Compresión de música con pérdida utilizando algoritmos evolutivos

Santiago Herrera
Facultad de Ingeniería
Universidad de la República
santiago.herrera.di@fing.edu.uy

Gabriel Kryger
Facultad de Ingeniería
Universidad de la República
gabriel.kryger@fing.edu.uy

***Resumen*—En el siguiente documento vamos a tratar el problema de la compresión de música con perdida (CMP) que definimos de la siguiente manera: reducir el tamaño de un archivo de música intentando mantener su calidad original lo más posible.**

I. Introducción

En este documento se sentarán las bases del problema CMP, se planteará una estrategia basada en un Algoritmo Evolutivo para resolver dicho problema. Para dicha estrategia se detallarán los operadores utilizados, el razonamiento de por qué se utilizan dichos operadores y se investigará cuáles son los parámetros de dichos operadores con los cuales el algoritmo logra un mejor desarrollo. También se comparará dicho algoritmo con MP3 para así poder ver realmente que tan buenas son las soluciones generadas.

II. Descripción del problema

Una canción puede ser definida como un conjunto de ondas continuas que poseen una amplitud y frecuencia que va cambiando a lo largo del tiempo (ver Figura 1). Esta onda puede ser discretizada, y a cada elemento se le llama "muestra".
Definimos al problema CMP como el siguiente: basándonos en una canción elegida, intentar generar métodos de recuperación de dicha canción que al ser representados ocupen menos memoria que la canción original.
Dicho método de generación no debe de ser perfecto, es decir, la canción generada y la canción elegida no tienen por qué ser iguales, mas se busca minimizar la diferencia entre dichas canciones.

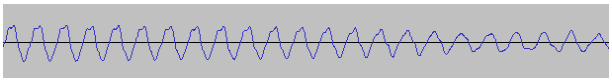

Figura 1. Onda de una canción

III. Estrategia de resolución

El método propuesto por nosotros consiste en partir la canción (cuyas muestras son un conjunto P) en N subconjuntos de muestras, y a cada una de estas particiones asociarle un conjunto de K coeficientes (reales), que una vez administrados a una función F aproximen a las muestras asociadas.
La memoria ocupada por una solución de esta forma seria $N * K * T + L$, siendo T el espacio en bytes ocupado por cada uno de los coeficientes, y L el largo de un mapa de bits que indica donde acaba cada partición.
Por lo tanto, deberíamos intentar aproximarnos a las muestras originales mientras minimizamos la cantidad de particiones usadas, la cantidad de coeficientes en cada una y la cantidad de memoria usada para almacenar cada coeficiente.
Antes de continuar consideramos importante definir un poco de notación para hacer la lectura más entendible:

- P_x : Partición x.
- $P(i)$: Muestra en la posición i-ésima, es decir, desde el inicio de la partición el elemento i.
- $P_x(i)$: Muestra en la posición i-ésima de la partición x.
- $F(P_x(i))$: Muestra generada por la solución, en la posición i de la partición x, es decir, de las muestras generadas por la función F de la de la solución desde el inicio de la partición x, el elemento i.
- $K_{j,i}$: Coeficiente i-ésimo de la partición j.

Estas particiones deben cumplir algunas propiedades:

- deben ser contiguas: $P_x = \{P(i), P(i+1), ..., P(j-1), P(j)\}$
- deben ser disjuntas: $\forall P_i, P_j, i \neq j, P_i \cap P_j = \emptyset$
- su unión debe formar *al menos* la canción original: $P \subseteq \cup_{i=1}^{N} P_i$

Nuestro objetivo con el algoritmo será minimizar la diferencia entre nuestra solución y el sonido objetivo, intentando minimizar también la diferencia máxima [2]. Por esto definimos que el fitness de nuestra solución sea la sumatoria del cuadrado de las diferencias, es decir la distancia vectorial de nuestra solución al objetivo:

- $Fitness = \sqrt{\Sigma\ (P(i) - F(P(i)))^2}$

Notamos que esta diferencia no es el único factor a considerar, ya que una solución la cual aproxime correctamente una sección de la muestra, mas no aproxime ninguna otra parte de la muestra podría llegar a tener menor, a estas instancias las consideramos soluciones inválidas, puesto que nuestro objetivo es aproximar toda la canción.

III-A. Operadores evolutivos

III-A1. Operador de mutación: El operador de mutación propuesto es que los coeficientes puedan aumentar o reducirse

hasta un tamaño V_m, es decir que exista una ventana de tamaño $2V_m$ en la cual cuando un coeficiente mute se le agregue o reste un valor entre 0 y V_m, a esta forma de mutación la denominaremos WindowMutation o Mutación por Ventana (WM).

También pensamos variar la cantidad de muestras que cada partición debe aproximar (el largo de cada partición) utilizando WM y agregando la restricción de que la suma del largo de cada una de las particiones debe de ser igual a la cantidad de muestras a aproximar. De esta forma, utilizando la misma cantidad de particiones N se podría llegar a mejores soluciones ya que la función F se podría adaptar mejor a los nuevos largos (por ejemplo, silencio o ruido harmónico).

Se idearon los operadores de mutación de ventana porque el explorar toda la ventana de soluciones cada vez que se realiza una mutación es ineficiente.
Esta mutación logra una buena exploración de la solución sin dar "saltos demasiado grandes", lo cual, gracias a que la música es una función continua, es lo que estamos buscando.

Algorithm 1 Operador de Mutación

Require: $P, p_m > 0, V_m > 0, C > 0$
  for cada particion P_i en P **do**
    si $|P_i| == 1$ se considera $p_m * 5$ temporalmente
    $n = i + random(0,\ 1)\ ?\ 1\ :\ -1$ ▷ se acota n entre 0 y $|P|$
    if $random() < p_m$ **then**
      $\Delta = min\{random(0, V_m),\ largo\ de\ P_n\ -\ 1\}$
      largo de $P_i += \Delta$
      largo de $P_n -= \Delta$
    end if
    while $random() < p_m$ **do**
      $j = random(0,\ K\ -\ 1)$
      $m = (random(0,\ 1)\ ==\ 0)\ ?\ 0{,}01\ :\ 1$
      $K_{i,j} += random(-C, C) * m$ ▷ cambia por [-C, C] o [-0.01C, 0.01C]
    end while
  end for

III-A2. Operador de cruzamiento: Planteamos utilizar un operador de cruzamiento el cual hace lo siguiente: toma dos padres, para luego elegir un punto al azar entre 0 y $|P|$, para tomar las particiones de cada padre que genera la muestra en el punto e intercambiar/promediar (aleatoriamente) el largo y los coeficientes de las particiones.
Con este operador se logran generar nuevas soluciones las cuales pueden tomar características positivas de los padres sin alterar de gran manera las demás particiones de dichas soluciones, lo cual es algo deseable en nuestro caso.
A este operador le llamamos CurvePointCrossover.

III-A3. Operador de selección: Se utilizó el BinaryTournamentSelection de la librería JMetal como operador de selección.
Se utiliza este operador de selección porque es un seleccionador que ha probado su eficacia en llegar a una convergencia y mantener una diversidad genética amplia.

Algorithm 2 Operador de Cruzamiento

Require: $P, p_0, p_1, p_c \geq 0, V_c > 0, C > 0$ ▷ p_0 y p_1 son los padres
  $i = random(0, |P|)$ ▷ toma un una posición al azar
  $p_A :=$ primera partición de p_0 que cubre $P(i)$
  $p_B :=$ primera partición de p_1 que cubre $P(i)$
  $K_A :=$ coeficientes de p_A
  $K_B :=$ coeficientes de p_B
  for j = 0, j ¡K, j++ **do**
    if $random() < p_c$ **then**
      if $random() < 0{,}5$ **then**
        $K_{A,j} = \frac{K_{A,j}+K_{B,j}}{2}$
      end if
      if $random() < 0{,}5$ **then**
        $K_{B,j} = \frac{K_{A,j}+K_{B,j}}{2}$
      end if
      if $random() < 0{,}5$ **then**
        $K_{A,j} = K_{B,j}$
      end if
      if $random() < 0{,}5$ **then**
        $K_{B,j} = K_{A,j}$
      end if
    end if
  end for

III-A4. Inicialización de soluciones: La inicialización se hace de manera aleatoria dentro de los parámetros de la solución, es decir:

- La suma del largo de todas las particiones de la solución es igual al tamaño del tramo a aproximar.
- Un coeficiente no puede exceder (en la inicialización) la amplitud máxima del tramo.

III-A5. Poblaciones distribuidas: Además, para aprovechar mejor la potencia computacional y mantener de mejor manera la diversidad genética se decidió implementar un modelo de poblaciones distribuidas, todas las subpoblaciones teniendo las mismas configuraciones paramétricas.

Algorithm 3 Algoritmo Evolutivo

Require: $maxPopulation > 0, maxEval > 0$
  $population :=$ lista de $maxPopulation$ soluciones creadas al azar
  $evaluations := 0$
  while $evaluations < maxEval$ **do**
    Aplicar operador de mutación sobre $population$
    Aplicar operador de cruzamiento sobre $population$
    Evaluar todas las soluciones de $population$
    $evaluations += maxPopulation$
    Aplicar operador de selección sobre $population$
  end while

Figura 2. Referencia de Tramo, Partición y Muestra

III-B. Instancias de problemas

Para poder lograr un buen compromiso entre el tamaño de nuestra solución y la precisión deseada, se decidió trabajar

siempre con números de 16 bits con signo. Las muestras están en un rango [-32768, 32767] (entero), mientras que nuestros coeficientes son representados usando una coma fija de formato 1.11.4, logrando así representar cualquier número en [-2048, 2047], con una precisión de hasta 1/16. De esta forma, nuestro T va a ser 2. También trabajamos con un único canal de audio (monoaural) y una tasa de muestreo de 8KHz.

El formato de entrada de cada instancia es un archivo PCM S16LE (cada 2 bytes representan un número de 16bits con signo en *little-endian*), y los mismos son obtenidos usando el programa *ffmpeg* [3] y el siguiente comando:

```
$ ffmpeg -i entrada.mp3 -ac 1 -ar 8000 \
  -acodec pcm_s16le -f s16le salida.pcm
```

A su vez, nuestra solución generada debe poder ser convertida a un archivo PCM con el mismo formato, para luego ser recodificada como MP3 (u otro formato como OGG, FLAC, etc) con el siguiente comando:

```
$ ffmpeg -ac 1 -ar 8000 -acodec \
  pcm_s16le -f s16le -i entrada.pcm \
  salida.mp3
```

IV. Evaluación experimental

Toda la evaluación experimental se realizó sobre la librería JMetal Para Java [4]

IV-A. Función de aproximación (F)

Se realizó una exploración de posibles funciones de aproximación de forma experimental, y se observó rápidamente que las funciones que contenían funciones trigonométricas en su interior lograban aproximar mejor la solución que aquellas que no.
Siendo más específicos, las funciones las cuales tenían una constante multiplicada por una función trigonométrica lograban fácilmente aproximar ondas básicas, lo cual está amparado por el análisis de música de Fourier [5] – al fin y al cabo, a la música se la puede interpretar como una onda muy compleja.
También se observó, experimentalmente y de acuerdo a la teoría, que al sumar dos funciones de esta familia se lograba obtener funciones que aproximaban ondas más complejas.
Con bases en este descubrimiento y gracias a la necesidad de reducir la cantidad de coeficientes se decidió optar por la siguiente función:

$$F(P_j(x)) = K_{j,0} + K_{j,1} * sin(K_{j,2} * x) + K_{j,2} * cos(K_{j,1} * x) + K_{j,1} * sin(K_{j,0} * x)$$

siendo x la distancia entre el inicio de la partición y la posición de la muestra a calcular y $K_{j,i}$ los coeficientes de la partición j.

IV-B. WM para coeficientes (K):

Se decidió para elegir la ventana de mutación de los coeficientes hacerlo proporcionalmente a la amplitud máxima ($max\ \{|P(i)|\}$).
Se observa que mientras la ventana de mutación esté entre un 25 % y un 65 % de la amplitud máxima el algoritmo logra converger eficazmente [6]. Resultados promedios de

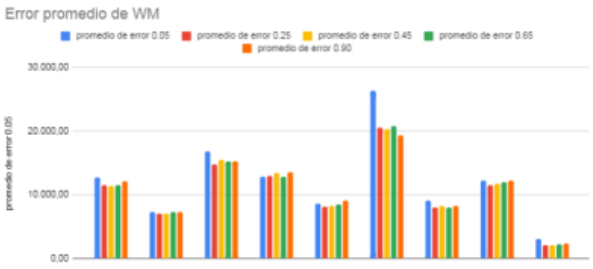

Figura 3. Error promedio alterando los parametros del WM para coeficientes con diferentes tramos

ejecuciones

- 5 % : 12.093,52
- 25 %: 10.717,84
- 45 %: 10.873,98
- 65 %: 10.913,34
- 95 %: 11.034,53

IV-C. WM para cantidad de muestras por partición (Largo):

Se eligió, de igual manera, para la ventana de mutación del largo de las particiones hacerlo proporcionalmente a la cantidad de muestras y la cantidad de particiones (específicamente dividiendo ambas constantes, a esto le llamaremos largo promedio).

Se observa que mientras la ventana de mutación esté entre un 5 % y un 65 % del largo promedio algoritmo logra converger eficazmente [7]. Resultados promedios de ejecuciones

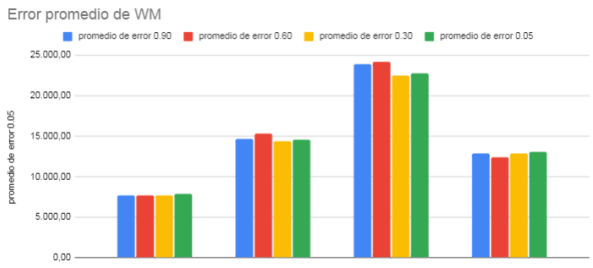

Figura 4. Error promedio alterando los parametros del WM para tamaños de particiones con diferentes tramos

- 5 % : 14.555,99
- 30 %: 14.346,92
- 60 %: 14.863,10
- 90 %: 14.777,35

IV-D. Cantidad de particiones por tamaño de solución (N)

Gracias a que el objetivo del algoritmo es la compresión sabemos que se debería llegar al objetivo de $N * K * T < |P|$, por lo tanto la cantidad de particiones (N) no debería superar $\frac{|P|}{K*T}$. Dejando eso a un lado se observa claramente como mientras aumenta la cantidad de particiones también disminuye el error de la solución final. Encontramos que un buen compromiso entre el error y la compresión [8]: que la cantidad de particiones sea $|P|/10$.

IV-D1. Tamaño de solución: Habiendo fijado todos los parámetros excepto N, se puede llegar al tamaño final de nuestra representación: son $(N * 0{,}1) * (3\ coeficientes * 2bytesporcoeficiente + 2bytesmarcadores)$ bytes en total, es decir $N * 0{,}8$ bytes. Sabiendo que la entrada son 2 bytes por muestra, logramos una tasa de compresión de 0.8:2, o 0.4:1.

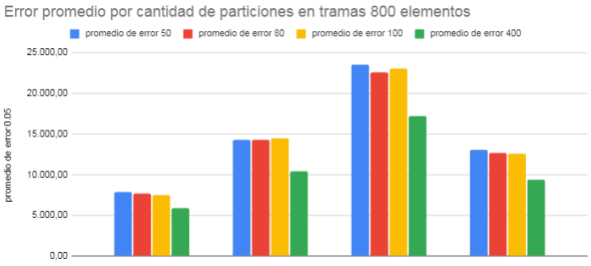

Figura 5. Error promedio alterando la cantidad de particiones por tramo

IV-E. Resultados de la aplicación del algoritmo

IV-E1. Canción de flauta: Esta instancia del problema proviene de una canción de flauta de 20 segundos de largo [9].

IV-E1a. Error: El algoritmo se ejecutó disjuntamente 12 veces sobre toda la instancia con los siguientes parámetros: una población de 200 individuos, y las primeras 10 veces se hizo con 1.000.000 de evaluaciones y luego, las siguientes dos, con 10.000.000 de evaluaciones, se obtuvo la siguiente gráfica de errores (distancia vectorial de nuestra solución al objetivo).

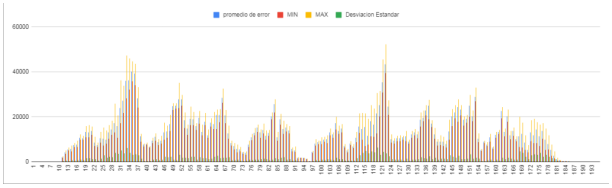

Figura 6. Resultados de un millón de iteraciones

para los siguientes resultados se obtuvo

- Error Promedio: 12.215,12
- Error Mínimo Promedio: 10.192,91
- Error Máximo Promedio: 14.644,72
- Desviación estándar promedio: 1.448,22

[10] [1]

[1]El error promedio es el promedio de los errores de todas las ejecuciones del algoritmo en una muestra

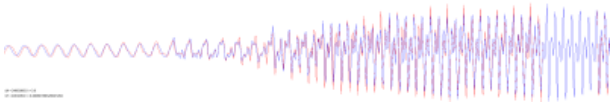

Figura 7. Resultados la ejecución del algoritmo en el tramo 11 siendo el azul la solución generada (Fitness = 1411.04)

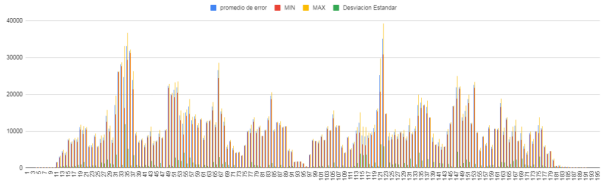

Figura 8. Resultado de 10 millones de iteraciones

para los siguientes resultados se obtuvo

- Error Promedio promedio: 9.492,24
- Error Mínimo Promedio: 8.821,89
- Error Máximo Promedio: 10.162,49
- Desviación estándar promedio: 1448,22

[11]

Se nota claramente como el algoritmo mejora sustancialmente con el aumento de generaciones, aun cuando la cantidad de evaluaciones es muy grande.

IV-E1b. Comparaciones: Decidimos comparar con el popular algoritmo MPEG Audio Layer (MP3), como todas las evaluaciones del algoritmo generan el mismo resultado el error promedio también será el error máximo y el error mínimo, y la desviación estándar es 0.

- Error Promedio: 241,6210912

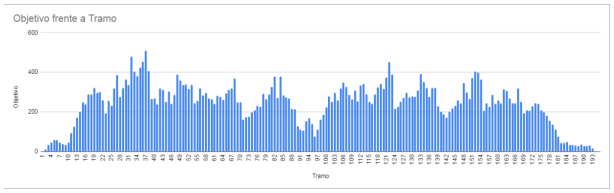

Figura 9. Resultado MP3s

IV-E2. Ruido de guitarra: Esta instancia del problema proviene de una guitarra, los siguientes son los resultados de realizar 5 ejecuciones sobre la instancia:

IV-E2a. Error:

- Error Promedio promedio: 31.113,66
- Error Mínimo Promedio: 27.092,52
- Error Máximo Promedio: 155.029,67
- Desviación estándar promedio: 3109,359645

[12]

IV-E3. Ruido de piano: Esta instancia del problema proviene de un piano, los siguientes son los resultados de realizar 5 ejecuciones sobre la instancia:

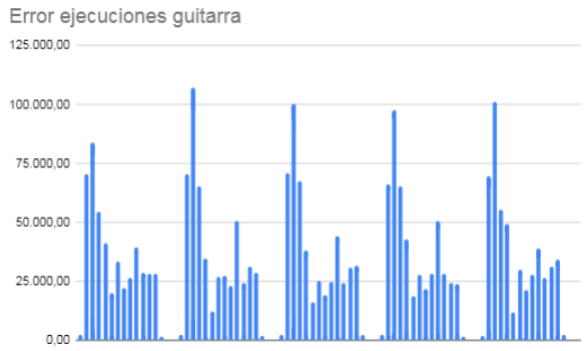

Figura 10. Resultado de 1 millón de iteraciones

Figura 11. Resultado de 1 millón de iteraciones promedio

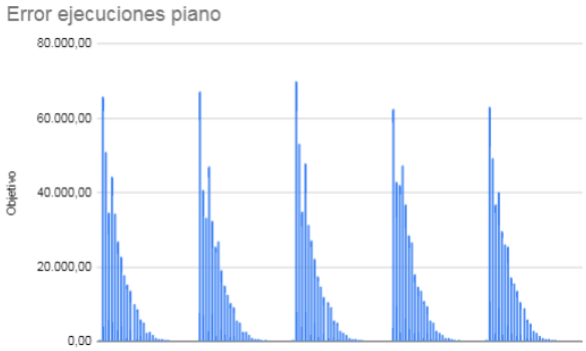

Figura 12. Resultado de 1 millón de iteraciones

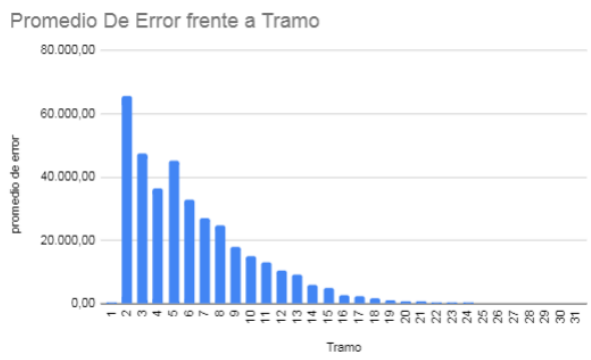

Figura 13. Resultado de 1 millón de iteraciones promedio

IV-E3a. Error:

- Error Promedio promedio: 11.838,29
- Error Mínimo Promedio: 10.883,95
- Error Máximo Promedio: 12.807,73
- Desviación estándar promedio: 795,00

[13]

IV-E4. Ruido de batería: Esta instancia del problema proviene de una batería, los siguientes son los resultados de realizar 5 ejecuciones sobre la instancia:

IV-E4a. Error:

- Error Promedio promedio: 37.057,57
- Error Mínimo Promedio: 34.687,52
- Error Máximo Promedio: 125.769,62
- Desviación estándar promedio: 53912,83461

[14]

IV-E5. Entorno de ejecución: Todas las ejecuciones fueron realizadas en nodos de cluster.uy [15] usando SLURM y ejecutándolos en la partición `normal`, con una cantidad de CPUs entre 4 y 16, y RAM adecuada dependiendo de los requisitos del problema.

Se utilizó una imagen Docker con Alpine Linux y OpenJDK 11 (headless) preinstalados [16], usando el software Singularity [17] para poder ejecutar la imagen. El JAR incluye el código principal, jMetal v5.10 [4] y sus dependencias.

Todos los scripts están escritos en `bash` y diseñados para ser ejecutados desde el directorio principal.

IV-E6. Tiempo de ejecución: Los resultados de los tiempos de ejecuciones para la flauta son los siguientes

- 1 Millón de evaluaciones, 10 ejecuciones y 8 hilos: 65459030ms = 65459.030s (como tiene 195 tramos es 167.84 segundos por tramo en promedio por ejecución).
- 10 Millón de evaluaciones, 2 ejecuciones y 8 hilos: 118552653ms = 118552.653s (como tiene 195 tramos es 303.98 segundos por tramo en promedio por ejecución).

V. Conclusiones

Se encontró que en la práctica nuestro algoritmo no logra una conversión mejor a MP3 en compresión, tiempo de ejecución, ni en correctitud, más no por esto la investigación en base a algoritmos evolutivos sobre el problema CMP está cerrada, ni mucho menos, en nuestra investigación comprobamos que a pesar de no llegar al rango de MP3 el algoritmo evolutivo puede acercarse a una solución correcta.

VI. Investigaciones a futuro

En nuestra investigación se ha trabajado solo con tramos de tamaño de 0.1 segundos de música porque estimamos que llegar al tiempo de convergencia de una solución la cual fuera más amplio sería también mucho mayor, futuras investigaciones podrían investigar como reacciona el algoritmo presentado a muestras de mayor tamaño.

Aunque nuestro equipo le dedicó gran cantidad de tiempo e investigación a la función de aproximación se estima que se

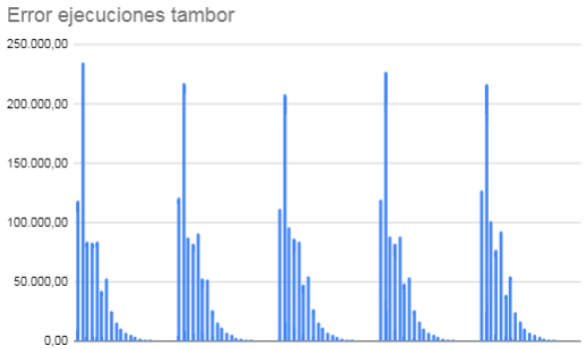

Figura 14. Resultado de 1 millón de iteraciones

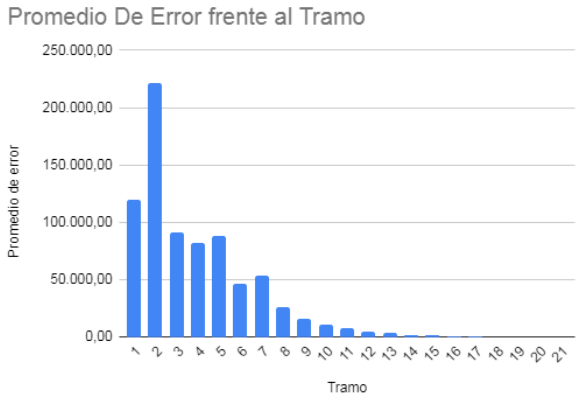

Figura 15. Resultado de 1 millón de iteraciones promedio

podrían encontrar maneras de reducir la cantidad de coeficientes o, en su defecto, encontrar maneras de aproximar mejor las muestras pudiendo así reducir la cantidad de particiones y de esta forma aumentar el poder de compresión.

También estimamos que dependiendo de los instrumentos utilizados en las muestras se podrían encontrar diferentes funciones que lograran aproximar la canción con mayor precisión.

También pensamos que el framework presentado se podría utilizar para diseñar un algoritmo evolutivo el cual a partir de una pieza musical de un solo instrumento, pueda deducir qué notas se están tocando en cada momento y así, solo con base en el sonido de la canción deducir qué notas se deben tocar en cada momento (similar al formato MIDI [18]).

Referencias

[1] M. Gulsen et al. (1995). ”A genetic algorithm approach to curve fitting”.
[2] Y. Mineur, M. Sevaux (2007). ”A curve fitting genetic algorithm for styling application”.
[3] Herramienta de procesamiento de audio y video
[4] Librería para Java que contiene algoritmos y utilidades para trabajar con algoritmos evolutivos
[5] The Historical Connection of Fourier Analysis to Music
[6] Totalidad de resultados de los experimentos con la mutación de ventana (coeficientes)
[7] Totalidad de resultados de los experimentos con la mutación de ventana (ancho de partición)
[8] Totalidad de resultados de los experimentos con la cantidad de particiones
[9] Principal ejemplo usado durante el desarrollo
[10] Totalidad de resultados de los experimentos con el sonido de flauta con un millón de interacciones
[11] Totalidad de resultados de los experimentos con el sonido de flauta con un diez millones de interacciones
[12] Totalidad de resultados de los experimentos con el sonido de guitarra
[13] Totalidad de resultados de los experimentos con el sonido de piano
[14] Totalidad de resultados de los experimentos con el sonido de tambor
[15] Cluster de cómputo usado para realizar las evaluaciones experimentales
[16] Imagen de Docker usada para la evaluación experimental
[17] Software que permite ejecutar procesos en ambientes pseudo-virtualizados
[18] Estándar industrial que define la interfaz entre instrumentos musicales y otros dispositivos digitales