Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Exploración del espacio de diseño multiobjetivo para balancear las características de ficheros binarios resultantes de la compilación mediante GCC

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Sergio Benlloch López
Tutor: David de Andrés Martínez

Curso 2020-2021

Resum

Donada la complexitat actual en el disseny de sistemes, incloent la varietat de components, interfícies de connexió, maneres d'operació, etc., cal fer una exploració sistemàtica de l'espai de disseny per poder determinar la configuració òptima per a uns requisits donats. Aquesta aproximació també es pot estendre a tots aquells àmbits en què les alternatives disponibles són massa nombroses com per considerar-les íntegrament. Un clar exemple el constitueix la compilació d'algoritmes descrits en C, ja que compiladors com GCC disposen de més de 200 opcions de compilació, que poden configurar-se a diferents nivells, i la combinació generarà diferents codis binaris amb diferents característiques.

En aquest treball s'han analitzat diferents estratègies d'exploració de l'espai de disseny multiobjectiu amb la finalitat de definir una metodologia que permeta determinar una configuració (sub-)òptima de les diverses opcions de configuració del compilador GCC que permeta optimitzar una sèrie d'objectius com a i) l'ús de memòria RAM, ii) la grandària del fitxer executable, iii) el temps d'execució de l'algorisme, iv) la utilització de la CPU i v) la robustesa del fitxer binari resultant, i permeta afegir fàcilment a futur nous objectius a optimitzar. Durant el treball s'analitzen diferents metodologies d'investigació operativa i es presenta la triada.

L'objectiu principal de l'eina creada és permetre als usuaris finals configurar adequadament GCC per optimitzar el binari resultant d'acord amb els objectius plantejats.

Paraules clau: Investigació operativa, Algorisme genètic, Exploració espai de disseny, Optimització, Compilació, MCDM.

Resumen

Dada la complejidad actual en el diseño de sistemas, incluyendo la variedad de componentes, interfaces de conexión, modos de operación, etc., es necesario realizar una exploración sistemática del espacio de diseño para poder determinar la configuración óptima para unos requisitos dados. Esta aproximación también puede extenderse a todos aquellos ámbitos en los que las alternativas disponibles son demasiado numerosas como para considerarlas en su totalidad. Un claro ejemplo lo constituye la compilación de algoritmos descritos en C, ya que compiladores como GCC disponen de más de 200 opciones de compilación, que pueden configurarse a diferentes niveles, y cuya combinación generará diferentes códigos binarios con diferentes características.

En este trabajo se han analizado diferentes estrategias de exploración del espacio de diseño multiobjetivo con el fin de definir una metodología que permita determinar una configuración (sub-)óptima de las diversas opciones de configuración del compilador GCC que permita optimizar una serie de objetivos como i) el uso de memoria RAM, ii) el tamaño del fichero ejecutable, iii) el tiempo de ejecución del algoritmo, iv) la utilización de la CPU y v) la robustez del fichero binario resultante, y permita añadir fácilmente a futuro nuevos objetivos a optimizar. Durante el trabajo se analizan diferentes metodologías de investigación operativa y se presenta la elegida.

El objetivo principal de la herramienta creada es permitir a los usuarios finales configurar adecuadamente GCC para optimizar el binario resultante de acuerdo a los objetivos planteados.

Palabras clave: Investigación operativa, Algoritmo genético, Exploración espacio de diseño, Optimización, Compilación, MCDM.

Abstract

Given the increasing complexity of current systems, their design space must be thoroughly explored to find the best possible configuration to meet a given set of constraints. This approach can also be applied to all those domains in which there exist too many alternatives to be completely analysed. For instance, compilers for C programs, like gcc, have more than 200 compilation flags that can be set to several levels and that will lead to binary codes with different characteristics.

This work presents a methodology that makes use of operational research techniques, like design space exploration and multi-criteria decision making, to enable the optimal configuration of gcc compilation flags to meet a set of goals like memory and CPU use, execution time, and size and robustness of the generated executable file, among others.

A tool has also been developed to support this methodology and enable final users to properly configure gcc to optimize the generated binary file according to selected goals.

Key words: Operational Research, Genetic Algorithm, Design Space Exploration, Optimization, Compilation, MCDM.

Índice general

Índice general v
Índice de figuras vii
Índice de tablas viii
Listings viii

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 2
1.3 Metodología . . . 2
1.4 Estructura de la memoria . . . 2
2 Contexto 5
2.1 Procesos de Optimización . . . 5
2.1.1 Diseño Factorial Completo. . . . 5
2.1.2 Diseño Factorial Fraccional. . . . 6
2.1.3 D-optimal. . . . 6
2.1.4 Algoritmos Genéticos. . . . 7
2.1.5 Redes Neuronales. . . . 8
2.2 Optimización multiobjetivo . . . 9
2.2.1 *Multiple-criteria decision-making* . . . 9
2.2.2 Normalización . . . 9
2.3 Optimización en el ámbito de los compiladores . . . 10
2.3.1 Herramientas que hacen uso de compiladores. . . . 11
2.3.2 Nuevos compiladores. . . . 12
2.4 Análisis y Conclusiones. . . . 13
3 Solución propuesta 15
3.1 *Setup* . . . 16
3.2 *Ejecución* . . . 17
3.3 *Análisis* . . . 17
4 Diseño de la solución 19
4.1 Herramientas de medición . . . 19
4.2 Método optimización de procesos . . . 23
4.2.1 Compilación . . . 24
4.2.2 Ejecución de pruebas . . . 24
4.2.3 Selección . . . 24
4.2.4 Cruce . . . 25
4.2.5 Mutación . . . 25
4.2.6 Solución . . . 26
5 Implementación 27
5.1 Herramientas utilizadas . . . 27
5.2 *Scripts* de medición . . . 28
5.3 Algoritmo genético . . . 31
5.3.1 Preparación . . . 31

5.3.2 Creación de la población inicial 33
5.3.3 Compilación 33
5.3.4 Ejecución de pruebas 35
5.3.5 Normalización y WSM 36
5.3.6 Selección 38
5.3.7 Límites de ejecución 38
5.3.8 Nueva generación 38
5.3.9 Salida 40
5.3.10 Conclusión 41
5.4 Herramientas de análisis y automatización de instalación 41

6 Experimentación **43**
6.1 *Setup* 43
6.2 Benchmarks 43
6.3 Experimentos y resultados 44
6.3.1 Evolución del tiempo de ejecución 44
6.3.2 Uso de Memoria RAM y Tiempo de ejecución 46
6.3.3 Optimización con todos los objetivos 49

7 Conclusión **53**
7.1 Relación con los estudios cursados 53
7.2 Conclusiones y trabajo futuro 53

Bibliografía **55**

Apéndices

A Versión Modelo de Islas **59**

B Manual de Uso **61**

C Opciones optimización GCC **63**

Índice de figuras

2.1 (a)Prototipo Diseño Antena evolutiva. (b)Estructura de un vehículo desarrollada por *National Crash Analysis Center*(NCAC) de la Universidad George Washington. 5
2.2 Flujo de funcionamiento de un Algoritmo Genético. 7
2.3 Flujo de cruce y mutación. 8
2.4 Normalización. 10
2.5 Flujo trabajo con un Compilador. 11
2.6 Diagrama Modelo De Islas. 12

3.1 Flujo de la Solución Propuesta. 15

4.1 Salida de *du -b <binario>*. 19
4.2 Salida de */usr/bin/time -f ' %e' <binario> <argumentos>*. 20
4.3 Salida de *Memory-profiler*. 20
4.4 Ejecución haciendo uso de *time* para medir la carga de la CPU. 21
4.5 Ejecución haciendo uso de *pidstat* para medir la carga de la CPU. 21
4.6 Salida de *Zero Overhead Fault Injector Project* (Zofi). 22
4.7 Diagrama funcional Algoritmo Genético. 23
4.8 Flujo módulo compilación. 24
4.9 Flujo módulo selección. 25
4.10 Flujo módulo cruce. 25
4.11 Flujo módulo mutación. 26

5.1 Representación fragmento de código para medición de tiempo de ejecución. 29
5.2 Patrón archivo *configparser*. 31
5.3 Representación Flujo creación estructura *flags*. 33
5.4 Flujo de compilación de individuos de la población. 34
5.5 Representación salida final ejecución. 40

6.1 Ejecución experimento 1, Tiempo de ejecución vs Generación. 45
6.2 Ejecución experimento 1, mínimo tiempo de ejecución frente a generación. 45
6.3 Ejecución experimento 2, WSM para el consumo de RAM con un peso de 0,4 y el Tiempo de ejecución con un peso de 0,6. 47
6.4 Tiempos de ejecución mínimos por generaciones. 47
6.5 Consumos de RAM mínimos por generaciones 48
6.6 Comparación final de opciones predeterminadas. 48
6.7 Ejecución experimento3, distribución WSM. 49
6.8 Evolución mínima consumo de RAM. 50
6.9 (a)Carga de la CPU. (b)Robustez del binario. 50
6.10 Evolución mínima tamaño del binario. 51
6.11 Evolución mínima tiempo de ejecución. 51
6.12 Algoritmo Genético vs Compilación sin optimización. 52

Índice de tablas

2.1 Experimentos por factor 6

6.1 Tabla de opciones. 44

C.1 Opciones Optimización GCC. 63

Listings

2.1 Pseudocódigo para Clipping. 10
5.1 *Script* medición tamaño de un binario. 28
5.2 Fragmento *script* medición carga de la CPU usando Pidstat. 29
5.3 Fragmento *script* medición carga de la CPU usando *time*. 29
5.4 Fragmento *script* medición Robustez. 30
5.5 Fragmento *script* medición RAM. 30
5.6 Patrón uso *configparser*. 31
5.7 Patrón uso *argparse*. 32
5.8 Estructura opciones en JSON. 32
5.9 Función executeWithOutput() 35
5.10 Función test(), lanzamiento pruebas de individuos usando *multithreading*. 35
5.11 Función pruebas(), ejecución pruebas. 36
5.12 Función normTiempo() como ejemplo de normalización completa. 37
5.13 Función normCpu() como ejemplo de clipping. 37
5.14 Función wsm() 38
5.15 Función selection() 38
5.16 Función crossover(). Proceso de creación de un individuo cruzado. 39
5.17 Función mutarComosoma(). Proceso de mutación. 39
B.1 Clonación repositorio 61
B.2 Instalación de dependencias. 61
B.3 Lanzamiento de ejecución AG. 61
B.4 Lanzamiento de ayuda AG. 62
B.5 Uso de programas auxiliares. 62

Siglas

AAV Arquitecturas Avanzadas

ACOVEA *Analysis of Compiler Options via Evolutionary Algorithm*

AE Algoritmos Evolutivos

AG Algoritmo Genético

AIC Arquitectura e Ingeniería de computadores

Clang *C Language compiler frontend*

CPA Computación paralela

CSD Concurrencia y Sistemas Distribuidos

CSV *Comma Separated Values*

DFC Diseño Factorial Completo

DFF Diseño Factorial Fraccional

DL *Deep Learning*

DSD Diseño de Sistemas Digitales

EDA Estructuras de Datos y Algoritmos

ETC Estructura de Computadores

FCO Fundamentos de Computadores

GCC *GNU Compiler Collection*

HET Hacking Ético

ICC *Intel C++ Compiler*

IIP Introducción a la Programación

LLVM *Low Level Virtual Machine*

LPP Lenguajes y entornos de programación paralela

LTP Lenguajes, tecnologías y paradigmas de la programación

LTS *Long term support*

MB *Megabytes*

MCDM *Multiple-criteria decision-making*

ML *Machine Learning*

MLGO *Machine Learning Guided Compiler Optimizations Framework*

PRG Programación

RRNN Redes Neuronales

TACT *Tool for Automatic Compiler Tuning*

WPM *Weighted Product Model*

WSM *Weighted sum model*

Zofi *Zero Overhead Fault Injector Project*

Agradecimientos

Me gustaría agradecer, en primer lugar, a David, por aceptar ser mi tutor, por aconsejarme en todo momento y resolver mis cientos de dudas. También a mis amigos, por compartir las dificultades de la carrera, apoyarnos unos a otros y compartir pasión por la informática. Y por último, a mi madre, a mi padre y a mi hermano, sin ellos todo esto hubiera sido imposible, por apoyarme durante mis estudios y por todo el apoyo que me han transmitido.

Mil gracias a todos.

CAPÍTULO 1
Introducción

Un compilador es una herramienta que permite llevar a cabo el proceso de convertir texto en instrucciones de nivel bajo para llevar a cabo una función. Comúnmente, los compiladores permiten a los usuarios que los usan cambiar el proceso de compilación. Es en este momento donde la entrada configurada puede mejorar el resultado de la salida final y, donde, dependiendo de la entrada podemos obtener un rendimiento diferente consiguiendo el mismo resultado funcional.

El resultado obtenido del proceso de compilación es un archivo binario, que representa en código binario las instrucciones más cercanas al sistema y el tipo de instrucciones que el procesador es capaz de llevar a cabo.

La optimización de binarios es un campo donde se intenta mejorar las características de un programa compilado respecto a algún objetivo y obtener una versión mejor respecto a la versión anterior o la versión normalmente utilizada.

La optimización multiobjetivo pretende resolver el problema de optimizar una solución en dirección a diferentes objetivos. Con esto se pretende presentar una solución que se ajuste a diferentes objetivos previamente definidos.

La optimización de un binario permite mejorar su rendimiento en diferentes objetivos. Si un binario es usado en una capa baja de un sistema o aplicación puede afectar en gran medida al resultado final del rendimiento de este mismo.

En este capítulo se incluye la introducción al trabajo realizado, donde podemos encontrar la motivación por la que se ha llevado a cabo el trabajo, los objetivos iniciales del mismo, las metodologías seguidas para llevar acabo la solución final y una estructura de la memoria realizada.

1.1 Motivación

Durante muchos años la tecnología a avanzado a pasos agigantados, obteniendo rendimiento de muchos formas. Los principales sistemas usan en sus cimientos grandes cantidades de código que debe ser compilado. Una pequeña mejora en el rendimiento del sistema puede afectar en gran medida al resultado final.

Existen muchas formas en las que medir la mejora en un programa, muchos objetivos y barreras a cumplir. Muchos trabajos intentan mejorar el tiempo de ejecución, haciendo que un proceso sea más rápido, pero no todos los objetivos a medir pueden consistir en el tiempo que tarda un proceso en llevarse a cabo. Se puede requerir medir otras características, o todas ellas a la vez, y no siempre con la misma importancia para cada característica a medir.

Existen muchos formas de optimizar un proceso, todas ellas recogidas en las técnicas de investigación operativa. Estos métodos se encargan de plantear posibles soluciones al problema de optimización.

Cada una de estas técnicas se aproxima de una forma a la solución y esta más indicada para un tipo de problema que otra. En este trabajo se van a estudiar un grupo de métodos para investigación operativa, eligiendo uno de ellos para llevar a cabo una propuesta de solución al problema de optimizar un binario respecto a diferentes objetivos, optimización multiobjetivo.

En este trabajo, también se van a estudiar los diferentes objetivos a optimizar, las múltiples herramientas para medirlos y todos los mecanismos necesarios para llevar a cabo una solución de un problema de optimización multiobjetivo.

1.2 Objetivos

El principal objetivo de este trabajo es presentar una propuesta de solución al problema de optimización multiobjetivo de programas compilados.

Proponemos diferentes objetivos para poder llevar a cabo el trabajo, estos son los siguientes:

- Repasar los diferentes métodos de investigación operativa para poder elegir uno de ellos para llevar a cabo la solución.
- Definir los diferentes objetivos a optimizar, elegir una herramienta para medirlos y llevar a cabo una solución poder usarlos junto al método de investigación operativa elegido.
- Repasar los diferentes métodos necesarios para llevar a cabo una optimización multiobjetivo, elegir los necesarios y implementarlos para poder ser usados por el método de investigación operativa.

1.3 Metodología

Para realizar el trabajo, se ha seguido una metodología *scrum*, donde se planteaba un pequeño problema o *sprint*, se resolvía, y se comprobaba la solución. Con esto se ha realizado todo el trabajo, subdividiendo un gran problema en pequeñas partes y iterando sobre ellas hasta llegar a la solución final.

Para el desarrollo de la aplicación, cabe añadir, que se ha hecho uso de herramientas de control de versiones como Git[1] y se ha usado como repositorio remoto a Github[2] , validando el resultado de cada *sprint* antes de ser unido al repositorio remoto.

1.4 Estructura de la memoria

En esta memoria podemos encontrar, en primer lugar, el contexto de las diferentes áreas de estudio de las que el trabajo forma parte.

Después, encontramos en diferentes capítulos el planteamiento de la idea llevado a cabo, desde una parte más creativa a una parte más funcional. Para seguir encontramos

[1]Disponible en: https://git-scm.com/
[2]Disponible en: https://github.com/

un capítulo relatando todas las partes de la implementación. Tras esto encontramos la experimentación llevada a cabo, y por último la conclusión.

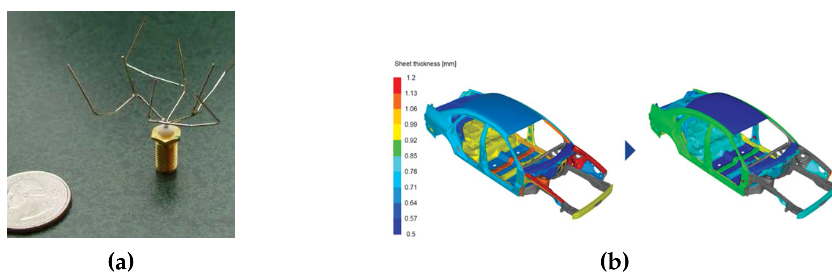
También cabe añadir que al final del documento podemos encontrar anexos con el manual de uso de la herramienta creada, con la lista de opciones de optimización disponibles en GCC y una breve explicación de otra versión de la herramienta.

CAPÍTULO 2
Contexto

Los problemas de optimización son problemas clásicos transversales a la ingeniería. Existen muchos métodos para llevar a cabo este proceso y la ingeniería informática ha facilitado su resolución . En este capitulo se van a repasar las diferentes técnicas conocidas en el ámbito de la optimización de procesos, y en la optimización de binarios, en particular haciendo uso de las herramientas proporcionadas por el compilador. Por último se va a presentar un análisis comparativo y sus respectivas conclusiones.

2.1 Procesos de Optimización

La Investigación Operativa es una disciplina estadística que se ocupa de la implementación de métodos analíticos para la toma de decisiones. Dentro de esta disciplina encontramos la optimización. En la industria y la ingeniería la optimización es un problema común que se centra en la selección del «mejor» elemento o el elemento óptimo de un conjunto. Podemos encontrar ejemplos de optimización en la elección de diseño de una antena para uso espacial [1] (Figura 2.1 a) o en el diseño de la estructura de un vehículo [2](Figura 2.1 b). En este punto vamos a repasar las técnicas actuales de optimización y en las que la computación ha intervenido.



(a) (b)

Figura 2.1: (a)Prototipo Diseño Antena evolutiva. (b)Estructura de un vehículo desarrollada por *National Crash Analysis Center*(NCAC) de la Universidad George Washington.

2.1.1. Diseño Factorial Completo.

Un Diseño Factorial Completo (DFC) [3] es un experimento con todas las posibles combinaciones de entradas, o factores, para un problema. Dentro de estos experimentos podemos encontrar los DFC a dos niveles, donde encontramos que los posibles factores pueden tomar dos únicos niveles que se denominan «alto» y «bajo» ó «+1» y «-1». Con esto podemos concluir con el cálculo en la Tabla 2.1 del número de experimentos relacionado con el número de factores.

Número de factores	Total de experimentos
2	4
3	8
4	16
...	...
k	2^k

Tabla 2.1: Experimentos por factor

Por lo tanto para un DFC común podemos concluir que habrá I^K experimentos, siendo I el número de niveles y K el número de factores. En este tipo de método se analizan todos las posibles combinaciones, por lo tanto es un método poco eficiente y muy costoso para un problema con un gran número de factores o niveles, aunque puede resultar en la elección de la mejor combinación de factores posible.

2.1.2. Diseño Factorial Fraccional.

Otro de los métodos que encontramos es el Diseño Factorial Fraccional (DFF) [4], método basado en DFC. En este caso este método facilita la experimentación con aquellas áreas donde nos encontramos un gran número de factores o niveles. ASQC(1983) *Glossary & Tables for Statistical Quality Control* [5] define el DFF como « Un experimento factorial en el que sólo se selecciona a una fracción adecuadamente elegida de las combinaciones de tratamiento requeridas para el experimento factorial completo». Esto facilita la experimentación, ya que con el DFC pueden requerir muchas ejecuciones resultando en un análisis costoso y en el DFF solo se requieren un subconjunto de estas.

En este caso al escoger un subconjunto de los factores podemos decir que habrá un total de I^{K-P}, siendo en este caso I el número de niveles, K el número de factores y $1/I^P$ el tamaño de la fracción con respecto a DFC [3].

Podemos encontrar aplicaciones de este tipo de experimentos en diversos campos de aplicación. Entre ellos podemos encontrar el artículo *Tuning synthesis flags to optimize implementation goals: Performance and robustness of the LEON3 processor as a case study* [6], donde se analiza el efecto de las opciones de implementación en el rendimiento y la robustez del procesador LEON3. En este artículo se hace visible la importancia de la elección correcta de los factores a analizar para que estén elegidos de forma equilibrada («la combinación de configuraciones de factores para cualquier grupo de factores tiene el mismo número de observaciones») y ortogonal («los efectos de cualquier factor se equilibran (suma cero) a través de los efectos de los otros factores»).

El mayor problema de este tipo de experimento es implementarlo para más de dos niveles al encontrarnos con un mayor espacio de diseño y haciéndolo más costoso.

2.1.3. D-optimal.

D-optimal [7] [8] surge para solucionar el gran costo de implementación para problemas que cuentan de más de dos niveles por factor. D-optimal facilita la resolución de problemas de optimización asimétricos. Es decir, D-optimal facilita la resolución para aquellos problemas que cada factor puede contar con un número diferente de niveles.

Según [7], «muchos de los diseños simétricos clásicos tienen características deseables, una de las cuales se llama D-optimalidad. El concepto de D-optimalidad también se puede aplicar para seleccionar un diseño cuando los diseños simétricos clásicos no se pueden utilizar, como cuando la región experimental no tiene forma regular, cuando el número de experimentos elegidos por un diseño clásico es demasiado grande o cuando

uno quiere aplicar modelos que se desvíen de los habituales de primer o segundo orden». D-optimalidad es uno de los posibles criterios para elegir una solución particular del espacio de diseño.

Uno de los ejemplos que podemos encontrar donde se utiliza D-optimal es el del artículo *Robustness-aware design space exploration through iterative refinement of D-optimal designs* [9], donde se usa este tipo de experimento para encontrar la mejor configuración de los parámetros ofrecidos por el *toolkit* EDA en cuanto a la robustez de una solución se refiere.

2.1.4. Algoritmos Genéticos.

Uno de los posibles métodos de optimización de procesos es el uso de un Algoritmo Genético (AG) [10]. Los AG son una solución basada en la evolución genética y forman parte de la familia de soluciones de los Algoritmos Evolutivos (AE) [10]. En este tipo de soluciones se sigue un esquema como el mostrado en la Figura 2.2.

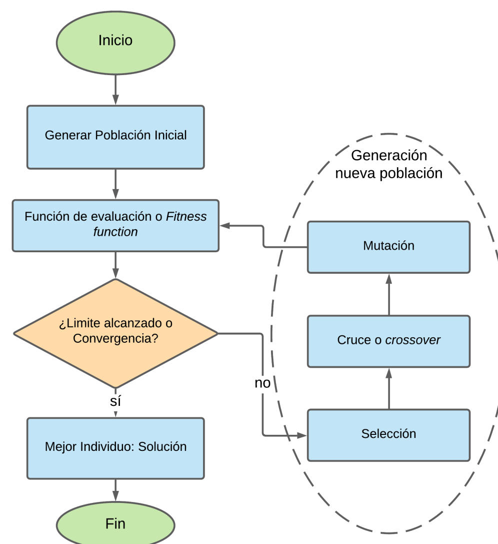


Figura 2.2: Flujo de funcionamiento de un Algoritmo Genético.

En el ámbito de los AG se utilizan los siguientes términos:

- **Individuo o cromosoma**: se llama de esta manera a cada una de las posibles soluciones del problema. Estos individuos están formados por **genomas**, que son cada una de las entradas y normalmente se codifican de forma binaria.
- **Población**: se denomina así a cada grupo de individuos.
- **Generación**: se denomina Generación a cada iteración del problema.
- **Cruce o *Crossover***: se denomina cruce a las operaciones utilizadas para obtener una nueva generación a partir de los individuos seleccionados de la anterior.
- **Mutación**: se denomina de esta manera a las operaciones de cambio que se realizan a cada individuo resultante del cruce. Normalmente estás operaciones son aleatorias y afectan a un único genoma o un pequeño grupo de genomas.

Como vemos muchos de estos términos son heredados directamente de un contexto biológico. Como forma de resumen de la Figura 2.2 podemos decir que un AG es la búsqueda evolutiva de una solución que consiga mejorar a la anterior o que consiga el límite

deseado por el usuario. Esta búsqueda consiste en la creación consecutiva de generaciones a partir de una generación aleatoria y la selección de los individuos que el usuario define como mejores, utilizando los mecanismos de cruce y mutación definidos por el usuario.

Una de las partes más cambiantes de este tipo de solución es el cruce y la mutación, en la Figura 2.3 podemos ver una de las posibles soluciones al cruce y la mutación. Normalmente la variación de estas soluciones consiste en número de individuos seleccionados, en el tamaño del grupo que se heredan de cada individuo y en la cantidad de genomas que se mutan. En esta figura vemos como se ha cogido un numero arbitrario para este ejemplo, en este caso el nuevo individuo hereda un genoma de cada uno de sus «padres» y tras esto se selecciona uno, normalmente aleatoriomente, para que cambie, puede acabar en el mismo resultado o en uno diferente. Estos mecanismos serán los que se tengan que adaptar de forma empírica a cada uno de los problemas.

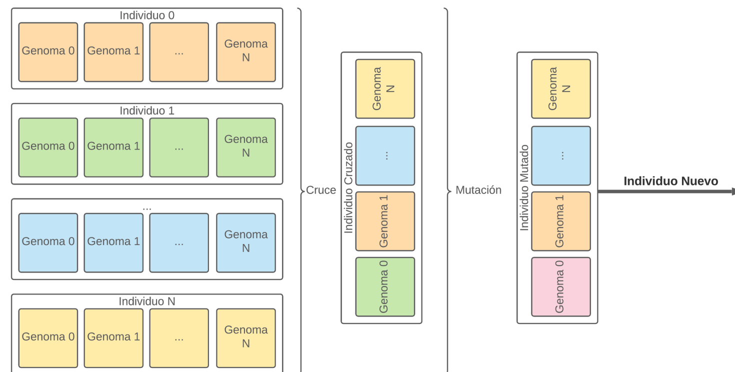


Figura 2.3: Flujo de cruce y mutación.

2.1.5. Redes Neuronales.

El campo de las Redes Neuronales (RRNN) [11] es inmenso, lleva avanzando exponencialmente en las ultimas décadas. Las RRNN pueden aplicarse a un gran número de problemas, entre ellos la optimización de procesos.

Existen diversas áreas que trabajan con RRNN, podemos encontrar áreas como *Machine Learning* (ML) o como el *Deep Learning* (DL), estas áreas se nutren de grandes conjuntos de datos con diversas soluciones preestablecidas para aprender de ellos y poder dar diferentes soluciones siguiendo la lógica de estos datos. El principal problema que se puede encontrar a la hora de utilizar este tipo de solución es la dificultad a la hora de crear el conjunto de datos, ya que puede ser costoso crear este *set* para entrenar la red neuronal. También puede resultar costoso computacionalmente, ya sea por el tiempo o por la cantidad de potencia de computación necesaria para hacer posible este entrenamiento.

Los ejemplos de uso de este método son muchos, pero entre los diversos artículos que tratan estas técnicas podemos encontrar artículos como *Neural network based optimization of drug formulations* [12], donde se hace uso de redes neuronales para optimizar la formulación de medicamentos. También podemos encontrar artículos como *Neural network based optimization approach for energy demand prediction in smart grid* [13], que hace un acercamiento desde la optimización haciendo uso de RRNN para predecir la demanda energética en redes inteligentes.

2.2 Optimización multiobjetivo

Hasta ahora se han presentado diferentes métodos para optimizar un problema, pero para poder optimizar un problema con diferentes objetivos a medir son necesarios mecanismos para poder medir todos esos objetivos. En este apartado se van a presentar algunos de los mecanismos a tener en cuenta a la hora de optimizar un problema multiobjetivo.

2.2.1. *Multiple-criteria decision-making*

Un *Multiple-criteria decision-making* (MCDM) [14] ayuda a la elección de una de las posibles soluciones dadas a partir de los resultados en cada uno de los criterios medidos. Existen múltiples MCDM pero no se puede saber cual de ellos es el mejor o más conveniente ya que esto es una paradoja [15], no se puede saber cual es el MCDM correcto porque para ello sería necesario un MCDM.

Entre los diferentes MCDM, podemos encontrar algunos como ASM, TOPSIS o VIKOR [16], entre ellos encontramos MCDM sencillos de implementar:

***Weighted sum model* (WSM)**

Podemos definir un resultado usando WSM con i alternativas y j criterios como el sumatorio para cada criterio del producto de w_j, que representa el peso del criterios j, por a_{ij}, que representa el resultado de la alternativa i en el criterio j:

$$A_i^{WSM-score} = \sum_{i=1}^{n} w_j * a_{ij}, \text{for i} = 1, 2, 3...m.$$

Con esto podremos comparar los valores y saber cual es mejor, el más grande si maximizamos o el más pequeño si minimizamos. Tenemos que tener en cuenta que todos los criterios deben ir en la misma dirección, en todos queremos maximizar o en todos queremos minimizar.

***Weighted Product Model* (WPM)**

En el caso de WPM, podemos definir la elección entre el resultado A_K y A_L (donde K representa el el resultado de una alternativa sobre un criterio y L otro alternativa sobre este mismo criterio) como el productorio de la división entre el resultado en el criterio j de A_K, representado como a_{K_j}, y el resultado de A_L en el criterio j, representado como a_{L_j}:

$$P(A_K/A_L) = \prod_{j=1}^{n} (a_{K_j}/a_{L_j})^{w_j}, \text{for K,L} = 1, 2, 3...m.$$

Si $P(A_K/A_L) >= 1$ indica que A_K es mejor que A_L, si es menor, que A_L es mejor que A_K.

2.2.2. Normalización

Para poder utilizar algunos MCDM, como WSM, sobre los resultados de diferentes objetivos estos deben estar en una misma escala, si estamos midiendo objetivos que tienen diferentes escalas debemos normalizar [17] los resultados.

La Normalización [17] es el proceso en el que los datos se transforman a una escala similar. Existen diferentes tipos, entre ellos podemos encontrar métodos como ranking,

estandarización, re-escalamiento o métodos de indicadores cíclicos [18]. Entre estos podemos encontrar métodos como escalamiento lineal, y muchos de estos se pueden encadenar con otro tipo de métodos que facilitan el trabajo de normalización.

Escalar a un rango

Consiste en escalar entre 0 y 1 todos los valores de los datos. Se utiliza la siguiente formula:

$$x' = (x - x_{min})/(x_{max} - x_{min})$$

Con esto los datos están distribuidos de la misma forma que los datos originales pero se encuentran escalados entre 0 y 1.

Clipping

Clipping no es un método de normalización, pero se asocia a estos métodos porque permite que se descarten los valores por encima o por debajo de unos valores dados, con esto podemos descartar los valores fuera de lo normal. Se podría implementar siguiendo el pseudocódigo del Algoritmo 2.1.

```
1 for (i=0; i< length(Data): i ++)
2     if Data[i] > max
3         Data[i] = max
4     else if Data[i] < min
5         Data[i] = min
6     end if
7 end for
```

Código 2.1: Pseudocódigo para Clipping.

Como vemos es la Figura 2.4, la normalización permite que se encadenen los diferentes métodos consiguiendo así las bondades de todos ellos, permite tener datos comparables y dentro de rangos normales.

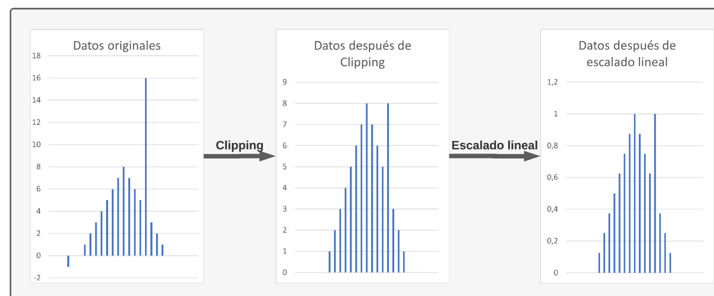


Figura 2.4: Normalización.

2.3 Optimización en el ámbito de los compiladores

Antes de nada, hay que repasar de forma sencilla como funciona un compilador. Como podemos ver en la Figura 2.5, en la entrada del compilador tenemos un archivo de texto de entrada que contiene el código fuente que al ser pasado como entrada a un compilador genera el código máquina o binario. Es en este proceso donde las herramientas de optimización de los compiladores afectan en el proceso.

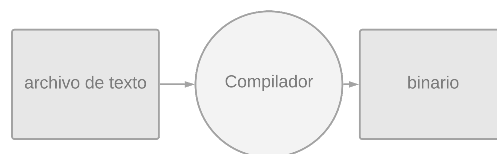


Figura 2.5: Flujo trabajo con un Compilador.

Los compiladores como *GNU Compiler Collection* (GCC)[1], *Intel C++ Compiler* (ICC)[2] o *C Language compiler frontend* (Clang)[3], son algunos compiladores que, entre otros lenguajes, compilan programas escritos con el lenguaje C.

Existen opciones, también llamadas *flags*, que nos permiten cambiar el proceso de compilación. Hay diferentes opciones que afectan cada una de una de ellas de forma diferente y en un momento de la compilación diferente. Esto permite generar código que cumple la misma función pero lo hace de forma diferente, y es aquí donde el resultado final puede mejorar en ciertos objetivos. Por defecto, compiladores como GCC dan diferentes opciones [19], entre ellas las de optimización, pensadas en un principio para mejorar el tiempo de ejecución, compilación o el tamaño del binario. También existen otras *flags* pensadas para conseguir otros objetivos, como las *flags* para ayudar a depurar el código o las pensadas para definir un estándar diferente de programación del lenguaje utilizado.

Estas opciones, entre otras cosas, reordenan el código o insertan nuevas funcionalidades al código. Estas opciones suelen ser utilizadas por diferentes herramientas para conseguir el mejor binario posible. Podríamos dividir la optimización en dos áreas, entre aquellas herramientas que hacen uso de compiladores ya existentes y sus respectivas opciones o aquellas herramientas que intentan mejorar el campo de los compiladores presentando uno nuevo. A continuación, vamos a repasar diferentes herramientas existentes y ver que métodos utilizan para conseguir su objetivo deseado.

2.3.1. Herramientas que hacen uso de compiladores.

Una de las herramientas más conocidas es *Analysis of Compiler Options via Evolutionary Algorithm* (ACOVEA) [20] [21] [22], esta herramienta *open source* hace uso de un AG para encontrar la «mejor» combinación de *flags* y obtener un binario más rápido o menos pesado pudiendo elegir el usuario que objetivos son los que el algoritmo debe medir. El autor de esta herramienta hace uso del Modelo de Islas, o como el dice, hace que el AG siga el patrón de las «manadas de leones» [21], donde hay diferentes grupos poblacionales y cada ciertas generaciones hay un intercambio entre manadas. Según el autor con esto intenta evitar recaer en máximos locales e intentar expandir la búsqueda por todo el espacio de diseño. Podemos ver en la Figura 2.6 como funciona a la hora del intercambio, en este caso se ha escogido 3 individuos a intercambiar pero podría variar.

[1]Disponible en: https://gcc.gnu.org/

[2]Disponible en: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html

[3]Disponible en: https://clang.llvm.org/

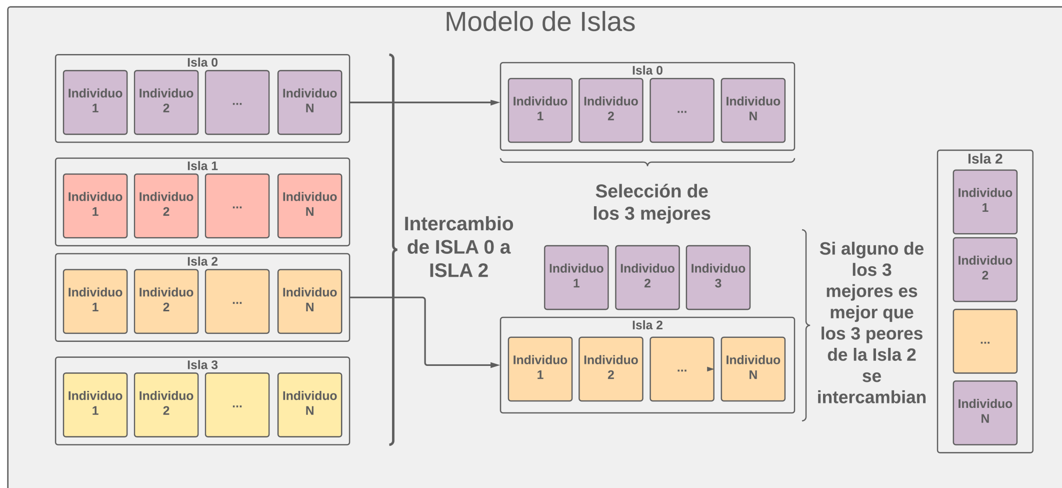


Figura 2.6: Diagrama Modelo De Islas.

Tras esta herramienta, Dmity Plotnikov et al publicaron un artículo donde presentaban la herramienta *Tool for Automatic Compiler Tuning* (TACT) [23]. Esta herramienta basándose en ACOVEA hace uso de una búsqueda evolutiva para encontrar la combinación de opciones Pareto-óptima para programas escritos en C y que van a ser ejecutados en la arquitectura ARMv7. En este caso están midiendo simultaneamiento el tiempo de compilación y el tiempo de ejecución, aunque indican que mediante *scripting* se pueden implementar más objetivos.

Pareto-frontier permite dar a conocer los mejores individuos de, en este caso, un AG. El principal problema o característica de Pareto-frontier es que no elige cual es el mejor individuo, si no que presenta las configuraciones que no puedan mejorar en un objetivo sin empeorar algún otro.

Siguiendo esta línea de investigación, Luis Alberto Vivas Tejuelo et al [24] presentaron un trabajo donde, basándose en las anteriores herramientas, introducían una herramienta que haciendo uso de un AG busca un conjunto de «flags» óptimo. Este trabajo introduce la novedad de que siendo desplegado en un clúster, puede hacer uso del modelo de islas para no saturar la red y tener un «isla» en cada máquina del clúster, consiguiendo con esto acelerar el análisis de las poblaciones.

2.3.2. Nuevos compiladores.

Pasando a otro tipo de técnicas, podemos encontrar MILEPOST [25], una herramienta «open source» que introduce un compilador basado en ML, y que devuelve ya un binario óptimo al pasarle un archivo de texto. Esta herramienta optimiza el resultado en cuanto a tiempo de ejecución, compilación y tamaño de programa se refiere y de forma simultánea.

Siguiendo con el ML, encontramos una de las herramientas más tempranas, *Machine Learning Guided Compiler Optimizations Framework* (MLGO) [26]. Esta herramienta presenta una solución que intenta optimizar el resultado de las compilaciones remplazando las heurísticas de optimización que implementa el *backend* de compilación *Low Level Virtual Machine* (LLVM) [27] por una solución que hace uso de ML. En este caso, esta solución se centra en la reducción del tamaño del binario resultante, consiguiendo alrededor de un 7 % de mejora.

2.4 Análisis y Conclusiones.

Para comprender la magnitud del problema de optimizar el resultado de una compilación usando las opciones del propio compilador antes hay que estudiar estas opciones. Para ello, a continuación, se presenta un análisis de las opciones de optimización de, en este caso, uno de los compiladores más utilizados hoy en día, como puede ser GCC.

En este análisis solo se van a tener en cuenta las *flags* de optimización recogidas en la documentación de GCC. Lo primero que debemos tener en cuenta son los tipos de flags que nos encontramos. Nos encontramos primero dos grupos de flags, las preestablecidas por GCC, como podrían ser *-O2* o *-O3*, estas opciones realmente representan un conjunto de opciones más amplio y en este caso no se van a tener en cuenta ya que se van a analizar cada *flag* individualmente y sin usar las preestablecidas por el compilador.

Con esto nos encontramos con el segundo grupo, que a su vez se pueden dividir en 3 subgrupos que a continuación se van a detallar.

- Flags Simples: nombramos *flags* simples a aquellas opciones que sólo tienen dos niveles. Es decir, están o no están. Es el grupo más grande y recoge a la mayoría de opciones.
- Flags Numéricas o de Rango: este grupo representa aquellas opciones que van acompañadas de un número que indica la profundidad de su acción. Entre estas opciones podemos encontrar algunas como *-funroll-loops=N*, es en este caso donde *N* puede ser sustituido por un número entero, comúnmente entre 0 y 65536. Por lo tanto cuentan con, como mínimo, de 65536 niveles, ya que normalmente el 0 o el 1 es equivalente a no poner esta *flag*.
- Flags Algorítmicas o de Intervalo: como último grupo estas opciones cuentan con una lista de posibles ajustes a acompañar la *flag*, por lo tanto contaran con tantos niveles como opciones tenga la lista, más uno, el no estar. Este elemento de la lista normalmente indica un algoritmo a seguir por GCC aunque no siempre es así.

Una vez presentados los tipos de flag se puede hacer una aproximación al problema, en este caso cogiendo como ejemplo todas las opciones incluidas en las opciones preestablecidas por GCC podemos hacer el siguiente cálculo. Sabiendo que este conjunto de opciones está formado por 116 *flags*, de las cuales 104 son simples con sólo 2 niveles, 4 son de rango con 65536 niveles que se podrían recortar en 5 niveles, acotado como los niveles que representan el mínimo, un cuarto, un medio, 3 cuartos y máximo, y 8 son de intervalo, de las cuales 4 tienen 3 niveles, las otras 4 otros 4 niveles, y siendo T el tamaño del espacio de diseño, podemos decir que:

$$T = 2^{104} * 5^4 * (3^4 * 4^4) = 2{,}628x10^{38}$$

Aquí vemos representado en el primer grupo de la suma las opciones Simples, en el segundo las de Rango y en el tercero las de Intervalo. Esto hacen un total de 200 sextillones de opciones, algo que claramente se puede considerar un espacio de diseño grande. Este espacio es variable, ya que cogiendo otras opciones el espacio puede reducirse o hacerse más grande.

Es con este análisis con el que vemos que métodos clásicos como DFC o DFF quedan descartados rápidamente, ya que por un lado el problema cuenta con un amplio espacio de diseño, cuenta con factores de más de dos niveles y con un espacio de diseño asimétrico.

Otra de las opciones estudiadas es el ML, el principal problema de usar este método es el problema de crear un conjunto de datos representativo, algo que llevaría mucho

trabajo y sería necesario hacerlo con *benchmarks* representativos de todos los ámbitos de uso que se deseen.

Por último, vemos porque gran parte de las herramientas preexistentes que hacen uso de las opciones de los compiladores usan los *AG*, porque, según [21], la habilidad de un AG es la búsqueda de una solución óptima en un espacio de diseño grande y este problema lo representa a la perfección.

CAPÍTULO 3
Solución propuesta

Como veíamos en el capítulo anterior, existen múltiples problemas y métodos de optimización. En este caso se quiere encontrar la mejor combinación de opciones de compilación respecto a unos objetivos. A continuación, en el diagrama 3.1 podemos ver el flujo de la solución propuesta, tras ello encontramos un análisis pormenorizado de cada una de las partes del problema y que novedades contempla.

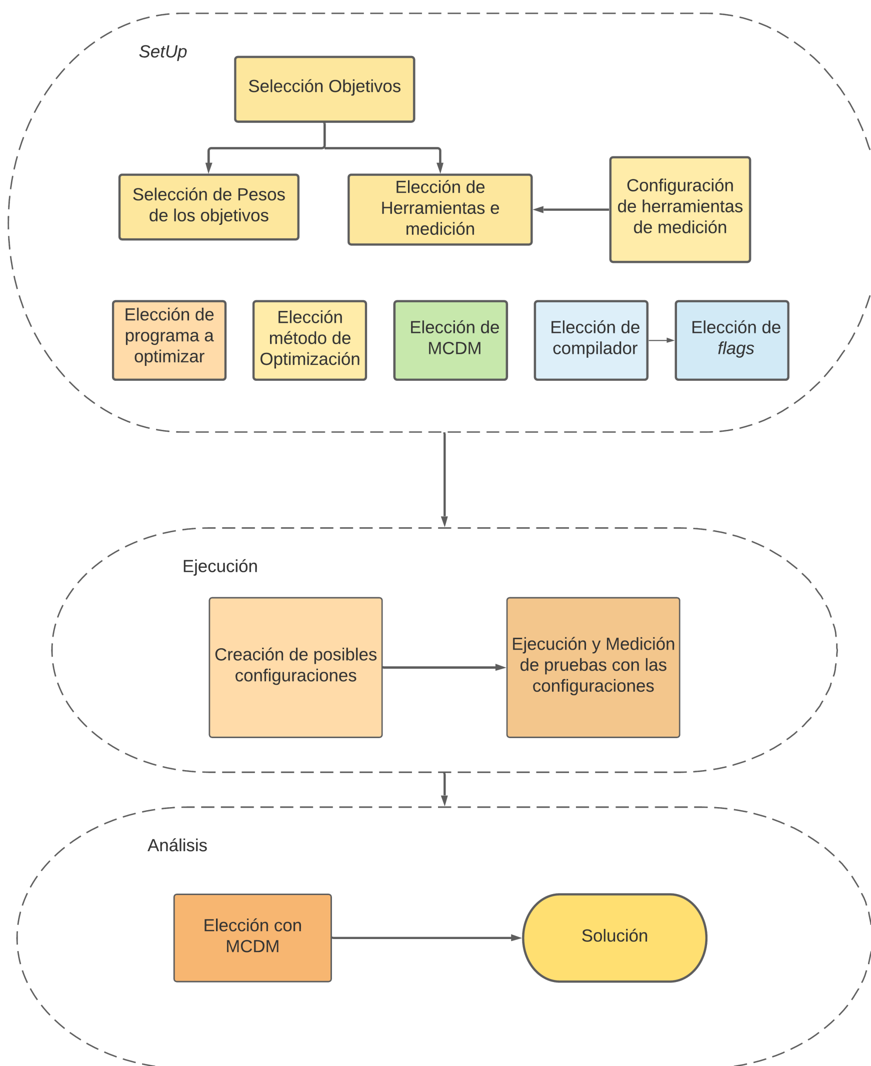


Figura 3.1: Flujo de la Solución Propuesta.

La Figura 3.1 representa una descripción a nivel alto de la idea, donde en cada una de las partes se describe los problemas a abordar. A continuación, se presentan cada una de las partes y se describe cada uno de los pequeños problemas o prerrequisitos a tener en cuenta.

3.1 *Setup*

En esta sección se abordan todos los prerrequisitos o problemas de configuración que surgen a la hora de plantear el problema. Como podemos ver en la Figura 3.1, lo primero que habría que abordar es la selección de objetivos.

En un principio se pensaron diferentes objetivos que podrían medirse a partir de un binario. Se repasaron muchos objetivos, como el tiempo de ejecución de un programa o su uso de memoria. Con esto podemos realizar la siguiente lista de objetivos:

- **Tiempo de ejecución**: como veíamos en el capítulo anterior, es uno de los objetivos principales que otras aplicaciones abordan. Las opciones de compilación dirigidas a optimización están pensadas para mejorar el tiempo de ejecución entre otras cosas, por lo tanto se espera encontrar mejora a la hora de analizar las diferentes opciones de compilación.

- **Tamaño del binario**: los lenguajes compilados tienen como resultado un binario ejecutable. El tamaño de este binario es otro de los objetivos que las opciones de optimización pretenden mejorar.

- **Consumo de memoria**: el consumo de memoria consiste en la medición de la memoria RAM usada por el programa. Se puede enfocar este objetivo a la medición del total de memoria usada, de la media de memoria usada por unidad de tiempo (total de memoria / tiempo de ejecución), pero en este caso decidimos medir el máximo de memoria usada, ya que un programa podrá ser ejecutado o no respecto a la memoria instalada en un ordenador y el máximo de memoria que utilice el binario a ser ejecutado.En este caso, para este objetivo no hay opciones exclusivamente indicadas para mejorar el consumo.

- **Carga de uso de la CPU**: este objetivo consiste en la medición de la carga de la CPU, se plantea calcular la media de consumo. Este objetivo es útil entre otras cosas por un lado para poder tener más instancias de un programa en ejecución en paralelo intentando que no se interrumpan una a otra.

- **Robustez del programa**: la robustez de un programa consiste en la habilidad de dicho programa a superar errores durante la ejecución o problemas en la entrada de datos [28]. En este caso se va a medir el porcentaje de ejecuciones de un programa que han superado con éxito un error, es decir lo han enmascarado.

Una vez presentados los posibles objetivos a medir, aunque se deberían poder añadir más objetivos fácilmente, se plantea la posibilidad de otorgar diferentes pesos, para así optimizar el resultado respecto a estos, y pudiendo así elegir uno o varios objetivos. También se plantea la posibilidad de tener diferentes herramientas de medición de cada objetivo y poder intercambiarlas sin muchas dificultades.

Otro de los pasos a tener en cuenta en el caso de una optimización multiobjetivo es la elección de MCDM, ya que se pretende que la elección sea totalmente automática y los criterios ya estarían definidos por los pesos planteados anteriormente. Se plantea la posibilidad de poder elegir uno entres múltiples MCDM existentes. Aunque, como se

comenta en el capítulo anterior, es una paradoja saber cual MCDM es el mejor, se plantea la posibilidad de cambiarlo fácilmente.

También se debería poder elegir el compilador a usar, en este caso la optimización va dirigida a programas compilados, en un principio escritos en C, pero se podría adaptar rápidamente a otros compiladores y con ello a otros lenguajes de programación compilados. Otra cosa a tener en cuenta es que también se podría cambiar el subconjunto de opciones seleccionadas para la optimización.

Por último, aparte de elegir el programa a optimizar, también hay que tener en cuenta el método de optimización elegido, en este caso solo se va a implementar uno, pero con las piezas existentes se podría adaptar a cualquier método. Con esto también se podría configurar los parámetros de este método haciéndolo cambiante y con múltiples opciones.

Como vemos, la idea es totalmente modular, se pretende permitir cambiar entre diferentes opciones y haciendo así adaptaciones a diferentes problemas.

3.2 *Ejecución*

Tras haber definido la configuración del problema hay que plantear la forma que tendría la ejecución de este mismo. En este caso, como veíamos en la Figura 3.1, la primera parte a plantear es el análisis que, por lo general, hacen los métodos de optimización de un subconjunto de las posibles configuraciones seleccionadas de una forma representativa de todas las opciones. En algunos casos se utilizan reglas para seleccionar este subconjunto de configuraciones y en otros se basan en heurísticos.

A la hora de ejecutar la prueba dependerá de las herramientas de medición y de los objetivos seleccionados. Normalmente la herramienta hará uso de la configuración deseada y devolverá un resultado.

3.3 *Análisis*

Tras la obtención de los resultados de las diferentes configuraciones, hay que elegir la más óptima respecto a los pesos configurados, para ello se utilizará el MCDM elegido.

Dependiendo del MCDM elegido habrá que normalizar los resultados. En este caso no es lo mismo estar hablando del tamaño del binario que puede estar fácilmente en una escala de 500000 *bytes* que de un porcentaje de uso de CPU que ronda entre 0 y 1.

También hay que tener en cuenta que se busca maximizar el resultado de las pruebas o minimizarlo, en los objetivos de Tamaño del binario, Memoria utilizada, Carga de CPU y Tiempo de ejecución se busca el mínimo, es decir minimizar, pero en el caso de la robustez se busca el máximo.

Aunque es más intuitivo maximizar, para minimizar bastaría invertir el porcentaje de robustez de cada ejecución Se puede calcular como 1 menos el porcentaje de enmascarados y ya se podría pasar por el MCDM que lo necesite.

Como veíamos en el capítulo 2, MCDM como WSM necesita de normalización o algunos como WPM no la necesitan. Por lo tanto tras normalizar o no podríamos ejecutar el MCDM teniendo como criterio los pesos seleccionados por objetivo y devolver la configuración más óptima.

CAPÍTULO 4

Diseño de la solución

En este capítulo se va a repasar la solución propuesta de una forma mas detallada, teniendo en cuenta un detalle funcional y definiendo cada una de las partes del problema a resolver,

4.1 Herramientas de medición

Una de las primeras partes que hay que definir son las herramientas de medición, para ello se van a ir repasando cada objetivo y enumerando las posibles alternativas y problemas encontrados en cada una de ellas y por último se presentará la elegida. Pero primero, para saber en que tipos de herramientas buscar, hay que definir el sistema. Se contempla la idea de utilizar una distribución basada en GNU/Linux, más concretamente se decide utilizar **Ubuntu 20.04** en su versión *Long term support* (LTS)[1], una de las distribuciones más usadas de GNU/Linux. Por lo tanto, todas las herramientas seleccionadas tienen que estas disponibles en este tipo de sistema.

A continuación, se va a presentar una lista con cada uno de los objetivos y las alternativas analizadas, muchos de estos objetivos se podrían medir de muchas maneras pero se presenta también la herramienta elegida.

- **Tamaño del binario**: en este caso se valoran dos alternativas, el uso de *size*[2] y el uso de *du*[3]. Ambas herramientas devuelven el tamaño, *size* lo devuelve desglosado por partes y *du*, dependiendo de las opciones que se le pasen devuelve el tamaño total en diferentes formatos. Se podrían utilizar cualquiera de las dos alternativas, pero se decide usar *du* por la sencillez para quedarse con el tamaño en *bytes* solamente. Más concretamente se decide usar la orden *du -b <binario>*, que devuelve el tamaño del binario en *bytes* y muestra una salida como la de la Figura 4.1. Como vemos, la salida esta formada por el número de *bytes* a la izquierda y el nombre del archivo.

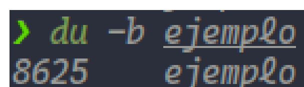


Figura 4.1: Salida de *du -b <binario>*.

- **Tiempo de ejecución**: en este caso se decide usar la orden ***time***[4] de Linux. Pero antes hay que saber que existen diferentes alternativas de la orden *time*, ya que *Bash*[5],

[1]Disponible en: https://releases.ubuntu.com/20.04/
[2]Manual disponible en: https://linux.die.net/man/1/size
[3]Manual disponible en: https://linux.die.net/man/1/du
[4]Manual disponible en: https://man7.org/linux/man-pages/man1/time.1.html
[5]Manual disponible en: https://www.gnu.org/software/bash/manual/bash.html#What-is-Bash$_0$03f

la *shell* por defecto de GNU, implementa su propia orden *time*. Pero, en la mayoría de sistemas Linux existe otra implementación de *time*, disponible si la ejecutamos como ***/usr/bin/time***. Es esta la alternativa que decidimos usar ya que es común a la mayoría de distribuciones de Linux y es la que viene documentada en las *man pages*. Mas concretamente se decide utilizar la orden */usr/bin/time -f ' %e' <binario> <argumentos>* que muestra una salida como la de la Figura 4.2. En este caso podemos ver simplemente la duración de la ejecución en segundos sin más detalles, pero hay que tener en cuenta que si el binario que se quiere medir tiene salida esta salida también se mostrará.

```
❯ /usr/bin/time -f '%e' sleep 5s
5.00
```

Figura 4.2: Salida de */usr/bin/time -f ' %e' <binario> <argumentos>*.

- **Consumo de memoria**: para este objetivo se evaluaron muchas herramientas, y el principal problema que encontramos es la diferencia de resultados entre ellas. Se evaluaron herramientas como *Valgrind* [29], pero al final se decidió usar ***Memory-profiler*** [30], ya que devuelve un archivo con el consumo de memoria de un binario cada cierto tiempo. En un principio se creó para ya que un perfil de uso de memoria para programas Python, pero se puede hacer servir para medir el uso de memoria de un binario cualquiera.

  Esta herramienta se puede usar como en la Figura 4.3. En esta figura se puede ver la llamada a *Memory-profiler* con su abreviación *mprof*, indicando el modo de ejecución y el binario a ejecutar. Como se puede ver la salida en este caso esta silenciada, pero también se vería la salida del programa. Por otro lado, esta el archivo creado, que contiene por linea el objetivo medido, el resultado en *Megabytes* (MB) y la marca de tiempo de la medición.

```
❯ mprof run ./ejemplo &>/dev/null
❯ head -n10 mprofile_20210819161323.dat
CMDLINE ./ejemplo
MEM 0.574219 1629382403.4951
MEM 0.574219 1629382403.5959
MEM 0.574219 1629382403.6966
MEM 0.574219 1629382403.7972
MEM 0.574219 1629382403.8978
MEM 0.574219 1629382403.9984
MEM 0.574219 1629382404.0988
MEM 0.574219 1629382404.1993
MEM 0.574219 1629382404.2998
```

Figura 4.3: Salida de *Memory-profiler*.

- **Carga de la CPU**: En este caso se evaluaron herramientas como Psrecord [31], que rápidamente se descartó por mostrar resultados totalmente fuera de lo común, como pueden ser resultados por encima del 100 % para programas que sólo hacen uso de un núcleo. También se evaluó el uso de ***time***, en su implementación de */usr/bin/time*.

  Otra de las herramientas que se evaluó fue Pidstat, implementada en el paquete de aplicaciones *Sysstat* [32]. Esta herramienta funciona de forma similar a como funciona *Memory-profiler*, haciendo mediciones dado un intervalo de tiempo.

En este caso se observó que las mediciones con *time*, no eran muy exactas, ya que otras de las soluciones mostraban un resultado más ajustado. En la mayoría de las mediciones, *time* devuelve un resultado poco preciso al no hacer uso de decimales, pero devuelve resultado. Hay un pequeño número de ejecuciones muy rápidas, de unos pocos milisegundos que no devuelve resultado. Pero *pidstat* no devolvía resultado para aplicaciones con un tiempo de ejecución pequeño, de alrededor de un segundo, por lo tanto se decidió crear una solución con las dos, para que se usara *time* en los casos que *pidstat* no devolvía resultado y teniendo fallo en las ejecuciones muy rápidas de unos pocos milisegundos ya que no se han encontrado aplicaciones capaces de medir esta carga.

Este es uno de los principales problemas encontrados a la hora de buscar herramientas para medir ciertos objetivos: se pueden encontrar muchas herramientas pero no todas devuelven los mismos resultados y no todas se encuentran bien documentadas. En este caso se eligió usar estas dos herramientas pero sería necesario hacer un estudio más amplio de las herramientas o incluso implementar una nueva, algo que se sale de los objetivos de este trabajo.

Como ejemplo de las ejecuciones encontramos la Figura 4.4, que muestra una ejecución con *time* y la Figura 4.5 que muestra el resultado haciendo uso de *pidstat*.

```
❯ /usr/bin/time -f '%P' ./ejemplo 1>/dev/null
99%
```

Figura 4.4: Ejecución haciendo uso de *time* para medir la carga de la CPU.

```
❯ pidstat 1 -u -e ./ejemplo
Linux 5.11.0-27-generic (portatil)     19/08/21       _x86_64_       (8 CPU)

16:55:17      UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
16:55:18     1000     21682   99,00    0,00    0,00    0,00   99,00     3  ejemplo
16:55:19     1000     21682  100,00    0,00    0,00    0,00  100,00     3  ejemplo
16:55:20     1000     21682  100,00    0,00    0,00    0,00  100,00     3  ejemplo
16:55:21     1000     21682  100,00    0,00    0,00    0,00  100,00     1  ejemplo

treebench (Std. C) run time: 4.610459


Average:     1000     21682   99,75    0,00    0,00    0,00   99,75     -  ejemplo
```

Figura 4.5: Ejecución haciendo uso de *pidstat* para medir la carga de la CPU.

- **Robustez**: para este caso se hizo una búsqueda por Github, donde se buscaban inyectores de fallos para binarios, porque es común encontrar herramientas de este tipo para FPGAs u otro tipo de sistemas empotrados, pero no para un sistema basado en Linux.

  Con esta búsqueda se encontró *Zero Overhead Fault Injector Project* (Zofi) [33], una herramienta de inyección de fallos. Esta herramienta se puede instalar fácilmente mediante los pasos indicados en [33] y muestra un resultado como el de la Figura 4.6.

  En esta Figura se puede ver que la aplicación permite configurar tanto el número de ejecuciones, como el de inyecciones pero, por el momento, solo se puede una ya que si no muestra un error al no estar implementada esta opción.

Tras lanzar la aplicación, podemos ver como al final se muestran los resultados, donde se muestra el porcentaje de ejecuciones enmascaradas, es decir que han tenido éxito, ejecuciones que se han vuelto infinitas, ejecuciones en las que la salida se ha corrompido, o ejecuciones que han devuelto una excepción. Se puede encontrar más información del funcionamiento de esta herramienta en [34].

```
❯ zofi -bin ejemplo -test-runs 97 -injections-per-run 1
------ Options ------
 -bin                    = ejemplo
 -injections-per-run     = 1
 -test-runs              = 97
---------------------
-- Original (Timing) Run --
  0.17r/s        1/1        : 0%======25%=======50%======75%=======100%
Original Duration: 6.05s

-- Test Runs --
  0.19r/s       97/97       : 0%======25%=======50%======75%=======100%

-------------------------------
Outcome          : Cnt,     %
-------------------------------
Masked           :   0,   0.000%
Exception        :  47,  48.454%
InfExec          :   0,   0.000%
Corrupted        :  50,  51.546%
```

Figura 4.6: Salida de Zofi.

Con esto estarían todas las herramientas necesarias para cada unos de los objetivos a medir. Pero, como podemos ver, algunas herramientas no solo devuelven el valor, también devuelven otros datos que para nuestro caso no son necesarios. En el capítulo de implementación se mostrará como se han creado ciertos *Bash scripts* que hacen uso de estas herramientas para devolver solamente una cifra con el resultado de la prueba.

4.2 Método optimización de procesos

Como se vio en el capítulo 2, existen múltiples métodos de optimización de procesos, pero en este caso se ha elegido un AG como método indicado, al encontrarnos con un espacio de diseño muy irregular y muy grande y al ser un AG el indicado para ello.

A continuación, podemos ver la Figura 4.7 que muestra la estructura del AG donde se ve representado el diagrama funcional del método de optimización elegido.

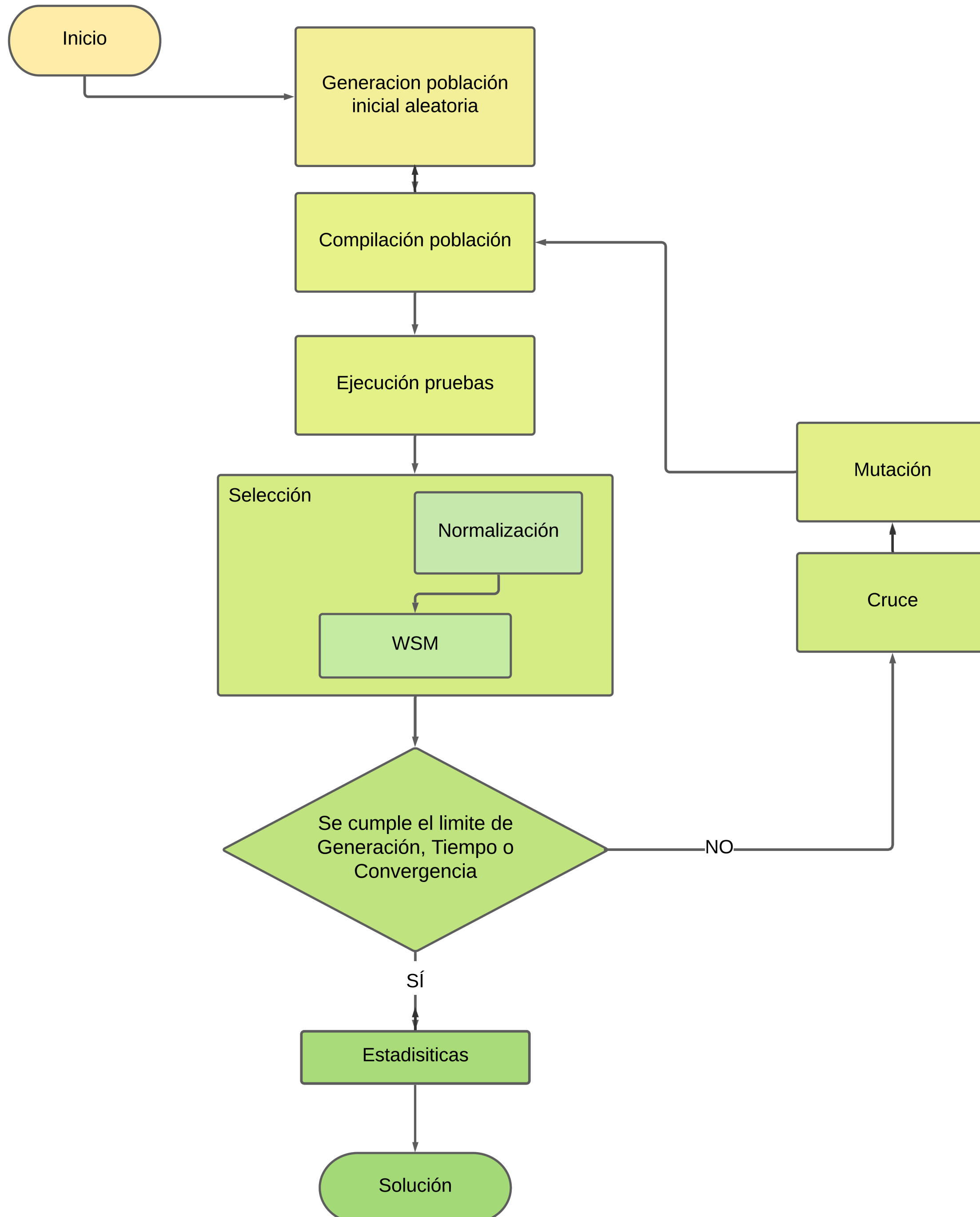


Figura 4.7: Diagrama funcional Algoritmo Genético.

A continuación, encontramos las diferentes partes y el razonamiento de porqué se ha elegido cada una de ellas para la construcción de este algoritmo. Pero como vemos en la Figura 4.7, el algoritmo está pensado de forma modular para que sea sencillo intercambiar una parte de él por otra, consiguiendo así una finalidad o funcionalidad diferente.

4.2.1. Compilación

Una vez generada la población aleatoria se pasa a compilar esta población para poder calcular los diferentes objetivos seleccionados.

En este caso se ha elegido hacer uso de GCC, un compilador diseñado por GNU y con múltiples opciones de optimización.

Como se indicaba anteriormente, se podría sustituir esta parte del algoritmo por otra con otro compilador. Para poder hacerlo habría que asegurarse de cumplir el esquema indicado en la Figura 4.8, pero una vez asegurado ese funcionamiento bastaría.

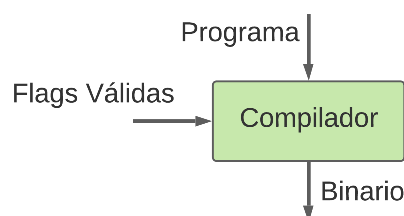


Figura 4.8: Flujo módulo compilación.

Con este cambio también habría que cambiar las opciones de entrada por unas válidas y adaptar, en el caso que fuera necesario, los tipos de *flags* definidos.

4.2.2. Ejecución de pruebas

Es en esta sección donde se unen los *scripts* anteriormente mencionados a la estructura del AG. Sería en esta sección donde se podría añadir diferentes objetivos pero, en este caso, al tratarse de un cambio mayor, si que sería necesario adaptar más partes del algoritmo.

4.2.3. Selección

Para poder seleccionar los mejores individuos podemos hacer uso de un MCDM, que nos permite buscar cual de estos se adapta a los criterios definidos.

En este caso se ha decidido usar WSM, uno de los MCDM más sencillos que podemos encontrar. El único requisito para usar este MCDM es normalizar los datos anteriores, ya que datos como el consumo de RAM puede ser bastante más grande escalarmente que, por ejemplo, el consumo de CPU. Por lo tanto sería necesario normalizarlos antes de poder hacer uso de WSM.

También hay que tener en cuenta que, en este caso, estamos minimizando, ya que se seleccionan los menores, siguiendo la regla *less is better*. El único objetivo que el mejor resultado es el más alto, es la robustez. Para eso, después de normalizar entre 0 y 1, podemos invertirlo fácilmente (1 - resultado normalizado).

Como en los anteriores casos, también se podría sustituir fácilmente los algoritmos utilizados para normalizar o para realizar la selección. Para cambiarlos habría que seguir

el esquema presentado en la Figura 4.9 donde podemos ver que cada una de las piezas es sustituible, simplemente habría que asegurarse el funcionamiento en cuanto a salida y entrada de datos se refiere.

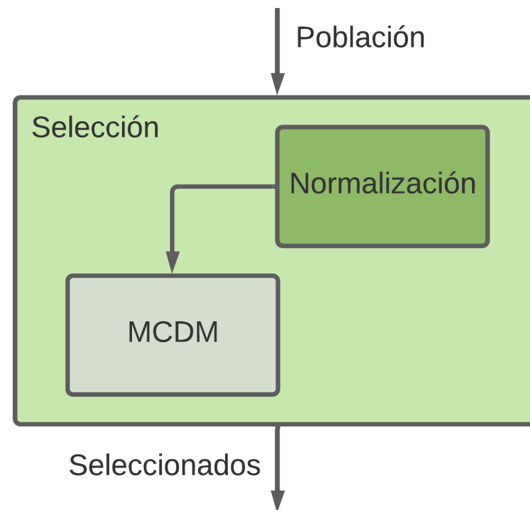


Figura 4.9: Flujo módulo selección.

4.2.4. Cruce

El cruce o *crossover* se encarga de, a partir de los individuos seleccionados, generar una población o una parte de ella que represente los individuos seleccionados. Existen muchos métodos a seguir a la hora de implementar el cruce. En este caso se ha seguido uno que se especificará más adelante en el capítulo de implementación pero se podría sustituir por cualquier otro mientras se siga el esquema presentado en la Figura 4.10.

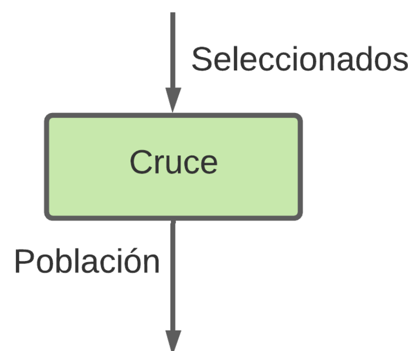


Figura 4.10: Flujo módulo cruce.

4.2.5. Mutación

La mutación se encarga de, a partir de un conjunto de individuos, generar ciertos cambios en su «cromosoma».

En este caso se ha seguido un esquema basado en la aleatorización, que más tarde se explicará en el capítulo de implementación, pero simplemente hay que seguir el flujo indicado en la Figura 4.11, donde vemos que de forma sencilla se puede cambiar el algoritmo de mutación por cualquier otro.

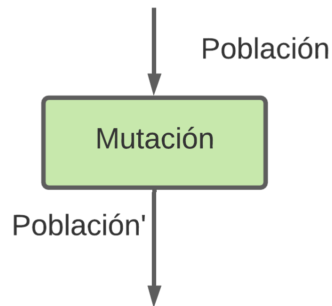


Figura 4.11: Flujo módulo mutación.

4.2.6. Solución

En la parte final del AG se devuelve el mejor individuo de la última generación, en este caso el individuo con menor WSM. Este individuo se representa como la línea de compilación de GCC necesaria para generarlo.

CAPÍTULO 5
Implementación

En este capítulo se van a repasar todos los pasos seguidos para crear la solución presentada. Por un lado encontraremos la selección de herramientas utilizadas para el desarrollo, el detalle del desarrollo de la solución y, para finalizar, las diferentes herramientas desarrolladas junto a la solución.

5.1 Herramientas utilizadas

A continuación, se presenta una lista de la herramientas de desarrollo utilizadas, donde se introduce el uso dado para cada una de ellas.

- **Ubuntu 20.04 LTS**: una de las distribuciones de GNU/Linux más conocidas. Elegimos esta versión por ser la última versión con soporte extendido y ser el sistema utilizado a la hora de elegir las herramientas de medición utilizadas.

- **Git**: para el control de versiones usamos Git. Nos permite tener un control claro e ir añadiendo nuevas características al trabajo ya realizado y en funcionamiento.

- **Github**: como repositorio *online* que hace uso de Git. Con esta plataforma podemos tener una copia en la nube del trabajo realizado y podemos compartir de forma sencilla la solución creada.

- **Python 3**[1]: como lenguaje de programación, usado por su sencillez y por la cantidad de herramientas y ejemplos de algoritmos genéticos disponibles. Usado para la implementación del AG

- **C**[2]: usado para la construcción de benchmarks. Es el lenguaje elegido a optimizar y para poder probar la solución creada hacemos uso de ciertos programas como benchmarks.

- **Bash *Scripting***: usado para construir pequeños *scripts* que hacen uso de las herramientas de medición para facilitar el uso por el AG construido en Python. También se usará para la implementación de un pequeño *script* de instalación de las dependencias del programa, haciendo así mas fácil su uso.

[1]Documentación disponible en: https://docs.python.org/3/
[2]Documentación disponible en: https://devdocs.io/c/

5.2 *Scripts* de medición

Las herramientas de medición no devuelven los resultados en el mismo formato ni tampoco devuelven únicamente los resultados que queremos medir. En este apartado se van a detallar las partes más importantes de los *scripts* que usan estas herramientas y que usando otras herramientas de Bash devuelven el valor deseado.

Antes de entrar a detallar el funcionamiento de estos *scripts*, hay que saber que Bash es la *shell* predeterminada de los sistemas GNU/Linux. Existen otras *shell* como ZSH o Fish, pero en este caso haremos uso de Bash al ser la que viene instalada por defecto y ser una de las pioneras en este campo. Por otro lado, un *script* es una secuencia de ordenes que funcionan como lenguaje interpretado. En nuestro caso un Bash *script* es una secuencia de ordenes Bash escritas en un archivo, que a la hora de ser ejecutado es interpretado por Bash.

A continuación, se detalla cada uno de los *scripts* de cada uno de los objetivos a medir con sus particularidades. Todos los *scripts* están disponibles en el repositorio de Github, accesible desde [35].

Tamaño del binario

En este objetivo se decidió usar **du** como herramienta de medición. Esta herramienta cuenta con múltiples opciones, entre ellas la opción «-b», que indica que se quiere calcular el tamaño en *bytes*. Su ejecución devuelve una salida con el resultado en *bytes* y el nombre del archivo o directorio medido. Dos parámetros de salida, por lo tanto simplemente quedándonos con el primer parámetro tendríamos el resultado deseado. Y como vemos en el fragmento de código 5.1, lo conseguiríamos haciendo uso de **awk**[3].

```
1 du -b $1 | awk '{print $1}'
```

Código 5.1: *Script* medición tamaño de un binario.

Tiempo de ejecución

Como se detallaba en la sección 4.1, en este caso se va a hacer uso de la orden ***time***, más concretamente de la implementación de Linux. *Time* cuenta con múltiples opciones, entre ellas la elección de formato que usaremos para poder quedarnos únicamente con la duración en segundos.

En este caso, como en los siguientes, el programa se debe ejecutar para poder obtener un resultado, también hay que descartar la salida del programa a medir. Para esto debemos saber que *time* hace uso de la salida estándar de errores, o **stderr**, para mostrar su resultado.

Con esto podemos simplemente mandar la salida estándar, o **stdout**, a «/dev/null», un dispositivo nulo, que en la práctica hace que descartemos esta salida, y redirigir la stderr por stdout sin ser descartada. En la figura 5.1 podemos ver como se guarda la sálida de *time* en la variable «mínimo» haciendo en este caso uso del binario guardado en «BINARY» con los argumentos guardados en «ARGS».

[3]Manual disponible en: https://man7.org/linux/man-pages/man1/awk.1p.html

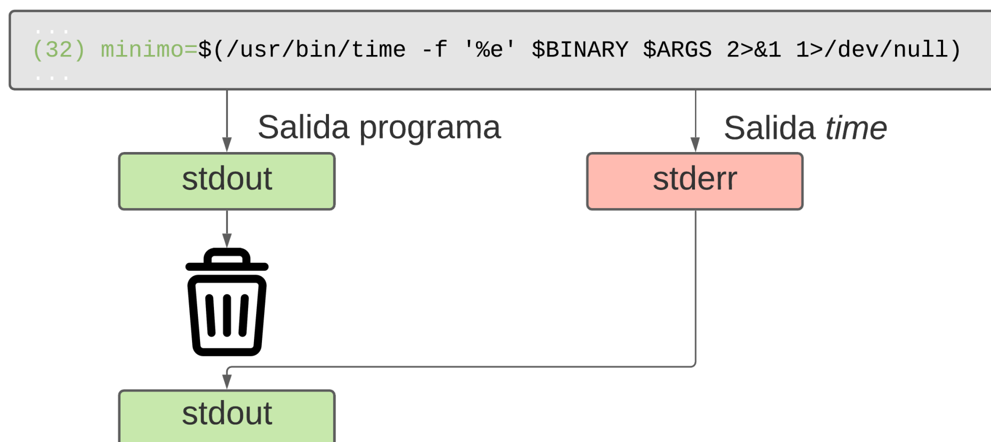


Figura 5.1: Representación fragmento de código para medición de tiempo de ejecución.

Para este *script* se ha añadido la funcionalidad de poder pasar el binario y sus argumentos como parámetros. También se ha añadido la posibilidad de ejecutar la prueba un número determinado de veces para quedarnos con la ejecución de tiempo mínimo.

Carga de la CPU

En este caso se ha decidido usar dos herramientas para implementar la solución. La primera de ellas es **pidstat**, la cual muestra el uso de la CPU dado un intervalo de tiempo y al finalizar muestra la carga media. Es esta última parte la que nos interesa, para ello simplemente ejecutamos la aplicación y seleccionamos el campo correcto.

En este caso usamos un redirector de salida a fichero, para más tarde poder imprimirlo. En el fragmento de código 5.2 podemos ver el funcionamiento de este, que órdenes se usan para seleccionar el campo deseado y cómo se transforma el campo para devolver un valor entre 0 y 1 que represente el porcentaje.

```
1 ...
2 pidstat $N -u -e $to_execute > $archivo;
3 ...
4 avg=$(cat $archivo | grep Average: | awk '{print $8}' | tail -1 | tr ',' '.');
5 ...
6 echo "scale=4; $avg/100" | bc | sed 's/^\./0./';
7 ...
```

Código 5.2: Fragmento *script* medición carga de la CPU usando Pidstat.

En este fragmento la variable «to_execute» representa el binario con los argumentos pasados, la variable «avg» representa la salida de la carga media. Es esta última variable la que se utiliza para saber si la herramienta ha fallado y usar la alternativa que hace uso de ***time***. Con esto sabemos que si «avg» no contiene nada podemos pasar a usar *time*, donde se hace uso del fragmento de código 5.3.

```
1 ...
2 avg=$(/usr/bin/time -f '%P' $to_execute 2>&1 1>/dev/null)
3 ...
4 avg2=$( echo $avg | sed 's/%//' );
5 ...
6 echo "scale=4; $avg2/100" | bc | sed 's/^\./0./';
7 ...
```

Código 5.3: Fragmento *script* medición carga de la CPU usando *time*.

En este *script* también se ha implementado un mecanismo para pasar los diferentes parámetros de configuración de las aplicaciones. Este código se encuentra disponible junto a los demás *scripts* en el repositorio de github [35].

Robustez

Para este objetivo se ha decido usar Zofi, y como se vio anteriormente, Zofi devuelve una salida muy completa pero solo nos interesa quedarnos con los valores finales con los resultados estadísticos de las inyecciones al binario.

Con esto podemos pasar a ver el fragmento de código 5.4, que muestra como se ejecuta Zofi pasando la salida a un archivo, y como se trata el archivo para seleccionar la parte deseada.

```
...
zofi -bin $BINARY -test-runs $N -args $ARGS &>/$archivo;
...
for element in $(cat $archivo | tail -n 4 | awk '{print $4}' | cut -d \. -f 1)
do
    echo "scale=2; $element/100" | bc | sed 's/^\./0./';
done
...
```

Código 5.4: Fragmento *script* medición Robustez.

Consumo de Memoria

Para medir el consumo de RAM se decidió usar Memory-profiler, una herramienta que devuelve un archivo con la marca de tiempo y la RAM en uso en ese momento. En este caso el *script* debe leer el archivo y quedarse con la mayor cantidad de memoria usada, ya que en este caso se quiere medir el máximo de memoria usada.

Con este objetivo se ha creado una función en Bash que recorre el fichero e imprime por pantalla el resultado. Esta función forma parte de un script que reúne todo el proceso de ejecución de la herramienta y el cálculo necesario. En el fragmento de código 5.5 podemos ver parte de este script donde se ejecuta la herramienta, se procesa el fichero previamente y se declara y llama a la función antes mencionada.

```
calculo () {
...
max=0;
for elemento in $(echo $salidaInVar)
do
...
if (( $(echo "$elemento > $max" |bc -l) )); then
  max=$elemento;
fi
done
...
echo $max;
}
...
mprof run --output $output $to_execute &>/dev/null
...
salidaInVar=$( tail -n +2 $output | awk '{print $2}');
calculo;
```

Código 5.5: Fragmento *script* medición RAM.

En este *script* también se ha realizado la entrada por parámetros, y el cálculo de la media y la mínima memoria usada.

5.3 Algoritmo genético

En esta sección se va a repasar todas las partes del AG, cómo se ha implementado cada una de ellas y las dificultades encontradas. Cada una de estas partes también las hemos llamado «módulos», al poder sustituirse y funcionar modularmente.

5.3.1. Preparación

En la primera parte del algoritmo se han inicializado todas las variables que configuran la solución, y tras esto se han creado los mecanismos para poder ejecutar el algoritmo.

Configuración AG

Por un lado, para la configuración se ha usado uno de los módulos de Python 3 creados para ello. En este caso se ha hecho uso del módulo *configparser* para la mayoría de configuraciones. En algunas excepciones se ha usado la configuración por parámetros, haciendo uso en este caso del módulo *argparse* para facilitar el trabajo.

El uso de *configparser* es bastante sencillo, simplemente hay que crear un archivo con extensión «.ini» siguiendo el patrón de la figura 5.2, cargar el archivo en el código y usarlo siguiendo el patrón mostrado en el código 5.6.

```
1    [Grupo1]
2    Var1 = Dato1
3    Var2 = Dato2
4
5    [Grupo2]
6    Var1 = Dato1
7    Var2 = Dato2
```

Figura 5.2: Patrón archivo *configparser*.

```
1 import configparser
2 parser = configparser.ConfigParser()
3 parser.read("NombreArchivo.ini")
4 Var1 = float(parser["Grupo1"]["Var1"])
5 Var2 = float(parser["Grupo2"]["Var2"])
```

Código 5.6: Patrón uso *configparser*.

Este tipo de configuración se ha decidido usar para las siguientes partes:

- Pesos de los objetivos.
- Configuración del algoritmo genético (tamaño, mutación, individuos a seleccionar, etc.).
- *Path* archivo con las opciones a usar.
- *Flags* necesarias para compilar el programa objetivo y ubicación de los archivos de los que depende
- Configuración de los límites del algoritmo.
- Configuración de las pruebas a realizar.

Para otras configuraciones como el programa objetivo, modo debug, imprimir resultados por pantalla o pasar los argumentos del programa objetivo con los que realizar las pruebas se ha decidido usar *argparse*. Este módulo permite hacer usos de parámetros para el programa fácilmente. Se puede usar siguiendo el patrón indicado en el fragmento de código 5.7.

```
argparser = argparse.ArgumentParser()
argparser.add_argument("-p", "--programa", dest="program", help="Path absoluto
    del programa a optimizar")
...
args = argparser.parse_args()
programa = args.program
```

Código 5.7: Patrón uso *argparse*.

Preparación opciones

Cada usuario puede elegir qué opciones quiere usar para ejecutar la optimización. Esto se ha decidido implementar usando un esquema concreto y haciendo uso para ello de un archivo **JSON**. Este tipo de archivos permite fácilmente estructurar la información de las diferentes opciones y puede ser leído con el módulo *json* de Python.

Se ha decidido seguir una estructura como la de la sección de código 5.8. Donde se definen claramente los tipos de opciones y sus características.

```
{
    "binarias": [
        {
            "flag": "-ftree-ter"
        }
    ],
    "rango": [
        {
            "flag": "-falign-loops=",
            "min": "1",
            "max": "65536"
        }
    ],
    "Intervalo": [
        {
            "flag": "-fira-region=",
            "intervalo": "all,mixed"
        }
    ]
}
```

Código 5.8: Estructura opciones en JSON.

Una vez definida la estructura del archivo json, hay que incorporar la lectura al código, algo simple al usar el módulo json. Simplemente habría que cargar el archivo y se podría recorrer por los diferentes campos.

Estructura de datos

A la hora de guardar los datos se ha seguido una estructura como la de la figura 5.3, donde al leer las flags se crean objetos que contienen toda la información y se almacenan esos objetos en una única lista.

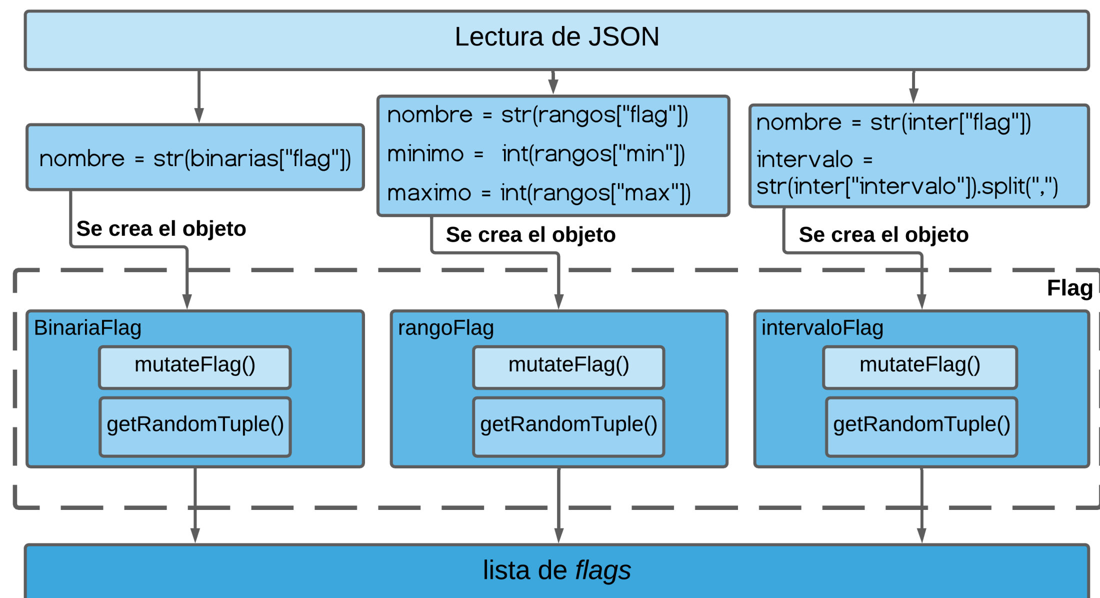


Figura 5.3: Representación Flujo creación estructura *flags*.

Se almacenan todos los objetos en una única lista, ya que para el resto del programa no existen los tipos. Esto es gracias al polimorfismo y herencia. Donde una clase abstracta nos sirve como base para las demás clases, abstrayendo así los tipos de flags para las demás partes del algoritmo y forzando a la implementación de los métodos abstractos.

5.3.2. Creación de la población inicial

Una vez inicializado el AG, hay que empezar con la optimización. Para ello se crea la primera generación o población inicial.

En este caso se crea una lista y se inicializa la variable «población», que representará la generación actual y está compuesta por objetos «cromosoma». Estos objetos contienen la información de un individuo, con los futuros resultados de las pruebas antes y después de normalizar, con el resultado tras calcular su peso usando WSM y con la línea de compilación usada para generarlo. El objeto «cromosoma» también indica si un individuo se ha ejecutado ya, para no tener que realizar otra vez las pruebas en futuras generaciones. También contiene las opciones que lo forman, un vector «tuplas» con la información del genoma codificada en una lista de pares formados por un booleano y la flag correspondiente.

Se ha decidido hacer uso de dos tamaños de población diferentes. Un tamaño para la población general y otra para el resto de generaciones. Esto es para poder tener una base de individuos más amplia en la primera generación, ya que por una parte se está aumentando el espacio de búsqueda y por otra estamos protegiéndonos para ejecuciones con opciones de GCC incompatibles, donde no todos los individuos son exitosos y se pueden crear.

5.3.3. Compilación

Esta parte es el primer módulo a la hora de iterar entre generaciones. Una vez creada la generación inicial o una generación cualquiera hay que convertirla en realidad: hay que compilarla y crear todas la estructura de ficheros necesaria para su uso.

Para compilar las diferentes soluciones se ha decidido usar GCC. Con esto solo falta generar la línea de compilación correcta para cada individuo, compilar y guardar el binario en el directorio creado.

Antes de nada, hay que saber que en la fase de preparación ya se ha creado la carpeta destino para la ejecución actual de la que se ha definido su ubicación en los archivos de configuración antes mencionados. Por lo tanto, ya contamos con la información del directorio base de la ejecución actual. Con esto, podemos ver en la figura 5.4, el funcionamiento detallado de este módulo.

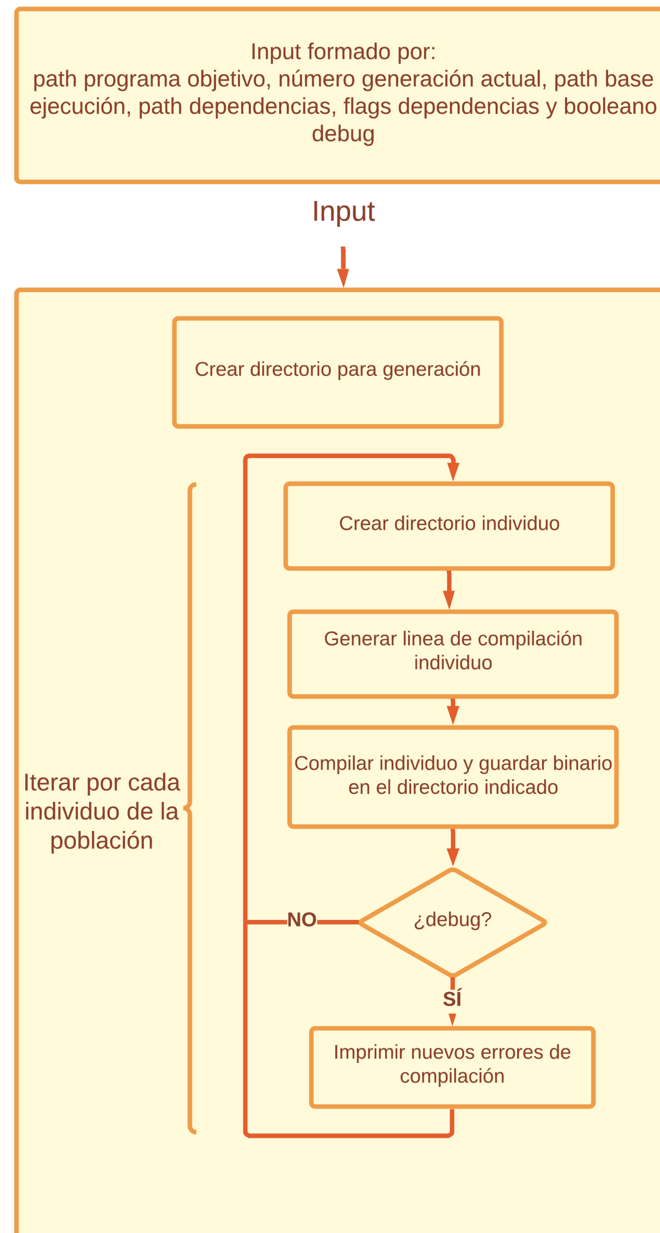


Figura 5.4: Flujo de compilación de individuos de la población.

En esta figura podemos ver como se van primero creando las ubicaciones de los diferentes directorios, se crean haciendo uso de el módulo os de Python, más concretamente con la función «os.system()» haciendo uso de la orden *mkdir* o con «os.mkdir()» indistintamente.

También podemos ver que se calcula la línea de compilación de cada individuo y se ejecuta, esta vez haciendo uso de una pequeña función que podemos ver en la sección de código 5.9, donde vemos el uso del módulo subprocess para ejecutar órdenes Linux y obtener los errores.

Este módulo nos permite guardar la salida de la ejecución del comando. En este caso devuelve la salida de GCC y su stderr. En el caso que este activada la opción de depuración esta sálida se mostrará por pantalla. Esta función se irá usando sucesivamente en los casos que sea necesario obtener el stdout o el stderr de la ejecución de una orden.

```
def executionWithOutput(command):
    result = subprocess.run(
        command,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        universal_newlines=True,
        shell=True,
    )
    return (result.stdout, result.stderr)
```

Código 5.9: Función executeWithOutput()

Una vez hecho esto con todos los individuos de la población ya solo faltaría devolver la ubicación del directorio de la generación compilada para poder usarla en siguientes fases.

5.3.4. Ejecución de pruebas

Una vez creados los individuos, empieza la parte de la *function fitness* donde se les otorga una puntuación a cada individuo. En este modulo se van a ejecutar las pruebas a los diferentes individuos, haciendo uso en este caso de los *scripts* anteriormente presentados.

Para agilizar las pruebas, al ser la parte más lenta de la ejecución, se ha decidido usar una ejecución multihilo, donde se va a crear una *pool* de hilos y se van a ir ejecutando individuos según vayan liberándose estos. Se usa una *pool* para no saturar la ejecución y se utiliza un valor configurado previamente como máximo de hilos concurrentes. Podemos ver la implementación realizada en el código 5.10. Para implementar la ejecución multihilo se ha usado el módulo *threading* de Python.

```
def test(poblacion, maximoNumeroHilos, directorioGeneracionActual):
    threads = []
    try:
        for i in range(len(poblacion)):

            if not poblacion[i].pruebasHechas:
                threads.append(
                    threading.Thread(
                        target=pruebas,
                        args=(poblacion[i], i, directorioGeneracionActual),
                    )
                )
                poblacion[i].pruebasHechas = True

        [t.start() for t in threads]
    except Exception as e:
        print("Excepcion en hilos")
        return
    finally:
        [t.join() for t in threads]
```

Código 5.10: Función test(), lanzamiento pruebas de individuos usando *multithreading*.

Tras lanzar la ejecución de un individuo se llama a la función prueba, donde se hace uso de semáforos para medir los hilos concurrentes y se llama a la ejecución de pruebas si se ha seleccionado en la configuración. Para ello se usa el peso de los objetivos, si el peso es 0 no se ejecutará la prueba.

Con esto podemos ver el fragmento de código 5.11, donde se puede ver la ejecución de una prueba estándar y la ejecución de robustez al ser la única donde se seleccionada el porcentaje de enmascarados y se invierte para después poder minimizar.

```
def pruebas(cromosoma, i, pathActual):
    try:
        pool.acquire()
        directorioActual = pathActual + "/Cromosoma" + str(i) + "/"
        executable = directorioActual + "Cromosoma" + str(i)
        if os.path.isfile(executable) and os.access(executable, os.X_OK):

            if Ram:
                cantRam = ram(executable, Argumentos, directorioActual).split("
                    \n")
                cantRam = cantRam[0] or -1.0
                cromosoma.resultRam = float(cantRam)
            ...
            if Rob:
                cantRob = robustness(executable, Argumentos,
                    ExecutionRobustness, directorioActual).split("\n")
                if len(cantRob) > 1:
                    cantRob = 1.0 - float(cantRob[0])
                else:
                    cantRob = -1.0
                cromosoma.resultRob = float(cantRob)
        pool.release()
    except Exception as e:
        print("Excepcion en hilos")
        return
```

Código 5.11: Función pruebas(), ejecución pruebas.

Como podemos ver no se le pasan todos los valores por parámetros, ya que han sido importadas las variables globales y son visibles para la función.

Cada una de las llamadas a las pruebas hace uso a su vez de los *scripts* de pruebas y el código 5.9 presentado anteriormente para su ejecución.

Por último, durante la compilación existe la posibilidad de que esta falle y no genere un binario. Para esto se hace la comprobación de ejecución del binario.

Si no fuera posible ejecutar las pruebas no se actualizaría el valor de los resultados quedándose el de por defecto que es -1, valor que ninguna prueba puede obtener y asignado así para poder descartarse fácilmente en la normalización. También se asigna -1 a las pruebas que fallan.

5.3.5. Normalización y WSM

Una vez realizadas las pruebas de los individuos, pasaríamos a normalizar y calcular el peso de cada uno de los individuos haciendo uso de WSM.

Antes de nada, hay que saber que se ha decidido normalizar los valores usando escalamiento lineal para dejarlos entre 0 y 1 por tanto no a todos los objetivos es necesario normalizarlos. Tanto la robustez como la carga de la CPU ya se encuentran entre 0 y 1. También se ha decidido usar *Clipping* para en este caso descartar los valores de los individuos que no han generado binario o han fallado las pruebas.

Por lo tanto, todos los objetivos tienen que descartar los valores negativos, y el uso de memoria, tiempo de ejecución y tamaño del binario también tiene que hacer uso de escalamiento lineal.

Como ejemplo de implementación de normalización para los objetivos que hacen uso de ambos métodos, normalización con escalamiento lineal, y filtro de datos con *Clipping*, podemos encontrar una implementación muy similar a la del código 5.12, donde primero se busca el máximo y mínimo sin tener en cuenta valores negativos, después se comprueba que son diferentes para no acabar en una división por 0, y por último, se realiza el escalado lineal y la sustitución de los valores negativos por 1.0, que representa el peor caso.

```
def normTiempo(poblacion):
    global minGlobalTiempo, maxGlobalTiempo
    for cromosoma in poblacion:

        if cromosoma.resultTiempo > 0.0
            and cromosoma.resultTiempo > maxGlobalTiempo:

            maxGlobalTiempo = cromosoma.resultTiempo

        if cromosoma.resultTiempo > 0.0
            and cromosoma.resultTiempo < minGlobalTiempo:

            minGlobalTiempo = cromosoma.resultTiempo

    if minGlobalTiempo == maxGlobalTiempo
        or maxGlobalTiempo < minGlobalTiempo:
        for cromosoma in poblacion:
            if (cromosoma.resultTiempo == minGlobalTiempo
                or cromosoma.resultTiempo == maxGlobalTiempo):
                cromosoma.afterNormTiempo = 0.0
            else:
                cromosoma.afterNormTiempo = 1.0
    else:
        for cromosoma in poblacion:
            if cromosoma.resultTiempo < 0.0:
                cromosoma.afterNormTiempo = 1.0
            else:
                cromosoma.afterNormTiempo =
                (cromosoma.resultTiempo - minGlobalTiempo)
                    / (maxGlobalTiempo - minGlobalTiempo)
```

Código 5.12: Función normTiempo() como ejemplo de normalización completa.

Como podemos ver se ha decidido usar mínimos y máximos globales, aunque no se vuelven a normalizar las generaciones anteriores cuando se actualiza el mínimo o máximo. Esto se hace para poder ver más clara la evolución a la hora de analizar los datos.

Como ejemplo de implementación para los objetivos que solo necesitan del *Clipping* podemos encontrar el código 5.13, donde simplemente se sustituye el -1 por el 1.0 y se sustituye el 1.0 por el 0.999 para no igualarlo a los que no funcionan.

```
def normCpu(poblacion):
    for cromosoma in poblacion:
        if cromosoma.resultCPU < 0.0:
            cromosoma.afterNormCpu = 1.0
        elif cromosoma.resultCPU == 1.0:
            cromosoma.afterNormCpu = 0.999
        else:
            cromosoma.afterNormCpu = cromosoma.resultCPU
```

Código 5.13: Función normCpu() como ejemplo de clipping.

Con esto solo quedaría ver como implementar WSM, algo bastante sencillo, ya que bastaría con ir recorriendo la población, y sumando el resultado de multiplicar el peso por la puntuación después de normalizar cada objetivo. En este caso, la implementación resulta como en el código 5.14

```
def wsm(poblacion, Ram, Tiempo, Peso, Rob, CPU):
    for cromosoma in poblacion:
        cromosoma.WSM = (
            cromosoma.afterNormRam * Ram
            + cromosoma.afterNormTiempo * Tiempo
            + cromosoma.afterNormPeso * Peso
            + cromosoma.afterNormRob * Rob
            + cromosoma.afterNormCpu * CPU
        )
```

Código 5.14: Función wsm()

5.3.6. Selección

Una vez podemos comparar un individuo con otro teniendo en cuenta todos los objetivos podemos pasar a seleccionar los menores, algo bastante simple. Bastará con ordenar la lista de la población respecto al WSM y quedarnos con los N primeros componentes. Lo podemos ver implementado en el código 5.15.

```
def selection(poblacion, N):
    poblacionAux = sorted(poblacion, key=lambda cromosoma: cromosoma.WSM)[:N]
    return poblacionAux
```

Código 5.15: Función selection()

Por ultimo, antes de pasar a la siguiente fase añadimos la lista de la población a una lista que guardará el histórico de generaciones.

5.3.7. Límites de ejecución

En la configuración existe la posibilidad de elegir entre tres límites implementados y configurarlos de diferente manera. Existen límites de generación, de tiempo y de convergencia.

El límite de generación consiste en acabar la ejecución tras un cierto número de generaciones. El límite de tiempo consiste en terminar la ejecución una vez se sobrepase un tiempo establecido. Y por último, el límite de convergencia consiste en la comprobación a partir de cierta generación del cambio de mejor individuo entre generaciones.

Este último límite se ha implementado simplemente comprobando el cambio de genoma entre los dos mejores individuos de la generación actual y la inmediatamente anterior.

Con la comprobación del límite seleccionado se devuelve un booleano, que en el caso de ser cierto hace que se salte a la fase de salida, si no, se sigue con la ejecución de la siguiente generación.

También existe la posibilidad de terminar la ejecución con [Control+C], ya que se ha implementado un manejador que hace uso de todas las operaciones de salida necesarias.

5.3.8. Nueva generación

En el caso de continuar con la ejecución, el siguiente paso sería formar la nueva generación. Para ello hay que elegir una lógica a implementar en el cruce y mutación.

En este caso hemos implementado una nueva generación formada por 3 grupos. Los elementos seleccionados sin cambios, los elementos resultantes del cruce y un porcentaje a definir en la configuración de elementos aleatorios nuevos.

Es en este momento donde la lista definida anteriormente como «población» se vacía y se vuelve a ir creando. En primer lugar se copian los elementos seleccionados a la nueva lista, después se calcula el número de los elementos aleatorios, que surgen del resultado de multiplicar el tamaño de la población por el porcentaje de aleatorios y truncar la multiplicación.

Una vez definido el número de aleatorios se usa la lógica creada para generar la generación inicial y se usa para crear una población aleatoria del tamaño calculado y añadir los elementos a la nueva lista de población.

Con esto solo queda definir los elementos resultantes del cruce y la mutación. Para la mutación se ha definido una variable llamada «radiación», la cual se va a utilizar para controlar la tasa de mutación. Con esto podemos pasar a describir el cruce.

El cruce escoge aleatoriamente un antecedente del individuo a crear, y de este antecedente obtiene una *flag* y su correspondiente tupla del genoma. Este proceso se repite como tantas opciones se hayan configurado, o lo que es lo mismo, tanto como el tamaño del individuo a crear. En el fragmento de código 5.16 podemos ver los pasos que se siguen para crear un individuo. Este proceso se repetirá tantas veces como individuos cruzados necesitemos.

```
1  # Cruza cromosomas aleatoriamente, muta el cromosoma resultante tantas veces
       como se indique en radiacion
2  def crossover(listaAntecedentes, radiacion):
3      tamanyo = len(listaAntecedentes[0].flags)  # Cantidad de flags
4      listaFlagsNuevo = []
5      listaTuplasNuevo = []
6      # Crear cromosoma
7      for i in range(tamanyo):
8          # Seleccion aleatoria del cromosoma padre
9          aleatorio = random.randint(0, len(listaAntecedentes) - 1)
10         listaFlagsNuevo.append(listaAntecedentes[aleatorio].flags[i])
11         listaTuplasNuevo.append(listaAntecedentes[aleatorio].tuplas[i])
12     # Crear objeto cromosoma
13     para_mutar = Cromosoma(listaFlagsNuevo, listaTuplasNuevo)
14     mutarCromosoma(para_mutar, radiacion)  # Mutar cromosoma
15     return para_mutar
```

Código 5.16: Función crossover(). Proceso de creación de un individuo cruzado.

Como podemos ver antes de devolver el individuo se muta. Este proceso consiste en repetir tantas veces como radiación se indique el proceso de elegir un genoma aleatorio y usar el método mutar del objeto previamente definido. En el código 5.17 podemos ver la implementación y como se repite el mismo proceso tantas veces como radiación se indique.

```
1  # Muta un cromosoma aleatoriamente la cantidad de veces que se indique en el
       parametro radiacion
2  def mutarCromosoma(cromosoma, radiacion):
3      for i in range(radiacion):
4          # Posicion aleatoria a mutar
5          a_mutar = random.randint(0, len(cromosoma.flags) - 1)
6          flagSeleccionada = cromosoma.flags[a_mutar]
7          cromosoma.flags[a_mutar].mutateFlag()  # mutar flag
8          # actualizar flag y rerandomizar la tupla
9          cromosoma.tuplas[a_mutar] = flagSeleccionada.getRandomTuple()
```

Código 5.17: Función mutarComosoma(). Proceso de mutación.

La radiación representa el número de veces que un individuo es mutado, pudiendo calcular la tasa de mutación de cada cromosoma como Radiación/Número de Opciones de compilación. Este parámetro nos permite así variar la cantidad de genomas que mutan y aumentar o disminuir la aleatoriedad.

Una vez creados los individuos aleatorios, los cruzados y los seleccionados ya estaría la nueva población a devolver y se volvería al proceso de compilación.

5.3.9. Salida

Cuando el límite elegido se cumple, pasamos a la fase final donde, gracias al histórico, podemos obtener todos los resultados e imprimirlos en diferentes archivos ubicados en el directorio de la ejecución configurado. Tras esto, se muestra la línea de compilación del mejor individuo de la última generación y se solicita al usuario si quiere hacer una comparación con las opciones de optimización por defecto. A continuación, se describen estas dos acciones en detalle.

Tras esto se imprime la información de la figura 5.5, donde se solicita al usuario si quiere comparar el resultado. Esto se hace compilando y usando todos los mecanismos ya definidos anteriormente, al finalizar muestra una comparación visual.

```
                    Ejecución finalizada

Ejecución finalizada por convergencia ejecución.

Se han generado automaticamente diferentes archivos de
estadísticas de la ejecución, los puedes encontrar en la
carpeta  Ejecucion05-08-2020_17:00:05 en el path que has
configurado.

Antes de finalizar:

- ¿Quieres comparar el resultado con los flags de opti-
  mización?[y/n]: y


---------------------------------------------------------


Esta es la línea de compilación seleccionada:

gcc <binario>  -o <flags> --> linea de compilación ejempl

[Comparación]:
[WSM]:
De mejor a peor(less is better):

- 01               <->
- OptimizaciónAG   <->
- Os               <->
- 03               <-------------------------->
- Sin optimización <------------------------------>
- 00               <------------------------------>
- 02               <-------------------------------->
- Ofast            <--------------------------------------->

[!]Saliendo...
```

Figura 5.5: Representación salida final ejecución.

La primera parte consiste en obtener y escribir en archivos los datos de evolución de todas las generaciones y obtener las opciones más usadas en general y en los mejores individuos. Esto se hace para poder analizar la evolución del algoritmo, y en este caso se ha elegido obtener estos datos del histórico. Se van recorriendo las generaciones en el histórico, se recorren a su vez los individuos y se obtiene sus resultados en las diferentes pruebas realizadas, su resultado de WSM, y por último las opciones utilizadas.

5.3.10. Conclusión

Durante las diferentes secciones se ha visto la modularidad de la aplicación, al poder cambiar fácilmente partes de la solución. Se ha visto la abstracción de tipos usando objetos. También se han aplicado conceptos de *threading* a la hora de ejecutar las diferentes pruebas.

Durante la realización del código no se han usado módulos externos a los oficiales de Python. Esto permite compilar el código usando herramientas como Nuitka3 [36], herramienta que crea un binario independiente a partir de código creado para intérpretes CPython.

En conclusión, se ha realizado una solución que sirve como base para otras soluciones que mejoren aspectos en los que esta no ha entrado, como escoger un MCDM respecto a resultados experimentales o añadir otros objetivos a medir o simplemente aplicar la optimización a otro lenguaje compilado.

5.4 Herramientas de análisis y automatización de instalación

Durante la creación de la solución se han presentado problemas para los cuales se han ido creando soluciones independientes. Podemos encontrar los siguientes programas auxiliares creados.

- **Scripts de instalación dependencias**: Se ha creado tanto un *script* para instalar las dependencias de la solución, mayoritariamente las aplicaciones de medición, como un script para la instalación de dependencias de todo el código usado, tanto auxiliar como la solución. Se encuentran disponibles junto a todo el código en Github [35].

- **Creación de fichero JSON con las opciones del compilador**: para facilitar la creación de JSON con muchas opciones se ha implementado una solución que transforma un archivo *Comma Separated Values* (CSV) con la estructura indicada en el programa a JSON. Se usa un CSV por la facilidad de crear una tabla con cualquier editor y guardarla en este formato. También se ha creado pequeña solución para crear el archivo automáticamente a partir de la entrada del programa.

- **Programas de visualización**: los archivos que devuelve la solución principal pueden ser muy extensos para una hoja de cálculo. Para ello se han creado dos pequeños programas que permiten la creación de gráficas a partir de los datos de los archivos en este caso haciendo uso de Matplotlib [37], un módulo de Python, externo en este caso.

CAPÍTULO 6
Experimentación

En este capitulo se analizará los resultados obtenidos de la experimentación de la solución de optimización creada. En primer lugar se presentará el sistema utilizado para realizar las pruebas, se presentará los benchmarks usados para estos experimentos y, por último, se presentarán los resultados obtenidos junto a un análisis.

6.1 *Setup*

Para realizar los experimentos se han usado dos configuraciones. La primera de ellas ha sido un portátil con Ubuntu 20.04 LTS instalado como sistema principal, que cuenta con un procesador Intel[1] de octava generación, el Intel i78550U, con 4 núcleos y 8 hilos. Este portátil cuenta con 16GB de RAM DDR4.

Otra configuración, en la que se han realizado la mayoría de pruebas, ha sido una máquina virtual de Azure[2]. Azure es un servicio, en este caso de Microsoft, que permite entre otras cosas alojar máquinas virtuales. En este caso se ha hecho uso de las máquinas *Standard_D4s_v3*.

Estas máquinas virtuales cuentan con 4 núcleos virtuales y 16GB de memoria RAM cada una, es una de las que más núcleos virtuales tiene de las disponibles en la evaluación gratuita. En estas máquinas virtuales se encontraba instalado Ubuntu Server 20.04. LTS[3].

6.2 Benchmarks

Para poder realizar las diferentes pruebas se ha hecho uso de diferentes benchmarks, o programas de prueba, que nos han servido para comprobar el funcionamiento del AG. A continuación, encontramos una lista con una pequeña descripción de los benchmarks usados y dónde encontrarlos.

- **Sort**: un programa de ordenación de vectores. Se ha programado de forma robusta, por medio de *N-Version Programming*. Cuenta con tres algoritmos de ordenación de vectores. A este programa se le pasa por parámetros el tamaño de los vectores, los cuales se crean y desordenan aleatoriamente, tras esto son ordenados y se devuelve si ha sido ordenado correctamente. Se decide la ordenación por votación. Lo podemos encontrar en el repositorio de Github junto a la solución creada.

[1]Documentación disponible en: https://www.intel.com/content/www/us/en/homepage.html
[2]Disponible en: https://azure.microsoft.com/es-es/overview/what-is-azure/
[3]Disponible en: https://ubuntu.com/download/server

- **Treebench**: es uno de los benchmarks propuestos por ACOVEA [20]. Este benchmark realiza algunas operaciones con estructuras de datos, más concretamente con árboles. Se puede encontrar en el Github de ACOVEA en [38].

- **Matrix**: uno de los benchmarks usados para realizar los experimentos realiza multiplicaciones de matrices de gran tamaño y se encuentra disponible en [39].

Estos benchmarks tienen ciertos parámetros a la hora de utilizarlos, en cada experimento se indicará la configuración del benchmark indicada.

6.3 Experimentos y resultados

A continuación, se van a presentar cada uno de los experimentos realizados, con su correspondiente configuración y análisis.

6.3.1. Evolución del tiempo de ejecución

En este caso se ha optimizado únicamente el tiempo, para ello se ha elegido el benchmark **treebench**, que no requiere de argumentos.

Para los ajustes de la solución se ha puesto el peso del tiempo de ejecución a 1, se ha usado un tamaño de población inicial de 150 individuos y un tamaño de población general de 44 individuos. Se ha hecho uso del límite de generaciones a 20. Para la ejecución se ha hecho uso de una lista de 17 opciones de compilación, que podemos ver en la tabla 6.1, también se encuentran disponibles con el resto de opciones en la tabla C.1, disponible en el Apéndice B. Sabiendo que se ha usado como radiación un valor de 2, podemos saber que la tasa de mutación es de alrededor del 11 % (Radiación/Número de opciones). Se han seleccionando 8 individuos por generación y generado un 10 % de individuos aleatorios.

Opciones de Compilación		
Simples	Rango	Intervalo
-ftree-ter	-falign-loops=[1, 65536]	-fexcess-precision=[fast, standard]
-floop-unroll-and-jam	-falign-functions=[1, 65536]	-fira-region=[all, mixed, one]
-fipa-sra	-finline-limit=[1, 65536]	
-fipa-bit-cp		
-finline-functions		
-fdefer-pop		
-fcode-hoisting		
-ftree-builtin-call-dce		
-fipa-reference-addressable		
-fcompare-elim		
-ftree-dce		
-fbranch-count-reg		

Tabla 6.1: Tabla de opciones.

Esta ejecución ha sido realizada haciendo uso de Azure, y se han utilizado 4 hilos de ejecución. Con esto obtenemos el resultado de la figura 6.1

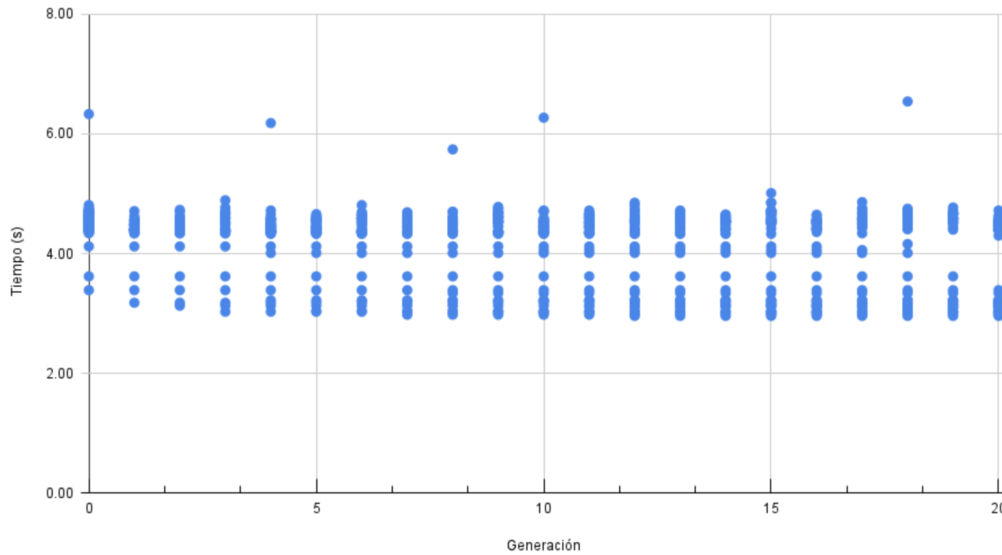


Figura 6.1: Ejecución experimento 1, Tiempo de ejecución vs Generación.

En esta imagen vemos la representación del resultado de todos los individuos para cada generación, podemos ver como poco a poco los mínimos descienden. La ejecución de este benchmark con estos mismos parámetros sin optimización ronda los 4,5 segundos. También podemos ver como cada ciertas generaciones la distribución se expande, y en otras se concentra más. Este es el comportamiento natural de un algoritmo genético, buscando los mejores resultados y expandiéndose cuando no puede reducirlos más. Por lo tanto podemos ver que ha habido una mejora, incluso trabajando con un pequeño grupo de opciones de compilación y en un número reducido de generaciones. La reducción del tiempo se puede ver mucho más claramente en la figura 6.2, donde vemos representados únicamente los mínimos.

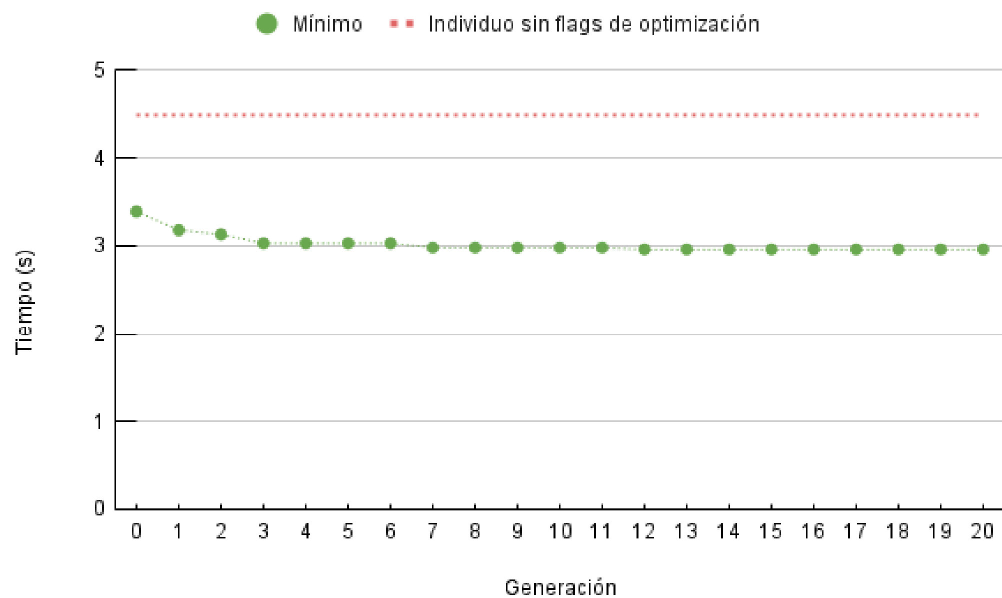


Figura 6.2: Ejecución experimento 1, mínimo tiempo de ejecución frente a generación.

Como vemos, en esta gráfica se puede apreciar la mejora, y la tendencia descendente de los tiempos. Teniendo en cuenta el valor inicial y el valor final (3.39 segundos y 2.96

segundos) podemos decir que ha habido una mejora del 12 %. Teniendo en cuenta una ejecución sin ninguna opción extra donde el tiempo es de alrededor de 4,5s (4,487s en una ejecución arbitraría) podemos decir que la solución ha presentado una mejora del 25 %. Estos resultados no son mejores que la mejor opción por defecto de GCC para esta configuración, ya que la opción «-O3» presenta una mejora del 58 %, arrojando unos 1.8 segundos de ejecución, pero los resultados presentan un comienzo donde se puede ver la evolución a un tiempo menor.

Para esta ejecución hay que tener en cuenta que solo se han ejecutado 1030 individuos (150 en la generación 0 + 44 por las 20 restantes). Con esto podemos calcular el porcentaje de espacio de diseño explorado. Sabiendo que la ejecución contaba con 17 opciones de compilación , de las cuales hay 12 simples, con sólo 2 niveles, 3 de rango, con 65536 niveles (para el cálculo se asumirán 5 niveles, mínimo, primer cuartil, media, segundo cuartil y máximo) y 2 opciones de intervalo con, 3 y 4 niveles respectivamente. Podemos decir que el tamaño del espacio de diseño es de alrededor de 12 millones ($2^{12} * 3^5 * 3 * 4$), eso haría que un 0.008 % del espacio de diseño se haya explorado.

También podemos ver que hay convergencia entre generaciones, incluso usando un conjunto de opciones con *flags* incompatibles entre sí. Nos encontramos con 20 ejecuciones correctas en la generación 0, lo cual hace que solo se computen el 50 % de individuos planeado, pero en la generación 20 nos encontramos con 40 individuos, un 100 % de individuos ejecutados.

6.3.2. Uso de Memoria RAM y Tiempo de ejecución

Siguiendo la linea del experimento anterior esta vez se decide usar el benchmark **treebench** para buscar una solución que busque optimizar el tiempo de ejecución y el consumo de memoria RAM.

En este caso se ha decidido usar un paquete de opciones de compilación más amplio, en este caso con 116 opciones. Se ha configurado los pesos de los objetivos como 0,4 para el consumo de memoria y 0,6 para el tiempo de ejecución. Se ha utilizado un tamaño inicial de población de 150 individuos y 90 individuos para la población general.

Sabiendo que se ha configurado la radiación a 15 podemos decir que tendremos una tasa de mutación del 12.5 % (Radiación/Número de *flags*). Se ha configurado el algoritmo para seleccionar 20 individuos y crear un 10 % de individuos nuevos aleatorios por generación.

Esta vez se ha usado el límite de convergencia configurado a 150 generaciones mínimas y para la ejecución se han hecho uso de 4 hilos concurrentes como máximo a la hora de realizar las pruebas.

Con esta configuración se han obtenido el resultado de WSM de la figura 6.3, donde podemos ver todos los resultados representados en un diagrama de dispersión.

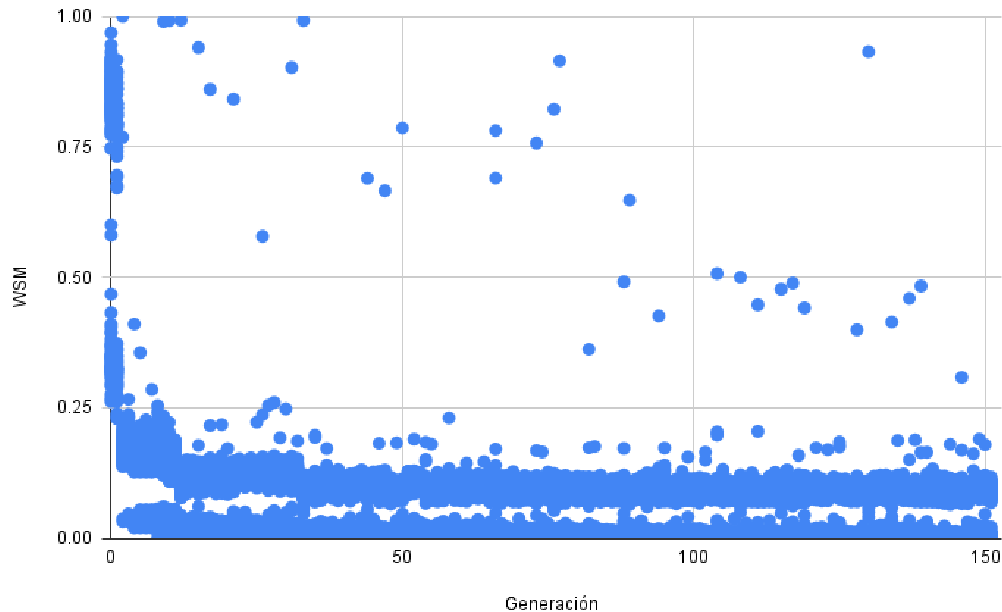


Figura 6.3: Ejecución experimento 2, WSM para el consumo de RAM con un peso de 0,4 y el Tiempo de ejecución con un peso de 0,6.

En este gráfico podemos ver que el límite de convergencia se ha activado al llegar al mínimo, esto es porque el mejor individuo de la ultima generación y el de la anterior eran el mismo. Podemos ver que según avanzan las generaciones el resultado se encuentra más agrupado, y que cada cierto número de de generaciones hay una pequeña expansión de los datos, cada vez menor.

En el caso de tratarse una optimización multiobjetivo usamos WSM para poder representar todos los objetivos, pero puede ser engañoso al poder estar mejorando uno y el otro no. Ya que se escoge la mejor combinación de ambos y eso no implica que todos los objetivos mejoren con el individuo escogido.

Para poder analizar el resultado de cada uno de los objetivos podemos representar el mínimo de cada una de las generaciones. En la figura 6.4, podemos ver el mínimo del tiempo, y en la figura 6.5, podemos ver el mínimo consumo de RAM.

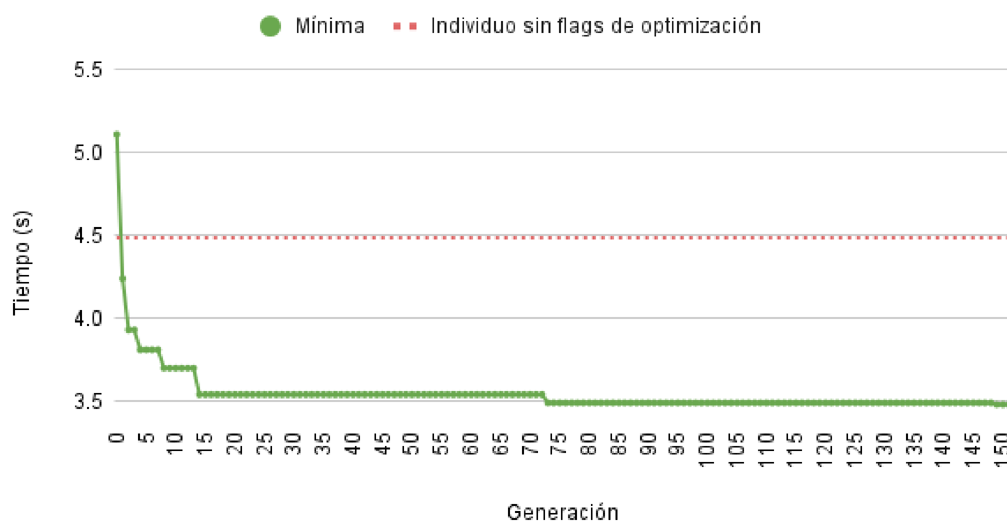


Figura 6.4: Tiempos de ejecución mínimos por generaciones.

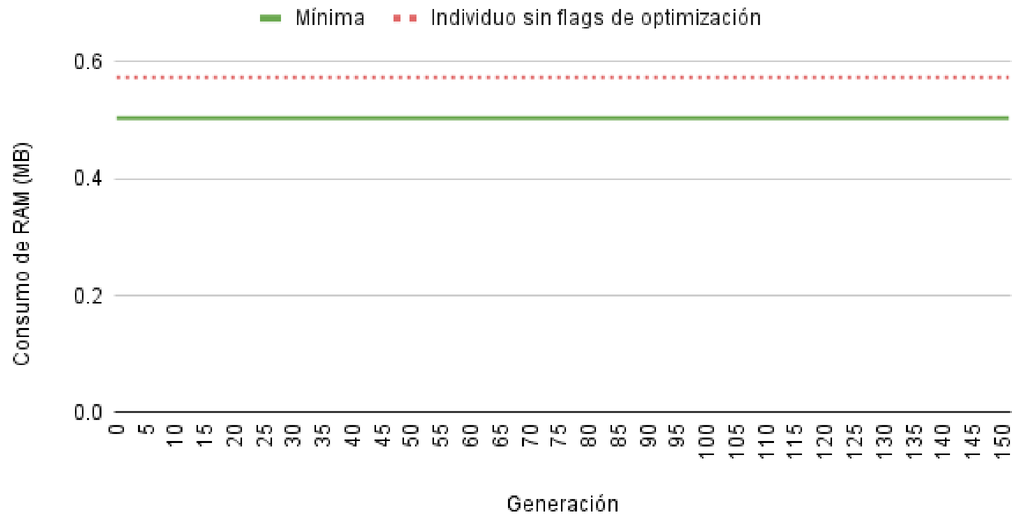


Figura 6.5: Consumos de RAM mínimos por generaciones

Como vemos, el tiempo de ejecución si que ha sufrido mejoras durante la optimización, pero el consumo de memoria RAM se ha establecido.

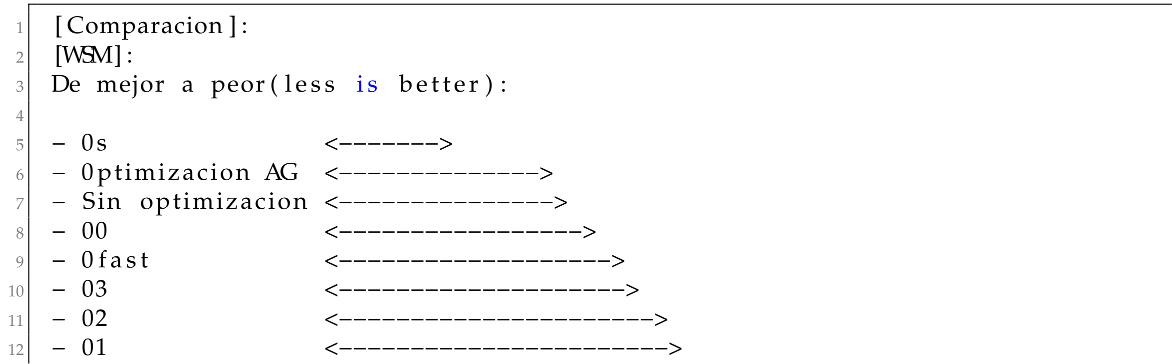
El mejor tiempo de la última generación es de 3,48 segundos. Sabiendo que el mejor tiempo de la primera generación es de 5,11 segundos podemos decir que ha habido una mejora del 31 % y si tenemos en cuenta el tiempo sin optimización que es de 4,48 segundos conseguiríamos una mejora del 22 %. En este caso la primera generación no presenta ninguna mejora sobre la opción sin optimizar.

En el caso de la RAM, vemos que no encontramos mejoras entre generaciones, se establece a un valor de 0.503906 MB y no mejora.

Pero debemos tener en cuenta , por un lado, que las opciones escogidas normalmente están pensadas para mejorar tiempo de ejecución y tamaño del binario en algunas ocasiones, no para consumo de RAM, y que el consumo de RAM sin optimización es de 0,573120 MB.

Esto ultimo indica que si que ha habido mejora, de un 12 %, pero que la cantidad de opciones que afectan al consumo de RAM son pocas. Puede que se hayan seleccionado todas las combinaciones de estas en la primera generación o que la ejecución no haya explorado nuevas zonas del espacio de diseño que si afectan a la RAM.

También hay que tener en cuenta, que puede que con alguna opción de compilación el tiempo mejore o empeore y este efecto tenga más peso a la hora de elegir que la mejora o el empeoramiento del consumo de la RAM.



```
[Comparacion]:
[WSM]:
De mejor a peor(less is better):

- Os                   <------->
- Optimizacion AG      <--------------->
- Sin optimizacion     <---------------->
- O0                   <------------------->
- Ofast                <--------------------->
- O3                   <---------------------->
- O2                   <------------------------>
- O1                   <------------------------->
```

Figura 6.6: Comparación final de opciones predeterminadas.

Como conclusión de este experimento, gracias al ID implementado, podemos saber que se han explorado 10719 individuos diferentes, y que el resultado final de la comparación hace que esta ejecución sea la segunda mejor opción tras la opción de compilación -*Os*, algo que podemos ver en la figura 6.6, que representa la comparación final hecha por el AG.

6.3.3. Optimización con todos los objetivos

Para este experimento se ha hecho uso el benchmark **Matrix**, en este caso este benchmark no recibe argumentos y se ha ejecutado en Azure.

La configuración usada para los pesos es de 0,3 para el consumo de RAM, 0,1 para la carga de la CPU, 0,1 para el tamaño del binario, 0,2 para la Robustez del binario y 0,3 para el Tiempo de ejecución. Con esto se ha realizado una ejecución haciendo uso de todos los objetivos.

La configuración del AG es 150 individuos de tamaño de población inicial, 100 de tamaño de población general y 45 generaciones como límite. Para crear las nuevas generaciones se seleccionan 20 individuos y se crean un 10 % de individuos nuevos aleatorios. Se ha hecho uso del mismo paquete de opciones de compilación del experimento anterior con 116 opciones de compilación, disponibles con el resto de opciones en la tabla C.1, disponible en el Apéndice B. En este caso la tasa de mutación es del 4 % al tener una radiación de 5.

La configuración de las pruebas ha sido de 4 hilos de ejecución máximos concurrentes y 97 ejecuciones para el cálculo de la robustez. Se ha hecho uso de 97 ya que se ha calculado el número de ejecuciones necesarias para tener un nivel de confianza del 95 % y un margen de error del 10 %. Hemos elegido estos valores ya que si seleccionamos un margen de error más pequeño se vuelve muy costosa la ejecución. Con 385 ejecuciones podríamos reducir el margen de error al 5 %, pero el benchmark actual tiene un tiempo de ejecución de 0,16 segundos, algo que haría gastar unos 61 segundos por individuo.

Con la configuración clara, el AG devuelve como distribución del WSM calculado la gráfica de la figura 6.7.

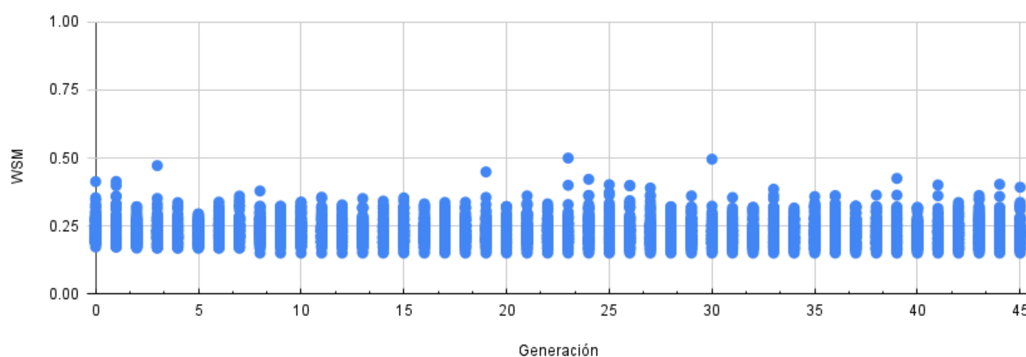


Figura 6.7: Ejecución experimento3, distribución WSM.

En este caso podemos ver como rápidamente se agrupan los resultados en un rango de valores, vemos también que cada ciertas generaciones los resultados se expanden y vuelven a converger.

A la hora de ver cada una de las evoluciones de los objetivos, ver como algunos mejoran y otros se mantienen o no tienen una evolución clara hemos representado el resultado mínimo de cada uno de los objetivos. Podemos ver el Consumo de RAM en la figura 6.8,

la Robustez y Carga de la CPU en la figura 6.9, el tamaño del binario en 6.10 y el tiempo de ejecución en la figura 6.11.

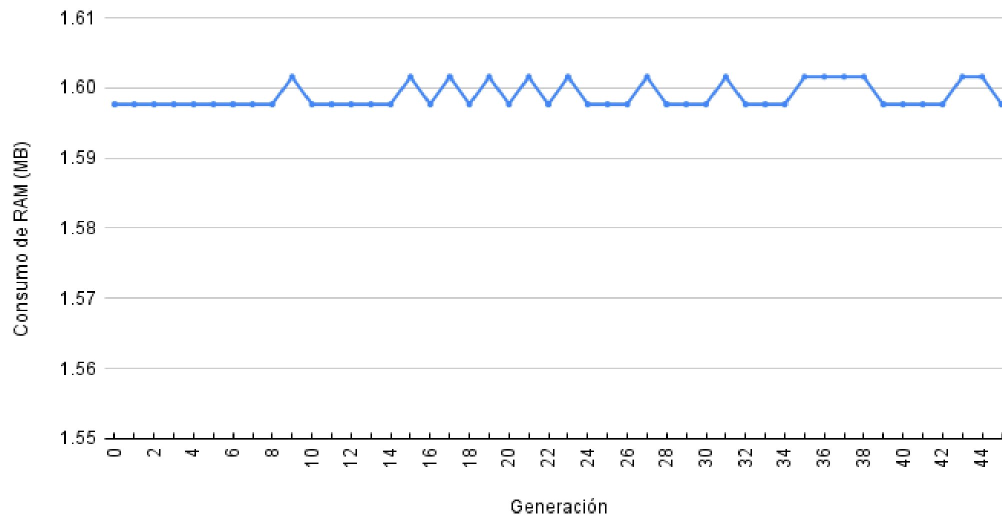


Figura 6.8: Evolución mínima consumo de RAM.

En la figura 6.8 podemos ver como el consumo de RAM se encuentra entre dos valores, y que se van alternando entre ellos, terminando la ejecución con el menor de ellos. Este resultado es un resultado similar al del experimento anterior, donde el valor de RAM no variaba durante la ejecución. En este caso si que varía pero no consigue mejorar en ningún momento el resultado inicial. La mejora es nula, porque el resultado final consigue el mismo consumo que la versión sin ninguna opción de compilación, 1,597656 MB.

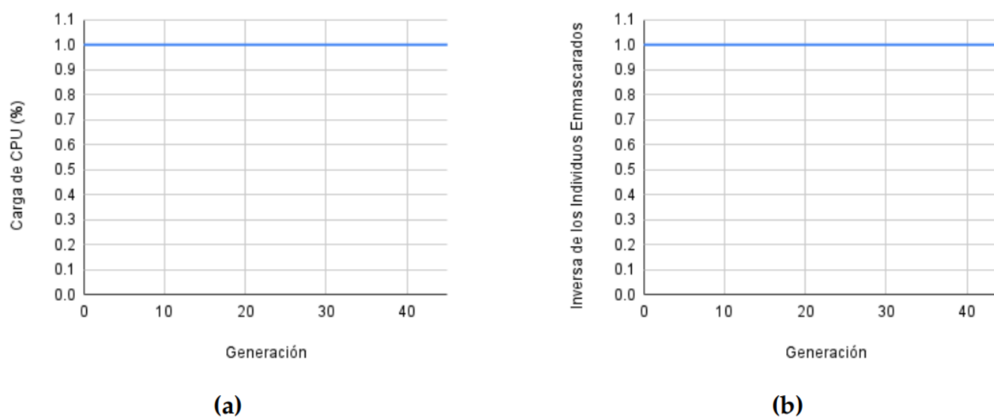


Figura 6.9: (a)Carga de la CPU. (b)Robustez del binario.

En la figura 6.9 vemos un resultado similar para los dos objetivos medidos en estos casos, vemos que el resultado no cambia en toda la evolución, siendo el mismo que la versión sin opciones de compilación, el problema surge cuando se compara con todas las opciones por defecto de optimización y todas arrojan los mismos resultados, en este caso todas arrojan un 100 % de carga de CPU y una tasa de avería de un 100 %, lo que haría un 0 % de enmascarados. Esto puede llevar a pensarnos que las opciones compilación no están afectando al resultado, aunque haría falta una experimentación más profunda.

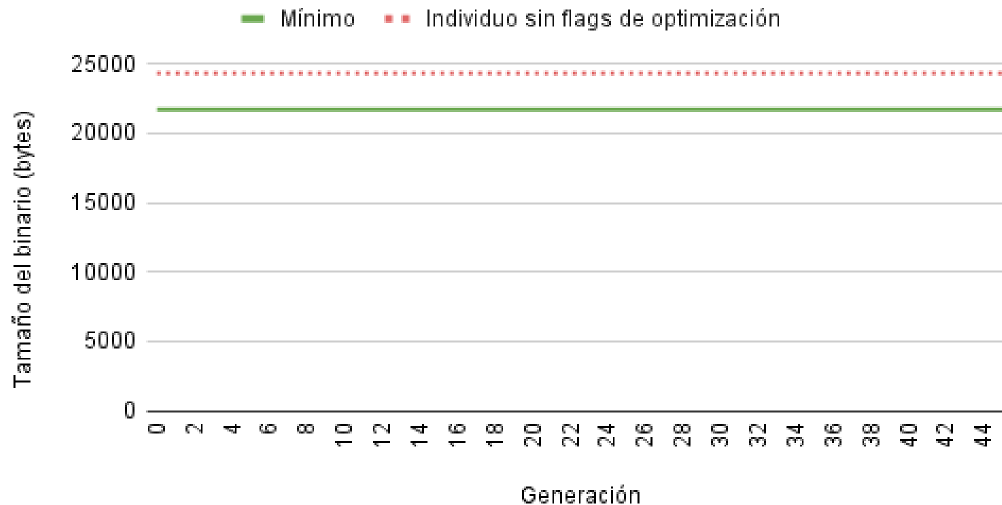


Figura 6.10: Evolución mínima tamaño del binario.

En la figura 6.10 vemos que el tamaño se mantiene durante la ejecución. Pero en este caso, como pasaba con la RAM en el experimento 2, el resultado es mejor que la compilación sin *flags*, con 24320 bytes para el binario sin opciones de compilación y 21720 bytes para la binario del AG, esto hace que consiga una mejora de tamaño del 10 %. Aunque en este caso, teniendo sólo en cuenta el tamaño, algunas opciones predeterminadas de optimización presentan mejores resultados, como en el caso de *-O1* o *-O2* donde el tamaño es de 17632 bytes lo que supondría una mejora del 27 % respecto a la opción sin opciones de compilación.

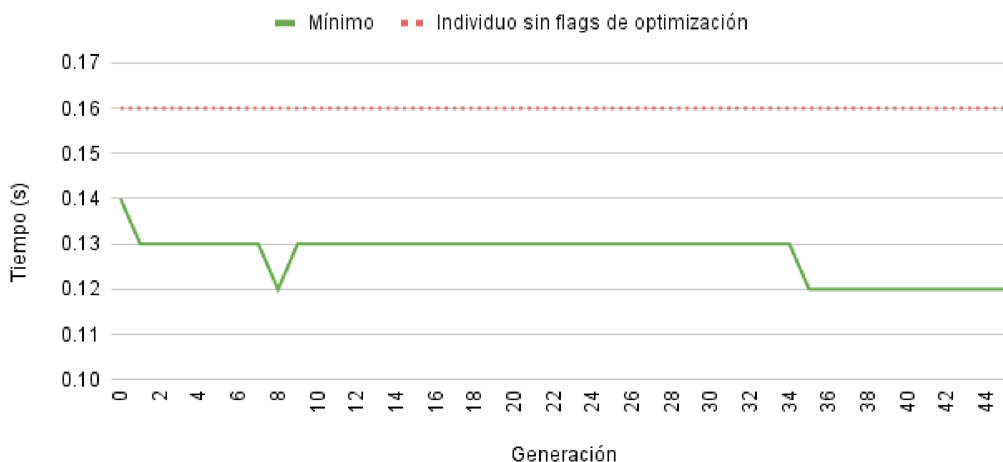


Figura 6.11: Evolución mínima tiempo de ejecución.

En la figura 6.11, vemos la evolución del tiempo de ejecución. En este caso si que podemos ver mejora del tiempo de los resultados mínimos, podemos ver como en la generación 9 el tiempo de ejecución aumenta respecto a la generación anterior. Esto puede suceder por la mejora en otros objetivos, y ser el WSM menor para el caso en el que siga este individuo. Al final de la ejecución obtenemos un tiempo de 0,12 segundos, sabiendo que la ejecución sin opciones de optimización es de 0,16 segundos podemos concluir con que ha habido una mejora del 25 % en el tiempo del individuo final.

Como conclusión, podemos ver en la figura 6.12, una comparación de los objetivos para esta ejecución del AG con los de resultados de la solución sin opciones de optimiza-

ción, recordar que en este caso estamos minimizando por lo tanto los mejores resultados son los menores. También vemos una comparación global, donde podemos ver que el resultado final del AG es globalmente superior a la compilación sin opciones de compilación.

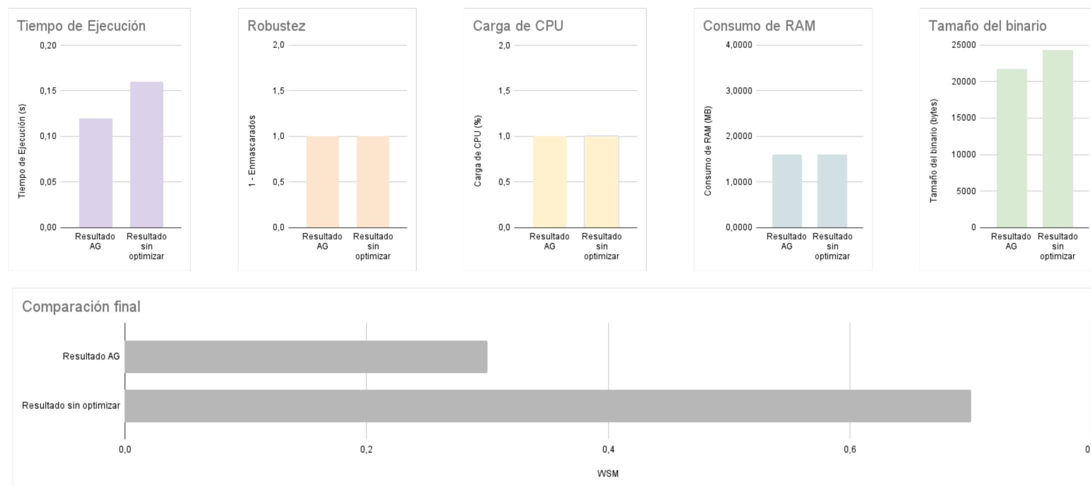


Figura 6.12: Algoritmo Genético vs Compilación sin optimización.

Por último, algo que afecta a todos los experimentos es la selección de opciones. No se han usado todas las opciones disponibles en GCC, ni todas las usadas por las opciones disponibles, aún así, como en el caso del experimento 2, se ha conseguido aproximar bastante el resultado a las opciones por defecto de GCC. Sería necesario realizar más experimentos con mayor número de opciones, algo muy costoso por el crecimiento exponencial del espacio de diseño, pero como conclusión, podemos decir que el algoritmo esta funcionando en ciertos objetivos y en otros haría falta más experimentación para determinar su funcionamiento, inabarcable en algunos casos como la robustez al ser un objetivo costoso de ejecutar.

CAPÍTULO 7
Conclusión

En el último capítulo del trabajo se van a presentar dos apartados, en el primero de ellos se van a repasar todas las asignaturas de los estudios cursados que han formado parte de los conocimiento previos necesarios para realizar el trabajo, y han ayudado a la hora de aprender nuevas técnicas. En el siguiente punto se repasan los objetivos alcanzados y los objetivos a futuro del trabajo, que partes sería interesante reforzar o nuevos trabajos a realizar a partir de la solución propuesta.

7.1 Relación con los estudios cursados

Durante el estudio de grado se han realizado muchas asignaturas. Todas ellas han ayudado a saber como afrontar nuevos problemas en el área de la informática y que metodologías seguir a la hora de realizar todo el trabajo, estas son algunas de ellas y como han facilitado la consecución del trabajo:

- **Fundamentos de Computadores (FCO)**, **Estructura de Computadores (ETC)**, **Arquitectura e Ingeniería de computadores (AIC)**, **Arquitecturas Avanzadas (AAV)**, **Diseño de Sistemas Digitales (DSD)** y **Hacking Ético (HET)**, en la comprensión del funcionamiento de un sistema a bajo nivel. Entendiendo lo que una pequeña mejora puede causar en el resto del sistema.
- **Introducción a la Programación (IIP)**, **Programación (PRG)**, **Lenguajes, tecnologías y paradigmas de la programación (LTP)** y **Estructuras de Datos y Algoritmos (EDA)**, en los conocimientos necesarios de programación, tecnologías y estructuras de datos necesarios para poder crear un proyecto de programación desde cero, elegir las mejores estructuras de datos para organizar los datos en el programa y comprender el funcionamiento y tipos de lenguajes de programación para elegir el lenguaje y las herramientas de optimización.
- **Concurrencia y Sistemas Distribuidos (CSD)**, **Computación paralela (CPA)** y **Lenguajes y entornos de programación paralela (LPP)**, para comprender el funcionamiento de los programas concurrentes y poder aplicar los conocimientos adquiridos al diseño de una solución concurrente.

7.2 Conclusiones y trabajo futuro

Tras analizar los diferentes métodos de investigación operativa, se decidió usar un Algoritmo Genético. Con este método se consiguió alcanzar el objetivo principal del trabajo, presentar una propuesta como herramienta de optimización multiobjetivo de binarios.

En un futuro existe la posibilidad de implementar una solución con otro método de investigación operativa, pero podemos decir que se ha cumplido el objetivo principal con la herramienta presentada.

Uno de los principales problemas encontrados es la selección de objetivos y cómo estos se ven afectados por las opciones de compilación. Aunque es necesaria más investigación y experimentación algunos objetivos, como la Robustez o la Carga de CPU, no se han visto afectados por las diferentes opciones usadas. Por lo tanto, uno de los trabajos futuros que surgen es la investigación y experimentación, en general y centrándose en estos objetivos, para concluir si afectan y como afectan las opciones de compilación en estos objetivos.

Otro de los problemas vistos, es la fiabilidad de las herramientas, sería necesario crear nuevas herramientas de medición fiables o aplicar técnicas como la metrología a la hora de devolver un resultado. También existe la posibilidad de crear una herramienta con algún método que devuelva una solución precomputada, donde los datos usados para llegar a la solución se hayan comprobado y donde se consiga eliminar la espera de ejecución del algoritmo. Muchos de estos problemas vienen derivados de crear una solución sobre un sistema operativo, donde es el propio sistema quien decide prioridades de ejecución y que dependiendo del contexto de ejecución del algoritmo puede causar más incertidumbre.

Siguiendo con posibles proyectos futuros podemos encontrar el uso e investigación de las opciones no creadas para optimización, experimentando si se puede conseguir mejorar algún objetivo planteado con estas. También sería un objetivo futuro el uso de opciones dependiente de la arquitectura del sistema o la implementación en sistemas empotrados. Para estos últimos sería especialmente útil, ya que suelen tener unos requisitos recortados en cuanto a potencia y una parte positiva de usarlos es poder instrumentalizarlos para tomar las medidas y eliminar la incertidumbre normalmente causada por el Sistema Operativo al no disponer de él.

Como vemos, el futuro del proyecto es inmenso y cada uno de estos futuros proyectos puede involucrar nuevos trabajos completos. Concluyendo, podemos decir que se han cumplido todos los objetivos planteados, presentando una posible solución al problema y un acercamiento de la optimización multiobjetivo de binarios.

Bibliografía

[1] G. Hornby, A. Globus, D. Linden, and J. Lohn, "Automated antenna design with evolutionary algorithms," *Collection of Technical Papers - Space 2006 Conference*, vol. 1, 09 2006.

[2] B. Jenkins. Design optimization: Past, present and future. [Online]. Disponible en: https://www.digitalengineering247.com/article/design-optimization-past-present-and-future/ (Accedido por última vez: 03/09/2021).

[3] NIST. Nist/sematech e-handbook of statistical methods. [Online]. Disponible en: https://www.itl.nist.gov/div898/handbook/pri/section3/pri333.htm (Accedido por última vez: 03/09/2021).

[4] NIST. Nist/sematech e-handbook of statistical methods. [Online]. Disponible en: https://www.itl.nist.gov/div898/handbook/pri/section3/pri334.htm (Accedido por última vez: 03/09/2021).

[5] ASQ Statistics Division, *GLOSSARY AND TABLES FOR STATISTICAL QUALITY CONTROL, FOURTH EDITION (E-BOOK)*. ASQ, 2004.

[6] I. Tuzov, D. de Andrés, and J.-C. Ruiz, "Tuning synthesis flags to optimize implementation goals: Performance and robustness of the leon3 processor as a case study," *Journal of Parallel and Distributed Computing*, vol. 112, pp. 84–96, 2018. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0743731517302708 (Accedido por última vez: 03/09/2021).

[7] P. de Aguiar, B. Bourguignon, M. Khots, D. Massart, and R. Phan-Than-Luu, "D-optimal designs," *Chemometrics and Intelligent Laboratory Systems*, vol. 30, no. 2, pp. 199–210, 1995. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/016974399400076X (Accedido por última vez: 03/09/2021).

[8] NIST. Nist/sematech e-handbook of statistical methods. [Online]. Disponible en: https://www.itl.nist.gov/div898/handbook/pri/section5/pri521.htm (Accedido por última vez: 03/09/2021).

[9] I. Tuzov, D. de Andrés, and J.-C. Ruiz, "Robustness-aware design space exploration through iterative refinement of d-optimal designs," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 23–30.

[10] A. M. Andaluz. Algoritmos evolutivos y algoritmos genéticos. [Online]. Disponible en: http://www.it.uc3m.es/~jvillena/irc/practicas/estudios/aeag (Accedido por última vez: 03/09/2021).

[11] D. Moss, P. Mejía-Alvarez, H. Aydin, and R. Melhem, "An incremental server for scheduling overloaded real-time systems," *IEEE Transactions on Computers*, vol. 40, no. 10, pp. 1347–1361, oct 2003.

[12] K. Takayama, M. Fujikawa, Y. Obata, and M. Morishita, "Neural network based optimization of drug formulations," *Advanced Drug Delivery Reviews*, vol. 55, no. 9, pp. 1217–1231, 2003, artificial neural network modeling for pharmaceutical research. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0169409X03001200 (Accedido por última vez: 03/09/2021).

[13] K. Muralitharan, R. Sakthivel, and R. Vishnuvarthan, "Neural network based optimization approach for energy demand prediction in smart grid," *Neurocomputing*, vol. 273, pp. 199–208, 2018. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/S0925231217313681 (Accedido por última vez: 03/09/2021).

[14] Z. Chourabi, F. Khedher, A. Babay, and M. Cheikhrouhou, "Multi-criteria decision making in workforce choice using ahp, wsm and wpm," *The Journal of The Textile Institute*, vol. 110, no. 7, pp. 1092–1101, 2019. [Online]. Disponible en: https://doi.org/10.1080/00405000.2018.1541434 (Accedido por última vez: 03/09/2021).

[15] E. Triantaphyllou and S. H. Mann, "An examination of the effectiveness of multi-dimensional decision-making methods: A decision-making paradox," *Decision Support Systems*, vol. 5, no. 3, pp. 303–312, 1989. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/0167923689900377 (Accedido por última vez: 03/09/2021).

[16] F. Jose, S. Greco, and M. Ehrgott, *Multiple criteria decision analysis: state of the art surveys*. Springer, 2016.

[17] Google. Data preparation and feature engineering for machine learning, normalization. [Online]. Disponible en: https://developers.google.com/machine-learning/data-prep/transform/normalization (Accedido por última vez: 03/09/2021).

[18] M. del Carmen Bas Cerdà, *Estrategias metodológicas para la construcción de indicadores compuestos en la gestión universitaria*, 2014, ch. Capitulo 3: Metodologías y proceso de construcción de un Indicador Compuesto, p. 88.

[19] GNU. Invoking gcc. gcc command options. [Online]. Disponible en: https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html#Invoking-GCC (Accedido por última vez: 03/09/2021).

[20] S. R. Ladd. Acovea: Using natural selection to investigate software complexities. [Online]. Disponible en: https://web.archive.org/web/20060805085957/http://coyotegulch.com/products/acovea/index.html (Accedido por última vez: 03/09/2021).

[21] S. R. Ladd. Acovea: Describing the evolutionary algorithm. [Online]. Disponible en: https://web.archive.org/web/20060721110220/http://www.coyotegulch.com/products/acovea/acoveaga.html (Accedido por última vez: 03/09/2021).

[22] S. R. Ladd. Acovea analysis of compiler options via evolutionary algorithm. [Online]. Disponible en: https://github.com/Acovea/libacovea (Accedido por última vez: 03/09/2021).

[23] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013, 2013 International Conference on Computational Science. [Online]. Disponible en: https://www.sciencedirect.com/science/article/pii/S1877050913004419 (Accedido por última vez: 03/09/2021).

[24] L. A. Vivas Tejuelo, J. Calvo-Zaragoza, F. Restrepo Calle, S. Cuenca-Asensi, A. Ortiz García, and A. Martínez-Álvarez, "Mejora del rendimiento de una aplicación mediante la determinación automática de las opciones óptimas de compilación," 2010-09.

[25] Fursin, G., Kashnikov, Y., Memon, A.W. et al., "Milepost gcc: Machine learning enabled self-tuning compiler," *Int J Parallel Prog*, vol. 39, p. 296–327, 2011. [Online]. Disponible en: https://link.springer.com/article/10.1007%2Fs10766-010-0161-2 (Accedido por última vez: 03/09/2021).

[26] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: a machine learning guided compiler optimizations framework," *CoRR*, vol. abs/2101.04808, 2021. [Online]. Disponible en: https://arxiv.org/abs/2101.04808 (Accedido por última vez: 03/09/2021).

[27] C. Lattner. The llvm compiler infrastructure. [Online]. Disponible en: https://llvm.org/ (Accedido por última vez: 03/09/2021).

[28] J.-C. Fernandez, L. Mounier, and C. Pachon, "A model-based approach for robustness testing," in *Testing of Communicating Systems*, F. Khendek and R. Dssouli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 333–348.

[29] J. Seward, et al. Valgrind documentation. [Online]. Disponible en: https://valgrind.org/docs/ (Accedido por última vez: 03/09/2021).

[30] F. Pedregosa and P. Gervais. Memory profiler. [Online]. Disponible en: https://github.com/pythonprofilers/memory_profiler (Accedido por última vez: 03/09/2021).

[31] Thomas Robitaille, et al. Psrecord. [Online]. Disponible en: https://github.com/astrofrog/psrecord (Accedido por última vez: 03/09/2021).

[32] Sebastien Godard. sysstat - system performance tools for the linux operating system. [Online]. Disponible en: https://github.com/sysstat/sysstat (Accedido por última vez: 03/09/2021).

[33] V. Porpodas and N. Voorhies. Zofi: The zero overhead fault injector project. [Online]. Disponible en: https://github.com/vporpo/zofi (Accedido por última vez: 03/09/2021).

[34] V. Porpodas, "ZOFI: zero-overhead fault injection tool for fast transient fault coverage analysis," *CoRR*, vol. abs/1906.09390, 2019. [Online]. Disponible en: http://arxiv.org/abs/1906.09390 (Accedido por última vez: 03/09/2021).

[35] S. B. López. Ombag: Optimización multiobjetivo de binarios usando algoritmos geneticos. [Online]. Disponible en: https://github.com/sg1o/OMBAG (Accedido por última vez: 03/09/2021).

[36] K. Hayen. Nuitka documentation. [Online]. Disponible en: https://nuitka.net/ (Accedido por última vez: 03/09/2021).

[37] J. D. Hunter. Matplotlib: Visualization with python. [Online]. Disponible en: https://matplotlib.org/ (Accedido por última vez: 03/09/2021).

[38] S. R. Ladd. Treebench benchmark. [Online]. Disponible en: https://github.com/Acovea/libacovea/tree/master/benchmarks (Accedido por última vez: 03/09/2021).

[39] T. Djärv. Matrix benchmark. [Online]. Disponible en: https://github.com/thundermoose/matrix_matrix_multiplication_benchmark (Accedido por última vez: 03/09/2021).

APÉNDICE A
Versión Modelo de Islas

Al finalizar con el trabajo realizado se ha creado una versión preliminar del AG siguiendo el Modelo de Islas. Esta versión recicla gran parte del código de la versión anterior, añadiendo simplemente ciertos bucles para generar poblaciones que representan a las Islas, estas poblaciones se encuentran separadas.

La única nueva pieza añadida es el intercambio de individuos entre islas. Este intercambio sucede cada cierto número de generaciones, configurables por el usuario, e inserta los mejores individuos de una isla en otra. Ese proceso se repite tantas veces como islas hay.

Los resultados de las pruebas preliminares parecen positivos. Parece que se consigue el mismo resultado que en la versión anterior pero con un número menor de generaciones. Sería necesario un estudio de las variables de configuración y un análisis más profundo para poder tener una comprobación del funcionamiento correcto del AG, pero, el uso del modelo de islas puede ser una alternativa a la solución planteada.

APÉNDICE B
Manual de Uso

La solución creada esta disponible Github[1]. Para poder usar la diferentes aplicaciones lo primero que tendremos que hacer es clonar el repositorio en nuestra máquina, para ello ejecutaremos la orden B.1.

```
1 $ git clone https://github.com/sg1o/TFG-optimization
```

Código B.1: Clonación repositorio

Instalación

Ambas versiones de los algoritmos están implementadas en Python 3, no usan módulos externos y no tienen dependencias de código, pero, para la medición de cada individuo es necesario tener instalado algunas aplicaciones.

Todas las aplicaciones son compatibles con sistemas basados en Debian, testeadas en Ubuntu 20.04 LTS y Ubuntu Server 20.04.3 LTS.

Para instalar las dependencias cuentas con un script de instalación. Pueden instalarse ejecutando:

```
1 # Para instalar las dependencias de la aplicacion de optimizacion:
2 $ sudo ./instalacion.sh
3 # Para instalar las dependencias del codigo auxiliar:
4 $ sudo ./auxiliarCode/instalacionDependenciasAuxiliares.sh
```

Código B.2: Instalación de dependencias.

Una vez instaladas las dependencias ya pueden ejecutarse los algoritmos genéticos.

Uso Algoritmo Genético

Para lanzar a ejecución puede usarse la orden del código B.3.

```
1 $ python3 optimize.py -p path/al/binario [ -a <ARGS> ]
```

Código B.3: Lanzamiento de ejecución AG.

Donde *-p* indica el *path* del programa a ejecutar y si tiene con *-a* pueden indicarse los argumentos.

Existen otros parámetros, y existe la opción *-h* para una ayuda con su uso, en la código B.4 podemos ver la ejecución de la ayuda.

[1]Disponible en: https://github.com/sg1o/OMBAG

```
$ python3 optimize.py -h
  usage: optimize.py [-h] [-p PROGRAM] [-a ARGUMENTS] [-i] [-dF]

  optional arguments:
    -h, --help              Mostrar este mensaje y salir
    -p PROGRAM, --programa PROGRAM
                            Path absoluto del programa a optimizar
    -a ARGUMENTS, --arguments ARGUMENTS
                            Argumentos del programa para hacer pruebas
    -i, --imprimir          Imprimir por pantalla los resultados de las pruebas
                            durante la ejecucion
    -dF, --debugFlags       Imprimir por pantalla los errores de compilacion
                            nuevos.
```

Código B.4: Lanzamiento de ayuda AG.

Programas auxiliares

En el codigo B.5 podemos ver el uso de los programas auxiliares creados.

```
$ python3 crearFlags.py # Programa que facilita la creacion manual del
                        # archivo JSON con las opciones de compilacion

$ python3 fileToJSON.py <path/to/file/> # Crea un archivo JSON para el
                                        # AG a partir de un archivo CSV

$ python3 graficasBarras.py <path/to/csv> # Crea un grafico de barras
                                          # a partir de un csv, creado
                                          # para los csv con las flags

$ python3 graficasScatter.py <path/to/csv> 'titutlo grafica 1' 'titulo
    grafica 2' 'titulo gradica 3'
                              # Muestra tres graficas con la distribucion
                              # de los resultados, la minima y la media
```

Código B.5: Uso de programas auxiliares.

APÉNDICE C
Opciones optimización GCC

En este apéndice podemos encontrar la tabla C.1, donde encontramos las opciones de optimización disponibles en GCC, indicando las usadas por cada experimento mostrado.

Tabla C.1: Opciones Optimización GCC.

Flags por defecto optimización GCC			
Tipo	Flag	Rango o intevalo	Usado en Experimento
intervalo	-fvect-cost-model=	dynamic, cheap, unlimited	2, 3
intervalo	-fsimd-cost-model=	dynamic, cheap, unlimited	2, 3
intervalo	-freorder-blocks-algorithm=	simple, stc	2, 3
intervalo	-flto-partition=	1to1, max, balanced, one, none	
intervalo	-flive-patching=	inline-clone, inline-only-static	2, 3
intervalo	-fira-region=	all, mixed, one	1, 2, 3
intervalo	-fira-algorithm=	priority, CB	2, 3
intervalo	-ffp-contract=	fast, off, on	2, 3
intervalo	-fexcess-precision=	fast, standard	1, 2, 3
rango	-ftree-parallelize-loops=	1, 4	
rango	-fsched-stalled-insns=	0, 65536	
rango	-fsched-stalled-insns-dep=	0, 65536	
rango	-finline-limit=	0, 65536	1, 2, 3
rango	-falign-loops=	0, 65536	1, 2, 3
rango	-falign-labels=	0, 65536	2, 3
rango	-falign-jumps=	0, 65536	
rango	-falign-functions=	0, 65536	1, 2, 3
simple	-fwpa		
simple	-fwhole-program		
simple	-fweb		
simple	-fvpt		
simple	-fversion-loops-for-strides		2, 3
simple	-fvect-cost-model		2, 3
simple	-fvariable-expansion-in-unroller		
simple	-fuse-linker-plugin		
simple	-funswitch-loops		2, 3
simple	-funsafe-math-optimizations		
simple	-funroll-loops		
simple	-funroll-all-loops		
simple	-funit-at-a-time		2, 3
simple	-funconstrained-commons		
simple	-ftree-vrp		2, 3
simple	-ftree-vectorize		
simple	-ftree-ter		1, 2, 3
simple	-ftree-tail-merge		2, 3
simple	-ftree-switch-conversion		2, 3
simple	-ftree-sra		2, 3

Tipo	Flag	Rango o intevalo	Usado en Experimento
simple	-ftree-slsr		
simple	-ftree-slp-vectorize		2, 3
simple	-ftree-sink		2, 3
simple	-ftree-scev-cprop		2, 3
simple	-ftree-reassoc		
simple	-ftree-pta		2, 3
simple	-ftree-pre		2, 3
simple	-ftree-phiprop		2, 3
simple	-ftree-partial-pre		2, 3
simple	-ftree-loop-vectorize		2, 3
simple	-ftree-loop-optimize		
simple	-ftree-loop-linear		
simple	-ftree-loop-ivcanon		
simple	-ftree-loop-im		
simple	-ftree-loop-if-convert		
simple	-ftree-loop-distribution		2, 3
simple	-ftree-loop-distribute-patterns		
simple	-ftree-fre		2, 3
simple	-ftree-forwprop		2, 3
simple	-ftree-dse		2, 3
simple	-ftree-dominator-opts		
simple	-ftree-dce		1, 2, 3
simple	-ftree-copy-prop		2, 3
simple	-ftree-coalesce-vars		2, 3
simple	-ftree-ch		2, 3
simple	-ftree-ccp		2, 3
simple	-ftree-builtin-call-dce		1, 2, 3
simple	-ftree-bit-ccp		2, 3
simple	-ftracer		
simple	-fthread-jumps		2, 3
simple	-fstrict-aliasing		2, 3
simple	-fstore-merging		2, 3
simple	-fstdarg-opt		
simple	-fssa-phiopt		2, 3
simple	-fssa-backprop		2, 3
simple	-fsplit-wide-types		2, 3
simple	-fsplit-paths		2, 3
simple	-fsplit-loops		2, 3
simple	-fsplit-ivs-in-unroller		
simple	-fsingle-precision-constant		
simple	-fsignaling-nans		
simple	-fshrink-wrap-separate		2, 3
simple	-fshrink-wrap		2, 3
simple	-fsemantic-interposition		
simple	-fselective-scheduling2		
simple	-fselective-scheduling		
simple	-fsel-sched-pipelining-outer-loops		
simple	-fsel-sched-pipelining		
simple	-fsection-anchors		
simple	-fschedule-insns2		2, 3
simple	-fschedule-insns		2, 3
simple	-fschedule-fusion		
simple	-fsched2-use-superblocks		
simple	-fsched-spec-load-dangerous		
simple	-fsched-spec-load		
simple	-fsched-spec-insn-heuristic		
simple	-fsched-spec		2, 3

Tipo	Flag	Rango o intevalo	Usado en Experimento
simple	-fsched-rank-heuristic		
simple	-fsched-pressure		
simple	-fsched-last-insn-heuristic		
simple	-fsched-interblock		2, 3
simple	-fsched-group-heuristic		2, 3
simple	-fsched-dep-count-heuristic		
simple	-fsched-critical-path-heuristic		
simple	-fsave-optimization-record		
simple	-frounding-math		
simple	-freschedule-modulo-scheduled-loops		
simple	-frerun-cse-after-loop		2, 3
simple	-freorder-functions		
simple	-freorder-blocks-and-partition		2, 3
simple	-freorder-blocks		2, 3
simple	-frename-registers		
simple	-free		
simple	-freciprocal-math		
simple	-fprofile-values		
simple	-fprofile-use		
simple	-fprofile-reorder-functions		
simple	-fprofile-correction		
simple	-fprefetch-loop-arrays		2, 3
simple	-fpredictive-commoning		2, 3
simple	-fpeel-loops		2, 3
simple	-fpartial-inlining		2, 3
simple	-foptimize-strlen		2, 3
simple	-foptimize-sibling-calls		2, 3
simple	-fomit-frame-pointer		2, 3
simple	-fno-zero-initialized-in-bss		
simple	-fno-trapping-math		
simple	-fno-toplevel-reorder		
simple	-fno-signed-zeros		
simple	-fno-sched-spec		
simple	-fno-sched-interblock		
simple	-fno-printf-return-value		
simple	-fno-peephole2		
simple	-fno-peephole		
simple	-fno-math-errno		
simple	-fno-ira-share-spill-slots		
simple	-fno-ira-share-save-slots		
simple	-fno-inline		
simple	-fno-guess-branch-probability		
simple	-fno-function-cse		
simple	-fno-fp-int-builtin-inexact		
simple	-fno-defer-pop		
simple	-fno-branch-count-reg		
simple	-fmove-loop-invariants		2, 3
simple	-fmodulo-sched-allow-regmoves		
simple	-fmodulo-sched		
simple	-fmerge-constants		2, 3
simple	-fmerge-all-constants		
simple	-flto-compression-level		
simple	-flto		
simple	-flra-remat		2, 3
simple	-floop-unroll-and-jam		1, 2, 3
simple	-floop-strip-mine		
simple	-floop-parallelize-all		

Tipo	Flag	Rango o intevalo	Usado en Experimento
simple	-floop-nest-optimize		
simple	-floop-interchange		2, 3
simple	-floop-block		
simple	-flive-range-shrinkage		
simple	-flimit-function-alignment		
simple	-fkeep-static-functions		
simple	-fkeep-static-consts		
simple	-fkeep-inline-functions		
simple	-fivopts		
simple	-fisolate-erroneous-paths-dereference		2, 3
simple	-fisolate-erroneous-paths-attribute		
simple	-fira-loop-pressure		
simple	-fira-hoist-pressure		
simple	-fipa-vrp		2, 3
simple	-fipa-stack-alignment		
simple	-fipa-sra		
simple	-fipa-reference-addressable		1, 2, 3
simple	-fipa-reference		1, 2, 3
simple	-fipa-ra		
simple	-fipa-pure-const		2, 3
simple	-fipa-pta		
simple	-fipa-profile		2, 3
simple	-fipa-modref		2, 3
simple	-fipa-icf		2, 3
simple	-fipa-cp-clone		2, 3
simple	-fipa-cp		2, 3
simple	-fipa-bit-cp		2, 3
simple	-finline-small-functions		2, 3
simple	-finline-functions-called-once		1, 2, 3
simple	-finline-functions		2, 3
simple	-findirect-inlining		2, 3
simple	-fif-conversion2		2, 3
simple	-fif-conversion		2, 3
simple	-fhoist-adjacent-loads		2, 3
simple	-fguess-branch-probability		2, 3
simple	-fgraphite-identity		
simple	-fgcse-sm		
simple	-fgcse-lm		2, 3
simple	-fgcse-las		
simple	-fgcse-after-reload		2, 3
simple	-fgcse		2, 3
simple	-ffunction-sections		
simple	-fforward-propagate		2, 3
simple	-ffloat-store		
simple	-ffinite-math-only		
simple	-ffinite-loops		2, 3
simple	-ffat-lto-objects		
simple	-ffast-math		
simple	-fexpensive-optimizations		2, 3
simple	-fearly-inlining		
simple	-fdse		2, 3
simple	-fdevirtualize-speculatively		2, 3
simple	-fdevirtualize-at-ltrans		
simple	-fdevirtualize		2, 3
simple	-fdelete-null-pointer-checks		2, 3
simple	-fdelayed-branch		2, 3
simple	-fdefer-pop		1, 2, 3

Tipo	Flag	Rango o intevalo	Usado en Experimento
simple	-fdce		2, 3
simple	-fdata-sections		
simple	-fcx-limited-range		
simple	-fcx-fortran-rules		
simple	-fcse-skip-blocks		2, 3
simple	-fcse-follow-jumps		2, 3
simple	-fcrossjumping		2, 3
simple	-fcprop-registers		2, 3
simple	-fcompare-elim		2, 3
simple	-fcombine-stack-adjustments		2, 3
simple	-fcode-hoisting		1, 2, 3
simple	-fcaller-saves		2, 3
simple	-fbranch-count-reg		1, 2, 3
simple	-fsched-spec		2,3