ unidades de tiempo. Dentro de esta estructura, hay una asignación, dos conversiones de lista, una resta y elevado potencial lo cual cuentan como 5 unidades de tiempo extra por el total de ciclos n.

Al final del algoritmo se tiene una asignación, operación simple y una conversión de tipo, lo que nos da 3 unidades de tiempo y además, un retorno, al cual de igual manera cuenta como una una unidad de tiempo más $(+4)$.

Si unimos los datos nos queda una complejidad total aproximada del algoritmo de:

$$(n+1)+5n+4=$$
$$n+1+5n+4=$$
$$6n+5$$

Dado que se desea obtener la complejidad,



como vimos en el ejemplo anterior podemos eliminar los valores que no crecerán más que n y la constante que la multiplica. Lo que nos da $O(n)$.

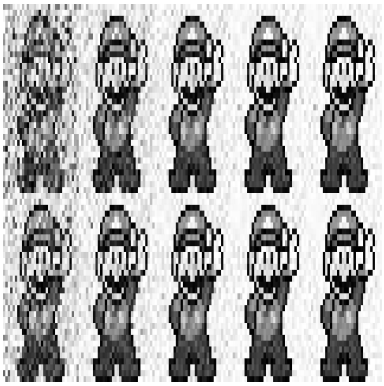
III. METODOLOGA

La investigación contó una serie de pruebas que nos permitieran comparar los tres diferentes algoritmos de adaptabilidad que buscarán la similitud entre dos imágenes.

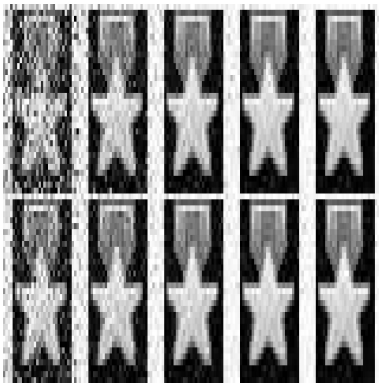
Prueba	Algoritmo	Color	Dimensiones	Generation	Imagen
Casa 1	MSE	L	32 X 32	20.000	Luigi.jpg
Caso 2	MSE	RGB	32 X 32	20.000	Luigi.jpg
Caso 2.2	MSE	RGB	32 X 32	10.000	Luigi.jpg
Caso 3	MSE	RGB	50 X 50	20.000	Zelda.jpg
Caso 3.1	MSE	L	50 X 50	10.000	Zelda.jpg
Caso 4	Euclideana	L	32 X 32	20.000	Zelda.jpg
Caso 5	Euclideana	L	32 X 32	10.000	Estrella.jpg
Caso 5.1	Euclideana	RGB	32 X 32	20.000	Estrella.jpg
Caso 6	Propia	RGB	32 X 32	20.000	Estrella.jpg
Caso 7	Propia	RGB	32 X 32	20.000	Print.jpg
Caso 8	Propia	L	32 X 32	20.000	Print.jpg
Caso 8.1	Propia	L	32 X 32	10.000	Print.jpg

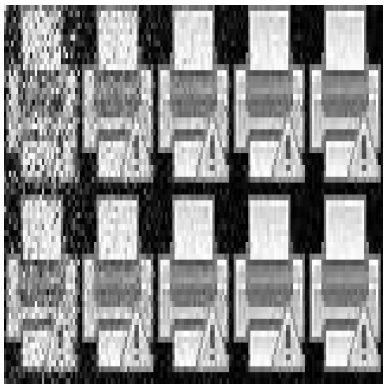
La captura de datos se realizó positivamente; para efectos demostrativos mostraremos tres casos bajo las mismas condiciones para poderlas compararlas. A continuación, se mostrarán tres resultados donde vendrá la imagen de evolución de generaciones (se lee de arriba hacia abajo, de izquierda a derecha), la primer gráfica contiene el comportamiento de los pixeles de cuatro generaciones $(2000, 5000, 10000, 20000)$ y la segunda gráfica se compara la mejor generación (20000) contra los pixeles de la imagen meta.

A. Resultado - Caso 1 (MSE)



B. Resultado - Caso 5 (Euclidiana)





C. *Resultado - Caso 8 (Propia)*

IV. DISCUSIONES

Como se muestra en los resultados, al procesar las imágenes se dieron evoluciones positivas obtenido la imagen meta casi a un 90% aunque no se llegó a converger en su totalidad. Nuestro experimento se basó en manejar entre 20.000 y 10.000 generaciones, las cuales, las primeras a como fue de esperar, fueron bastantes malas y denotan que nuestros algoritmos serían ineficientes para pocas generaciones como por ejemplo 1000 ya que son muchos pixeles que se deben ajustar.

Desde el punto de vista teórico podemos afirmar que bajo nuestro anlisis de que la euclidiana es $O(mxn)$, nuestro propio algoritmo es $O(n)$ y Mean Square Error - MSE es $O(n)$ se recomienda cualquier algoritmo para aplicar poblaciones pequenas.

Desde el punto de vista práctico si esperamos una respuesta satisfactoria no es recomendable usar estos algoritmos para imágenes de más $100x100$ pixeles ya que las cantidades de comparaciones que se realizan para computadores con procesadores estándar requieren de mucho tiempo para poder llegar a converger una imagen. Para problemas de alta complejidad las funciones de evaluación pueden tornarse demasiado costosas en términos de tiempo y recursos.

Visualizando las gráficas suministradas por el programa como se había pensado en la hipótesis, conforme van pasando las generaciones, las líneas de pixeles se van a cercando a la imagen meta al punto que se llegan a parecer satisfactoriamente, notándose que las poblaciones se fueron adaptando y los genes menos aptos se fueran descartados.

De igual manera, ejecutamos el programa con imágenes a color(RGB) y el comportamiento fue bastante ineficiente durante todo el proceso, sin embargo se llegó a converger lo suficiente como para asemejarse con la imagen meta validando de que nuestras funciones de adaptabilidad estaban cumpliendo con su tarea.

V. BIBLIOGRAFÍA

Demiriz, Ayhan. (1999). Semi-Supervised Clustering Using Genetic Algorithms. http://s3.amazonaws.com/academia.edu.documents/40354638/Semi-SupervisedClustering UsingGenetic20151124-29087-s0rvnf.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3AExpires=1489283802Signature=dH5rD%2BmS%2BoAbi66SclLa3HrVh4c%3Dresponse-content-disposition=inline%3B%20filename%3DSemi-supervisedclusteringusinggenetic.pdf. Rivera-G.(2006). A GENETIC ALGORITHM FOR SOLVING THE EUCLIDEAN DISTANCE MATRICES COMPLETION PROBLEM. http://slapper.apam.columbia.edu/bib/papers/river_b_99.pdf

Annimo. (2014). How-To: Python Compare Two Images http://www.pyimagesearch.com/2014/09/15/python-compare-two-images/ Zhou Wang.(2004). IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 13, NO. Image Quality Assessment: From Error Visibility to Structural Similarity. http://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf