UNIVERSITAT DE BARCELONA

Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Generación de Ritmos Funk con Algoritmos Genéticos

Autor: Francisco Diaz Ruiz

Director: Carlos Borrego Iglesias
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 20 de juny de 2021

Abstract

With the advance in technology of the current age, the capacity of computers to create or replicate things that normally the human being oversees making is rising. Most of those advancements helped or improved the work that people do, whatever is in construction with the usage of robotics or just daily help with automated systems. Those improvements even appeared in leisure, for example in the cinematic or music sectors, where it can be found movie projections fully generated by computer or songs made from scratch by informatic methods.

In this work, we expect to see if a computer can or cannot create, with the use of a concrete algorithm, that is the genetic, one specific type of music in a specific genre: the Funk.

Before getting in details, we will see bits of music theory and a brief explanation of what it is a genetic algorithm, to obtain some previous knowledge to help us understand the rest of the content.

Resumen

Con el avance de la tecnología en la era actual, la capacidad que tienen los ordenadores para crear o replicar elementos que normalmente el ser humano se encarga, está aumentando. Muchos de estos avances han servido para facilitar y mejorar el trabajo a las personas, ya sea en la construcción, con usos de robots, o simplemente ayuda en el día a día con sistemas automatizados. Han aparecido también en partes del ocio, como pueden ser la música o el cine, en donde podemos encontrar proyecciones generadas directamente por ordenador, sin intervención alguna de ninguna persona, o canciones que han sido creadas desde la nada por procesos informáticos.

En este trabajo, esperamos exponer este avance para ver si un ordenador puede o no, utilizando un algoritmo en particular que recibe el nombre de genético, crear un tipo específico de música de un género en concreto: el *Funk*.

Antes de entrar en detalle, veremos un poco de teoría musical y una breve explicación de que es un algoritmo genético y como está estructurado, para obtener así un conocimiento previo que nos ayude a entender mejor el resto del contenido que ofrece este proyecto.

Resum

Amb l'avanç de la tecnologia en l'era actual, la capacitat que tenen els ordinadors per crear o replicar elements que normalment el esser humà s'encarrega, esta creixent. Molts d'aquests avanços han servit per facilitar i millorar el treball a les persones, ja sigui en l'àmbit de la construcció, amb l'ús de robots, o simplement ajuda en el dia a dia amb sistemes automatitzats. Han aparegut també en les parts d'oci, com poden ser la musica o el cinema, on podem trobar projeccions generades directament per ordinador, sense intervenció alguna de cap persona, o cançons que han sigut creades des de cero per processos informàtics.

En aquest treball, esperem exposar aquest avanç per veure si un ordinador pot o no, utilitzant un algoritme en particular que rep el nom de genètic, crear un tipus específic de musica d'un gènere en concret: el *Funk*.

Abans d'entrar en detall, veurem una mica de teoria musical y una breu explicació de que es un algoritme genètic y com esta estructurat, per així obtenir un coneixement previ que ens ajudarà a entendre millor la resta del contingut que aquest projecte ofereix.

Agradecimientos

Después de muchos percances y problemas en los que he visto envuelto, por fin puedo decir que ha llegado el final de una etapa más de mi vida. Una etapa importante, en la cual sin la ayuda de todas las personas que han creído en mí, pienso que no podría haber superado como lo he hecho. Os agradezco desde el fondo de mi corazón que me hayáis soportado todo este tiempo y apoyado cuando más falta hacía.

En primer lugar, debo agradecer tanto a Carlos Borrego, como también a Simone Balocco, por proporcionarme toda la ayuda que me han dado, así como entender mi situación actual y apoyarme en todo lo posible para finalizar este trabajo cómodamente.

En segundo lugar, agradezco a toda mi familia y amigos que han estado ahí para darme un empujón cuando más lo necesitaba y apoyarme en todo momento a lo largo de mi carrera universitaria, llena de altibajos.

Para acabar, agradecer tanto al personal docente como a los compañeros que han cursado conmigo, que han hecho posible que haya llegado a este punto.

Sinceramente y de todo corazón, muchas gracias a todos, ha sido un placer y un orgullo poder haber contado con vosotros.

ÍNDICE

1. Introducción

La gran motivación detrás de este proyecto se encuentra en cómo, hoy en día, existen ya formas en la que los ordenadores pueden crear cualquier tipo de elemento mientras este esté dentro del alcance de ellos. Uno de estos elementos en concreto es la música y existen ya programas que facilitan la creación de canciones para los artistas, pero también son una realidad los programas informáticos que generan canciones de manera autónoma y esos son los que esperamos ver en acción. Visualizar como una inteligencia artificial puede crear un ritmo musical solo con unas pautas especificas es algo que es muy entretenido de observar y en este trabajo se ha puesto a prueba.

Para dar paso al proyecto es necesario hablar también de los conceptos de inteligencia artificial y algo de historia sobre la música, más en concreto, la historia del *Funk*, el género con el que se trabaja en este.

La Inteligencia Artificial, o IA como abreviatura, es algo que ha ido evolucionando con el largo de los años desde que apareció como concepto en 1956 por parte de John McCarthy, donde la definió como “la ciencia e ingenio de hacer maquinas inteligentes”. Ahora, el concepto de IA es más amplio, aunque siempre acaba reduciéndose a una simulación de la inteligencia humana que reciben las maquinas programadas para realizar diversas tareas y replicar las acciones del ser humano cuando estos lo hacen.
Una de las características principales de la IA es que esta disponga de la habilidad de tomar y realizar acciones por sí misma, sin interacción alguna de un operador, para alcanzar una meta en específico. Pero la IA también se puede dividir en diferentes partes como puede ser el *Machine Learning*, que consiste en que dicha IA pueda aprender y adaptarse a la información que se le proporciona sin ayuda, o el *Deep Learning*, técnicas que asisten a este aprendizaje automatizado a partir de grandes cantidades de datos que la inteligencia irá asimilando.

Por otra parte, el *Funk* es un género musical que se originó en Estados Unidos a mediados de los años 60, donde los grupos que tocaban este género eran principalmente de origen afroamericano. Este estilo musical es una fusión del *Soul*, *Jazz* y *Rhythm & Blues*. Una de las características de este estilo de música es que se basan en la repetición de un acorde a lo largo de toda la canción, logrando un distintivo muy particular que los otros estilos de los que parte no tienen. Este estilo tiene como “padre” a James Brown y su banda, quienes lo popularizaron y establecieron, relacionando el concepto “*Funk*” a un ritmo agresivo y sincopado, donde el primer golpe es el más fuerte/sonoro de toda la medida de tiempo, a la vez que las letras de las canciones hacen referencias a los problemas sociales y culturales de la época. Los instrumentos que normalmente son usados para tocar este tipo de canciones son el bajo, la batería, la guitarra eléctrica, el sintetizador o instrumentos de viento variados entre otros.
De este estilo ha influido en otros tipos de música como pueden ser el *Disco* o el *Boogie* y ha sido utilizado para crear nuevas canciones de los estilos como el *Hip Hop*, el *House* o el *Drum and Bass*.

2. Contexto Previo

2.1. Teoría Musical

Habiendo introducido un poco de la historia del origen de la música *Funk*, ahora es momento de hablar de cosas más técnicas sobre la música en general. Esta se puede dividir en dos elementos diferentes, uno es el que nosotros percibimos con nuestra audición, que recibe el nombre de acústica, y la forma escrita de ella, utilizando partituras y una notación especial para representar cada elemento de una canción.

Todas estas características se recogen dentro del término Teoría Musical encontrado dentro de la musicología, que es el campo de estudio dedicado a estructurar la metodología utilizada para realizar los análisis, la compresión y la composición de la música. La parte de esta teoría que se necesita saber es la que hace referencia a la escritura de partituras, así como entender que es una melodía, los acordes que contiene y que ritmo sigue. A continuación, se proveerán unas breves definiciones de la Real Academia Española[1], para cada palabra clave que utilizaremos a lo largo de este manuscrito:

- **Ritmo**: Proporción guardada entre acentos, pausas y repeticiones de diversa duración en una composición musical.
- **Compás**:
  - Signo que determina el ritmo en cada composición o parte de ella y la relación de valor entre los sonidos.
  - Ritmo o cadencia de una pieza musical.
  - Espacio del pentagrama en que se escriben todas las notas correspondientes a un compás y se limita por cada lado con una raya vertical.
- **Tiempo**: Cada una de las partes de igual duración en que se divide el compás.
- **Nota**:
  - Cada uno de los sonidos en cuanto está producido por una vibración de frecuencia constante.
  - Cada uno de los signos que se usan para representar las notas musicales.
  - La figura de una nota está formada por un *corchete*, que es como una cola, que sobresale por el extremo superior del cuerpo o *plica* y finalmente la *cabeza* que se encuentra en la posición de la partitura que corresponde a la nota musical a tocar.
- **Tono:**
  - Cualidad de los sonidos, dependiente de su frecuencia, que permite ordenarlos de graves a agudos.
  - Cada una de las escalas que para las composiciones musicales se forman, partiendo de una nota fundamental, que le da nombre.
- ***Ostinato*:** Motivo que se repite insistentemente durante una buena parte de una composición musical.
- **Sincopado:** Dicho de una nota: que se halla entre dos o más de menos valor, pero que juntas valen tanto como ella. Toda sucesión de notas sincopadas toma un movimiento contrario al orden natural, es decir, va contratiempo.

[1] Enlace a la R.A.E.: https://dle.rae.es/

En el caso de este proyecto, se busca encontrar un ritmo de cuatro tiempos, con 16 notas por compás, lo más próximo al *Funk* utilizando instrumentos de percusión de batería. Estos instrumentos son los siguientes:

- **Platillos tipo Hi-Hat (*Charles*)**: estos platillos son los que se encargan de crear el *Ostinato* a lo largo del compás. En este caso son utilizados de dos modos: abierto y cerrado. La representación en partitura de las notas producidas por este instrumento es con una x, pero para diferenciar entre ambos modos, la x abierto se encuentra o dentro de un círculo o tiene dicho circulo arriba de la partitura. En las partituras para percusión, las variaciones que se usan se encuentran en la parte más alta de esta.

- **Bombo**: Este instrumento, que en inglés recibe el nombre de *Bass/Kick drum*, es utilizado como normalmente se usaría en cualquier composición de percusión. Es representado en la partitura como un Fa que tiene la *plica*, o cuerpo, direccionada hacia abajo.

- **Tambor (Caja):** En el caso del tambor, o *Snare drum* en inglés, se utiliza de dos formas, una para notas normales en el DO de la partitura y la otra para representar notas fantasmas, que se explicaran a continuación, también en el DO.

- **Nota Fantasma**: Una nota categorizada como fantasma o muerta es una aquella que tiene peso rítmico, pero carece de tonalidad distintiva, por ende, estas se usan como para realizar efectos que incitan a percibir una nota en específico. Así mismo, también son usadas como notas completas para embellecer la composición. Estas pueden ser tocadas por los intérpretes o bien ignoradas si ellos lo desean. Para representar este tipo de notas se utiliza la X o bien con paréntesis envolviendo la cabeza, siendo esta última la que se ha utilizado debido a que se usa la X para representar a los platillos.

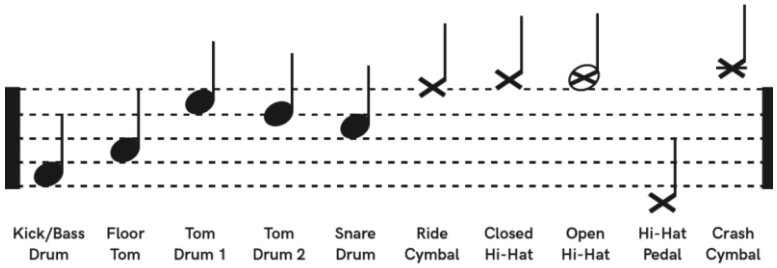

Fig. 1 Notación de notas de percusión[2]

Para la representación de ritmos de percusión no se suele usar una clave de sol, sino que en cambio se utiliza la clave neutra o de percusión, debido que tienen una posición para cada instrumento, como se puede observar en la figura anterior.

Fig. 2 Clave Neutra o de percusión

[2] Extraído del artículo de *Drum Notation for Beginners* de *schoolofrock.com*: https://anon.to/DqyKzt

En lo que respecta a las partituras, son las representaciones graficas de las composiciones musicales que los intérpretes deben tocar.

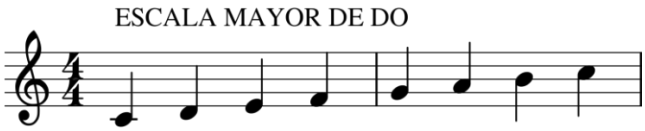

Fig. 3 Partitura de la escala mayor de Do[3]

En este proyecto, se emplea únicamente la "*Particella"*, nombre que recibe una partitura de solamente una parte de la composición, correspondiente a la sección de percusión. Esta sección se ha definido en este proyecto como una con compas cuaternario (4/4), eso quiere decir que en un total van a haber cuatro tiempos en total. Entonces los ritmos que se crearán tendrán una estructura de *1e&a 2e&a 3e&a 4e&a*, donde los dígitos corresponden a los tiempos y el resto a los intermedios.

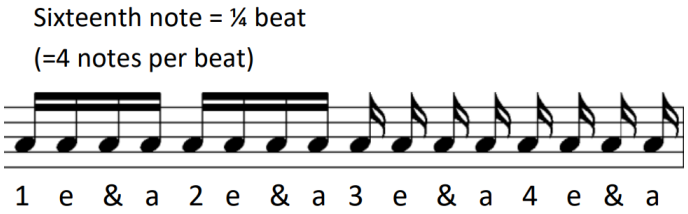

Fig. 4 Estructura de un ritmo de 4/4 de 16 notas [4]

Con la finalidad de recrear el género del *Funk* dentro del programa se han establecido unas características que casi todas las composiciones que forman parte de este estilo contienen, algunas mencionadas en la introducción de este documento, y son las siguientes:

- Golpe sonoro en el inicio de tiempo.
- Golpes de notas fantasma de caja sincopados en las posiciones 'e' y 'a' del ritmo.
- Repetición o *Ostinato* con los platillos, ya sea en posición abierta o cerrada, a lo largo de la composición.
- Los dos primeros tiempos de la caja, el bombo o el platillo abierto son iguales a los dos últimos de uno de los otros dos.

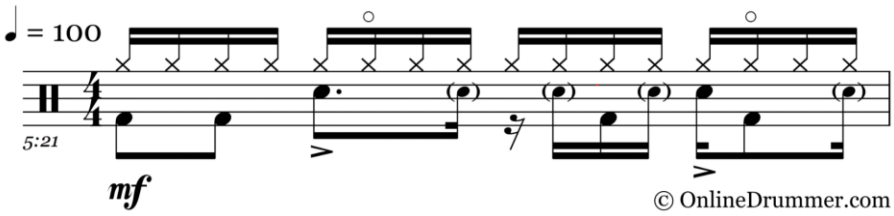

Fig. 5 Fragmento de la partitura de Funky Drummer de James Brown[5]

[3] Imagen originaria del articulo Progresión de acordes, los grados de la web: https://xaviguitarrabcn.com/progresion-de-acordes-grados/
[4] Extraído de *Rhythm Chart*, documento perteneciente a *Coeur D'Alene Public Schools*: https://www.cdaschools.org/cms/lib/ID01906304/Centricity/Domain/2057/Week%205%20-%20BASS.pdf
[5] Extraído del post de Steve Ley en *OnlineDrummer.com*: https://www.onlinedrummer.com/drum-beats/funky-drummer-james-brown-drum-break/

2.2 Algoritmo Genético

Creado originalmente por John Herny Holland en la década de los 70, pionero de la tecnología en la época y considerado como el padre de este, un algoritmo genético es una metodología evolutiva que aplica inteligencia artificial con la intención de recrear el desarrollo biológico para resolver determinados problemas, habitualmente de búsqueda y optimización. De aquí recibe el nombre de genético, debido a que se encarga de crear una población de individuos o *cromosomas* para luego someterlos a un proceso evolutivo con la finalidad de obtener los mejores individuos, donde estos serían los resultados para el problema planteado. La información que dichos *cromosomas* contienen recibe el nombre de *gen*.

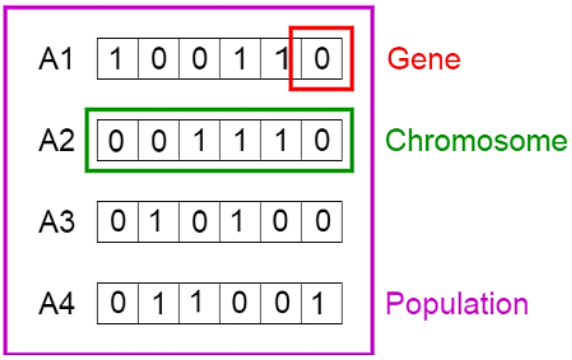

Fig. 6 Representación de una población[6]

GA(S)

parameter(s): S – set of blocks
output: superstring of set S

Initialization :
$t \leftarrow 0$
Initialize P_t *to random individuals from* S^*
EVALUATE-FITNESS-GA(S, P_t)

while *termination condition not met*
do
- *Select individuals from* P_t *(fitness proportionate)*
- *Recombine individuals*
- *Mutate individuals*
- EVALUATE-FITNESS-GA$(S, modified\ individuals)$
- $P_{t+1} \leftarrow$ *newly created individuals*
- $t \leftarrow t + 1$

return *(superstring derived from best individual in* P_t*)*

procedure EVALUATE-FITNESS-GA(S, P)
S – *set of blocks*
P – *population of individuals*
for each *individual* $i \in P$
do
- *generate derived string* $s(i)$
- $m \leftarrow$ *all blocks from* S *that are not covered by* $s(i)$
- $s'(i) \leftarrow$ *concatenation of* $s(i)$ *and* m
- $fitness(i) \leftarrow \frac{1}{\|s'(i)\|^2}$

Fig. 7 Seudocódigo de un algoritmo genético estándar[7]

[6] Imagen originaria del artículo de *Analytics Vidya* por Shubham Jain: https://www.analyticsvidhya.com/blog/2017/07/introduction-to-genetic-algorithm/
[7] Seudocódigo originario del documento *Zaritsky, Assaf & Sipper, Moshe. (2004). The Preservation of Favored Building Blocks in the Struggle for Fitness: The Puzzle Algorithm*.

Para obtener buenos resultados en la selección de la población generada, se someten a los individuos a un proceso de evaluación para observar su adaptabilidad, donde los que tienen mejor índice son los supervivientes, que luego se mezclarán para generar una nueva población, y los de menor índice serán descartados y reemplazados por los nuevos.

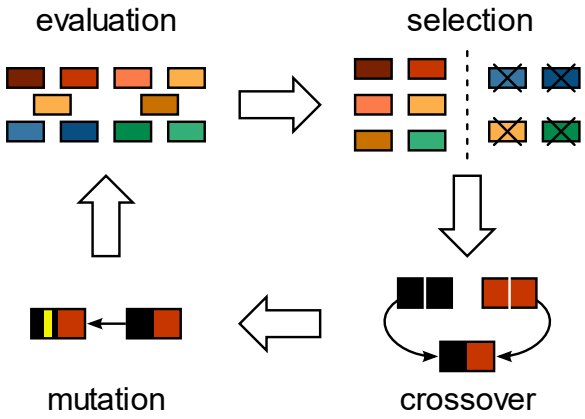

Fig. 8 Representación del proceso de Evolución

El proceso de funcionamiento de un algoritmo genético es sencillo de explicar y se puede separar en diferentes etapas, presentadas a continuación:

1) **Generación inicial de una población de individuos.**
    a. Esta población está conformada por los individuos del tipo de resultado que se busca obtener.

2) **Evaluación de adaptabilidad (*Fitness*).**
    a. Una vez se ha realizado el paso 1, se somete a todos los individuos a diferentes pruebas para determinar si susodicho individuo cumple con los requerimientos introducidos previamente. Estos requerimientos son las características que se buscan obtener en toda la población, por ende, son una especie de filtro que sirve para hacer una selección y, posteriormente, un descarte o emparejamiento.

3) **Creación de una nueva población:**

    a. **Emparejamiento**
        i. Una vez la población ha sido evaluada, se hacen parejas de individuos para combinarlos en dos o más individuos con las mejores características de la pareja. La terminología que se utiliza es de *padres* e *hijos*.

    b. **Mutación**
        i. Una vez ya se han creado los hijos, estos sufren un cambio genético o mutación, donde parte de la información de estos se modifica aleatoriamente.

4) Se repiten los pasos 2 y 3 hasta encontrar la solución más cercana a lo deseado, que correspondería con el individuo con mayor valor de adaptabilidad o Fitness**.**

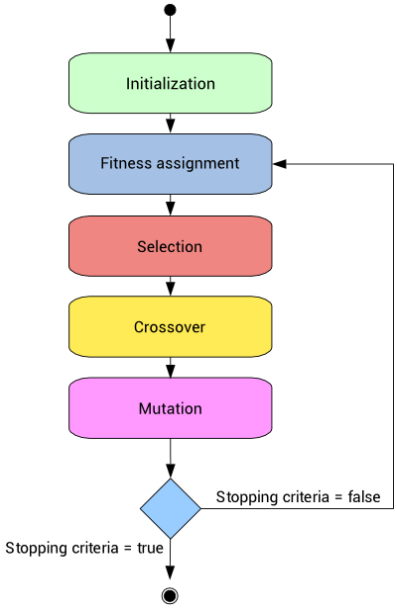

Fig. 9 Diagrama de flujo de un Algoritmo Genético[8]

Dependiendo de que uso se le dé a este tipo de algoritmo, puede tener un criterio para finalizar o no. En el caso de este proyecto, este criterio no existe y se deja al algoritmo correr hasta llegar al máximo de generaciones establecido y se extrae el resultado más valorado, pero puede haber otros proyectos donde el criterio puede ser encontrar un resultado en concreto y hacer que se detenga la ejecución a la mitad de las generaciones posibles.

Pero este algoritmo también tiene sus desventajas, por ejemplo:

- Cuando se tiene que tratar con una gran cantidad de datos complejos, ya que puede aumentar el tiempo de tarda el programa para obtener resultados y la calidad de la solución puede decaer rápidamente.

- Cuando se busca hallar resultados extremos, es decir, entre bien o mal, debido a que el algoritmo no responderá adecuadamente al generar resultados aleatorios. Se debe dar un margen de aceptación para encontrar resultados satisfactorios.

- No se define una función de *Fitness* correctamente, provocando que los resultados no salgan como deseado. Un ejemplo sería no penalizar la puntuación cuando no cumple con una condición, haciendo que el algoritmo siga validando este individuo en vez de descartarlo.

Algunas de las situaciones donde se aplica este tipo de algoritmo son:

- Para evolucionar programas informáticos para tareas específicas, tales como para ordenar redes informáticas o para optimización de tareas, como puede ser el *Travelling Salesman Problem* (TSP)[9].
- *Machine Learning*: Diseñar redes neuronales para sistemas clasificadores, para sistemas de producción
- Aplicaciones económicas para predecir movimientos en los mercados y generar estrategias comerciales.

[8] Imagen proveniente del artículo de *Neural Designer*: https://www.neuraldesigner.com/blog/genetic_algorithms_for_feature_selection
[9] Para más información: https://en.wikipedia.org/wiki/Travelling_salesman_problem

3. Objetivo

Una vez explicado el contexto previo y así facilitar la comprensión de este documento, es momento de indicar el objetivo principal de este proyecto.

Dicho objetivo es:

- Establecer un algoritmo genético que genere una población de ritmos de percusión, utilizando los instrumentos mencionados anteriormente, para crear así uno que se asemeje a un ritmo perteneciente a la corriente musical del *Funk*.

Este proyecto también tiene objetivos complementarios que son:

- Poder exportar los ritmos resultantes de las ejecuciones del programa a un fichero de tipo MIDI para poder reproducirlos en un reproductor habilitado para este tipo de ficheros.

- A partir de los datos del ritmo generado, exportarlo en formato de partitura y poder visualizarla en una imagen o fichero con extensión *XML*.

El proyecto estará programado en el lenguaje de programación de *Python 3*, en concreto un *Jupyter Notebook*. Para la parte de las partituras se utilizarán los programas de *Lilypond* y *MuseScore 3*, que sirven para crear y visualizar partituras.

Fig. 10 Logo Lilypond

Fig. 11 Logo MuseScore 3

Por lo que respecta al algoritmo, se partirá de una implementación de clase abstracta extraída de un artículo de investigación[10] que detalla el funcionamiento de un algoritmo genético simple y se extenderán las funcionalidades de este para poder alcanzar el objetivo principal, por lo que será necesario definir las funciones de mutación, emparejamiento (*Pair*), adaptabilidad (*Fitness*) así como también especificar los parámetros iniciales y el *main* del programa.

Fig. 12 Logo Python

[10] Enlace del artículo del Dr. Robert Kübler: https://towardsdatascience.com/an-extensible-evolutionary-algorithm-example-in-python-7372c56a557b

4. Implementación

En esta sección se presenta la estructura del código fuente del proyecto y una breve explicación de cada clase y función que este contiene.

4.1 Clase Abstracta - Individuo

El principio de la implementación del algoritmo genético empieza con el modelo de clase abstracta que definirá las funciones principales clave que contendrá el programa. Dichas funciones son las mencionadas previamente y están declaradas de la siguiente manera:

- ***__init__(self, value=None, init_params=None)***
    - Este es el inicializador de cada individuo. *Value* representa el contenido de cada cromosoma, mientras este no tenga un valor previo, creará uno aleatoriamente con las características introducidas por el parámetro *init_params* usando la función *_random_init*. Sirve para crear la población inicial.
- ***pair(self, other, pair_params)***
    - La función encargada de realizar el cruce entre dos individuos que han sido emparejados. *Other* es el otro individuo y *pair_params* las condiciones que tienen que cumplir durante el cruce.
- ***mutate(self, mutate_params)***
    - Los hijos creados como resultado de la función pair() activan esta función para mutar una parte de su contenido (*value*) de una manera especificada utilizando el parámetro *mutate_params.*
- ***_random_init(self, init_params)***
    - Función responsable de crear individuos con valores aleatorios a partir de *init_params.*

4.2 Clase *Rhythm* – Individuo

Ahora que la clase abstracta ha sido explicada, se puede empezar a explicar la clase principal de este programa, el individuo *Rhythm*. Esta clase usa la clase abstracta para definir su funcionalidad. En este caso solo se explicarán los detalles de cada función de la clase abstracta que se ha modificado para que se puedan crear los ritmos y trabajar con ellos.

- ***_random_init*:**
    - En este caso, se deben crear los ritmos de los 5 instrumentos, contando los platillos cerrados y abiertos como dos diferentes y las notas fantasmas de la caja como otro instrumento más. Cada instrumento es un array con una longitud definida por *init_params,* que en nuestro caso es 16 por el compás definido, y por ello se llama a una función auxiliar (*rand_bin_arr()*) que se encarga de crear dichas listas e inicializarlas con valores de 0 y 1 aleatoriamente. Cuando una nota es 1, quiere decir que el instrumento sonará en ese instante de tiempo mientras que, si es 0, este se ausentará.

      Finalmente, todos los instrumentos están guardados en otra lista que es la que corresponde al *value* del individuo.

- ***Pair:***

  - Esta función se puede dividir en diferentes partes, ya que para hacer el emparejamiento tiene que seguir unas pautas, declaradas en la función *Fitness* que se explicará en detalle más adelante, para cada instrumento. Básicamente comprueba que instrumento es mejor de entre los dos padres para luego que los hijos resultantes puedan heredarlo.

    Entonces, primero comprobará que el instrumento suene, ya que se ha decidido que todos los instrumentos deben sonar en la composición, aunque sea únicamente una vez.

    Recurrirá a utilizar dos funciones auxiliares, que son las siguientes:

    - ***ostinato_check(arr1, arr2, narr, osti)***

      - Esta auxiliar se encarga de mirar si los platillos de cada padre, tanto si son abiertos como cerrados, tienen la repetición a lo largo del compás.
        Arr1 y *arr2* corresponden a los instrumentos de cada padre, *narr* será el mejor instrumento de los dos y *osti* es un booleano que indica si se cumple la condición de repetición o no. Cada platillo tiene su propio booleano que entrará por parámetro y su valor será modificado en el proceso, para indicar cuál de los dos es el que cumple la condición, ya que solo uno de ellos puede.

    - ***rhyt_check(arr1, arr2, check)***

      - Al igual que la anterior función auxiliar, esta compara los demás instrumentos y selecciona el mejor de los dos que entran por parámetro. El parámetro *check* sirve para indicar que el instrumento que se está comparando es el que corresponde a las notas fantasma de la caja. Si el booleano que corresponde a que se cumple la repetición en el platillo cerrado es verdadero, el platillo abierto utilizará esta auxiliar para realizar la comparación. En cambio, si es falso, usará la anterior función auxiliar.

    Una vez que se han comparado los instrumentos y formado el hijo resultante, este se retorna como un nuevo objeto de tipo *Rhythm* para rellenar de nuevo la población.

- **Mutate:**

  - Una vez se han creado los hijos a partir del emparejamiento con la función *pair,* los hijos invocan esta función con la intención de mutar una parte de su *value.* En este caso, las mutaciones son la alteración completa de uno de sus instrumentos en formato de negación. Esto quiere decir que un instrumento aleatoriamente cambiará las notas que tenía como 0 a 1 y viceversa. El *mutate_param* en este caso es una probabilidad de mutación, donde se mutarán más o menos notas dependiendo lo grande que esta probabilidad.

4.3 Función *Fitness* – Evaluación de Adaptabilidad

Se puede asumir que esta es una, por lo no decir más importante, de las funciones de este proyecto, pues se encarga de hacer de filtro para los individuos para evaluar aquellos que cumplen lo mejor posible las condiciones establecidas.

En este caso, todas las condiciones que esta función utiliza para escoger los mejores resultados corresponden a las características del género musical con el que estamos trabajando.

Cuando los individuos proceden a ser evaluados, se les asigna un resultado de adaptabilidad o "*Fitness Score"* y si cumplen las condiciones, se le incrementará el valor de su score, pero se penalizará en el caso de que no las cumplan. A partir de este proceso de evaluación se obtendrán los mejores resultados correspondiendo al número más alto de fitness de entre toda la población.

Recordemos que estamos trabajando con un ritmo de 16 notas, es decir, un compás de 4/4 que se divide en las siguientes partes:

1e&a 2e&a 3e&a 4e&a

Las características por las que los individuos son evaluados están listadas a continuación:

i. Uno de los platillos, **Abierto** o **Cerrado**, debe tener una repetición a lo largo de los cuatro tiempos. Eso quiere decir que el ritmo tiene que ser consistente para cada parte de la división del compás. Por ejemplo: **1** e & a **2** e & a **3** e & a **4** e & a

ii. La **Caja**, el **Bombo** y/o el platillo **Abierto** están presentes en el primer tiempo (1) y en el resto se ausentan (e&a). De esta manera se consigue afianzar el rasgo del *Funk*, que es golpe inicial potente al inicio de cada cuarto de tiempo.

iii. Si la **Caja** contiene **notas fantasmas**, estas deben estar presentes en los tiempos "e/a". De esta manera se obtiene un ritmo con notas sincopadas, proporcionándole un distintivo del género. En el caso de que se encuentren en el mismo tiempo que las notas normales de la caja, se penalizará el score.

iv. El número total de notas presentes de la **Caja** y el platillo **Abierto** no deben superar 5. Un exceso de estos dos instrumentos haría que el ritmo fuese inconsistente con lo que se desea alcanzar.

v. El platillo **Abierto** no tendrá más de 4 apariciones a lo largo de todo el compás. Mucha aparición de este instrumento comportaría a un ritmo no consistente.

vi. No deben aparecer tres notas consecutivas del **Bombo**, ya que esto no es una práctica que se utiliza en el estilo musical.

vii. En el caso de que un ritmo tenga notas de la **Caja** presentes en los tiempos 2 y 4 y **notas fantasmas** en los tiempos "e" y "a", será un punto muy positivo para el ritmo ya que esto hace que sea "*Funky*".

viii. Si entre la **Caja**, el **Bombo** y/o el **Abierto** hay una respuesta, es decir, los últimos dos tiempos de uno de ellos (3e&a 4e&a) son similares a los dos primeros (1e&a 2e&a) de otro, deriva en que el ritmo sea más "*Funky*". En caso de que sea **Caja**-**Bombo**, es más "*Funky*" que si se tratara con el **Abierto**.

ix. Si entre dos notas del **Bombo** y la **Caja** hay una separación de 6 tiempos (por ej. 1e&a 2e&) donde no hay notas, el ritmo es sincopado y será más "*Funky*".

Como se había mencionado anteriormente cuando se documentaba la función *pair*(), algunas de estas condiciones también se usan para comparar a los padres y seleccionar los instrumentos que los hijos heredaran.

Una vez se han detallado las características que debe cumplir un ritmo, solo hace falta hacer la comprobación de que dicho individuo las cumple. Ya que los ritmos son listas de listas, es decir, que funcionan como una matriz, solo es necesario hacer iteraciones como mucho para detectar si pasan o no.

4.4 Clase *Population* – Población de Individuos

Esta clase representa la población total de individuos con los que el algoritmo trabaja, además de que es la encargada de crear la población inicial, por ende, es la que se ocupa de crear todos los objetos de tipo *Rhythm*, evaluarlos con la función de *Fitness* y ordenarlos de menor valor a mayor valor, siendo el último elemento el mejor individuo de todos.

Las funciones internas que esta clase contiene son las siguientes:

- ***__init__(self, size, fitness, individual_class, init_params)***
    - Esta función crea la población inicial del tamaño deseado indicado por el parámetro *size.* El parámetro *fitness* es el valor de fitness para cada individuo y se usará para hacer la ordenación.
- ***replace(self, new_individuals)***
    - Función encargada de introducir los nuevos individuos generados por el pareado y eliminar a los peores individuos que están destinados a ser descartados. También vuelve a ordenar la población según su *score.*
- ***get_parents(self, n_offsprings)***
    - Función destinada a buscar los padres a partir de la posición de cada padre y el número de hijos que han engendrado.

4.5 Clase *Evolution* – Evolución de individuos

Esta clase es la principal del programa, en otras palabras, esta es la que inicia todo el programa y lo pone en funcionamiento, ya que el objeto de tipo evolución es el responsable de llamar a *Population* con un determinado tamaño, que indica el total de individuos con el que se trabajará, establecer que la clase del individuo es de tipo *Rhythm,* indicar los atributos y el número de hijos que se quiere obtener a la hora de hacer el emparejamiento, así como la probabilidad para la mutación y los *init_params* de los primeros individuos.

Esta clase tiene las siguientes funciones:

- ***__init__(self, pool_size, fitness, individual_class, n_offsprings, pair_params, mut ate_params, init_params):***
    - Esta función hace una llamada a la clase *Population* e inicializa el programa.

- ***step(self):***

    - Función responsable de hacer que pasen los ciclos de ejecución, donde se generaran se emparejaran individuos, crearan nuevos hijos, estos mutaran y reemplazaran a los individuos antiguos con peor *fitness score*

4.6 Función *midiExport* – Creador de ficheros MIDI

Esta función, como su nombre indica, es la encargada de convertir los datos de los ritmos resultantes, que son listas de ceros y unos, en ritmos musicales que se pueden reproducir. El ritmo por convertir, así como el nombre del fichero resultante de dicha acción son introducidos como parámetros.

Para llevar a cabo esta conversión, se necesita una librería que permita hacer este tipo de transformación. En este caso se ha utilizado la librería de *MIDIUtils* y su módulo de creación de ficheros con extensión MIDI para realizar la tarea. Dicho módulo requiere de información con la que se va a crear el nuevo fichero, la información es la siguiente:

- Las notas de los instrumentos, en nomenclatura MIDI. Todos los instrumentos que un fichero MIDI puede reproducir tienen un dígito asignado, o grado, y están comprendidos en un repertorio. En este caso, son necesarios los dígitos correspondientes a los 3 instrumentos que se utilizan, así como las formas alternativas de estos, esos dígitos son los presentados a continuación:

    - 42 – Platillo Cerrado
    - 36 – Bombo
    - 40 – Tambor / Caja
    - 46 – Platillo Abierto
    - 38 – Caja (Nota Fantasma)

- El número de pistas que se desea utilizar, donde cada pista corresponde a una canción diferente. En este caso, como un ritmo generado por el programa corresponde a una sola canción, solo es necesario una pista, por ello se usa la configuración predeterminada de valor 0 para la variable *track*, que indica que es pista única.

- El canal de mensajes del lenguaje MIDI con el que se desea trabajar. Los canales son el indicador de tipo de instrumentos que se usan en la composición, por ello, dependiendo del canal que se escoja, pueden sonar instrumentos de viento, percusión, cuerda e incluso de tecla, como puede ser el piano.

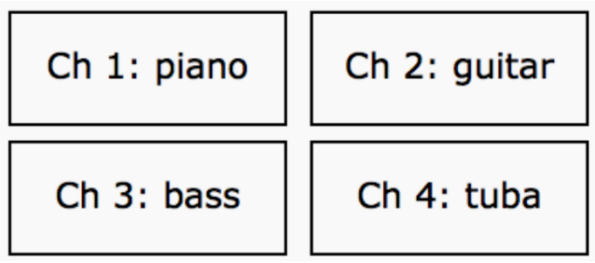

Fig. 13 Ejemplo de canales MIDI

- El tiempo, en *Beats* (golpes o notas), en el que iniciará la pista, en este caso, como es una totalmente nueva solo es necesario indicar que empieza desde el principio, en otras palabras, 0. Este tiempo se usará más adelante para ir añadiendo las notas conforme se va haciendo la conversión.

- La duración, también en *Beats*, que durará una nota cuando se reproduzca. Cuanto más alto sea este valor, más tiempo sonará dicha nota.

- El tempo que utilizará la pista, definido en golpes por minuto (*BPM*). Este tempo hace referencia a la velocidad en el que la pista reproducirá las notas, mientras que un valor alto hará que la pista tenga un sonido alegre, un valor bajo resultará en un ritmo lento y letárgico.

- El Volumen, en un valor de 0 a 127 como estándar de los ficheros MIDI, con el que sonarán las notas al reproducirse.

Una vez se han establecido la información necesaria, solo es necesario crear el fichero, para ello se llama a la función de la librería llamada *MIDIFile* y se le introduce por parámetro el número total de ficheros que se desean crear, en este caso solo 1. Posteriormente se le introduce a este nuevo fichero la información del número de pista, el tiempo de inicio y el tempo que debe seguir.

Ya hecho todo lo anterior, el siguiente paso es introducir las notas por instrumentos. Para realizar dicha tarea, se recorre la lista de instrumentos (*value*) del individuo y para cada instrumento se introduce una nota cada vez que se encuentre un 1 en la lista. A la vez que esto pasa, el tiempo se va incrementando para así progresar a lo largo de la pista. Finalmente, cuando todos los instrumentos ya han sido insertados en la pista, se hace una operación de guardado clásico de *Python* indicando que la extensión del fichero es de tipo "*.mid*", que corresponde al fichero MIDI.

Para poder reproducir dichos ficheros, es necesario que el reproductor que se vaya a utilizar disponga de un códec que permita realizar la lectura de dichos ficheros, de lo contrario estos ficheros no se podrán reproducir.

4.7 Función m2SheetExport – Creador de partituras

Esta función se encarga de convertir los ritmos resultantes de la ejecución del algoritmo genético en partituras que se podrán visualizar tanto en formado de imagen *.PNG* como también en fichero *XML*. Para ello es necesario recurrir a una librería de *Python* que utiliza el formato de fichero *MusicXML*[11] y en este caso se ha utilizado *music21*. Entran como parámetros el nombre del fichero junto con el ritmo a convertir y el titulo que recibirá la partitura resultante.

Music21 es una librería para generar ficheros *MusicXML* creada por el M.I.T.[12] (Instituto de Tecnología de Massachusetts) que tiene la finalidad de ayudar a resolver problemas de musicología utilizando ordenadores para así estudiar grandes *datasets* de música, al igual que generar ejemplos musicales, enseñar fundamentos de la teoría musical o componer música.

Con la ayuda de esta librería es posible hacer la representación de los individuos generados a partitura, para ello, hace falta especificar algunos datos al igual que se hizo con la conversión a fichero MIDI.

[11] Enlace a la página principal de MusicXML: https://www.musicxml.com/
[12] Enlace a la página principal del MIT: https://web.mit.edu/

Estos son los datos:

- Las notas musicales correspondientes a la posición del instrumento que se va a añadir. En este caso, las notas deben estar categorizadas según la escala musical, y el listado que se ha usado es el siguiente:

    - G5 – Platillo Cerrado y Platillo Abierto
    - F4 – Bombo
    - C5 – Tambor / Caja y Nota Fantasma

    Se comparten algunas notas ya que son el mismo instrumento, pero a la hora de representarlas, son diferentes debido al uso alternativo de estos.

- La notación musical correspondiente a las notas. Cada instrumento es representado por una nota en la partitura, en diferentes alturas de esta, y a su vez tienen también algunas características como las que se mencionaron en la explicación previa de la teoría musical.

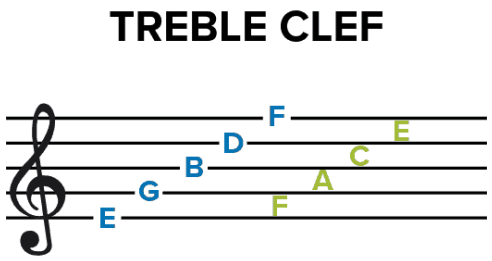

Fig. 14 Nomenclatura de partitura con clave de Sol

Una vez ya se ha aclarado los datos necesarios para la transformación a partitura, es necesario explicar el proceso que se lleva a cabo para realizar esta tarea. Este proceso lo podemos dividir en varios pasos, expuestos a continuación:

1) Se debe crear un objeto de tipo *Stream.Part,* del paquete de *music21*, donde se van a añadir las diferentes notas de los instrumentos. Por parámetro, se le pasa al constructor de este objeto los detalles que se desean tener, que en este caso corresponden al compás 4/4 así como el tempo del ritmo. Así mismo, se le añade un objeto *Metadata,* también del mismo paquete, y se le indica el título de la partitura.

2) Luego se procede a crear un de tipo *Stream.Measure,* que es la colección de las notas que después se añadirá al *Stream.part.*

3) Se hace una lectura sincronizada de todas las listas de valores, que corresponden con los instrumentos del ritmo (*value*), y para cada 1 que se encuentre en las listas, se añade una nota correspondiente al instrumento dentro del objeto *Stream.Meaure*. En el caso de que haya dos o más instrumentos que suenan al mismo tiempo, se crea un objeto de tipo *Chord (acorde),* que tendrá como parámetro una lista de las notas que toca añadir.

4) Una vez ya se han introducido todas las notas, se añade el objeto de tipo *Stream.Measure* dentro del objeto de tipo *Stream.Part.*

5) Ya finalizada la creación de la partitura en el formato de *music21*, es momento de hacer la conversión de fichero *XML* y a partitura en formato *PNG*. Para llevar a cabo esto, es importante que primero de todo definir un entorno donde aparezcan las rutas de los programas que se encargaran de hacer esta tarea, que en este caso son *Lilypond* y *MuseScore 3*.

```
us = environment.UserSettings()
us['lilypondPath'] = 'C:/Program Files (x86)/LilyPond/usr/bin/lilypond.exe'
us['musescoreDirectPNGPath'] = 'C:/Program Files/MuseScore 3/bin/MuseScore3.exe'
us['musicxmlPath'] = 'C:/Program Files/MuseScore 3/bin/MuseScore3.exe'
```

Fig. 15 Ejemplo de configuración de rutas de entorno

Estas rutas harán posible que la función utilice esos programas para llevar a cabo la conversión.

6) Finalmente, y con todo ya preparado, se usan las funciones *ConverterLilypond()* y *ConverterMusicXML()* que se hayan dentro del paquete de *music21*, en las que pasan como parámetros el objeto *Stream.Part*, el formato a convertir, el nombre del fichero más la ruta donde se quiere guardar el fichero resultante y un formato alternativo si se desea, que en este caso es *PDF* y *PNG*.

4.8 Función auxiliar data_collection – Generador de gráficos.

Función que se encarga de mostrar gráficamente por pantalla un gráfico con la evolución del *fitness score* de los tres mejores individuos a lo largo de las generaciones utilizando la librería de *Python Matplotlib*[13]. Genera y almacena un fichero *SVG* con el resultado en una carpeta que crea en caso de que esta no exista.

4.9 Función Main – Ejecutor del programa

Para acabar, se llama a la función *main*() del programa que recibe como parámetros el número total de individuos con los que se desea trabajar, el número de generaciones o épocas que el programa llevara a cabo y la probabilidad de mutación. Posteriormente crea un objeto de tipo *Evolution* donde añade toda esta información como parámetros de esta.

Ya creado el objeto anterior, la función procede a cronometrar la ejecución, es decir, el número de generaciones en que los individuos se van juntando y procreando, usando el paquete de *datetime* de *Python*.

Una vez pasan todas las generaciones especificadas por el parámetro de entrada *epo,* la función extrae de la población a los tres mejores individuos, que coinciden con los tres últimos individuos del conjunto, los muestra por pantalla y los pasa a formato MIDI utilizando *midiExport*. Además, el mejor de todos también es convertido en partitura mediante *m2SheetExport.* Finalmente llama a la función auxiliar *data_collection* para generar un gráfico de los resultados.

[13] Sitio web oficial de *Matplotlib*: https://matplotlib.org/

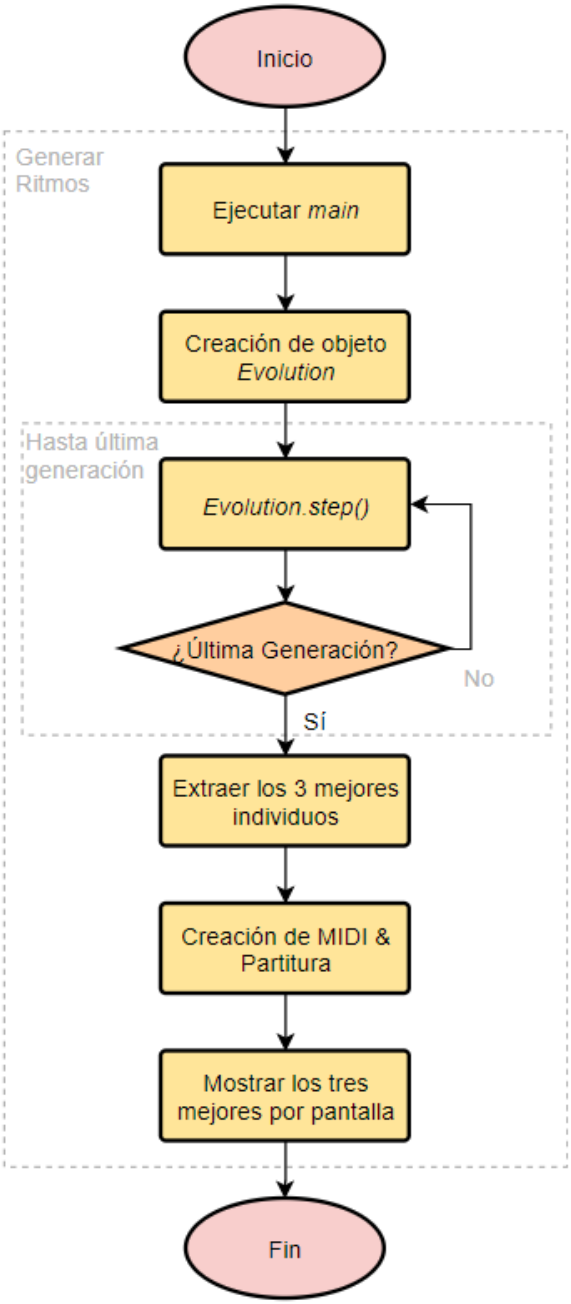

Fig. 16 Proceso de ejecución del Main

5. Resultados

Habiendo explicado todas las funciones y clases del proyecto, así como el proceso que llevan a cabo, ahora solo queda mostrar los resultados obtenidos.

5.1 Resultados del Algoritmo Genético

Para comenzar, un resultado de una ejecución de este programa se visualiza de la siguiente manera:

```
Time Elapsed: 0:04:03.184240

Mutation Probability: 0.6

Mejor Ritmo - Epoch: 7000  Pool: 500

 charles [0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0.]
 bombo [1. 0. 0. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 0. 1. 0.]
 snare [0. 1. 1. 0. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0. 1. 0.]
 ghost snare [0. 1. 0. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 1. 1. 0.]
 charles open [1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

 fitness score =  145
----------------------------------------

Segundo Mejor Ritmo - Epoch: 7000  Pool: 500

 charles [1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 1. 0. 0. 1. 0. 1.]
 bombo [1. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1.]
 snare [1. 0. 0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 1. 1. 0. 1.]
 ghost snare [1. 1. 0. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 0. 0. 0.]
 charles open [0. 1. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 1. 1. 0. 1.]

 fitness score =  140
----------------------------------------

Tercer Mejor Ritmo - Epoch: 7000  Pool: 500

 charles [0. 0. 1. 1. 1. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 1.]
 bombo [1. 0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 1. 0. 0. 1. 1.]
 snare [1. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0.]
 ghost snare [1. 1. 1. 1. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 0. 0.]
 charles open [0. 1. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0.]

 fitness score =  140
----------------------------------------
```

Fig. 17 Ejemplo de un resultado de 500 ritmos con 7000 generaciones y probabilidad de 0.6 sobre 1 de mutar

En la imagen se puede apreciar el tiempo que ha tardado en dar resultados el programa, cual ha sido la probabilidad de mutación que ha sido aplicada a todos los hijos, así como los tres mejores ritmos resultantes con los instrumentos mostrados junto a su valor de *fitness*. Para hacer un registro de la evolución del score de los tres mejores individuos se ha recurrido a hacer gráficos *Fitness Score-Generaciones*[14].

[14] Figura 18 – Pagina 24.

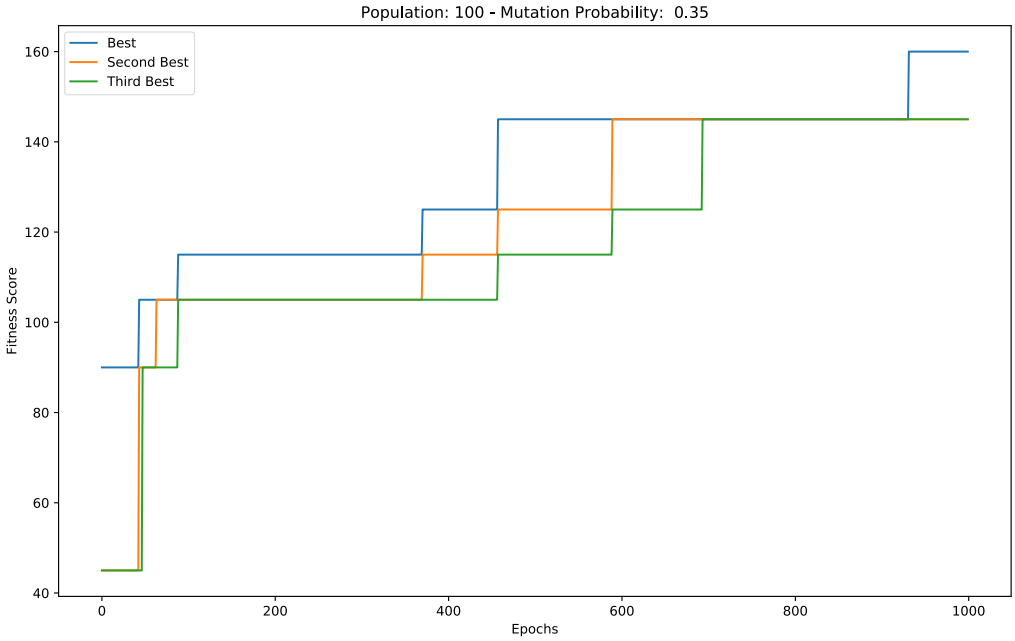

Fig. 18 Ejemplo gráfico de la evolución de Fitness de los tres mejores resultados de una ejecución.

Dependiendo de si se varia cualquier de los parámetros necesarios se pueden obtener resultados diversos. Estas variaciones se pueden observar a continuación, donde se verán resultados dependiendo si se modifica la probabilidad de mutación, la cantidad de individuos que forman la población o bien el número de generaciones por la que dicha población pasa.

5.1.1 Variación de Población de Individuos

- Variación del número de Individuos con una cantidad estática de generaciones, en este caso 5000 y una probabilidad invariable de 0.35:

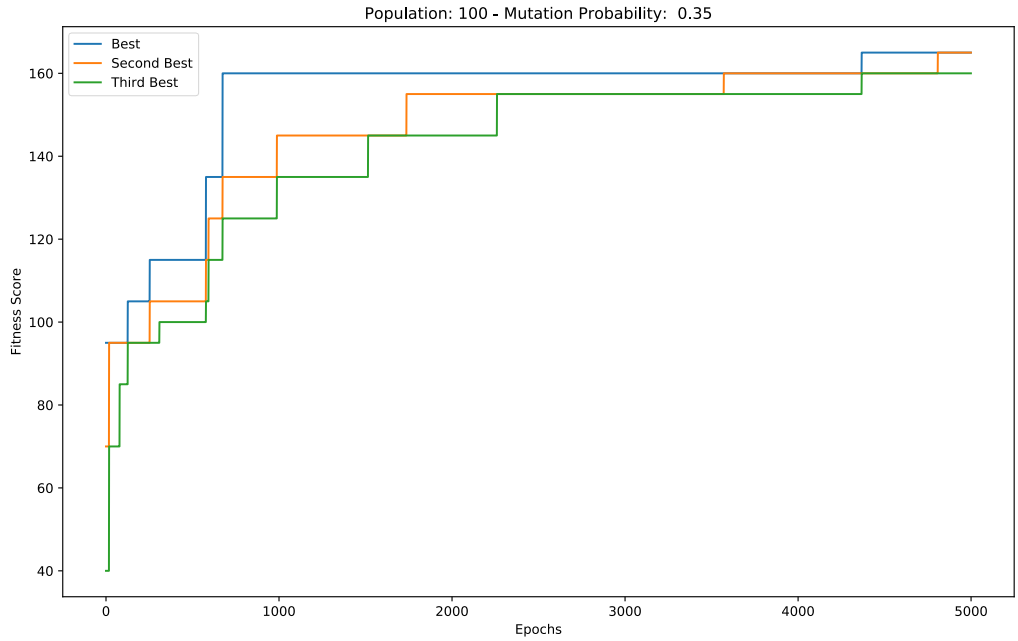

Fig. 19 Resultado con una población de 100 individuos

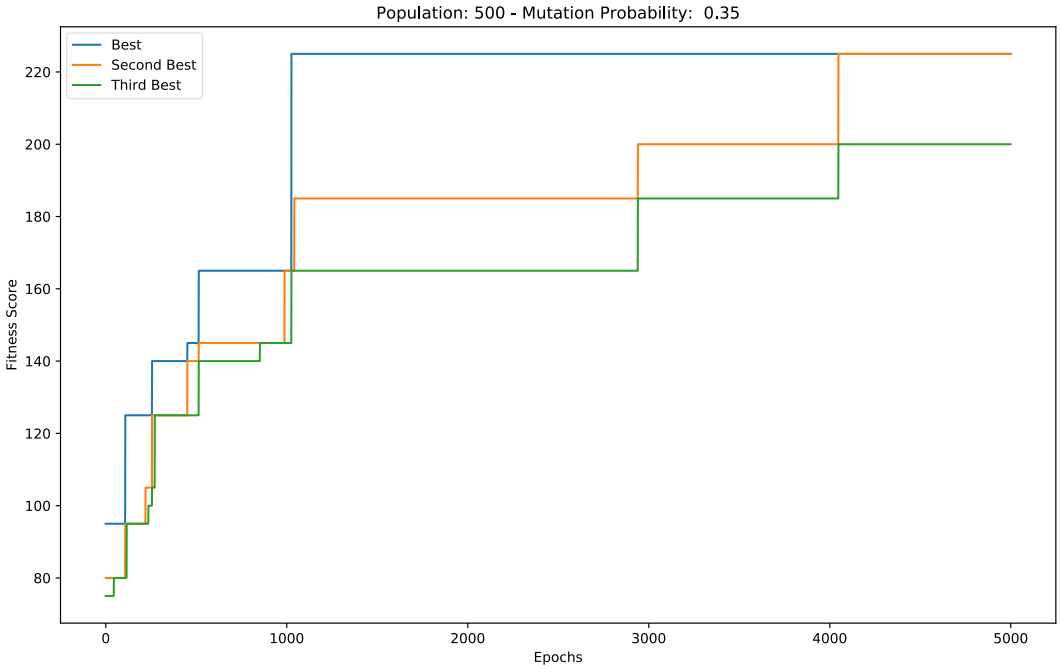

Fig. 20 Resultado con una población de 500 individuos

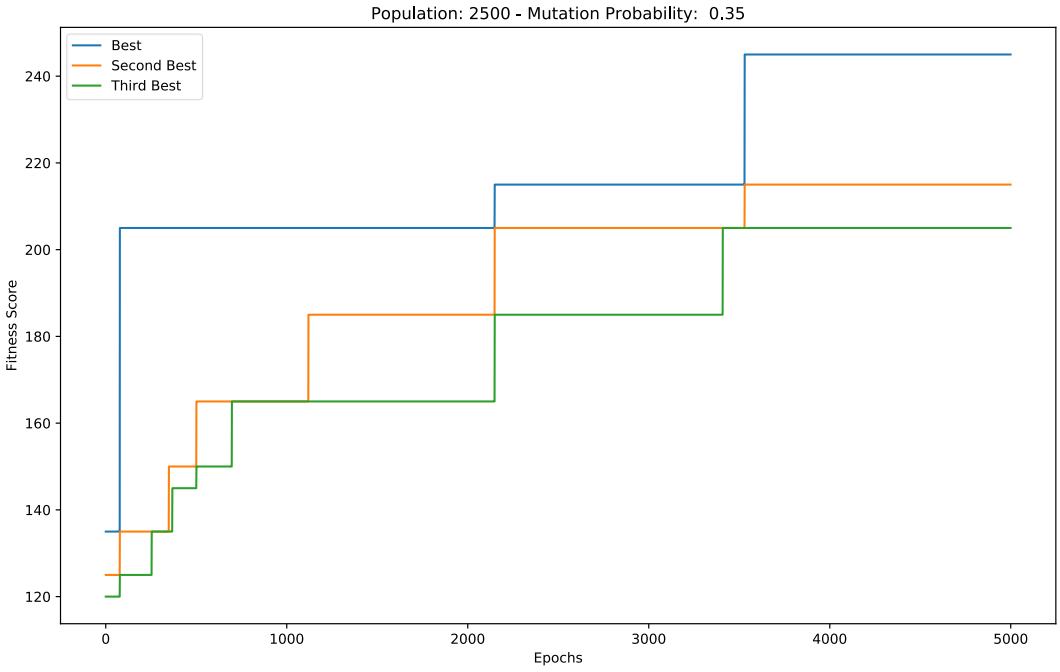

Fig. 21 Resultado con una población de 2500 individuos

5.1.2 Variación de Generaciones

- Variación de generaciones con probabilidad de 0.35 y población 250:

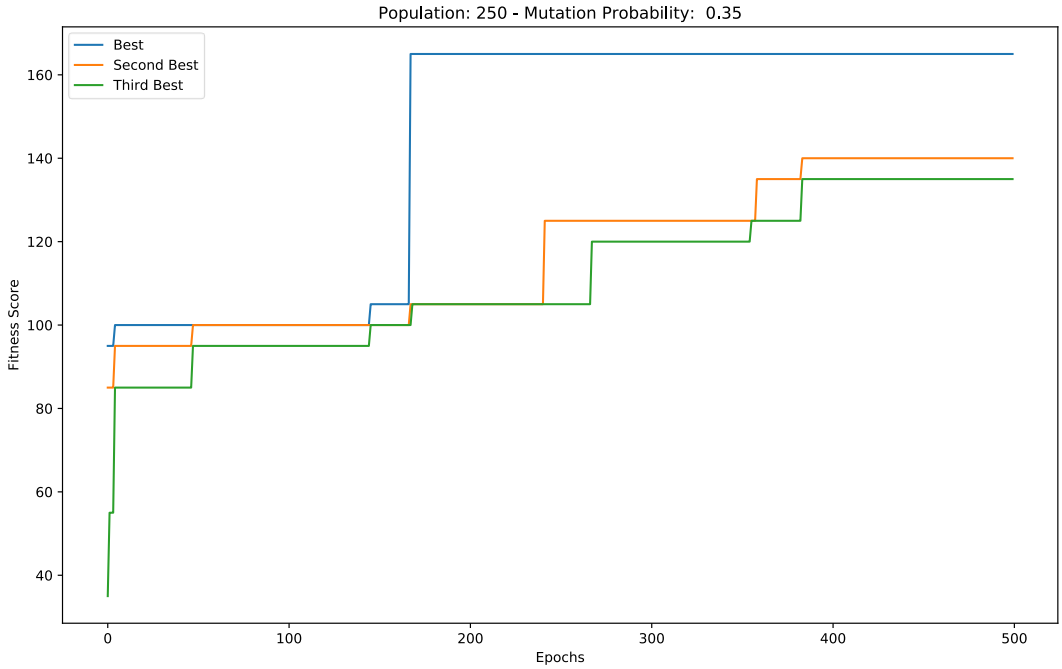

Fig. 22 Resultado de 500 generaciones

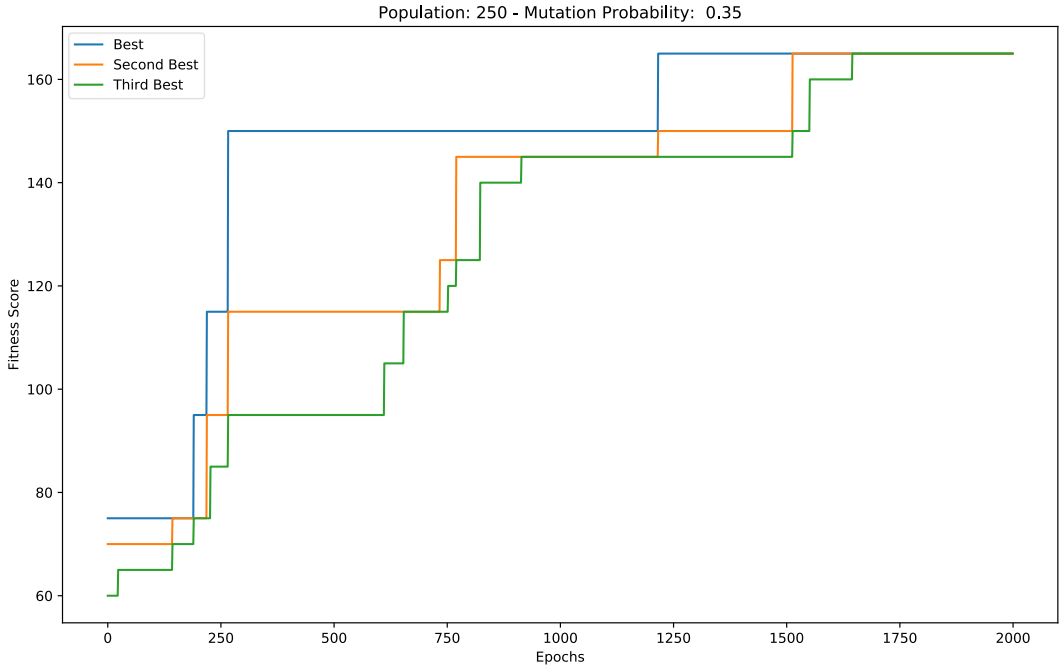

Fig. 23 Resultado de 2000 generaciones

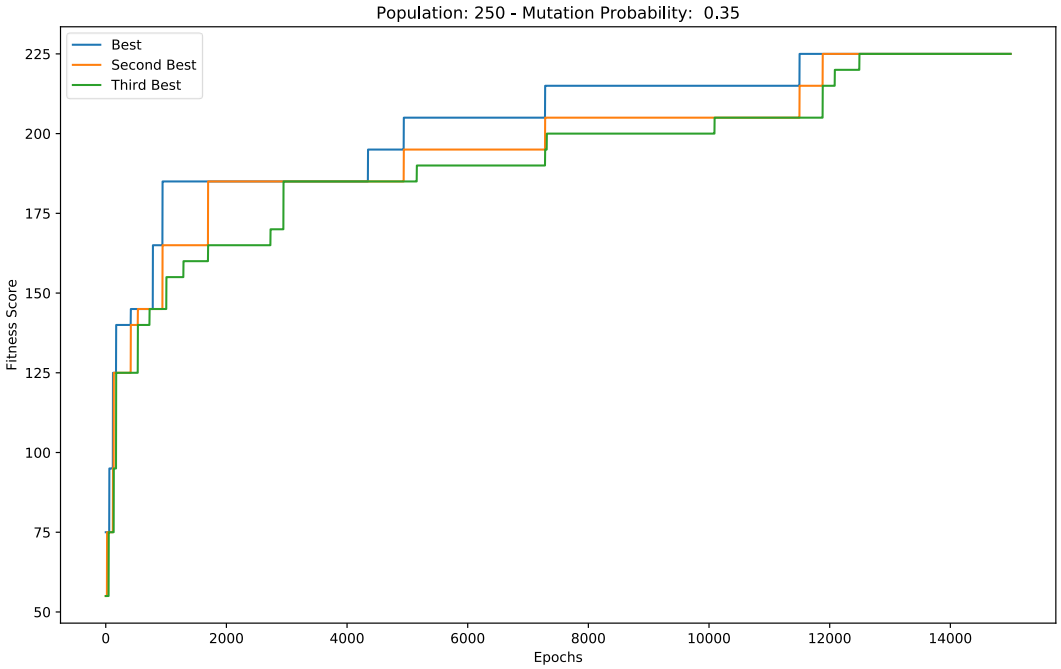

Fig. 24 Resultado de 15.000 generaciones

5.1.3 Variación de Probabilidades

- Variación de probabilidad con población 250 y 2000 generaciones:

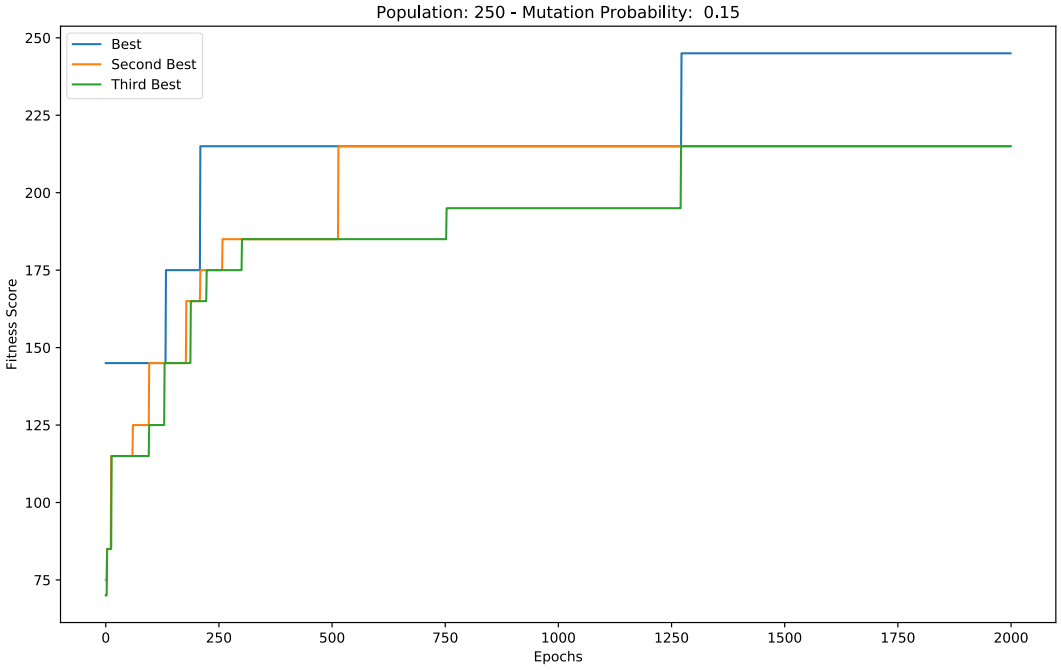

Fig. 25 Resultado con probabilidad de mutación de 0.15

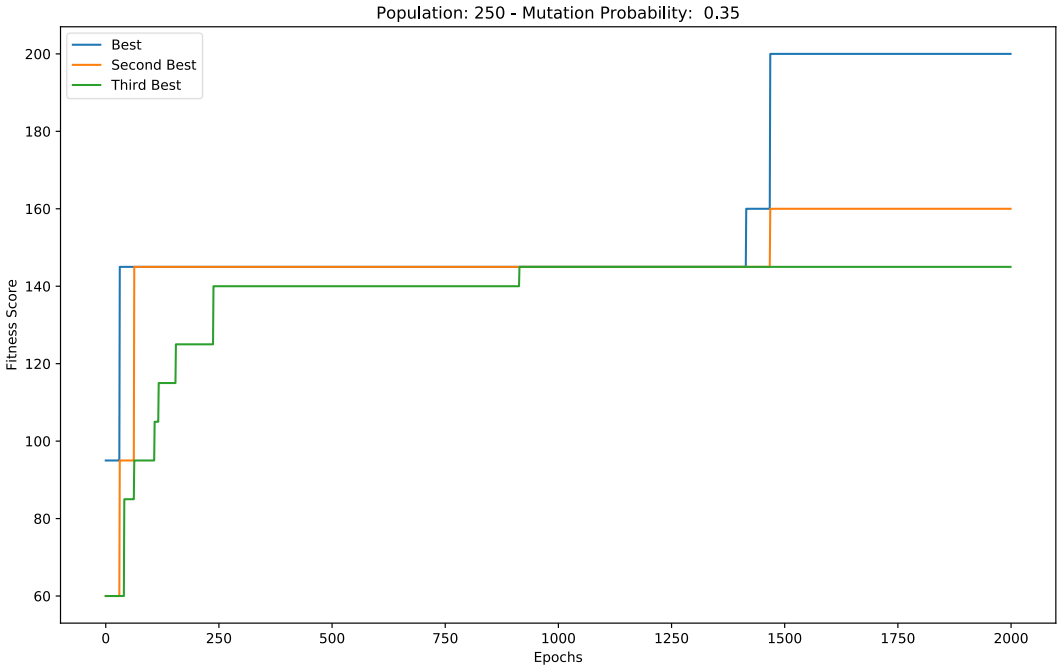

Fig. 26 Resultado con probabilidad de mutación de 0.35

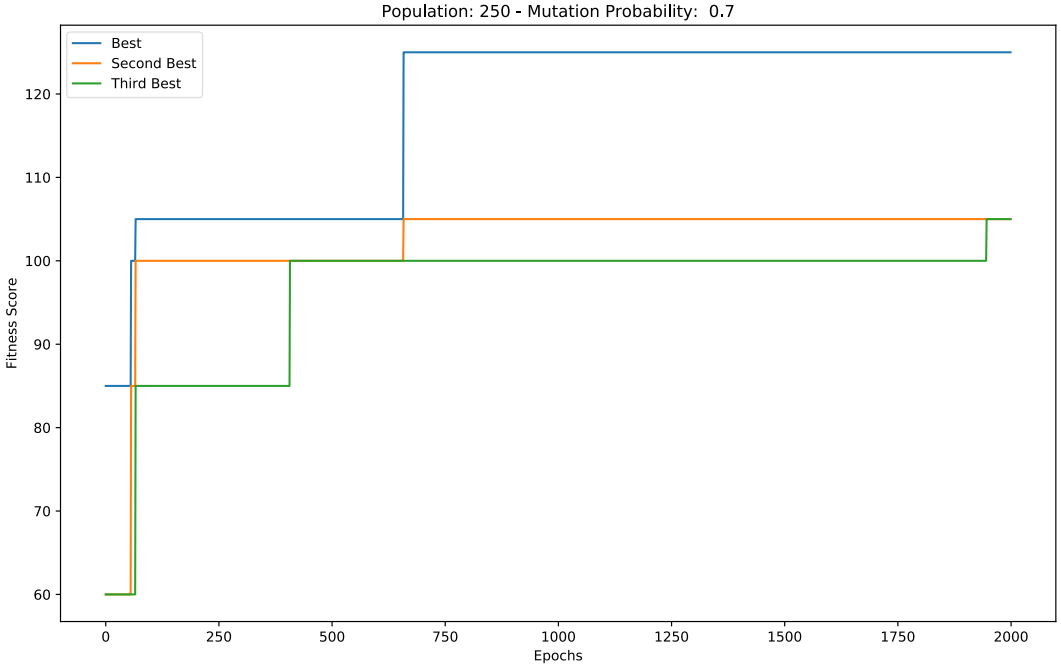

Fig. 27 Resultado con probabilidad de mutación de 0.7

Dependiendo de que parámetro clave se modifica, se puede observar que cuantas más generaciones pasan, los resultados obtienen una mejora en el *score* que obtienen, pero a cambio el tiempo de ejecución se incrementa exponencialmente, ya que al haber más individuos y dar más tiempo para que los individuos evolucionen adecuadamente. Aunque el elemento más crítico para obtener mejores resultados es el de mayor numero de generaciones por la que la población puede pasar, debido a que no es necesario tener un gran número de ritmos ya que todos buscan tener los rasgos musicales del *Funk*.

Por lo que respecta a la probabilidad de mutar, la variación afecta en el tiempo en que se obtiene un score con alto valor porque puede afectar a cualquier instrumento, eso quiere decir que, si una combinación de individuos previa a la mutación tiene un score alto, puede acabar teniendo uno bajo de este valor a causa de la mutación.

5.2 Ficheros MIDI

Una vez que se han conseguido los mejores ritmos, estos inician el proceso de conversión a MIDI. Con la finalidad de poder apreciar mejor el ritmo resultante, durante la conversión se duplica el número de notas que hay que introducir dando camino a un ritmo con el doble de duración de lo establecido.

Fig. 28 Resultados de conversión a fichero MIDI de resultados.[15]

En la figura anterior se puede observar cómo los resultados llegan a ser muy diferentes gracias al proceso evolutivo del algoritmo genético. A más, estos resultados pueden ser reproducidos tal y como se esperaba.

[15] Imágenes obtenidas a partir de la visualización de dos resultados en el programa *MidiEditor*: https://www.midieditor.org/

5.3 Partituras

Finalmente, los mejores resultados pasan a ser representados por partituras, como las que se pueden observar a continuación:

Best of 250 Individuals in 15000 Generations. Mutation: 35.0%

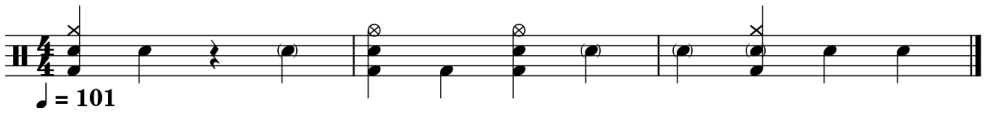

Fig. 29 Partitura del mejor individuo de una población de 250, evolucionada durante 15000 épocas y probabilidad de mutar de 0.35.

Best of 150 Individuals in 5000 Generations. Mutation: 20.0%

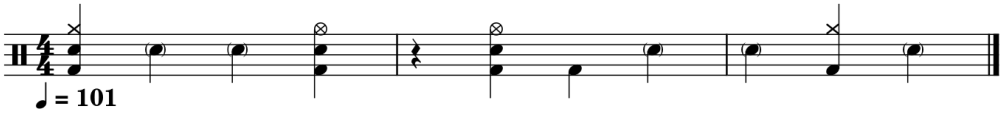

Fig. 30 Partitura del mejor resultado entre 150 individuos después de 5000 generaciones con 0.2 de probabilidad de mutar.

Best of 500 Individuals in 7000 Generations. Mutation: 60.0%

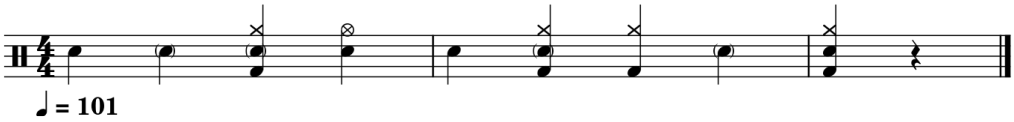

Fig. 31 Partitura del mejor individuo de entre 500 pasadas 7000 iteraciones con una probabilidad de mutar de 0.6

Se puede observar a simple vista que estos resultados difieren de las partituras normales, debido a que, en el momento de hacer la conversión, hay interferencias entre la inclusión simultanea de múltiples instrumentos y las corcheras, que causa que solo puedan hacerse una de las dos debido a como esta confeccionado el paquete de *music21*. Por consecuencia, se ha preferido poder introducir acordes correctamente. Otra de las cosas que se puede notar es la ausencia de los cuatro tiempos en las partituras. Esto se debe a que, al generar la partitura, el conversor no detecta bien las notas del final y genera una partitura con la ausencia de la parte del cuarto tiempo. Una alternativa a esta complicación seria hacer la conversión del MIDI resultante del anterior apartado a partitura, pero requiere que el MIDI este generado con el propio paquete de *music21* y realizarlo así es mucho más complejo porque se debe ir nota a nota cuidadosamente, sin poder usar un bucle *for* para recorrer todos los instrumentos.

6. Conclusiones

Para concluir este proyecto, se puede confirmar que el objetivo principal se ha podido cumplir con éxito, puesto que se ha podido desarrollar un algoritmo genético que genera ritmos lo más próximos al género musical del *Funk*, así como también se han podido exportar los resultados tanto en ficheros de formato MIDI como en partitura, tanto en formato de imagen *PNG* como también en fichero de tipo *XML*.

Se ha podido demostrar que trabajar con los algoritmos genéticos es una tarea que puede llegar a ser fascinante dependiendo del hábito en que se implemente, pues pueden llegar a hacer grandes proyectos si se dispone del tiempo y recursos para ello.

Cabe destacar que este proyecto tiene mucho más futuro si se profundiza en él, ya que las capacidades de mejora son vastas, por ejemplo, se podrían añadir nuevos instrumentos, nuevas condiciones a la función de *Fitness*, así como también dar pie a poder generar ritmos de otros estilos de música diferentes, no solo del *Funk*.

Pero no solo añadir nuevo contenido, sino afinar las condiciones actuales que residen ya como parte del código, así como también mejorar la parte de exportación de la partitura, ya que la librería de *music21* es muy extensa y contiene una gran variedad de funciones que no se han utilizado en este proyecto. Una manera óptima de solucionar este conflicto seria realizar el proyecto alrededor de este paquete de *Python*, ya que tiene una gran cantidad de funciones para cada situación puesto que es una herramienta que usan los músicos para elaborar partituras.

Referencias

[1] Dr. Robert Kübler - An extensible Evolutionary Algorithm Example in Python, consultado el 22 de setiembre de 2020: https://towardsdatascience.com/an-extensible-evolutionary-algorithm-example-in-python-7372c56a557b

[2] Computerphile - The Knapsack Problem & Genetic Algorithms, consultado el 13 de octubre de 2020: https://www.youtube.com/watch?v=MacVqujSXWE

[3] Yannick - Drum Sheet Music: How to Read & Write It (Including Drum Key) consultado el 13 de octubre de 2020: https://www.kickstartyourdrumming.com/drum-sheet-music/

[4] Sound Field - How James Brown Invented Funk, consultado el 13 de octubre de 2020: https://www.youtube.com/watch?v=AihgZv1D5-4

[5] Nahre Sol - Formula for a Funky Feel, consultado el 13 de octubre de 2020: https://www.youtube.com/watch?v=AXk6Bt0QrD0

[6] MATLAB - What is a Genetic Algorithm, consultado el 14 de febrero de 2021: https://www.youtube.com/watch?v=1i8muvzZkPw

[7] MIT OpenCourseWare, MIT Course by Instructor Patrick Winston - Learning: Genetic Algorithms, consultado el 14 de febrero de 2021: https://www.youtube.com/watch?v=kHyNqSnzP8Y

[8] Kie Codes - Genetic Algorithms Explained by Example, consultado el 15 de febrero de 2021: https://www.youtube.com/watch?v=uQj5UNhCPuo

[9] Vijini Mallawaarachchi - How to define a Fitness Function in a Genetic Algorithm? Consultado en marzo de 2021: https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4

[10] D. Beasley, D.R. Bull & R.R. Martin - An Overview of Genetic Algorithms, consultado en marzo de 2021: https://mat.uab.cat/~alseda/MasterOpt/Beasley93GA1.pdf

[11] Lima, José & Gracias, Nuno & Pereira, Henrique & Rosa, Agostinho. (1996). Fitness Function Design for Genetic Algorithms in Cost Evaluation Based Problems. 207-212. 10.1109/ICEC.1996.542362.

[12] Ahmed Gad (2018) – Genetic Algorithm Implementation in Python, consultado en marzo de 2021: https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6

[13] Erdem Isbilen (2020) – Abstract Base Classes in Python: Fundamentals for Data Scientists, consultado en marzo de 2021: https://towardsdatascience.com/abstract-base-classes-in-python-fundamentals-for-data-scientists-3c164803224b

[14] Huangwei Wieniaswska (2020) – Convert midi file to numpy array (Piano Roll), consultado en abril de 2021: https://medium.com/analytics-vidhya/convert-midi-file-to-numpy-array-in-python-7d00531890c

[15] Mark Conway Wirt - MIDIUtil documentation, consultado en abril de 2021: https://readthedocs.org/projects/midiutil/downloads/pdf/latest/

[16] Varvalen Pavani Neto (2020) – Editing MIDI files with Python article, consultado en mayo de 2021: https://dev.to/varlen/editing-midi-files-with-python-2m0g

[17] Music21 Documentation, , consultado en mayo de 2021: https://web.mit.edu/music21/doc/

[18] Rickey Vincent Funk article, consultado en mayo de 2021: https://www.britannica.com/art/funk

[19] Jake Frankenfield Artificial Intelligence article (2021), consultado en mayo de 2021: https://www.investopedia.com/terms/a/artificial-intelligence-ai.asp

[20] Built In AI article, consultado en mayo de 2021: https://builtin.com/artificial-intelligence

[21] School of Rock Drum notation for Beginners, consultado en mayo de 2021: https://www.schoolofrock.com/resources/drums/drum-notation-for-beginners#:~:text=Some%20of%20the%20most%20common,normally%20occupied%20by%20two%20notes.

[22] F. Gomez, A. Quesada & R. Lopez – Genetic Algorithms for Feature Selection, consultado en abril de 2021: https://www.neuraldesigner.com/blog/genetic_algorithms_for_feature_selection

[23] Shubham Jaim (2017) - Introduction to Genetic Algorithm & their application in data science, consultado en abril de 2021: https://www.analyticsvidhya.com/blog/2017/07/introduction-to-genetic-algorithm/

[24] John Gibson, Indiana University Bloomington, Jacobs School of Music – Introduction to the MIDI Standard, consultado en abril de 2021: https://cecm.indiana.edu/361/midi.html

[25] Coeur D'Alene Public Schools - Rhythm Chart, consultado en mayo de 2021: https://www.cdaschools.org/cms/lib/ID01906304/Centricity/Domain/2057/Week%205%20-%20BASS.pdf

[26] Zaritsky, Assaf & Sipper, Moshe. (2004). The Preservation of Favored Building Blocks in the Struggle for Fitness: The Puzzle Algorithm. Evolutionary Computation, IEEE Transactions on. 8. 443 - 455. 10.1109/TEVC.2004.831260.

Anexo

Registro numérico General MIDI de instrumentos de percusión:

MIDI Note Number	Note Type	MIDI Note Number	Note Type
27	Laser	60	High Bongo
28	Whip	61	Low Bongo
29	Scratch Push	62	Conga Dead Stroke
30	Scratch Pull	63	Conga
31	Stick Click	64	Tumba
32	Metronome Click	65	High Timbale
34	Metronome Bell	66	Low Timbale
35	Bass Drum	67	High Agogo
36	Kick Drum	68	Low Agogo
37	Snare Cross Stick	69	Cabasa
38	Snare Drum	70	Maracas
39	Hand Clap	71	Whistle Short
40	Electric Snare Drum	72	Whistle Long
41	Floor Tom 2	73	Guiro Short
42	Hi-Hat Closed	74	Guiro Long
43	Floor Tom 1	75	Claves
44	Hi-Hat Foot	76	High Woodblock
45	Low Tom	77	Low Woodblock
46	Hi-Hat Open	78	Cuica High
47	Low-Mid Tom	79	Cuica Low
48	High-Mid Tom	80	Triangle Mute
49	Crash Cymbal	81	Triangle Open
50	High Tom	82	Shaker
51	Ride Cymbal	83	Sleigh Bell
52	China Cymbal	84	Bell Tree
53	Ride Bell	85	Castanets
54	Tambourine	86	Surdu Dead Stroke
55	Splash cymbal	87	Surdu
56	Cowbell	91	Snare Drum Rod
57	Crash Cymbal 2	92	Ocean Drum
58	Vibraslap	93	Snare Drum Brush
59	Ride Cymbal 2		

Código Fuente del Proyecto:

```
from datetime import datetime
from abc import ABC, abstractmethod
from midiutil.MidiFile import MIDIFile
from itertools import groupby
import matplotlib.pyplot as plt
import numpy as np
import os

# -- imports para los exporters (XML & Partitura)

from music21 import *
from music21.converter.subConverters import ConverterMusicXML
from music21.chord import Chord

"""

Para que el exportar funcione hace falta configurar el music21 para que utilice estos dos
programas: Lilypond & MuseScore, para asi tener tanto los ficheros XML como la partitura en PDF

"""

us = environment.UserSettings()
us['lilypondPath'] = 'C:/Program Files (x86)/LilyPond/usr/bin/lilypond.exe'
us['musescoreDirectPNGPath'] = 'C:/Program Files/MuseScore 3/bin/MuseScore3.exe'
us['musicxmlPath'] = 'C:/Program Files/MuseScore 3/bin/MuseScore3.exe'

#----------------------------------------------------------------------------------------------

### ABSTRACT CLASS MODEL

class Individual(ABC):
    def __init__(self, value=None, init_params=None):
        if value is not None:
            self.value = value
        else:
            self.value = self._random_init(init_params)

    @abstractmethod
    def pair(self, other, pair_params):
        pass

    @abstractmethod
    def mutate(self, mutate_params):
        pass

    @abstractmethod
    def _random_init(self, init_params):
        pass

    ###
    #
    # En este caso el ritmo es el individuo y contiene 5 arrays de 16 elementos que son los instrumentos, que equivalen
a ser el value del individuo.
    #
    # Estos instrumentos son los siguientes: Closed Hi-hat, Bass drum, Snare drum, Open Hi-hat y Ghost notes (Snare)
    #
    # Los parametros que hay en pair y mutation son parametros definidos por el usuario para especificar como quiere
que se realicen las mutaciones
    # asi como las parejas de ritmos diferentes, pueden no haber parametros ya que pueden ser totalmente al azar o
predefinidos dentro del metodo.
    #
    ###

    class Rhythm(Individual):

        # Esta funcion crea un nuevo ritmo a partir de una parejas de ritmos ya creados. Se encarga de generar estos
ritmos mezclando
```

```
    # los instrumentos de cada uno, obteniendo el mejor de cada, para que el nuevo ritmo obtenga un indice de fitness más alto.
    # Toda esta función sigue los criterios de seleccion que también sigue la funcion fitness. (C1,C2,C3)

    def pair(self, other, pair_params):

        temp = np.array([np.zeros(16),np.zeros(16),np.zeros(16),np.zeros(16),np.zeros(16)])
        osti_cc = False
        osti_co = False

        # CHARLES CERRADO - OSTINATO - C1
        def ostinato_check(arr1,arr2,narr,osti):

            if (arr1 == 0).all():
                narr = arr2

            elif (arr2 == 0).all():
                narr = arr1

            else:

                if (arr1 != arr2).all():

                    if ((arr1[:4] == arr1[4:8]).all() and (arr1[:4] == arr1[8:12]).all()
                        and (arr1[:4] == arr1[12:16]).all()):
                            narr = arr1
                            osti = True

                    elif ((arr2[:4] == arr2[4:8]).all() and (arr2[:4] == arr2[8:12]).all()
                        and (arr2[:4] == arr2[12:16]).all()):
                            narr = arr2
                            osti = True
                    else:
                        if (all(item in arr1 for item in arr2)):
                            narr = arr2
                        elif (all(item in arr2 for item in arr1)):
                            narr = arr1
                        else:
                            narr = random.choice([arr1,arr2])

                else:
                    narr = arr1

        ostinato_check(self.value[0],other.value[0],temp[0],osti_cc)

        #BOMBO, SNARE, CHARLES ABIERTO

        # Esta funcion hace una comparacion de instrumentos entre los dos ritmos para decidir
        # cual de los dos instrumentos heredara el nuevo ritmo creado.

        def rhyt_check(arr1, arr2, check):
            narr = np.zeros(16)

            if (arr1 != 0).all() and (arr2 != 0).all():

                if check == 0: #C2

                    if all([arr1[0] == 1,arr1[1] == 0,arr1[2] == 0,arr1[3] == 0]):
                        if any([arr2[0] != 1,arr2[1] != 0,arr2[2] != 0,arr2[3] != 0]):
                            narr = arr1

                    if all([arr2[0] == 1,arr2[1] == 0,arr2[2] == 0,arr2[3] == 0]):
                        if any([arr1[0] != 1,arr1[1] != 0,arr1[2] != 0,arr1[3] != 0]):
                            narr = arr2

                    for i in range(len(arr1)):
                        if (arr1[i] == arr2[i]):
                            narr[i] = arr1[i]
```

```
                    if ((arr1[i] == 1 and arr2[i] != 1) or (arr1[i] != 1 and arr2[i] == 1)):
                        if (i in (0,4,8,12)):
                            narr[i] = 1

                else: #C3 - syncopation


                    if all([arr1[0] == 0,arr1[1] == 1,arr1[2] == 0,arr1[3] == 1,
                            arr1[4] == 0,arr1[5] == 1,arr1[6] == 0,arr1[7] == 1,
                            arr1[8] == 0,arr1[9] == 1,arr1[10] == 0,arr1[11] == 1,
                            arr1[12] == 0,arr1[13] == 1,arr1[14] == 0,arr1[15] == 1]):

                        if any([arr2[1] != 1,arr2[3] != 1,arr2[5] != 1,arr2[7] != 1,arr2[11] != 1,
                                arr2[13] != 1,arr2[15] != 1]):
                            narr = arr1

                    if all([arr2[0] == 0,arr2[1] == 1,arr2[2] == 0,arr2[3] == 1,
                            arr2[4] == 0,arr2[5] == 1,arr2[6] == 0,arr2[7] == 1,
                            arr2[8] == 0,arr2[9] == 1,arr2[10] == 0,arr2[11] == 1,
                            arr2[12] == 0,arr2[13] == 1,arr2[14] == 0,arr2[15] == 1]):

                        if any([arr1[1] != 1,arr1[3] != 1,arr1[5] != 1,arr1[7] != 1,arr1[11] != 1,
                                arr1[13] != 1,arr1[15] != 1]):
                            narr = arr2

                    for i in range(len(arr1)):
                        if (arr1[i] == arr2[i]):
                            narr[i] = arr1[i]

                        if ((arr1[i] == 1 and arr2[i] != 1) or (arr1[i] != 1 and arr2[i] == 1)):
                            if (i in (1,3,5,7,11,13,15)):
                                narr[i] = 1

                elif (arr1 == 0).all():
                    narr = arr2
                else:
                    narr = arr1

                return narr

            temp[1] = rhyt_check(self.value[1],other.value[1],0) #BOMBO
            temp[2] = rhyt_check(self.value[2],other.value[2],0) #SNARE

            if osti_cc: #the ostinato is located in the closed charles
                temp[3] = rhyt_check(self.value[3],other.value[3],0) #CHARLES ABIERTO
            else:
                ostinato_check(self.value[3],other.value[3],temp[3],osti_co)

            temp[4] = rhyt_check(self.value[4],other.value[4],1) #GHOST SNARE

            return Rhythm(temp)

        ###
        #
        # Esta funcion invierte las notas del ritmo en proporcion a la probabilidad de mutar, cuanto mas pequeña es esta proporcion,
        # menos notas se cambiaran, ya que funciona de la siguiente manera:
        #
        # notas a cambiar = longitud del track (16) * probabilidad de mutar
        #
        # Entonces si tenemos por ejemplo que la p=1/2, solo la mitad de las notas de cada instrumento se invirtiran aleatoriamente.
        #
        ###
        def mutate(self, mutate_params):

            inverse = int(len(self.value[0])*mutate_params['rate']) #todos los tracks comparten tamaño = 16

            for r in self.value:
```

```
            for i in range(inverse):
                choice = np.random.choice(range(0,len(r)))
                if (r[choice] == 0):
                    r[choice] = 1
                else:
                    r[choice] = 0

        # Funcion que genera un ritmo al azar con 5 instrumentos.

        def _random_init(self, init_params): #init params puede no ser nada tranquilamente pero esta vez va a ser el tamaño de la partitura, es decir 16

            # Esta funcion crea una array de 0s donde despues altera k numeros aleatoriamente
            def rand_bin_arr(N):
                arr = np.zeros(N)
                k = np.random.randint(0,N+1,dtype='int')
                arr[:k] = 1
                np.random.shuffle(arr)
                return arr

            bombo = rand_bin_arr(init_params)
            snare = rand_bin_arr(init_params)
            ghost= rand_bin_arr(init_params)
            charles_C = rand_bin_arr(init_params)
            charles_O = rand_bin_arr(init_params)

            rhyt = [charles_C, bombo, snare, charles_O, ghost]

            return rhyt

    class Population:
        def __init__(self, size, fitness, individual_class, init_params):
            self.fitness = fitness
            self.individuals = [individual_class(init_params=init_params) for _ in range(size)]
            self.individuals.sort(key=lambda x: self.fitness(x))

        def replace(self, new_individuals):
            size = len(self.individuals)
            self.individuals.extend(new_individuals)
            self.individuals.sort(key=lambda x: self.fitness(x))
            self.individuals = self.individuals[-size:]

        def get_parents(self, n_offsprings):
            mothers = self.individuals[-2 * n_offsprings::2]
            fathers = self.individuals[-2 * n_offsprings + 1::2]

            return mothers, fathers

    class Evolution:
        def __init__(self, pool_size, fitness, individual_class, n_offsprings, pair_params, mutate_params, init_params):
            self.pair_params = pair_params
            self.mutate_params = mutate_params
            self.pool = Population(pool_size, fitness, individual_class, init_params)
            self.n_offsprings = n_offsprings

        def step(self):
            mothers, fathers = self.pool.get_parents(self.n_offsprings)
            offsprings = []

            for mother, father in zip(mothers, fathers):
                offspring = mother.pair(father, self.pair_params)
                offspring.mutate(self.mutate_params)
                offsprings.append(offspring)

            self.pool.replace(offsprings)

    # opt = ritmo del que vamos a definir la fitness.

    """
```

```
    La fitness en este caso es definida por los siguientes atributos:
        C1 - tiene el charles cerrado tiene repeticion en los cuatro tiempos, (es decir los tiempos 1,2,3,4)
        C2 - o si la caja, el bombo o el charles con el bombo están presentes en el primer tiempo (1) y en (e&a) se ausentan
(0)
        C3 - o también si la caja (golpes fantasma) tiene presencia en los tiempos "e/a".
        C4 - La suma de los golpes de caja y charles abierto no puede superar 5.
        C5 - El charles abierto no tiene más de 4 apariciones por compás.
        C6 - Tres bombo seguidos son penalizados ya que no són muy funky.
        C7 - Si la caja tiene notas en los tiempos 2 & 4 y el ghost en el "e" & "a" asi como el bombo en el 1 tendriamos un
ritmo
            muy funky.
        C8 - Un ritmo es funky si de los cuatro tiempos que tiene un compás el 3 y el 4 es una respuesta al tiempo 1 y al
2.
            Esto es así cuando la frase formada por la parte rítmica (caja, charles abierto, bombo) de los tiempos 3 y 4
            es muy parecida y con instrumentos distintos. Funciona mejor entre el bombo y la caja.
        C9 - Un ritmo es funky si es sincopado. Esto ocurre, entre otras situaciones, cuando la distancia entre el bombo y
            a caja es de una negra con puntillo, es decir un tiempo y medio del compás (6 tiempos).

    Los valores que se calculan de esta fitness actualmente son de 0 si no cumple ninguna de las
    condiciones especificadas pero lo ideal es que los valores aproximados a este resultado tengan
    tambien un valor gradual de fitness para evitar que hayan 0s y asi aproximar un ritmo funk.
    """

    #
    #  dev comment: hay que checkear los elif para ver si siguen las caracteristicas planteadas
    #  y si son una buena aplicacion para definir la fitness de los ritmos. En el caso de que
    #  no sean convenientes, eliminar las nuevas condiciones para establecer unas correctas.
    #

    def fitness(opt):

        fitness_value = 0 #porcentaje

        charles_C, bombo, snare, charles_O, ghost = opt.value[0], opt.value[1], opt.value[2], opt.value[3], opt.value[4]

        def bombo_rep_check(arr):
            rep = [(k, sum(1 for i in g)) for k,g in groupby(arr)]

            for i in rep:
                if(i[0] == 1 and i[1] >= 3):
                    return True
                else:
                    return False

        # Esta función auxiliar comprueba que haya un espacio de 6 tiempos entre la caja y el bombo ya que esto indica
que
        # hay sincopacion entre estos dos.
        def syncoChck(arr,arr2):
            sync_bool = False

            for i in range(len(bombo)):

                if (arr[(i%16)] == 1).all() and (arr2[i%16:(i+6)%16] == 0).all():
                    if (arr2[(i+7)%16] == 1).all():
                        sync_bool = True
                elif (arr2[i] == 1).all() and (arr[i:(i+6)%16] == 0).all():
                    if (arr[(i+7)%16] == 1).all():
                        sync_bool = True

            return sync_bool


        # CHARLES CERRADO & OSTINATO (C1)


        if (np.any(charles_C == charles_O)):
            fitness_value -= 30

        elif (np.all(charles_C == 0)):
```

```
            fitness_value -= 30

        elif all([charles_C[0] == 1,charles_C[4] == 1,charles_C[8] == 1,charles_C[12] == 1]):
            fitness_value += 50

        elif any([charles_C[0] == 1,charles_C[4] == 1,charles_C[8] == 1,charles_C[12] == 1]):
            fitness_value += 25

        else:
            #and (charles_C[:4] == charles_C[8:12]).all() /  and (charles_O[:4] == charles_O[8:12]).all()
            if (((charles_C[:4] == charles_C[4:8]).all() and (charles_C[4:8] == charles_C[8:12]).all() and
                (charles_C[8:12] == charles_C[12:16]).all() and (charles_C[:4] == charles_C[8:12]).all()
                 and (charles_C[:4] == charles_C[12:16]).all())

            or ((charles_O[:4] == charles_O[4:8]).all() and (charles_O[4:8] == charles_O[8:12]).all()
                and (charles_O[8:12] == charles_O[12:16]).all()) and (charles_O[:4] == charles_O[8:12]).all()
                and (charles_O[:4] == charles_O[12:16]).all()):

                fitness_value += 100


        # SNARE (C2, C4 & C7)

        # snare tempo 1 presente
        if (np.count_nonzero(snare) + np.count_nonzero(charles_O) > 5): #C4
            fitness_value -= 20

        if (np.all(snare[:4] == np.array([1,0,0,0]))): #C2
            fitness_value += 40

        if all([snare[4] == 1, snare[8] == 1]): #C7
            fitness_value += 30

        elif (np.all(snare == 0)):
            fitness_value -= 20


        #GHOST SNARE (C3) - SYNCOPATION (1e&a == 0101) 1,3,5,7,9,11,13,15

        if (np.any(snare == ghost)):
            fitness_value -= 35

        elif all([ghost[1]==1,ghost[3]==1,ghost[5]==1,ghost[7]==1,ghost[9]==1,ghost[11]==1,ghost[13]==1,ghost[15]==1]):
            fitness_value += 40

        elif
    any([ghost[1]==1,ghost[3]==1,ghost[5]==1,ghost[7]==1,ghost[9]==1,ghost[11]==1,ghost[13]==1,ghost[15]==1]):
            fitness_value += 20

        elif (np.all(ghost == 0)):
            fitness_value -= 20


        # BOMBO (C2 & C6)

        if (np.all(bombo[:4] == np.array([1,0,0,0]))):
            fitness_value += 40

        if (bombo_rep_check(bombo)):
            fitness_value -= 20

        elif any([bombo[0] == 1,bombo[4] == 1,bombo[8] == 1,bombo[12] == 1]):
            fitness_value += 20

        elif (np.all(bombo == 0)):
            fitness_value -= 10


        #CHARLES ABIERTO - MELODY or OSTINATO (C2, C4 & C5)
```

```
        if (np.count_nonzero(charles_O) > 4): #C5
            fitness_value -= 25

        if (np.all(charles_O[:4] == np.array([1,0,0,0]))):
            fitness_value += 40

        elif any([charles_O[0] == 1,charles_O[4] == 1,charles_O[8] == 1,charles_O[12] == 1]):
            fitness_value += 20

        elif (np.all(charles_O == 0)):
            fitness_value -= 10

        # SUPERFUNK - C7

        if all([snare[4] == 1, snare[8] == 1,ghost[1]==1,ghost[3]==1,ghost[5]==1,
                ghost[7]==1, ghost[9]==1,ghost[11]==1,ghost[13]==1,ghost[15]==1,
                bombo[0] == 1,bombo[1] == 0,bombo[2] == 0,bombo[3] == 0]):
            fitness_value += 100

        # C8 - ANSWER

        if (bombo[0:7] == snare[8:15]).all() or (snare[0:7]==bombo[8:15]).all():
            fitness_value += 100
        elif (snare[0:7]==charles_O[8:15]).all() or (bombo[0:7]==charles_O[8:15]).all():
            fitness_value += 50
        elif (charles_O[0:7]==bombo[8:15]).all() or (charles_O[0:7]==snare[8:15]).all():
            fitness_value += 50

        # C9 - BOMBO-SNARE SYNCOPATION

        if (syncoChck(bombo,snare)):
            fitness_value += 50


        return fitness_value

    # Esta funcion se encarga de exportar los ritmos creados a un fichero reproducible de tipo MIDI. Recibe por parametros el ritmo
    # asi como el nombre del archivo a exportar. Solo crea una track, donde añadira cada instrumento con sus correspondientes notas.

    def midiExport(rhyt,name):

        if not os.path.exists("./MIDIs"):
            os.mkdir("./MIDIs")

        # degrees -> MIDI note number in order: closed hi-hat, bass, snare, open hi-hat, snare acoustic (ghost) (37/38)
        degrees  = [42, 36, 40, 46, 38]
        track    = 0
        channel  = 9   # Drums are the channel nº9
        time     = 0   # In beats
        duration = 1    # In beats
        tempo    = 114  # In BPM
        volume   = 100  # 0-127, as per the MIDI standard

        MyMIDI = MIDIFile(1)  # One track, defaults to format 1 (tempo track is created
                              # automatically)
        MyMIDI.addTempo(track, time, tempo)

        note = 0
        for r in rhyt.value:
            time = 0
            for i in range(len(r)*2):
                if r[i%16] == 1:
                    MyMIDI.addNote(track, channel, degrees[note], time + i/3, duration, volume)
                else:
                    time + i/3
            note += 1
```

```
        with open("./MIDIs/"+name+".mid", "wb") as output_file:
            MyMIDI.writeFile(output_file)

    # Funcion que se encarga de coger los ritmos y exportarlos de forma que puedan visualizarse en
    # partituras, asi como tambien en pdf. Para visualizar el fichero XML es necesario utilizar el
    # programa de MuseScore, en cambio, para el pdf primero hace falta ejecutarlo como fichero de
    # Lilypond, que este programa te genera el pdf con toda la partitura.

    def m2SheetExport(rhyt,fileName,title):
        save_path = "./mSheets/"+fileName

        if not os.path.exists("./mSheets"):
            os.mkdir("./mSheets")

        degrees  = ['G5','F4','C5','G5','C5']  # closed hi-hat, bass, snare, open hi-hat, snare acoustic(ghost)

        drums  =  stream.Part([clef.PercussionClef(),  meter.TimeSignature('4/4'),  tempo.MetronomeMark(number=101)])
#drumpart
        drums.metadata = metadata.Metadata()
        drums.metadata.title = title

        dMeasure = stream.Measure()

        for r0,r1,r2,r3,r4 in zip(rhyt.value[0],rhyt.value[1],rhyt.value[2],rhyt.value[3],rhyt.value[4]):
            # NOTES

            # Closed Hi-Hat
            n0 = note.Note(degrees[0])
            n0.notehead = 'x'
            n0.stemDirection = 'up'

            # Bass drum / kick
            n1 = note.Note(degrees[1])
            n1.stemDirection = 'down'

            # Snare
            n2 = note.Note(degrees[2])
            n2.stemDirection = 'down'

            # Open Hi-Hat
            n3 = note.Note(degrees[3])
            n3.notehead = 'circle-x'
            n3.stemDirection = 'down'

            # Snare ghost
            n4 = note.Note(degrees[4])
            n4.stemDirection = 'down'
            n4.noteheadParenthesis = True

            if r0 == r3:
                r0 = 0
            if r2 == 1:
                r4 = 0

            if all([r0==1,r1==1,r2==1,r3==1]): #1 1 1 1
                if (r4 == 1): dMeasure.append(Chord([n0,n1,n2,n3,n4]))
                else: dMeasure.append(Chord([n0,n1,n2,n3]))

            elif all([r0==1,r1==1,r2==1,r3==0]): # 1 1 1 0
                if (r4 == 1): dMeasure.append(Chord([n0,n1,n2,n4]))
                else: dMeasure.append(Chord([n0,n1,n2]))

            elif all([r0==1,r1==1,r2==0,r3==0]): # 1 1 0 0
                if (r4 == 1): dMeasure.append(Chord([n0,n1,n4]))
                else: dMeasure.append(Chord([n0,n1]))

            elif all([r0==1,r1==0,r2==1,r3==1]): # 1 0 1 1
```

```
            if (r4 == 1): dMeasure.append(Chord([n0,n2,n3,n4]))
            else: dMeasure.append(Chord([n0,n2,n3]))

        elif all([r0==1,r1==1,r2==0,r3==1]): # 1 1 0 1

            if (r4 == 1): dMeasure.append(Chord([n0,n1,n3,n4]))
            else: dMeasure.append(Chord([n0,n1,n3]))

        elif all([r0==0,r1==1,r2==1,r3==1]): # 0 1 1 1
            if (r4 == 1): dMeasure.append(Chord([n1,n2,n3,n4]))
            else: dMeasure.append(Chord([n1,n2,n3]))

        elif all([r0==0,r1==0,r2==1,r3==1]): # 0 0 1 1
            if (r4 == 1): dMeasure.append(Chord([n2,n3,n4]))
            else: dMeasure.append(Chord([n2,n3]))

        elif all([r0==0,r1==1,r2==1,r3==1]): # 0 1 1 0
            if (r4 == 1): dMeasure.append(Chord([n1,n2,n4]))
            else: dMeasure.append(Chord([n1,n2]))

        elif all([r0==0,r1==1,r2==1,r3==1]): # 0 1 0 1
            if (r4 == 1): dMeasure.append(Chord([n1,n3,n4]))
            else: dMeasure.append(Chord([n1,n3]))

        elif all([r0==1,r1==0,r2==0,r3==1]): # 1 0 0 1
            if (r4 == 1): dMeasure.append(Chord([n0,n3,n4]))
            else: dMeasure.append(Chord([n0,n3]))

        elif all([r0==1,r1==0,r2==0,r3==1]): # 1 0 1 0
            if (r4 == 1): dMeasure.append(Chord([n0,n2,n4]))
            else: dMeasure.append(Chord([n0,n2]))

        elif all([r0==0,r1==0,r2==0,r3==0,r4==0]): # 0 0 0 0 / 0
            dMeasure.append(note.Rest())


        elif all([r0 == 1,r1==0,r2==0,r3==0,r4==0]): # closed hi-hat
            dMeasure.append(n0)
        elif  all([r0 == 0,r1==1,r2==0,r3==0,r4==0]): # bass drum
            dMeasure.append(n1)
        elif  all([r0 == 0,r1==0,r2==1,r3==0,r4==0]): # snare drum
            dMeasure.append(n2)
        elif  all([r0 == 0,r1==0,r2==0,r3==1,r4==0]): # open hi-hat
            dMeasure.append(n3)
        elif all([r0 == 0,r1==0,r2==0,r3==0,r4==1]): # snare drum ghost
            dMeasure.append(n4)

    drums.append(dMeasure)

    # Partitura

    conv = converter.subConverters.ConverterLilypond()
    conv.write(drums.flat, fmt='lilypond',fp=save_path+fileName, subformats =['pdf'])

    # MusicXML

    conv_mxml = ConverterMusicXML()
    conv_mxml.write(drums.flat, fmt='musicxml',fp=save_path+'.xml', subformats =['png'])

# ----------------- MAIN DEL PROGRAMA ----------------------------- #

"""
pool_size -> max num individuos
n_offsprings -> "hijos" que se generan por cada pareja
pair, mutate & init params -> parametros que se pueden modificar para cambiar el programa.
epochs -> numero de veces que se generan nuevos individuos.

En este caso, el main hace las generaciones indicadas con los individuos deseados, asi como también una
probabilidad de mutar indicada por parametro. Una vez ha hecho las generaciones, exporta los tres ultimos ritmos
```

```
    obtenidos, que corresponden a los tres mejores ritmos segun la fitness de cada uno, a formato MIDI. Finalmente,
    muestra por pantalla estos ritmos y su fitness correspondiente.

    """

    # Funcion auxiliar del Main que se encarga de generar un grafico de evolucion de los tres mejores individuos.
    # El grafico es guardado en una carpeta que crea la propia funcion en formato svg.

    def data_collection(flist1,flist2,flist3, pool, epochs, probability):

        if not os.path.exists("./graphs"):
            os.mkdir("./graphs")

        # Draw Plot
        plt.figure(figsize=(13,8))
        plt.plot(flist1)
        plt.plot(flist2)
        plt.plot(flist3)

        plt.title("Population: "+str(pool)+" - Mutation Probability:  "+str(probability))
        plt.legend(['Best', 'Second Best', 'Third Best'])

        plt.ylabel('Fitness Score')
        plt.xlabel('Epochs')

        plt.savefig("./graphs/results -"+str(epochs)+"_"+str(pool)+"_"+str(probability)+".svg",format='svg')

    def main(psize,epo,mut_prob):

        evo = Evolution(
            pool_size = psize , fitness=fitness, individual_class=Rhythm, n_offsprings=3,
            pair_params = {'one_zero' : [1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0], 'four_tempo' : [1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0]},
            mutate_params = {'rate' : mut_prob},
            init_params = 16
        )
        n_epochs = epo

        start = datetime.now()
        best_evo = []
        best2_evo = []
        best3_evo = []
        for i in range(n_epochs):
            best_evo.append(fitness(evo.pool.individuals[-1]))
            best2_evo.append(fitness(evo.pool.individuals[-2]))
            best3_evo.append(fitness(evo.pool.individuals[-3]))
            evo.step()

        end = datetime.now()

        best = evo.pool.individuals[-1]
        best2 =evo.pool.individuals[-2]
        best3 =evo.pool.individuals[-3]

        midiExport(best,"best")
        midiExport(best2,"best2")
        midiExport(best3,"best3")

        m2SheetExport(best,"Best", "Best of "+ str(psize)+" Individuals in " + str(epo)+" Generations. Mutation:
"+str(mut_prob*100)+'%')

        print("Time Elapsed: {}" .format(end-start))
        print("\nMutation Probability:", mut_prob)
        print("\nMejor Ritmo - Epoch:",i+1," Pool:",len(evo.pool.individuals)," \n",
              "\n charles", best.value[0],
              "\n bombo", best.value[1],
              "\n snare", best.value[2],
              "\n ghost snare",best.value[4],
              "\n charles open",best.value[3])
        print("\n fitness score = ",fitness(best))
```

```
print("-----------------------------------------")

print("\nSegundo Mejor Ritmo - Epoch:",i+1," Pool:",len(evo.pool.individuals)," \n",
      "\n charles", best2.value[0],
      "\n bombo", best2.value[1],
      "\n snare", best2.value[2],
      "\n ghost snare",best2.value[4],
      "\n charles open",best2.value[3])
print("\n fitness score = ",fitness(best2))
print("-----------------------------------------")

print("\nTercer Mejor Ritmo - Epoch:",i+1," Pool:",len(evo.pool.individuals)," \n",
      "\n charles", best3.value[0],
      "\n bombo", best3.value[1],
      "\n snare", best3.value[2],
      "\n ghost snare",best3.value[4],
      "\n charles open",best3.value[3])
print("\n fitness score = ",fitness(best3))
print("-----------------------------------------")

data_collection(best_evo,best2_evo,best3_evo,psize,epo,mut_prob)
```