Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Diseño e implementación de un algoritmo exacto para la optimización del uso del instrumento astronómico EMIR.

Design and implementation of an exact algorithm for optimizing the use of the astronomical instrument EMIR.

Gian Luis Bolívar Diana

La Laguna, 23 de mayo de 2024

D. **Jorge Riera Ledesma**, profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"Diseño e implementación de un algoritmo exacto para la optimización del uso del instrumento astronómico EMIR"

ha sido realizada bajo su dirección por D. **Gian Luis Bolívar Diana**.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 23 de mayo de 2024

Agradecimientos

A Estela, mi mejor amiga y compañera de vida, por ver tanto potencial en mí, mostrarme que soy capaz de obtener todo lo que me proponga, ser la luz que ilumina mis noches lluviosas y alegrarme hasta que la última estrella se mantenga en el firmamento.

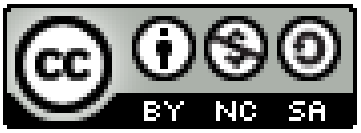
A mis padres, por apoyarme en cada decisión tomada y soportar todas esas prácticas de presentaciones interminables, siempre han confiado en mí y me han proporcionado un futuro increíble.

A mis amigos Alejandro, Fabio y Pablo, por apoyarme emocionalmente y ayudarme en mis inicios en la informática.

A mi tutor Jorge Riera por tener flexibilidad y poder aclarar de manera efectiva todas las dudas que me surgieron.

A todos los usuarios de Reddit que se tomaron el tiempo de responder este post [1] por guiarme en el desarrollo de proyectos con *C++* moderno.

Licencia



Resumen

El objetivo de este Trabajo de Fin de Grado es el de modelizar un problema de optimización que surge a partir de la gestión de un espectrógrafo multiobjeto en un telescopio, e implementar un algoritmo exacto basado en este modelo para facilitar dicha gestión.

Las características de este dispositivo permiten observar múltiples objetos simultáneamente, dado que dispone de cincuenta y cinco sensores que pueden actuar de forma concurrente. En algunos casos, la observación de un objeto requiere de la confluencia de múltiples sensores, dando lugar a ciertas restricciones de sincronización, que también formarán parte de nuestro modelo.

Para hacer uso de este instrumento, los astrónomos e investigadores elaboran una lista de objetos a observar. Puesto que posee una alta demanda, el gestor del telescopio impone una limitación de tiempo, esto obliga a seleccionar únicamente un subconjunto de estos objetos planteados. Esta limitación temporal se traducirá en una familia de restricciones de nuestro modelo.

Tras efectuar la modelización e implementación del mencionado algoritmo, se presenta una experiencia computacional que evalúa los límites computacionales de nuestra propuesta, pudiéndose resolver casos de tamaño 40, 50, 60 y 70 para cualquier nivel de sincronización.

Palabras clave: telescopio, EMIR, TSP, optimización, CPLEX, C++, grafos, sincronización, planificación.

Abstract

The objective of this Final Degree Project is to model an optimization problem that arises from the management of a multi-object spectrograph in a telescope, and to implement an exact algorithm based on this model to facilitate such management.

The characteristics of this device allow multiple objects to be observed simultaneously, since it has fifty-five sensors that can act concurrently. In some cases, the observation of an object requires the confluence of multiple sensors, giving rise to certain synchronization constraints, which will also be part of our model.

To make use of this instrument, astronomers and researchers draw up a list of objects to observe. Since it has a high demand, the telescope manager imposes a time constraint, this forces us to select only a subset of these proposed objects. This time constraint will translate into a family of constraints in our model.

After modeling and implementing this algorithm, we present a computational experience that evaluates the computational limits of our proposal, being able to solve cases of size 40, 50, 60 and 70 for any level of synchronization.

Keywords: telescope, EMIR, TSP, optimization, CPLEX, C++, graphs, synchronization, scheduling.

Índice general

Índice de figuras

Índice de Código

Capítulo 1 Introducción

1.1 Contexto

El Gran Telescopio Canarias, también conocido como GTC, es en estos momentos uno de los telescopios ópticos más grandes del mundo y se encuentra entre las Infraestructuras Científicas y Técnicas Singulares de España. Este telescopio se halla en el Observatorio del Roque de los Muchachos, en la isla canaria de La Palma. Recibe financiación tanto del Gobierno Autonómico de Canarias como del Gobierno de España, Fondos Europeos, y también de socios internacionales, como son el Instituto de Astronomía de la Universidad Nacional Autónoma de México y la Universidad de Florida, entre otros.

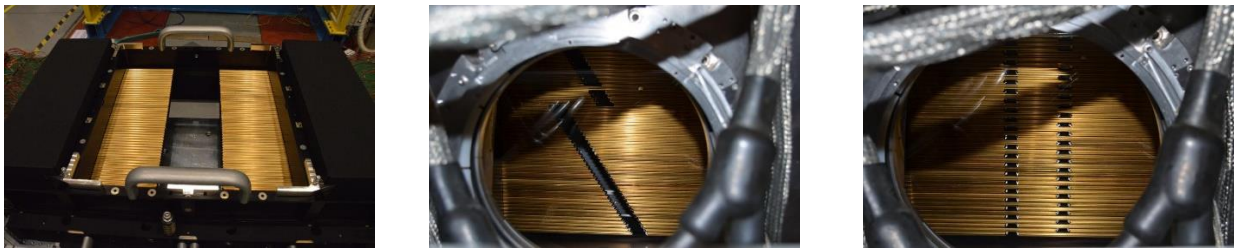


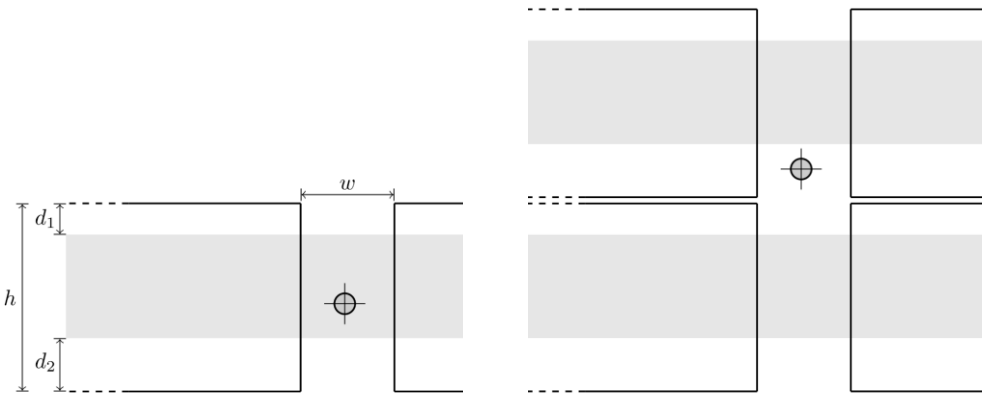
Figura 1.1.1: Fotografías de la CSU con diferentes configuraciones

Un rasgo distintivo de esta infraestructura científica es su habilidad para ofrecer datos punteros mediante la utilización de nuevos instrumentos. Entre los más recientes que se han incorporado al GTC destaca EMIR [2], que está en funcionamiento desde el año 2017. Se trata de una cámara-espectrógrafo de infrarrojo cercano de gran campo de visión, capaz de observar objetos en un campo espectral de 4×6, 64 minutos de arco al cuadrado. Una de las particularidades más llamativas de EMIR es la inclusión de una unidad robótica criogénica que facilita la configuración remota de un patrón de múltiples rendijas en su campo de visión, donde el GTC proyecta la imagen del cielo. Dicho dispositivo se conoce como CSU (véase *Figura 1.1.1*) y está formado por 55 pares de barras deslizantes enfrentadas que pueden desplazarse en una única dirección para formar una rendija. De esta manera, el campo de visión se segmenta en 55 bandas, cada una de ellas asociada a dos barras retráctiles opuestas de una altura de h segundos de arco. Este valor, h, está predefinido por el dispositivo. El ancho de la rendija, generada por un par de barras, es seleccionable por el astrónomo en función de las condiciones observacionales y la resolución espectral que se necesite.

1.2 Problemática

Podemos partir de la premisa de que el campo de visión en el cielo viene dado y contiene un conjunto de objetos de interés para el astrónomo. Teniendo en cuenta el tiempo concedido para la observación, no todos los objetos van a poder ser observados. El problema de optimización que se plantea en este trabajo consiste en seleccionar un subconjunto de objetos y la secuencia de las observaciones que se pueden realizar dentro del tiempo de observación

otorgado. Algunos objetos en el campo de visión podrían requerir la rendija creada por las barras de varias bandas contiguas. Las bandas asignadas a cada objeto y el ancho de la rendija dependen de las características del objeto (brillo y densidad) y de su posición en el campo de visión.



(a) *(b)*

Figura 1.2.1: Detalles sobre bandas, rendijas y zona muerta.

Cada banda se compone de tres áreas horizontales, tal como se ilustra en la *Figura 1.2.1(a)* una zona muerta superior de longitud *d1* segundos de arco, una zona central y una zona muerta inferior de longitud *d2* segundos de arco. Los valores *d1* y *d2* están determinados por la necesidad de evitar los bordes de barra no uniformes en estas ubicaciones, debido al diseño mecánico de las barras, que deben ajustarse y deslizarse suavemente sobre las barras inferiores y superiores adyacentes a cada una.

Un objeto situado en el área central de una banda puede ser observado mediante la única rendija formada por las dos barras de dicha banda. Cuando un objeto se encuentra en la zona muerta superior o inferior de una banda, por razones técnicas, la observación de este objeto requiere la rendija creada por las barras de (al menos) dos bandas contiguas.

La *Figura 1.2.1(a)* muestra las tres áreas que dividen una banda, con un objeto en el área central. Las dos barras opuestas configuran una rendija que enfoca el objeto. La *Figura 1.2.1(b)* muestra dos bandas contiguas con un objeto situado en una zona muerta. Para observar dicho objeto, las barras en las dos bandas deben configurarse adecuadamente para generar una rendija que rodee al objeto.

Como se ha mencionado, algunos objetos podrían necesitar tres o más bandas contiguas para ser observados.

Dado que las barras pueden configurarse de manera independiente, EMIR puede realizar muchas observaciones simultáneas (hasta 55), pero la sincronización de las barras, necesaria para cada observación, es fundamental si buscamos una gestión eficiente del telescopio. Además, mover una barra consume bastante tiempo, por lo que los tiempos de reconfiguración en el CSU son largos en EMIR. El uso eficiente del tiempo del telescopio impone una planificación cuidadosa para maximizar el rendimiento científico en el tiempo de observación disponible y limitado.

Nos referimos a este problema de optimización como *Orienteering Problem with*

Synchronization (OPS), dado que tiene características del *Orienteering Problem (OP)*.

1.3 Objetivo

Este Trabajo de Fin de Grado persigue establecer las bases para elaborar una herramienta de ayuda a la toma de decisión que permita determinar qué objetos de cada solicitud deben seleccionarse, así como establecer la configuración de la *CSU* para que el instrumento EMIR proporcione el mejor servicio a los investigadores.

Con el fin de alcanzar este objetivo se definirá formalmente y se modelizará en el Capítulo 2 uno de los problemas de optimización que surgen a partir de la gestión de EMIR. A continuación, se implementará (Capítulo 4) un algoritmo exacto basado en este modelo para facilitar dicha gestión, utilizando un resolutor general comercial, conocido como CPLEX. Finalmente, a fin de estudiar el comportamiento del algoritmo bajo ciertos requerimientos de sincronización, efectuaremos una amplia experiencia computacional sobre varios conjuntos de entradas artificiales.

1.4 Antecedentes

Detallamos a continuación los antecedentes del *OPS* y su relación con otros problemas conocidos:

1.4.1 Problemas de orientación

Los Problemas de Orientación (*OPS*) representan una familia de problemas de enrutamiento de vehículos que tienen en cuenta situaciones prácticas donde servir a un cliente es opcional, como sucede en nuestro problema, generando un beneficio si el servicio se realiza dentro de un límite de tiempo. Remitimos a la revisión de [3] para un estudio extenso de esta familia de problemas y sus aplicaciones prácticas.

Relajar el *OPS* eliminando las restricciones de sincronización resulta nos lleva a un conjunto de casos independientes del bien conocido *Orienteering Problem* (*OP*) (ver, por ejemplo, [4] o [5]). El *OP* tiene como objetivo seleccionar un subconjunto de clientes y diseñar una ruta cuya duración total no sea mayor que un límite de tiempo dado, para maximizar el beneficio total recolectado. Es una combinación del Problema del Ciclo y el Problema de la Mochila, donde la duración total del ciclo no tiene que ser minimizada pero sí está restringida.

Aunque esta versión relajada del *OPS* implique un conjunto de vehículos para servir a algunos clientes, no debe confundirse con el *Team Orienteering Problem* (*TOP*) (ver, por ejemplo, [6] o [7]). El *TOP* es una extensión del *OP* que también considera múltiples rutas para servir a los clientes, pero donde un cliente debe ser servido por un solo vehículo. En otras palabras, el *TOP* no incluye ningún problema de sincronización y no hay preasignación de vehículos a clientes. Sin embargo, el *OPS* coincide con el *TOP* cuando cada objeto requiere solo un par de bandas para ser observado.

1.4.2 Problemas de sincronización

Según [8], el *OPS* es un caso de sincronización exacta de operaciones, lo que significa que

dos vehículos deben comenzar a ejecutar sus operaciones en sus respectivas ubicaciones al mismo tiempo. La literatura sobre problemas de enrutamiento de vehículos con esta característica describe enfoques exactos principalmente utilizando formulaciones basadas en variables de arco que emplean variables de tiempo, las cuales representan el inicio de la ejecución de cada trabajo. Estas formulaciones son implícitamente no lineales, y su linealización requiere el uso de la técnica de la big-M, tal y como hacemos en nuestra propuesta (Capítulo 2); ver, por ejemplo, [9], [10], [11], [12], [13] y [14].

1.4.3 Problemas de asignación de personal

Los problemas de asignación de personal describen una familia de problemas de sincronización exacta de operaciones que surge en varios contextos donde realizar cada trabajo requiere la cooperación entre equipos, posiblemente con diferentes habilidades. La asignación de técnicos a trabajos de servicio, dónde resolver cada trabajo específico requiere una combinación de técnicos con habilidades individuales, es un ejemplo de esta familia de problemas. Aunque estos problemas comparten características en común con nuestro problema, como la sincronización exacta de operaciones, también difieren en otras características como la estructura de las rutas, las funciones objetivo y el requisito de procesar todos los trabajos.

En el artículo [10] introducen el Problema de Asignación de Personal como un problema multicriterio y proponen un algoritmo de *simulated annealing* para resolverlo heurísticamente. Por otro lado, en [11] citan una aplicación donde los técnicos son enviados desde una oficina central a varios lugares en el Puerto de Singapur para realizar trabajos de servicio que pueden requerir múltiples técnicos. [15] describen una aplicación en la industria cinematográfica donde algunas escenas filmadas en diferentes ubicaciones requieren enrutamiento de equipos de filmación, como actores, operadores de cámara, técnicos de iluminación y maquilladores. [14] introducen otra aplicación relacionada con la gestión aeroportuaria: después de llegar al aeropuerto, la aeronave se detiene en diferentes ubicaciones y se despliega personal de tierra con diferentes responsabilidades para realizar varios trabajos, como manejo de equipaje, limpieza de aeronaves, servicio de deshielo de aeronaves, servicio de suministro de combustible para aviación y verificación de mantenimiento de aeronaves. La literatura también menciona ejemplos que surgen en el sector de atención domiciliaria, donde trabajadores especializados viajan entre los hogares de los pacientes, lo que exige trabajo colaborativo.

Capítulo 2 Formulación

2.1 Parámetros de entrada

Sea $J = \{1, \ldots, n\}$ el conjunto de trabajos u observaciones potenciales. Ningún trabajo en el conjunto J es de ejecución obligatoria. Si procesamos un trabajo j este problema da lugar a un beneficio b_j, asociado con su prioridad. Cada observación asociada con un trabajo $j \in J$ requiere un tiempo de procesamiento de p_j unidades. El proceso de configuración de un procesador k desde que acaba el trabajo i con el fin de procesar el trabajo j consume un tiempo de transición c_{ij}. Por conveniencia en la notación, denotamos $t_{ij} = p_i + c_{ij}$. De cara al modelo que presentaremos después, también consideramos trabajos 0 y $n+1$ que son trabajos ficticios iniciales y finales, respectivamente, y representan un estado de seguridad de los procesadores (en la aplicación astronómica representan el estacionamiento del par de barras de cada banda en medio del campo de visión). Sea $p_j = b_j = 0$ para $j \in \{0, n+1\}$, y $V = J \cup \{0, n+1\}$. Sea $K = \{1, \ldots, m\}$ el conjunto de procesadores paralelos no idénticos configurables. Cada trabajo j requiere el subconjunto $K_j \subseteq K$ de procesadores. Todos los procesadores en K_j necesitan estar simultáneamente en configuraciones específicas durante al menos p^j unidades de tiempo para realizar j. De manera similar, denotamos por $J^k \subseteq J$ el conjunto de trabajos que requieren el procesador k. Sea $V^k = J^k \cup \{0, n+1\}$.

Sea $G^k = (V^k, A^k)$ el grafo de transición para el procesador k, donde A^k es el conjunto de arcos entre todos los vértices en V^k, excepto los que salen de $n+1$ o entran en 0. Sea $G = (V, A)$ el grafo agregado donde A es un conjunto de tripletas (i, j, k) tal que $k \in K$ y $(i, j) \in A^k$. Nótese que G es un multigrafo, ya que dos vértices pueden estar vinculados por varios arcos. Cada tripleta $a = (i, j, k) \in A$ está asociada con una longitud t_a igual al tiempo t_{ij}. Finalmente, sea L la limitación de tiempo dada para realizar los trabajos seleccionados. Suponemos que $L \geq t_{0j} + t_{j,n+1}$ para todo $j \in J^k$ y $k \in K$.

2.2 Caracterización de las soluciones factibles

Caracterizamos una solución factible para el *OPS* mediante un camino elemental $P^k \subset A^k$ desde 0 hasta $n+1$ en G^k para cada procesador $k \in K$. El camino P^k representa la secuencia de trabajos que el procesador k debe procesar, comenzando y terminando en las posiciones de estacionamiento. No todos los trabajos necesitan ser visitados por un camino, pero cuando un camino visita un trabajo j, entonces todos los caminos P^k con $k \in K_j$ deben visitar j. Además, debe existir un valor s_j para cada $j \in V$ que cumpla:

$$s_i - s_j \leq -t_{ij} \qquad k \in K, (i, j) \in P^k \qquad (1)$$

$$s_{n+1} - s_0 \leq L \qquad (2)$$

El valor s_j representa el instante de comienzo del trabajo j y, junto con las restricciones (1), garantiza la sincronización de los procesadores al realizar ese trabajo. Es decir, todos los procesadores involucrados en el procesamiento de un trabajo específico j pueden llegar a la configuración para comenzar a procesar j en diferentes momentos, pero todos necesitan comenzar el proceso j al mismo tiempo s_j. La desigualdad (2) limita el tiempo de

procesamiento total.

El conjunto de desigualdades (1) y (2) es conocido como *System of Difference Constraints* (ver [16]). El *OPS* tiene como objetivo encontrar los caminos P^k y los valores s_j que maximicen la suma de los premios asociados con los trabajos seleccionados. Desde esa perspectiva, el *OPS* también puede considerarse como un problema de rutas, donde los trabajos son los clientes y los procesadores son los vehículos.

2.3 Modelo

Presentamos una formulación matemática para describir el *OPS*. Esta formulación se construye sobre las siguientes dos familias de variables. Para cada procesador $k \in K$ y cada arco $(i,j) \in A^k$, la variable binaria x_{ij}^k representa si el procesador k está configurado para procesar el trabajo j inmediatamente después de haber procesado el trabajo i. Para cada trabajo $j \in J$, la variable binaria y_j determina si el trabajo j es seleccionado para ser procesado.

Nuestro modelo es una formulación compacta que también utiliza variables continuas para determinar el instante de inicio de cada observación seleccionada. Linealizamos las restricciones de tiempo y sincronización (1) y (2) utilizando el conocido método de la M grande.

Para simplificar la notación, definimos la siguiente notación estándar. Dado un procesador $k \in K$ y un subconjunto de vértices $S \subset V^k$, utilizamos $\delta_-^k(S) = \{(i,j) \in A^k \mid i \in V^k \backslash S,\ j \in S\}$ y $\delta_+^k(S) = \{(i,j) \in A^k \mid i \in S,\ j \in V^k \backslash S\}$. Si $S = \{i\}$, simplemente escribimos $\delta_-^k(i)$ y $\delta_+^k(S)$. También utilizamos $\delta_-(S) = \{(i,j,k) \in A \mid (i,j) \in \delta_-^k(S), k \in K\}$ y $\delta_+(S) = \{(i,j,k) \in A \mid (i,j) \in \delta_+^k(S), k \in K\}$.

Escribimos $x^k(F)$ en lugar de $\sum_{a \in F} x_a^k$ sí $F \subseteq A^k$, y $x(F)$ en lugar de $\sum_{a \in F} x_a$ si $F \subseteq A$.

Como habíamos avanzado anteriormente, el modelo requiere variables continuas s_j para cada tarea j en V. La variable s_j representa el instante de inicio de la tarea j. A continuación, el siguiente modelo lineal entero mixto describe el *OPS*:

$$\max \sum_{j \in J} b_j y_j \tag{3}$$

sujeto a:

$$x^k(\delta_+^k(0)) = 1 \qquad k \in K \tag{4}$$

$$x^k(\delta_-^k(n+1)) = 1 \qquad k \in K \tag{5}$$

$$x^k(\delta_+^k(j)) - x^k(\delta_-^k(j)) = 0 \qquad k \in K,\ j \in J^k \tag{6}$$

$$x^k(\delta_-^k(j)) = y_j \qquad k \in K,\ j \in J^k \tag{7}$$

$$s_i - s_j + L x_{ij}^k \leq L - t_{ij} \qquad k \in K, (i,j) \in A^k \tag{8}$$

$$s_{n+1} - s_0 \leq L \tag{9}$$

$$x_{ij}^k \in \{0,1\} \qquad k \in K, (i,j) \in A^k \tag{10}$$

$$y_j \in \{0,1\} \qquad j \in J \tag{11}$$

La función objetivo (3) maximiza la ganancia asociada con las tareas seleccionadas. Las ecuaciones (4) y (5) establecen, respectivamente, la configuración de inicio y fin para cada procesador. Las ecuaciones (6) y (7) definen las variables y_j, y exigen que cada procesador $k \in K_j$ debe procesar la tarea j si $y_j = 1$ para cada $j \in J$. La restricción (9) es (2), y las restricciones (8) linealizan las restricciones (1). Tienen un triple papel: primero, garantizan la conectividad de las configuraciones para cada procesador en la tarea j con sus respectivas configuraciones de estacionamiento inicial y final, 0 y $n + 1$; segundo, fuerzan que cada procesador $k \in K_j$ esté adecuadamente configurado antes de comenzar a procesar una tarea seleccionada j; y tercero, establecen una limitación en la duración del tiempo en cada procesador. Las restricciones (8) son conocidas en la literatura como desigualdades de Miller-Tucker-Zemlin (ver, por ejemplo, [17]). Finalmente, las cotas (10) y (11) establecen los límites de las variables x_{ij}^k y y_j, respectivamente.

Capítulo 3 Lenguaje y herramientas

3.1 C++

C++ [18] es un lenguaje de programación compilado de propósito general creado en la época de los 80 y ampliamente utilizado desde entonces. Diseñado como una extensión del lenguaje *C*, este lenguaje añade, entre otras características, mecanismos que permiten la manipulación de objetos. Entre sus usos más comunes, encontramos el desarrollo de aplicaciones que requieran un alto rendimiento, debido a su capacidad de controlar la memoria, su programación eficiente a nivel de hardware, sus bibliotecas especializadas y su cercanía al lenguaje de máquina.

Se ha escogido este lenguaje para el desarrollo del proyecto gracias a la disponibilidad de la librería del resolutor de *IBM*, añadiendo la necesidad de utilizar un lenguaje de bajo nivel para realizar operaciones con suma rapidez.

3.2 CMake

CMake [19] es un sistema de meta construcción de código abierto multiplataforma que puede crear, probar y empaquetar software. Se puede utilizar para admitir múltiples entornos de compilación nativos, incluidos *make*, *xcode* de *Apple* y *Microsoft Visual Studio*. *CMake* es ampliamente usado para proyectos escritos en *C* o *C++*, aunque puede ser utilizado para construir código fuente de otros lenguajes (véase [20] para tutorial de *C++*).

Puesto que este proyecto requiere del uso de diversas dependencias, se utilizará este generador de sistemas de compilación para poder incluirlas y usarlas de la misma forma que utilizamos las librerías del sistema.

3.3 IBM ILOG CPLEX Optimization Studio

IBM ILOG CPLEX Optimization Studio [21] es una solución de análisis prescriptivo que permite un rápido desarrollo e implementación de modelos de optimización de decisiones mediante programación matemática y de restricciones. Esta librería nos permite producir decisiones precisas y lógicas para problemas de planificación y asignación de recursos, ofreciéndonos algoritmos paralelos distribuidos para programación de enteros mixtos y solucionadores de programación matemática flexibles y de alto rendimiento para programación lineal y programación de enteros mixtos.

Se hará uso de esta herramienta una vez hayamos definido en el código las variables a utilizar, la función objetivo del resolutor y las restricciones aplicables sobre este, generándonos así una solución exacta al problema planteado.

3.4 Librerías adicionales

3.4.1 InputParser

La librería *InputParser* [22] se ha desarrollado en *C++23*, con el objetivo de poder utilizar los argumentos proporcionados por la línea de comando al utilizar el ejecutable. Ofrece una implementación sencilla y flexible, que nos permite configurar cuales son los parámetros esperados, como se van a utilizar y bajo que restricciones se tienen que ceñir para poder proporcionarse correctamente. A su vez, nos facilita la implementación en nuestro proyecto utilizando *CMake* 3.22.

A pesar de que aún no se encuentre en su versión final, utilizaremos este paquete para indicar, por ejemplo, cuál es el archivo que el resolutor utilizará, las clases de instancias a resolver, o el nombre del fichero que contenga la solución factible.

3.4.2 Nlohmann JSON

El usuario Nlohmann ha diseñado una librería [23] en *C++11* que permite analizar gramaticalmente un archivo con formato *JSON*, creando un tipo de dato de primera clase, una característica trivial implementada nativamente en lenguajes como *Python*.

El código consiste esencialmente de un archivo cabecera que se incluye, sin la necesidad de utilizar subproyectos, dependencias o un sistema de compilación complejo. Cuenta con una cobertura del 100% y, tras un análisis usando *Valgrind* [24] y los *Clang Sanitizers*, es libre de pérdida de memoria.

Puesto que la entrada de datos viene dada en un fichero con este formato, y la representación de la solución será expulsada también con este tipo de dato, se ha incluido esta "librería" para facilitarnos la tarea de conversión.

3.4.3 GoogleTest

GoogleTest [25] es el framework de pruebas unitarias e imitación de funcionalidades para *C++* utilizado por Google. Esta librería ofrece una sintaxis sencilla de entender y escribir, utilizando su conjunto completo de *macros* para realizar comprobaciones y ejecutar pruebas con diferentes conjuntos de datos, permitiéndonos a su vez la inicialización común de las mismas (consulte [26]). Cuenta con una integración nativa con diferentes sistemas de compilación, entre ellos *CMake*.

Junto a la extensión *C++ TestMate* [27] para *Visual Studio Code*, se desarrollarán pruebas unitarias haciendo uso de este framework, las cuales serán capaces de comprobar la funcionalidad de nuestro resolutor.

3.5 Otras herramientas

3.5.1 Clang-Format

Clang-Format [28] es una herramienta que se utiliza para formatear automáticamente código en diversos lenguajes, entre ellos *C++*, de modo que los desarrolladores no tengan que preocuparse por problemas de estilo durante las revisiones del código. Este proporciona

una opción para definir cuales reglas se deben aplicar sobre el proyecto, utilizando archivos con formato *YAML*, denominados *.clang-format* o *_clang-format*.

En este trabajo se ha definido este archivo con todas las reglas a aplicar, buscando así ser constantes en el estilo y agilizar el proceso de desarrollo.

3.5.2 Clang-Tidy

Clang-Tidy [29] es un analizador de código para *C++* basado en *Clang*. Este se ejecuta después de una compilación exitosa y proporciona un marco extensible para diagnosticar y corregir errores de programación típicos mediante análisis estático, como violaciones de estilo, junto a recomendaciones de código para modernizar la sintaxis. Esta es la herramienta de análisis predeterminada al usar el conjunto de herramientas *LLVM/clang-cl*, disponible tanto en *MSBuild* [30] como en *CMake*.

De la misma forma que con la herramienta anterior, se incluirá un fichero con su configuración y se añadirá las llamadas requeridas para que se ejecute cada vez que se compila el programa. De esta forma nos aseguraremos de que el código escrito cumple una serie de normas aceptadas ampliamente.

3.5.3 Valgrind

Valgrind [24] es una herramienta de depuración y perfilado, de código abierto, que se utiliza para detectar problemas de memoria, como *memory leaks*, uso incorrecto de la misma, por ejemplo, el acceso a partes de memoria después de haber sido liberada, y problemas de rendimiento. Funciona en sistemas operativo de tipo Unix, y proporciona informes detallados identificando posibles problemas que pueden conducir a fallos de segmentación.

Se utilizará cada vez que se realice algún cambio en el código fuente, con el objetivo de mantener al máximo la seguridad de este y evitar fallos futuros.

Capítulo 4 Fases y Desarrollo

4.1 Preparación del entorno de trabajo

Como se mencionó anteriormente, se utilizará el lenguaje de programación *C++* para el desarrollo de este proyecto, más concretamente el estándar 23, por lo que necesitaremos un compilador que sea capaz de soportar esta versión. En este trabajo se optó por hacer uso del *Windows Subsystem for Linux* (WSL) [31], puesto que el sistema operativo del ordenador disponible es *Windows* (aunque en otras circunstancias se podría trabajar directamente en un ordenador con sistema Linux). Dentro de dicha máquina virtual ya viene instalado el compilador *GNU*, sin embargo, la versión más reciente que soporta es *C++14* por lo que debemos instalarlo aparte con los siguientes comandos (véase [32] para más información):

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt install gcc-13 g++-13
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-13 \
  100 --slave /usr/bin/g++ g++ /usr/bin/g++-13
```

Figura 4.1.1: Cómo instalar g++ 13 en Linux.

Con el compilador instalado, la siguiente herramienta a configurar es el generador de sistemas de compilación *CMake*, el cual viene por defecto con su versión 3.22. Esta nos ofrece todas las funcionalidades necesarias, por lo que no hace falta actualizarla a una más reciente.

El entorno de trabajo estará contenido en el directorio denominado *src*, este a su vez se dividirá en seis subdirectorios, uno para cada subproyecto a definir. En cada uno de estos subdirectorios existirá un archivo *CMakeLists.txt* con los archivos a compilar, la localización de los archivos cabecera y la inclusión de las dependencias necesarias. De forma adicional se asignará un alias a cada subproyecto, de forma que sea intuitiva la adición de este en otros subproyectos.

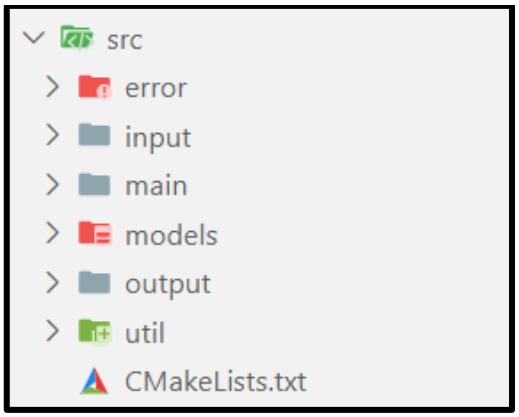


Figura 4.1.2: Estructura de directorios

Posteriormente, procederemos a instalar el resolutor de problemas lineales *CPLEX*. Para obtener el instalador deberemos pagar una suscripción mensual [33] a *IBM*. Una vez

comprada la licencia pasamos a ejecutar el programa, seleccionando como ruta destino *lib/CplexStudio2211* (si esta ruta es alterada, tenga en cuenta que debe cambiarla a su vez cuando se importe en los archivos de configuración de *CMake*). La forma de especificar la ruta de esta librería es distinta a las demás, por lo que se hará especial énfasis en esta en comparación al resto. Para indicar correctamente a *CMake* como añadir esta dependencia, debemos copiar el siguiente código en el archivo *CMakeLists.txt* ubicado en el subproyecto que la utilice:

```
project(project_name)

# Add source files ...
# Add alias ...

set(CPX_PATH ${CMAKE_SOURCE_DIR}/lib/CplexStudio2211/)
set(CPX_LIB_PATH ${CPX_PATH}/cplex/lib/x86-64_linux/static_pic/)
set(CON_LIB_PATH ${CPX_PATH}/concert/lib/x86-64_linux/static_pic/)

find_library(CPX_LIBRARY libcplex.a HINTS ${CPX_LIB_PATH})
find_library(ILO_LIBRARY libilocplex.a HINTS ${CPX_LIB_PATH})
find_library(CON_LIBRARY libconcert.a HINTS ${CON_LIB_PATH})

target_include_directories(${PROJECT_NAME}
    PUBLIC ${CPX_PATH}/cplex/include
    PUBLIC ${CPX_PATH}/concert/include
)

target_link_libraries(${PROJECT_NAME}
    ${CON_LIBRARY}
    ${ILO_LIBRARY}
    ${CPX_LIBRARY}
)
```

Código 4.1.1: Instalación de CplexStudio.

Proseguimos con la instalación de librerías adicionales, para las cuales el proceso es idéntico (exceptuando *GoogleTest*, donde la configuración requiere pasos extra). Estas se añadirán a través de *CMake*, incluyéndose sólo en el primer subproyecto que la utiliza, de forma que, si este subproyecto es a su vez incluido en otro, no hará falta la inyección de esta dependencia. Para añadirlas, adaptaremos el siguiente código en el archivo *CMakeLists.txt* donde se añadan:

```
project(project_name)

include(FetchContent)
FetchContent_Declare(
  dependency_name
  URL https://example.com/release/dependency-1.2.3.zip
)

# Only for GoogleTest
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)

FetchContent_MakeAvailable(dependency_name)

target_link_libraries(${PROJECT_NAME}
  dependency_name
)

# Only for GoogleTest
include(GoogleTest)
gtest_discover_tests(${PROJECT_NAME})
```

Figura 4.1.3: Instalación general de librería.

Por último, se instalarán el resto de las herramientas usadas para el desarrollo de un código moderno y limpio. Empezando por *clang-tidy* y *clang format*, donde la mayoría de los pasos son reciclados entre estas, permitiendo la ejecución de las siguientes instrucciones para su obtención:

```
sudo wget https://apt.llvm.org/llvm.sh
sudo chmod +x llvm.sh
sudo ./llvm.sh 19
sudo apt-get update
sudo apt update
sudo apt-get install -y clang-tidy-19 clang-format-19
```

Figura 4.1.4: Instalación de clang-tidy y clang-format v19.

Y finalizando por el analizador *Valgrind*, el cual tiene una instalación muy sencilla y directa:

```
sudo apt-get install -y valgrind
```

Figura 4.1.5: Instalación de Valgrind.

4.2 Obtención de batería de problemas

La batería de problemas vendrá proporcionada de un generador [34]. Podemos asumir que los objetos están dispuestos de manera aleatoria en el plano celeste, por lo que la generación tendrá esta característica. Esta se compone de 180 instancias, generadas para simular datos provenientes de la aplicación del telescopio que motiva este trabajo, donde las barras se mueven en "una línea" y, por lo tanto, el tiempo de transición depende de una sola

coordenada.

Para posteriormente medir el impacto de la sincronización, la batería es a su vez dividida en tres clases, de acuerdo con el número de bandas (procesadores) requeridos para observar (procesar) cada objetivo (trabajo).

El primer paso fue generar 5 catálogos, cada uno conteniendo 500 objetivos, generados de forma aleatoria en el campo de visión del EMIR de 240.0 x 398.4 segundos de arco al cuadrado. Caracterizamos cada objetivo i por la tupla (u_i, v_i, p_i, b_i). La coordenada horizontal es la variable u_i y la vertical la variable v_i, estas han sido seleccionadas de forma aleatoria dentro del intervalo continuo $[-120.0, 120.0]$ y $[-199.2,\ 199.2]$ respectivamente. El tiempo de procesado p_i es un número entero uniformemente generado dentro del intervalo $[1, 200]$. La variable b_i representa el beneficio y pertenece al set $\{1, 10, 100\}$. Se han caracterizado las configuraciones iniciales (0 y $n+1$) por la tupla (0,0,0,0). No se hará uso de los catálogos como una instancia, sino de una plantilla.

El tiempo de transición es definido como el movimiento horizontal de la barra deslizante $d_{ij} = |u_i - u_j|$.

Consideramos $|K| = 55$ pares de barras, cada barra k cubriendo los objetivos en la posición (u_i, v_i) con $398.4 \times (k-1)/55 - 199.2 \leq v_i < 398.4 \times k/55 - 199.2$ para $k = 1, \ldots, 55$. Creamos V^k para cada instancia y banda $k \in K$ de acuerdo con nuestras tres clases. Las instancias en la clase g (para $g \in \{1,2,3\}$) tienen g bandas asignadas a cada objetivo ($|K_j| = g$ para $j \in V \backslash \{0, n+1\}$). Las bandas asignadas a un objetivo con posición (u_i, v_i) han sido seleccionadas minimizando la distancia v_i al centro de la banda. Al hacer esto, las instancias de clase 1 no requieren sincronización. Por lo tanto, estas instancias nos ayudan a analizar el comportamiento de nuestro algoritmo en el problema sin sincronización, lo que en un sentido es similar al clásico *Orienteering Problem*. Por otra parte, los requerimientos de la sincronización tienen un mayor impacto en las instancias de clase 3, las cuales son las más difíciles de resolver. Las instancias en la aplicación del telescopio combinan características de las tres clases.

Por cada catálogo de objetivos se han generado instancias del mismo tamaño, escogidos a partir de un tamaño razonable que se pueda utilizar con la última clase. Este se ha escogido a partir de un experimento computacional previo que buscaba evaluar el comportamiento del algoritmo, que van desde casos fáciles a difíciles. El número n de objetivos para cada clase está en el set $\{40,50,60,70\}$. Los n objetivos de cada instancia son escogidos de manera aleatoria de los 500 objetivos del catálogo.

Definimos la limitación del tiempo para cada instancia como $L = \lceil 2 \cdot \alpha \cdot (max\{u_j \mid j \in J\} - min\{u_j \mid j \in J\}) \rceil$, donde α es un parámetro de entrada en $\{0.25, 0.50, 0.75\}$.

4.3 Lectura de entradas de prueba

A partir de la batería de problemas podemos basar el primer subproyecto, encargado de la lectura del archivo de instancia. Localizado en la subcarpeta *input*, se importará la librería Nlohmann *JSON* [23] para la asignación inmediata de valores.

La clase *OpsInstance* será la responsable de almacenar todos los atributos provenientes

del fichero inicial, esta representa la instancia más básica del *OPS*. La estructura de datos queda definida con la siguiente composición:

- name : Nombre de la instancia.
- date_stamp : El momento en el que fue generada la instancia.
- type : Tipo de instancia (asociada al nombre).
- objects_per_sliding_bar : Objetos que pueden ser observados por cada barra deslizante.
- time_to_process : Matriz con el tiempo necesario para procesar el objeto i y transitar al objeto j.
- priorities : Beneficio (o prioridad) obtenido al procesar cada objeto.
- time_limit : Tiempo máximo para usar el telescopio.
- alpha : Porcentaje total del tiempo máximo.
- scaling_factor : Factor de conversión aplicado al tiempo.

El operador >> estará sobrecargado para permitir la lectura desde un *input stream*. Por dentro creará un objeto de tipo *nlohmann::json* del cual se extraerán los atributos necesarios.

```
std::istream &
operator>>(std::istream &input_stream, OpsInstance &ops_instance) {
  nlohmann::json json_instance;
  input_stream >> json_instance;
  ops_instance.setFromJson(json_instance);
  return input_stream;
}
```

Código 4.3.1: Sobrecarga del operador >> (OpsInstance).

El método *setFromJson* será el encargado de la asignación de atributos, realizando la conversión de tipos necesaria.

```
void OpsInstance::setFromJson(const nlohmann::json &json_instance) {
  time_to_process_ = json_instance["T"].get<std::vector<std::vector<int>>>();
  name_ = json_instance["id"][0].get<std::string>();
  date_stamp_ = stringToDateStamp(json_instance["id"][1].get<std::string>());
  type_ = json_instance["type"].get<int>();
  priorities_ = json_instance["b"].get<std::vector<int>>();
  objects_per_sliding_bar_ =
    json_instance["Jk"].get<std::vector<std::vector<unsigned int>>>();
  alpha_ = json_instance["alpha"].get<double>();
  time_limit_ = json_instance["L"].get<int>();
}
```

Código 4.3.2: Asignación de atributos desde json.

Al construir el modelo se mencionó que a cada barra deslizante le corresponde un grafo, donde los nodos son los objetos que esa barra puede visualizar y los arcos son las transiciones

entre objetos. Para representar esta estructura se han desarrollado tres clases. La clase *Graph* posee dos contenedores, uno para los arcos, y otro para los nodos. Un arco (*Arc*) está conformado por un nodo origen, un nodo destino y un coste de transición. Un nodo (*Node*) se compone de una colección de nodos sucesores y otra de nodos predecesores, junto con un identificador único. Para representar una unión de puntos se instancia un objeto de tipo *ArcEndpoints*, el cual posee un identificador para cada extremo del arco (origen y destino).

La acción de para añadir un arco entre dos nodos aplicando cierto coste consiste en primero comprobar la existencia previa de ambos nodos en el grafo, y en caso de que no se encuentren, se creará un puntero dentro del mismo, el cual será compartido entre los nodos involucrados y el arco.

```
const std::shared_ptr<Node> &Graph::searchNode(const unsigned int node_id) {
  if (const auto &iterator = nodes_.find(node_id); iterator != nodes_.end()) {
    return iterator->second;
  }
  nodes_[node_id] = std::make_shared<Node>(node_id);
  return nodes_[node_id];
}

void Graph::addArc(const ArcEndpoints end_points, const int cost) {
  const auto &from_node = searchNode(end_points.origin_id);
  const auto &to_node = searchNode(end_points.destination_id);
  arcs_.emplace_back(from_node, cost, to_node);
}
```

Código 4.3.3: Adición de un arco en el grafo.

Cabe destacar que se hará uso de otra clase que heredará de *OpsInstance*, incorporando a su vez la estructura de grafo previamente mencionada. En esta se añadirá el atributo *graphs_*, un contenedor de estos grafos generado a partir de las barras deslizantes. La sobrecarga del operador >> será reescrita para utilizar la implementada por el padre y asignar los arcos del grafo.

```
std::istream &operator>>(std::istream &input_stream, OpsInput &ops_input) {
  input_stream >> static_cast<OpsInstance &>(ops_input);
  ops_input.createGraphs();
  return input_stream;
}
```

Código 4.3.4: Sobrecarga del operador >> (OpsInput).

En cada grafo creado existirá como mínimo un arco desde el nodo 0 al nodo $n+1$, de forma que, en caso de no poder visitar algún objeto, por limitaciones de tiempo o por la ausencia de estos, el algoritmo no falle en ese grafo. Los objetos por visualizar tienen una transición a cada uno de ellos, omitiendo la transición a sí mismo, lo que forma un grafo dirigido fuertemente convexo. A su vez, cada nodo tiene una transición obligatoria desde el nodo inicial y otra hasta el nodo final.

```cpp
void OpsInput::addGraphArcs(const int graph_idx) {
  auto &graph = graphs_[graph_idx];
  const auto amount_of_objects = (unsigned int)getAmountOfObjects();
  const auto &objects_in_sliding_bar = getObjectsPerSlidingBar(graph_idx);
  graph.addArc({.origin_id = 0, .destination_id = amount_of_objects - 1}, 0);
  for (const auto &origin_id : objects_in_sliding_bar) {
    graph.addArc(
      {.origin_id = 0, .destination_id = origin_id},
      getTimeToProcess({0, origin_id})
    );
    graph.addArc(
      {.origin_id = origin_id, .destination_id = amount_of_objects - 1},
      getTimeToProcess({origin_id, amount_of_objects - 1})
    );
    for (const auto &destination_id : objects_in_sliding_bar) {
      if (origin_id == destination_id) { continue; }
      graph.addArc(
        {.origin_id = origin_id, .destination_id = destination_id},
        getTimeToProcess({origin_id, destination_id})
      );
    }
  }
}

void OpsInput::createGraphs() {
  const auto amount_of_sliding_bars = getAmountOfSlidingBars();
  graphs_.resize(amount_of_sliding_bars);
  for (auto graph_idx = 0; graph_idx < amount_of_sliding_bars; ++graph_idx) {
    addGraphArcs(graph_idx);
  }
}
```

Código 4.3.5: Creación de arcos en grafos.

Resumiendo, nuestra lectura de datos consiste en una clase base que contiene todos los atributos provenientes del archivo de entrada. Esta implementa una librería externa para poder extraer los datos del fichero, la clase hija hereda los atributos y añade una colección de grafos, uno por cada barra deslizante, compuestos a su vez en nodos, objetos a visualizar, y arcos, transiciones entre los mismos.

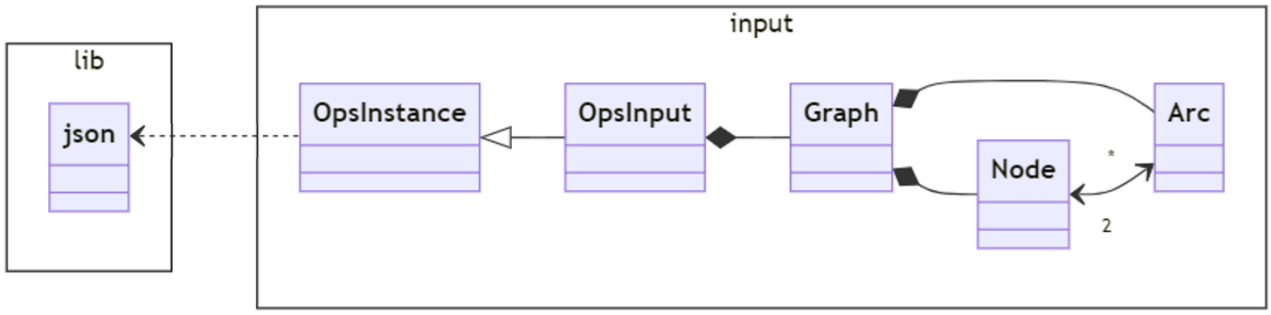


Figura 4.3.1: Diagrama UML de entrada de datos.

4.4 Representación de soluciones factibles

Una vez implementada la lectura de datos a partir de un archivo con formato *JSON*, el siguiente paso en el desarrollo es la manera en que representaremos las soluciones obtenidas a partir del resolutor y su comprobación de factibilidad. El segundo subproyecto estará ubicado en la subcarpeta *output* y se incluirá la dependencia al subproyecto *input* con el objetivo de tener acceso a los datos de instancia, más concretamente a los grafos generados.

Se hará uso de la clase *OpsOutput*, encargada de contener los valores que posteriormente se volcarán en un archivo de tipo *JSON*. Los atributos anteriormente mencionados son los que se listan a continuación:

- input : Entrada inicial con grafos.
- used_arcs : Conexiones entre nodos por grafo.
- observed_objects : Objetos observados en la solución.
- time_at_objects : Tiempo transcurrido desde el inicio del procesamiento hasta que el objeto se procesa.
- time_elapsed : Tiempo transcurrido en la resolución del problema.

Además de estos, está clase nos dirá el beneficio total de la instancia, el cual es calculado a partir de sumar el beneficio de esos objetos que han sido observados en la solución del problema.

De manera análoga a la clase *OpsInput*, el operador << fue sobrecargado para poder representar la solución en un *output stream* con el formato adecuado, generalmente creando un archivo con este contenido.

```
std::ostream &operator<<(std::ostream &output_stream, const OpsOutput &output) {
  return output_stream << nlohmann::json({
    {"x", output.used_arcs_},
    {"y", output.observed_objects_},
    {"s", output.time_at_objects_},
    {"profit", output.getTotalProfit()},
    {"time_elapsed", output.time_elapsed_}
  }).dump(2);
}
```

Código 4.4.1: Sobrecarga del operador << (OpsOutput).

Hemos desarrollado una clase *OpsError* que sirva de abstracción a la ofrecida por el estándar, *std::exception* [35]. Esta representa un error generado en la arquitectura de clases del proyecto, aunque su uso principal se resume al subproyecto *output,* y utiliza *std::format* [36] para la asignación del mensaje de error. La ubicación de esta clase la podemos encontrar en el subproyecto *error* (siendo el único archivo en este, sin contar el de configuración para *CMake*).

```cpp
class OpsError : public std::exception {
 public:
  template <typename... Args>
  explicit OpsError(std::format_string<Args...> format, Args &&...args) :
    message_ {std::format(format, std::forward<Args>(args)...)} {}

  [[nodiscard]] const char *what() const noexcept override {
    return message_.c_str();
  }

 private:
  std::string message_;
};
```

Código 4.4.2: Representación de error personalizado.

4.4.1 Asignación de variables

Para asignar cuales arcos han sido utilizados en la solución se proporcionará la variable del modelo matemático x (*used_arcs_* dentro del resolutor). Antes de la asignación, se comprobará que el modelo le ha dado un valor correcto ($x \in \{0,1\}$), en caso contrario, un error con esta información será lanzado.

```cpp
void OpsOutput::setUsedArcs(const std::vector<double> &used_arcs) {
  for (int graph_idx = 0; graph_idx < input_->getAmountOfSlidingBars();
       ++graph_idx) {
    const auto &graph = input_->getGraph(graph_idx);
    for (const auto &arc : graph.getArcs()) {
      const double value = std::round(used_arcs[arc.getId()]);
      if (isEqual(value, 0.0)) { continue; }
      if (!isEqual(value, 1.0)) {
        throw OpsError(
          "Invalid value for used arc ({} -> {}): {}. It must be 1 or 0.",
          arc.getOriginId(), arc.getDestinationId(), value
        );
      }
      used_arcs_[graph_idx][arc.getOriginId()] = arc.getDestinationId();
    }
  }
}
```

Código 4.4.3: Asignación de arcos usados.

En la asignación de objetos observados utilizaremos la variable y, proveniente del modelo matemático (*observed_objects_* en el resolutor). En primera instancia, asumimos que el primer y último objeto han sido visitados. Al igual que con la variable anterior, si alguno de los valores no es correcto ($y \in \{0,1\}$) se mostrará el error correspondiente.

```cpp
void OpsOutput::setObservedObjects(const std::vector<double> &observed_objects
) {
  if (observed_objects.empty()) {
    throw OpsError("The visited objects must be set.");
  }
  const auto amount_of_objects = input_->getAmountOfObjects();
  observed_objects_[0] = true;
  observed_objects_[amount_of_objects - 1] = true;
  for (int idx = 1; idx < amount_of_objects - 1; ++idx) {
    const double value = observed_objects[idx - 1];
    if (isEqual(value, 0.0)) { continue; }
    if (!isEqual(value, 1.0)) {
      throw OpsError(
        "Invalid value for object {}: {}. It must be 1 or 0.", idx, value
      );
    }
    observed_objects_[idx] = true;
  }
}
```

Código 4.4.4: Asignación de objetos observados.

Por último, para la asignación del tiempo transcurrido desde el inicio del procesamiento hasta que se procesa el objeto contamos con la variable s (*time_at_objects_* dentro del resolutor), de forma similar esta pertenece al modelo. La restricción que se aplica en este caso es que sus valores no pueden ser inferiores a 0.

```cpp
void OpsOutput::setTimeAtObjects(const std::vector<double> &time_at_objects) {
  const auto amount_of_objects = input_->getAmountOfObjects();
  time_at_objects_[0] = 0;
  if (time_at_objects.empty()) {
    throw OpsError("The accumulated time at each node must be set.");
  }
  for (auto idx = 1; idx < amount_of_objects; ++idx) {
    const double value = time_at_objects[idx - 1];
    if (value < 0) {
      throw OpsError(
        "Invalid value for object {} with time {}. It must be non-negative.",
        idx, value
      );
    }
    time_at_objects_[idx] = value / input_->getScalingFactor();
  }
}
```

Código 4.4.5: Asignación de tiempos acumulados hasta el procesamiento.

4.4.2 Comprobación de soluciones factibles

Como hemos explicado a lo largo de este trabajo, una solución es factible cuando se

cumplen ciertas restricciones de tiempo y los grafos de cada barra deslizante se definen de una forma concreta. Para que los grafos sean factibles, el nodo inicial debe ser visitado (además de ser el primero) en cada una de las barras, al igual de que el último nodo también debe ser visitado en todas las barras (además de ser el último). Los demás nodos deben tener la misma cantidad de arcos entrantes que arcos salientes y solo tendrán arcos si ese nodo es marcado como visitado. Para realizar estas comprobaciones contaremos la cantidad de veces que un arco entra a un nodo y las que un arco sale de un nodo. Este método lo encontramos definido como *countArrivesAndDepartures*.

```
std::pair<std::vector<int>, std::vector<int>>
OpsOutput::countArrivesAndDepartures() const {
  const auto amount_of_objects = input_->getAmountOfObjects();
  const auto amount_of_sliding_bars = input_->getAmountOfSlidingBars();
  std::vector<int> arrives_per_node(amount_of_objects, 0);
  std::vector<int> departures_per_node(amount_of_objects, 0);

  for (int graph_idx = 0; graph_idx < amount_of_sliding_bars; ++graph_idx) {
    const auto &graph = input_->getGraph(graph_idx);
    for (const auto &arc : graph.getArcs()) {
      const ArcEndpoints arc_endpoints {
        .origin_id = arc.getOriginId(), .destination_id = arc.getDestinationId()
      };
      if (arcWasUsed(graph_idx, arc_endpoints)) {
        arrives_per_node[arc_endpoints.destination_id]++;
        departures_per_node[arc_endpoints.origin_id]++;
      }
    }
  }
  return {arrives_per_node, departures_per_node};
}
```

Código 4.4.6: Conteo de arcos entrantes y salientes.

Posterior al conteo de arcos por cada uno de los nodos, se realizará la comprobación explicada anteriormente, y, en caso de que no tengan el resultado esperado, un error de tipo *OpsError* será generado.

```cpp
void OpsOutput::checkArcs() const {
  const auto amount_of_objects = input_->getAmountOfObjects();
  const auto amount_of_sliding_bars = input_->getAmountOfSlidingBars();
  const auto [arrives_per_node, departures_per_node] =
    countArrivesAndDepartures();
  if (departures_per_node[0] != amount_of_sliding_bars) {
    throw OpsError(
      "The first node must be visited in each sliding bar. It has {} visits.",
      departures_per_node[0]
    );
  }
  if (arrives_per_node[amount_of_objects - 1] != amount_of_sliding_bars) {
    throw OpsError(
      "The last node must be visited in each sliding bar. It has {} visits.",
      arrives_per_node[amount_of_objects - 1]
    );
  }
  for (std::size_t idx = 1; idx < amount_of_objects - 1; ++idx) {
    if (departures_per_node[idx] != arrives_per_node[idx]) {
      throw OpsError(
        "Node {} must have the same number of arrival and departure arcs. "
        "It has {} arrivals and {} departures.",
        idx, arrives_per_node[idx], departures_per_node[idx]
      );
    }
    if (observed_objects_[idx] != (arrives_per_node[idx] > 0)) {
      throw OpsError(
        "Node {} must be visited in order to have arrival / departure arcs.", idx
      );
    }
  }
}
```

Código 4.4.7: Comprobación de arcos.

La comprobación del tiempo factible de la solución consiste en asegurarnos que no existe algún objeto que se haya procesado después del tiempo máximo (se añadió un margen de tiempo *kMaxTimeMargin* con un valor de 0.01). Como en las anteriores comprobaciones, en

caso de no cumplirse, se lanzará un error con la información de este.

```cpp
void OpsOutput::checkTime() const {
  if (time_at_objects_.empty() || time_at_objects_[0] == -1) {
    throw OpsError("The time spent at each node must be set.");
  }
  const double real_maximum_time =
    double(input_->getTimeLimit()) / input_->getScalingFactor();
  for (const auto &time_at_object : time_at_objects_) {
    if (time_at_object > real_maximum_time + OpsOutput::kMaxTimeMargin) {
      throw OpsError(
        "The time spent at moment of visiting each node must be less than the "
        "maximum time. The maximum time is {} and the time spent is {}.",
        real_maximum_time, time_at_object
      );
    }
  }
}
```

Código 4.4.8: Comprobación de tiempo.

El proyecto entonces nos queda definido de la siguiente forma: tenemos una clase para representar la solución obtenida del modelo, sus variables son asignadas desde el resolutor y es capaz comprobar la condición de factibilidad, generando un error personalizado en caso contrario. Esta utiliza una librería externa para poder crear archivos con formato *JSON* y agrega la instancia inicial para poder realizar, entre otras cosas, el cálculo del valor de la solución.

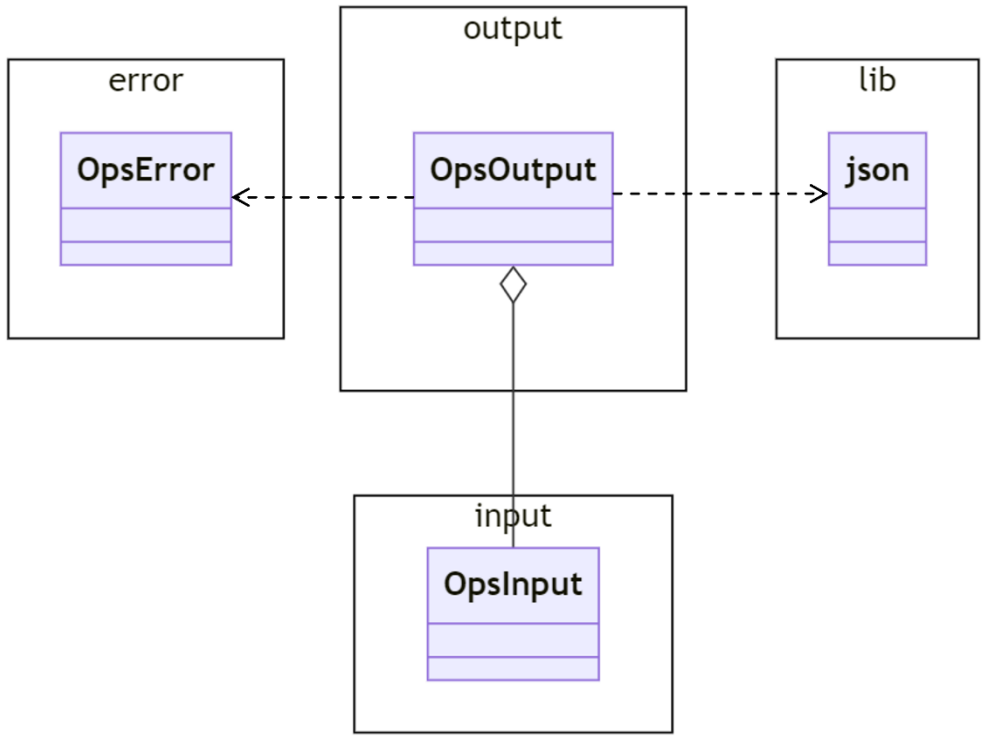


Figura 4.4.1: Diagrama UML de salida de datos.

4.5 Implementación de resolutor del OPS

Ya definida la lectura de instancias y representación de soluciones factibles, el último subproyecto que nos queda por definir es el del resolutor. Este se encontrará bajo el subdirectorio *models*, y en él se implementará el resolutor desarrollado. Se ha abstraído el almacenamiento de la entrada y la representación de soluciones del método de resolución, de forma que se ha definido una clase general *OpsSolver* con un atributo para la entrada y otro para la salida, además de uno extra utilizado para la medición del tiempo transcurrido en dar un resultado. Adicionalmente, se definen un conjunto de métodos protegidos que permitirán a la clase hija acceder de forma indirecta a los componentes mencionados.

La clase *OpsCplexSolver* será la que utilizaremos en este trabajo, esta heredará de la clase anterior e incluye, entre otros atributos, el resolutor *IloCplex* para la resolución del problema de programación lineal al que nos enfrentamos. Se definirán todas las variables explicadas en el modelo matemático, asignará la función objetivo y establecerá las restricciones explicadas en anteriores apartados. La clase posee la siguiente composición:

- *environment_*: El entorno donde se establecerá la memoria y los identificadores usados en el modelo.
- *cplex_*: Algoritmo utilizado para resolver el problema de programación lineal.
- *model_*: Modelo utilizado para representar el problema.
- *used_arcs_*: Vector binario que indica si cierto arco está siendo utilizado o no.
- *observed_objects_*: Vector binario indicando si un objeto ha sido observado.
- *time_at_objects_*: Vector de flotantes que guarda cuanto tiempo ha pasado desde el inicio de la observación hasta el momento que el objeto se observa.

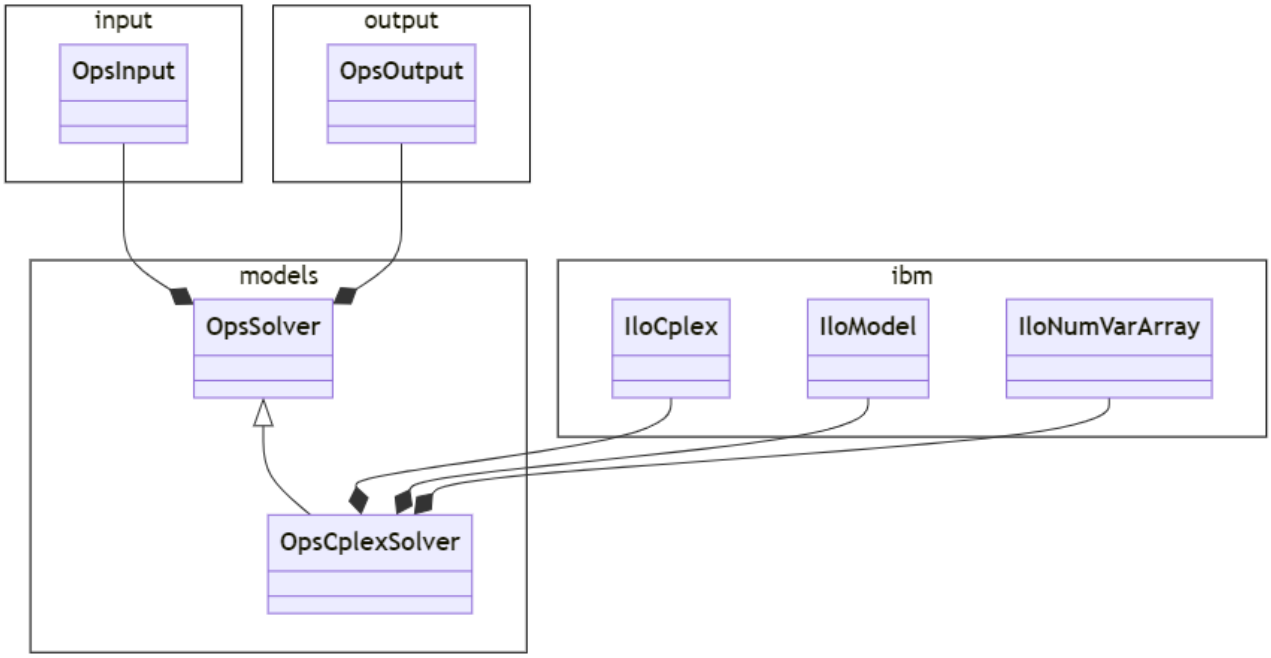


Figura 4.5.1: Diagrama UML del resolutor.

4.5.1 Especificación de variables

Para utilizar el modelo, el primer paso que debemos dar es especificar el formato de todas aquellas variables que se utilizaran en este. Cada una de las variables ya han sido explicadas anteriormente, sin embargo, se definirán de forma breve en el momento de mostrar su implementación en *C++*.

Para empezar, las variables dentro de y (*observed_objects_* en la clase) tomarán valores de 0 o 1 y se representarán en el modelo como "*y_j*" para $j \in J$. La configuración de esta implica recorrer cada uno de los objetos en la instancia inicial, sin contar el primero ni el último, puesto que no forman parte de la solución del problema.

```
void OpsCplexSolver::addYVariable() {
  const auto &input = getInput();
  for (int node_idx = 1; node_idx < input.getAmountOfObjects() - 1;
       ++node_idx) {
    observed_objects_.add(IloNumVar(
      environment_, 0, 1, IloNumVar::Bool, std::format("y_{}", node_idx).c_str()
    ));
  }
  model_.add(observed_objects_);
}
```

Código 4.5.1: Especificación de las variables "y".

Las siguientes a configurar se encuentran dentro de la variable s (*time_at_objects_* dentro de la clase). Estas tomarán valores entre 0 e infinito, aunque en la realidad no tomará valores superiores al tiempo máximo establecido, y serán representados como "*s_j*" para $j \in V$, esta vez sí contando los nodos añadidos al principio y al final.

```
void OpsCplexSolver::addSVariable() {
  const auto &input = getInput();
  for (int node_idx = 0; node_idx < input.getAmountOfObjects(); ++node_idx) {
    time_at_objects_.add(IloNumVar(
      environment_, 0, IloInfinity, IloNumVar::Float,
      std::format("s_{}", node_idx).c_str()
    ));
  }
  model_.add(time_at_objects_);
}
```

Código 4.5.2: Especificación de las variables "s".

Para finalizar esta sección, las últimas variables a especificar son las contenidas en x, las cuales tendrán un valor de 0 o 1 y serán representadas en el modelo con el formato "*x_k_i_j*" donde $(i, j, k) \in A$. Este grupo requiere la iteración dentro de cada grafo, para obtener los objetos encontrados en los extremos de cada arco.

```
void OpsCplexSolver::addXVariable() {
  const auto &input = getInput();
  for (int k = 0; k < input.getAmountOfSlidingBars(); ++k) {
    const auto graph = input.getGraph(k);
    for (const auto &arc : graph.getArcs()) {
      const auto &origin_id = arc.getOriginId();
      const auto &destination_id = arc.getDestinationId();
      used_arcs_.add(IloNumVar(
        environment_, 0, 1, IloNumVar::Bool,
        std::format("x_{}_{}_{}", k + 1, origin_id, destination_id).c_str()
      ));
    }
  }
  model_.add(used_arcs_);
}
```

Código 4.5.3: Especificación de las variables "x".

4.5.2 Configuración de función objetivo

Una vez concluido con la especificación de las variables utilizadas dentro del modelo matemático, asignaremos cual es la función objetivo que el algoritmo debe perseguir. En nuestro caso se trata de un problema de maximización de cantidad de objetos visualizados, más concretamente, se busca maximizar el valor obtenido a partir de sumar el beneficio por procesar aquellos objetos que son visibles en la solución generada.

```
void OpsCplexSolver::addObjective() {
  const auto &input = getInput();
  IloExpr expression(environment_);
  for (int node_idx = 1; node_idx < input.getAmountOfObjects() - 1;
       ++node_idx) {
    expression += input.getPriority(node_idx) * observed_objects_[node_idx - 1];
  }
  model_.add(IloMaximize(environment_, expression));
  expression.end();
}
```

Código 4.5.4: Configuración de función objetivo.

4.5.3 Configuración de restricciones

El último paso que nos queda para tener preparado el modelo es configurar aquellas restricciones que se apliquen a la hora de calcular una solución factible. Esta sección esencialmente especifica cuales son las características que debe tener una solución para que cumpla esa condición de factibilidad. Podemos dividir las restricciones en tres grandes grupos, uno para los arcos entrantes, otra para los arcos salientes y por último las limitaciones de tiempo

Como ya se ha mencionado anteriormente, si un nodo es visitado este solo puede serlo una vez por grafo (véase condición 6 y 7). Esto implica que el número de arcos entrantes a

un nodo debe ser igual a 1, con la excepción del nodo inicial, del cual no deben entrar ningún arco (condición 5). Esto se traduce en nuestro modelo como se muestra en el *Código 4.5.5*.

```
void OpsCplexSolver::addDeltaPlusConstraints(IloRangeArray &constraints) {
  const auto &input = getInput();
  for (auto k = 0; k < input.getAmountOfSlidingBars(); ++k) {
    const auto &graph = input.getGraph(k);
    for (const auto &origin_id : graph.getNodesId()) {
      IloExpr expression(environment_);
      const auto arcs_id = graph.getSuccessorsArcsId(origin_id);
      if (arcs_id.empty()) { continue; }
      for (const auto &arc_id : arcs_id) { expression += used_arcs_[arc_id]; }
      if (origin_id != 0) { expression -= observed_objects_[origin_id - 1]; }
      const double is_root_node = origin_id == 0 ? 1.0 : 0.0;
      constraints.add(IloRange(
        environment_, is_root_node, expression, is_root_node,
        std::format("deltaplus_{}_{}", k + 1, origin_id).c_str()
      ));
      expression.end();
    }
  }
}
```

Código 4.5.5: Restricciones de arcos de entrada.

De manera análoga a la anterior restricción, que un nodo sólo pueda ser visitado una vez por grafo también implica que el número de arcos salientes de un nodo debe ser igual a 1, considerando a su vez la excepción del nodo final, del que no debe salir ningún arco (condición 5). Este caso se ve encapsulado en el método *addDeltaMinusConstraints* de la clase *OpsCplexSolver*, el cual podemos ver ilustrado en el *Código 4.5.6*.

```
void OpsCplexSolver::addDeltaMinusConstraints(IloRangeArray &constraints) {
  const auto &input = getInput();
  const auto last_node_id = input.getAmountOfObjects() - 1;
  for (auto k = 0; k < input.getAmountOfSlidingBars(); ++k) {
    const auto &graph = input.getGraph(k);
    for (const auto &node_id : graph.getNodesId()) {
      IloExpr expression(environment_);
      const auto arcs_id = graph.getPredecessorsArcsId(node_id);
      if (arcs_id.empty()) { continue; }
      for (const auto &arc_id : arcs_id) { expression += used_arcs_[arc_id]; }
      if (node_id != last_node_id) {
        expression -= observed_objects_[node_id - 1];
      }
      const double is_last_node = node_id == last_node_id ? 1.0 : 0.0;
      constraints.add(IloRange(
        environment_, is_last_node, expression, is_last_node,
        std::format("deltaminus_{}_{}", k + 1, node_id).c_str()
      ));
      expression.end();
    }
  }
}
```

Código 4.5.6: Restricciones de arcos de salida.

La condición (8) define que el tiempo al procesar un objeto añadido al tiempo utilizado para transitar al siguiente objeto no puede exceder al tiempo en el que se procesa este último. Esta restricción puede parecer simple e intuitiva, sin embargo, al tratarse de un problema de programación lineal debemos linealizarla haciendo uso del valor *big-M*. Para ello se ha escogido el valor más grande entre el arco de mayor longitud y el tiempo límite establecido. La mayor cantidad de restricciones aplicadas al modelo se encuentran aquí, puesto que hará falta recorrer cada uno de los arcos de cada grafo generado.

```
void OpsCplexSolver::addMTZConstraints(IloRangeArray &constraints) {
  const auto &input = getInput();
  const int BIG_M = std::max((int)input.getMaxArc(), input.getTimeLimit()) + 1;
  for (auto k = 0; k < input.getAmountOfSlidingBars(); ++k) {
    const auto &graph = input.getGraph(k);
    for (const auto &arc : graph.getArcs()) {
      const auto &origin_id = arc.getOriginId();
      const auto &destination_id = arc.getDestinationId();
      IloExpr expression(environment_);
      expression = BIG_M * used_arcs_[arc.getId()] +
                   time_at_objects_[origin_id] -
                   time_at_objects_[destination_id];
      constraints.add(IloRange(
        environment_, -IloInfinity, expression,
        BIG_M - input.getTimeToProcess({origin_id, destination_id}),
        std::format("MTZ_{}_{}_{}", k + 1, origin_id, destination_id).c_str()
      ));
      expression.end();
    }
  }
}
```

Código 4.5.7: Restricciones para visitar un nodo.

Finalizando con el modelo, la condición (9) nos restringe la posibilidad de exceder el tiempo máximo. Como se puede apreciar en el *Código 4.5.8*, se establece a su vez otra restricción, que obliga al resolutor a empezar el procesamiento con un tiempo igual a 0.

```
void OpsCplexSolver::addLimitConstraints(IloRangeArray &constraints) {
  const auto &input = getInput();
  IloExpr start_time_expression(environment_);
  start_time_expression = time_at_objects_[0];
  constraints.add(IloRange(environment_, 0, start_time_expression, 0, "Limit0"));
  start_time_expression.end();

  IloExpr end_time_expression(environment_);
  end_time_expression = time_at_objects_[(long)input.getAmountOfObjects() - 1];
  constraints.add(IloRange(
    environment_, -IloInfinity, end_time_expression, input.getTimeLimit(),
    "Limit"
  ));
  end_time_expression.end();
}
```

Código 4.5.8: Restricciones de tiempo.

4.6 Ejecución del código

Se ha desarrollado un subproyecto dentro de la subcarpeta *main*, capaz de recibir argumentos por la línea de comando y resolver una instancia dada o un grupo de instancias

de cierta clase. El programa generará un archivo con la solución de la instancia, el cual estará ubicado en una ruta similar a la de archivo o clase inicial, a la vez que omitirá los registros generados por el resolutor. Para gestionar los argumentos, se ha incorporado la librería *InputParser*, que define un conjunto de opciones para poder configurar como se pasarán ciertos valores y las restricciones que deben cumplir.

Para cada argumento se establecerá una opción dentro de la clase *Parser*, utilizando el método *addOption*. Esta librería nos ofrece el atajo *addHelpOption* para asignar el argumento *--help* y *-h*, el cual, como cabría esperar, nos mostrará la forma correcta de ejecutar el programa.

Se proporcionará la capacidad de asignar un archivo con la instancia a evaluar a través del argumento *--input* o *-i*. Será una opción no obligatoria (puesto que también se puede pasar una clase) y tendrá como restricción que la ruta debe llevar a un archivo que exista.

```
.addOption([] {
  return input_parser::SingleOption("-i", "--input")
    .addDescription("Path to the input file to be processed")
    .addDefaultValue(std::string())
    .addConstraint<std::string>(
      [](const auto &value) -> bool {
        return value.empty() || std::filesystem::exists(value);
      },
      "The file must exist!"
    );
})
```

Código 4.6.1: Opción para obtener archivo a procesar.

De forma alternativa, el programa recibirá un conjunto de clases utilizando el argumento *-c* o *--classes*. Por el mismo motivo que en la anterior, será una opción no obligatoria, y podrá tomar los valores del set {“A”, “B”, “C”}, donde las clases no se especificarán más de una vez.

```
.addOption([] {
  return input_parser::CompoundOption("-c", "--classes")
    .addDescription("List of class to be processed")
    .addDefaultValue(std::vector<std::string>())
    .addConstraint<std::vector<std::string>>(
      [](const auto &values) -> bool {
        return std::ranges::all_of(values, [](const auto &value) {
          return value == "A" || value == "B" || value == "C";
        });
      },
      "The classes must be one of the following: A, B, C"
    )
    .addConstraint<std::vector<std::string>>(
      [](const auto &values) -> bool {
        return std::ranges::all_of(
          values,
          [values](const auto &value) -> bool {
            return std::ranges::count(values, value) == 1;
          }
        );
      },
      "The classes cannot be specified more than once"
    );
})
```

Código 4.6.2: Opción para obtener clases a procesar.

Se ha añadido una tercera opción con carácter opcional. Este será accesible a través del argumento *--tolerance o -t*, y representa que tanta tolerancia se le asignará al resolutor. En este caso simboliza cuánta es la diferencia máxima, entre la mejor posible solución y la actual solución, que se acepta antes de dar un resultado. Se ha añadido la restricción de que no puede tener valores menores o iguales a cero.

```
.addOption([] {
  return input_parser::SingleOption("-t", "--tolerance")
    .addDescription("Tolerance for the solver")
    .addDefaultValue(std::string("1e-4"))
    .toDouble()
    .transformBeforeCheck()
    .addConstraint<double>(
      [](const auto &value) -> bool { return value > 0; },
      "The tolerance must be greater than 0"
    );
})
```

Código 4.6.3: Opción para asignar la tolerancia del resolutor.

Se ha definido una serie de funciones, utilizadas solo en este programa principal puesto que no tienen cavidad fuera del subproyecto. Estas serán llamadas dependiendo si se proporciona una instancia concreta o una clase de instancias.

Para procesar una instancia se estableció una función que recibirá la ruta de los ficheros de entrada y salida, además de la tolerancia del resolutor, generando la solución de este en el archivo especificado y obviando los registros generados por la clase *IloCplex*.

```
struct PathConfig {
  std::string input_path;
  std::string output_path;
};

void processInstance(const PathConfig &path_config, const double tolerance) {
  std::ofstream output_os(path_config.output_path);
  std::stringstream string_stream;
  output_os << solve<emir::OpsCplexSolver>(
    path_config.input_path, tolerance, string_stream
  );
}
```

Código 4.6.4: Procesamiento de una instancia.

Por otro lado, el procesamiento de una clase de instancias consiste en recorrer toda la carpeta donde se encuentran esas instancias (en el proyecto se encuentran en la ruta *data/{class}/instances*) y resolverlas de manera secuencial. Los ficheros de salida generados se encontrarán en la carpeta *data/{class}/outputs* con el mismo nombre del fichero de entrada.

```
void processModelClass(const std::string &model_class, const double tolerance) {
  const auto input_folder = std::format("data/{}/instances", model_class);
  const auto output_folder = std::format("data/{}/outputs/", model_class);
  if (!fs::exists(output_folder)) { fs::create_directory(output_folder); }
  for (const auto &file : fs::directory_iterator(input_folder)) {
    std::cout << file.path() << '\n';
    processInstance(
      {.input_path = file.path(),
       .output_path = output_folder + file.path().filename().string()},
      tolerance
    );
  }
}
```

Código 4.6.5: Procesamiento de clase de instancias.

De forma que en el programa principal lo primero que haremos será crear el *Parser* con las opciones especificadas, comprobar que los argumentos proporcionados son correctos y procesar la instancia o clase según se indique.

```cpp
int secureMain(int argc, char *argv[]) {
  auto parser = createParser();
  try {
    parser.parse(argc, argv);
  } catch (const input_parser::ParsingError &e) {
    std::cerr << e.what() << '\n';
    return 1;
  }
  const auto &input_path = parser.getValue<std::string>("--input");
  const auto &classes = parser.getValue<std::vector<std::string>>("--classes");
  const auto tolerance = parser.getValue<double>("--tolerance");
  if (!input_path.empty()) {
    processInstance(
      {.input_path = input_path, .output_path = "solution.txt"}, tolerance
    );
  } else if (!classes.empty()) {
    for (const auto &model_class : classes) {
      processModelClass(model_class, tolerance);
    }
  }
  return 0;
}
```

Código 4.6.6: Programa principal.

4.7 Elaboración de experimento computacional

En la fase final del desarrollo de este trabajo se ha creado un ejecutable, diferente al programa principal, que permita comprobar que nuestro resolutor nos da el resultado correcto para cada instancia. Para ello se utilizará las generadas en la batería de problemas, comentado en el apartado 4.2, además de la librería *GoogleTest* para la creación de pruebas unitarias. Este subproyecto se encuentra localizado bajo el directorio *test*.

4.7.1 Pruebas unitarias

Para cada clase de instancias (A, B y C) se realizará una prueba unitaria. En esta se comprobará que el resolutor desarrollado no genera ningún tipo de error al comprobar la factibilidad de los datos de salida, además de que el valor final coincide con el esperado, el cual lo obtenemos a partir de un archivo con la solución del problema.

Como cabe esperar, las pruebas guardan muchas similitudes entre sí, siendo la única diferencia la clase con la que trabajar, es decir, el directorio con las instancias de entrada, por lo que se ha creado una función general que recibe cual es la clase por probar y esta se encargará de realizar las llamadas correspondientes a las *macros* de la librería de pruebas.

```cpp
void testModelClass(const std::string &model_class) {
  const auto input_folder = std::format("data/{}/instances", model_class);
  const auto solution_folder = std::format("data/{}/outputs/", model_class);
  const double tolerance = 1e-4;
  nlohmann::json solution;
  std::stringstream string_stream;
  for (const auto &entry : fs::directory_iterator(input_folder)) {
    std::ifstream solution_file(
      solution_folder + entry.path().filename().string()
    );
    emir::OpsCplexSolver solver(
      createFromFile<emir::OpsInput>(entry.path()), tolerance
    );
    solver.addLog(string_stream);
    ASSERT_NO_THROW(solver.solve());
    solution_file >> solution;
    EXPECT_EQ(solver.getProfit(), solution["profit"].get<double>());
    string_stream.str(std::string());
  }
}
```

Código 4.7.1: Comprobación general de clase de instancia.

Una vez definida esta función, crearemos las pruebas unitarias haciendo uso de esta, únicamente especificando la clase a probar.

```cpp
TEST(OpsTest, OneBandNeeded) {
  testModelClass("A");
}
TEST(OpsTest, TwoBandsNeeded) {
  testModelClass("B");
}
TEST(OpsTest, ThreeBandsNeeded) {
  testModelClass("C");
}
```

Código 4.7.2: Prueba unitaria por cada clase.

Al ejecutarlo, nos damos cuenta de que el resolutor no ha generado error alguno y nos ha proporcionado el resultado que se esperaba, por lo que podemos afirmar con confianza que funciona correctamente.

4.7.2 Análisis de gráficas

Para finalizar el experimento, podemos observar la relación que existe entre el tiempo medio de procesado de la instancia por clase, y la cantidad de objetos a observar. Como cabría esperar, a mayor nivel de sincronización requerido más será el tiempo necesario para encontrar la solución óptima de esa instancia. El mismo resultado lo podemos deducir a partir de la cantidad de objetos a visualizar, mientras más nodos tengamos en el grafo, más posibilidades hay que considerar y, por lo tanto, el tiempo de procesamiento aumenta. Esta relación la podemos ver reflejada en la *Figura 4.7.1*.

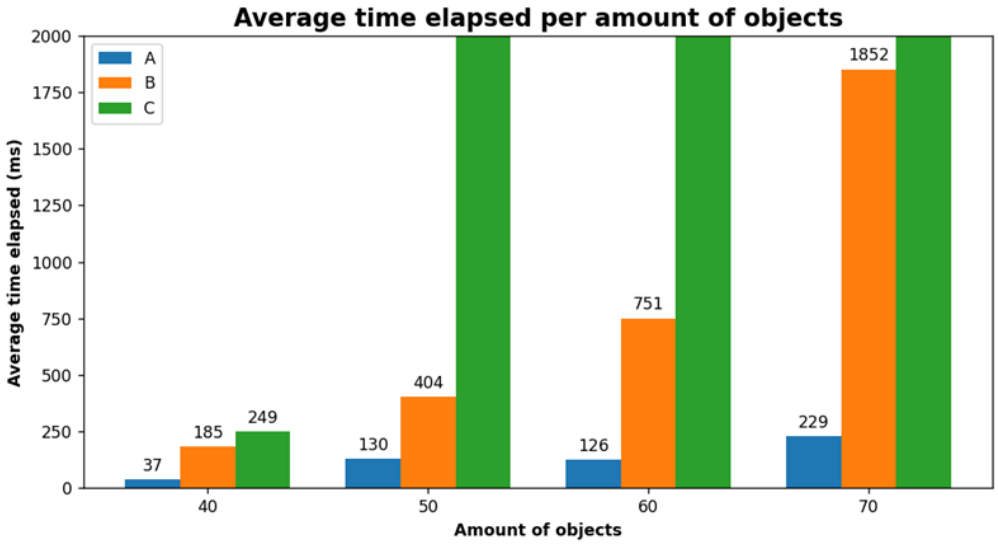


Figura 4.7.1: Tiempo de procesamiento por cantidad de objetos (milisegundos).

Para la clase C, medir el tiempo en magnitud de milisegundos se nos queda corto, por lo que la *Figura 4.7.2* simplemente escala la gráfica anterior a minutos para una mejor representación.

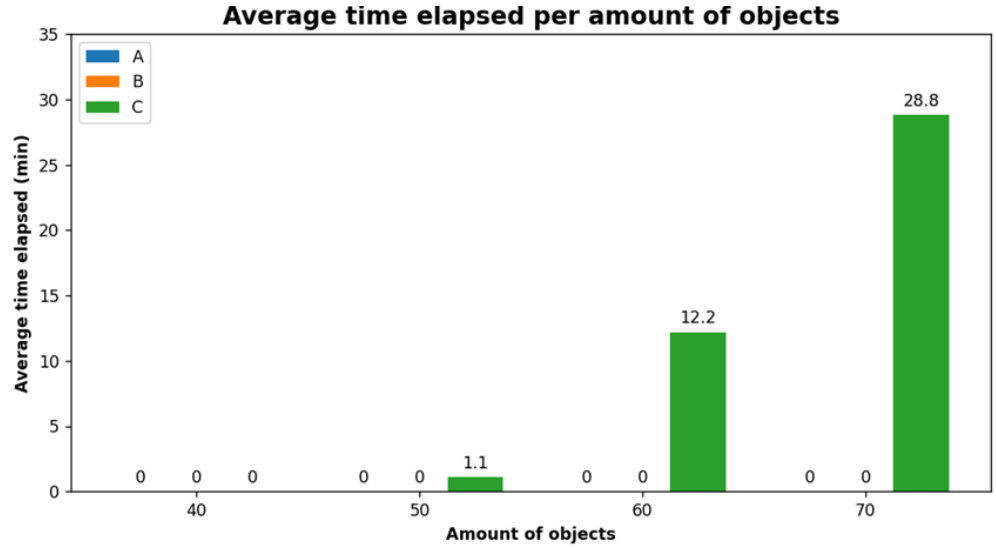


Figura 4.7.2: Tiempo de procesamiento por cantidad de objetos (minutos).

Podemos realizar el mismo experimento con el valor que se le asigna a la variable *alpha*, puesto que afecta directamente a la cantidad de objetos que se pueden visualizar. La relación comentada anteriormente vuelve a aparecer y con la misma proporción por clase.

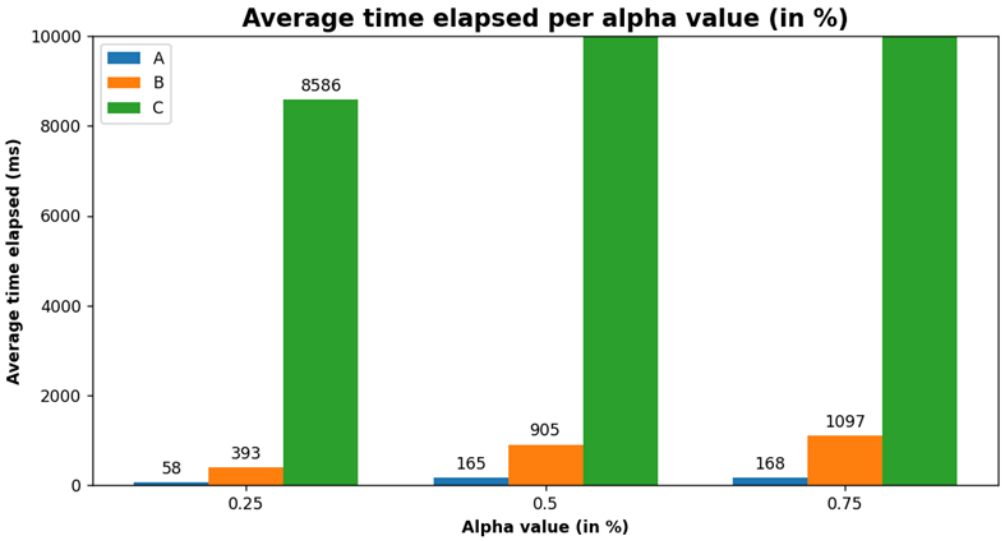


Figura 4.7.3: Tiempo de procesamiento por valor alfa (milisegundos).

De manera análoga, escalaremos la anterior figura para poder observar los valores que toma la clase C.

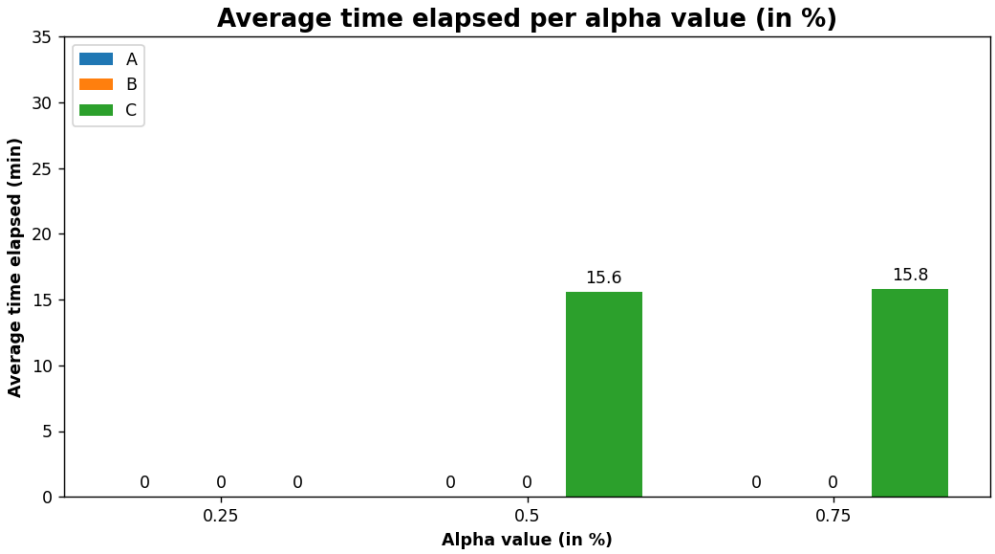


Figura 4.7.4: Tiempo de procesamiento por valor alfa (minutos).

Podemos hacer énfasis en cada una de estas instancias y graficar solo las que tienen características similares. Si comparamos aquellas que sean de la misma clase y posean la misma cantidad de objetos, podemos observar la evolución del tiempo de procesamiento en función del aumento del valor alfa. Cabe destacar que para las instancias que no se encontró solución óptima ($time\_elapsed = 3.6 * 10^6$) no tienen ningún valor en la gráfica, más del saber que no se pudo resolver en el tiempo máximo propuesto, por lo que solo se representarán valores inferiores al mencionado.

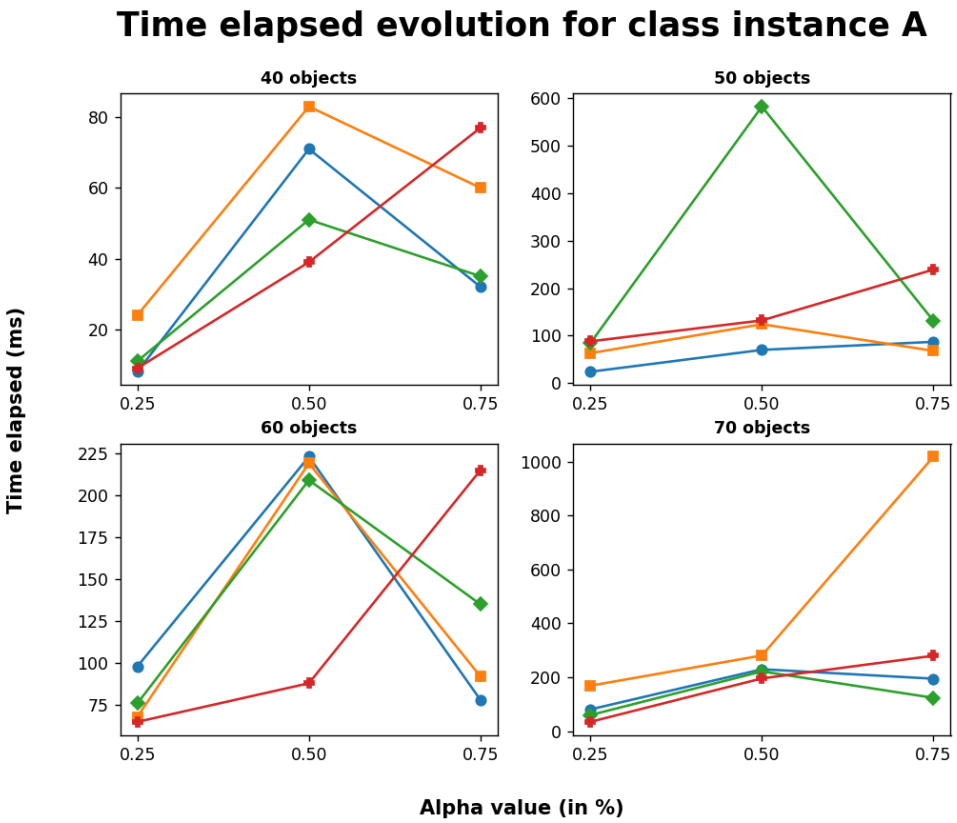


Figura 4.7.5: Evolución del tiempo en función del valor alfa (clase A).

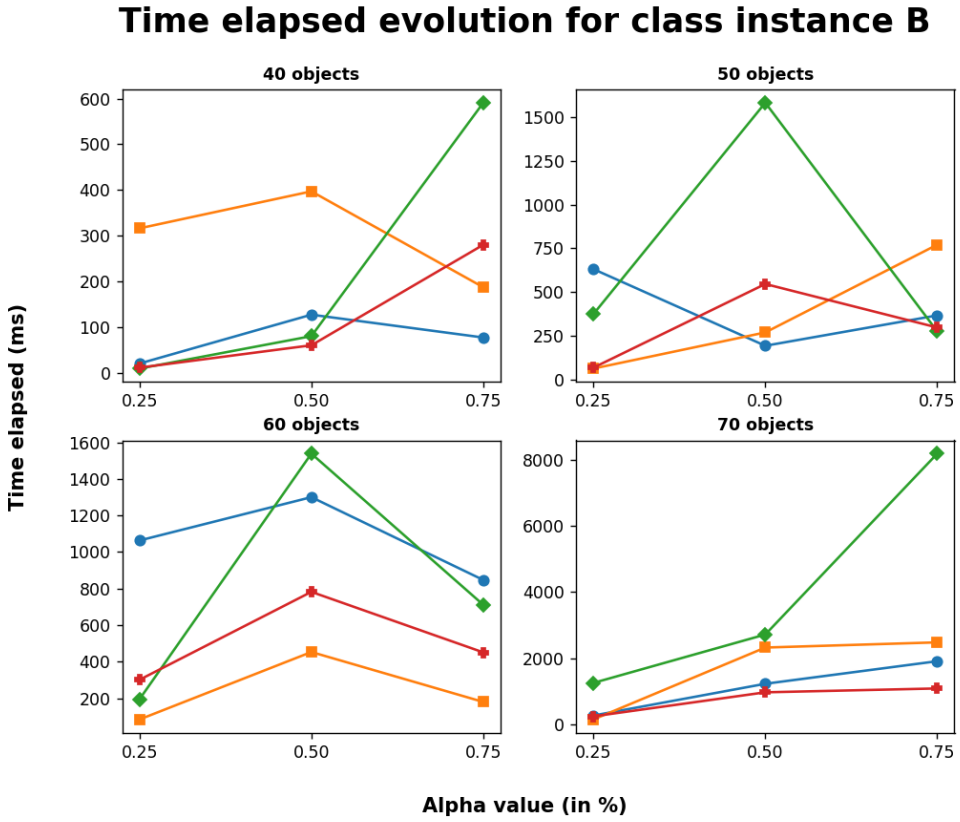


Figura 4.7.6: Evolución del tiempo en función del valor alfa (clase B).

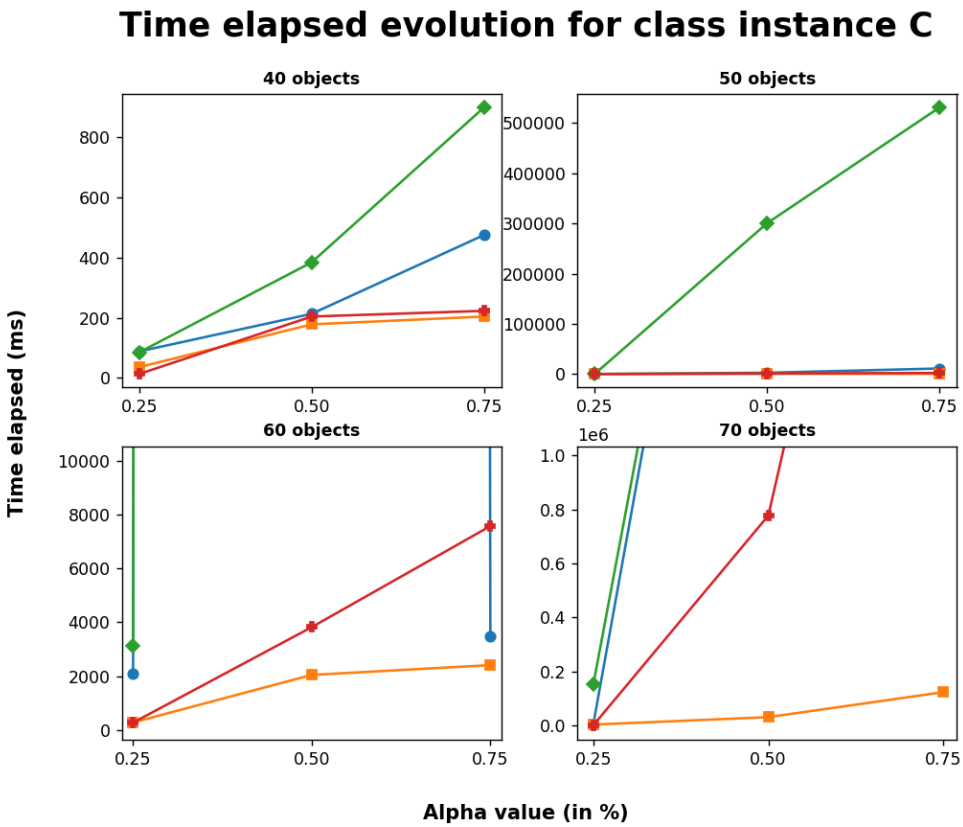


Figura 4.7.7: Evolución del tiempo en función del valor alfa (clase C).

Estos datos reflejan un patrón que se repite en la mayoría de las instancias, independientemente de su clase o cantidad de objetos: las instancias con un valor alfa de 0.25 son las más fáciles de resolver, de hecho, nuestro resolutor ha sido capaz de dar la solución óptima en cada una de estas, tardando como máximo dos minutos y medio para una instancia de clase C con 70 objetos a visualizar (véase *Figura 4.7.7*). Este resultado puede ser originado a causa de la limitación tan extensa de tiempo que se impone sobre la misma, puesto que elimina una cantidad importante de posibilidades que se podrían haber considerado si se contara con más tiempo. En cambio, para el resto de los valores de alfa, este comportamiento no es constante, dependiendo de la instancia, se puede tomar más tiempo en resolver para un alfa de 0.50 que para uno de 0.75 o viceversa. Estos a su vez varían en función de la cantidad de objetos y clase, por lo que su interpretación no es tan trivial.

Capítulo 5 Conclusiones y líneas futuras

A lo largo del desarrollo de este trabajo se ha estudiado un problema de optimización que surge durante la gestión de un instrumento, denominado EMIR, para el telescopio GRANTECAN, que puede efectuar hasta cincuenta y cinco observaciones simultáneas de objetos celestes. Dicho problema de optimización es el resultado de combinar otro conocido problema, denominado *The Orienteering Problem* con restricciones de sincronización. De ahí que lo hayamos denominado *Orienteering Problem with Synchronization* u *OPS*.

Para describir formalmente el problema, lo hemos modelizado de forma compacta. Es decir, utilizando un número de restricciones polinomial con respecto al tamaño de la entrada, frente a otras alternativas que utilizan un número no polinomial de restricciones. De esta manera no necesitaremos abordar la resolución del problema con técnicas algorítmicas iterativas más complejas, como *branch-and-cut*.

Precisamente, con el fin de hacer dicho modelo compacto, hemos modelizado las restricciones de sincronización utilizando el conocido método de la M-grande o *big-M*, a través de una variación de las restricciones de *Miller–Tucker-Zemlim*. No obstante, estamos al corriente de que este método, en ocasiones, puede dar lugar a un comportamiento no deseado del algoritmo de resolución, produciendo largos procesos de ramificación, y, en consecuencia, ralentizando la obtención de una solución. De forma colateral, estas restricciones también evitarán la formación de *subtours* en la solución.

Una vez descrito el problema, hemos implementado un algoritmo exacto trasladando nuestra formulación a un resolutor general para problemas MIP conocido como CPLEX. CPLEX dispone de *APIs* que nos permiten conectar nuestra aplicación en *C++* con el resolutor, permitiendo la elaboración de una aplicación compleja que resuelva el *OPS* para diferentes entradas.

El comportamiento de nuestro algoritmo bajo diferentes características de los datos de entrada ha sido evaluado durante una extensa experiencia computacional, que pone de manifiesto las dificultades de nuestro algoritmo a medida que aumenta el peso de la sincronización.

El proyecto ha resultado ser mucho más extenso y complejo de lo esperado, lo que llevó a una dedicación profunda del tema. Se buscó optimizar lo más posible la entrada y salida de datos, aunque claramente el cuello de botella se encontraba en el resolutor. Sin embargo, se logró acotar significativamente el tiempo de procesado, con respecto a implementaciones tempranas de este.

Trabajar con *C++* en un proyecto de tal magnitud ha servido para contemplar nuevamente la importancia de los lenguajes de bajo nivel por su capacidad de procesamiento de operaciones matemáticas.

Algunas propuestas para el continuo uso de este trabajo o futuras investigaciones podrían

ser el uso de una interfaz gráfica (por ejemplo Qt [37]) para visualizar los grafos obtenidos a partir de la solución factible; la división del problema en instancias más pequeñas, separadas tanto verticalmente como horizontalmente, de forma que se mantenga el requerimiento de sincronización pero con un número menor de objetos y/o barras; o la incorporación de programación multihilos para la resolución del problema, dividiendo el trabajo de manera similar a la propuesta anterior. Otra alternativa para mejorar el proyecto sería utilizar un algoritmo aproximado, que sea capaz de encontrar la mejor solución local y se le pueda aplicar un algoritmo de búsqueda local para mejorar la solución.

Capítulo 6 Summary and Conclusions

Throughout the development of this work, we have studied an optimization problem that arises during the management of an instrument, called EMIR, for the GRANTECAN telescope, which can perform up to fifty-five simultaneous observations of celestial objects. This optimization problem is the result of combining another well-known problem, called The Orienteering Problem with synchronization constraints. Hence, we have called it the Orienteering Problem with Synchronization or OPS.

To formally describe the problem, we have modelled it in a compact form. That is, using a polynomial number of constraints with respect to the size of the input, as opposed to other alternatives that use a non-polynomial number of constraints. In this way we will not need to approach the resolution of the problem with more complex iterative algorithmic techniques, such as branch-and-cut.

Precisely to make such a model compact, we have modelled the synchronization constraints using the well-known big-M method, through a variation of the Miller-Tucker-Zemlim constraints. However, we are aware that this method, at times, can lead to undesired behavior of the solving algorithm, producing long branching processes, and consequently slowing down the obtaining of a solution. Collaterally, these restrictions will also prevent the formation of subtours in the solution.

Having described the problem, we have implemented an exact algorithm by translating our formulation to a general solver for MIP problems known as CPLEX. CPLEX has APIs that allow us to connect our C++ application with the solver, allowing the development of a complex application that solves the OPS for different inputs.

The behavior of our algorithm under different input characteristics has been evaluated during extensive computational experience, which highlights the difficulties of our algorithm as the synchronization weight increases.

The project has turned out to be much more extensive and complex than expected, which led to a thorough dedication to the subject. The aim was to optimize as much as possible the input and output of data, although the bottleneck was clearly in the solver. However, the processing time was significantly reduced compared to earlier implementations of the solver.

Working with C++ in a project of this magnitude has served to contemplate once again the importance of low-level languages for their ability to process mathematical operations.

Some proposals for the continued use of this work or future research could be the use of a graphical interface (e.g. Qt [37]) to visualize the graphs obtained from the feasible solution, the division of the problem into smaller instances, separated both vertically and horizontally, so that the synchronization requirement is maintained but with a smaller number of objects and/or bars, the incorporation of multithreaded programming for the resolution of the

problem, dividing the work in a similar way to the previous proposal. Another alternative to improve the project would be to use an approximate algorithm, which can find the best local solution and can apply a local search algorithm to improve the solution.

Capítulo 7 Presupuesto

En esta sección se detallarán los costes del trabajo realizado, contabilizando las horas trabajadas, así como el precio de las licencias utilizadas. Cabe destacar que la mayoría de las tecnologías incluidas son de uso libre, por lo que no se agregarán al conteo. Como la asignatura contempla en la guía docente un total de 300 horas, esta será la cantidad con la que trabajaremos.

Concepto	**Coste por hora**	**Horas**	**Coste total**
Diseño del modelo matemático	20	70	1400€
Implementación del resolutor	20	218	4360€
Documentación	20	12	240€
Licencia mensual de IBM ILOG CPLEX STUDIO	-	-	316€
	Total	300	6316€

Tabla 1: Presupuesto planteado.

Apéndice A

Repositorio GitHub

En el siguiente enlace se encuentra el repositorio de GitHub del código utilizado y expuesto en este trabajo.

https://github.com/gianluisdiana/OpsCplex

Bibliografía

[1] G. L. Bolívar Diana, «Reddit: Argument parser to modern C++,» January 2024. [En línea]. Available: https://www.reddit.com/r/cpp/comments/1942mei/argument_parser_to_modern_c/.

[2] Instituto Astrofísico de Canarias, «EMIR,» [En línea]. Available: https://www.gtc.iac.es/instruments/emir/.

[3] A. Gunawan, H. C. Lau y P. Vansteenwegen, «Orienteering Problem: A survey of recent variants, solution approaches and applications,» *European Journal of Operational Research,* vol. 255, nº 2, pp. 315-332, 2016.

[4] T. Tsiligiridis, «Heuristic Methods Applied to Orienteering,» *Journal of the Operational Research Society,* vol. 35, nº 9, pp. 797-809, 1984.

[5] B. L. Golden, L. Levy y R. Vohra, «The orienteering problem,» *Naval Research Logistics (NRL),* vol. 34, nº 3, pp. 307-318, 1987.

[6] S. E. Butt y T. M. Cavalier, «A heuristic for the multiple tour maximum collection problem,» *Computers & Operations Research,* vol. 21, nº 1, pp. 101-111, 1994.

[7] I.-M. Chao y B. L. Golden, «The team orienteering problem,» *European Journal of Operational Research,* vol. 88, nº 3, pp. 464-474, 1996.

[8] M. Drexl, «Synchronization in Vehicle Routing---A Survey of VRPs with Multiple Synchronization Constraints,» *Transportation Science,* vol. 46, nº 3, pp. 297-316, 2012.

[9] J. Desrosiers, Y. Dumas, M. M. Solomon y F. Soumis, «Time constrained routing and scheduling,» de *Handbooks in Operations Research and Management Science*, vol. 8, Elsevier Science B.V., 1995, pp. 35-139.

[10] A. Lim, B. Rodrigues y L. Song, «Manpower Allocation with Time

Windows,» *The Journal of the Operational Research Society,* vol. 55, nº 11, pp. 1178-1186, 2004.

[11] Y. Li y A. Lim, «Manpower allocation with time windows and job-teaming constraints,» *Naval Research Logistics,* vol. 52, nº 4, pp. 302-311, 2005.

[12] A. Dohn, E. Kolind y J. Clausen, «The manpower allocation problem with time windows and job-teaming constraints: A branch-and-price approach,» *Computers & Operations Research,* vol. 36, nº 4, pp. 1145-1157, 2009.

[13] C. E. Cortés, M. Matamala y C. Contardo, «The pickup and delivery problem with transfers: Formulation and a branch-and-cut solution method,» *European Journal of Operational Research,* vol. 200, nº 3, pp. 711-724, 2010.

[14] S. Gelareh, R. Merzouki, K. McGinley y R. Murray, «Scheduling of Intelligent and Autonomous Vehicles under pairing/unpairing collaboration strategy in container terminals,» *Transportation Research Part C: Emerging Technologies,* vol. 33, pp. 1-21, 2013.

[15] Z. Luo, H. Qin, W. Zhu y A. Lim, «Branch-and-price-and-cut for the manpower routing problem with synchronization constraints,» *Naval Research Logistics (NRL),* vol. 63, nº 2, pp. 138-171, 2016.

[16] R. K. Ahuja, T. L. Magnanti y J. B. Orlin, «Network flows: theory, algorithms, and applications,» 1988. [En línea]. Available: https://www.dl.behinehyab.com/Ebooks/NETWORK/NET005_354338_www.behinehyab.com.pdf.

[17] M. A. A. Martínez, J.-F. Cordeau, M. Dell'Amico y M. Iori, «A Branch-and-Cut Algorithm for the Double Traveling Salesman Problem with Multiple Stacks,» 20 December 2011. [En línea]. Available: https://doi.org/10.1287/ijoc.1110.0489.

[18] B. Stroustrup, «Cpp Reference,» [En línea]. Available: https://en.cppreference.com/w/.

[19] Kitware, «CMake documentation,» [En línea]. Available:

https://cmake.org/documentation/.

[20] Kitware, «CMake tutorial,» [En línea]. Available: https://cmake.org/cmake/help/latest/guide/.

[21] IBM, «ILO CPLEX Optimization Studio manual,» [En línea]. Available: https://www.ibm.com/docs/en/icos/22.1.1?topic=optimizers-cplex-c-reference-manual.

[22] G. L. Bolívar Diana, «GitHub: InputParser,» 2024. [En línea]. Available: https://github.com/gianluisdiana/InputParser.

[23] Nlohmann, «GitHub: json,» [En línea]. Available: https://github.com/nlohmann/json.

[24] Valgrind Developers, «Valgrind,» [En línea]. Available: https://valgrind.org.

[25] Google, «GitHub: GoogleTest,» [En línea]. Available: https://github.com/google/googletest.

[26] Google, «GoogleTest User's Guide,» [En línea]. Available: https://google.github.io/googletest/.

[27] Matepek, «GitHub: vscode-catch2-test-adapter,» [En línea]. Available: https://github.com/matepek/vscode-catch2-test-adapter.

[28] LLVM, «Clang-Format documentation,» [En línea]. Available: https://clang.llvm.org/docs/ClangFormat.html.

[29] LLVM, «Clang-Tidy documentation,» [En línea]. Available: https://clang.llvm.org/extra/clang-tidy/.

[30] Microsoft, «MSBuild documentation,» 2024. [En línea]. Available: https://learn.microsoft.com/en-us/visualstudio/msbuild/?view=vs-2022.

[31] Microsoft, «How to install Linux on Windows with WSL,» [En línea]. Available: https://learn.microsoft.com/en-us/windows/wsl/install.

[32] D. Nek, «How to Upgrade GCC on Ubuntu,» [En línea]. Available: https://webhostinggeeks.com/howto/how-to-upgrade-gcc-on-ubuntu/.

[33] IBM, «IBM ILOG CPLEX Optimization Studio,» [En línea]. Available:

https://www.ibm.com/es-es/products/ilog-cplex-optimization-studio?mhsrc=ibmsearch_a&mhq=ILOG%20CPLEX%20Optimization%20Studio.

[34] J. Riera Ledesma, «GitHub: OPS-Benchmark,» [En línea]. Available: https://github.com/RieraULL/OPS-Benchmark.

[35] CppReference, «std::exception,» [En línea]. Available: https://en.cppreference.com/w/cpp/error/exception.

[36] CppReference, «std::format,» [En línea]. Available: https://en.cppreference.com/w/cpp/utility/format/format.

[37] Qt Group, «Qt,» 2024. [En línea]. Available: https://www.qt.io.

[38] Ttroy50, «Github: cmake-examples,» [En línea]. Available: https://github.com/ttroy50/cmake-examples.

[39] Visual Paradigm Online, «¿Cuáles Son Los Seis Tipos De Relaciones En Los Diagramas De Clases UML?,» 9 February 2022. [En línea]. Available: https://blog.visual-paradigm.com/es/what-are-the-six-types-of-relationships-in-uml-class-diagrams/.

[40] Mermaid, «Mermaid Live Editor,» [En línea]. Available: https://mermaid.live.