ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

SISTEMA DE COMPILACIÓN INTELIGENTE

INTELLIGENT COMPILATION SYSTEM

Realizado por
Miguel de la Morena Pérez

Tutorizado por
Gabriel Jesús Luque Polo

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2019

Fecha defensa:

Fdo. El/la Secretario/a del Tribunal

Resumen

Este trabajo de fin de grado (TFG) titulado "Sistema de Compilación Inteligente" ofrece al programador una herramienta con la que pueda optimizar los ficheros ejecutables que han desarrollado para los lenguajes C y C++ mediante el compilador "GNU Compiler Collection" (gcc). Esto se basa en la utilización de las opciones activables que dicho compilador ofrece para llegar a una versión lo más optima posible en un tiempo computable del programa en cuestión. Partiendo de esta premisa, podemos optimizar el ejecutable en dos ámbitos, por el tiempo de ejecución o por el tamaño del fichero. La gran ventaja de utilizar esta herramienta es la de ser una inversión a largo plazo, que permite consumir un poco más de tiempo en la fase de desarrollo para incrementar la eficiencia en el producto final. En definitiva, el programa ofrece una alternativa a la opción de compilado "O3" que ofrece gcc y que no se adapta a las necesidades específicas de cada programa.

Palabras clave: C, C++, Compilación, Flags.

Abstract

This Final Degree Project (FDP) titled "Intelligent Compilation System" offers the programmer a tool with which he could optimize the executable files they developed for C and C++ languages through the "GNU Compiler Collection" (gcc). This is based on usage of boolean flags offered by this compilator to achieve the best version possible in a computable time. Starting off this premise , we can optimize the program in two different ways, by the execution time or by the size of the file. The best advantage of using this tool is to be a long-term investment, that allows you to consume a little bit of time in the development phase to increase the efficiency on the final product. The program offers an aternative to "O3" option presented by gcc , which doesn´t fit with the specific program´s requirements.

Keywords: C, C++, Compilation, Flags.

Índice

Contenido

Introducción 1
1.1 Motivación. 1
1.2 Objetivos. 2
1.3 Estructura de la memoria. 3
Tecnología y Recursos 5
2.1 Herramientas para el desarrollo. 5
2.3 Herramientas para la gestión. 7
2.3 Herramientas para la documentación. 9
Análisis y Diseño 11
3.1 Requisitos funcionales 11
3.2 Requisitos no funcionales 12
3.3 Diagrama de Casos de Uso 13
3.4 Diagrama de Clase 17
3.5 Diagrama de Secuencia 20
3.6 Diagrama de Navegación 21
Desarrollo e Implementación 23
4.1 Desarrollo de la Interfaz de Usuario 23
4.2 Desarrollo del motor de optimización 25
Pruebas 31
Conclusiones y Líneas Futuras 37
5.1 Conclusiones 37
5.2 Líneas Futuras 37
Bibliografía 39

1

Introducción

En este capítulo mostraremos los motivos que llevaron a la realización de este proyecto y sus principales objetivos. Finalmente mostraremos la organización de este documento.

1.1 Motivación.

Desde los comienzos de la programación, siempre hemos tenido la necesidad de poder traducir nuestro lenguaje de alto nivel, a uno más legible para los computadores, bien sea mediante un intérprete o un compilador.

El proceso de compilación y enlazado para obtener el código ejecutable final es un proceso complejo que puede modificarse mediante opciones del compilador. Dependiendo de las opciones elegidas podemos obtener códigos con diferentes características de rendimiento y tamaño. La dificultad viene debido a que el número de opciones disponibles es muy grande (compiladores como GCC para C/C++ disponen de varios centenares de opciones) y es complicado para el programador decidir cuáles son las más adecuadas para su código.

Muchos compiladores optan por ofrecen modos de compilación que agrupan muchas opciones con el propósito de ofrecer de forma general un código resultante eficiente. Un ejemplo de esto son las opciones -O3 o -Ofast del compilador GCC para C/C++, que suelen generar códigos muy eficientes. Pero las opciones activadas por esos modos son genéricas y no se generan de forma dinámica atendiendo al código a compilar en ese momento. Esto plantea la pregunta de si sería posible adaptar las opciones (flags) de compilación para código concreto.

Este ajuste personalizado de las opciones permitiría obtener un ejecutable más eficiente. Hay que tener en cuenta para muchos sistemas (sistemas críticos, en tiempo real, sistemas que se ejecutan constantemente como drivers, protocolos

de red, …) pequeñas mejoras pueden suponer un aumento del rendimiento importante a lo largo del día.

Por lo tanto, la principal motivación de este proyecto es la carencia de una herramienta que ofreciera algo similar, a la vez que creíamos que podría ayudar a los programadores a mejorar su producto sin que tuvieran que hacer un estudio de los flags para cada programa que realizaran.

1.2 Objetivos.

El objetivo principal de este TFG consistirá en el desarrollo de una aplicación que, para un programa determinado escrito en C o C++, sea capaz de proporcionarle al programador el conjunto de flags u opciones de compilado que mejor prioricen los objetivos del programador (tamaño o eficiencia), dentro de un tiempo razonable.

Hay que tener en cuenta que este desarrollo no es trivial, ya que como se comentó previamente, los compiladores recientes disponen de centenares de opciones y probar el efecto en un programa, supone compilarlo con la opción oportuna y posteriormente ejecutarlo. Esto es un tiempo muy considerable que impide probar todas las posibles opciones de forma enumerativa y exhaustivamente.

Para cumplir este objetivo el proyecto se ha dividido en varias fases, empezando por un estudio previo de los flags de compilación de GCC. Con ello buscábamos incrementar la calidad de los algoritmos posteriores mediante la lectura y enlazado de términos que aparecieran en el programa vinculados a flags que ya sabíamos que se comportaban bien en dicha situación.

Habiendo realizado el estudio, nuestro siguiente paso fue realizar un algoritmo que clasifique los flags en función de si mejoran o no el rendimiento del proceso base (sin ningún flag). Contando con dicha información, quedaría implementar uno o más algoritmos que la trataran y devolvieran la combinación más óptima. Todo esto, implementado en una aplicación modular de escritorio multiplataforma que dispone de cliente GUI (Graphic User Interface) y CLI (Command Line Interface).

En todo momento seguiremos una metodología ágil, concretamente Scrum, para tener nuevas funcionalidades al final de cada una de las cinco iteraciones y un seguimiento adecuado del proyecto a través de las fechas de entrega propuestas.

1.3 Estructura de la memoria.

En este apartado expondremos de forma breve la organización de este documento, dividido en los siguientes puntos:

- **Capítulo 1:** Introducción

  En este capítulo presentaremos las motivaciones y objetivos de nuestro proyecto junto a un breve resumen de este documento.

- **Capítulo 2:** Tecnología y Recursos

  Este segundo capítulo describirá las tecnologías y recursos requeridos para la realización de este proyecto.

- **Capítulo 3:** Análisis y Diseño

  Aquí comentaremos los requisitos y las decisiones de diseño de este proyecto software, junto a los diagramas pertinentes.

- **Capítulo 4:** Implementación

  En esta parte del documento se mostraran las distintas iteraciones que componen el desarrollo de la aplicación.

- **Capítulo 5:** Pruebas

  En este capítulo hablaremos de las pruebas desarrolladas para el software y del estudio que hemos realizado respecto a los distintos algoritmos.

- **Capítulo 6:** Conclusiones y Líneas Futuras

  Por último comentaremos las conclusiones a las que hemos llegado a lo largo del proyecto y posibles continuaciones.

2

Tecnología y Recursos

2.1 Herramientas para el desarrollo.

En este apartado, expondremos todas las herramientas que hemos utilizado para poder desarrollar la aplicación.

2.1.1 Java

Java es un lenguaje de programación orientado a objetos muy popular actualmente , de fácil uso y que tiene una gran comunidad y es muy versátil, pudiendo realizar desde aplicaciones móviles hasta aplicaciones webs pasando por aplicaciones de escritorio.

Decidimos hacer uso de este lenguaje porque teníamos la facilidad de haber trabajado con el con anterioridad, y por tanto conocíamos sus ventajas y desventajas. Uno de los puntos que tuvimos en cuenta fue la portabilidad del lenguaje, que es completamente independiente del sistema operativo y sabíamos que nuestra aplicación sería capaz de ejecutarse en muchas plataformas.

Además, decidimos hacer uso de "IntelliJ IDEA", un entorno de desarrollo integrado (o IDE) para el lenguaje java, del que hablaremos más adelante en este apartado.

2.1.2 IntelliJ IDEA

"IntelliJ IDEA" es un IDE desarrollado por "Jetbrains" que al igual que muchos IDEs, ofrece toda clase de funcionalidades. Lo que nos hizo decantarnos por este IDE frente a otros como pueden ser "NetBeans" o "Eclipse" fue su herramienta de desarrollo de interfaces gráficas.

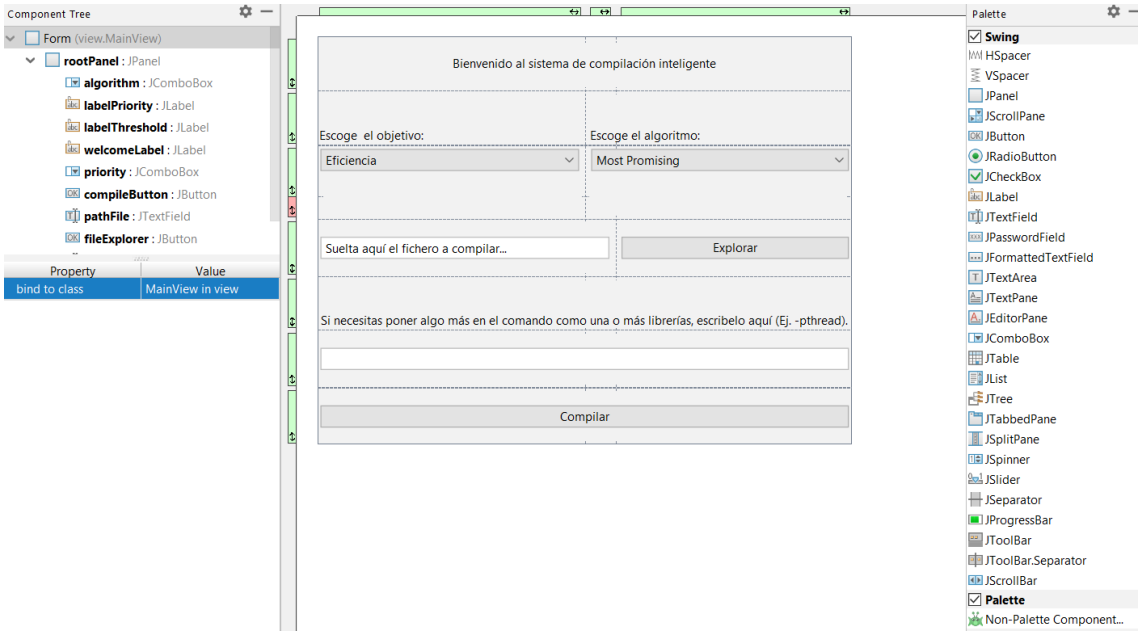

Ilustración 1: Herramienta de desarrollo de interfaces de IntelliJ IDEA.

Ya lo habíamos podido usar en el pasado al utilizar "Android Studio", IDE basado en IntelliJ IDEA y que ofrece una herramienta de desarrollo de interfaces muy similar, solo que enfocada a dispositivos móviles.

2.1.3 C y C++

C y C++ son 2 lenguajes de programación muy utilizados, debido a su simplicidad y eficiencia sin nombrar que en caso del lenguaje C, se usa con mucha frecuencia (como podemos comprobar en la siguiente ilustración que corresponde a la tabla de tiobe) sobre todo en sistemas empotrados, sistemas críticos y de tiempo real, donde es muy importante el tiempo de ejecución y el tamaño.

Sep 2019	Sep 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.661%	-0.78%
2	2		C	15.205%	-0.24%
3	3		Python	9.874%	+2.22%
4	4		C++	5.635%	-1.76%
5	6	▲	C#	3.399%	+0.10%
6	5	▼	Visual Basic .NET	3.291%	-2.02%
7	8	▲	JavaScript	2.128%	-0.00%
8	9	▲	SQL	1.944%	-0.12%
9	7	▼	PHP	1.863%	-0.91%
10	10		Objective-C	1.840%	+0.33%

Ilustración 2: Tabla Tiobe.

Nos parecieron grandes candidatos a realizar un proyecto cuyo objetivo fuera mejorar los ejecutables que se generaban para estos dos lenguajes. Además, nos decantamos por el compilador GCC por su popularidad, pero es sistema es fácilmente adaptable a cualquier otro.

Dichos lenguajes además los hemos utilizado para poder desarrollar pruebas que veremos a lo largo del capítulo 5.

2.3 Herramientas para la gestión.

En este apartado vamos a pasar a exponer todas las herramientas que hemos necesitado para gestionar el proyecto a lo largo de las iteraciones.

2.2.1 Git y GitHub

Necesitábamos algún sistema de control sobre nuestra aplicación, que nos permitiera tener almacenada a la misma, en caso de tener algún problema con los equipos o bien una manera de restituir la aplicación a un punto anterior si fuera necesario.

Por ello, nos decantamos por Git, una herramienta con la cual ya habíamos trabajado que permite guardar una copia de nuestro proyecto , restaurar una versión anterior si fuera necesario o desarrollar funcionalidades en paralelo sin que haya conflictos, entre otras.

Concretamente hemos utilizado la plataforma Github, siendo este el repositorio de la misma:

https://github.com/MykexMP/TFG

2.2.2 Trello

Habíamos decidido utilizar una metodología de carácter ágil, lo cual entre otras cosas, implicaba realizar entregas periódicas del producto a lo largo de 5 iteraciones. Este tipo de metodología son muy útiles cuando se prevé que el proyecto tenga un gran volumen de cambios o para proyectos que no están completamente definidos desde un comienzo.

En nuestro caso nos decantamos por Scrum con un periodo de 2 semanas por iteración, tras lo cual nos reuníamos para comprobar si se habían cumplido los objetivos propuestos y proponer nuevos objetivos para la siguiente iteración.

En nuestro caso, nos parecía que "Trello" cumplía muy bien dicha función, un gestor de tareas en formato de "tablero de corcho" donde organizar las tareas. En la ilustración podemos observar como quedó el tablero tras finalizar las 5 iteraciones:

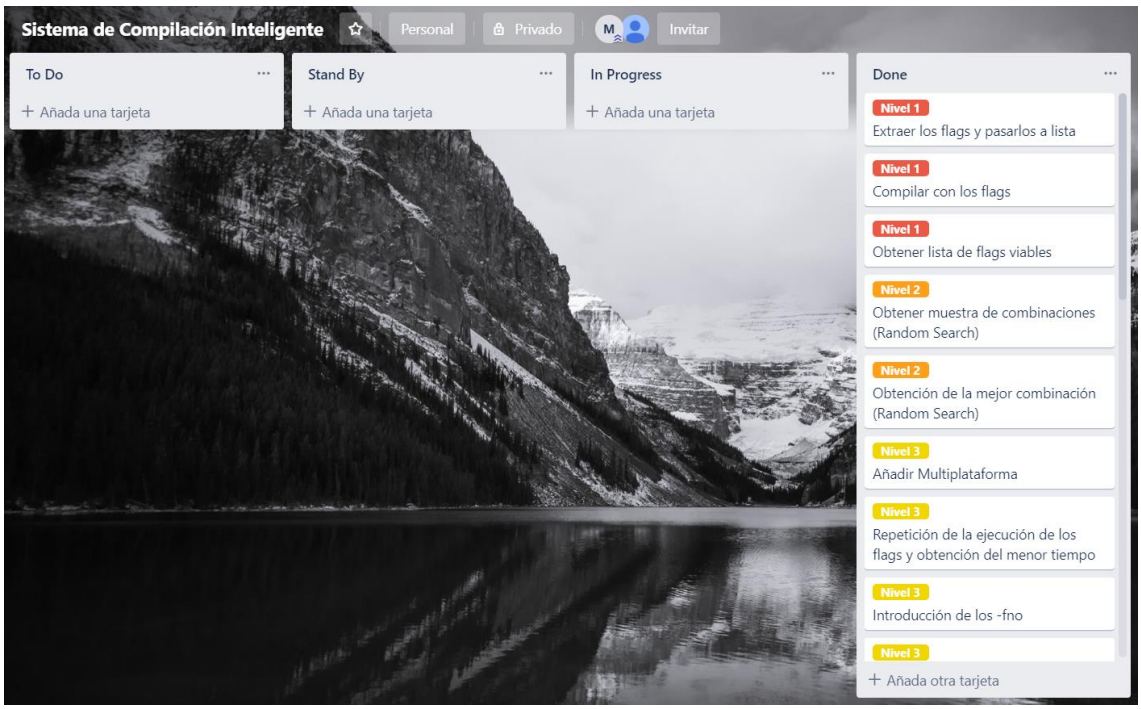

Ilustración 3: Tablero Scrum tras finalizar el proyecto.

En nuestro caso nos decantamos por el ya conocido "To do, In progress and Done" además de añadir una columna extra, "Stand by", donde pusimos tareas que en general, requerían de terminar otras para poder continuarse pero que previamente habíamos empezado.

2.3 Herramientas para la documentación.

En este apartado queda englobado todo lo referente a la realización de esta memoria.

2.3.1 Magic Draw

Nos dimos cuenta de que también nos sería útil una herramienta para poder plasmar los distintos diagramas de la aplicación. Hemos hecho uso de varias herramientas de modelado con anterioridad, como son "Use" , "Papyrus" y "Magic Draw".

De todas estas , la herramienta que más nos convenció fue "Magic Draw", dado que era la que mejor dominábamos y nos gustaba el acabado que le daba a los diagramas y la facilidad de su uso de la misma.

2.3.2 Microsoft Word

Herramienta que hemos utilizado para redactar esta memoria junto a otros recursos utilizados en la misma, como las tablas de casos de uso.

Esta herramienta nos lleva siendo familiar muchísimo tiempo y, aprovechando la licencia que poseemos como estudiantes y algunos conocimientos que presentábamos frente a otros editores de texto que partíamos del desconocimiento, nos acabamos decantando por ella.

3

Análisis y Diseño

3.1 Requisitos funcionales

En este apartado incluiremos los requisitos funcionales, es decir, aquellas funcionalidades que debe ofrecer el programa al usuario:

ID	NOMBRE	DESCRIPCIÓN
RF01	Buscar Fichero	El sistema debe permitir al usuario poder, desde un explorador de archivos, seleccionar el programa que desea compilar.
RF02	Elegir Algoritmo	El sistema debe permitir al usuario poder escoger que algoritmo quiere utilizar para la búsqueda.
RF03	Elegir Objetivo	El sistema debe permitir al usuario poder escoger cual quiere que sea el objetivo.
RF04	Agregar Librerías	El sistema debe permitir al usuario poder introducir las opciones activables necesarias para permitir la compilación de un programa que contenta dependencias externas.
RF05	Obtener Ejecutable con Menor Tamaño	El sistema debe entregar al usuario un ejecutable que corresponda con el código ofrecido como entrada y que pese lo menos posible en las iteraciones establecidas.
RF06	Obtener Ejecutable con Menor tiempo	El sistema debe entregar al usuario un ejecutable que corresponda con el código ofrecido como entrada y que tarde lo menos posible en las iteraciones establecidas.

Tabla 1: Requisitos funcionales

3.2 Requisitos no funcionales

En este apartado incluiremos los requisitos no funcionales, es decir, aquellos requisitos que no son una funcionalidad que el usuario vaya a utilizar de manera directa:

ID	NOMBRE	DESCRIPCIÓN
RNF01	Extensibilidad	El sistema debe estar diseñado para facilitar futuras extensiones del mismo, bien sea por nuevos algoritmos de optimización, otros compiladores u otros objetivos de compilación.
RNF02	Tolerancia a fallos	El sistema debe estar preparado para, en caso de tener algún fallo, poder recuperarse del mismo. En concreto, debe poder ser capaz de solventar cuando falla la ejecución o compilación de un programa.
RNF03	Facilidad de Aprendizaje	El sistema debe ser intuitivo y de fácil uso.
RNF04	Portabilidad	El sistema deberá estar disponible para Windows y otros Sistemas Operativos basados en Linux.

Tabla 2: Requisitos no funcionales.

3.3 Diagrama de Casos de Uso

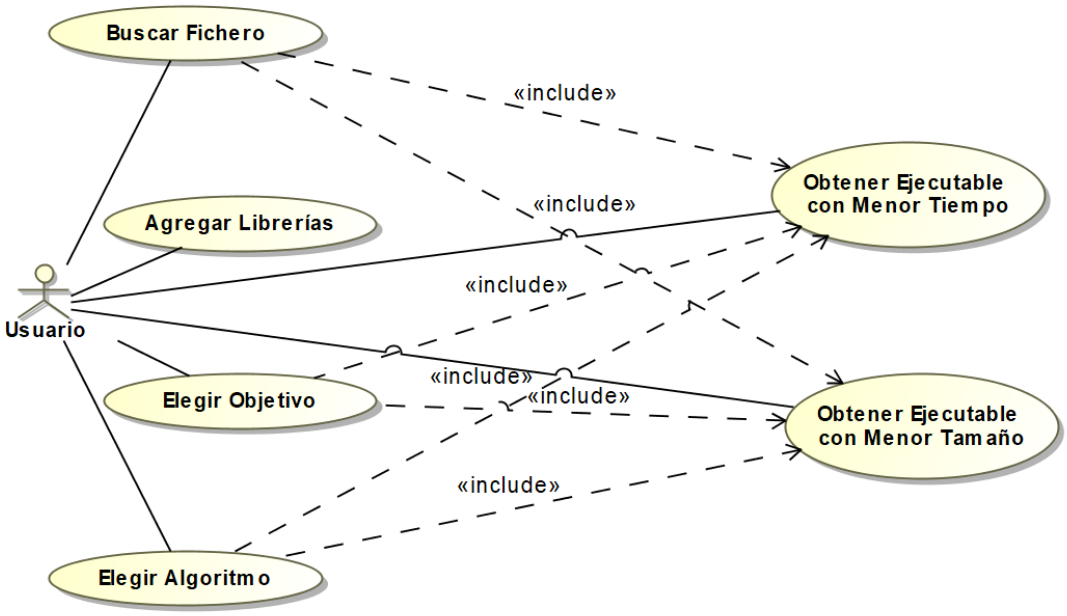

Ilustración 4: Diagrama de Casos de Uso

- Buscar Fichero:

NOMBRE	Buscar Fichero
IDENTIFICADOR	CU01
PRECONDICIÓN	Encontrarse en la pantalla principal.
POSTCONDICIÓN	Encontrarse en la pantalla principal con el fichero seleccionado.
DESCRIPCIÓN	El usuario utiliza el botón de explorar para buscar el fichero a compilar.
DESARROLLO DEL CASO	1 - El usuario hace click en explorar. 2.1 - El usuario encuentra el fichero y se cierra la ventana. 2.2 - El usuario no encuentra el fichero y cierra la ventana.

Tabla 3: Caso de Uso CU01

- Elegir Algoritmo:

NOMBRE	Elegir Algoritmo
IDENTIFICADOR	CU02
PRECONDICIÓN	Encontrarse en la pantalla principal.
POSTCONDICIÓN	Encontrarse en la pantalla principal con el algoritmo seleccionado.
DESCRIPCIÓN	El usuario utiliza el desplegable para seleccionar el algoritmo de compilación.
DESARROLLO DEL CASO	1.1 - El usuario hace click en el desplegable. 2.1 - El usuario selecciona el algoritmo que prefiera. 1.2 - El usuario no hace click en el desplegable y se selecciona la opción por defecto.

Tabla 4: Caso de uso CU02.

- Elegir Objetivo:

NOMBRE	Elegir Objetivo
IDENTIFICADOR	CU03
PRECONDICIÓN	Encontrarse en la pantalla principal.
POSTCONDICIÓN	Encontrarse en la pantalla principal con el objetivo seleccionado.
DESCRIPCIÓN	El usuario utiliza el desplegable para seleccionar el objetivo de la compilación.
DESARROLLO DEL CASO	1.1 - El usuario hace click en el desplegable. 2.1 - El usuario selecciona el objetivo que prefiera. 1.2 - El usuario no hace click en el desplegable y se selecciona el objetivo por defecto.

Tabla 5: Caso de uso CU03.

- Agregar Librerías:

NOMBRE	Agregar Librerías
IDENTIFICADOR	CU04
PRECONDICIÓN	Encontrarse en la pantalla principal.
POSTCONDICIÓN	Encontrarse en la pantalla principal con el campo de librerías rellenado.
DESCRIPCIÓN	El usuario rellena el campo de librerías que necesite su programa para compilar.
DESARROLLO DEL CASO	1.1 - El usuario rellena el campo con las librerías. 1.2 - El usuario no rellena el campo con librerías, por lo que se toma el valor por defecto.

Tabla 6: Caso de uso CU04.

- Obtener Ejecutable con Menor Tiempo:

NOMBRE	Obtener Ejecutable con Menor Tiempo.
IDENTIFICADOR	CU05
PRECONDICIÓN	Encontrarse en la pantalla principal y haber realizado los anteriores casos de uso.
POSTCONDICIÓN	Encontrarse en la pantalla principal y el fichero en el directorio correspondiente.
DESCRIPCIÓN	El usuario pulsa en el botón de compilar y obtiene un fichero ejecutable que corresponde al ofrecido por él y que mejora el tiempo de ejecución.
DESARROLLO DEL CASO	1 - El usuario hace click en compilar. 2.1 - El usuario espera a la finalización del programa. 3.1 - El usuario obtiene su ejecutable optimizado. 2.2 - Sucede un error durante el compilado y saltamos al paso 1.

Tabla 7: Caso de uso CU05.

- Obtener Ejecutable con Menor Tamaño:

NOMBRE	Obtener Ejecutable con Menor Tamaño.
IDENTIFICADOR	CU06
PRECONDICIÓN	Encontrarse en la pantalla principal y haber realizado los anteriores casos de uso.
POSTCONDICIÓN	Encontrarse en la pantalla principal y el fichero en el directorio correspondiente.
DESCRIPCIÓN	El usuario pulsa en el botón de compilar y obtiene un fichero ejecutable que corresponde al ofrecido por él y que mejora el tamaño del mismo.
DESARROLLO DEL CASO	1 - El usuario hace click en compilar. 2.1 - El usuario espera a la finalización del programa. 3.1 - El usuario obtiene su ejecutable optimizado. 2.2 - Sucede un error durante el compilado y saltamos al paso 1.

Tabla 8: Caso de uso CU06.

3.4 Diagrama de Clase

A continuación, vamos a exponer el diagrama de clase de la aplicación y a explicar las funciones de las que consta. En este diseño hemos queridos seguir patrones de diseño y buenas prácticas vistas en Ingeniería del Software, como el patrón "Factory" o "Fábrica":

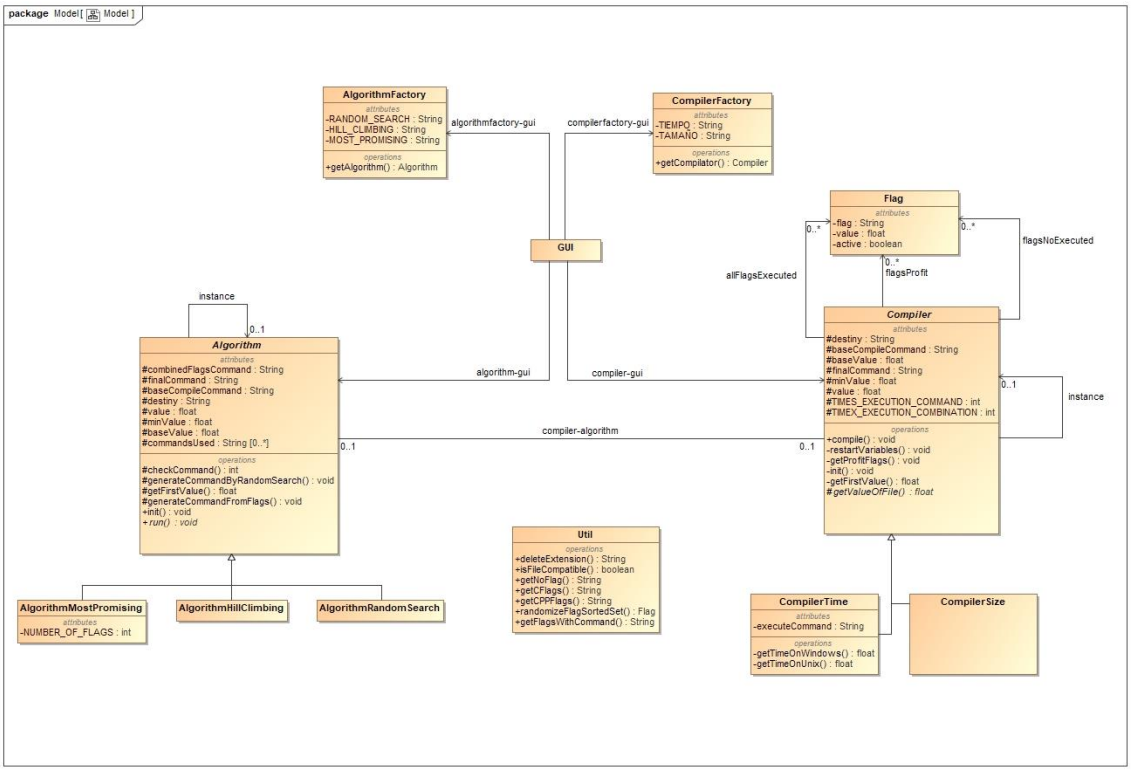

Ilustración 5: Diagrama de Clase

- **GUI**: Es una clase que representa la interfaz con la que interactuará el usuario en modo gráfico.

- **Util**: Esta clase engloba todas aquellas funciones que, no hemos visto apropiadas incluir en ninguna de las otras clases pero requeríamos de poder llamarlas para evitar una repetición del código. Sus métodos son:

MÉTODO	DESCRIPCIÓN
DeleteExtension	Elimina la extensión de un fichero dado.
IsFileCompatible	Devuelve true si el fichero es un fichero C o C++.
GetNoFlag	Devuelve el flag negado de un flag dado.

GetCFlags	Devuelve una lista con todos los flags de compilación de C.
GetCPPFlags	Devuelve una lista con todos los flags de compilación de C++.
RandomizeFlagSorted	Mezcla una colección de flags.
GetFlagsWithCommand	Devuelve una lista de flags en función del comando de entrada.

Tabla 9: Métodos de la clase Util.

- **Flag:** Una clase utilizada para darle forma de objeto al concepto de flag. Sus atributos son:

ATRIBUTO	DESCRIPCIÓN
Flag	El flag en sí.
Value	El valor asociado en la ejecución a este flag en solitario.
Active	Si se encuentra o no activo el flag actualmente.

Tabla 10: Atributos de la clase Util.

- Algorithm: Una clase abstracta de la que heredan todas las clases que usan un algoritmo para encontrar la mejor combinación de opciones activables. Gracias a ella, la extensión de la funcionalidad del programa es bastante más sencilla.

MÉTODO	DESCRIPCIÓN
CheckCommand	Encargada de probar un flag y comprobar si da error.
GenerateCommandByRandomSearch	Genera un comando aleatorio que se usa como base a algunos algoritmos.
GetFirstValue	Obtiene el valor del comando base.
GenerateCommandFromFlags	Genera un comando en función de la lista de flags y su valor "active".
Init	Inicializa la clase para su uso.
Run	Método abstracto que implementan los algoritmos que lo heredan. Es el encargado de devolver el mejor comando.

Tabla 11: Métodos de la clase Algorithm.

ATRIBUTO	DESCRIPCIÓN
CombinedFlagsCommand	Un valor auxiliar para probar combinaciones de flags.
FinalCommand	El comando que genera el mejor ejecutable.
BaseCompileCommand	El comando base (sin flags) sobre el que se trabaja.

Destiny	La ruta del fichero de salida.
Value	Un valor auxiliar para comprarar el rendimiento de cada combinación.
MinValue	El valor del mejor ejecutable.
BaseValue	El valor del ejecutable del comando base.
CommandsUsed	Una lista de comandos ya usados para no malgastar iteraciones.

Tabla 12: Atributos de la clase Algorithm.

La clases "*AlgorithmHillClimbing*", "*AlgorithmRandomSearch*" y "*AlgorithMostPromising*" solo implementan el método run, a excepción de esta última que tambien tiene un valor "*NUMBER_OF_FLAGS"* , necesario para su método run.

- Compiler: Clase principal donde se realiza la selección de flags y las llamadas a las funciones de "*Algorithm*". Además, es la clase de la que se hereda para darle un nuevo objetivo a la compilación. Comparte atributos que cumplen la misma función que en "*Algorithm*", a excepción de:

ATRIBUTOS	DESCRIPCIÓN
TIMES_EXECUTION_COMMAND	El número de veces que se prueba un comando.
TIMES_EXECUTION_COMBINATION	EL número de iteraciones de los algoritmos.
AllFlagsExecuted	Una lista con todos los flags que se han podido ejecutar individualmente.
FlagsProfit	Una lista de los flags que mejoran el rendimiento respecto al comando base.
FlagsNoExecuted	Una lista de los flags que no se han podido ejecutar.

Tabla 13: Atributos de la clase Compiler.

En el caso de las funciones, comparten "*Init*" y "*GetFirstValue*":

MÉTODO	DESCRIPCIÓN
Compile	Es el método principal a la que se llama, desde ahí se llaman al resto de métodos.
RestartVariables	Devuelve las variables al estado anterior a la ejecución.
GetProfitFlags	Obtiene una lista de los flags que mejoran el rendimiento (individual) del sistema.
GetValueOfFile	Este método es abstracto y es el que deben implementar todos los "objetivos". Devuelve el valor que le queramos asignar a un fichero.

Tabla 14: Métodos de la clase Compiler

"*CompilerSize*" y "*CompilerTime*" son 2 ejemplos de cómo se debe usar el método "*getValueOfFile*". Este último utiliza 2 funciones, dependiendo del sistema operativo (algo que comentaremos en mayor profundidad en el apartado 4) y un valor auxiliar necesario para lo mismo , "*executeCommand*".

Las 2 últimas clases que quedan , "*AlgorithmFactory*" y "*CompilerFactory*", tan solo aplican el patrón de diseño "*Factory*", respectivamente.

3.5 Diagrama de Secuencia

En este apartado vamos a mostrar el diagrama de secuencia del caso de uso CU05 y CU06, que son idénticos (lo único que varía es que se devuelven en el mensaje 4), obtener ejecutable:

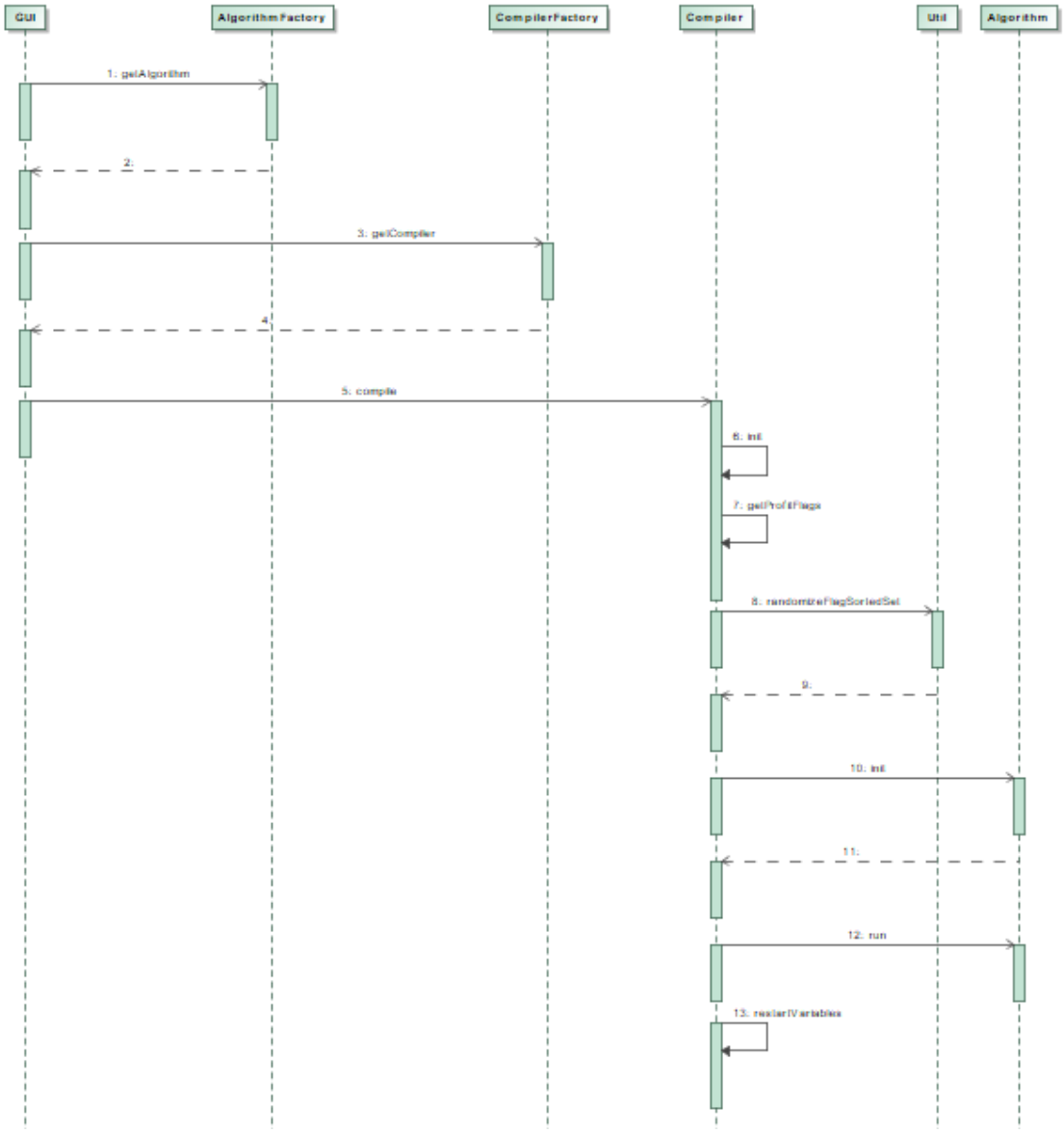

Ilustración 6: Diagrama de Secuencia de CU03

No vamos a mostrar el resto debido a su sencillez y que no tienen llamadas anidadas a métodos.

3.6 Diagrama de Navegación

No nos parecía lógico introducir un diagrama de navegación, pues en la práctica, solo visitaremos una o ninguna vista, dependiendo si usamos la interfaz CLI o GUI.

4

Desarrollo e Implementación

4.1 Desarrollo de la Interfaz de Usuario

La interfaz gráfica que hemos diseñado buscaba ser minimalista y que no abrumara al usuario al iniciar la aplicación.

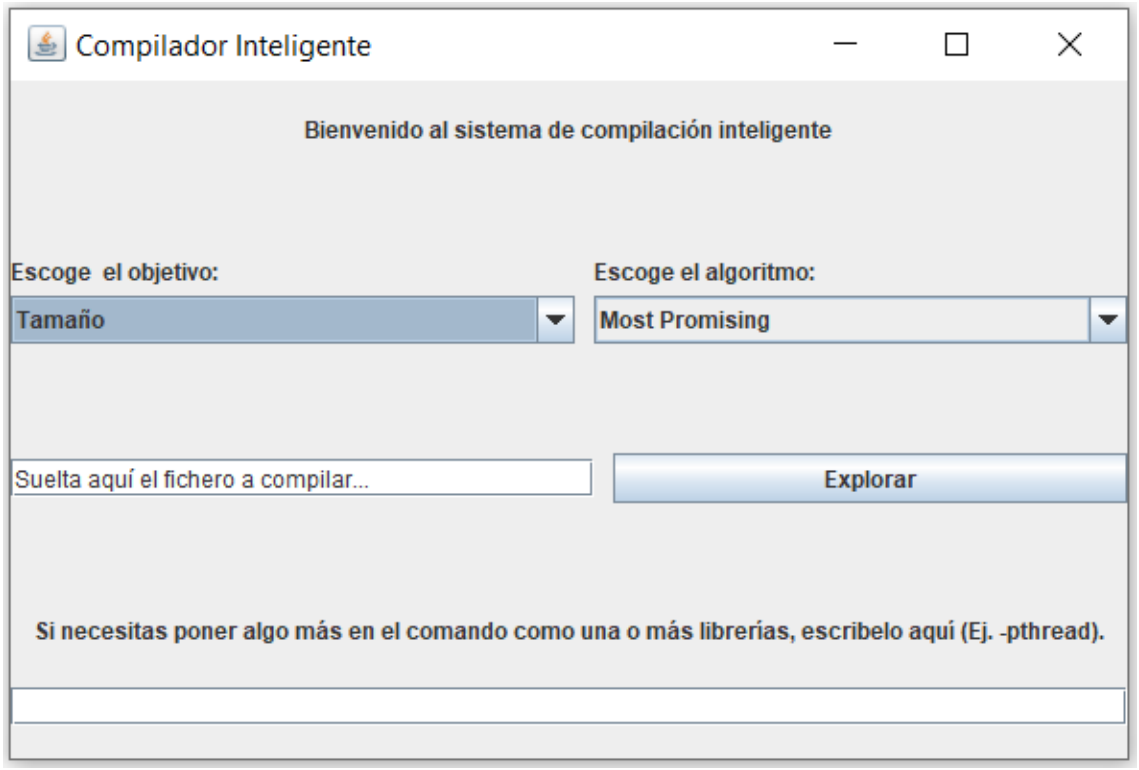

Ilustración 7: Interfaz Gráfica 1.

Como podemos observar, la interfaz nos muestra los parámetros que podemos modificar , un explorador de archivos que también permite "soltar" el fichero en lugar de buscarlo y una barra para poder , en caso de necesitarlo, escribir los

flags necesarios para librerías utilizadas en la aplicación, por ejemplo "-pthread" .

Cuando hayamos encontrado el fichero mediante el explorador o bien soltándolo en el hueco correspondiente, la interfaz descubrirá el botón de compilar, evitando así que el usuario pueda pulsarlo hasta que no se haya completado el rellenado de los parámetros mínimos, quedando de la siguiente forma:

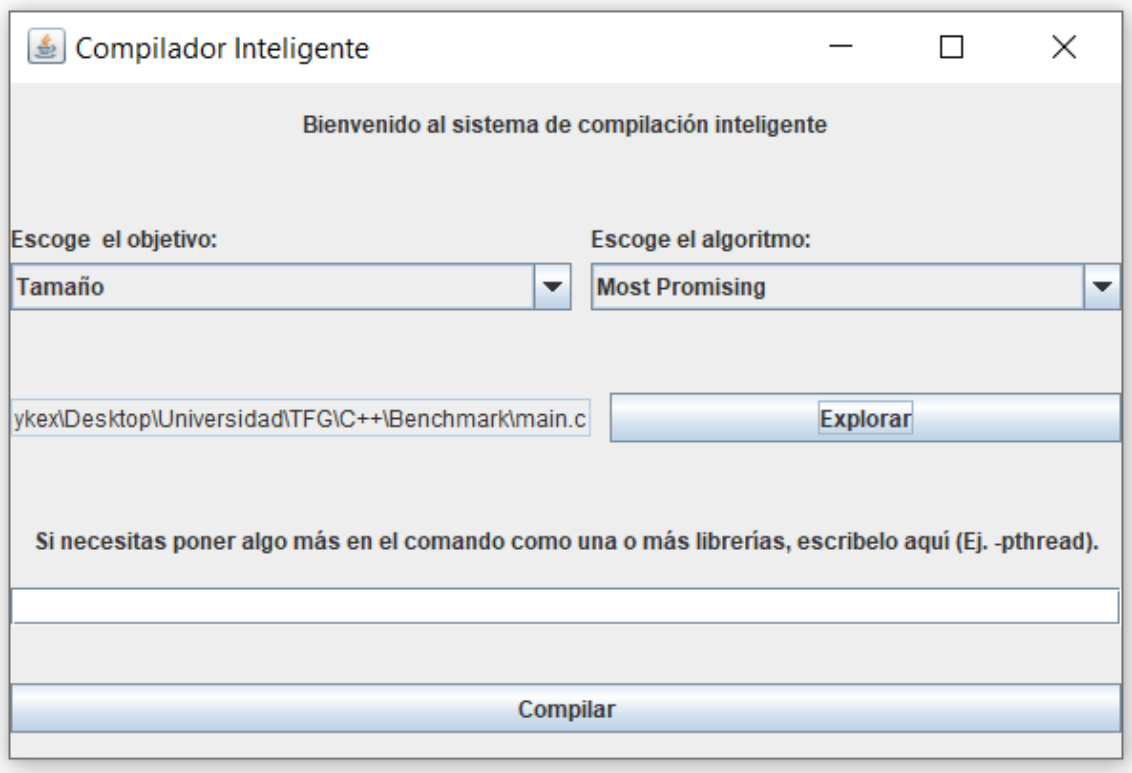

Ilustración 8: Interfaz Gráfica 2

Cada uno de los desplegables nos permite escoger entre 3 algoritmos y 2 objetivos de compilación, de los cuales hablaremos con más detalle en el apartado 4.2.

Esta interfaz está desarrollada con la herramienta de desarrollo de interfaces de “Intellij IDEA”, que es la misma que utiliza Android Studio por estar basado en la misma, y que al haberla usado con anterioridad y habernos obtenido resultados muy satisfactorios, lo consideramos adecuado para su utilización en este proyecto.

También cabe destacar la existencia de una interfaz CLI o Command Line Interface, que nos permite utilizar la aplicación desde la línea de comandos. Este tipo de interfaz es para un tipo de usuario más avanzados ya que es un poco más compleja la interacción, pero ofrece una características muy interesantes, el ser capaz de lanzar múltiples pruebas (por ejemplo, con diferentes opciones para compilar el mismo programa, o para ejecutarlo con diferentes problemas) de manera desatendida (que es muy útil para grandes programas y proyectos).

4.2 Desarrollo del motor de optimización

En este apartado vamos a explicar en detalle el funcionamiento interno de la aplicación y de cada una de las clases que no componen la interfaz.

A rasgos generales, el sistema ha sido pensado de forma modular para facilitar su extensión. Del mismo modo se han utilizado patrones que encajaban con la aplicación e intentaban optimizar al máximo a la misma.

Empecemos con, lo que es denominado como GUI en el diagrama. Esta clase cumple varias funciones importantes para el programa, además de servir como interfaz. Es la encargada de cargar los flags de C y C++ (solo una vez para evitar peso innecesario y haciendo una llamada a la clase "Util") y de seleccionar el algoritmo y el objetivo en función de la entrada recibida, como podemos ver en la siguiente ilustración:

```
/**
 *
 * @param command El comando que decide si los flags son de c o C++.
 * @return Una lista de los flags de C o C++, dependiendo de command.
 */
private static List<String> getFlagsWithCommand(String command){
    List<String> flags = new ArrayList<>();

    try {
        Scanner s = new Scanner(Runtime.getRuntime().exec(command).getInputStream());
        while(s.hasNext()){
            String token = s.next();
            if(!token.contains("=") && token.length()>1 && token.substring(0,2).equals("-f")) {flags.add(token);}
        }
    } catch (IOException e) {}

    return flags;
}
```

Ilustración 9: Obtención de Flags

En esta funcionalidad es obligatorio el uso del método "exec" de Runtime, pues necesitamos poder ejecutar comandos en la terminal que nos devuelvan los conjuntos correspondientes a los flags de C y C++, excluyendo aquellos que requieran de parámetros.

Habiendo obtenido dichos flags, pasaremos a llamar al método "*compile*" y comenzamos el proceso de prueba, donde inicializamos la clase "*Compiler*" y buscamos aquellos flags que mejoren el resultado respecto al comando base.

```java
/**
 * @param flags Es el conjunto de flags que exploraremos para obtener los que den una mejor eficiencia.
 */
private void getProfitFlags(List<String> flags) {
    try{
        System.out.println("El comando base tarda " + baseValue);

        for (String flag : flags) {
            if(Runtime.getRuntime().exec( command: baseCompileCommand + " " + flag).waitFor()==0) {
                value = getValueOfFile();
                System.out.println("El flag '" + flag + "' hace que tarde " + value + " Milisegundos");
                allFlagsExecuted.add(new Flag(flag,value));
                if(value < baseValue) flagsProfit.add(new Flag(flag,value));
                if(value < minValue ) {minValue = value; finalCommand = baseCompileCommand + " " + flag;}
                value = Float.MAX_VALUE;
            }
            else flagsNoExecuted.add(new Flag(flag));
        }
    }
    catch (Exception e) { System.out.println("No se ha podido ejecutar el comando"); }
}
```

Ilustración 10: Obtención de flags de mejora

Este método nos permite obtener aquellos flags que mejoran nuestro valor base (un comando que ejecutamos sin ningún tipo de flag activo), aquellos que no se hayan podido ejecutar y todos los que se hayan podido ejecutar, independientemente de si mejoran o no respecto del valor base. Esto último es debido a que, en un principio, pensamos que sería posible que aunque los flags no mejoraran, al aplicar una segunda fase pudieran hacerlo debido a dependencias entre ellos que impidieran que quedándonos solo con lo que mejoraran, no pudiéramos obtener. Sin embargo, tras nuestras pruebas, hemos visto que realmente, no es algo a tener en cuenta, pues no había mejora al usar todos los flags.

En este método también vemos por primera vez el método "getValueOfFile", que será así en el caso de querer medir el tamaño:

```java
/**
 *
 * @return Devuelve el tamaño del fichero.
 */
@Override
public float getValueOfFile(){return new File(destiny).length();}
```

Ilustración 11: Obtención del Tamaño

O bien así, en caso de ser el tiempo:

```java
/**
 *
 * @return Devuelve el tiempo de ejecución de el fuchero en cuestión.
 */
@Override
public float getValueOfFile(){
    executeCommand = System.getProperty("os.name").startsWith("Windows") ? "powershell.exe Measure-Command{" +
            destiny + "}" : "/usr/bin/time -f \"%e\" " + destiny;
    return System.getProperty("os.name").startsWith("Windows") ? getTimeOnWindows() : getTimeOnUnix();
}
```

Ilustración 12: Obtención del Tiempo.

Este método era totalmente posible hacerla sin utilizar procesos del Shell, sin embargo, conocemos las limitaciones del lenguaje que hemos decidido usar para el proyecto, y sabiendo que no es el más eficiente en cuestión de tiempo y que hay ciertos atributos que no se pueden controlar (como por ejemplo, cuando se activa el recolector de basura), creíamos que podría falsear los tiempos y decidimos utilizar los comando que Windows y Unix ofrecen.

```
private float getTimeOnUnix() {
    Scanner s;
    float result;
    float minResult = Float.MAX_VALUE;
    try {
        for(int i=0;i<TIMES_EXECUTION_COMMAND;i++) {
            s = new Scanner(Runtime.getRuntime().exec(executeCommand).getInputStream());
            String aux = "";
            while(s.hasNext()) aux = s.next();
            result = Float.parseFloat(aux.replace( target: ",",  replacement: "."));
            result *= 1000;
            if (result < minResult) minResult = result;
            s.close();
        }
    }catch (Exception e) {System.out.println("Error calculando el tiempo");}
    return minResult;
}
```

Ilustración 13: Obtención del tiempo en Unix.

```
private float getTimeOnWindows() {
    Scanner s;
    float result;
    float minResult = Float.MAX_VALUE;
    try {
        for(int i=0;i<TIMES_EXECUTION_COMMAND;i++) {
            s = new Scanner(Runtime.getRuntime().exec(executeCommand).getInputStream());
            while (s.hasNext()) {
                if (s.next().equals("TotalMilliseconds")) {
                    s.next();
                    result = Float.parseFloat(s.next().replace( target: ",",  replacement: "."));
                    if (result < minResult) minResult = result;
                    break;
                }
            }
            s.close();
        }
    }catch (Exception e) {System.out.println("Error calculando el tiempo");}
    return minResult;
}
```

Ilustración 14: Obtención del tiempo en Windows.

Por otro lado, en las funciones de tiempo "getTimeOnWindows" y "getTimeOnUnix" que mostramos arriba, podemos ver que se prueba el comando varias veces (según la constante "TIMES_EXECUTION_COMMAND" actualmente inicializada a tres) en cada una. Como sabemos, los Sistemas Operativos tienen tareas que el usuario no ve y mucho menos controla, y por tanto puede coincidir que la ejecución de alguno de estos comandos coincida con alguna tarea del sistema. Por ello, al ejecutar varias veces el comando y quedarnos con el tiempo menor, podemos aproximar el resultado al que sería el más real. El valor de la constante “TIMES_EXECUTION_COMMAND” afecta mucho al rendimiento de nuestra aplicación, pero la utilización de un valor inferior

a tres consideramos que podría falsear los resultados debido a interferencias externas.

Ambas funciones devuelven el tiempo en milisegundos, pues creemos que segundos, dependiendo del programa, puede ser poco exacto, y en el límite inferior, si bien contemplamos la posibilidad de utilizar nanosegundos, finalmente nos decantamos por utilizar milisegundos cogiendo las cifras decimales.

Tenemos todos los comandos que mejoran el tiempo, ahora debemos de encontrar aquella combinación que sea la óptima, o al menos la mejor dentro de las iteraciones previstas.

Dado que todos los algoritmos utiliza el método run, vamos a echarle un vistazo a su documentación:

```
/**
 *
 * @param compiler El compilador que da el objetivo de compilación.
 * @param entrySet El conjunto de flags de entrada, los que mejoran.
 * @param numberIterations El número de iteraciones, dado por el usuario.
 * @param baseCompileCommand El comando base de compilación.
 */
public abstract void run(Compiler compiler, Set<Flag> entrySet, int numberIterations, String baseCompileCommand);
```

Ilustración 15: Método abstracto run.

En el sistema contamos con tres algoritmos distintos que podemos utilizar para este motivo, de los cuales pasaremos a hablar a continuación. Antes de explicarlos, comentar que se han elegido esos tres algoritmos por su simplicidad y que representan diferentes características a modo de ejemplo, pero el nuestro permite incorporar otro tipo de algoritmos de manera muy sencilla.

El primero de todos es el conocido como "Random Search" o "Búsqueda aleatoria".

```
@Override
public void run(Compiler compiler, Set<Flag> entrySet, int numberIterations, String baseCompileCommand) {
    try {
        for(int i=0;i<numberIterations;i++) {
            generateCommandByRandomSearch(entrySet);
            System.out.println("Número: " + i);
            if(!commandsUsed.contains(combinedFlagsCommand)){
                commandsUsed.add(combinedFlagsCommand);
                System.out.println("El comando a ejecutar es: \n" + combinedFlagsCommand);
                if(checkCommand()==-1) i--;
            }else {System.out.println("Comando ya usado");i--;}
        }
        if(Runtime.getRuntime().exec(finalCommand).waitFor()==0)
        {
            System.out.println("Ejecutable generado con éxito");
        }
    } catch (Exception e) { System.out.println("No se ha podido ejecutar el comando"); }
}
```

Ilustración 16: Método run de “Random Search”.

Este comando se basa en la premisa de generar soluciones probando de manera aleatoria en el espacio de soluciones. Aunque pudiera parecer algo impensable, la búsqueda aleatoria ha dado grandes resultados en las pruebas, de las que hablaremos en el capítulo 5.

El segundo y sorprendentemente, el que peor hemos visto que funcionaba, ha sido el algoritmo "Hill Climbing" o "Escalada Simple".

```
@Override
public void run(Compiler compiler, Set<Flag> entrySet, int numberIterations, String baseCompileCommand) {
    try {
        HashSet<Flag> flagSet = new HashSet<>(entrySet);

        Iterator<Flag> iter = flagSet.iterator();
        Flag selected = null;
        generateCommandByRandomSearch(flagSet);
        checkCommand();

        for(int i=0;i<numberIterations;i++) {
            System.out.println("Número:"+i);
            if(iter.hasNext()) selected = iter.next();
            else {iter = flagSet.iterator(); selected = iter.next();}
            selected.setActive(!selected.isActive());
            generateCommandFromFlags(flagSet);
            if(!commandsUsed.contains(combinedFlagsCommand)){
                commandsUsed.add(combinedFlagsCommand);
                if(checkCommand()==-1) i--;
            }else {System.out.println("Comando ya usado");i--;}
            if (!finalCommand.equals(combinedFlagsCommand)) selected.setActive(!selected.isActive());
        }
        if(Runtime.getRuntime().exec(finalCommand).waitFor()==0) System.out.println("Ejecutable generado con éxito");
    } catch (Exception e) { System.out.println("No se ha podido ejecutar el comando"); }
}
```

Ilustración 17: Método run de “Hill Climbing”.

En etapas tempranas del proyecto, daba grandes resultado , sin embargo, en una de las iteraciones, decidimos que era lógico introducir un contador, que en caso de que una iteración fallara, esta no contara. Esto incremento la mejoría en todos los algoritmos, sin embargo también disparo su tiempo de ejecución, haciendo que este algoritmo quede prácticamente inviable si dispones de los otros 2 implementados.

Por último hablaremos de un algoritmo propio que hemos denominado "Most Promising" o "El más Prometedor":

```
@Override
public void run(Compiler compiler, Set<Flag> entrySet, int numberIterations, String baseCompileCommand) {
    Set<Flag> flagSet = new TreeSet<>(Comparator.comparing(Flag::getValue));
    Iterator<Flag> iter = entrySet.iterator();
    for (int i=0;i<NUMBER_OF_FLAGS;i++){if(iter.hasNext())flagSet.add(iter.next());}

    if(flagSet.size()>0)
    {
        double limit = Math.min(Math.pow(2,flagSet.size()),numberIterations);
        char[] combination;

        for (int j=0;j<limit;j++){
            combination = String.format("%0"+flagSet.size()+"d",Integer.parseInt(Integer.toBinaryString(j))).toCharArray();
            iter = flagSet.iterator();

            System.out.println("Número: " + (j));
            System.out.println("Correspondencia en Binario: " +
                    String.format("%0"+flagSet.size()+"d",Integer.parseInt(Integer.toBinaryString(j))));

            for (int k=0;k<combination.length;k++){
                if(combination[k]=='0') iter.next().setActive(false);
                else iter.next().setActive(true);
            }
            generateCommandFromFlags(flagSet);
            checkCommand();
        }
    }
}
```

Ilustración 18: Método run de “Most Promising”.

En este algoritmo se escogen los X mejores flags, donde X es un valor entero elegido por el usuario, y se realiza un sondeo en profundidad probando todas las combinaciones posibles dentro de dicho conjunto. Nos damos cuenta enseguida del principal problema de este algoritmo, y es que su complejidad es de 2^n . Y si bien es cierto que podría resultar inviable, con entradas pequeñas como por ejemplo 10, ha mostrado grandes mejorías, siendo este el que en término medio, se comporta mejor.

5

Pruebas

Este apartado vamos a exponer unas pruebas que hemos realizado con nuestro programa para comparar , con respecto a otras opciones, si mejora y en qué medida lo hace.

Estás pruebas están efectuadas en un dispositivo cuyo sistema operativo es Windows 10. Utilizaremos tres ficheros C distintos, uno propio muy simple (1) y otros dos más complejos extraídos de benchmarks de C.

En todos los casos, los algoritmos de "Random Search" y "Most Promising" han mostrado un comportamiento similar en cuanto a resultado y tiempo de ejecución del algoritmo. Como comentábamos en el apartado 4, el algoritmo "Hill Climbing" dispara enormemente el tiempo de ejecución.

Los datos de cada tabla han sido calculados hallando la media de cada proceso.

Para el primer fichero, vemos los datos reflejados en la tabla para su tiempo de ejecución:

OPCIÓN	TIEMPO(ms)
Sin flags	59,2592
Most Promising	31,7158
Random Search	38,8611
Hill Climbing	50,8684

Tabla 15: Tiempos de ejecución para el fichero 1.

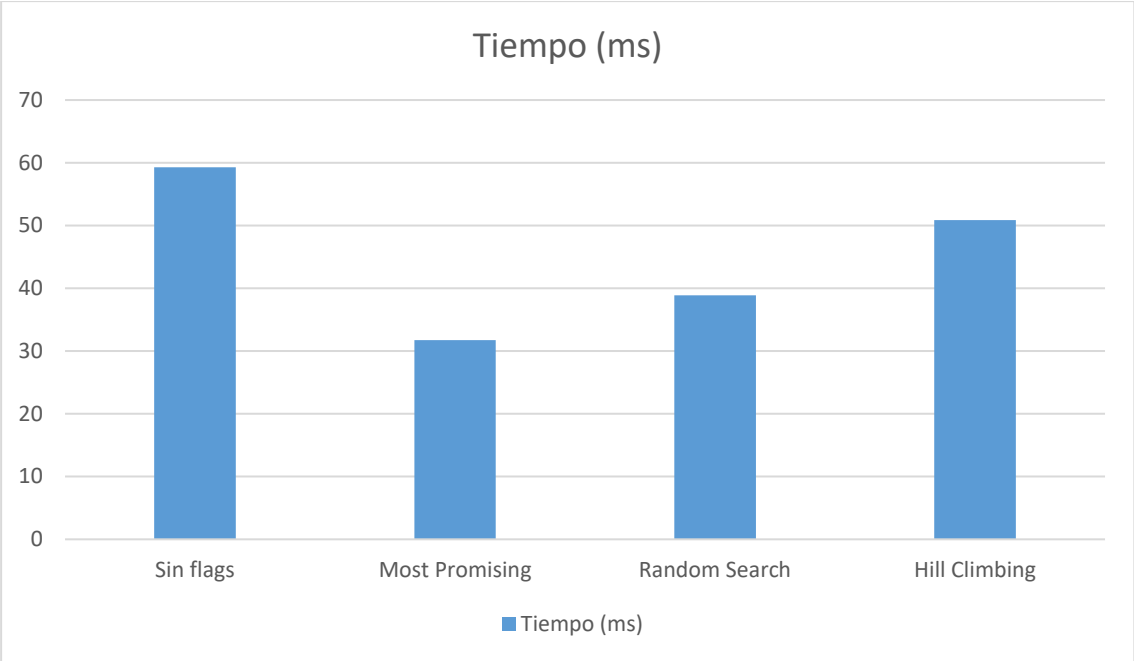

Ilustración 19: Tiempos de ejecución para el fichero 1.

Como podemos observar en la tabla, los datos están muy dispersos, esto es debido a los procesos internos del sistema, que al ser un tiempo tan pequeño, están afectando enormemente al tiempo de ejecución.

Para su tamaño:

OPCIÓN	TAMAÑO(KB)
Sin flags	24
Most Promising	24
Random Search	24
Hill Climbing	24

Tabla 16: Tamaño para el fichero 1.

Podemos observar que, al ser un fichero tan pequeño, no presenta variaciones independientemente de la compilación.

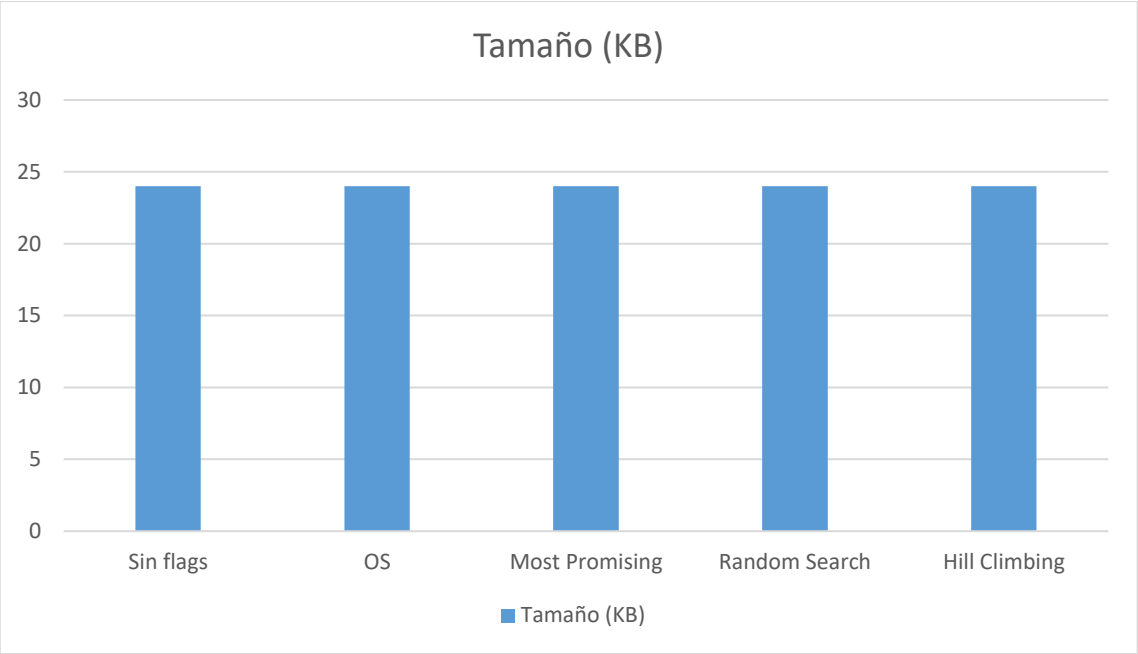

Ilustración 20: Tamaño para el fichero 1.

En el caso del segundo fichero:

OPCIÓN	TIEMPO(ms)
Sin flags	70,002
Most Promising	11,4047
Random Search	17,7482
Hill Climbing	26,253

Tabla 17: Tiempos de ejecución para el fichero 2.

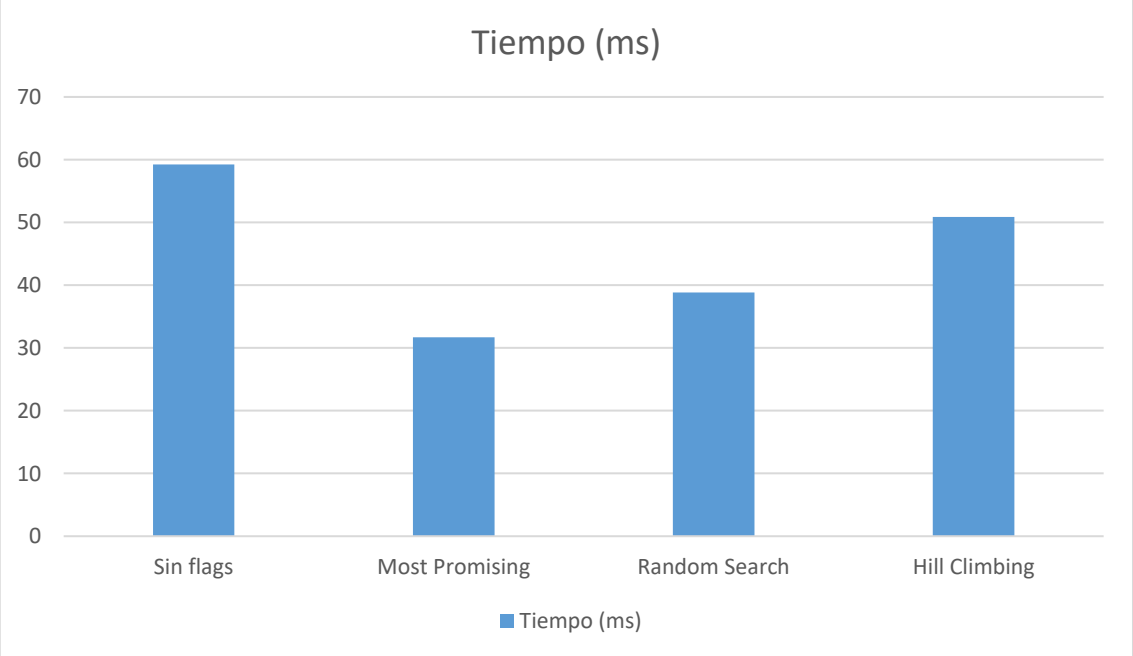

Ilustración 21: Tiempos de ejecución para el fichero 2.

OPCIÓN	TAMAÑO(KB)
Sin flags	26
Os	26
Most Promising	26
Random Search	26
Hill Climbing	26

Tabla 18: Tamaño para el fichero 2.

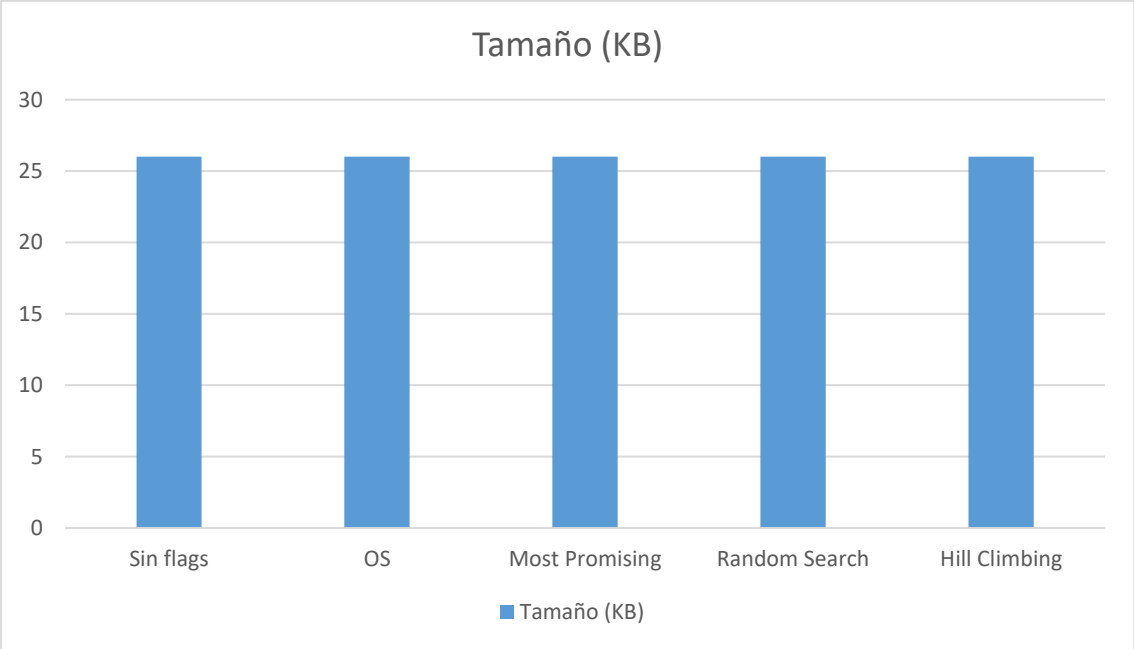

Ilustración 22: Tamaño para el fichero 2.

Por último, para el tercer fichero:

OPCIÓN	TIEMPO(ms)
Sin flags	5,3803
Most Promising	4,8622
Random Search	4,742
Hill Climbing	4,635

Tabla 19: Tiempos de ejecución para el fichero 3.

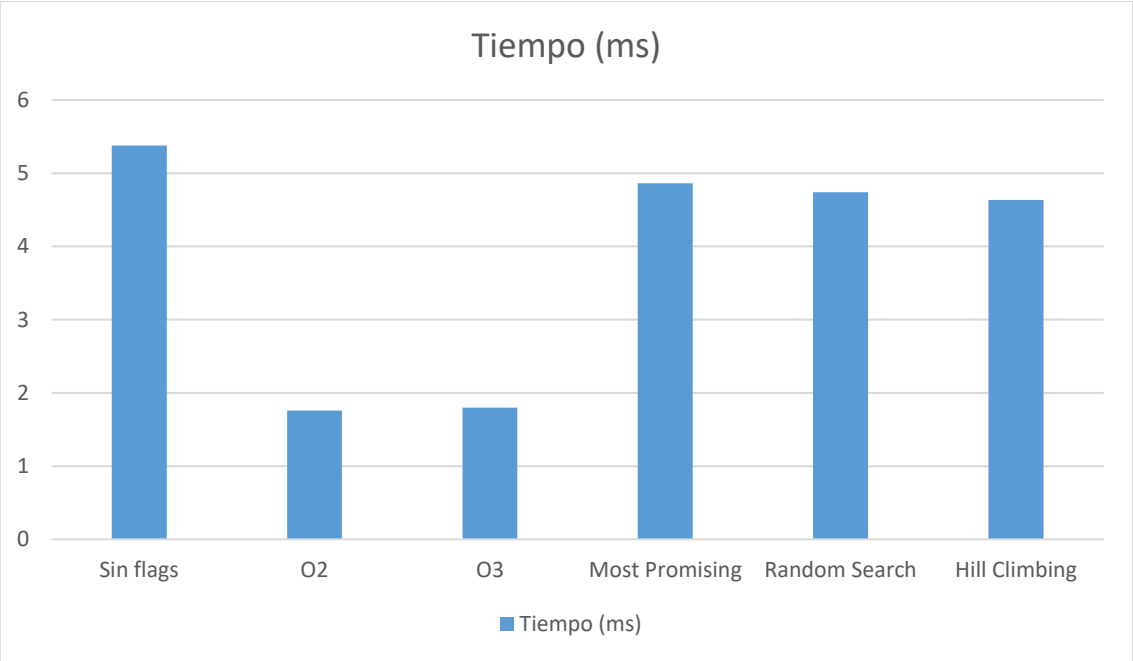

Ilustración 23: Tiempos de ejecución para el fichero 3.

Y para su tamaño:

OPCIÓN	TAMAÑO(KB)
Sin flags	29
Most Promising	29
Random Search	29
Hill Climbing	29

Tabla 20: Tamaño para el fichero 3.

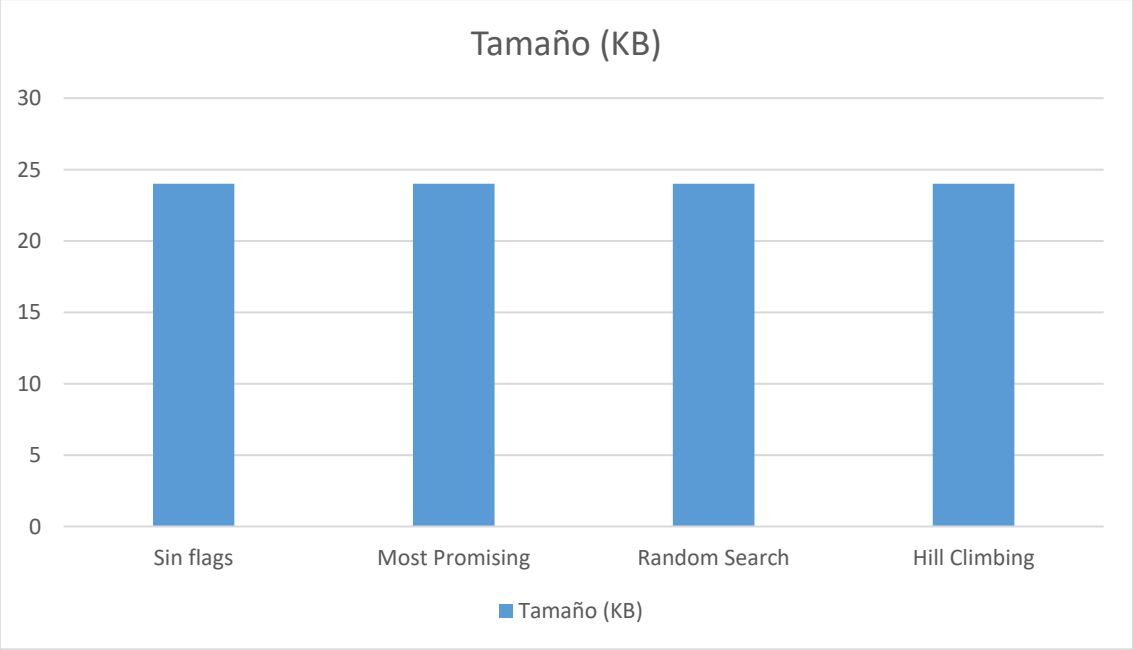

Ilustración 24: Tamaño para el fichero 3.

Podemos observar cómo nuestro algoritmo ha superado a los tiempos bases. En tamaño es probable que sea necesario utilizar benchmarks de mayor tamaño de los que disponemos para poder concluir una diferencia.

Por último, nos gustaría comentar que nos ha parecido bastante curioso que, en la mayoría de pruebas efectuadas, los algoritmos escogen combinaciones que solo contienen un flag.

6

Conclusiones y Líneas Futuras

5.1 Conclusiones

Al final del desarrollo, hemos conseguido tener una aplicación que cumple el objetivo planteado y mejora el rendimiento de los ejecutables, además de ser portable y requerir de pocos requisitos para su uso.

Como podemos observar, ha resultado ser una herramienta bastante potente que optimiza código C y C++ para programas específicos.

Además del producto obtenido, he podido explorar ciertas funciones de Java que no había utilizado, como ejecutar comandos de la terminal desde el propio código Java o utilizar la función de diseño de interfaz gráfica de "IntelliJ IDEA".

5.2 Líneas Futuras

En cuanto a posibles continuaciones del mismo, tenemos bastante claro los puntos a tratar:

- Nuevos algoritmos: Debido al diseño modular que decidimos utilizar para nuestro proyecto, sería tremendamente sencillo incluir algoritmos que sean mejores que los implementados actualmente.

- Nuevos objetivos de compilación: Por supuesto, no son solo los algoritmos los que han sido pensados de forma modular, sino también las clases encargadas de dar el objetivo al sistema. Por ello, dado que podemos buscar la eficiencia por tamaño y tiempo, sería interesante aplicarlo a otros factores, por ejemplo, la energía consumida por el proceso.

- Agregar Interacción con el usuario: Esto nos parece realmente interesante, pues el sistema en su estado actual, puede trabajar con programas que no esperen una entrada en mitad de la ejecución, y sería muy interesante poder incluir dicha función para aumentar el espectro de programas que sean compatibles con este sistema de compilación. Una opción podría ser que el usuario proporcionara un fichero en cierto formato, y que el sistema lea las entradas del mismo.

- Otros compiladores: Debido al diseño poco acoplado de nuestro sistema , sería una opción interesante modificar el código para poder adaptarlo a otros compiladores de C o C++, permitiendo llegar a más usuarios.

Bibliografía

Bjarne Stroustrup, "*The C++ Programming Language*", Fourth Edition, Addison-Wesley, Mayo 2013

Joshua Bloch, "*Effective Java*", Third Edition, Addison-Wesley, Diciembre 2017

Scott Chacon y Ben Straub, "*Pro Git*", Second Edition, Apress, 2014

Jeff Sutherland y JJ Sutherland, "*Scrum The Art of Doing Twice The Work In Half the Time", Septiembre 2014*

Alfred V. Aho, Monica S.Lam, Ravi Sethi, Jeffrey D.Ullman, *"Compilers: Principles, Techniques, and Tools",* Second Edition, Addison Wesley, Septiembre 2006

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein, "Introduction to Algorithms", Julio 2009

Benchmarksgame, Binary Trees, https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/binarytrees-gpp-2.html

Benchmarksgame, Fasta, https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/fasta-gcc-5.html

GCC, Documentación Oficial GCC: https://gcc.gnu.org/

Apéndice A
Manual de Uso

Requerimientos:

- Tener instalado Java 8 o superior.
- Tener instalado GNU Compiler Collection 4.8.1 o superior.

Debido al usuario objetivo que tiene nuestra aplicación, suponemos que cumplirá con ambos requisitos.

Dado que el sistema no requiere de instalación, solo necesitamos de estos requisitos y hacer doble click en la aplicación para iniciar su interfaz gráfica o bien si queremos la versión con interfaz CLI, ejecutar el siguiente comando:

*java -jar Compiler.jar -cli *argumentos**

A.1 Uso de la interfaz gráfica

El uso de la interfaz gráfica es muy intuitivo.

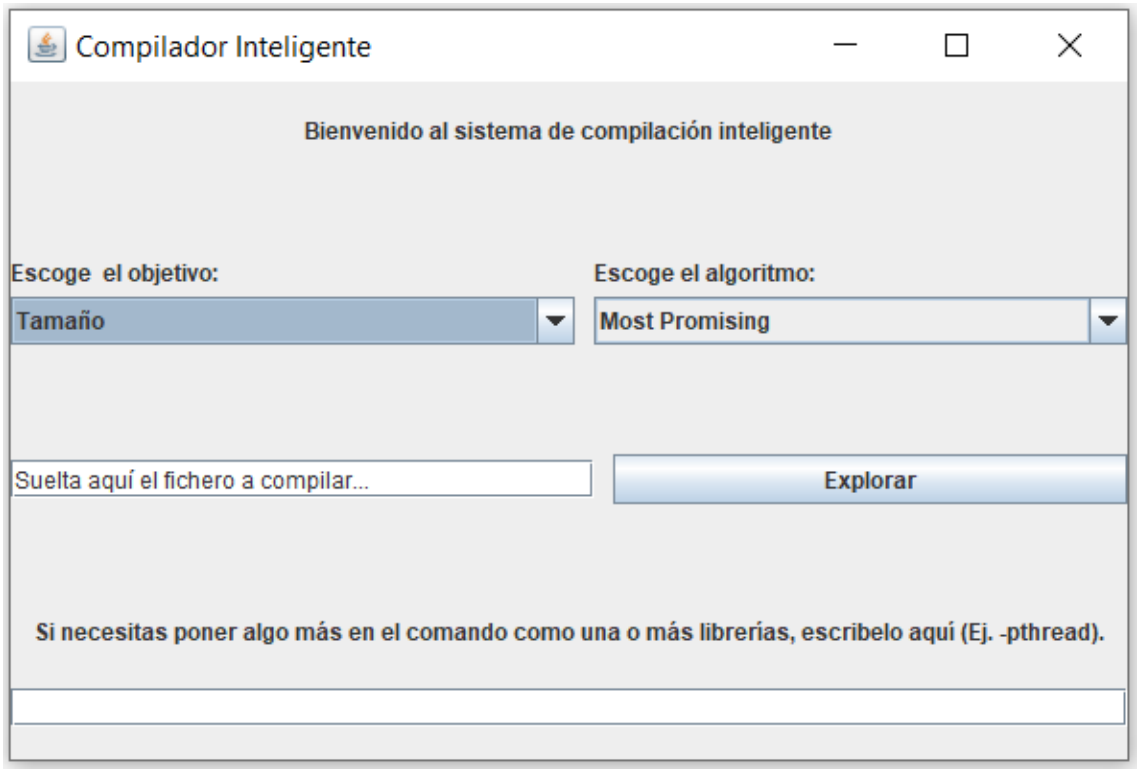

Ilustración 25: Interfaz 1.

Debemos escoger el objetivo de compilación y el algoritmo (ambos usan un valor por defecto en caso de no utilizar los desplegables) y seleccionar el fichero, bien vea mediante el explorador o arrastrándolo al campo que lo específica. Estas acciones se pueden realizar en el orden que se prefiera.

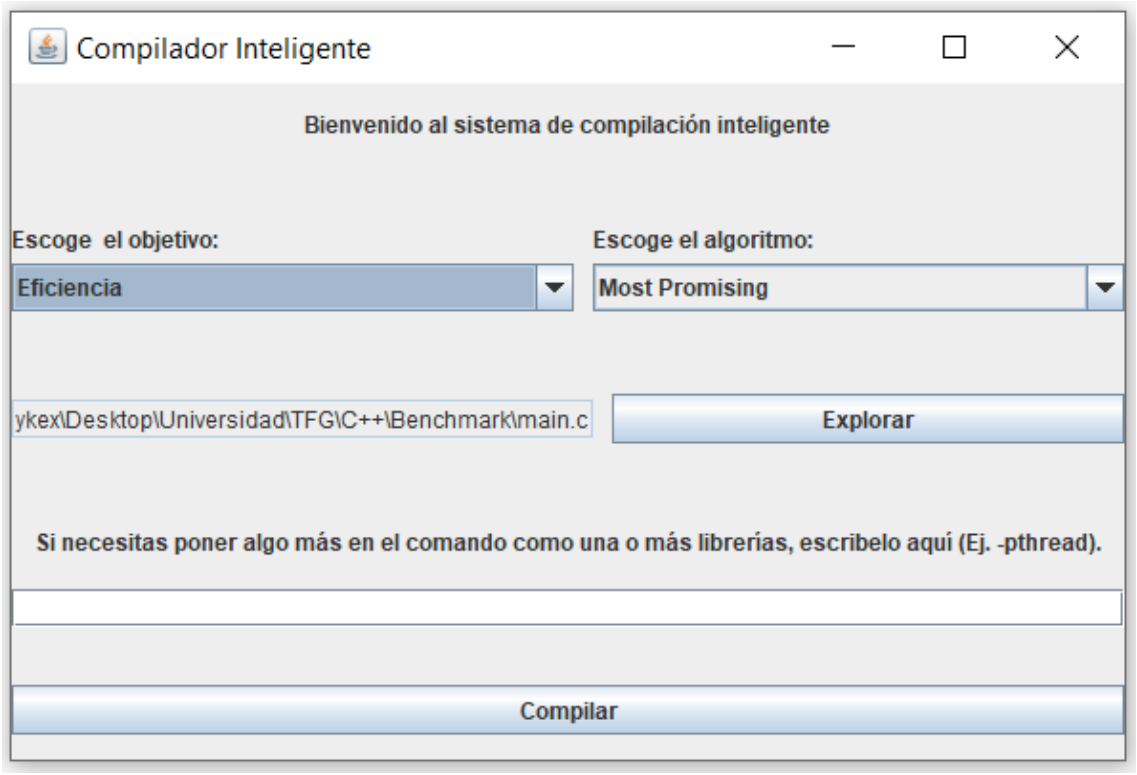

Ilustración 26: Interfaz 2.

Hecho esto, el programa nos dará la posibilidad de compilar el fichero seleccionado, tras lo cual deberemos esperar hasta el final de la ejecución para obtener nuestro fichero.

A.2 Uso de la interfaz CLI

La interfaz CLI si tiene algunos detalles a tener en cuenta.

Como norma general, recomendamos el uso de la interfaz gráfica frente a la CLI, por la comodidad del usuario, sin embargo, si se desea hacer pruebas automáticas, puedes utilizar la interfaz CLI para ello.

Esta interfaz tiene un formato muy concreto de comando, que pasamos a explicar a continuación:

Supongamos un comando como el siguiente:

java -jar Compiler.jar -cli *rutaOri* *objetivo* *algoritmo* *librerías*

Argumento por argumento, debemos rellenar todos a excepción del último, que es de carácter opcional:

- RutaOri: Como se puede suponer, será sustituido por la ruta absoluta del fichero a compilar , esta vez sin modificar.

- Objetivo: Teniendo 2 opciones, sustituiremos por "-time" si queremos obtener el menor tiempo posible de ejecución y por "-size" si queremos un mejor tamaño.

- Algoritmo: Que sustituiremos por "-rs" , "-hc" o "-mp" dependiendo si queremos "Random Search", "Hill Climbing" o "Most Promising".

- Librerías*: Si nuestro programa requiere de algún flag especial para compilar, es el lugar adecuado para ponerlo. Este parámetro es totalmente opcional.

Un ejemplo de comando sería:

java -jar Compiler.jar - C:\Users\Mykex\Desktop\main.c -time -mp -pthread