POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Implementación de una biblioteca de metaheurísticas
avanzadas en Python

Autor: Miguel A. Marcos Pérez

Director: Antonio LaTorre de la Fuente

MADRID, JULIO 2017

The computer was born to solve
problems that did not exist before

Bill Gates

La composición de este documento se ha realizado con LaTeX.

Agradecimientos

A mis padres, gracias por la paciencia.
A mi hermana, gracias por aguantarme.
A mi familia, gracias por todo.
A mis amigos, gracias por estar ahí siempre.
A mis profesores, en especial a mi Tutor.
Gracias por todo el apoyo recibido durante estos años.

Índice

1. Resumen **1**

2. Introducción **3**

2.1. ¿Qué es un Algoritmo Evolutivo? 3
2.2. Tipos de Algoritmos Evolutivos básicos 4
2.3. Uso actual de los Algoritmos Evolutivos 4
2.4. Futuro de los Algoritmos Evolutivos 5

3. Objetivos **7**

4. Trabajos previos **9**

4.1. Introducción 9
4.2. Soft Computing 10
4.3. Algoritmos Genéticos 10
4.4. Evolución Diferencial 13
4.5. Búsqueda local 14
4.6. MOS 14
4.7. Conclusión 15

5. Desarrollo **17**

5.1. Introducción 17
5.2. Análisis y diseño 17
5.3. Implementación de MOS 19
5.3.1. Diseño de MOS 19
5.3.2. Clase MOSParticipationFunction 20
5.3.3. Clase MOSQualityFunction 21
5.4. Implementación algoritmos 22
5.4.1. Clase Algorithm 22
5.4.2. Clase Genetic Algorithm 23
5.4.3. Clase Local Search 24
5.4.4. Clase Differential Evolution 26
5.5. Implementación de clases contenedores 27
5.5.1. Clase Genome 27
5.5.2. Clase GAD1ArrayGenome 28
5.5.3. Clase Population 28
5.6. Implementación de operadores 29
5.6.1. Clase Crossover 29
5.6.2. Clase Mutator 30
5.6.3. Clase Selector 31

5.6.4. Clase Elitism . . . 32
5.7. Implementación de problemas . . . 33
5.7.1. Clase Problem . . . 33
5.8. Fase final y mejora . . . 34

6. Resultados y estadísticas **37**
6.1. Metodología del análisis de resultados . . . 37
6.2. Estadísticas . . . 38

7. Conclusión y trabajos futuros **45**
7.1. Conclusiones . . . 45
7.2. Trabajos futuros . . . 46

8. Anexo **47**

Índice de figuras

1. Antena diseñada mediante Algoritmos Evolutivos 4
2. Problema Card Pole [1] 6
3. Diagrama Metaheurísticas 11
4. Diagrama One-point crossover 11
5. Variantes DE [2] 13
6. Esquema general MOS [3] 15
7. Diagrama MOS 19
8. Implementación del bucle MOS 20
9. Ejemplo de participación dinámica 21
10. Ejemplo de problema de minimización con valor óptimo de -450 22
11. Clase Algorithm 22
12. Herencia en clases Algorithm 23
13. Clase Genetic Algorithm 24
14. Clase Genome 27
15. Clase GA1DArrayGenome 28
16. Clase Population 29
17. Clase OnePointCrossover 30
18. Clase FlipMutator 31
19. Clase TournamentSelector 32
20. Clase PopulationElitism 33
21. Clase TestProblem 34
22. Gráfica de Error medio por dimensión 40
23. Gráfica del tiempo de ejecución medio por dimensión 43

Índice de algoritmos

1. Algoritmo Genético 12
2. Método Update de MOSParticipationFunction 20
3. Método Evolve de GA 25
4. Operador MTS_LS1 26
5. Algoritmo DE 27
6. Método OnePointCrossover 30
7. Método FlipMutator 31
8. Método TournamentSelector 32
9. Método PopulationElitsm 33

1. Resumen

Resumen

Este proyecto consiste en la traducción de un código ya existente codificado en C++ a Python. El motivo de la necesidad de realizar este trabajo es que con el paso del tiempo y las numerosas manos que han estado involucradas en el código antiguo este se había vuelto inestable y añadirle funcionalidad se había convertido en un trabajo de grandes dimensiones y complejidad alta. Gracias a esta traducción se pretende que futuras modificaciones a este algoritmo se puedan realizar de una manera más rápida y sencilla. El lenguaje de destino que se ha elegido es Python, considerado por muchos como el lenguaje con más proyección a corto plazo.
La base de Python es su sintaxis clara y fácil de entender, dentro de un escenario de programación multiparadigma entre los que cabe destacar la programación orientada a objetos que será una de las aproximaciones más utilizados en este proyecto. Una garantía que ofrece Python es el término *"Batteries included"* dado que es un lenguaje que contiene de base multitud de funciones de gran utilidad y de esta forma se evita el uso de librerías externas.

Abstract

This project consists in the translation of an existing C++ code into Python. The reason to perform this work is because the old code has become unstable due to all the modifications that have been made to it, and, currently, adding functionality to this code means a lot of additional work. Thanks to this new implementation, future modifications to this algorithm can be conducted in a faster and simpler way. The target programming language that has been chosen is Python, considered by many as the language with the best short-term projection.
The basis of Python is its clear and easy to understand syntax, in a scenario of multi-paradigm programming where our algorithm could be implemented in an object-oriented way, that will be the most used approach within this project. A guarantee offered by Python is the *"Batteries included"* concept, on which the development of this project has been based, used to get the whole algorithm implemented without using any external libraries and obtaining all of the functionality that is needed for this project.

2. Introducción

La búsqueda de soluciones a problemas del mundo real habitualmente no son triviales como para poder ser modelizados mediante la resolución de problemas convencionales. Es por eso que es necesario buscar nuevas formas de resolución de problemas que permitan optimizar funciones complejas que carecen de métodos matemáticos para su resolución. En este campo los Algoritmos Evolutivos han mostrado ser especialmente adecuados para solventar este problema.

2.1. ¿Qué es un Algoritmo Evolutivo?

Un Algoritmo Evolutivo es un método de optimización, especialmente útil al resolver problemas de tipo técnico que son muy complicados de resolver utilizando técnicas tradicionales de optimización. Estos algoritmos están basado en la evolución biológica de los seres vivos, proceso natural en el que una población, al reproducirse, puede presentar cambios genéticos en la nueva generación.

Esta es la idea principal en la que se basó John Holland, en el año 1960 [4], para diseñar las estrategias evolutivas proponiendo utilizar los procesos naturales de selección y supervivencia en el ámbito de la resolución de problemas. Al principio se nombró a esta metodología como técnica de optimización estocástica , con el tiempo el nombre pasó a ser Algoritmos Evolutivos. En este contexto, surgieron una serie de investigaciones relacionadas, como son: los Algoritmos Genéticos, la Programación Genética, etc.
Dentro de los Algoritmos Evolutivos se trabaja con poblaciones de individuos, siendo cada uno de estos una posible solución al problema a optimizar. A esta población se la somete a un bucle de iteraciones evolutivas en las que se realizan principalmente los procesos de cruce y mutación junto con un proceso de selección.
La población entrante primeramente es analizada y de esta se obtiene un conjunto de individuos considerados los mejores para resolver el problema de optimización. Una vez se tiene el conjunto de mejores individuos, es cuando se producen las dos operaciones antes citadas, teniendo una población de salida con el mismo tamaño que la anterior en la que los individuos resultantes, idealmente, son mejores comparados con la generación anterior.
Una vez la población en su análisis inicial cumple una condición de parada prefijada, el bucle evolutivo se detiene y el resultado de este se considera una población de soluciones aceptadas y que optimizan la función deseada.

Dentro de este campo cabe destacar los siguientes algoritmos dado que son los que hemos utilizado en el TFG:

- Algoritmo genético

- Evolución diferencial

2.2. Tipos de Algoritmos Evolutivos básicos

Dentro del campo de los *EAs* cabe destacar los siguientes tipos de algoritmos:

- **Algoritmos Genéticos(GAs):** algoritmos que tienen de entrada una población inicial en la que se realizan una serie de operaciones obteniendo de este modo una población resultante mejor que la inicial. Este tipo de algoritmos son los que hemos utilizado en la elaboración de este TFG [4].

- **Programación Genética:** Especialización de los Algoritmos Genéticos en la que cada individuo es un programa de ordenador que es objeto de la evolución por parte del algoritmo [5].

- **Programación Evolutiva:** Variación de un Algoritmo Genético en la que los individuos son representados como un conjunto de tres valores que representan un autómata [6].

- **Estrategias de Evolución:** Algoritmo parecido a los *GAs* pero diferenciándose en el operador de mutación que obtiene las cifras aleatorias siguiendo una distribución normal que evoluciona al mismo tiempo que los individuos [7].

2.3. Uso actual de los Algoritmos Evolutivos

Hoy en día la aplicación de estos algoritmos se da en numerosos campos dada la gran utilidad que estos poseen en la resolución de problemas multivariable. Su aplicación se da tanto en el campo de la Medicina, con sistemas de soporte de decisión en oftalmología [8] y oncología [9], como en el campo del desarrollo informático, con sistemas de búsqueda de *BUGs* [10]. Actualmente las áreas en las que más énfasis sé esta dando son:

Figura 1: Antena diseñada mediante Algoritmos Evolutivos

- **Aprendizaje automático**: Una de las áreas que más esta creciendo en la actualidad debido a la necesidad de sistemas inteligentes que posean una alta complejidad. Dentro de este campo cabe destacar la fusión entre redes neuronales junto con EAs a la que se le llama neuroevolución [11].

- **Predicción del mundo financiero**: Al ser el mundo de la bolsa uno de los que más dinero mueve en la actualidad, es obvio pensar la necesidad de prever acontecimientos que se dan en entornos financieros, esto puede ayudar a los usuarios a realizar previsiones mucho más avanzadas de lo que serían sin utilizar ningún tipo de software [12].

- **Diseño automatizado**: Uno de los usos que se le puede dar a los algoritmos evolutivos, como podemos ver en la figura 1, es optimizar el diseño de componentes para que estos sean más eficientes en el tipo de problema para los que están pensados. El caso de la figura 1 es el de una antena utilizada en un satélite de la $NASA$ que fue optimizada por medio de algoritmos genéticos[13].

- **Seguridad informática**: El mundo de la seguridad informática es un mundo en continua extensión, lo que provoca que sea necesaria la implementación de antivirus heurísticos, los cuales poseen características especiales que les dotan de un cierto aprendizaje automatizado. Estos pueden reconocer patrones similares a los que son empleados por virus y, de esta forma, bloquear ejecuciones que pueden poner en riesgo al sistema [14].

- **Análisis lingüístico**: Gracias a los avances en este campo se ha podido llegar a implementar un software de corrección gramatical basada en algoritmos evolutivos. *"Probabilistic context free grammars"*(PCFGs) es el nombre que se le ha dado a este tipo de software[15].

2.4. Futuro de los Algoritmos Evolutivos

Dentro de los actuales usos de los Algoritmos Evolutivos y su futura mejora y expansión, cabe destacar diferentes focos de investigación y su actual y futura repercusión.

Uno de estos focos es el aprendizaje automatizado, gracias al cual un programa es capaz de, sin conocer el medio o su funcionalidad final, mejorar en cada ronda evolutiva de tal forma que acabe optimizando su funcionamiento hasta tal punto que se considere valido dentro de la problemática del medio que se está probando, un problema que ha tenido mucho recorrido en este campo es '*THE CART POLE PROBLEM*' en el cual el sistema debe mantener una especie de objeto erguido sin que se caiga moviendo para ello una especie de coche que se encuentra en la parte inferior, como podemos observar en la figura 2.

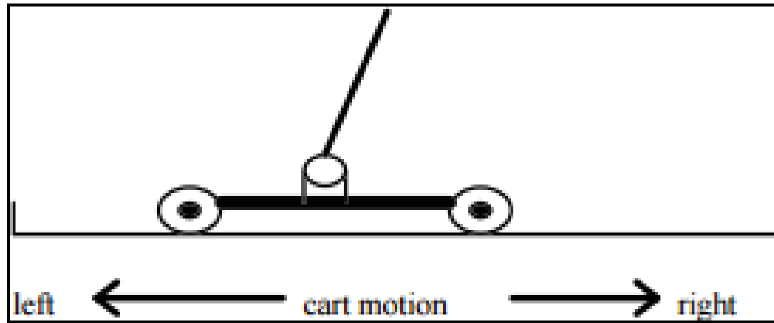


Figura 2: Problema Card Pole [1]

Uno de los aspectos de mejora más importantes de los Algoritmos Evolutivos es el tiempo de ejecución [16] dado que este se eleva de manera exponencial con la complejidad del problema. Hay numerosos problemas [1], como el problema llamado "*travelling salesperson problem*"que es uno de los más analizados en este campo, aunque tambien hay problemas mucho más prácticos como resolver planes de estrategia, horarios complejos, etc...

Cabe destacar que una de las heurísticas más utilizadas en el campo del aprendizaje automático es la neuro-evolución que es la fusión de los algoritmos genéticos con las redes neuronales con el fin de que redes de neuronas puedan ser modificadas en tiempo de ejecución para así poder optimizar el problema que se desea resolver [17].
En Internet hay numerosos vídeos que muestran de forma visual y didáctica el uso que se le da a este tipo de algoritmos [18].

El campo del soft computing es cada día más prometedor y el desarrollo e investigación de sistemas biológicos, como los sistemas de enjambres, abren un abanico infinito de posibilidades entre las que podemos destacar la conducción automática de vehículos o el manejo simultaneo de múltiples vehículos aéreos [19].

3. Objetivos

Principalmente los objetivos de este proyecto son el diseño e implementación de un framework, llamado MOS, partiendo como base de un código anterior en C++.
El código resultante debe proporcionar una serie de características, en mayor parte de carácter heurístico, que el anterior código ya no cumplía debido a sus continuas modificaciones y cuya implementación podría volverse demasiado costosa.

El código resultante de este proyecto debe contemplar la futura modificación de este, por lo que debe incluir gran número de comentarios que fomenten la legibilidad del código y facilite las futuras mejoras que se realicen en él. Para ello, el diseño debe ser muy particionado con el fin de que cualquier tipo de modificación que se realice en este no suponga un aumento de la complejidad del mismo.

Un aspecto importante que se debe tener en cuenta en todo tipo de algoritmos de optimización es el tiempo de ejecución. Cuanto menor sea, mayor usabilidad otorgará a nuestra aplicación y garantizará su uso futuro. Es un punto clave del proyecto y para su cumplimiento será necesaria una gran carga de esfuerzo. Para ello serán utilizadas técnicas de optimización de software a través de $profilers$.

Para posibilitar su uso a investigadores que no tengan un amplio conocimiento técnico acerca de cómo poder ejecutar el programa, un objetivo secundario será facilitar su uso de manera que únicamente sea necesario modificar un fichero de configuración en el que se contienen todos los parámetros necesarios para la ejecución del programa.

Si hablo de objetivos académicos, el principal es mi evolución a la hora de resolver un proyecto de gran tamaño por mí mismo, pudiendo buscar soluciones integrales y aprender de los errores de diseño que haya podido cometer. Una de las características más importantes de este proyecto es el tamaño del sistema a implementar lo que suponer un reto personal.

Otro objetivo de gran importancia es la ampliación de conocimiento en el ámbito de la inteligencia artificial prestando gran interés a los Algoritmos Evolutivos que es el tema principal que expongo en mi TFG.

4. Trabajos previos

En este capítulo se hablará del estado actual de los campos tratados en este *TFG* como son el *Soft computing* y los *Algoritmos Geneticos*, comenzando por una descripción de la Inteligencia Artificial.

4.1. Introducción

Para empezar a hablar sobre inteligencia artificial, primero debemos saber lo que este término significa. Según la RAE esta es la definición de IA,

> *Disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o razonamiento lógico.*

Como podemos observar la IA es una aproximación del razonamiento humano al mundo de la computación. Dentro de este gran campo cabe destacar una diferenciación que se realiza y separa a las máquinas según la forma que estas tienen de aproximarse a nuestra forma de razonar. Según Stuart Russell y Peter Norvig [20] estos son los grupos dentro de la Inteligencia Artificial:

El primer grupo es el que realiza la aproximación en la forma de pensar, tomar decisiones, resolver problemas y aprender. Un algoritmo importante dentro de este campo es el que se trata en este proyecto, es decir, los Algoritmos Genéticos. Otro ejemplo son las redes neuronales, diseñadas para, teniendo unos parámetros de entrada obtener una solución. La principal característica de este tipo de algoritmos es su aprendizaje y por tanto la mejora de soluciones que devuelven.
El segundo grupo tiene como aproximación la manera de actuar. Dentro de este conjunto tenemos la robótica, cuya misión es que máquinas actúen de manera similar al ser humano o cualquier tipo de ser vivo.
El tercer conjunto engloba los algoritmos basados en percepción y razonamiento, como son los sistemas expertos utilizados para monitorizar, diseñar, planificar, controlar y simular diferentes campos actuales de investigación como por ejemplo la medicina o el diseño molecular.
En el cuarto grupo se encuentran los sistemas inteligentes que pretenden asemejarse al raciocinio del ser humano. Un ejemplo dentro de esta categoría son los agentes inteligentes que son entidades que hacen una percepción del entorno y basándonos en este realizan una acción racional que consideran correcta.

En esta memoria nos centraremos en el primero de estos grupos con el fin de profundizar en el campo de algoritmos genéticos.

4.2. Soft Computing

El término '*Soft Computing*' se le da a una de las ramas del campo de la inteligencia artificial que se dedica a resolver problemas sin un algoritmo de resolución conocido que manejan datos incompletos o con incertidumbre.

Dentro de este campo se pueden destacar las siguientes técnicas, destacando los Algoritmos Evolutivos utilizados en este TFG:

- Machine learning
  - Redes neuronales
  - Máquinas de vectores de soporte (SVM)
- Sistemas Fuzzy
- Computación Bio-inspirada
  - Algoritmos evolutivos
  - Colonias de hormigas
  - Técnicas de enjambres
- Redes bayesianas
- Teoría del caos

Esta definición nos ayuda, como pretexto, a la siguiente sección en la que se hablará del grupo que contiene a los Algoritmos Genéticos, como es la computación Bio-inspirada. Este conjunto se encuentra dentro de un grupo de algoritmos determinados como Metaheurísticas, que es un conjunto de algoritmos(Figura 3) utilizados para resolver problemas que no tienen una forma convencional de ser resueltos.

4.3. Algoritmos Genéticos

Los algoritmos genéticos (GA) son unas técnicas de optimización basadas en las leyes biológicas de selección natural. Propuestos en 1975 por John Henry Holland[4], son actualmente una de las vertientes más estudiadas dentro de los algoritmos de optimización.
Este algoritmo es el resultado de la abstracción de un proceso biológico natural, como es la evolución, a una serie de heurísticas que se describirán posteriormente.
Un GA se caracteriza porque dentro de él se realizan tres operaciones, principalmente, a su vez abstraídas del anterior citado proceso evolutivo natural:

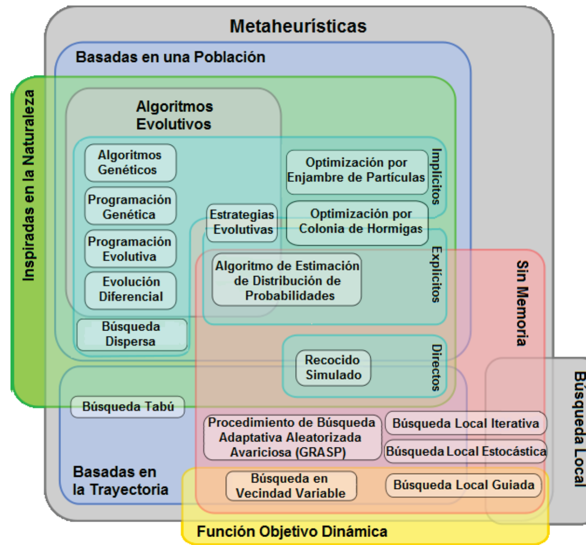


Figura 3: Diagrama Metaheurísticas

- **Selección:** Operador encargado de seleccionar individuos, siguiendo para ello procesos estocásticos, a los que evolucionar mediante los diferentes operadores genéticos. Hay diferentes tipos de selectores como por ejemplo:
  - **Selector por Torneo**: Se selecciona una muestra de N individuos y de esta se elige al mejor de ellos.
  - **Selector por Ruleta**: Cada individuo tiene una probabilidad de ser elegido proporcional a lo bueno que este sea.
- **Cruce:** También llamado operador de cruce, es el encargado de realizar combinaciones de individuos con el fin de optimizarlos. Existen diferentes tipos de operadores de cruce como por ejemplo:
  - **Cruce por un punto**: Se selecciona aleatoriamente un punto el cual dividirá a ambos individuos en dos partes, que intercambiarán la segunda parte entre ellos (Figura 4).

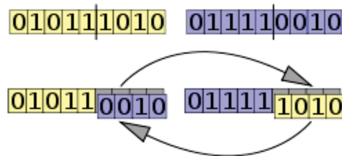


Figura 4: Diagrama One-point crossover

- **Cruce por dos puntos**: Al igual que One-point crossover pero dividiendo en tres partes a los individuos que intercambiaran la parte central de la división.
- **Cruce uniforme**: Cada valor tiene una probabilidad de ser cruzado con el otro individuo.

- **Mutación:** Es el operador encargado de abrir el campo de posibles soluciones a la muestra de individuos a través de modificaciones aisladas aleatorias. Cabe destacar los siguientes tipos de Mutators:
  - **Mutación uniforme**: Modifica un valor del individuo con un valor obtenido aleatoriamente.
  - **Mutación Gaussiana**: Modifica un valor del individuo obteniendo el valor a través de un método de obtención de números aleatorios por medio de Gaussianas.
- **Recombinación:** Este operador se encarga de, por medio de técnicas estocásticas, eliminar a la mitad de la población, de manera que los mejores individuos tienen mayor probabilidad de sobrevivir. Su misión es cortar ramas de soluciones equívocas o que no están lo suficientemente optimizadas.

Datos: Conjunto de n individuos
Resultado: Conjunto de n individuos óptimos
Evaluación inicial;
mientras *not condición de parada* **hacer**
    selección de $n/2$ individuos;
    cruce de individuos seleccionados;
    mutación de la población;
    remplazo para quedarnos con n individuos;
fin

Algoritmo 1: Algoritmo Genético

Como podemos observar en el algoritmo 1, partimos de una población inicial de individuos, cada uno es considerado como solución al problema a analizar. El resultado esperado es una población del mismo tamaño que la inicial. Para conseguir este resultado es necesaria una evolución de sus individuos obtenida de usar los tres operadores antes citados.

Una de las operaciones que se realizan en el algoritmo 1 es la **evaluación** de la población. Este paso es la asignación de una puntuación, también llamada *Score*, a cada individuo dependiendo del resultado obtenido al ejecutar el problema a resolver

con los valores del individuo. Gracias a este proceso los operadores de selección y recombinación pueden realizar su cometido dado que necesitan el *Score* de los individuos a analizar.

Otra de las operaciones que cabe destacar es la **condición de parada** que es la responsable de parar el bucle evolutivo en caso de darse una situación determinada, como por ejemplo puede ser un bajo nivel de dispersión entre individuos o un número máximo de evaluaciones. En caso de darse la condición, se considera que la población ha llegado a estabilizarse o que ha llegado a una convergencia prematura y por lo tanto se puede comenzar el análisis de los resultados para poder considerar el resultado como una solución óptima al problema que se le plantea.

4.4. Evolución Diferencial

Al igual que los Algoritmos Genéticos, la Evolución Diferencial (DE) es un algoritmo de optimización cuya estructura es similar a la antes citada. La diferencia entre la DE y el resto de algoritmos englobados dentro de los Algoritmos Evolutivos es el uso de un operador de mutación diferente.
Hay muchas variaciones para este tipo de algoritmo y, para determinar el nombre de cada una de ellas, se utiliza una notación muy sencilla en la que se separa por barras todas las características de la variante, como sería ,por ejemplo *DE/rand/1* en el que se muestra el nombre del algoritmo, el método de selección utilizado y el numero de vectores que se utilizan en el proceso de mutación.

DE/rand/1	$v_i = x_{r1} + F_1(x_{r2} - x_{r3})$
DE/best/1	$v_i = x_{best} + F_1(x_{r2} - x_{r3})$
DE/rand to best/1	$v_i = x_{r1} + F_1(x_{r2} - x_{r3}) + F_2(x_{best} - x_{r1})$
DE/curr. to best/1	$v_i = x_i + F_1(x_{r2} - x_{r3}) + F_2(x_{best} - x_i)$
DE/rand/2	$v_i = x_{r1} + F_1(x_{r2} - x_{r3} + x_{r4} - x_{r5})$
DE/best/2	$v_i = x_{best} + F_1(x_{r2} - x_{r3} + x_{r4} - x_{r5})$

Figura 5: Variantes DE [2]

De todas las variantes que hay la que hemos implementado en nuestro código es DE/rand/1 en la que la selección tiene que obtener tres individuos y realizar el proceso de mutación mediante esta función:

$$X_n = \text{A} + F\ (\text{B} + \text{C})$$

Siendo F un valor de control del operador que puede se cualquier valor comprendido entre 0 y 1.
Para que este algoritmo pueda funcionar, el operador de selección debe ser específico para este algoritmo dado que tenemos que obtener a 3 instancias dentro de la muestra sin que estas se repitan.
A su vez también es necesario el uso de operadores de cruce específicos. Existen varios tipos de operadores de cruce, el más utilizado es el operador *Exponencial* pero en nuestro proyecto hemos implementado el cruce *Binomial*. Este último realiza un bucle en el que el individuo resultante va obteniendo valores tanto del padre como de la madre de manera aleatoria.
El operador de recombinación en este caso lo único que debe realizar es comparar X_n con el antiguo individuo de la misma posición y quedarse con el que sea mejor de los dos.

4.5. Búsqueda local

La Búsqueda Local (LS) es un método heurístico de optimización de problemas. Este tipo de algoritmos se caracterizan por que realizan una búsqueda dentro del espacio de posibles soluciones. Hay multitud de algoritmos LS como son por ejemplo la búsqueda *Tabú* y el algoritmo *Hill climbing*. En nuestro caso vamos a comentar el algoritmo *Multiple Trajectory Search* (MTS) [21], específicamente su variante *MTS-LS1* que es la que ha sido implementada en este TFG.

En general, todo tipo de algoritmo de búsqueda se basa en desplazar las componentes de las posibles soluciones observando si consiguen maximizar o minimizar el resultado del problema a analizar. En nuestro caso las funciones que utilizaremos para desplazar las componentes de las posibles soluciones en ambos sentidos son las siguientes:

$$X_n = X_o + (Adjust * SR)$$
$$X_n = X_o - (Adjust * SR)$$

En el capítulo de Desarrollo veremos en profundidad el algoritmo *MTS-LS1* utilizado en la implementación final.

4.6. MOS

Como ya hemos visto, existen diferentes tipos de heurísticas de optimización y cada una de ellas tiene unos pros y contras a la hora de encontrar la solución a un problema en cuestión. La decisión de que heurística utilizar se ha convertido en una premisa esencial a la hora de optimizar cualquier tipo de problema. Es por ello que la necesidad de poder unir diferentes técnicas en un único algoritmo se ha vuelto de vital importancia en el mundo de la optimización y es por ello que se diseñó el *framework MOS* [3].

Este *framework* permite evaluar el funcionamiento de cada heurística y poder modificar el porcentaje de participación en función de los resultados obtenidos por cada técnica. Como podemos observar en la figura 6 el algoritmo es muy parecido a cualquier EA diferenciándose en que en vez de realizar los operadores propios del EA se ejecutan una serie de técnicas.

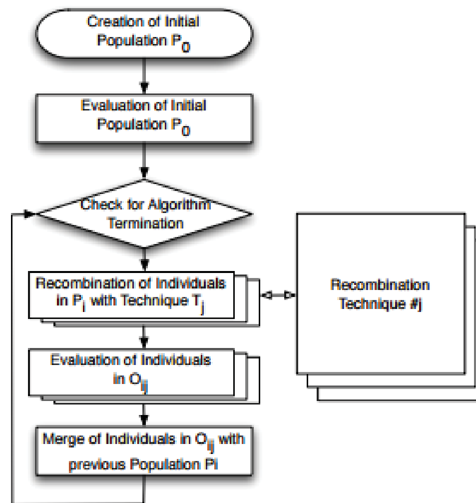


Figura 6: Esquema general MOS [3]

Una vez se completa cada bucle de la figura 6 se debe actualizar la calidad de cada algoritmo y por consiguiente al finalizar la ejecución de todas las técnicas se actualizara el ratio de participación de cada una de ellas.

4.7. Conclusión

Como hemos podido observar en este capítulo de la memoria, el desarrollo y los avances que se están teniendo dentro del campo de los algoritmos evolutivos actualmente se encuentran en una posición muy importante dado que la necesidad de resolver problemas del mundo real que antes no podían ser resueltos por medio de técnicas matemáticas convencionales.

Una de las principales características que tienen los algoritmos evolutivos es que pueden ser utilizados en numerosos tipos de problemas gracias a que son capaces de adaptarse sin necesidad de conocer el entorno del problema.

Es obvio ver la necesidad de sistemas que puedan contener varios algoritmos evolutivos al mismo tiempo, dado que cada uno de ellos tiene sus pros y sus contras. Es por ello que en el siguiente capítulo de la memoria comentaré el trabajo realizado en torno

al *framework* antes descrito *MOS* sobre el que trata este TFG.

5. Desarrollo

En esta sección documentaré las fases del proyecto, junto con los diseños finales empleados en la implementación de este proyecto.

5.1. Introducción

Como hemos visto en la sección de objetivos este TFG trata de la traducción de un código escrito en C++ a Python. Aparte de este cambio es necesario un cambio de diseño, dado que en el anterior código las diferentes modificaciones que se realizaron lo convirtieron en un código bastante inestable.
Uno de los cambios más importantes es la agrupación de los algoritmos en una clase interfaz *Algorithm* que sea implementada por los algoritmos anteriormente vistos. Gracias a este cambio los algoritmos pueden ser ejecutados individualmente y a su vez a través del *framework* MOS.
Para poder hablar de la implementación es necesario hablar de la estructura del proyecto. Este se divide en:

- **Algoritmos**: Es la parte que contiene la implementación de las heurísticas antes vistas. Están implementadas de tal forma que se posibilita su ejecución individualmente y con el *framework MOS*.

- **Operadores**: Es el conjunto de todos los operadores genéticos empleados en el TFG, también contiene la implementación del algoritmo de búsqueda local *MTS-LS1*.

- **Contenedores**: Son las clases utilizadas para almacenar individuos y poblaciones de individuos.

- **Problemas**: En este grupo se encuentran las clases que definen las funciones de objetivo a resolver, es decir los problemas de optimización.

- **MOS**: Esta parte del TFG contiene todas las clases relacionadas con el funcionamiento del *framework MOS*. Para el correcto funcionamiento son necesarias dos clases auxiliares que se ocupan de calcular la calidad de cada algoritmo y de establecer la participación dependiendo de la calidad.

5.2. Análisis y diseño

La primera fase del proyecto, fue el análisis del código anterior codificado en C++. Este código estaba compuesto por 64 ficheros contando implementación*(.cc)* y definición*(.h)*.
Cabe destacar los ficheros relacionados con los algoritmos a implementar que en el código anterior se llamaban Técnicas, las cuales heredaban de la interfaz MOSTechnique. Esta a su vez implementaba de la clase GAID que era una clase que definía cada algoritmo por un identificador único.
Una de las condiciones para la implementación es pasar de llamar Técnica a Algoritmo y poder unificar mediante una única clase y así facilitar la implementación del framework MOS.
Para facilitar el análisis del código ya existente se utilizó la herramienta *doxygen* para realizar una documentación más exhaustiva sobre la implementación ya realizada. Partiendo de esta base se realizaron los diseños para su nueva codificación, basándose en un sistema jerárquico en el cual cada clase tenga su interfaz y de esta manera poder implementar nuevas clases manteniendo el estilo de la implementación.
Uno de los puntos fuertes del proyecto es el framework llamado *'MOS'*, ya antes citado, que otorga al algoritmo la posibilidad de utilizar simultáneamente diferentes implementaciones de la clase Algorithm. Esta fase del proyecto es muy avanzada pero su diseño es esencial para comprender el resto del proyecto, por eso hay que plantear el código para que su implementación no sea muy costosa.

Junto con la codificación del proyecto, se realizara simultáneamente la fase de *testing*, utilizando para ello una herramienta de Python llamada *unitest*, la cual nos otorga mucha facilidad a la hora de realizar test unitarios y test de integración que utilizaremos para testear todos los operadores, algoritmos y posteriormente el *framework MOS*.
Uno de los usos planteados para esta aplicación es su uso en ordenadores distribuidos. Para ello el código debe garantizar su ejecución bajo una versión estable de Python. Las dos mejores opciones son tanto la versión 2.7 como la 3.4. Esta última facilitaría la definición de los límites cuando estos son infinito, dado que uno de sus principales cambios fue la inclusión del infinito como palabra clave dentro del lenguaje. A pesar de esta gran ventaja, se ha optado por codificar utilizando la versión 2.7 por ser la más estable hasta la fecha.
Por último, para que este algoritmo pueda ser ejecutado de una forma sencilla, habrá que implementar una clase ejecutora *main*, la cual tendrá un fichero de configuración que recibirá como parámetro de entrada. En este fichero se podrán obtener todos los datos necesarios para la correcta configuración e inicialización del algoritmo y que tras la ejecución guarde en un fichero la salida de la población y todos los datos útiles de la ejecución.

5.3. Implementación de MOS

5.3.1. Diseño de MOS

Podemos definir MOS como un framework que permite la hibridación entre diferentes algoritmos evolutivos con el fin de obtener lo mejor de cada uno de estos. Para comenzar su descripción empezaremos analizando su diagrama de clases:

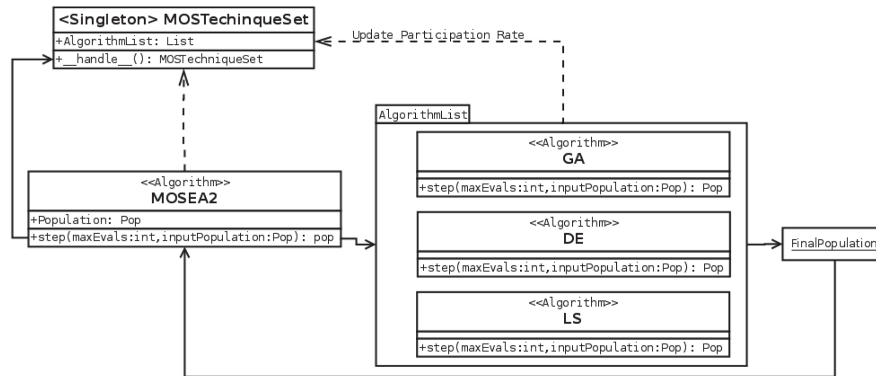


Figura 7: Diagrama MOS

Como podemos observar la clase base del *framework* es MOSEA2, que hereda de la clase Algorithm. Esta clase es una de las más importantes dentro del *framework* dado que es la encargada de iterar sobre los algoritmos que queremos emplear.
Esta colección de algoritmos se encuentra en una clase *singleton* llamada MOSTechniqueSet que además de tener la lista de algoritmos a utilizar también tiene métodos para gestionarlos.
En el gráfico 8 podemos observar el bucle interno de la clase MOSEA2, en el cual podemos observar dos de las clases que forman parte de MOS, las cuales son MOSQualityFunction y MOSParticipationFunction, que serán descritas posteriormente.

Por lo que podemos extraer del diagrama superior, *MOSEA2* es una clase encargada de obtener la lista de Algoritmos desde *MOSTechniqueSet*, como antes hemos citado. Una vez tenemos la lista de algoritmos con su participación llamamos a su método *evolve*, que es un método que veremos posteriormente, al cual se le facilita una población y el número de evaluaciones máximo.
Una vez termina la ejecución de la función del algoritmo debemos actualizar el valor de calidad del algoritmo, calculado por la clase *MOSQualityFunction*, y una vez tengamos la calidad de todos los algoritmos tendremos que actualizar la participación de cada algoritmo otorgando al mejor algoritmo mayor número de evaluaciones para así favorecerle en ejecuciones futuras, este proceso lo realiza la clase *MOSParticipationFunction*.
Las dos clases auxiliares serán descritas a continuación.

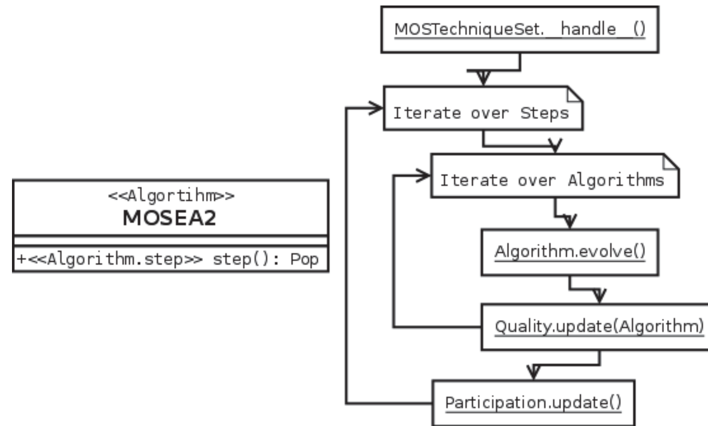


Figura 8: Implementación del bucle MOS

5.3.2. Clase MOSParticipationFunction

Esta clase es una de las dos clases auxiliares que utiliza MOS para garantizar su correcto funcionamiento. En este caso esta clase es la encargada de, dependiendo de la calidad de cada algoritmo, otorgar una participación para favorecer o no los cambios que este algoritmo da a la población.

Datos: MOSTechniqueSet
Resultado: None
Mientras *(Loop over AlgorithmList)* **hacer**
  diff = (bestQuality - Algorithm.getQuality()) / (bestQuality - *base*);
  partInc = Algorithm.Participation * *adjust* * diff;
  si *Algorithm is not best* **entonces**
    sharedRatio = partInc / numberBestAlgorithms;
    Mientras *bestAlgorithm in BestAlgorithms)* **hacer**
      bestAlgorithm.Participation += sharedRatio;
    fin
    Algorithm.Participation -= partInc
  fin
fin

Algoritmo 2: Método Update de MOSParticipationFunction

Dentro del *framework* hay dos tipos de funciones de participación: *ConstantParticipation* y *DynamicParticipation*. En nuestro caso vamos a comentar la segunda dado que la primera no tiene tanta complejidad dado que otorga a todos los algoritmos la misma participación y esta no se modifica respecto de la calidad. Por lo tanto *DynamicParticipation* es generalmente la función que más se va a utilizar.

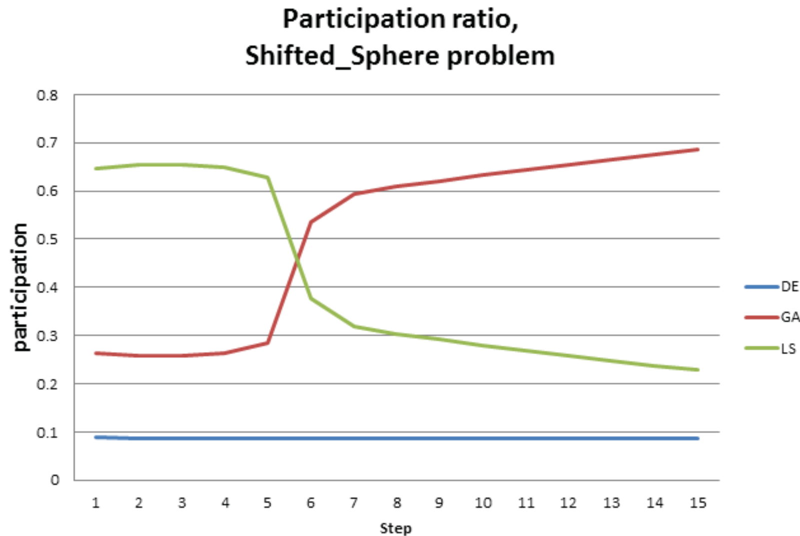


Figura 9: Ejemplo de participación dinámica

Si analizamos el algoritmo 2 podemos ver como lo único que realiza es un bucle que recorre todos los Algoritmos que se encuentren en la lista de *MOSTechiqueSet*. Por cada algoritmo se realiza una primera comprobación para saber si es uno de los que mejores resultados ha obtenido y de ser este el caso se aumenta su participación de la misma manera que se reduce entre los algoritmos que no han resultado ser los mejores. Gracias a este código los algoritmos que favorecen la optimización del problema son los que mayor número de evaluaciones utiliza como podemos ver en la figura 9.

Uno de los objetivos de esta función es que la suma de todas las participaciones de los algoritmos sea el 100 %, es por ello que a la vez que se incrementa la participación se debe decrementar en el resto de algoritmos.

5.3.3. Clase MOSQualityFunction

Segunda clase auxiliar utilizada por MOS. Esta se encarga de dar un valor de calidad a cada algoritmo dependiendo de las mejoras que estos han introducido en la población. Hay distintos tipos de funciones de calidad como son:

- **Average Fitness Quality**: Calcula el *Score* medio de la población afectada por el algoritmo.
- **Average Fitness Increment Quality**: Calcula la media del incremento del *Score* entre dos soluciones.

- **Average Diversity Quality**: Calcula la distancia entre los diferentes individuos de una población.

Esta clase es llamada para actualizar el valor de calidad del algoritmo. Este valor depende principalmente de lo que ha mejorado la población con el uso del algoritmo. Como hemos visto hay diferentes tipos de cálculo de este valor. En términos de este TFG el más utilizado ha sido *Average Fitness Increment Quality* dado que es el más complejo y calcula la mejora que se ha obtenido gracias un algoritmo entre una generación y la siguiente. Esta función utiliza para ello una clase auxiliar que almacena un árbol genealógico de los individuos y de esta forma almacenamos el incremento de *score* que sufre la población.

Figura 10: Ejemplo de problema de minimización con valor óptimo de -450

5.4. Implementación algoritmos

5.4.1. Clase Algorithm

Algorithm
-pop: Population -params: List -conv: Float
+initialize() +run(maxEvals:int): nEvals, Population +evolve(maxEvals:int,pop): nEvals, Populatio +best(): Genome +population(): Population *+step(maxEvals:int,pop:Population)* *+done(): boolean*

Figura 11: Clase Algorithm

Esta clase está diseñada para crear una jerarquía entre las diferentes clases hijas que conformaran todos los tipos de algoritmos utilizados por nuestro *framework*.

Esta clase está diseñada para poder ser usada junto con el *framework MOS*, dado que la herencia otorgaría al código propiedades básicas para que se puedan ejecutar diferentes tipos de algoritmos fijados anteriormente.
Este principio hereditario es empleado en todo el proyecto para otorgar esta propiedad a todo tipo de clases tanto contenedoras, como implementación de operadores. Los tres tipos de algoritmos planteados para ser ejecutados son el Algoritmo Genético (GA), el de Búsqueda Local (LS) y el algoritmo de Evolución Diferencial (DE). A su vez la clase encargada de ejecutar el *framework*, MOSEA2, también hereda de esta clase. En la figura 12 podemos observar la herencia que se da en las clases que heredan de *Algorithm*
Como podemos ver en el diagrama 11, la clase contiene una variable llamada Population, esto se debe a que si la ejecución es de únicamente un algoritmo, no sería necesaria la instanciación del *framework* por completo que es el que se encargaría de manejar la población. En ese caso únicamente se ejecutaría el método *Evolve* (**??**) del algoritmo en cuestión.
También contiene una serie de métodos para ser utilizados en caso de la ejecución de únicamente un algoritmo como son los métodos relacionados con la población y con el criterio de finalización.

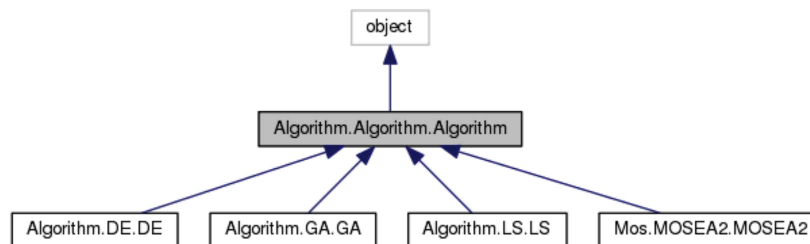


Figura 12: Herencia en clases Algorithm

5.4.2. Clase Genetic Algorithm

Uno de los tres algoritmos implementados se encuentra en esta clase.

Como hemos comentado anteriormente, implementa la clase interfaz Algorithm, de la cual extiende principalmente el método *evolve* que es el que realiza el principal bucle de iteración. Para garantizar la distribución del código se han creado dos métodos auxiliares llamados *offspring* y *offspring_internal* los cuales realizan las operaciones biológicas básicas.
Por ejemplo, *offspring*, es el método que realiza la selección a través de un bucle que

```
<<Algorithm>>
GeneticAlgorithm
-pop: Population
-evals: Int
+mutator: Mutator
+crossover: Crossover
+selector: Selector
+problem: Problem
+elitism: Elitism
+step(maxEvals:Int,pop:Population): evals,
       Population, conv
-offspring(maxEvals:Int,pop:Population): evals
            Population, conv
-offspring_internal(dad:Genome,mom:Genome,
                    nSons:Int): Genome, Genome
+done(): Boolean
+getEvals(): Int
+setEvals(value:Int): void
```

Figura 13: Clase Genetic Algorithm

recorre toda la población, una vez hemos realizado la selección de dos miembros se llamaría a *offspring_internal* que recibe como parámetros los dos individuos seleccionados y el número de hijos que queremos obtener. En este segundo método es donde se realizan las operaciones de cruce y mutación.

Como podemos ver en el pseudocódigo 3, el método *Evolve*, contenido en la clase GeneticAlgorithm, es uno de los métodos más importantes del proyecto. Éste se encarga de recorrer la población hasta que se da una condición de parada. A grandes rasgos el algoritmo selecciona dos individuos aleatoriamente en los que se realiza el operador de cruce obteniendo dos hijos y a partir de estos, realizar el proceso de mutación.
Una vez se recorre toda la población, se llama al método *Elitism* el cual se encarga de seleccionar a los mejores individuos tanto de la nueva generación como de la antigua y con ellos generar una población nueva con el tamaño inicial que tenía, la cual será analizada de nuevo y se comprobará la condición de parada para ver si el algoritmo debe parar su ejecución, en este caso, se considerará que la población se ha agrupado en torno a una misma solución, esta será considerada óptima para el problema planteado.

5.4.3. Clase Local Search

Es otro de los algoritmos implementados en el código. Al extender la clase interfaz Algorithm posee la misma estructura que la clase GA, lo que cambia en este caso es la forma de buscar la solución.
Por ejemplo en el algoritmo de GA el procedimiento es utilizar operadores para favorecer, en cierto modo, la evolución de soluciones que optimicen al problema, en cambio el algoritmo de búsqueda local se basa en buscar la solución utilizando para ello una especie de fuerza bruta.
Técnicamente lo que se realiza es ver como se comporta el valor *Score* aumentando

Datos: Population, maxEvals
Resultado: NewPopulation, Evals
Evals = 0;
mientras *(not condición de parada)* ***y*** *(Evals <maxEvals)* **hacer**
  Mientras *i=0 ; i¡Population.size ; i+2* **hacer**
    dad = Selector(Population);
    mom = Selector(Population);
    son1, son2 = Crossover(dad, mom);
    son1 = Mutator(son1);
    son2 = Mutator(son2);
    newPop.append(son1, son2);
  fin
  Problem(newPop);
  finalPop = Elitism(Pop);
fin

Algoritmo 3: Método Evolve de GA

y decrementando el valor interno de las soluciones y así conseguir llegar a la solución directamente.
El algoritmo en cuestión no se incluye directamente en la clase de LS sino que es un operador de búsqueda, en este caso el algoritmo implementado es MTS_LS1:

Como podemos observar en el algoritmo 4, hay una serie de parámetros de clase específicos como son por ejemplo SR, MTSAdjustFailed, MTSMoveLeft y MTSMoveRight. Estos parámetros son instanciados al comienzo de la ejecución del código y su significado es el siguiente:

- **SR**: Es el parámetro que define el desplazamiento que se va a aplicar a cada uno de los componentes del individuo en cuestión.

- **MTSAdjustFailed**: Es el parámetro que regula el ajuste de precisión del *SR*. En caso de que no se obtenga mejora con la precisión actual es cuando hay que realizar desplazamientos más pequeños.

- **MTSMoveLeft y MTSMoveRight**: Definen la proporción del *SR* que se debe aplicar en cada uno de los dos sentidos posibles.

Resumiendo el algoritmo MTS_LS1, podemos ver como su misión es incrementar o decrementar el valor de la solución hasta que llega a un punto que su mejora únicamente se da aumentando su precisión. Cuando incrementando se llega a un extremo de los valores posibles quiere decir que la solución que optimiza al problema se encuentra en

el extremo en concreto.

```
Datos: Genome, maxEvals
Resultado: NewGenome, Evals
Evals = 0;
mientras (Loop over Genome) y (Evals <maxEvals) hacer
    si Precision reached entonces
        SR *= MTSAdjustFailed;
    fin
    //Decrement part of code
    new_gen = old_gen - (MTSMoveLeft * SR);
    si new_gen best old_gen entonces
        Continue decrement;
    fin
    //Increment part of code
    new_gen = old_gen + (MTSMoveRight * SR);
    si new_gen best old_gen entonces
        Continue increment;
    fin
fin
```

Algoritmo 4: Operador MTS_LS1

5.4.4. Clase Differential Evolution

Al igual que en los otros dos algoritmos ya vistos, la evolución diferencial también es considerada como un algoritmo de optimización estocástica agrupado en el conjunto de la computación evolutiva.
La base es similar a la del resto de algoritmos salvo en el paso de mutación en el que se muestra una notable diferencia, además de tener una nomenclatura especifica en la que el individuo se llama *Target vector* y el conjunto de individuos utilizados para la mutación forman el *Mutant vector*.
En este caso en vez de comparar entre individuos, se pasa a comparar soluciones con vectores mutantes que son formados por un conjunto de individuos de la misma población. En el algoritmo 5 podemos ver el proceso de selección, mutación y recombinación.

Como vemos en el algoritmo 5 el bucle itera sobre una población de individuos y por cada elemento genera un vector mutante de tres objetos que obtiene mediante el método *Selector*. Una característica importante que caracteriza al selector es que los tres individuos seleccionados deben ser distintos entre si y con respecto al *target_vector*.

Datos: Population, maxEvals
Resultado: NewPopulation, Evals
Evals = 0;
mientras *(Loop over Population)* ***y*** *(Evals <maxEvals)* **hacer**
    g1, g2, g3 = Selector(Population) //Mutant vector;
    mutant_vector = g1 + F * (g2 - g3);
    new_gen = Crossover(target_vector, mutant_vector);

    si *new_gen* ***mejor que*** *target_vector* **entonces**
        NewPopulation.append(new_gen);
    en otro caso
        NewPopulation.append(target_vector);
    fin
fin

Algoritmo 5: Algoritmo DE

Una vez tenemos la selección realizada procederemos a comprobar si es mejor la unión de los tres individuos, *mutant_vector*, o si en cambio era mejor el *target_vector*. Para esto calculamos lo que sería el próximo individuo a través de la formula que vemos en el algoritmo y posteriormente comprobamos si es mejor o peor y en caso de mejorar al *target_vector* se sustituye por este, que es el paso de recombinación pero reducido a la comprobación individual de cada elemento.

5.5. Implementación de clases contenedores

5.5.1. Clase Genome

Esta clase es la interfaz de todas las "clases contenedor", es decir, todas las clases que contengan en su interior las posibles soluciones al problema. Dentro del lenguaje de algoritmos genéticos, esta clase es la abstracción de los individuos de una población.

Genome
-score: Float
+setScore(value:Float): void +getScore(): Float

Figura 14: Clase Genome

Por temas de herencia, el único objeto que contiene es el valor *Score*, dado que es el único objeto que van a compartir todos los contenedores de soluciones, dado que cada uno de ellos contendrá su propia implementación para almacenar los valores como por ejemplo una simple lista de datos, una matriz o un único valor.

Esta clase debe soportar todo tipo de objetos a almacenar y al ser una clase interfaz, esta definición no puede realizarse dentro de esta, por lo que la definición del objeto contenedor se deja a elección de la clase hija que implemente el contenedor deseado como por ejemplo GAD1ArrayGenome explicada a continuación.

5.5.2. Clase GAD1ArrayGenome

Esta clase es una de las diferentes implementaciones que pueden darse a partir de la clase Genome. Esta en concreto es un conjunto de valores en forma de array unidimensional. Cada uno de estos elementos tiene asociado una dupla llamada *allele* que es utilizada para definir los valores máximos y mínimos que se pueden contener.

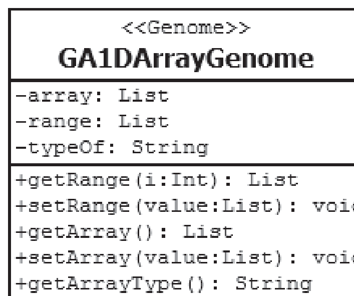


Figura 15: Clase GA1DArrayGenome

La parte de codificación de esta clase no es costosa, pero es complicada conceptualmente. Uno de los errores observados era el tiempo que consumía en realizar una copia de si mismo. En *Python* los dos tipos de copia que existen son el *Soft Copy* y el *Deep Copy*. Por temas de diseño se planteó utilizar la segunda opción dado que cada uno de los objetos deben almacenarse en diferentes posiciones de memoria para que no se puedan modificar entre si, el problema encontrado es que usando esta función, el tiempo de ejecución llegaba a ser 30 veces superior. En busca de una solución al problema se encontraron diferentes implementaciones de funciones de *Deep Copy* realizadas en C que reducían el tiempo consumido en gran parte. Es uno de los aspectos a tener en cuenta para posibles modificaciones al código.
Este problema será tratado en el capítulo de fase final y mejora. Este problema también se ha encontrado en clases que contienen *Genomes* como la clase *Population*.

5.5.3. Clase Population

Este fichero es la clase contenedora de individuos. En ella residen todos los métodos de manipulación de la propia población.

Population
-genomeList: List
+get(): List +get(i:Int): Genome +set(input:List): void +set(input:Genome,i:Int): voic +getSize(): Int +add(input:List): void +add(input:Genome): void

Figura 16: Clase Population

Esta clase puede ser considerada como la clase que contiene a los Genomas. También en ella residen algunas de las operaciones para modificar el conjunto de Genomas.
El problema de copia también se encontró en esta clase, por lo tanto en futuras modificaciones se tiene que tener en cuenta.
La ventaja que nos aporta la herencia es que gracias a ella una Población puede contener todo tipo de Genomas contenedores y no afectar a la funcionalidad de programa en sí.

5.6. Implementación de operadores

5.6.1. Clase Crossover

Esta clase es una de las implementaciones de los **operadores** empleados en el algoritmo genético
Dentro del esquema anterior del Genetic Algorithm, esta clase implementa el proceso de cruce. Hay diferentes tipos de implementaciones de cruce:

- Usados por GA:
  - el cruce por un punto
  - el cruce por dos puntos
- Usados por DE:
  - el cruce binomial
  - el cruce exponencial

La implementación finalmente realizada ha sido únicamente el cruce por un punto el cual explicaremos en el algoritmo 6.

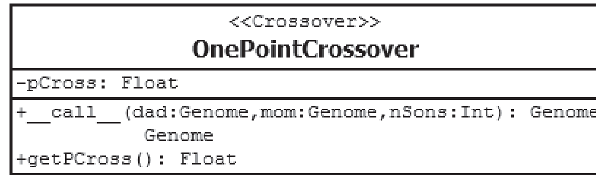


Figura 17: Clase OnePointCrossover

Datos: Dad, Mom, nSons
Resultado: Son1, Son2
si *Dad.len == Mom.len* **entonces**
    randInt = Random(Dad.len);
    Son1 = Dad [0 : randInt] + Mom [randInt : END] ;
    Son2 = Mom [0 : randInt] + Dad [randInt : END] ;
fin

Algoritmo 6: Método OnePointCrossover

Como podemos observar en el algoritmo superior, este algoritmo consiste en calcular una posición aleatoria que este comprendida entre 0 y el tamaño de los Genomas de entrada.
Esta posición será por la que se realice la división de los Genomas. Los hijos resultantes serán la mezcla de la primera parte del primer Genoma con la segunda parte del segundo y al revés para el siguiente hijo.
Con esto obtenemos dos Genomas resultantes con el mismo tamaño que el que tenían los entrantes.
Uno de los principios de los algoritmos genéticos es la aleatoriedad por lo que hay una probabilidad residente de que no se realice el cruce entre dos individuos al igual que en el resto de operadores.

5.6.2. Clase Mutator

Esta clase es una de las implementaciones de los **operadores** empleados en el algoritmo genético
La existencia de esta clase, dentro de un algoritmo genético, se debe al problema de que teniendo una población inicial aleatoria y utilizando únicamente el operador de cruce dentro de la población, el espacio de soluciones abordado permanece casi estático, es por ello que es necesario un nuevo operador que expanda el espacio de soluciones analizadas. Este operador es la mutación, que es la encargada de modificar un valor aleatoriamente dentro de los limites definidos por los alelos de un genoma. Gracias a esta operación, con el paso de las generaciones, el espacio de soluciones analizado se expande, consiguiendo un aumento considerable del *score*.

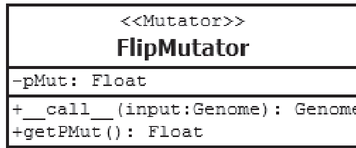


Figura 18: Clase FlipMutator

Datos: Genome
Resultado: Genome
para cada *value in Genome* **hacer**
  si *pMut is true* **entonces**
    value = Random(Genome.min, Genome.max);
  fin
fin

Algoritmo 7: Método FlipMutator

El proceso de Mutación realizado a un Genoma es el recorrido del contenedor de este realizando un calculo con la probabilidad residente que decida si se realiza Mutación o no.
Si esta es realizada se obtiene un valor aleatorio situado entre los dos limites de los datos a mutar, valores que podemos encontrar con el método *getRange()* contenido en el Genoma.
Una vez tengamos ese valor aleatorio se lo asignamos en la misma posición que estaba. El resultado de esto es la apertura de nuevos caminos posibles dentro del abanico de soluciones. Gracias a este proceso nuevos espacios de soluciones son analizados y en caso de que sean caminos aceptables, su futuro análisis es incentivado.
Hay diferentes tipos de operadores de mutación como los que hemos visto en la sección 4.3.

5.6.3. Clase Selector

Esta clase es una de las implementaciones de los **operadores** empleados en el algoritmo genético
A la hora de realizar el cruce de dos individuos, estos tienen que ser seleccionados anteriormente. Para obtenerlos hay diferentes algoritmos de selección, como son el algoritmo Roulette, el algoritmo del torneo o el algoritmo de selección uniforme. Para la implementación de este proyecto se ha desarrollado el algoritmo del torneo.

Figura 19: Clase TournamentSelector

Datos: Population
Resultado: Genome
list = new list;
para cada *n in nGrop* **hacer**
| list.append(Population.get(Random()))
fin
return max(list, score)

Algoritmo 8: Método TournamentSelector

El algoritmo de torneo consiste en una preselección de n individuos aleatoriamente, de este conjunto se realizará una ordenación de mayor a menor teniendo en cuenta el valor del *Score*, una vez estén ordenados devolveremos al mejor individuo.
Este proceso es realizado para por ejemplo realizar el cruce de dos individuos y que la probabilidad de escoger una buena solución sea mayor y así espacios de soluciones que no sean óptimos no sean recorridos.
Otro algoritmo que podría ser empleado es el de selección uniforme, este consiste en una búsqueda aleatoria entre todos los individuos de la Población. Este algoritmo es más sencillo de implementar, la parte negativa es que dentro del abanico de soluciones se pueden recorrer caminos que no son óptimos, lo que originaría un consumo de cómputo innecesario.
Por último el algoritmo de selección por Ruleta establece una probabilidad de selección a cada individuo directamente proporcional a lo bueno que es para el problema definido. Por lo que con este algoritmo cualquier individuo puede ser seleccionado con mayor o menor probabilidad. Para la utilidad de un algoritmo genético este funcionamiento es optimo, pero comparado con el resto de selectores, su consumo de tiempo de cómputo es más elevado.
Es por esto por lo que se ha decidido implementar el selector de Torneo.

5.6.4. Clase Elitism

Esta clase es una de las implementaciones de los **operadores** empleados en el algoritmo genético
Dentro del algoritmo genético, esta clase es la encargada de seleccionar los mejores genomas una vez hayan sido realizados los procesos de cruce y mutación, y con estos crear una nueva población con el tamaño inicial de individuos.

Figura 20: Clase PopulationElitism

Datos: ParentPopulation, ChildrenPopulation
Resultado: FinalPopulation
TemporalList = ParentPopulation + ChildrenPopulation;
FinalPopulation = TemporalPopulation.sort()[0:TemporalPopulation.size()/2];

Algoritmo 9: Método PopulationElitsm

Como podemos observar en el algoritmo superior, el resultado de llamar a este operador es una Población compuesta por la unión de los mejores individuos dentro de las dos Poblaciones que se pasan como parámetro de entrada.
Este operador dentro del algoritmo genético es el abstraído de la selección natural de la que hablaba Darwin[22] en su libro de investigación "El origen de las especies".
Dentro del mundo de los algoritmos genéticos, este operador es el encargado de definir los caminos de soluciones que serán posteriormente analizados creando una población óptima utilizando para ello las dos que han sido pasadas como parámetros. También es el encargado de cerrar caminos que no obtengan un *Score* bueno para seguir investigando.
En el proceso de análisis que fue realizado al principio de este proyecto, se observó en un vídeo[18] un algoritmo de reemplazo que se asemeja en mayor parte a la realidad que el que ha sido implementado. Este algoritmo ordenaba de mejor a peor a todos los individuos de la población, y se realizaba una selección de la mitad de estos en base en que los peores individuos tenían mayor probabilidad de no ser escogidos y al revés con los mejores de la población. De esta manera se garantizaría el principio de aleatoriedad que tiene este algoritmo.
Por cuestiones de mantener la estructura del código anterior, se decidió implementar el algoritmo de reemplazo como el descrito en la figura anterior.

5.7. Implementación de problemas

5.7.1. Clase Problem

En esta clase se encuentra la definición del problema que queremos resolver. Uno de los diferentes métodos que contiene es el encargado de otorgar a un individuo introducido como parámetro de entrada al método, un *score* asociado a su Genoma. Este valor es el empleado para decidir que individuo es mejor que otro. También contiene diferentes métodos asociados a la anterior definición de problema, como son el nombre

de este y el tipo.

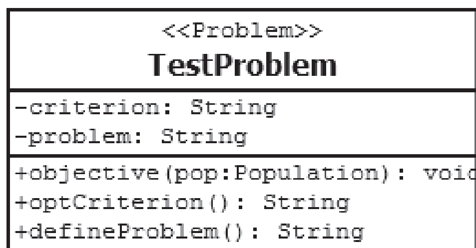


Figura 21: Clase TestProblem

Para la realización del test de integración con el sistema, se optó por realizar una clase de Problema a modo de ejemplo posteriormente se implementó toda la colección de problemas definidas en *SOCO*.
Todos los problemas de esta colección están enfocados en funciones de minimización y la mayoría se consideran problemas "desplazados" dado que su valor optimizado no se encuentra en el valor de 0.
Dentro del código final encontramos 12 problemas, 11 corresponden a *SOCO 2011* [23] de los 19 que están definidos en el *benchmark* y 1 es un problema de test, creado por mi, que es el único problema de maximización que tenemos el cual otorga un mejor *Score* a los individuos cuanto mayor valor tiene, es decir cuanto más se acerque a su alelo máximo.

5.8. Fase final y mejora

Una vez llegado el punto de realizar la integración y los testeos, cabe destacar una serie de inconvenientes observados a la hora de implementar el código.
El primero de estos problemas era el coste de tiempo que tenía la función DeepCopy implementada en el código de Python base. Para poder detectar el problema tuvimos que utilizar una serie de utilidades desarrolladas en Python llamadas *Profilers*, los cuales realizan una ejecución del código viendo que métodos son los más costosos en términos de tiempo de ejecución. Los resultados fueron claros y precisos dado que se podía observar que al llamar a métodos que realizaran copias en profundidad de objetos el tiempo que se necesitaba ocupaba un 60 % del tiempo total de ejecución. Para solucionarlo después de realizar una investigación se decidió utilizar una librería incluida en Python llamada *Pickle* la cual esta implementada en C y reduce el tiempo de ejecución hasta llegar a un 16 % del total.

Otro de los problemas observados fue a la hora de dar un identificador a cada genoma. Python tiene un método que se utiliza para este fin, el problema es que no garantiza que el identificador sea único dado que este es la posición en la memoria que ocupa el objeto. Para solventar este error existe una librería externa a Python que es capaz de generar uid pero su uso no fue necesario dado que las ids de los objetos únicamente las utilizamos cuando estos aún permanecen en memoria. Para poder optar por esta solución cabe destacar la forma en la que Python borra de la memoria a los objetos instanciados, para que esta situación se dé el objeto en cuestión debe dejar de ser utilizado en la integridad del programa dado que este tiene un contador de veces llamado y en caso de que este contador llegue a 0 es cuando este es borrado.

6. Resultados y estadísticas

En este capitulo se comentarán los resultados obtenidos como estadísticas de tiempo de ejecución.
Para analizar el estado actual del framework se realizará una clase ejecutora que probará todos los problemas con el fin de obtener tiempos de ejecución y analizar la salida final de cada problema. Para la codificación de esta clase se utilizaran programas externos como *Excel* con el fin de poder dibujar gráficas detallas para facilitar el análisis de la ejecución. En la tabla 1 podemos observar el ordenador que realizará la prueba.

PC	Intel i7-2600k 4 cores 3.9Ghz
OS	Windows 10
Prog. language	Python 2.7

Cuadro 1: Configuracion PC

6.1. Metodología del análisis de resultados

Para poder estudiar el funcionamiento del *framework* implementado se ha creado un script que ejecuta una lista de problemas propuestos en el congreso SOCO 2011 [23]. Estos problemas son una serie de funciones de optimización continua con los que podremos observar el funcionamiento del *framework* implementado.
Como metodología para facilitar el análisis vamos a ejecutar los problemas inicialmente con los tres algoritmos implementados en el código para ver como se comportan cada uno de ellos individualmente para posteriormente podamos comparar los resultados con los del *framework MOS*. Como número de evaluaciones máximas utilizaremos 5000 veces el número de la población. También para poder comparar los resultados ejecutaremos los algoritmos con un tamaño de población de 25, 50, 100 y 200.
Para poder analizar los resultados vamos a emplear los parámetros que se muestran en la tabla 2 que son los que se siguieron en el análisis del funcionamiento del antiguo código en C++ [24] a excepción del operador de cruce del algoritmo DE y del numero de pasos.

Parámetro	Valor	Parámetro	Valor
Tamaño población	15	DE F	0.5
Prob Mut	0.1	DE Crossover	Binomial
Prob Cross	0.5	LS Operator	MTS_LS1
GA Selection	Tournament	MOS min part	0.05
GA Mutator	Flip	Number os steps	10
GA Crossover	One Point		

Cuadro 2: Configuracion MOS

6.2. Estadísticas

Error obtenido

A continuación veremos una serie de tablas que muestran los resultados obtenidos como la diferencia entre el valor óptimo del problema y el mejor individuo del algoritmo.

Gen Size	25	Max Evals		125000
Problema	GA	DE	LS	MOS
Schewefel Problem	1.01E+01	9.31E+01	4.49E-12	4.78E+01
Shifted Ackley	3.99E-01	9.95E+00	3.13E-13	1.18E+01
Shifted Rastrigin	1.02E+00	3.63E+01	1.32E+01	5.15E+01
Shifted Rosenbrock	2.58E+02	7.13E+08	9.42E+01	3.85E+00
Shifted Griewank	2.33E+01	1.12E+01	1.95E+01	2.80E-01
Shifted Sphere	8.25E-01	1.53E+03	3.97E-13	9.08E+02
f9	1.94E+01	3.54E+01	1.80E+02	2.77E+00
f8	1.55E+03	5.20E+03	1.54E-14	1.50E+01
f10	8.36E-01	2.28E-05	7.85E-28	5.90E-15
f7	2.63E-01	2.32E-03	7.55E-14	5.26E-08
f11	1.67E+01	1.45E+01	2.87E+01	7.28E+01
Media	1.71E+02	6.48E+07	3.05E+01	1.01E+02

Cuadro 3: Resultados dimensión 25

Gen Size	50	Max Evals		250000
Problema	GA	DE	LS	MOS
Schewefel Problem	1.34E+01	1.04E+02	4.77E-10	6.74E+01
Shifted Ackley	4.72E-01	1.39E+01	7.11E-13	9.25E+00
Shifted Rastrigin	6.56E-01	1.59E+02	6.80E+01	1.03E+02
Shifted Rosenbrock	5.75E+02	9.53E+08	1.32E+01	9.10E+01
Shifted Griewank	4.81E+01	4.06E+01	4.12E+01	3.32E+01
Shifted Sphere	2.21E+00	2.83E+03	5.68E-13	4.32E+03
f9	3.54E+01	1.00E+02	2.32E+02	1.34E+02
f8	6.71E+03	5.83E+04	5.70E-12	4.77E+03
f10	1.23E+00	1.55E+02	1.17E-27	6.07E-14
f7	5.34E-01	2.22E+00	1.57E-13	1.11E-07
f11	2.76E+01	1.26E+02	9.16E+01	1.45E+02
Media	6.74E+02	8.66E+07	4.05E+01	8.80E+02

Cuadro 4: Resultados dimensión 50

Gen Size	100	Max Evals		500000
Problema	GA	DE	LS	MOS
Schewefel Problem	2.36E+01	1.40E+02	3.55E-10	4.57E+01
Shifted Ackley	8.60E-01	1.59E+01	1.14E-12	9.60E+00
Shifted Rastrigin	3.64E+00	3.73E+02	1.20E+02	2.70E+02
Shifted Rosenbrock	3.44E+03	1.13E+10	4.30E+01	8.81E+01
Shifted Griewank	9.56E+01	1.87E+02	7.58E+01	8.01E+01
Shifted Sphere	1.08E+01	4.28E+04	1.02E-12	1.06E+04
f9	9.56E+01	3.69E+02	3.35E+02	3.82E+02
f8	1.74E+04	2.12E+05	1.62E-03	3.24E+04
f10	6.49E+00	7.31E+02	2.78E-27	1.91E-13
f7	1.82E+00	8.02E+01	3.59E-13	2.15E-07
f11	9.84E+01	3.55E+02	4.33E+02	3.37E+02
Media	1.92E+03	1.02E+09	9.15E+01	4.02E+03

Cuadro 5: Resultados dimensión 100

Gen Size	200	Max Evals		1000000
Problema	GA	DE	LS	MOS
Schewefel Problem	3.86E+01	1.57E+02	1.95E-09	5.10E+01
Shifted Ackley	3.65E+00	1.90E+01	2.93E-12	1.42E+01
Shifted Rastrigin	9.43E+01	1.10E+03	2.13E+02	7.06E+02
Shifted Rosenbrock	1.17E+06	8.97E+10	9.77E+01	2.08E+02
Shifted Griewank	1.50E+02	1.09E+03	1.80E+02	4.34E+02
Shifted Sphere	6.53E+02	1.02E+05	2.04E-12	4.65E+04
f9	3.89E+02	1.15E+03	6.95E+02	8.97E+02
f8	9.43E+04	7.60E+05	1.90E+01	8.52E+04
f10	1.50E+02	5.08E+03	4.86E-27	1.50E-12
f7	1.96E+01	3.08E+02	6.00E-13	1.05E-06
f11	4.01E+02	1.19E+03	4.46E+02	9.05E+02
Media	1.15E+05	8.16E+09	1.50E+02	1.23E+04

Cuadro 6: Resultados dimensión 200

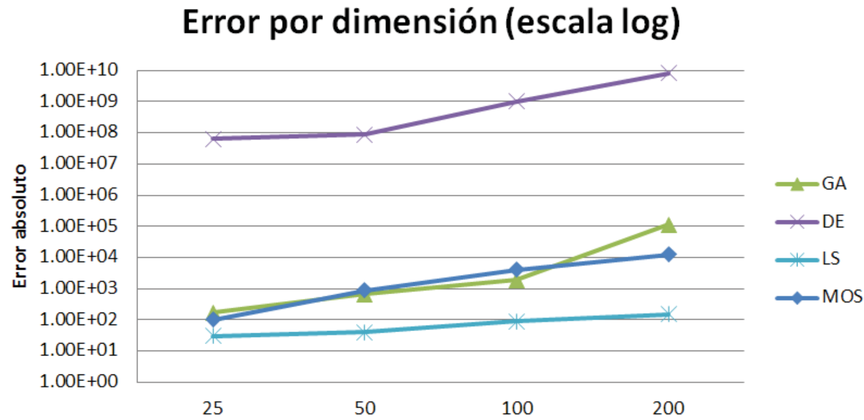


Figura 22: Gráfica de Error medio por dimensión

Si realizamos un análisis de los resultados obtenidos podemos observar como el algoritmo que mejor resultados obtiene es el de búsqueda local, seguido de *MOS* que incluso supera a los otros dos algoritmos propuestos. En términos de resultados el algoritmo que muestra peores resultados es el de Evolución Diferencial teniendo incluso el peor resultado de la muestra por bastante diferencia, el cual es obtenido con el problema *Rosenbrock*. En la sección de trabajos futuros se menciona la necesidad de analizar este resultado.
Cabe destacar que al ser *MOS* una hibridación de los algoritmos propuestos, sus resultados no sean tan buenos como los esperados por este tipo de situaciones, por lo que una vez se detecten los errores que provocan estas situaciones el framework mejorará sus resultados notablemente.
Estas situaciones se pueden identificar por medio de la gráfica 22 en la que se puede observar como los resultados obtenidos por el algoritmo DE están en otra escala al resto de algoritmos.
En esa gráfica también podemos observar como el error va aumentando junto con la dimensión del problema, a su vez, como veremos en la siguiente sección el tiempo escala exponencialmente lo cual dificulta el análisis de mayores dimensiones, las soluciones las veremos en la sección de Trabajos Futuros.

Tiempo de ejecución

A continuación veremos los resultados de ejecutar los test vistos anteriormente en segundos. Cabe destacar la gráfica 22 en la cual podemos ver que ejecutar el script con mayor dimensión podría costar mucho tiempo al crecer exponencialmente.

Gen Size	25	Max Evals		125000
Problema	GA	DE	LS	MOS
Schewefel Problem	5.07E+01	5.52E+01	2.68E+00	3.62E+01
Shifted Ackley	5.27E+01	6.02E+01	3.77E+00	3.92E+01
Shifted Rastrigin	5.28E+01	6.07E+01	4.07E+00	3.95E+01
Shifted Rosenbrock	5.56E+01	6.56E+01	5.37E+00	1.14E+02
Shifted Griewank	5.57E+01	6.20E+01	4.14E+00	4.04E+01
Shifted Sphere	5.20E+01	5.42E+01	2.34E+00	3.49E+01
f9	5.58E+01	6.59E+01	5.44E+00	1.07E+02
f8	5.01E+01	5.40E+01	2.55E+00	9.45E+01
f10	5.30E+01	6.01E+01	3.89E+00	1.06E+02
f7	4.99E+01	5.38E+01	2.37E+00	9.68E+01
f11	5.43E+01	6.29E+01	4.55E+00	4.24E+01
Media	5.30E+01	5.95E+01	3.74E+00	6.83E+01

Cuadro 7: Resultados tiempo dimensión 25

Gen Size	50	Max Evals		250000
Problema	GA	DE	LS	MOS
Schewefel Problem	1.23E+02	1.49E+02	7.83E+00	9.43E+01
Shifted Ackley	1.31E+02	1.65E+02	1.16E+01	1.03E+02
Shifted Rastrigin	1.32E+02	1.67E+02	1.42E+01	1.05E+02
Shifted Rosenbrock	1.44E+02	1.89E+02	1.85E+01	3.15E+02
Shifted Griewank	1.34E+02	1.70E+02	1.30E+01	1.08E+02
Shifted Sphere	1.20E+02	1.41E+02	6.51E+00	9.88E+01
f9	1.45E+02	1.91E+02	1.85E+01	1.30E+02
f8	1.21E+02	1.42E+02	6.82E+00	1.12E+02
f10	1.34E+02	1.70E+02	1.29E+01	2.93E+02
f7	1.24E+02	1.44E+02	6.64E+00	2.52E+02
f11	1.40E+02	1.82E+02	1.54E+01	1.17E+02
Media	1.32E+02	1.65E+02	1.20E+01	1.57E+02

Cuadro 8: Resultados tiempo dimensión 50

Gen Size	100	Max Evals		500000
Problema	GA	DE	LS	MOS
Schewefel Problem	3.28E+02	4.38E+02	2.37E+01	2.59E+02
Shifted Ackley	3.60E+02	4.98E+02	3.99E+01	3.00E+02
Shifted Rastrigin	3.65E+02	5.10E+02	4.75E+01	3.18E+02
Shifted Rosenbrock	4.12E+02	5.96E+02	6.73E+01	9.52E+02
Shifted Griewank	3.71E+02	5.19E+02	4.54E+01	3.21E+02
Shifted Sphere	3.19E+02	4.14E+02	2.02E+01	2.33E+02
f9	4.14E+02	6.00E+02	6.57E+01	3.83E+02
f8	3.22E+02	4.14E+02	2.14E+01	2.90E+02
f10	3.69E+02	5.16E+02	4.40E+01	8.20E+02
f7	3.19E+02	4.13E+02	2.01E+01	6.84E+02
f11	3.90E+02	5.53E+02	5.48E+01	3.46E+02
Media	3.61E+02	4.97E+02	4.09E+01	4.46E+02

Cuadro 9: Resultados tiempo dimensión 100

Gen Size	200	Max Evals		1000000
Problema	GA	DE	LS	MOS
Schewefel Problem	1.00E+03	1.46E+03	8.39E+01	8.41E+02
Shifted Ackley	1.16E+03	1.74E+03	1.54E+02	1.04E+03
Shifted Rastrigin	1.23E+03	1.77E+03	1.74E+02	9.44E+02
Shifted Rosenbrock	1.34E+03	2.08E+03	2.57E+02	3.24E+03
Shifted Griewank	1.18E+03	1.78E+03	1.71E+02	1.14E+03
Shifted Sphere	9.76E+02	1.37E+03	6.87E+01	8.11E+02
f9	1.43E+03	2.24E+03	2.67E+02	1.36E+03
f8	1.06E+03	1.47E+03	7.88E+01	9.51E+02
f10	1.26E+03	1.92E+03	1.79E+02	2.94E+03
f7	1.04E+03	1.46E+03	7.29E+01	2.34E+03
f11	1.34E+03	2.00E+03	2.11E+02	1.16E+03
Media	1.18E+03	1.75E+03	1.56E+02	1.52E+03

Cuadro 10: Resultados tiempo dimensión 200

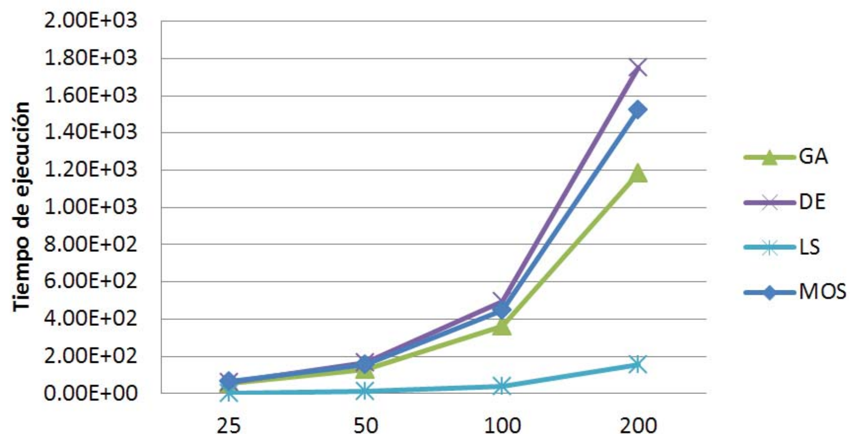


Figura 23: Gráfica del tiempo de ejecución medio por dimensión

Como resumen a esta parte, y analizando el gráfico 23 podemos decir que el algoritmo que mejor se comporta en términos de tiempo es el algoritmo de Búsqueda Local que incluso con la mayor dimensión analizada obtiene resultados muy buenos.
Al poder considerar el tiempo de cómputo como uno de los problemas más importantes de este *TFG* es necesario buscar posibles soluciones, con los análisis realizados a través de *profilers* podemos ver que este problema es provocado principalmente por el interprete de *Python* dado que es un lenguaje interpretado y cuenta con una serie de limitaciones a la hora de poder ser optimizado nativamente.
Hoy en día tenemos diferentes soluciones que pueden reducir el tiempo de cómputo hasta poder llegar a compararlo al que tendría en el código inicial en *C++*. Las diferentes soluciones son comentadas en la sección de Trabajos Futuros.

7. Conclusión y trabajos futuros

7.1. Conclusiones

Como conclusiones quiero destacar dos puntos que son los que más se han trabajado durante la realización de este *TFG*.

- El primero de estos puntos es el que está relacionado con los problemas de optimización en especial de los algoritmos metaheurísticos, como los que hemos visto en este *TFG*.
  Es asombroso ver como la abstracción de leyes biológicas pueden ser la solución a problemas que carecen de algoritmos de resolución matemática. En términos de implementación cabe destacar que son algoritmos bastante complejos de entender y sobretodo de implementar dado que tienen muchas características que los hacen únicos.
  Mi opinión personal sobre este tema es muy positiva dado que actualmente son usados en numerosas aplicaciones y los avances que se dan en el campo los hacen aún más importantes dado que su uso cada vez se da en más aplicaciones. Uno de los campos que más me han llamado la atención es el *Soft Computing* puesto que es uno de los campos de la informática que más avances está teniendo.

- El segundo punto es el estado actual de *Python*. Para mi es uno de los mejores lenguajes de la actualidad gracias a su sintaxis clara y a la multitud de librerías externas que lo dan soporte. Como hemos visto en términos de optimización no es uno de los mejores lenguajes, pero la comunidad actualmente está volcada en este tema para conseguir hacer de *Python* un lenguaje que pueda ser usado en problemas de optimización como los vistos en el desarrollo de este *TFG*.
  La posibilidad de usar compiladores estáticos puede suponer un avance muy prometedor para este lenguaje incluso los compiladores en tiempo real pueden ser una solución viable.

Si hablamos de las conclusiones a la hora de analizar los resultados podemos decir que no han sido todo lo buenos que fueron en su código en *C++*. Pero una vez analizados los resultados podemos reducir la búsqueda a los resultados obtenidos por el algoritmo DE, dado que es el que peores resultados obtiene. Una vez se solucione el problema posiblemente los resultados mejoren notablemente.

Para concluir me gustaría destacar el aprendizaje en el tema de la optimización que he recibido gracias al desarrollo de este *TFG*, dado que era uno de los objetivos propuestos. También la implementación de un sistema tan grande, como lo es *MOS*, me ha valido para ser capaz de resolver problemas que se van dando en un proyecto de estas características.

7.2. Trabajos futuros

Uno de los problemas más importantes a la hora de obtener resultados, fue el tiempo de ejecución, dado que Python en su forma estándar no está pensado para la optimización de código sería necesario barajar entre las diferentes opciones a la hora de optimizar código en Python.

La primera de las vías es usar un intérprete optimizado como por ejemplo *PyPy* [25] el cual realiza una compilación en tiempo real. Según la web oficial de *PyPy*, la mejora es notable en programas que no utilicen librerías externas de C y que el código contenga alta complejidad por lo que nuestro TFG sería teóricamente óptimo para realizar un estudio comparativo entre *PyPy* y Python.
A parte de la vía de la compilación *Just-In-Time* también existen diferentes opciones que están basadas en el estándar de Python como es *Cython*, que es un compilador estático de Python a C que obtiene muy buenos resultados usándolo correctamente [26].

También se puede destacar el uso de *Numpy* para su uso en la función a optimizar. Esta opción parece una de las menos esperanzadoras dado que la optimización solo se daría en una parte del TFG pero como hemos visto en la comparativa anterior [26], tiene una mejora de 31 veces el tiempo que se tardaba usando Python, además la función objetivo es una de las que más se ejecutan en el proyecto por lo que usar *Numpy* al final puede resultar en una mejora considerable.

Otro trabajo a parte que se podría realizar es el análisis de los valores atípicos mostrados en las gráficas vistas en la sección de Resultados y Conclusiones que comienza en la página 37.

Por último cabe mencionar completar los operadores implementados con los que hay en el código en C++ además de testear el sistema de genealogía para comprobar su correcto funcionamiento incluso implementar esta librería en *Cython* para así conseguir tanto optimizar tiempo como memoria consumida.

8. Anexo

Código fuente del proyecto

`https://gitlab.com/mamang126/tfg`

Referencias

[1] Marcus Randall. The future and applications of genetic algorithms. *Bond University*, 1997.

[2] Università di Padova. Elettrotecnica computazionale. `http://www.dii.unipd.it/~alotto/didattica/corsi/Elettrotecnica%20computazionale/DE.pdf`.

[3] LaTorre A. *A framework for hybrid dynamic evolutionary algorithms: multiple offspring sampling (mos)*. PhD thesis, Universidad Politécnica de Madrid, 2009.

[4] John Henry Holland. *Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Systems, The University Press of Michigan*. 1975.

[5] J. R Koza. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. 1990.

[6] L. O. Fogel. *Artificial Intelligence through Simulated Evolution*. 1966.

[7] Ingo Rechenberg. *Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. 1971.

[8] Krzysztof Krawiec and Mikołaj Pawlak. Genetic programming with alternative search drivers for detection of retinal blood vessels. *European Conference on Applications of Evolutionary Computation, EvoAPPS'15, At Copenhagen, Denmark*, 2015.

[9] Conor ; Medernach David Fitzgerald, Jeannie ; Ryan and Krzysztof Krawiec. An integrated approach to stage 1 breast cancer detection. *Gecco 2015*, 2015.

[10] Tetsuya Higuchi Hitoshi Iba, Sumitaka Akiba and Taisuke Sato. A bug-based search strategy using genetic algorithms. 1992.

[11] Ricardo A. Téllez and Cecilio Angulo. Generando un agente robótico autónomo a partir de la evolución de subagentes simples cooperativos. *V Workshop en Agentes Físicos Girona, libro de actas. Girona Edicions.*, pages 113–118.

[12] Francisco J. Soltero Pablo Fernández-Blanco, Diego J. Bodas-Sagi and J. Ignacio Hidalgo. Technical market indicators optimization using evolutionary algorithms. *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, pages 1851–1858.

[13] Jonas Dino. Evolvable systems. `https://www.nasa.gov/centers/ames/spanish/research/exploringtheuniverse/exploringtheuniverse-evolvablesystems.html`, 2008.

[14] Gary B. Lamont Kenneth S. Edge and Richard A. Raines. A retrovirus inspired algorithm for virus detection & optimization. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 103–110, 2006.

[15] Michael Collins. Probabilistic context-free grammars (pcfgs). http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/pcfgs.pdf, 2008.

[16] Luger G; Subblefield. *Artificial Intelligence. Addison Wesley.* 1983.

[17] SethBling. Mari/o - machine learning for video games.

[18] Carykh. Evolution simulator of creatures jumping vertically. https://www.youtube.com/watch?v=DTUlgZ2qLg8, 2016.

[19] Muhammad Khaerul Ardi Yusuf Priyo Anggodo, Amalia Kartika Ariyani and Wayan Firdaus Mahmudy. Optimization of multi-trip vehicle routing problem with time windows using genetic algorithm. *Journal of Environmental Engineering & Sustainable Technology*, 3(2):92–97.

[20] Stuart Russell and Peter Norvig. Inteligencia artificial: Un enfoque moderno. http://aima.cs.berkeley.edu/contents.html.

[21] Lin-Yu Tseng and Chun Chen. Multiple trajectory search for large scale global optimization. *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 2008.

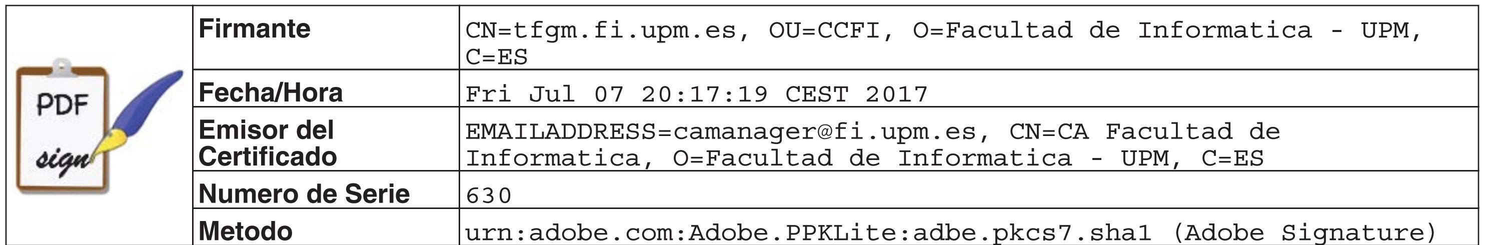
[22] Charles Darwin. *On the Origin of Species*. 1859.

[23] Suganthan P MacNish C Chen Y Chen C Yang Z Tang K, Yao X. Benchmark functions for the cec 2008 special session and competition on large scale global optimization. *Technical Report, Nature Inspired Computation and Applications Laboratory, USTC, China*, 2007.

[24] Santiago Muelas Antonio LaTorre and José-María Peña. A mos-based dynamic memetic differential evolution algorithm for continuous optimization. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 2010.

[25] Pypy. https://pypy.org/features.html.

[26] The performance of python, cython and c on a vector. http://notes-on-cython.readthedocs.io/en/latest/std_dev.html.



Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Fri Jul 07 20:17:19 CEST 2017
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)