

UNIVERSIDAD DE LA LAGUNA  
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA



PROYECTO FINAL DE CARRERA

**CSUOptimizer:**  
**Optimización de apuntados en**  
**el proyecto EMIR**

Pedro Orlando Hernández Martín

LA LAGUNA, a 16 de julio de 2013



# Preliminares

D. Francisco de Sande González, profesor de la Escuela Técnica Superior de Ingeniería Informática, adscrito al Departamento de Estadística I. O. y Computación

## Certifica

Que la presente memoria titulada

*CSUOptimizer: Optimización de apuntados en el proyecto EMIR*

Ha sido realizada bajo su dirección por D. Pedro Orlando Hernández Martín, y constituye su Proyecto para optar al grado de Ingeniero en Informática por la Universidad de La Laguna.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos que haya lugar, firman la presente en La Laguna a 16 de julio de 2013.

Fdo: Francisco de Sande González



*La calidad nunca es un accidente, es siempre  
el resultado de un esfuerzo inteligente*

John Ruskin

*A mis padres y a Nazaret, por no cansarse nunca de animarme*





# Agradecimientos

Quiero agradecer a mi familia y amigos, por todo el apoyo que me han brindado, especialmente a mis padres, que nunca han dejado de creer en mi, a mi hermano, por ayudarme con las imágenes del documento, y a Nazaret, que siempre ha tenido tiempo para escuchar mis quejas y darme ánimos para continuar. Sin ellos me habría dado por vencido.

También me gustaría agradecer el trabajo realizado a Francisco de Sande, cuya guía ha resultado imprescindible para poder caminar por un sendero desconocido y lleno de dudas, y cuya exigencia ha conseguido sacar lo mejor de mí.



# Índice general

<b>1. Motivación</b>	<b>19</b>
1.1. Problema a tratar . . . . .	23
1.1.1. Beam Switching . . . . .	24
<b>2. Objetivos</b>	<b>27</b>
<b>3. Tecnologías y herramientas relacionadas</b>	<b>29</b>
3.1. Parsers XML . . . . .	29
3.1.1. Xerces-C++ . . . . .	30
3.1.2. Mini-XML . . . . .	31
3.1.3. TinyXML . . . . .	31
3.1.4. PugiXML . . . . .	31
3.1.5. RapidXML . . . . .	32
3.1.6. Libxml++ . . . . .	32
3.2. Allegro . . . . .	33
3.3. Algoritmos de clustering . . . . .	34
3.3.1. K-means . . . . .	35
3.3.2. DBSCAN . . . . .	36
3.4. Heurísticas . . . . .	39
3.4.1. GRASP . . . . .	40
3.4.1.1. Algoritmo GRASP . . . . .	41
3.4.1.2. Algoritmo de fase de construcción del GRASP. . . . .	41
3.4.1.3. Algoritmo de búsqueda local. . . . .	41
3.4.1.4. Algoritmo voraz. . . . .	42

<b>4. Algoritmos</b>	<b>43</b>
4.1. Clustering . . . . .	43
4.2. Grasp . . . . .	45
4.3. Algoritmo constructivo . . . . .	46
4.3.1. Fase 1 . . . . .	47
4.3.2. Fase 2 . . . . .	50
<b>5. La Aplicación CSUOptimizer</b>	<b>53</b>
5.1. Repositorio de código . . . . .	54
5.2. Instalación . . . . .	54
5.2.1. Dependencias . . . . .	54
5.2.1.1. make . . . . .	54
5.2.1.2. Compilador de C++ . . . . .	54
5.2.1.3. libxml++ . . . . .	55
5.2.1.4. Doxygen . . . . .	55
5.2.1.5. LaTeX . . . . .	56
5.2.1.6. Allegro . . . . .	56
5.2.2. Instalación . . . . .	58
5.3. CSUOptimizer . . . . .	58
5.3.1. La clase Config . . . . .	58
5.3.2. La clase Element . . . . .	59
5.3.2.1. Atributos de Element . . . . .	59
5.3.2.2. Métodos relevantes de Element . . . . .	60
5.3.3. La clase CSU . . . . .	60
5.3.3.1. Atributos de CSU . . . . .	60
5.3.3.2. Métodos relevantes de CSU . . . . .	63
5.3.4. La clase Parser . . . . .	68
5.3.4.1. Atributos de Parser . . . . .	72
5.3.4.2. Métodos relevantes de Parser . . . . .	74
5.3.5. La clase Dbscan . . . . .	74
5.3.5.1. Atributos de Dbscan . . . . .	74
5.3.5.2. Métodos relevantes de Dbscan . . . . .	74
5.3.6. Algoritmos principales . . . . .	75
5.4. Modo de empleo de CSUOptimizer . . . . .	77
5.5. Herramientas de apoyo . . . . .	78

5.5.1. “Editor de mapas” . . . . .	79
5.5.2. Conversor . . . . .	79
<b>6. Resultados</b>	<b>81</b>
6.1. Ejemplos sintéticos . . . . .	82
6.1.1. Ejemplo: <code>exacto.xml</code> . . . . .	82
6.1.2. Ejemplo: <code>4_0solapes.xml</code> . . . . .	83
6.1.3. Ejemplo: <code>4_2solapes.xml</code> . . . . .	84
6.1.4. Ejemplo: <code>denso.xml</code> . . . . .	85
6.1.5. Ejemplo: <code>denso2000.xml</code> . . . . .	85
6.1.6. Ejemplo: <code>disperso.xml</code> . . . . .	87
6.1.7. Ejemplo: <code>clusters.xml</code> . . . . .	88
6.2. Ejemplos reales . . . . .	88
6.2.1. Ejemplo: <code>real1.xml</code> . . . . .	89
6.2.2. Ejemplo: <code>real2.xml</code> . . . . .	91
6.2.3. Ejemplo: <code>cluster1.xml</code> . . . . .	92
6.2.4. Ejemplo: <code>cluster2.xml</code> . . . . .	92
6.2.5. Ejemplo: <code>cluster3.xml</code> . . . . .	94
<b>7. Conclusiones y Trabajos Futuros</b>	<b>95</b>
<b>A. Códigos</b>	<b>99</b>
A.1. <code>colisiones</code> . . . . .	99
A.2. <code>grasp</code> . . . . .	101
A.3. Salida <code>exacto.xml</code> . . . . .	103
<b>Bibliography</b>	<b>106</b>
<b>Índice de figuras</b>	<b>109</b>
<b>Índice de tablas</b>	<b>111</b>
<b>Índice de listados</b>	<b>112</b>



# Prólogo

En este documento se refleja el trabajo realizado durante el tiempo que ha llevado completar este Proyecto de Final de Carrera (PFC) de los estudios en Ingeniería en Informática cursados en la Escuela Técnica Superior de Ingeniería Informática (ETSII) de la Universidad de La Laguna (ULL). El Proyecto Final de Carrera se ha desarrollado en torno a un elemento del proyecto de investigación EMIR y está destinado a solventar un problema real con el que se encuentran los investigadores a la hora de utilizar de manera óptima los recursos de los que disponen.

En los capítulos que componen esta memoria se intenta sintetizar el trabajo realizado, mediante la aplicación de los conceptos aprendidos y la práctica adquirida durante toda la carrera en la ETSII de la ULL. El objetivo de este documento no trata sólo de recoger la memoria del PFC sino de servir, a su vez, como manual de usuario de la aplicación desarrollada.

El Espectrógrafo Multiobjeto InfraRojo (EMIR) [1], es un instrumento de apoyo para el proyecto GOYA [2] (*Galaxias: Orígenes y Evolución a Alto z*) desarrollado por el Instituto de Astrofísica de Canarias (IAC) [3]. GOYA es un programa científico e instrumental diseñado para el Gran Telescopio de Canarias (GTC) [4], liderado por el Instituto de Astrofísica de Canarias (IAC) y situado en La Palma. El principal objetivo científico es caracterizar la población de galaxias durante la época de máxima formación estelar en la historia del universo. Según estudios recientes, esta época crítica ocurrió cuando el universo tenía un 10-40% de su edad actual, lo que corresponde a desplazamientos al rojo  $1 < z < 2$ . A estos desplazamientos al rojo, la ventana óptica - la región del espectro que ha sido estudiada en mayor profundidad en las galaxias cercanas - está desplazada hacia el infrarrojo cercano (1-2.5 micras). Los principales estudios espectroscópicos por debajo de 1.8 micras están siendo planeados actualmente para investigar

las propiedades ópticas de las galaxias en el sistema de reposo y la formación estelar global del universo hasta  $z=1$ . Se propone llevar a cabo un estudio comprensivo de las galaxias con  $2 < z < 3$ , que incluya: morfología, estructura, cinemática, población estelar, tasa de formación estelar, metalicidad, luminosidad y funciones de masa, agrupamiento en cúmulos, y estructura a gran escala. El objetivo es entender la naturaleza de estas galaxias distantes y evaluar su papel en la historia de la formación estelar del universo, comparando directamente sus propiedades ópticas en el sistema de reposo con las de la población cercana. GOYA será el primer estudio importante que extenderá estas investigaciones al universo a alto  $z$ .

EMIR es un cámara de gran campo y espectrógrafo multiobjeto de resolución intermedia en el infrarrojo cercano para el telescopio GTC. Está equipado, entre otros, con tres subsistemas de alta tecnología de última generación, algunos especialmente diseñados para este Proyecto: un sistema robótico reconfigurable de rendijas (para obtener espectros de en torno a 50 objetos simultáneamente); elementos dispersores formados mediante la combinación de redes de difracción de alta calidad, fabricadas mediante procedimientos fotorresistivos, y prismas convencionales de gran tamaño, y el detector HAWAII-2 de Rockwell, diseñado para el infrarrojo cercano con un formato de  $2048 \times 2048$  píxeles, y dotado de un novedoso sistema de control, desarrollado por el equipo del proyecto. EMIR es un instrumento de segunda generación que se instalará en el foco Nasmyth de GTC.

El objetivo fundamental de nuestro Proyecto ha consistido en el desarrollo de una aplicación, completamente funcional, que resuelva de manera óptima la configuración del subsistema de rendijas mencionando anteriormente.

Esta Memoria del PFC está estructurada en torno a siete capítulos, cuyos contenidos se describen brevemente a continuación.

El primer capítulo, Motivaciones, se pretende situar al lector en el contexto del proyecto EMIR, explicando detalladamente sus objetivos, componentes, funcionamiento y tendencias futuras. También se definirá el problema que nos atañe, y se repasará la importancia de este proyecto a nivel internacional.

En el segundo capítulo se definen los objetivos marcados a la hora de realizar este Proyecto, de los cuales se ha conseguido completar satisfactoriamente la gran mayoría.

El capítulo Tecnologías y herramientas relacionadas hace un recorrido por aquellas tecnologías que se encuentran a nuestro alcance y que, si bien no han sido seleccionadas para formar parte del Proyecto, han sido investigadas con esta finalidad. Se intenta hacer comprender las decisiones tomadas durante el transcurso del Proyecto para elegir qué tecnologías se utilizaban o cuáles fueron los motivos por los que se desecharon.

El cuarto capítulo, Algoritmos, explica los pasos que realiza la aplicación para obtener el resultado, así como el por qué de la elección de estos métodos. Entre su contenido se encuentran pseudo-código y esquemas que tratan de aportar aún más información sobre su comportamiento.

En el capítulo referente a la Aplicación, se halla el manual de usuario y la documentación del código entregado. Aquí se especificarán los requisitos previos necesarios para instalar la aplicación, así como la explicación de cómo instalarla; también se hará un recorrido por el contenido de los directorios. En la documentación se encuentra la descripción de las clases, métodos y estructuras más relevantes de nuestra aplicación, para un mayor entendimiento de la misma.

El sexto capítulo recopila los resultados, comprendidos por una breve descripción del ejemplo introducido como entrada, capturas de la salida de nuestra aplicación, una tabla de tiempos y resultados obtenidos variando los parámetros de entrada y una explicación de estos tiempos.

Se finaliza con unas conclusiones, que recogen las impresiones sobre el trabajo realizado y aportan, a su vez, una futura línea en la que se puede seguir desarrollando el Proyecto.



# Capítulo 1

## Motivación

EMIR, que actualmente está entrando en su etapa de fabricación y en fase de AIV, será uno de los primeros instrumentos para usuarios en común para el GTC, el telescopio de 10 metros del proyecto GRANTECAN ubicado en el Observatorio del Roque de los Muchachos (Islas Canarias, España). EMIR está siendo construido por un Consorcio de institutos españoles y franceses liderado por el IAC. EMIR está diseñado para llevar a cabo una de las metas centrales de los telescopios de 10 metros, permitiendo a los observadores obtener espectros para un gran número de fuentes tenues de una forma temporalmente eficiente. EMIR está diseñado para operar principalmente como un MOS en la banda K, pero ofrece un amplio rango de modos de observación, incluyendo imágenes y espectroscopía, ambos de larga fisura y multiobjeto, en una longitud de onda en un rango de 0.9 a 2.5 micrómetros. Está equipado con dos subsistemas innovadores: una máscara robótica reconfigurable multi-ranura y elementos dispersivos formados por la combinación de prismas convencionales y rejillas difractantes de alta calidad, ambos localizados en el corazón del instrumento. El desarrollo y la fabricación de EMIR está financiado por el GRANTECAN y el Plan Nacional de Astronomía y Astrofísica.

La nueva generación de telescopios ópticos de 10 metros cercanos al infrarrojo, con la intención de sondear el Universo más profundamente, mantienen la promesa de proveer, por vez primera, una visión directa de los procesos que dieron forma a la creación de estrellas, galaxias, y el propio Universo. Proveerán también, por vez primera, la capacidad de detectar y aislar estrellas extragalácticas y regiones de formación de estrellas con

una sensibilidad sin procedentes y poder de resolución, ambos espaciales y espectrales. Un esfuerzo colectivo de instrumentación está en camino para permitir a estas nuevas infraestructuras su uso a pleno potencial. Las capacidades científicas de los nuevos telescopios se estiman como enormes, no sólo por su gran área de recolección de fotones, sino especialmente por los nuevos instrumentos, los cuales, debido a importantes avances tecnológicos, se espera que sean órdenes de magnitud más eficientes que sus homólogos de hoy en día. Como añadidura, estos retos tecnológicos establecerán los primeros pasos hacia la construcción de instrumental para la próxima generación de telescopios de más de 30 metros, que se encuentran en la fase principal de su diseño conceptual.

El Observatorio del Roque de los Muchachos, en la isla de La Palma, operado por el IAC, es el lugar de emplazamiento del Gran Telescopio Canarias (GTC) de 10 metros, que vio su primera luz en 2007. El GTC será el mayor telescopio óptico del mundo. Junto con este esfuerzo, una asociación de instituciones españolas y francesas dedicadas a la investigación está trabajando en el diseño y construcción de EMIR, un avanzado espectrógrafo-multiobjeto NIR para el GTC, objeto central de interés de este PFC.

EMIR (Espectrógrafo Multi-Objeto Infrarrojo) es un especlógrafo de campo ancho de usuario común operando en las longitudes de onda cercanas al infrarrojo (NIR) 0.9-2.5 micrómetros, usando máscaras multi-ranura criogénicas como selectores de campo. Las especificaciones están listadas en la Tabla 1.1. EMIR proveerá al GTC de imágenes de rendija larga y espectroscopías de varios objetos. El consorcio EMIR está formado por el IAC, la Universidad Complutense de Madrid (UCM, España), el Laboratorio de Astrofísica de Marsella-Provenza (OAMP, Francia) y el Laboratorio de Astrofísica de los Pirineos Medios (LAOMP, Francia).

EMIR proveerá a la comunidad de usuarios de GTC con nuevas capacidades clave de observación. Se espera que sea uno de los primeros especlógrafos criogénicos multiobjeto completos (MOS) en un telescopio de 10 metros, de manera que será capaz de observar en la banda K a 2,2 micrómetros sin la desventaja del alto fondo instrumental común en otros instrumentos conceptualmente similares. Algunos MOS NIR similares existentes o planificados para otros telescopios, no son enfriados y alcanzan hasta los 1,8 micrómetros solamente. Extender las capacidades de un MOS hasta los 2,2 micrómetros es el siguiente paso natural en el diseño de este instrumental. EMIR abrirá, por primera vez, el estudio de la naturaleza

Wavelength range	0.9-2.5 $\mu\text{m}$
Optimization	1.0-2.5 $\mu\text{m}$
Observing modes	Multi-object spectroscopy Wide-field Imaging
Top priority mode	K band Multi-object spectroscopy
Spectral resolution	5000,4250,4000 (JHK) for 0.6" (3-pixel) wide apertures
Spectral coverage	One observing window (Z, J, H or K) per single exposure
Array format	2048×2048 HgCdTe (Rockwell-Hawaii2)
Scale at detector	0.2 arcsec / pixel
OH suppression	In software
Image quality	$\theta_{80} < 0.3$ arcsec <b>Multi-object spectroscopic mode</b>
Slit area	6×4 arcmin, with approx. 50 slitlets of ~7" long and width varying between 0.4 and 1 arcsec
Sensitivity	$K < 20.1$ , $t = 2\text{hrs}$ , $S/N = 5$ per FWHM (continuum) $F > 1.4 \times 10^{-18} \text{erg}^{-1}\text{s}^{-1}\text{cm}^{-1}\text{\AA}^{-1}$ , $t = 4\text{hr}$ , $S/N = 5$ per FWHM (line) <b>Image mode</b>
FOV	6×6 arcmin
Sensitivity	$K < 22.8$ , $t = 1\text{hr}$ , $S/N = 5$ , in 0.6" aperture

Tabla 1.1: Especificaciones de alto nivel en EMIR

de galaxias muy lejanas con tendencia al rojo más allá de  $z=2$  con una profundidad y campo de visión sin precedentes. Con estas tendencias al rojo, el resto de galaxias bien estudiadas, en particular, la fuerte línea H alfa, se mueve a la banda K, permitiendo diagnósticos clave sobre la historia de la formación de estrellas en el Universo. EMIR permitirá puentear los estudios extensivos de tendencias al rojo más bajas llevados a cabo en los años noventa con telescopios de 4 metros y aquellos superiores a  $z=6$  planeados para el futuro próximo usando longitudes de onda pertenecientes al lejano infrarrojo milimétricas. EMIR también proveerá un vínculo entre las capacidades espectroscópicas actuales y aquellas que serán disponibles una vez que el Telescopio Espacial James Webb (JWST) esté operativo a finales de ésta década.

El diseño de EMIR ha sido ampliamente determinado por los requerimientos de su conductor científico principal, el estudio de galaxias tenues muy distantes, el proyecto GOYA, anteriormente conocido como COSMOS. Aún siendo un instrumento de usuario común, ha sido diseñado para ser conocido por la mayoría de la gran comunidad astronómica. Es, por lo tanto, un instrumento versátil que cumplirá con una amplia mayoría de proyectos científicos que comprenden desde los cuerpos estelares extragalácticos, a la astronomía del Sistema Solar interestelar.

La construcción de EMIR impulsa los retos de la instrumentación para los grandes telescopios hacia nuevos límites. La apertura de 10 metros del GTC se traduce en una gran superficie física focal. Emparejando las imágenes dadas por el telescopio con el pequeño tamaño de los detectores de hoy en día, requiere grandes ópticas con cámaras rápidas. Las grandes y pesadas ópticas necesitan un diseño mecánico avanzado y un buen modelado para reducir la flexión a niveles aceptables. Para trabajar en una región más allá de los 1,8 micrómetros, el sistema óptico de EMIR y su estructura mecánica, serán enfriados a temperaturas criogénicas. Un módulo clave de EMIR es una unidad de máscara criogénica que permite varias configuraciones diferentes de máscaras multi-ranura, disponibles cada noche, apropiado para las observaciones en cola intencionadas en el GTC, sin tener que calentar el espectrógrafo. Todos los aspectos ya mencionados, necesitan esfuerzo en desarrollo, debido a que la tecnología no está disponible ó no se puede utilizar para soluciones ya existentes.

## 1.1. Problema a tratar

El “sistema robótico reconfigurable de rendijas” con el que cuenta EMIR presenta una estructura rectangular de 240 segundos de arco de ancho y 400 segundos de arco de alto. En el interior de este espacio se encuentran 55 pares de barras horizontales, cuya altura es de  $400/55$  ( $\sim 7.27$ ) segundos de arco. En la Figura 1.1 se observa este sistema físicamente. Cada pareja de barras se desplaza horizontalmente de forma independiente permitiendo dejar un espacio entre ellas y observar el objeto espacial que se deseé, o bien cerrarse del todo para no medir nada en ese punto; de esta forma es posible configurar cada par de barras para medir hasta 55 elementos simultáneamente (ver Figura 1.2a). A este subsistema de barras se le conoce como CSU (Configurable Slit Unit).

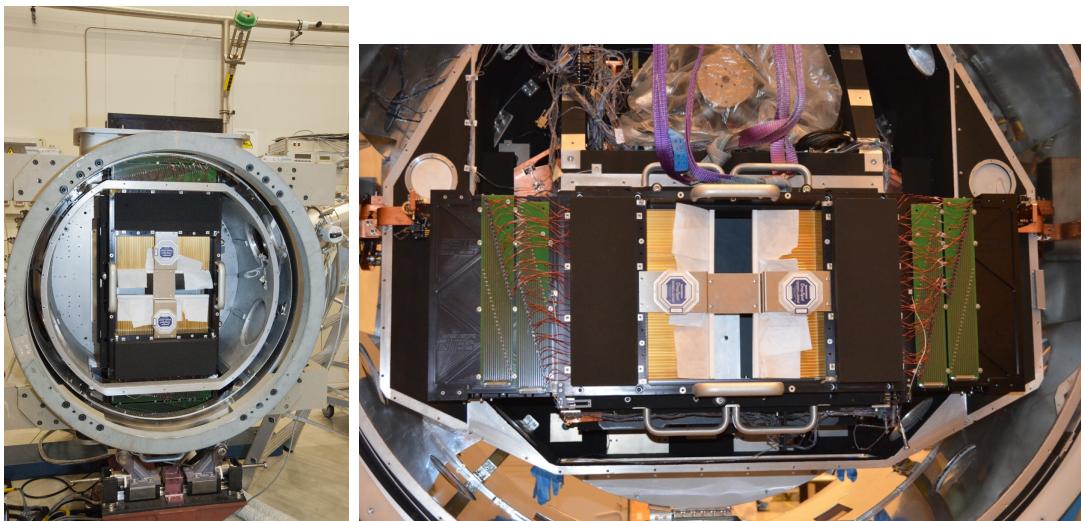


Figura 1.1: Unidad de rendijas configurables de EMIR

EMIR rotará alrededor de su eje óptico durante el apuntado y operación, llevando consigo a la CSU. A efectos prácticos, esto significa que la CSU puede configurarse en cada posición de apuntado del telescopio con ángulo de posición en el rango  $[0, 180]$  según las necesidades del usuario, como se muestra en la Figura 1.2b. Al ser simétrica, abarca todo el rango de giro posible, y puede estar colocada en cualquier punto del espacio visible (área de cielo observable de dimensiones mucho mayores que las del subsistema). Llamamos “apuntado” a una posición concreta de la CSU, especificada por un centro  $(x, y)$  y un ángulo de rotación, junto con la configuración de cada una de las 55 barras.

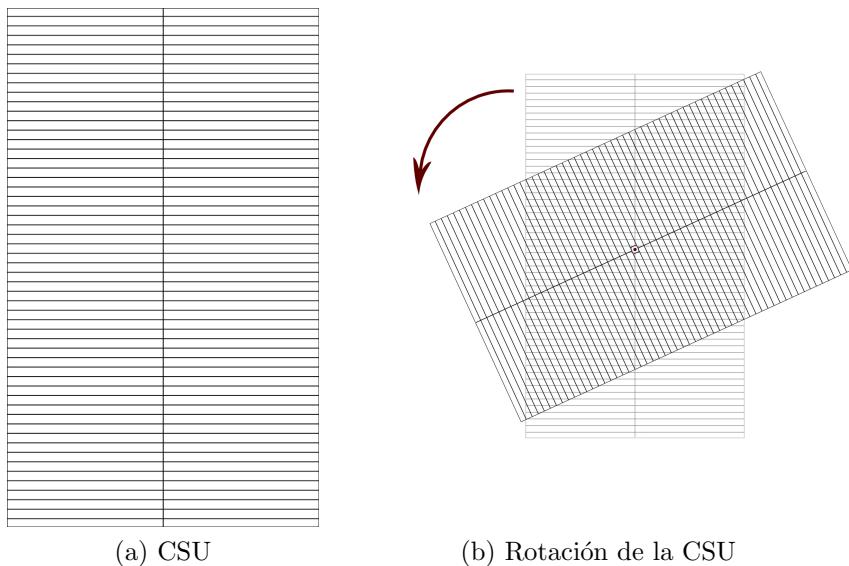


Figura 1.2: Representación gráfica de una CSU

Generalmente, para observar todos los objetos de interés astronómico de un determinado campo es necesario utilizar más de un apuntado. Debido a que el proceso de observación (medida) en cada apuntado de la CSU es muy lento, se hace necesario minimizar el número de apuntados para cubrir todos, o al menos la mayoría de objetos de alta prioridad. Para solucionar este problema se ha desarrollado en el marco de este PFC una aplicación escrita en C++ que, dada una entrada de puntos a medir, devuelve una lista de apuntados que resuelven el problema. La Figura 1.3 muestra un ejemplo de apuntado para cuatro objetos.

### 1.1.1. Beam Switching

El *beam switching* es un método para obtener mediciones de un objeto de una forma más precisa. El método consiste (aplicado a nuestro problema) en medir tanto la luz perteneciente al objeto celeste como la de la propia bóveda celeste, mover a continuación ligeramente la posición del medidor (dejando los objetos donde antes no había nada) y volver a realizar las mediciones oportunas. La Figura 1.4 muestra un ejemplo explicativo. De esta forma se obtienen una serie de valores que pueden ser procesados posteriormente por el investigador.



Figura 1.3: Apuntado con 4 barras ocupadas

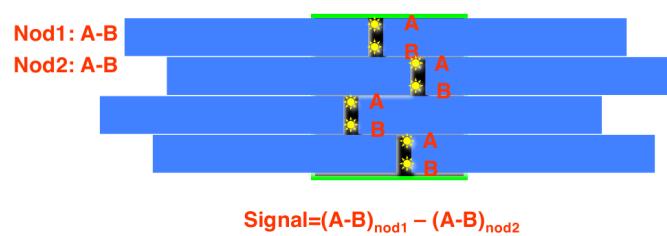


Figura 1.4: Muestra de funcionamiento de beam switching



# Capítulo 2

## Objetivos

A continuación se listan los objetivos propuestos para este PFC, de los cuales se han completado totalmente la mayoría mientras que la dedicación a algunos de ellos se ha tenido que minimizar u obviar por falta de tiempo.

- Desarrollar una aplicación plenamente funcional que resuelva de modo óptimo el problema de optimización de apuntados de la CSU del proyecto EMIR.
- Realizar la resolución del problema en el menor tiempo posible.
- Tratar las prioridades entre elementos observables.
- Contemplar el uso por parte de los investigadores del método conocido como *beam switching*.
- Estudiar diferentes alternativas para la resolución del problema computacional planteado, así como realizar una revisión bibliográfica de métodos y técnicas de optimización relacionados con dicho problema.
- Involucrarse en el conocimiento de un proyecto real de investigación aplicada, el proyecto EMIR, siendo capaz de aportar al mismo un elemento importante para su desarrollo.
- Conocer con cierto grado de detalle la tecnología involucrada en el desarrollo del proyecto EMIR así como el contexto en el que esta información se utiliza.

- Participación en un proceso de “release” de software (empaqueado, testing, validación, etc).
- Poner en práctica de modo efectivo en una aplicación real los conocimientos adquiridos en la titulación en materia de Programación e Ingeniería del Software.
- Diseño e implementación de una interfaz gráfica amigable al usuario. Éste punto ha sido eliminado por falta de tiempo, dejando como parte gráfica la visualización de los resultados.

# Capítulo 3

## Tecnologías y herramientas relacionadas

A continuación se dan a conocer algunas tecnologías o conocimientos que han resultado necesarios para el proyecto, o se han planteado como tal.

### 3.1. Parsers XML

Un analizador sintáctico [5] (en inglés parser) es una de las partes de un compilador que transforma su entrada en un árbol de derivación. El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

El uso más común de los analizadores sintácticos es como parte de la fase de análisis de los compiladores, de modo que tienen que analizar el código fuente del lenguaje. Los lenguajes de programación tienden a basarse en gramáticas libres de contexto, debido a que se pueden escribir analizadores rápidos y eficientes para éstas.

Las gramáticas libres de contexto tienen una expresividad limitada y sólo pueden expresar un conjunto limitado de lenguajes. Informalmente la razón de esto es que la memoria de un lenguaje de este tipo es limitada, la gramática no puede recordar la presencia de una construcción en una entrada arbitrariamente larga y esto es necesario en un lenguaje en el que, por ejemplo, una variable debe ser declarada antes de que pueda ser referenciada. Las gramáticas más complejas no pueden ser analizadas de forma eficiente. Por estas razones es común crear un analizador permisivo para una gramática libre de contexto que acepta un superconjunto del lenguaje (acepta algunas construcciones inválidas); después del análisis inicial las construcciones incorrectas pueden ser filtradas.

Puesto que la entrada y salida de la aplicación `CSUOptimizer` se encuentran descritas en XML, se aprovechará unos de los muchos parsers opensource disponibles en Internet para, por un lado, comprobar que la entrada no tiene errores, y por otro, transformar estos datos en las estructuras que necesita nuestra aplicación. Existen muchas librerías que nos facilitan la lectura y escritura de este tipo de ficheros; como se desea distribuir la aplicación para que cualquiera la pueda utilizar, es muy recomendable emplear una librería estándar (o muy popular) que sea eficiente.

### 3.1.1. Xerces-C++

Xerces-C++ [6] es un analizador de XML válido escrito en un subconjunto portable de C++. Xerces-C++ hace que sea fácil darle a su aplicación la capacidad de leer y escribir datos XML. Proporciona una biblioteca compartida para analizar, generar, manipular y validar documentos XML utilizando DOM, SAX y APIs de SAX2. Xerces-C++ es fiel a la recomendada XML 1.0 y a muchos estándares asociados.

El analizador proporciona un alto rendimiento, modularidad y escalabilidad. Código fuente, muestras y documentación de la API se proporcionan con el analizador. Para que resulte portable se ha tenido cuidado de hacer un uso mínimo de las plantillas, no usa RTTI, y un uso mínimo de `#ifdefs`. Sin embargo este analizador se descartó debido a que encontramos otro que se ajustaba mejor a nuestras necesidades.

### 3.1.2. Mini-XML

Mini-XML [7] es una pequeña librería XML que se utiliza para leer y escribir XML y archivos de datos estilo XML en nuestra aplicación sin necesidad de usar grandes librerías no estandarizadas. Mini-XML sólo requiere de un compilador de C compatible con ANSI (GCC funciona, al igual que la mayoría de compiladores ANSI C).

Mini-XML permite la lectura de UTF-8 y UTF-16 y la escritura de UTF-8 en ficheros XML codificados y cadenas. Los datos se almacenan en una lista enlazada con estructura de árbol, conservando la jerarquía de datos XML, y los nombres de elementos arbitrarios, atributos y valores de atributos son soportados sin límites preestablecidos, pero sólo disponibles en memoria.

Este parser no estaba disponible para su descarga por problemas de servidores en el momento que se buscó, por lo que fue descartado.

### 3.1.3. TinyXML

TinyXML [8] es un analizador de XML para C++ simple, pequeño, mínimo, que se puede integrar fácilmente en otros programas. Lee XML y crea objetos de C++ que representan el documento XML. Los objetos se pueden manipular, modificar y guardar de nuevo como XML.

TinyXML es de tipo DOM, cuya curva de aprendizaje es muy elevada y, además, suele ser lento en comparación con los SAX, por lo que también se ha descartado.

### 3.1.4. PugiXML

Pugixml [9] es una librería ligera de procesamiento de XML para C++. Consiste en una interfaz tipo DOM con ricas capacidades de recorrido/modificación, un analizador XML extremadamente rápido que construye el árbol DOM desde un archivo/buffer XML, y una implementación XPath 1.0 para las consultas de los árboles por datos complejos. También está disponible el soporte completo para Unicode, con variantes de interfaz

Unicode y conversiones entre diferentes codificaciones Unicode. La biblioteca es muy fácil de transportar y fácil de integrar y utilizar. Pugixml es desarrollado y mantenido desde 2006 y tiene muchos usuarios. Todo el código se distribuye bajo la licencia MIT, por lo que es totalmente gratuito para su uso en las aplicaciones de código abierto y propietario.

Pugixml permite un procesamiento muy rápido, práctico y eficiente de documentos XML. Sin embargo, desde que pugixml tiene un analizador DOM, no puede procesar documentos XML que no caben en la memoria; y el analizador no valida, así que se descarta porque no cumple con los propósitos de este PFC.

### 3.1.5. RapidXML

RapidXML [10] es un intento de crear el analizador XML más rápido posible, sin perder capacidad de uso, portabilidad y compatibilidad razonable W3C. Es un analizador in situ escrito en C++ moderno, con velocidad de análisis próxima a la de la función `strlen`, ejecutada en los mismos datos.

RapidXML ha estado presente desde 2006, y está siendo utilizado por muchas personas. HTC lo utiliza en algunos de sus teléfonos móviles. No se ha elegido este analizador por los mismos motivos que se descartó el TiniXML.

### 3.1.6. Libxml++

libxml++ [11] es un wrapper de C++ para la librería libxml XML parser.

Libxml2 es el analizador XML de C y es un kit de herramientas desarrolladas para el proyecto Gnome (pero utilizable fuera de la plataforma Gnome), que es software libre disponible bajo la licencia MIT. XML es un metalenguaje para diseñar lenguajes de etiquetas. HTML es el lenguaje de etiquetas más conocido. Aunque la librería está escrita en C, es posible encontrar adaptaciones en muchos lenguajes en otros entornos.

Libxml2 es conocido por ser muy portable, la librería debe generarse y trabajar sin graves problemas en una variedad de sistemas (Linux, Unix, Windows, CygWin, MacOS, MacOS X, RISC Os, OS/2, VMS, QNX, MVS,

VxWorks, ...).

Libxml2 implementa varios estándares existentes relacionadas con lenguajes de etiquetas: the XML standard, Namespaces in XML, XML Base, RFC 2396: Uniform Resource Identifiers, XML Path Language (XPath) 1.0, HTML4 parser, XML Pointer Language (XPointer) Version 1.0, XML Inclusions (XInclude) Version 1.0, ISO-8859-x encodings, así como rfc2044 [UTF-8] y rfc2781 [UTF-16] Unicode encodings (y más si se utiliza de apoyo iconv), XML Catalogs Working Draft 06 August 2001, Canonical XML Version 1.0, Relax NG, ISO/IEC 19757-2:2003, W3C XML Schemas Part 2: Datatypes REC 02 May 2001, W3C xml:id Working Draft 7 April 2004.

Debido a su portabilidad, número de estándares acogidos y gran contenido y soporte disponible en la red, sobre todo por el hecho de estar presente en Gnome (y ser éste uno de los entornos de escritorio más extendidos en distribuciones de Linux), ha sido el candidato elegido para apoyar nuestra aplicación. Se explica de qué manera se integra en **CSUOptimizer** en el capítulo referente a la aplicación [5](#).

## 3.2. Allegro

Para visualizar gráficamente los resultados de nuestra aplicación, se emplea Allegro [\[12\]](#). Allegro es una librería libre y de código abierto para la programación de videojuegos desarrollada en lenguaje C. Allegro es un acrónimo recursivo de «Allegro Low Level Game Routines» (rutinas de bajo nivel para videojuegos). Fue originalmente creado por Shawn Hargreaves para el Atari ST a principios de 1990, pero la idea fue abandonada más adelante. Alrededor de 1998, Allegro se ramificó en varias versiones. Se creó una distribución para Microsoft Windows (WinAllegro) y también una para Unix (XwinAllegro). Allegro 4.0 sería la primera versión estable de Allegro para múltiples plataformas.

La librería cuenta con funciones para gráficos, manipulación de imágenes, texto, sonidos, dispositivos de entrada (teclado, ratón y mandos de juego) y temporizadores, así como rutinas para aritmética de punto fijo y acceso al sistema de archivos. Hay 2 versiones de Allegro que cuentan con soporte oficial por parte de los desarrolladores, la versión clásica (Allegro 4) y la nueva versión (Allegro 5). La versión más reciente de Allegro 4 incluye soporte

para el manejo de archivos de datos y una implementación por software de funciones para gráficos en 3D. La versión 5 de Allegro cuenta con una nueva API y cambia la implementación por software de las rutinas gráficas por una implementación basada en OpenGL o Direct3D.

Aunque Allegro ofrece una API en lenguaje C, actualmente existen envolventes y librerías adicionales que permiten utilizarlo en otros lenguajes como Python, D, Lua y Pascal.

La versión 4 de Allegro, versión utilizada en nuestra aplicación, cuenta con varias librerías adicionales creadas por la comunidad de usuarios; entre ellas se encuentran las que agregan soporte para varios formatos de archivo multimedia (por ejemplo PNG, GIF, JPEG, MPEG, Ogg, MP3 y más).

Librerías adicionales como AllegroGL y OpenLayer utilizan OpenGL para añadir aceleración por hardware a los programas de Allegro. Tenga en cuenta que, en combinación con Glide y MesaFX (utilizando el hardware 3dfx), AllegroGL es una de las pocas soluciones de código abierto disponibles para hardware de aceleración 3D bajo DOS.

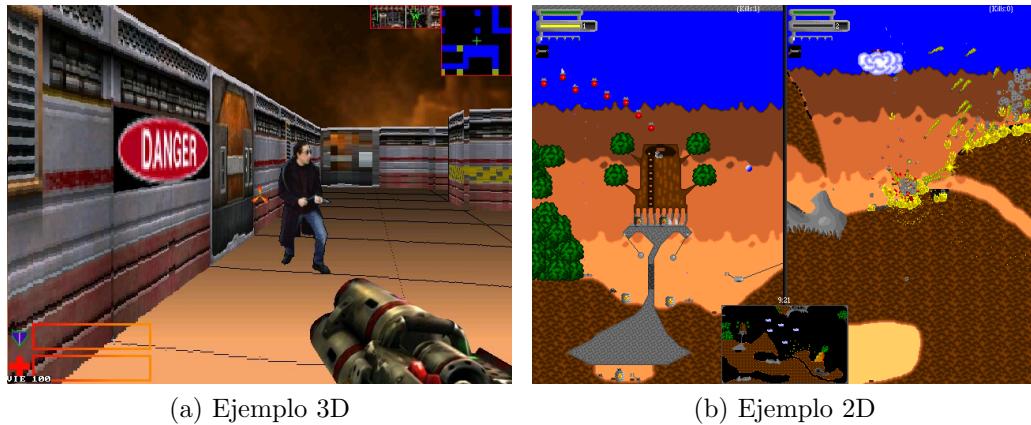


Figura 3.1: Ejemplos de la potencia gráfica de la librería Allegro

### 3.3. Algoritmos de clustering

El primer enfoque de nuestra aplicación fue basarnos en la densidad de objetos en el espacio, con el objetivo de centrarnos en áreas con mayor

densidad (y por tanto, mayor número de objetos a introducir en la CSU). Dado que nos interesa agrupar elementos en un espacio reducido, delimitado por el tamaño de la CSU, prestaremos atención a los algoritmos de clustering.

Un algoritmo de agrupamiento (en inglés, clustering) es un procedimiento de agrupación de una serie de vectores de acuerdo con un criterio. Esos criterios son por lo general distancia o similitud. La cercanía se define en términos de una determinada función de distancia, como la euclídea, aunque existen otras más robustas o que permiten extenderla a variables discretas. La medida más utilizada para medir la similitud entre los casos es la matriz de correlación entre los  $n \times n$  casos. Sin embargo, también existen muchos algoritmos que se basan en la maximización de una propiedad estadística llamada verosimilitud.

Generalmente, los vectores de un mismo grupo (o clústers) comparten propiedades comunes. El conocimiento de los grupos puede permitir una descripción sintética de un conjunto de datos multidimensional complejo. De ahí su uso en minería de datos. Esta descripción sintética se consigue sustituyendo la descripción de todos los elementos de un grupo por la de un representante característico del mismo.

Ninguno de los algoritmos clásicos de clustering cumple con las características exactas de nuestro problema, por lo que se tuvo que investigar un poco como funciona cada uno de ellos. Hay una gran diversidad de algoritmos, algunos muy antiguos y otros bastante recientes, pero casi todos tienen en común la dependencia de cierto tipo de parámetros condicionantes. Este tipo de dependencia varía bastante el resultado, y no siempre resulta fácil saber qué se está cambiando o cómo adecuarlo a nuestro problema. A continuación se muestran algunos de los que se han analizado y estudiado con el fin de utilizarlos.

### 3.3.1. K-means

K-means [13] es un método de agrupamiento, que tiene como objetivo la partición de un conjunto  $n$  en  $k$  grupos en el que cada observación pertenece al grupo más cercano a la media. Esto da lugar a una compartimentación del espacio de datos en celdas de Voronoi.

El problema es computacionalmente difícil (NP-duro). Sin embargo, hay

heurísticas eficientes que se emplean comúnmente y convergen rápidamente a un óptimo local. Estos suelen ser similares a los algoritmos de esperanza-maximización de mezclas de distribuciones gausianas por medio de un enfoque de refinamiento iterativo empleado por ambos algoritmos. Además, los dos algoritmos usan los centros que los grupos utilizan para modelar los datos, sin embargo k-means tiende a encontrar grupos de extensión espacial comparable, mientras que el mecanismo esperanza-maximización permite que los grupos tengan formas diferentes.

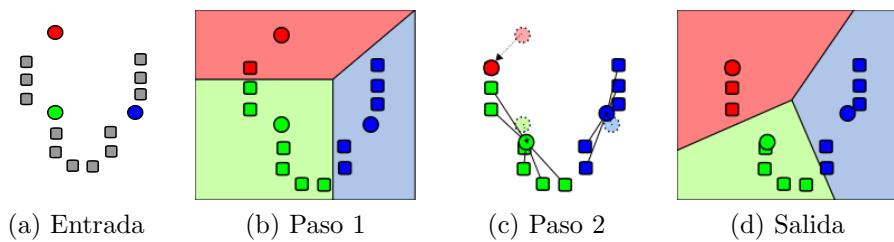


Figura 3.2: Ejemplo de cómo funciona el K-means

Como se trata de un algoritmo heurístico, no hay ninguna garantía de que converja al óptimo global, y el resultado puede depender de los grupos iniciales. Como el algoritmo suele ser muy rápido, es común para ejecutar varias veces con diferentes condiciones de partida. Sin embargo, en el peor de los casos, k-means puede ser muy lento para converger: en particular, se ha demostrado que existen conjuntos de determinados puntos, incluso en 2 dimensiones, en la que k-means toma tiempo exponencial.

### 3.3.2. DBSCAN

DBSCAN: Density-based spatial clustering of applications with noise (Agrupación espacial basada en la densidad de aplicaciones con ruido) [14], [15] es un algoritmo de clustering de datos propuesto por Martin Ester, Hans-Peter Kriegel, Jörg Sander y Xiaowei Xu en 1996. Se trata de un algoritmo de clustering basado en la densidad, ya que encuentra un número de clusters a partir de la distribución de la densidad estimada de nodos correspondientes. DBSCAN es uno de los algoritmos de agrupamiento más común y también el más citado en la literatura científica.

La definición de un conjunto en DBSCAN se basa en la noción de accesibilidad-densidad. Básicamente, un punto  $q$  es directamente accesible

desde un punto  $p$  si no está más lejos que una distancia dada  $\varepsilon$  (es decir, es parte de su  $\varepsilon$ -vecindad) y si  $p$  está rodeado por suficientes puntos de tal manera que uno pueda considerar  $p$  y  $q$  para ser parte de un clúster.  $q$  se denomina densidad alcanzable a partir de  $p$  si hay una secuencia de  $p_1, \dots, p_n$  de puntos con  $p_1 = p$  y  $p_n = q$  donde cada  $p_{i+1}$  es directamente accesible desde  $p_i$ .

Tenga en cuenta que la relación de la densidad alcanzable no es simétrica.  $q$  podría estar al borde de un clúster, que tiene insuficiente cantidad de vecinos a contar tan denso en sí. Esto detendría el proceso de encontrar una ruta que se detiene con el primer punto no denso. Por el contrario, empezar la ruta desde  $q$  podría encontrar un camino hasta  $p$ . Debido a esta asimetría, se introduce la noción de densidad-conectada: dos puntos  $p$  y  $q$  están densamente-conectados si hay un punto  $o$  de tal manera que tanto  $p$  como  $q$  son la densidad alcanzable a partir de  $o$ . La densidad-conectada es simétrica.

Un clúster, que es un subconjunto de los puntos de la base de datos, satisface dos propiedades:

- Todos los puntos dentro de la agrupación son mutuamente denso-conectados.
- Si un punto está denso-conectado con cualquier punto de la agrupación, forma parte del grupo también.

DBSCAN requiere dos parámetros:  $\varepsilon$  (EPS) y el número mínimo de puntos requeridos para formar un clúster (MinPts). Se inicia con un punto de partida arbitrario que no ha sido visitado. La  $\varepsilon$ -vecindad de este punto se recupera, y si contiene suficientes puntos, se inicia un clúster. De lo contrario, el punto es considerado como ruido. Tenga en cuenta que este punto más adelante se podría encontrar en una  $\varepsilon$ -ambiente suficientemente grande de un punto diferente y por lo tanto, formar parte de un clúster.

Si se encuentra un punto para ser una parte densa de un clúster, su  $\varepsilon$ -vecindad es también parte de ese grupo. Por lo tanto, se añaden todos los puntos que se encuentran dentro de la  $\varepsilon$ -vecindad, así como su propia  $\varepsilon$ -vecindad cuando también son densos. Este proceso continúa hasta que el clúster de densidad-conectada se halla por completo. Entonces, un nuevo

punto no visitado es recuperado y procesado, llevando al descubrimiento de un clúster adicional o ruido.

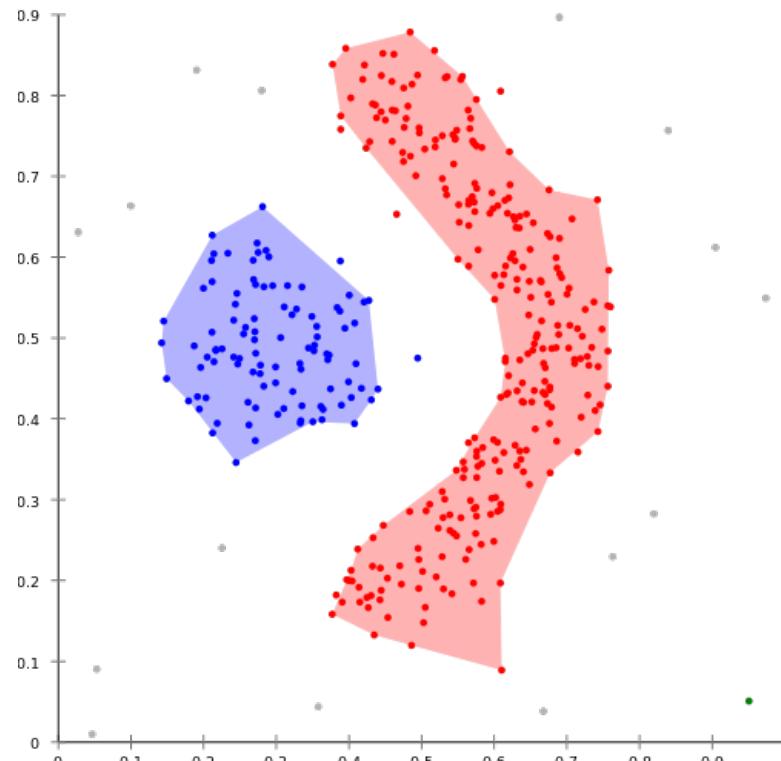


Figura 3.3: Ejemplo del resultado del DBSCAN. Los puntos en gris son ruido

#### Ventajas

- DBSCAN no requiere especificar el número de grupos en los datos a priori, como hace el k-means.
- DBSCAN puede encontrar grupos de forma arbitraria. Se puede incluso encontrar un clúster completamente rodeado por (pero no conectado a) un clúster diferente. Debido al parámetro MinPts, el llamado efecto de enlace único (diferentes grupos que se conectan por una delgada línea de puntos) se reduce.
- DBSCAN tiene una noción de ruido.
- DBSCAN requiere sólo dos parámetros y es insensible a la ordenación de los puntos en la base de datos. (Sin embargo, los puntos que se encuentren en el borde de dos grupos diferentes pueden intercambiar

pertenencia al clúster si se cambia el orden de los puntos, y la asignación de clúster es único sólo hasta el isomorfismo.)

- DBSCAN está diseñado para su uso con bases de datos que pueden acelerar las consultas de región, por ejemplo, mediante un árbol R\*.

#### Desventajas

- La calidad de DBSCAN depende de la medida de distancia utilizado en la función  $\text{regionQuery}(P, \varepsilon)$ . La distancia métrica más común es la distancia euclídea. Especialmente para los datos de grandes dimensiones, este indicador puede resultar casi inútil debido a la llamada “maldición de la dimensionalidad”, por lo que es difícil encontrar un valor adecuado para  $\varepsilon$ . Este efecto, sin embargo, también está presente en cualquier otro algoritmo basado en la distancia euclídea.
- DBSCAN no puede agrupar conjuntos de datos correctamente si estos tienen una gran diferencia en la densidad, ya que la combinación  $\text{MinPts}-\varepsilon$  no puede entonces ser elegida adecuadamente para todos los grupos.

Todos estos algoritmos trabajan bastante bien cuando existen áreas más o menos diferenciadas unas de otras, sin embargo la mayoría de las ocasiones se trata con datos dispersos de manera uniforme, con lo que el uso de estos algoritmos puede ser más un inconveniente que una ventaja. Por otro lado, todos nuestros objetos son relevantes, por lo que no podemos arriesgarnos a que se les tome por ruido y, por lo tanto, se pierdan.

## 3.4. Heurísticas

En computación, dos objetivos fundamentales son encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una heurística es un algoritmo que abandona uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que

siempre será así. Las heurísticas generalmente son usadas cuando no existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.

A menudo, pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Aun así, estas instancias concretas pueden ser ignoradas porque no deberían ocurrir nunca en la práctica por ser de origen teórico. Por tanto, el uso de heurísticas es muy común en el mundo real.

A continuación se presenta la heurística que se ha evaluado, el método GRASP, recomendada por el Doctor Marcos Moreno Vega.

### 3.4.1. GRASP

La palabra GRASP proviene de las siglas de Greedy Randomized Adaptive Search Procedures que se puede traducir como: Procedimientos de Búsqueda Voraces Aleatorizados y Adaptativos.

El método GRASP se introduce inicialmente por Feo y Resende en [16]. Es un procedimiento iterativo que consiste en: una fase de construcción y una fase de búsqueda local. Se obtiene una solución factible durante la fase de construcción aplicando un procedimiento voraz. En cada iteración del procedimiento voraz se agrega un nuevo elemento a la solución de acuerdo al valor de una función voraz. En lugar de escoger siempre el mejor elemento candidato, se construye una lista con los mejores candidatos, de donde se selecciona uno aleatoriamente. El término adaptativo se refiere al hecho de que los beneficios asociados con cada elemento son actualizados en cada iteración de la fase de construcción para reflejar los cambios producidos por selecciones previas. Una vez que se construye una solución utilizando un procedimiento voraz, se realiza un procedimiento de búsqueda local.

En el Algoritmo 1 se muestra el pseudocódigo de esta heurística y en las Secciones 3.4.1.2, 3.4.1.3 y 3.4.1.4 se explican los procedimientos que intervienen en el algoritmo.

### 3.4.1.1. Algoritmo GRASP

**while** (*criterio no satisfecho*) **do**

    Construye una solución inicial usando el procedimiento voraz

    Realiza una búsqueda local para mejorar la solución construida.

**Algoritmo 1:** Pasos del GRASP

### 3.4.1.2. Algoritmo de fase de construcción del GRASP.

Sea  $f$  una función voraz,  $\alpha$  es valor del parámetro para controlar la aleatoriedad del procedimiento,  $x$  una solución parcial,  $C$  el conjunto de elementos candidato y  $RCL$  la lista restringida de candidatos.

$x \leftarrow \emptyset$

Inicializa el conjunto de candidatos  $C$

**while**  $x$  *infactible* **do**

$a = \min\{f(t), t \in C\}$

$b = \max\{f(t), t \in C\}$

$RCL = \{c \in C, f(c) \leq a + \alpha(b - a)\}$

    Seleccionar aleatoriamente un elemento  $c \in RCL$

$x \leftarrow x \cup \{c\}$

    Actualizar  $C$

**Algoritmo 2:** Pseudo-código constructivo para GRASP

### 3.4.1.3. Algoritmo de búsqueda local.

Sea  $f(x)$  la función a minimizar,  $x$  una solución factible y  $N(x)$  una estructura de vecindad.

$CriterioParada \leftarrow$  falso

**while** *no se satisfaga CriterioParada* **do**

$x' = \operatorname{argmin}\{f(y) : y \in N(x)\}$

**if** ( $f(x') < f(x)$ ) **then**

$x \leftarrow x'$

**else**

$CriterioParada \leftarrow$  cierto

**Algoritmo 3:** Pseudo-código de la búsqueda local

### 3.4.1.4. Algoritmo voraz.

Sean  $S$  una solución parcial,  $E$  el conjunto factible de elementos que pueden ser añadidos a la solución parcial,  $e_i \Delta$  los elementos del conjunto  $E$  y  $g$  una función voraz.

$S \leftarrow \emptyset$

**while**  $S$  no factible **do**

    Evaluar la función voraz  $g$  para cada elemento de  $E$

    Seleccionar el mejor elemento  $e \in E$  de acuerdo con el valor de la función voraz  $g$

$S \leftarrow S \cup \{e\}$

    Actualizar  $E$

**Algoritmo 4:** Método voraz para seleccionar los mejores elementos de la búsqueda

El objetivo de utilizar este método es mejorar la solución inicial obtenida con nuestro algoritmo constructivo. Se explicará este proceso y hasta qué punto se ha utilizado dentro de **CSUOptimizer** en el Capítulo 5.

# Capítulo 4

## Algoritmos

El objetivo del `CSUOptimizer` es cubrir (abarcar) todos los objetos presentes en el campo observado, con el menor número de apuntados posible. Para resolver este problema se plantearon varios posibles frentes de actuación, explicados brevemente a continuación, y exhaustivamente al final del Capítulo, donde se detalla el algoritmo utilizado.

### 4.1. Clustering

El primer enfoque que se exploró fue el de utilizar la densidad que presentan los objetos en su distribución espacial, de forma que se actúe primero sobre zonas donde existe un mayor número de puntos. Para diferenciar las zonas en las que se agrupan los objetos se ha utilizado el algoritmo DBScan. Para ello se partió del código [17] adaptándolo y traduciéndolo a C++.

El Algoritmo 5 muestra un pseudo-código para el DBScan.

```

DBSCAN( $D, \text{eps}, \text{MinPts}$ ) {
     $C = 0$ 
    for (cada punto P no visitado en el cjto de datos D) do
        marcar  $P$  como visitado
         $\text{NeighborPts} = \text{regionQuery}(P, \text{eps})$ 
        if ( $\text{sizeof}(\text{NeighborPts}) < \text{MinPts}$ ) then
            marcar  $P$  como RUIDO
        else
             $C = \text{siguiente cluster}$ 
             $\text{expandCluster}(P, \text{NeighborPts}, C, \text{eps}, \text{MinPts})$ 
    }
     $\text{expandCluster}(P, \text{NeighborPts}, C, \text{eps}, \text{MinPts}) \{$ 
        añadir  $P$  al cluster  $C$ 
        for (cada punto  $P'$  en NeighborPts) do
            if ( $P'$  no visited) then
                marcar  $P'$  como visitado
                 $\text{NeighborPts}' = \text{regionQuery}(P', \text{eps})$ 
                if ( $\text{sizeof}(\text{NeighborPts}') \geq \text{MinPts}$ ) then
                     $\text{NeighborPts} = \text{NeighborPts}$  unido a  $\text{NeighborPts}'$ 
                    if ( $P'$  todavía no es miembro de algún cluster) then
                        añadir  $P'$  al cluster  $C$ 
                }
            regionQuery( $P, \text{eps}$ ) {
                return todos los puntos vecinos de  $P'$  (incluyendo  $P$ )
            }
        
```

**Algoritmo 5:** Pseudo-código del algoritmo DBSCAN

Los clusters obtenidos dependen de dos parámetros,  $\text{eps}$  y  $\text{Min\_Pts}$  que indican respectivamente la distancia que se permite entre puntos y el número mínimo de objetos requeridos para formar un cluster. Para obtener buenos resultados es necesario un conocimiento profundo del problema de entrada, puesto que se deben variar los parámetros según la distribución de los objetos del mismo. Este procesado resulta costoso y poco rentable, puesto que por lo general este tipo de algoritmos presenta pobres resultados en problemas con distribuciones espaciales homogéneas.

En la Figura 4.1 se puede observar el resultado del algoritmo aplicado al ejemplo **real11.xml** (cuya representación se puede ver en la Figura 6.8a). Los parámetros utilizados han sido  $\text{eps} = 86$  y  $\text{MinPts} = 4$ .

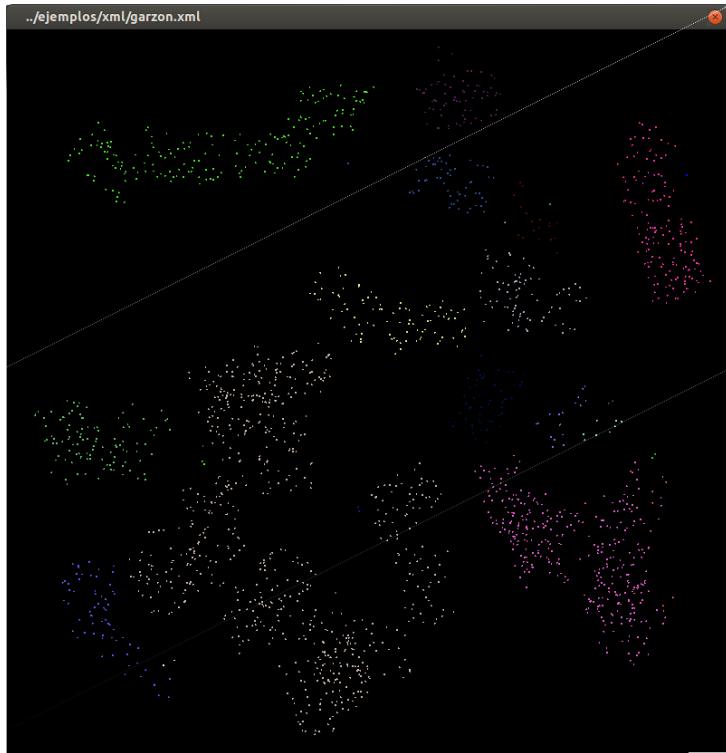


Figura 4.1: Resultado de aplicar DBSCAN al `real1.xml`

Aunque se ha mantenido el uso del algoritmo DBScan como algo opcional (utilizando la opción `--dbscan`) en el `CSUOptimizer`, en virtud de los resultados obtenidos, no se aconseja su uso por defecto. Para algunas distribuciones de los objetos a observar, el uso previo del DBScan puede reducir algo el tiempo de cómputo total, aunque esta reducción no es extensible a todas las instancias de entrada. Si el usuario desea utilizar el DBScan, los parámetros a modificar se encuentran en `bin/dbscan.C`, explicados en la Sección 5.3.5.

## 4.2. Grasp

El algoritmo GRASP (*Greedy Randomized Adaptive Search Procedure*) implementado no es exactamente el que se ha explicado en la Sección 4.2, puesto que empezó siendo una base para dicha heurística, pero no se llegó a desarrollar del todo debido a los buenos resultados obtenidos por el algoritmo

constructivo que se explicará en la Sección 4.3. Un posible pseudo-código del algoritmo implementado es el que se muestra en el Algoritmo 6 y consiste en generar varias soluciones aleatorias de las que se escoge la mejor (aquella con menor número de apuntados). De esta mejor solución se separan aquellos apuntados independientes (no colisionan con ningún otro, véase Sección 4.3.2) de los que presentan algún solape con otra CSU. Los independientes pasan a formar parte de la solución, pues han de estar necesariamente para cubrir todos los objetos, mientras que los restantes se introducen en un conjunto de CSUs candidatas (RCL), junto con el resto de apuntados del resto de soluciones generadas. La RCL es ordenada de mejor a peor resultado, y se va eligiendo una de las mejores candidatas aleatoriamente hasta cubrir todos los puntos.

```

grasp(resultado, puntos) {
    Generar N posibles soluciones aleatorias
     $S_{ini} \leftarrow$  mejor solución generada
     $S_{fin} \leftarrow \emptyset$ 
    for ( $\forall$  apuntado  $C \in S_{ini}$ ) do
        if ( $C$  no colisiona con  $S_{ini} - \{C\}$ ) then
             $S_{fin+} = C$ 
         $RCL = S_{ini} \cup$  resto de soluciones generadas
        Ordenar  $RCL$  por mejores apuntados
        while ( $puntos cubiertos en S_{fin} < puntos$ ) do
             $C =$  apuntado aleatorio  $\in RCL$  elegido entre los mejores
             $C_{fin+} = C$ 
        if  $C_{fin}$  mejor que resultado then
            resultado =  $C_{fin}$ 
    return resultado
}

```

**Algoritmo 6:** Pseudo-código de un algoritmo GRASP

### 4.3. Algoritmo constructivo

El algoritmo que construye la solución del problema se estructura en dos fases. En primer lugar se crea una solución inicial basada en la posición cartesiana de los objetos en el espacio. Luego se hace un procesado de los apuntados obtenidos, procurando eliminar aquellos que se puedan unificar para dar lugar a un número menor de apuntados.

### 4.3.1. Fase 1

Los pasos que realiza este primer método que crea la solución inicial se muestran en el Algoritmo 7.

```

Solucion[tipo orden] ← Ø
for (Cada tipo de ordenación) do
    ordenar los objetos
    while (queden objetos por cubrir) do
        p ← siguiente punto de la lista no eliminado
        C = mejor CSU ∈ crearApuntados(p, puntos)
        puntos- = ∀ puntos ∈ C
        Solucion[tipo orden] = Solucion[tipo orden] ∪ {C}
    Sol = min(Solucion[tipo orden])

```

**Algoritmo 7:** Pseudo-código del algoritmo que crea la solución inicial

La función `crearApuntados()` se encarga de generar todas las posibles CSUs válidas, y se entiende por válida cualquier CSU que tenga al menos un objeto en alguna de sus barras, cuyo centro está situado sobre el objeto especificado por parámetro. Una vez generado dicho apuntado, se realizan una serie de movimientos de mejora sobre el mismo, con el fin de “capturar” todos los objetos posibles. Los pasos se muestran en el Algoritmo 8.

```

posibles ← Ø
for (Todas las rotaciones posibles) do
    Crear CSU con centro en p
    rellenar_con_puntos(CSU, puntos)
    while (número de puntos en CSU cambie) do
        movimiento de mejora arriba
        rellenar_con_puntos(CSU, puntos)
        movimiento de mejora abajo
        rellenar_con_puntos(CSU, puntos)
        movimiento de mejora izquierda
        rellenar_con_puntos(CSU, puntos)
        movimiento de mejora derecha
        rellenar_con_puntos(CSU, puntos)
    posibles = posibles ∪ {CSU}

```

**Algoritmo 8:** Pseudo-código del método de creación de apuntados

Los movimientos de mejora consisten en cambiar el centro del apuntado

sin modificar su ángulo de posición, con la restricción de que todos los objetos que se encuentran en ese instante en sus barras, deben permanecer dentro de ese apuntado. Cuando se dice que un punto está dentro de una CSU quiere decir que dicha CSU tiene una barra en la que el objeto encaja perfectamente. Se permite que estos objetos “inamovibles” cambien de una barra a otra, se desplacen en una misma barra, o se acerquen/alejen en menor o mayor medida del borde de la barra, siempre y cuando sigan perteneciendo al apuntado. Este tipo de movimientos están explicados con mayor detalle en la Sección 5.3.3.2.

La Figura 4.2 muestra de forma gráfica estos pasos. En el primer caso existe un apuntado con un objeto *A* en su barra número 26. A continuación se muestra el resultado del movimiento de mejora arriba, tras el que se ha introducido el objeto *B* en la barra 36, mientras que el objeto *A* ha pasado a la barra 0. En el tercer paso se ha producido el movimiento de mejora abajo, que sitúa a *B* en la barra 54 y, por tanto, a *A* en la barra número 18. El siguiente paso es realizar el movimiento de mejora izquierda. Dado que la coordenada X de *B* es menor que la de *A*, es *B* el objeto que se sitúa más a la izquierda, permitiendo la incorporación del objeto *C* en la barra número 31. Por último se realiza el movimiento opuesto al anterior, el cual sitúa a *C* en el borde derecho de la CSU. El resultado global de estos movimientos ha sido pasar de un único objeto *A cubierto* por la CSU considerada a *cubrir* tres objetos.

La función `rellenar_con_puntos` del Algoritmo 8 es la que se encarga de analizar objeto por objeto si éste está dentro o no de un apuntado, y en caso afirmativo, lo añade al mismo. Se considera que un punto está dentro de una CSU si:

- Se encuentra en el área abarcada por la CSU
- Se sitúa sobre una barra que no tiene ningún objeto asociado a ella y
- El objeto cumple las restricciones físicas elegidas por el usuario, como puede ser que no sobrepase el límite de distancia respecto al borde de la barra o que se encuentre en la parte superior de la misma.

Existen ocasiones en las que un objeto no ajusta en una barra por un margen muy pequeño, bien porque la barra esté ocupada ya por otro punto o porque esté en un rango no válido de la misma. En estas ocasiones, en lugar de

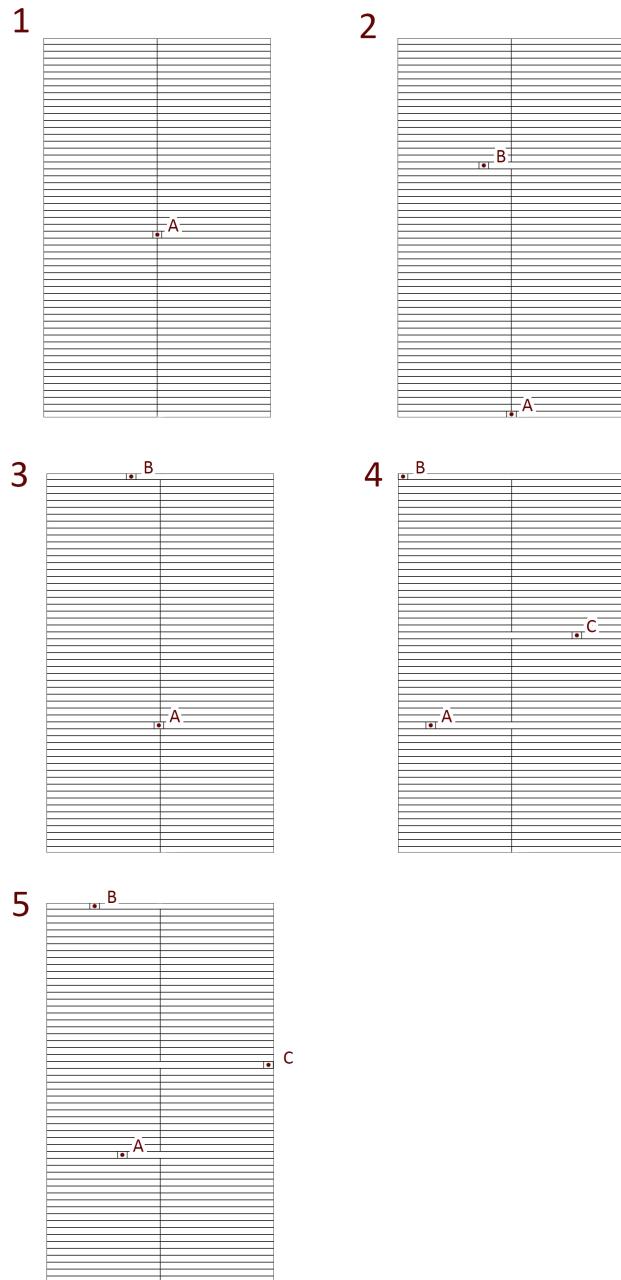


Figura 4.2: Movimientos de una CSU en el proceso de mejora

descartar el objeto y buscar otro que sí ajuste, se ha hallado que es preferible adaptar la CSU (siempre que ello sea posible, cumpliendo siempre con las

restricciones previamente definidas) para que éste entre en el apuntado.

#### 4.3.2. Fase 2

Una vez se tiene una solución al problema, es decir, un conjunto de apuntados que “cubren” todos los objetos del problema, se intenta mejorar la solución eliminando o reduciendo las colisiones entre apuntados. Diremos que un apuntado *colisiona* con otro cuando la intersección de las áreas cubiertas es no vacía. Ello puede ocasionar que un objeto sea asociado a más de un apuntado. Supongamos que se asocia con dos apuntados  $A$  y  $B$ . En este caso lo que se intenta es crear un nuevo apuntado (CSU)  $C$  con posición diferente a las anteriores, que o bien las sustituya o bien consiga una mejora respecto a alguna, es decir,  $C$  mejor que  $A$ ,  $C$  mejor que  $B$  o  $C \equiv A \cup B$ . Esto se consigue iterando sobre el subconjunto de objetos son abarcados por  $A$  y  $B$  con un número mucho mayor de rotaciones de CSU. El Algoritmo 9 presenta el esquema del algoritmo.

```

 $lista_{fin} \leftarrow \emptyset$ 
 $lista_{ini} = \text{resultado}$ 
while (Mientras se mejore el resultado) do
     $lista_{fin} = \text{los que no colisionan con ninguno.}$ 
    while (queden objetos por cubrir) do
        CSU  $C \leftarrow \text{primer apuntado} \in lista_{ini}$ 
         $subcto\_puntos = \text{puntos de } C$ 
        Quitar  $C$  de  $lista_{ini}$ 
        while (queden CSUs  $Q$  por mirar en  $lista_{ini}$ ) do
            if ( $Q$  colisiona con  $C$ ) then
                 $subcto\_puntos+ = \text{puntos de } Q$ 
                Quitar  $Q$  de  $lista_{ini}$ 
            if (colisión innecesaria) then
                Obtener_nuevos_apuntados( $subcto\_puntos$ )
                if (consigue mejorar) then
                     $lista_{ini}+ = \text{nuevos apuntado}$ 
                else
                    Introducir en  $lista_{ini}$  los que se quitaron
                    Poner en  $lista_{fin}$  el apuntado  $C$ 
             $lista_{ini} = lista_{fin}$ 

```

**Algoritmo 9:** Pseudo-código del algoritmo para reducir colisiones

Puesto que cada objeto puede tener una prioridad superior a la del resto (siendo el 1 la mayor prioridad), se ha añadido también prioridad a los apuntados, de modo que la prioridad del mismo la decide el objeto con mayor prioridad contenido en dicha CSU. Una vez obtenidos los resultados, los apuntados son ordenados por prioridades, situando primero los de mayor prioridad. En caso de que varios apuntados presenten la misma prioridad, se ordenarán según el número de objetos que abarquen los mismos, colocando en primer lugar aquellos apuntados cuyas barras estén completas.

En el Capítulo 5 se expone en detalle la implementación de todos estos algoritmos y en qué punto del código de la aplicación se encuentran.



# Capítulo 5

## La Aplicación CSUOptimizer

En este capítulo se explica en detalle la aplicación `CSUOptimizer`, cómo instalarla, cómo funciona, qué opciones tiene disponibles, etc. `CSUOptimizer` ha sido desarrollada en C++ sobre una plataforma Linux, más concretamente una distribución Ubuntu. En este Capítulo asumiremos que el usuario final dispone de un PC con esta distribución instalada y operativa.

La aplicación se entrega con la siguiente estructura de directorios:

- `bin`: Aplicación `CSUOptimizer`.
- `documentacion`: Documentación de código generada con Doxygen.
- `ejemplos`: Algunos ejemplos de entrada.
  - `conversor`: Aplicación para convertir archivos.
  - `xml`: Ficheros XML correspondientes a los ejemplos.
- `map_editor`: Aplicación para generar ejemplos sintéticos.
- `memoriaEmir`: Código LaTEX de este documento.
- `Resultados`: Ficheros de resultados de la aplicación.

Para la implementación de la aplicación se barajaron dos lenguajes de programación, JAVA y C++, siendo elegido el segundo debido a la

importancia de una respuesta rápida a la hora de obtener soluciones por parte de la aplicación. Se ha preferido utilizar un lenguaje compilado frente a uno interpretado en un intento de satisfacer este requisito.

## 5.1. Repositorio de código

El código de la aplicación, así como el manual de la misma, se encuentra disponible en un repositorio remoto alojado en Bitbucket [18] del que es posible descargarla. Bitbucket es un servicio de alojamiento basado en web, para proyectos que utilizan el sistema de control de versiones Mercurial y/o Git. En el caso de este PFC hemos optado por utilizar Mercurial [19].

## 5.2. Instalación

### 5.2.1. Dependencias

Antes de instalar la aplicación ha de asegurarse que se satisfacen todas las dependencias de la misma, que se enumeran a continuación:

#### 5.2.1.1. make

La herramienta make es necesaria para la automatización del compilado. Puede instalarse desde la consola mediante:

```
sudo apt-get install make
```

#### 5.2.1.2. Compilador de C++

El compilador g++ es imprescindible para poder compilar el código. Normalmente viene instalado por defecto en cualquier distribución Linux, pero si no fuera el caso, se puede instalar usando:

```
sudo apt-get install g++
```

### 5.2.1.3. libxml++

Ésta librería es el parser XML que se ha usado con aplicación, tal y como se ha descrito en la Sección 3.1.6. Para instalar `libxml++` se precisan los siguientes comandos:

```
sudo apt-get install libxml++2.6-dev libxml++2.6-doc  
sudo apt-get install libgtkmm-2.4-dev
```

### 5.2.1.4. Doxygen

Doxygen [20] es un acrónimo de dox(document) gen(generator), generador de documentación para código fuente. Se trata de un generador de documentación para diversos lenguajes, entre ellos C++, C, o Java. Dado que es fácilmente adaptable, funciona en la mayoría de sistemas Unix así como en Windows y Mac OS X. La mayor parte del código de Doxygen ha sido escrito por Dimitri van Heesch. Varios proyectos como KDE usan Doxygen para generar la documentación de su API. KDevelop incluye soporte para Doxygen.

Para descargar Doxygen e intalarlo han de seguirse estos pasos:

- Descargar y descomprimir el código fuente:

```
wget http://ftp.stack.nl/pub/users/dimitri/doxygen-1.8.4.src.tar.gz  
tar -xzvf doxygen-1.8.4.src.tar.gz
```

- Situarse en el directorio generado e intentar pre-instalar

```
cd doxygen-1.8.4/  
. ./configure
```

- Si falla alguna de las comprobaciones, instalar dependencias.

- para graphviz:

```
sudo apt-get install graphviz
```

- para flex:

```
sudo apt-get install flex
```

- para bison:

```
sudo apt-get install bison
```

- Una vez ejecutado el `./configure`

```
make  
sudo make install
```

#### 5.2.1.5. LaTeX

LaTeX [21] es un sistema de composición de textos orientado especialmente a la creación de libros, documentos científicos y técnicos. LaTeX está formado por un gran conjunto de macros de TeX, escrito por Leslie Lamport en 1984, con la intención de facilitar el uso del lenguaje de composición tipográfica, TEX, creado por Donald Knuth. Es bien conocido en entornos académicos y científicos en los que la calidad tipográfica de los documentos generados es muy apreciada. LaTeX es software libre bajo licencia LPPL.

Esta herramienta únicamente es necesaria si se necesita generar este documento, y es posible instalarla con:

```
sudo apt-get install texlive-extra-utils  
sudo apt-get install texlive-latex-extra  
sudo apt-get install texlive-latex-recommended  
sudo apt-get install texlive-full  
sudo apt-get install texlive-science
```

#### 5.2.1.6. Allegro

La librería Allegro, descrita en la Sección 3.2, se utiliza en el CSUOptimizer para la representación gráfica de los apuntados.

Para instalar Allegro se debe utilizar el siguiente paquete:

```
sudo apt-get install liballegro4.2-dev
```

Para utilizar Allegro en el código, basta con incluir el fichero de cabecera allegro.h en el código tal como se muestra en la línea 1 del Listado 5.1. Este listado muestra cómo utilizar Allegro para inicializar y crear una ventana.

```

1 #include <allegro.h>
2 // Para poder cerrar la ventana pulsando el boton de cerrar
3 volatile int close_button_pressed = FALSE;
4 void close_button_handler(void) {
5     close_button_pressed = TRUE;
6 }
7
8 int main (int argc, char **argv) {
9     allegro_init(); // Inicializamos allegro
10    install_keyboard();
11    set_color_depth(16);
12    set_window_title("Nombre de la ventana");
13    // Definimos los tamanios de la ventana (800x800) pixeles de resolucion
14    int w = 800, h = 800;
15    //Generamos la ventana
16    if(set_gfx_mode(GFX_AUTODETECT_WINDOWED,w,h,0,0) != 0) { //modo grafico
17        set_gfx_mode(GFX_TEXT,0,0,0,0);
18        allegro_message("imposible iniciar el modo video\n%s\n",
19                        allegro_error);
20        return 1;
21    }
22    clear_bitmap(screen); // Limpiamos la pantalla
23    while (!key[KEY_ESC]&&!close_button_pressed) {}
```

Listado 5.1: Ejemplo básico de uso de la librería Allegro

Si lo que se desea es dibujar puntos o líneas, se debe crear un nuevo BITMAP sobre el que dibujar, modificarlo, y luego enviarlo a la pantalla. El Listado 5.2 muestra un ejemplo de cómo hacerlo.

```

1 BITMAP* bmp = create_bitmap(ancho, alto);
2 // Creamos una linea roja de (x1,y1) a (x2,y2)
3 line(bmp, x1, y1, x2,y2,makecol(255,0,0));
4 // Creamos un punto azul de 4 pixeles de radio en (x,y)
5 circlefill(bmp, x, y, 4, makecol(0,0,255));
6 // Enviamos lo pintado en bmp a la pantalla
7 stretch_blit(bmp, screen, 0,0,bmp->w, bmp->h, 0,0, SCREEN_W, SCREEN_H);
```

Listado 5.2: Ejemplo de dibujo con Allegro

Estas dos combinaciones son capaces de dibujar tanto los elementos que forman el problema como los apuntados que lo solucionan.

### 5.2.2. Instalación

Una vez comprobadas las dependencias, no hay más que descargar (bien descargando el fichero `.tar.gz`, o bien clonando el repositorio) y descomprimir el fichero del proyecto. Para compilar la aplicación y utilizarla basta con ejecutar:

```
cd bin/  
make
```

Esto creará el archivo ejecutable `csuoptimizer`. Para comprobar que funciona correctamente se puede probar con alguno de los ejemplos incluidos en el directorio `ejemplos/xml/`.

```
./csuoptimizer ..//ejemplos/xml/exacto.xml
```

La salida debería ser igual a la mostrada en la Figura 6.1b.

## 5.3. CSUOptimizer

`CSUOptimizer` utiliza cinco clases, una de configuración y cuatro de estructuras. La clase de configuración tiene como finalidad establecer opciones entre todas las instancias del resto de las clases, según las necesidades del usuario. En esta sección se explicará cada una de estas estructuras, detallando el significado de los atributos de cada clase y el funcionamiento de los métodos más relevantes de cada una de ellas.

### 5.3.1. La clase Config

La clase `Config` se encarga de encapsular todas las opciones establecidas por el usuario mediante parámetros introducidos en línea de comandos. Las

opciones que puede utilizar la aplicación se detallan en la Sección 5.4. Estas opciones se establecen al comienzo de la ejecución y permanecen invariables hasta la finalización de la misma. Por ello se crea una única instancia de esta clase y se asocia estáticamente con todas las instancias de las clases `Element` y `CSU`.

### 5.3.2. La clase Element

Esta clase representa al elemento principal del problema abordado: los objetos de interés astronómico a observar con el instrumento. Dado que estos objetos observables tienen unas coordenadas ( $X, Y$ ) que los definen, los trataremos como si fueran puntos en un plano, por lo que las referencias de “objeto”, “punto” y “elemento” se refieren en este documento al mismo concepto. La definición de la clase se encuentra en el fichero `bin/element.h` y la implementación de sus métodos está recogida en el archivo `bin/element.c`.

#### 5.3.2.1. Atributos de Element

Los objetos `Element` tienen información invariable desde el momento de su creación, como son el valor de sus coordenadas, recogidas en los atributos `x`, `y`, y su prioridad, almacenada en `prioridad`.

Se definen además los siguientes atributos:

- **identificador**: común para todas las instancias; sirve para asignarle un valor único autoincremental al identificador (`id`) del objeto.
- **id**: identificador del objeto, usado para diferenciar instancias de esta clase.
- **barra**: barra del apuntado en la que se encuentra el objeto. Se especifica al asociar el objeto con una `CSU`.
- **dist**: distancia euclídea que separa el punto del borde lateral izquierdo del apuntado.
- **zona**: área específica de la barra donde se sitúa el objeto, comprendida por valores entre 0,1 y 7,26, puesto que la altura de la barra tiene un borde mínimo de seguridad de 0,1 de alto por cada lado.

- **orden**: establece la manera de ordenar los objetos. Puede tomar los valores MENORX, MAYORX, MENORY, MAYORY y PRIORI.

### 5.3.2.2. Métodos relevantes de Element

Esta clase no tiene métodos complejos que necesiten una mención especial. Cada atributo tiene un par de métodos getters y setters.

También dispone de operadores de comparación sobrecargados para que se puedan ordenar correctamente los objetos al insertarlos en un conjunto de la librería `set` del estándar de C++. El operador “menor que”, junto con el atributo de ordenación de la clase, permite que los puntos estén organizados según el valor de `x`, el valor de `y` o el de su `prioridad`.

### 5.3.3. La clase CSU

La clase `CSU` representa cada apuntado de `CSU` posible. La cabecera de esta clase se encuentra en el archivo `element.h` y la implementación de sus métodos en el fichero `csu.c`.

En `element.h` se encuentran definidas las constantes que se utilizan en el código, incluidas aquellas que definen el tamaño de una `CSU` y de sus barras. Como se ha comentado en capítulos anteriores, una `CSU` presenta un tamaño de 400 arcosegundos de alto y 240 de ancho. Sin embargo, puesto que a la hora de rotar el apuntado era necesario realizar operaciones trigonométricas que requerían de la mitad de esos valores, se ha declarado como ancho de una `CSU` el valor de 120 y su alto como 200.

```

1 const int ANCHO = 120; /*< Mitad del ancho de la CSU*/
2 const int ALTO = 200; /*< Mitad del alto de la CSU*/

```

Listado 5.3: Constantes de tamaño de la `CSU`

#### 5.3.3.1. Atributos de CSU

Al igual que ocurría en la clase `Element`, una instancia de `CSU` está definida por un par de coordenadas ( $X, Y$ ), al que se le añade un ángulo de rotación

cuyos valores se encuentran en el rango [0, 180], por los motivos mencionados en el Capítulo 1. A continuación se definen los atributos de CSU:

- **identificador**: común para todas las instancias; sirve para asignarle un valor único autoincremental al **id** del apuntado.
- **orden**: establece la manera de ordenar los apuntados. Puede tomar los valores **MENORX**, **MAYORX**, **MENORY**, **MAYORY**, **PRIORI** y **TAMPUN**.
- **id**: identificador del apuntado, usado para diferenciar instancias de esta clase.
- **x** e **y**: son las coordenadas, en el eje X y en el eje Y respectivamente, del centro del apuntado.
- **alfa**: define el ángulo de rotación del apuntado, el cual gira en sentido anti horario. El valor se establece en radianes, en el rango  $[0, \pi]$ . Cuando la CSU tiene  $\alpha = 0$  se encuentra en posición vertical, y cuando tiene valor de  $\frac{\pi}{2}$  se encuentra en posición horizontal.
- **prior**: prioridad del apuntado, definida por la mayor prioridad de los puntos que pertenecen a la instancia.
- **orientacion**: indica qué lado de la barra se está tomando como válido durante la técnica de beam switching. Toma los valores de **ARRIBA** y **ABAJO** para indicar el tercio correcto de la barra.
- **puntos**: conjunto de puntos que se encuentran en el interior del apuntado y para los cuales hay una barra asociada. Por ésto último el número máximo de puntos permitido es el del número de barras de la CSU, 55.
- **Ax**, **Ay**, **Bx**, **By**, **Cx**, **Cy**, **Dx**, **Dy**: estos atributos corresponden a las coordenadas de cada uno de los vértices que delimitan la CSU (ver Figura 5.1).
- **ma**, **mb**: son las pendientes de la recta formada por los vértices *A-D* y su perpendicular, respectivamente.
- **barras\_ocupadas**: recoge el estado de las barras, marcando aquellas que se ocupan con un objeto.
- **distancia\_punto**: guarda la distancia de cada objeto con el borde izquierdo del apuntado para poder situar correctamente el borde que separa cada par de barras.

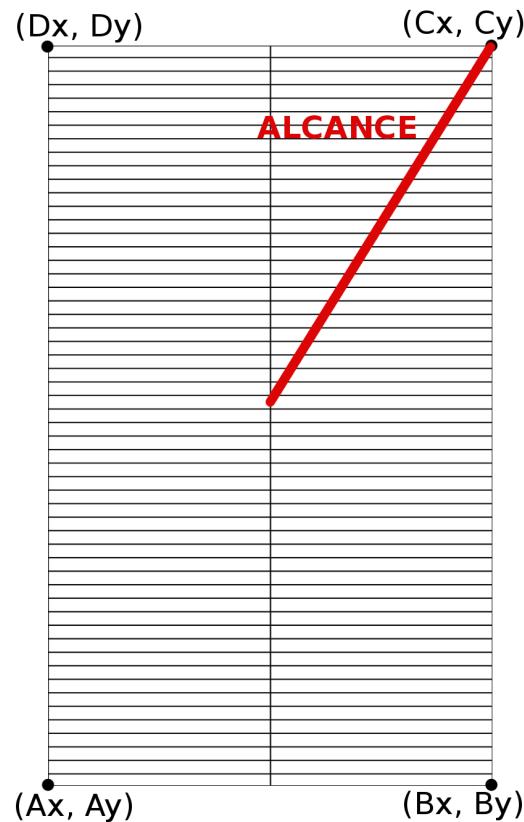


Figura 5.1: Vértices de la CSU

- `dist_min`, `dist_max`: hacen referencia a los valores mínimos y máximos almacenados en el atributo `distancia_punto`, indicando la separación lateral existente entre los puntos y los bordes. El segundo atributo es en realidad la distancia mínima respecto al borde derecho, pero se ha optado por denominarlo de esa forma al coincidir con la máxima del borde izquierdo.
- `marginUp`, `marginDown`: indican cuál es la distancia mínima entre un objeto y los límites superiores e inferiores.
- `limiteS`, `limiteI`: estos límites indican dónde acaba la zona útil de una barra.

```

1 #define edge(x1, y1, x2, y2) (- (y2 - y1) * p.getx() + (x2 - x1) * p.gety() -
2   (- (y2 - y1) * x1 + (x2 - x1) * y1))
3 bool CSU::estaDentro2(const Element &p) const {
4   set<Element>::iterator it = puntos.find(p);
5   if (it != puntos.end())
6     return false;
7   register double distancia = (double)sqrt((p.getx() - x) * (p.getx() - x)
8     + (p.gety() - y) * (p.gety() - y));
9   if (distancia <= ALCANCE) {
10     static int D1, D2, D3, D4;
11     D1 = edge(Ax, Ay, Dx, Dy);
12     D2 = edge(Dx, Dy, Cx, Cy);
13     D3 = edge(Cx, Cy, Bx, By);
14     D4 = edge(Bx, By, Ax, Ay);
15     return (D1 <= 0 && D2 <= 0 && D3 <= 0 && D4 <= 0);
16   }
17   return false;
18 }
```

Listado 5.4: Código del ray casting simplificado

### 5.3.3.2. Métodos relevantes de CSU

Uno de los métodos más importantes de esta clase es aquél en el que se decide si un objeto está dentro o no del apuntado. El método utiliza una simplificación del método conocido como “ray casting”, utilizado en el desarrollo de videojuegos para determinar colisiones entre polígonos. Este método verifica si un punto se encuentra situado a la izquierda o a la derecha de uno de los vértices del polígono. En nuestro caso, si un objeto se encuentra a la izquierda de todas las aristas que forman el rectángulo del apuntado, significa que está dentro del mismo. El código que comprueba esto es la función auxiliar `estaDentro2`, que se muestra en el Listado 5.4.

*ALCANCE* es una constante cuyo valor corresponde a la distancia entre el centro del apuntado y cualquiera de sus vértices (ver Figura 5.1). Si un punto está a una distancia superior a esa, está fuera del radio de acción de la CSU, por lo que no podrá estar dentro sea cual sea el ángulo  $\alpha$  del apuntado. Si un objeto se encuentra entre los vértices, se comprueba de que su posición sea válida respecto a las barras. Esto se hace en la función pública `estaDentro`, cuyo código se muestra en el Listado 5.5.

La función `testeo` calcula el punto de intersección entre la recta formada

```

1 int CSU::estaDentro(const Element &p, int &barra, double &distancia,
2     double &rango) const {
3     static double new_x, new_y, zone;
4     if (estaDentro2(p)) {
5         testeo(p, new_x, new_y);
6         zone = sqrt((Ax - new_x) * (Ax - new_x) + (Ay - new_y) * (Ay - new_y))
7             ;
8         barra = (int)(zone / DIST_BARRAS);
9         if (barra == NUM_BARRAS)
10            return 0;
11         rango = zone - (barra * DIST_BARRAS);
12         distancia = (double)sqrt((p.getx() - new_x) * (p.getx() - new_x)
13             + (p.gety() - new_y) * (p.gety() - new_y));
14         if (distancia > 2*ANCHO)
15            return 0;
16         if (barras_ocupadas[barra])
17             return p_potencial(rango, barra);
18         //Evitamos que esten justo donde van las barras
19         static double redondeo; //demasiada precision en doubles
20         redondeo = ceil(rango * PRECISION) / PRECISION;
21         if (redondeo < BORDE_INF_DEF || redondeo > BORDE_SUP_DEF){
22             return p_potencial(rango, barra);
23         }
24         //Miramos los bordes (1 arcseg por cada lado)
25         if (conf.noborder)
26             if (redondeo <= BORDE_INF || redondeo >= BORDE_SUP)
27                 return p_potencial(rango, barra);
28         if (conf.beamswitching) {
29             if (orientacion == ARRIBA)
30                 if (redondeo < BARRA2_3)
31                     return p_potencial(rango, barra);
32             if (orientacion == ABAJO)
33                 if (redondeo > BARRA1_3)
34                     return p_potencial(rango, barra);
35         }
36         return !barras_ocupadas[barra];
37     }
38     return 0;
39 }
```

Listado 5.5: Función que comprueba si un objeto está en la CSU

por los vértices A y D de la CSU (lado izquierdo) y un objeto dado. Cuando el método `estaDentro` comprueba que el punto coincide perfectamente con una barra libre, devuelve el valor 1, en caso contrario hace una llamada al método `p_potencial`, que comprueba si el objeto podría pertenecer al apuntado si se le realizara un mínimo movimiento de ajuste al mismo. Dicho método devuelve los valores que aparecen en la Tabla 5.1.

	Objeto en parte superior	Objeto en parte inferior
Barra actual libre	2	5
Barra superior libre	3	0
Barra inferior libre	0	4

Tabla 5.1: Posibles valores del método `p_potencial`

Los resultados pares indican que se ha de subir el centro del apuntado, de manera que los objetos “bajen”, los resultados impares implican lo contrario, y el valor 0 indica que ese punto no puede pertenecer a la CSU puesto que no ha superado las condiciones necesarias. Las restricciones que se usan se pueden observar en el Listado 5.6.

Cuando un punto se ajusta perfectamente con una barra y no hay que mover el apuntado, se utiliza el método `pointAdd`, que comprueba los límites y márgenes del apuntado respecto al nuevo punto, lo introduce en el conjunto de puntos que componen el apuntado y marca la barra en la que se encuentra como ocupada. Sin embargo, cuando hay que hacer ajustes mínimos para que el objeto pueda ser introducido, se utiliza la función `pointAddMove`, cuya implementación se muestra en el Listado 5.7.

En la función `pointAddMove` se calcula cuál es la distancia mínima necesaria para que el objeto pueda ser introducido, dependiendo de la opción en la que se encuentre el mismo (véase la Tabla 5.1). Una vez movido el centro de la CSU se invocan dos métodos muy importantes de esta clase. El primero, `extremos`, recalcula la posición de los vértices del apuntado, puesto que se ha movido de sitio; y el segundo, `actualizar2`, se encarga de comprobar de nuevo todos aquellos atributos que dependen de la posición de los objetos, tales como las distancias mínimas recogidas en el atributo `distancia_punto` o los nuevos márgenes de movimiento del apuntado.

En el Capítulo 4 se hacía referencia a una serie de movimientos de mejora que se podían efectuar sobre la CSU, con el objetivo de abarcar un mayor número de objetos. Estos posibles movimientos son cuatro y se agrupan en

```
1 int CSU::p_potencial(const double &rango, int &barra) const {
2     if (rango > limiteS) { //Parte superior de la barra
3         if (!barras_ocupadas[barra] && rango - limiteS + 0.01 <= margenDown)
4             return 2;
5         if (barra < NUM_BARRAS-1 && !barras_ocupadas[barra+1])
6             if (DIST_BARRAS - rango + limiteI + 0.01 <= margenUp) {
7                 barra++;
8                 return 3;
9             }
10    }
11    if (rango < limiteI) { //Parte inferior de la barra
12        if (limiteI - rango + 0.01 <= margenUp) {
13            if (barras_ocupadas[barra])
14                return 0;
15            else
16                return 5;
17        }
18        if (barra > 0 && !barras_ocupadas[barra-1])
19            if (rango + (DIST_BARRAS-limiteS) + 0.01 <= margenDown) {
20                barra--;
21                return 4;
22            }
23    }
24    return 0;
25 }
```

Listado 5.6: Función que comprueba si un objeto podría estar en la CSU

```

1 void CSU::pointAddMove(Element &p, const int &move) {
2     static double dist;
3     if (move == 2 || move == 4) { //Hay que subir la CSU
4         if (move == 2) //esta muy arriba
5             dist = p.getzona() - limiteS + 0.01;
6         if (move == 4) //pasar a barra inferior
7             dist = p.getzona() + (DIST_BARRAS - limiteS) + 0.01;
8         if (alfa == CERO) { //es vertical, subimos la Y
9             y += dist;
10        }
11        else if (alfa == PIMEDIO) { //es horizontal, movemos la X
12            x -= dist;
13        }
14        else {
15            static double raiz;
16            raiz = sqrt((dist*dist)/(1 + ma * ma));
17            y -= ma * raiz;
18            x -= raiz;
19        }
20        margenDown -= dist;
21        margenUp += dist;
22    }
23    else { //Hay que bajar la CSU
24        if (move == 3) //Pasar a barra superior
25            dist = DIST_BARRAS - p.getzona() + limiteI + 0.01;
26        if (move == 5) { //Esta demasiado abajo
27            dist = limiteI - p.getzona() + 0.01;
28        }
29        if (alfa == CERO) { //es vertical, bajamos la Y
30            y -= dist;
31        }
32        else if (alfa == PIMEDIO) { //es horizontal, movemos la X
33            x += dist;
34        }
35        else {
36            static double raiz;
37            raiz = sqrt((dist*dist)/(1 + ma * ma));
38            y += ma * raiz;
39            x += raiz;
40        }
41        margenDown += dist;
42        margenUp -= dist;
43    }
44    extremos();
45    puntos.insert(p);
46    barras_ocupadas[p.getbarra()] = true;
47    if (p.getprior() < prior)
48        prior = p.getprior();
49    actualizar2();
50}

```

Listado 5.7: Método que añade un punto que requiere de ciertos ajustes

dos: movimientos verticales y movimientos horizontales. En los primeros, se produce un cambio de barra de los objetos, situando aquél que se encuentre en la barra más cercana al borde inferior de la CSU en la barra 0 del apuntado (en el caso de la mejora hacia arriba) o trasladando el objeto que ocupa la barra más cercana al borde superior en la última barra (en el otro caso). En este tipo de movimiento la zona de la barra en la que se encuentra un objeto no se ve afectada. Por el contrario, en el segundo tipo de movimiento, el horizontal, lo que no se ve afectado es la barra de cada uno. En este caso lo que se persigue es mover los objetos hasta el borde izquierdo o derecho del apuntado, obteniendo una nueva área de búsqueda alrededor de la CSU. Nuevamente se ha de llamar al método **extremos** para calcular los nuevos vértices del apuntado con cada movimiento. En el Listado 5.8 se muestra un ejemplo de cada uno de los tipos de mejora mencionados.

En ambos casos se utiliza la función **actualizar**, que funciona de modo análogo a **actualizar2**, actualizando los objetos y el contenido de los atributos.

El método **colision** es otra de las claves de esta clase, pues se encarga de determinar si entre dos apuntados existe una colisión o solape. La forma de comprobarlo es mediante proyecciones de planos e intersecciones de rectas: se proyectan los vértices de una CSU *B* sobre las rectas definidas por los vértices AD y AB de la CSU *A*. Si en alguna de estas proyecciones el rango dado por el par de vértices (mínimo y máximo) proyectados no tiene puntos en común con el rango formado por el par de vértices AD, o AB según el caso, podemos afirmar que no hay colisión. En caso de no poder realizar tal afirmación, comprobamos si existe algún punto de *A* en *B*, en cuyo caso se afirmaría que existe un solapamiento entre ambas. Se debe tener en cuenta que aunque se llegue al final del método y no haya ningún punto de *A* en *B*, es posible que sí haya alguno de *B* en *A*, por lo que se debe hacer doble comprobación. El código de este método se muestra en el Listado 5.10.

### 5.3.4. La clase Parser

**Parser** hereda de la clase **SaxParser**, definida en la librería **libxml++**, cuya función es actuar como un parser de XML. **Parser** es una adaptación a nuestro problema particular de los métodos que aparecen en la documentación de la librería. Sus objetivos son los de comprobar que la estructura del fichero de entrada a **CSUOptimizer** sea el correcto y

```
1 void CSU::mejoraArriba() {
2     static int i, j, z;
3     static double dist;
4     i = CERO;
5     while (!barras_ocupadas[i])
6         i++;
7     if (i != CERO) {
8         for (j = CERO; j+i < NUM_BARRAS; j++){
9             barras_ocupadas[j] = barras_ocupadas[j+i];
10        }
11        for (j = NUM_BARRAS - i; j < NUM_BARRAS; j++){
12            barras_ocupadas[j] = false;
13        }
14        dist = i * DIST_BARRAS;
15        if (alfa == CERO) { //es vertical, subimos la Y
16            y += dist;
17        }
18        else if (alfa == PIMEDIO) { //es horizontal, movemos la X
19            x -= dist;
20        }
21        else {
22            static double raiz;
23            raiz = sqrt((dist*dist)/(1 + ma * ma));
24            y -= ma * raiz;
25            x -= raiz;
26        }
27        extremos();
28        actualizar(-i);
29    }
30 }
```

Listado 5.8: Código del movimiento de mejora vertical hacia arriba

```
1 void CSU::mejoraIzquierda() {
2     static int j;
3     static double dist;
4     if (dist_min > 0.2) {
5         dist = dist_min - 0.1;
6         if (alfa == CERO) { //es vertical, movemos la X
7             x += dist;
8         }
9         else if (alfa == PIMEDIO) { //es horizontal, movemos la Y
10            y += dist;
11        }
12        else {
13            static double raiz;
14            raiz = sqrt((dist*dist)/(1 + mb * mb));
15            if (alfa < PIMEDIO) {
16                x += raiz;
17                y += mb * raiz;
18            } else {
19                x -= raiz;
20                y -= mb * raiz;
21            }
22        }
23        extremos();
24        actualizar(0);
25    }
26 }
```

Listado 5.9: Código del movimiento de mejora horizontal izquierdo

```
bool CSU::colision (const CSU &csu) const {
    csu.getABCD(s2); //Ax,Ay,Bx,By,Cx,Cy,Dx,Dy
    //RECTA AD
    for (i = 0; i < 8; i+=2) { //cada uno de los puntos de CSU2
        if (alfa != CERO && alfa != PIMEDIO) { // No es ni vertical ni
            horizontal
            punto.first = (ma * Ax - mb * s2[i] + s2[i+1] - Ay) / (ma - mb);
            punto.second = ma * (punto.first - Ax) + Ay;
        }
        else if (alfa == CERO){ // Es vertical
            punto.first = Ax;
            punto.second = s2[i+1];
        }
        else{ // Es horizontal
            punto.first = s2[i];
            punto.second = Ay;
        }
        minimoX = min(minimoX, punto.first);
        maximoX = max(maximoX, punto.first);
        minimoY = min(minimoY, punto.second);
        maximoY = max(maximoY, punto.second);
    }
    //Comprobamos que no haya solape
    miX = min(Ax,Dx); maX = max(Ax,Dx);
    if (maximoX < miX)
        return false;
    if (minimoX > maX)
        return false;
    miX = min(Ay,Dy); maX = max(Ay,Dy);
    if (maximoY < miX)
        return false;
    if (minimoY > maX)
        return false;
    //Repetimos para el lado DC
    //Comprobamos si los puntos colisionan
    CSU vacia (csu.getx(), csu.gety(), csu.getalfa());
    for (p = puntos.begin(); p != puntos.end(); p++)
        if (vacia.estaDentro(*p,b,d,z))
            return true;
    return false;
}
```

Listado 5.10: Código del método que comprueba las colisiones

```
<Observables [widht='Ancho_cielo', height='Alto_cielo']>
  <Element id='Identificador'>
    <X>PosX</X>
    <Y>PosY</Y>
    <Prioridad>[1..99]</Prioridad>
  </Element>
  <!-- ... -->
</Observables>
```

Listado 5.11: Estructura de los ficheros de entrada para CSUOptimizer

devolver los datos contenidos en dicha entrada en las estructuras explicadas anteriormente. La estructura del fichero de entrada es la que se muestra en el Listado 5.11.

En el Listado 5.12 se muestra un extracto de un ejemplo de entrada con datos reales, extraídos del fichero `exacto.xml`.

Los datos facilitados a la aplicación han de estar en arcosegundos, y nunca con valores negativos. En caso de tener una instancia de entrada que no cumpla este requisito puede utilizarse el conversor que se explica en la Sección 5.5.2.

#### 5.3.4.1. Atributos de Parser

- `Q`: conjunto de objetos contenidos en el fichero de entrada.
- `fich`: fichero XML de entrada.
- `punto`: objeto donde se crean las instancias de los elementos contenidos en el archivo entrante.
- `nodo`: identificador del nodo en el que se encuentra el analizador durante el procesado.
- `nuevo`: indica el nivel de procesado que lleva en la estructura, en nuestro caso cuando llega a 3 se determina que ha completado un elemento.
- `widht, height`: ancho y alto especificados para definir el tamaño de la pantalla de Allegro.

```
<Observables width="900" height="900">
    <Element id="0">
        <x>573</x>
        <y>402.4242</y>
        <Prioridad>1</Prioridad>
    </Element>
    <Element id="1">
        <x>595</x>
        <y>409.6968</y>
        <Prioridad>1</Prioridad>
    </Element>
    <Element id="2">
        <x>458</x>
        <y>416.9694</y>
        <Prioridad>1</Prioridad>
    </Element>
    <Element id="3">
        <x>409</x>
        <y>424.242</y>
        <Prioridad>1</Prioridad>
    </Element>
    <!-- ... -->
    <Element id="54">
        <x>409</x>
        <y>795.1446</y>
        <Prioridad>1</Prioridad>
    </Element>
</Observables>
```

Listado 5.12: Ejemplo de entrada para CSUOptimizer, extraído de `exacto.xml`

### 5.3.4.2. Métodos relevantes de Parser

Cuando se construye una instancia de esta clase comienza el procesado del archivo especificado por parámetro. En el constructor se comprueba línea a línea si el fichero sigue la estructura especificada, mediante llamadas a las funciones virtuales de la clase. Éstas son métodos que especifican qué se comprueba en cada nivel y cuando se ha terminado de construir un objeto para añadirlo a Q. El método que realmente interesa es `getList`, que devuelve el conjunto de datos de entrada del problema.

## 5.3.5. La clase Dbscan

Esta es la clase que implementa el algoritmo descrito en la Sección 4.1. El código de `Dbscan` es el resultado de una adaptación y traducción a C++ de un código Java que implementaba este algoritmo de clustering. Se ha mantenido el tipo de los atributos para evitar alterar el correcto funcionamiento del código.

Los objetos (de la clase `Element`) se agrupan en vectores, definidos en la clase estándar `vector` de C++. Este contenedor agrupa los objetos por orden de llegada.

### 5.3.5.1. Atributos de Dbscan

- `e`: indica la distancia utilizada en el cálculo de vecindades.
- `minpt`: mínimo de puntos que forman un núcleo.
- `resultList`: vector de listas de vectores de elementos. Cada lista representa un clúster de puntos.
- `pointList`: es el conjunto de datos de entrada.

### 5.3.5.2. Métodos relevantes de Dbscan

Se utiliza `setList` y `getList` para establecer u obtener el conjunto de objetos de entrada, y el método `start` para comenzar el proceso de

```

set<CSU> diferentes[5];
sort(diferentes[0], TAMPUN);
int mej, min = MAX;
CLOCK_Start(chrono);
for (int i = MENORX; i < 5; i++) {
    diferentes[i] = apuntadosRandom(parser.getList(), i);
    if (diferentes[i].size() < min) {
        min = diferentes[i].size();
        mej = i;
    }
}
set<CSU> centros = diferentes[mej];

```

Listado 5.13: Primera fase del algoritmo constructivo: generar la solución inicial

agrupamiento de los objetos. Para obtener el vector de resultados, contenido en el atributo `resultList` se usa el método `getResult`.

### 5.3.6. Algoritmos principales

El archivo `main.C` organiza el uso de todas las clases descritas para implementar el algoritmo constructivo planteado en la Sección 4.3. El código para crear la solución inicial de dicho algoritmo es el que se presenta en el Listado 5.13.

En el Listado 5.14 se detalla la función `apuntadosRandom`, encargada de generar varias soluciones del problema sobre las que se elige la mejor.

La segunda fase, en la que se realiza el proceso de eliminar solapes entre apuntados, se realiza dentro de `colisiones`. Este método se encuentra detallado en el Apéndice A.1 debido a su extensión.

La implementación del algoritmo GRASP definido en la Sección 4.2 se encuentra en la función `grasp` del fichero `main.C`, mostrada en el Apéndice A.2.

```
set<CSU> apuntadosRandom(set<Element> puntos, const int &opcion) {
    static set<Element>::iterator v;
    static set<CSU> posibles;
    set<CSU> resultado;
    posibles.clear();
    static set<CSU>::iterator i;
    static CSU mejor_opcion;
    static int contador;
    if (opcion <= 4) {
        sort(puntos, opcion);
        set<Element> eliminados;
        set<Element> cluster = puntos;
        static set<Element>::iterator e1;
        while (!puntos.empty())
            for (v = cluster.begin(); v != cluster.end(); v++) {
                progreso();
                e1 = eliminados.find(*v);
                if (e1 == eliminados.end()) {
                    posibles = selectAP(*v, puntos);
                    i = posibles.begin();
                    mejor_opcion = *i;
                    resultado.insert(mejor_opcion);
                    eliminar_puntos(mejor_opcion.getPoints(), puntos, eliminados);
                }
            }
        }
        // ... //
    return resultado;
}
```

Listado 5.14: Código de la función que genera soluciones aleatorias

## 5.4. Modo de empleo de CSUOptimizer

Para utilizar `CSUOptimizer` una vez compilada (usando `make`), hay que especificar una serie de opciones y un archivo XML con los objetos de entrada.

```
$ ./csuoptimizer [opciones] entrada.xml
```

Las opciones disponibles en `CSUOptimizer` son:

- `--help`: Muestra una ayuda con las opciones disponibles y un ejemplo de uso.
- `--dbscan`: indica a `CSUOptimizer` que se quiere hacer uso del algoritmo DBScan explicado en la Sección 5.3.5. Desactivado por defecto.
- `--grasp`: habilita el uso de la heurística GRASP para intentar mejorar la solución. Desactivada por defecto.
- `--graphic`: muestra en la ventana generada por Allegro los pasos que está realizando `CSUOptimizer` durante su ejecución. Este paso está pensado para labores de depuración y desarrollo y ralentiza la ejecución de la aplicación.
- `-NR`: especifica el número de rotaciones que hará una CSU durante el proceso de búsqueda de objetos inicial. Por defecto este valor es igual a 20, pues se han obtenido buenos resultados en todos los ejemplos. Un ejemplo de su uso sería `./csuoptimizer -NR 15 entrada.xml`.
- `--verbose`: genera un archivo `verbose.txt` con información de control, útil para depuración y desarrollo.
- `--dat`: indica el modo de salida de los resultados. Por defecto se usa el formato XML, cuyo ejemplo se puede ver en el Anexo A.3, sin embargo es posible que el usuario requiera de otro tipo de salida. Esta estructura adicional se muestra en el Listado 5.15.
- `--beams`: habilita el uso del `beam switching`.
- `--noborder`: especifica que no se quieren objetos cercanos a la barra, por lo que se deshabilita un espacio de 1 arcosegundo en cada uno de los bordes.

```
*****
coordenadaX     coordenadaY     anguloInclinacion
    posicionBarra_1
    posicionBarra_2
:
:
    posicionBarra_55
*****
coordenadaX     coordenadaY     anguloInclinacion
:
:
```

Listado 5.15: Estructura del modo de resultado .dat

Las opciones `--beams` y `--noborder` afectan al área útil de una barra. En la Figura 5.2 se muestran los casos posibles de una barra. La zona blanca es la zona útil de barra, donde se puede detectar un objeto, la zona en color gris hace referencia al área eliminada por el usuario mediante las opciones.

Figura 5.2: Disponibilidad de las barras según las opciones de entrada. a) Barra normal. b) Barra con `--noborder`. c) Barra con `--beams`. d) Barra con ambas opciones

## 5.5. Herramientas de apoyo

Con el objetivo de comprobar el buen funcionamiento de `CSUOptimizer` y de facilitar al usuario herramientas útiles que permitan una mayor familiarización con la aplicación, se aportan dos herramientas complementarias explicadas en las siguientes Secciones.

### 5.5.1. “Editor de mapas”

En el directorio `map_editor` del proyecto se encuentra un programa `mapEditor` que genera entradas para el `CSUOptimizer` con apuntados específicos. Los ejemplos sintéticos que se presentan en la Sección 6.1 de esta Memoria han sido generados mediante esta aplicación. Se cuenta con un archivo `Makefile` para automatizar la compilación.

El `mapEditor` generado por la compilación muestra un menú por consola con las siguientes opciones:

- 1 **Crear CSU**: permite especificar la posición de una CSU así como su ángulo de inclinación. Automáticamente se añade un objeto en cada barra de forma aleatoria. La pantalla gráfica de esta aplicación tiene un tamaño de  $900 \times 900$  arcosegundos, por lo que se deben introducir valores en este rango para poder visualizarlos.
- 2 **Visualizar CSUs**: muestra la disposición de las CSUs en el espacio visible.
- 3 **Visualizar Puntos**: muestra únicamente la nube de objetos generada, la cual será la entrada a la aplicación `CSUOptimizer`.
- 4 **Visualizar Espacio**: muestra de manera conjunta las dos opciones anteriores, es decir, cada apuntado con sus respectivos objetos.
- 5 **Guardar puntos**: genera un archivo XML con los objetos creados, el cual será el fichero de entrada de `CSUOptimizer`.
- 6 **Salvar datos**: guarda el estado actual de la aplicación en un fichero `save.dat` para reutilizarlo cuando convenga.
- 7 **Cargar datos**: carga en memoria un estado antiguo de la aplicación desde un fichero especificado por el usuario.

### 5.5.2. Conversor

Los ejemplos facilitados por el responsable del proyecto EMIR no se presentan en el formato de entrada requerido por `CSUOptimizer`, razón por la que se ha incluido un conversor que:

```
./conversor-dat-xml [-g2a] archivo.dat
```

- transforma un fichero con datos en texto plano agrupados por columnas (véase el Listado 5.16) al formato de entrada requerido.
- realiza un cambio de coordenadas para evitar valores negativos en CSUOptimizer.
- permite cambiar de grados a arcosegundos con la opción -g2a.

Esta herramienta genera dos archivos de salida:

- Uno con la misma estructura que la entrada, es decir, la que se muestra en el Listado 5.16, terminado en `-converted.dat` en `ejemplos/` para que se tenga un control de cambios en los objetos debido al cambio del centro de origen.
- Un fichero con el mismo nombre que el de origen pero en XML situado en `ejemplos/xml/` listo para su uso en CSUOptimizer.

Ejemplo de uso:

```
1 posicionXobjeto1 posicionYobjeto1 prioridadObjeto1
2 posicionXobjeto2 posicionYobjeto2 prioridadObjeto2
3
4     ...
5     ...
6     ...
6 posicionXobjetoN posicionYobjetoN prioridadObjetoN
```

Listado 5.16: Estructura de entrada de los ficheros .dat

# Capítulo 6

## Resultados

En este capítulo se presentan los resultados obtenidos con diversas distribuciones de objetos, tanto generadas de forma sintética y ex-profeso para comprobar la aplicación, como correspondientes a casos reales. Todos estos ejemplos se incluyen en el directorio `ejemplos/` de la aplicación.

Los resultados se muestran en tablas que recogen el número de CSUs que constituye la solución junto al tiempo (en minutos y segundos) que ha tardado en calcularse la misma, dependiendo de las opciones que se utilicen en la ejecución. Las opciones se muestran a la izquierda, por filas, mientras que el tipo de resultado se muestra en la parte superior, en las columnas.

También se muestran pares de imágenes, correspondientes a la entrada y la salida (sin especificación de ningún tipo de parámetros adicionales) del programa, para facilitar la comprensión del ejemplo. La figura de la izquierda representa la nube de objetos sobre la que el `CSUOptimizer` va a buscar los apuntados, y la figura de la derecha muestra el resultado obtenido. Se dibuja de un color diferente cada apuntado para facilitar su diferenciación.

Para los cálculos se ha utilizado una máquina virtual con Ubuntu 12.04 LTS, dos procesadores y 2 GB de memoria RAM, que corre bajo un Windows 7 nativo, con un Intel i7 920 (2.67 GHz) y 6 GB de memoria RAM DDR3.

## 6.1. Ejemplos sintéticos

Son ejemplos generados por el “editor de mapas” (`mapEditor`) para depurar la aplicación y comprobar su comportamiento. Algunas de las distribuciones se generaron de forma aleatoria, por lo que no hay manera de saber si los resultados son óptimos o no. Otros ejemplos se han generado siguiendo un patrón concreto, para el que conocemos de antemano la solución óptima. Para ello situamos en el espacio de trabajo una CSU “virtual” sobre la que colocamos un cierto número de objetos distribuidos de tal modo que sabemos que ajustan perfectamente con la barra de esta CSU “virtual”. Si el `CSUOptimizer` realiza correctamente su trabajo, colocará una CSU (apuntado) configurado exactamente igual que la CSU “virtual” que ubicamos con el `mapEditor`.

### 6.1.1. Ejemplo: `exacto.xml`

Este ejemplo consiste en 55 objetos a observar distribuidos de forma que cada uno de ellos ajuste perfectamente dentro de una barra de la CSU, siendo el resultado óptimo un único apuntado. Al tratarse de un ejemplo creado con el “editor de mapas” la distancia de cada punto es la necesaria para que el resultado no varíe entre las diversas opciones que se permiten.

RESULTADOS	Apuntados	Tiempo
Sin opciones	1	1.02”
Sin bordes	1	0.76”
Beam Switching	1	0.72”
Sin bordes + Beam S.	1	0.89”

Tabla 6.1: Resultados del ejemplo `exacto.xml`

El resultado es justamente el esperado, por lo que comprobamos que la aplicación es capaz de ajustar correctamente un apuntado de manera óptima para este tipo de instancias de entrada.

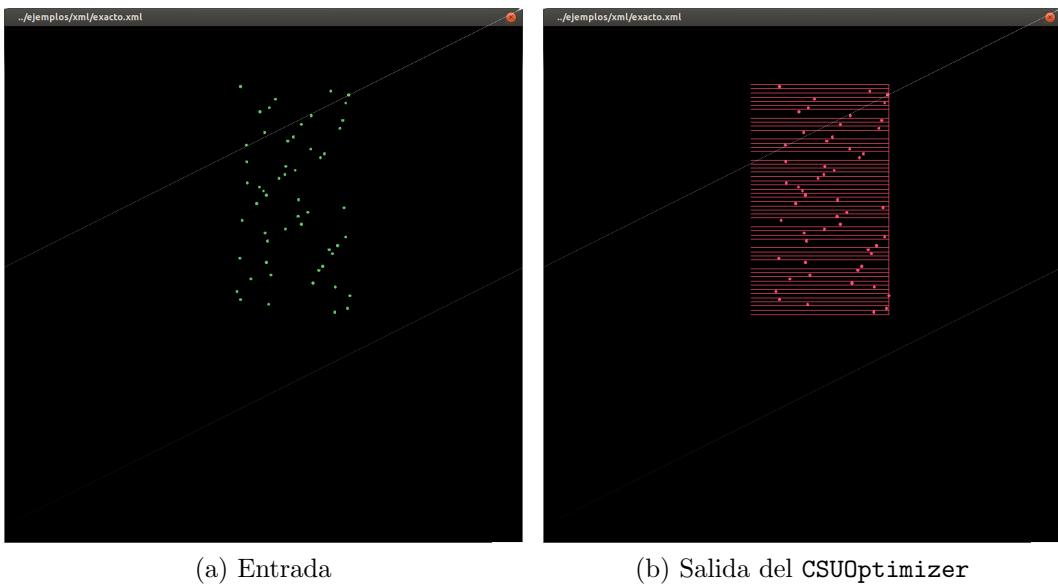


Figura 6.1: Ejemplo `exacto.xml`. 55 objetos que se ajustan perfectamente a una única CSU

### 6.1.2. Ejemplo: `4_0solapes.xml`

Este ejemplo consta de 220 objetos, que corresponden a cuatro apuntados generados de forma que ninguno de ellos se solape con otro, con lo que el resultado esperado son cuatro apuntados.

RESULTADOS	Apuntados	Tiempo
Sin opciones	4	1.72"
Sin bordes	5	2.97"
Beam Switching	4	3.09"
Sin bordes + Beam S.	4	3.19"

Tabla 6.2: Resultados del ejemplo `4_0solapes.xml`

Comprobamos en la Tabla 6.2 que la aplicación se comporta tal y como se pretende, salvo en uno de los casos. Este fallo de un apuntado de más se debe a que en la solución inicial se crean 6 apuntados, cuya disposición ocasiona que existan dos parejas de apuntados que solapan entre sí. Cuando el método de reducción de colisiones evalúa estos solapes, consigue reducir aquella pareja que se ha creado indebidamente, sin embargo la otra pareja corresponde

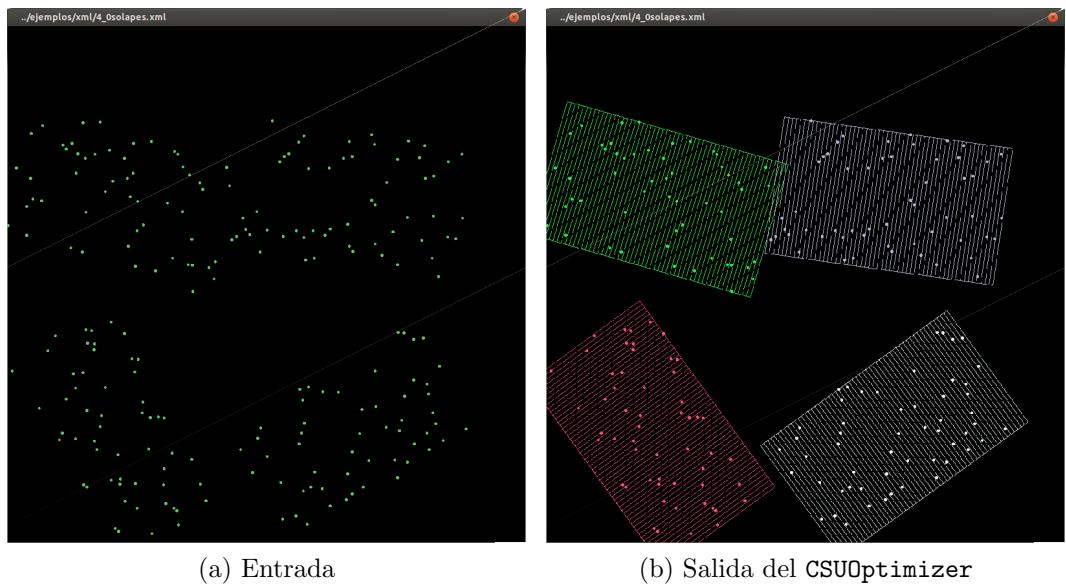


Figura 6.2: Ejemplo 4\_0solapes.xml. 4 apuntados distanciados entre si

con un apuntado completo (con 55 objetos) y otro con un único objeto, de forma que ambos apuntados son necesarios. Dada la disposición entre los apuntados generados no se detectan colisiones a resolver, ocasionando el resultado obtenido.

### 6.1.3. Ejemplo: 4\_2solapes.xml

Este ejemplo tiene el mismo número de objetos que el anterior, 220, pero en este caso se ha forzado a que dos apuntados solapen notablemente, para probar la calidad del algoritmo que reduce las colisiones. Debido al alto grado de solape existente se espera una solución relativamente mala.

RESULTADOS	Apuntados	Tiempo
Sin opciones	5	4.26"
Sin bordes	5	3.64"
Beam Switching	5	4.29"
Sin bordes + Beam S.	5	3.67"

Tabla 6.3: Resultados del ejemplo 4\_2solapes.xml

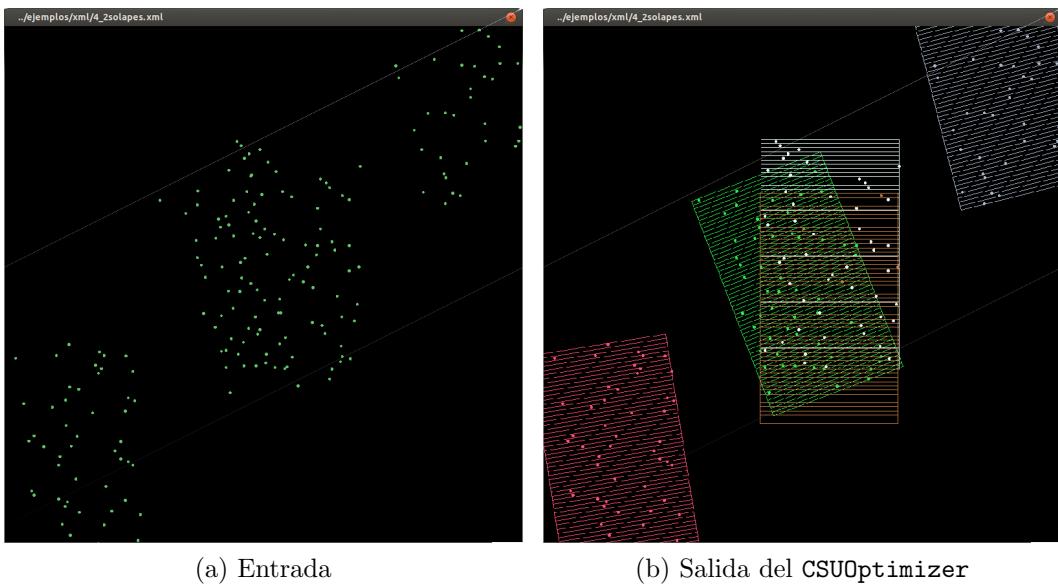


Figura 6.3: Ejemplo 4\_2solapes.xml. Ejemplo de “colisión” de dos apuntados

Pese al alto grado de solapamiento, la aplicación sólo ha introducido un apuntado de más, por lo que se considera que el algoritmo ofrece resultados de razonable calidad teniendo en cuenta el tiempo de ejecución.

#### 6.1.4. Ejemplo: denso.xml

Este ejemplo no ha sido generado con la herramienta `mapEditor` sino que se ha creado mediante una distribución aleatoria de los objetos, razón por la cual no podemos conocer a priori el resultado óptimo. El ejemplo nos sirve para estudiar cómo se comporta la aplicación ante grandes nubes de objetos con una distribución espacial homogéneas. El ejemplo consta de 999 puntos.

#### 6.1.5. Ejemplo: denso2000.xml

Se trata del mismo ejemplo anterior, incrementando el número de objetos hasta 2000. Esta entrada se generó con el fin de evaluar el tiempo tarda el `CSUOptimizer` en generar soluciones para ejemplos relativamente grandes.

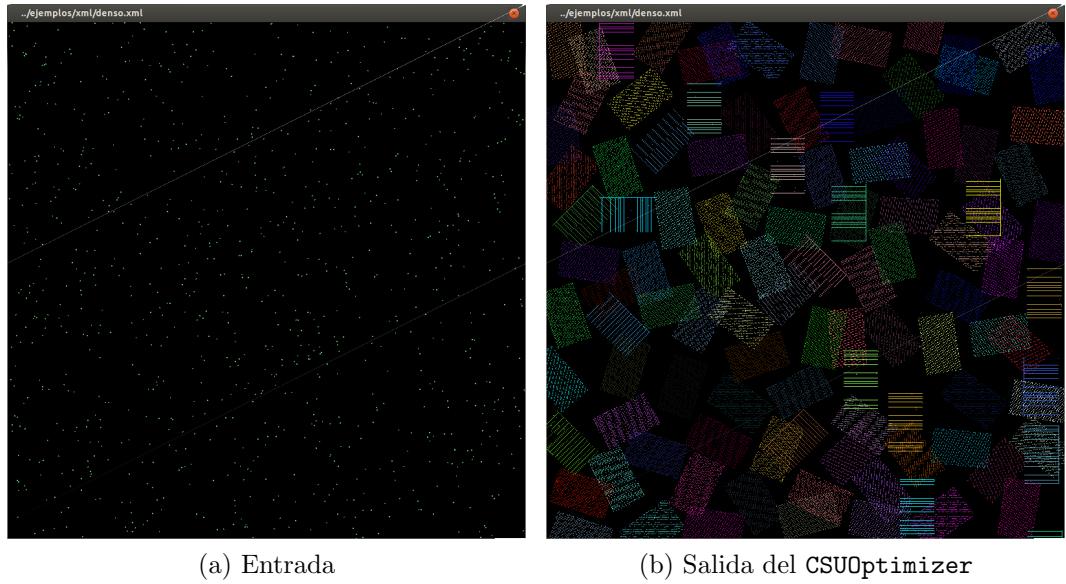


Figura 6.4: Ejemplo `denso.xml`. Gran número de objetos distribuidos homogéneamente

RESULTADOS	Apuntados	Tiempo
Sin opciones	110	26.93"
Sin bordes	118	20.02"
Beam Switching	150	42.87"
Sin bordes + Beam S.	175	1' 1.78"

Tabla 6.4: Resultados del ejemplo `denso.xml`

RESULTADOS	Apuntados	Tiempo
Sin opciones	135	1' 42.69"
Sin bordes	152	2' 17.52"
Beam Switching	217	3' 44.53"
Sin bordes + Beam S.	296	3' 20.97"

Tabla 6.5: Resultados del ejemplo `denso2000.xml`

Si se comparan los resultados del ejemplo anterior (Tabla 6.4) con los de este caso, se comprueba que los tiempos no aumentan linealmente conforme aumenta el tamaño del problema: se han duplicado los objetos pero los tiempos no presentan esta proporcionalidad. Este efecto se debe a que el tiempo final no depende tanto del número de puntos sino de cómo estos están

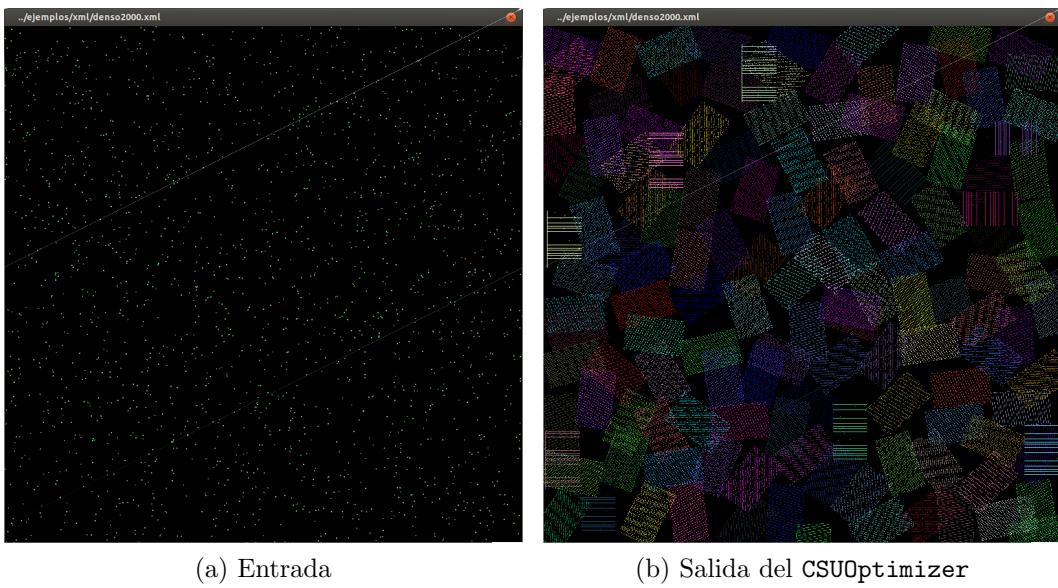


Figura 6.5: Ejemplo `denso2000.xml`. 2000 objetos generados aleatoriamente y distribuidos de forma homogénea

distribuidos espacialmente. A mayor número de colisiones, mayor número de comprobaciones tendrá que hacer la aplicación.

### 6.1.6. Ejemplo: `disperso.xml`

Este ejemplo consta de 200 objetos repartidos aleatoriamente, de forma que no se formen nubes de gran densidad, obteniendo así un caso en el que los objetos tienen una cierta distancia entre ellos. Es este caso el número de apuntados debería ser cercano al número de objetos, puesto que con esta disposición es poco probable encontrar grupos cercanos agrupables en una única CSU.

RESULTADOS	Apuntados	Tiempo
Sin opciones	55	2.20"
Sin bordes	55	2.16"
Beam Switching	61	2.66"
Sin bordes + Beam S.	67	2.85"

Tabla 6.6: Resultados del ejemplo `disperso.xml`

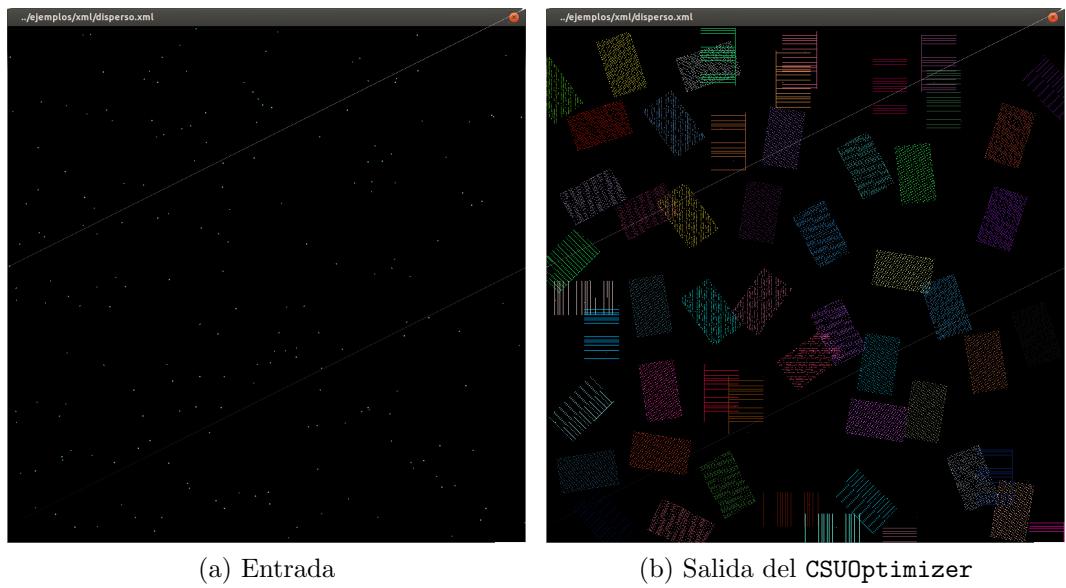


Figura 6.6: Ejemplo `disperso.xml`. 200 objetos distribuidos en un campo de un grado cuadrado

El número de apuntados obtenido presenta una media de unos 4 objetos por apuntado, significativamente inferior a su capacidad de 55 objetos, como cabe esperar de la baja densidad del ejemplo.

### 6.1.7. Ejemplo: `clusters.xml`

Para este ejemplo se han generado tres nubes de objetos separadas entre si, cuyos objetos están dispuestos aleatoriamente, impidiendo conocer a priori el resultado óptimo. El número total de objetos es de 146. El objetivo de esta prueba era comprobar que la función que reduce las colisiones no trataba apuntados o elementos que estuvieran separados.

## 6.2. Ejemplos reales

En esta sección se presentan ejemplos que no han sido generados de manera sintética como los anteriores, sino que han sido aportados por el

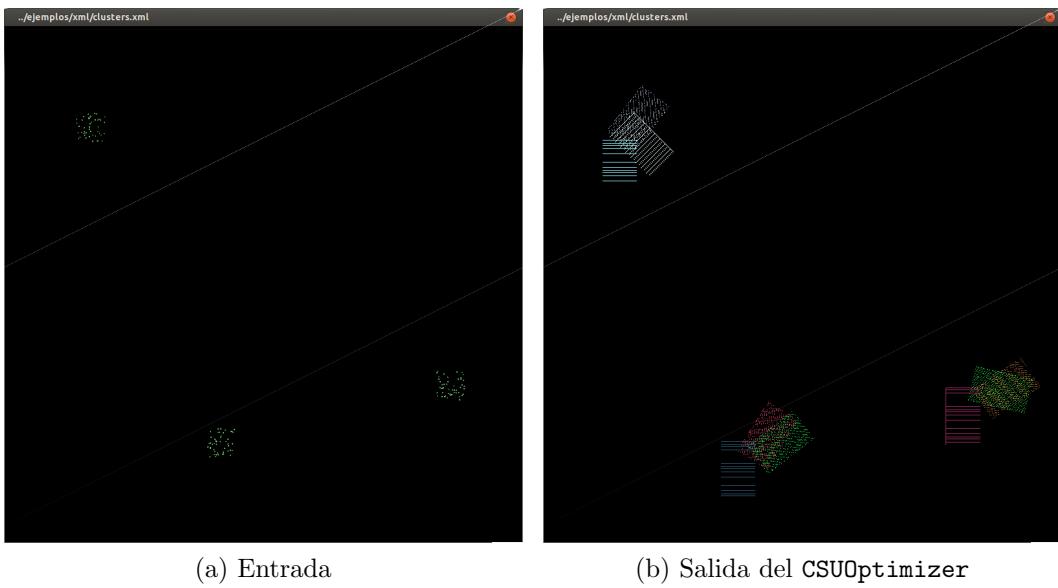


Figura 6.7: Ejemplo `clusters.xml`. Pequeño grupo de objetos agrupados en tres clústers independientes

RESULTADOS	Apuntados	Tiempo
Sin opciones	9	2.79"
Sin bordes	9	2.69"
Beam Switching	12	5.18"
Sin bordes + Beam S.	15	5.43"

Tabla 6.7: Resultados del ejemplo `clusters.xml`

responsable del proyecto EMIR para comprobar el funcionamiento de la aplicación ante distribuciones de objetos reales (o al menos más realistas).

### 6.2.1. Ejemplo: `real1.xml`

Este ejemplo ha sido clave para comprobar la calidad de los resultados obtenidos, puesto que se conoce a priori el número óptimo de apuntados necesarios para resolverlo, siendo este valor de 65 apuntados. El ejemplo consta de un conjunto de aproximadamente 2000 objetos dispersos en un campo de un grado cuadrado, esto es,  $3600 \times 3600$  arcosegundos.

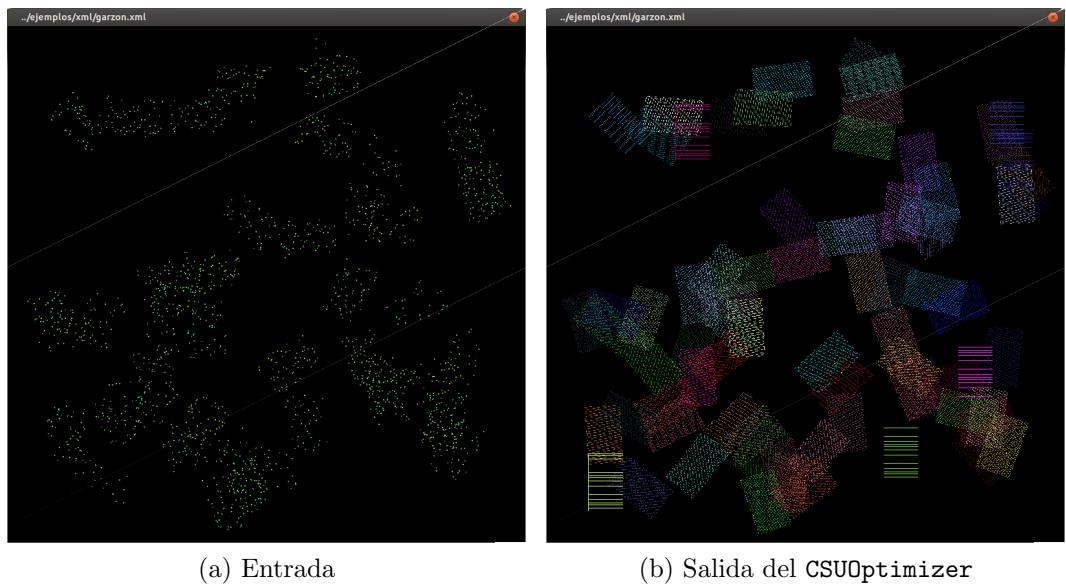


Figura 6.8: Ejemplo `real1.xml`. Caso real de 2000 objetos que siguen una distribución uniforme

RESULTADOS	Apuntados	Tiempo
Sin opciones	75	2' 53.69"
Sin bordes	91	2' 3.21"
Beam Switching	142	4' 1.28"
Sin bordes + Beam S.	181	5' 33.68"

Tabla 6.8: Resultados del ejemplo `real1.xml`

El resultado obtenido se acerca al óptimo, dando 10 apuntados más de los deseados. Puede verse como una extensión del problema `4_2solapes` analizado anteriormente. Al existir mucho solape entre apuntados, el algoritmo tarda al intentar reducirlos, aunque no siempre toma la mejor elección posible de objetos, situando en un apuntado un elemento que iría mejor en otro. No obstante el resultado es razonablemente bueno, confirmando nuestra hipótesis sobre el buen funcionamiento del algoritmo.

### 6.2.2. Ejemplo: real2.xml

El ejemplo que sigue a continuación tiene una cantidad de objetos mucho menor a la que podríamos esperar de un caso real, puesto que consta solamente de 297 objetos. Se desconoce su resultado óptimo.

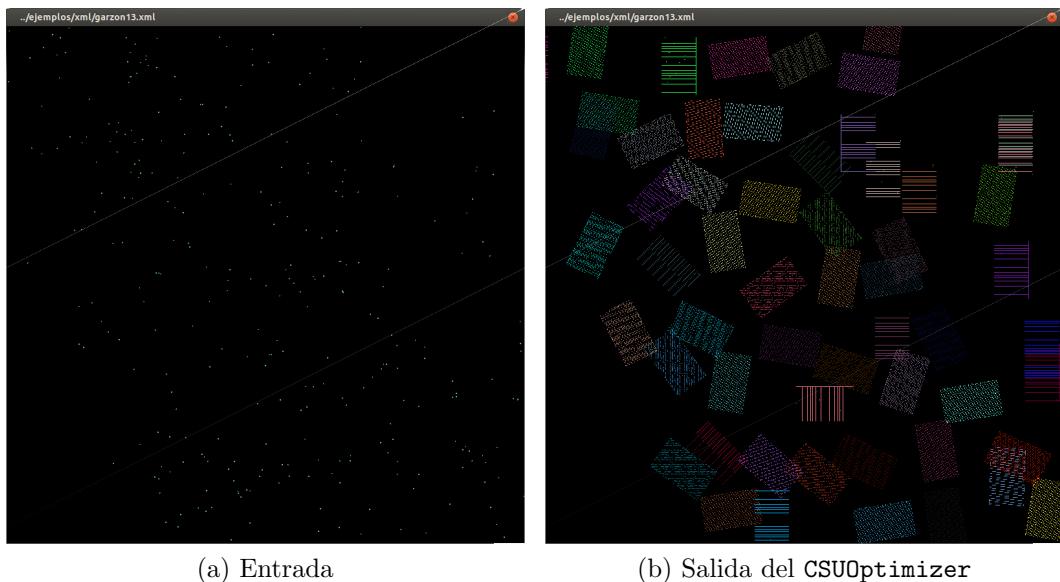


Figura 6.9: Ejemplo **real2.xml**. Aproximadamente 300 objetos reales

RESULTADOS	Apuntados	Tiempo
Sin opciones	57	3.85"
Sin bordes	56	3.91"
Beam Switching	70	6.24"
Sin bordes + Beam S.	79	7.08"

Tabla 6.9: Resultados del ejemplo **real2.xml**

En el segundo caso, en el cual se ejecuta la opción de quitar un mínimo espacio a los bordes, obtenemos un mejor resultado. Esto se debe, como podemos intuir, a que al obligar que los objetos se sitúen en un espacio más reducido, es posible que la forma en la que se coloquen evite que un elemento que antiguamente nos producía un mal intercambio de puntos entre a formar parte del apuntado.

### 6.2.3. Ejemplo: cluster1.xml

El siguiente ejemplo es un caso real con 1398 objetos situados formando un círculo alrededor del centro de un espacio de 4187 arcosegundos de ancho por 4183 arcosegundos de alto.

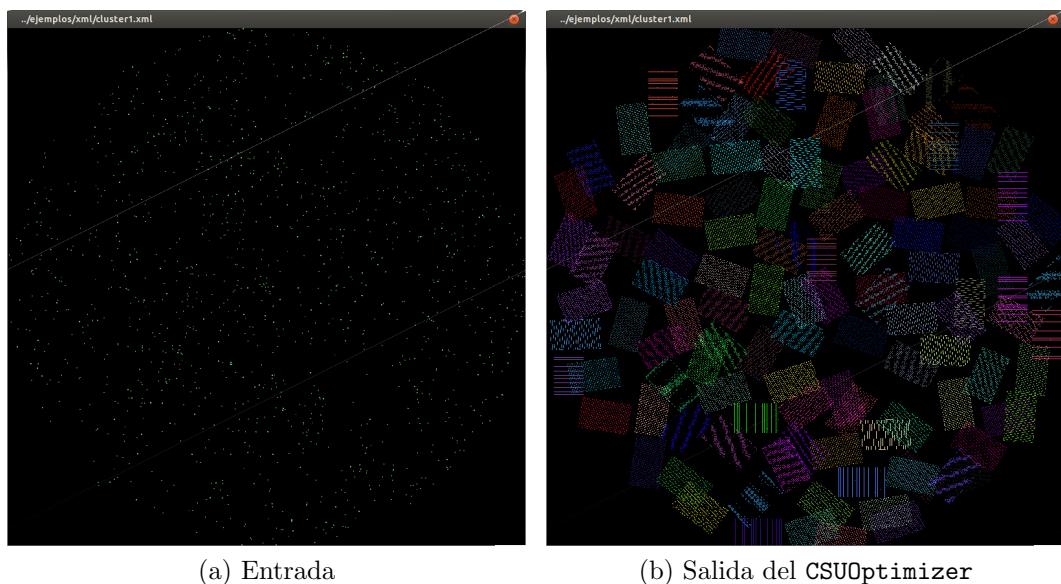


Figura 6.10: Ejemplo `cluster1.xml`. Caso real cuyos objetos muestran una distribución concéntrica.

RESULTADOS	Apuntados	Tiempo
Sin opciones	123	45.04"
Sin bordes	138	45.05"
Beam Switching	180	1' 29.35"
Sin bordes + Beam S.	218	1' 59.17"

Tabla 6.10: Resultados del ejemplo `cluster1.xml`

### 6.2.4. Ejemplo: cluster2.xml

Este caso es similar al anterior, consta de 1842 elementos, aproximadamente 400 objetos más que antes, por lo que podríamos suponer un aumento considerable en los tiempos de ejecución. Sin embargo, como ya

hemos comentado anteriormente, el tiempo depende más de la disposición de los elementos que de su número, aunque, debido a la semejanza entre los ejemplos, asumimos que son condiciones iguales con un mayor número de puntos y esperamos tiempos mayores.

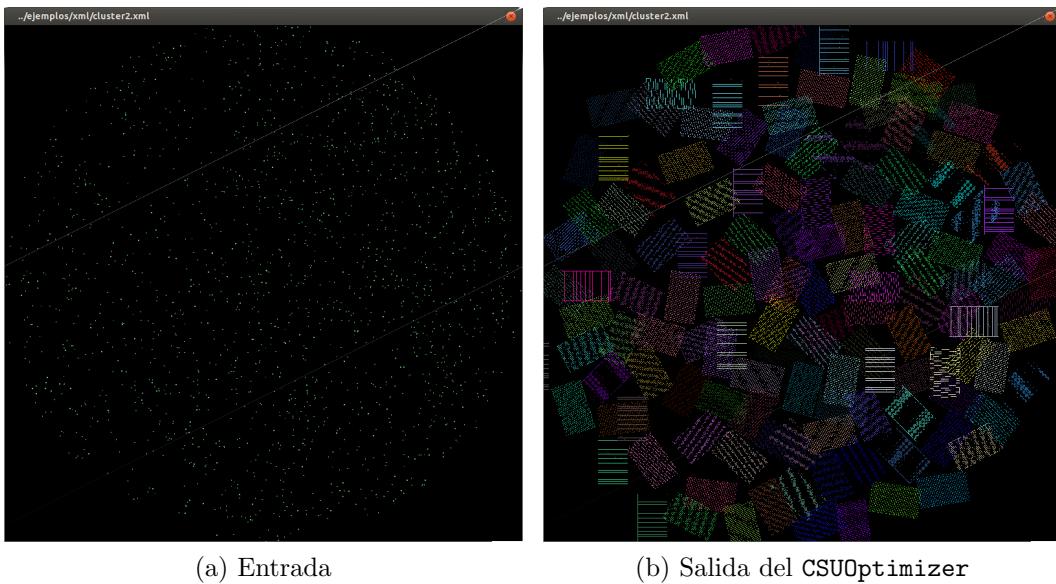


Figura 6.11: Ejemplo `cluster2.xml`. Segundo ejemplo cuyos objetos muestran una distribución concéntrica.

RESULTADOS	Apuntados	Tiempo
Sin opciones	138	1' 57.42"
Sin bordes	152	2' 1.68"
Beam Switching	213	2' 19.05"
Sin bordes + Beam S.	263	4' 19.05"

Tabla 6.11: Resultados del ejemplo `cluster2.xml`

Como se observa en los resultados obtenidos, a pesar de que el número de apuntados necesarios para recoger todos los objetos no es mucho mayor (una diferencia de 15 apuntados), sí lo es el tiempo que tarda, pasando de aproximadamente dos minutos a algo más de cuatro minutos y medio. No obstante, si nos fijamos en el tercer caso, en el que se usa la técnica del beam switching, vemos que con una diferencia de 33 apuntados, apenas aumenta 1 minuto el tiempo.

### 6.2.5. Ejemplo: cluster3.xml

Este ejemplo es igual que los dos anteriores, es decir, tiene una disposición circular y consta de 1074 objetos. Igual que en el anterior esperábamos tiempos mayores debido a que era el ejemplo de mayor tamaño, en este esperamos mejores tiempos al ser el que presenta menor número de elementos.

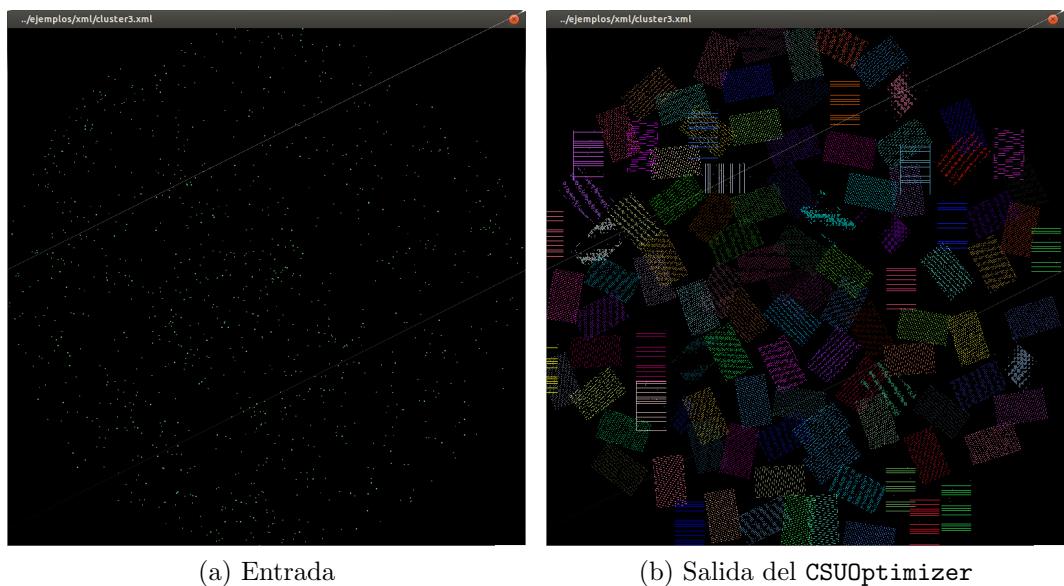


Figura 6.12: Ejemplo `cluster3.xml`. Tercer caso real cuyos objetos muestran una distribución concéntrica.

RESULTADOS	Apuntados	Tiempo
Sin opciones	109	31.09"
Sin bordes	117	38.57"
Beam Switching	153	1' 4.49"
Sin bordes + Beam S.	180	1' 16.68"

Tabla 6.12: Resultados del ejemplo `cluster3.xml`

Los resultados obtenidos son similares a los del primer ejemplo, y no distan tanto entre sí como los dos primeros a pesar de tener más o menos la misma diferencia de objetos.

# Capítulo 7

## Conclusiones y Trabajos Futuros

En este PFC, se ha abordado la resolución del problema de minimización del número de apuntados del telescopio GTC necesarios para observar mediante el instrumento EMIR un conjunto de objetos de interés astronómico. Se trata de un problema real que tienen los investigadores del proyecto EMIR y ello ha supuesto para el autor un acicate en nuestro esfuerzo para acreditar que somos capaces de resolver eficientemente problemas de ingeniería.

El resultado más tangible del PFC es la aplicación `CSUOptimizer`. Se trata de una pequeña aplicación escrita en C++ siguiendo el paradigma orientado a objetos de unas 3300 líneas de código. No obstante quisieramos destacar desde el punto de vista académico que el PFC, como no puede ser de otro modo, ha representado un escenario real de trabajo en el que el autor ha podido desplegar de forma práctica técnicas, metodologías, herramientas y conceptos estudiados en los años de discencia en la ETSI Informática.

Las mayores dificultades halladas en el desarrollo del proyecto han sido:

- El estudio y selección de algoritmos que resuelvan el problema planteado.
- La comprensión precisa de las características intrínsecas al problema.
- La adaptación del lenguaje de la astronomía al campo que nos es propio

de la computación.

- La mejora de la calidad de las soluciones obtenidas por la aplicación.
- La reducción de los tiempos de ejecución del algoritmo que finalmente se ha implementado.

Queremos destacar la importancia del trabajo de depuración y optimización del código que compone el `CSUOptimizer` para obtener una versión que resuelve satisfactoriamente el problema propuesto. Asimismo es destacable el trabajo realizado por el autor para lograr elaborar un producto listo para usar por parte de usuarios ajenos a la informática, poniendo a su disposición esta memoria del PFC, que constituye al mismo tiempo la mejor documentación disponible para la propia aplicación desarrollada. Importante ha sido en el desarrollo el trabajo con las herramientas de versionado y gestión de repositorio, sin las cuales las tareas de depuración y publicación de *release* hubieran sido mucho más tediosas.

En este documento se han revisado los logros conseguidos en la ejecución del proyecto. Relacionándolos directamente con los objetivos planteados inicialmente, podemos señalar como más significativos los siguientes logros:

- Se ha realizado una investigación sobre posibles métodos para resolver el problema propuesto.
- Se ha desarrollado una aplicación plenamente operativa que resuelve el problema de optimización de los apuntados de forma que consideramos satisfactoria.
- El software que se entrega como resultado ha sido desarrollado teniendo en mente que posiblemente sea tomado como punto de partida para desarrollos ulteriores que, utilizando esta implementación como base, la doten de funcionalidades adicionales, particularmente en el área de interfaz de usuario. Este punto de vista ha obligado al autor a un trabajo concienzudo de documentación y estructuración razonada del código.
- Se aporta el presente documento que constituye la documentación técnica de la aplicación desarrollada (Capítulo 5). Este documento será de gran ayuda para cualquier ingeniero que tenga que hacerse cargo posterior de este desarrollo tanto a efectos de mejora, depuración u optimización.

Consideramos por otra parte que el problema propuesto por el equipo investigador del proyecto EMIR no queda cerrado con la conclusión de este PFC. Debido a las limitaciones temporales que afectan a un PFC en Informática hay diversos aspectos del trabajo realizado que claramente cabe mejorar. Algunas mejoras que hubieran tenido cabida en el marco del PFC no entrañan gran dificultad. Si no han sido desarrolladas, ello se debe a que hasta el último momento, los esfuerzos del trabajo se han centrado exclusivamente en la mejora de las soluciones encontradas por la aplicación. Reseñamos a continuación en forma de posibles trabajos futuros a acometer en el marco del Proyecto EMIR algunas de estas mejoras:

- Un estudio más concienzudo de la calidad de las soluciones que se obtienen con el **CSUOptimizer** respecto a las soluciones óptimas ha quedado fuera del trabajo realizado.
- Hay margen de mejora en cuanto a los tiempos de respuesta del **CSUOptimizer**. Aunque se encuentran dentro de los márgenes inicialmente establecidos en la especificación de requisitos, pensamos que no es un aspecto difícil de mejorar. Un análisis preliminar mediante profiling de la aplicación nos hace ser conocedores de las funciones que habría que optimizar y una tecnología de paralelización utilizando OpenMP se nos ocurre como la solución a implementar inicialmente.
- La interfaz de usuario es otro aspecto que claramente merece atención en cuanto a necesidades de desarrollo. A pesar de que los usuarios del **CSUOptimizer** pertenecen al ámbito científico-tecnológico, en el que hay una mayor permisividad respecto a las limitaciones en cuanto a interfaz de usuario de una aplicación, pensamos que estos usuarios se beneficiarían de una pasarela que permitiera una selección cómoda de los objetos a observar. En este sentido, hemos optado por elegir XML como formato de entrada de datos a la aplicación, lo cual flexibiliza la conexión con una futura GUI.



# Apéndice A

## Códigos

### A.1. colisiones

```
set<CSU> colisiones (set<CSU> lista_ini, const int &total_puntos) {
    static int finalizar;
    finalizar = 0;
    static set<Element> puntos, p_parcial;
    static set<Element>::iterator p;
    sort(lista_ini, TAMPUN);
    static set<CSU>::iterator jt, it;
    static set<CSU> porcion, no_coli;
    set<CSU> lista_fin;
    no_coli.clear();
    while (finalizar != (lista_ini.size()+no_coli.size())){
        lista_fin.clear();
        finalizar = lista_ini.size();
        set<Element> eliminados;
        eliminados.clear();
        it = lista_ini.begin();
        while (it != lista_ini.end()){
            static bool quitable;
            quitable = true;
            jt = lista_ini.begin();
            while (jt != lista_ini.end() && quitable){
                if (it != jt && (it->colision(*jt) || jt->colision(*it))) //Si
                    colisiona, no lo podemos quitar
                    quitable = false;
                jt++;
            }
            if (quitable) {
```

```

        lista_fin.insert(*it);
        p_parcial = it->getPoints();
        for (p = p_parcial.begin(); p != p_parcial.end(); p++)
            eliminados.insert(*p);
        static set<CSU>::iterator borrar;
        borrar = it; //Guardamos la posicion que se va a eliminar
        it++; //pasamos a mirar la siguiente
        lista_ini.erase(borrar); //
    }
    else
        it++;
}
while (eliminados.size() != total_puntos) { //mientras queden puntos
    por mirar
    puntos.clear();
    p_parcial.clear();
    porcion.clear();
    it = lista_ini.begin(); //it es el principio de la lista de CSUs
    // Cogemos los puntos del primero y los colocamos en la lista de
    mejora
    p_parcial = it->getPoints(); //p_parcial son todos los puntos de la
    primera CSU
    for (p = p_parcial.begin(); p != p_parcial.end(); p++)
        puntos.insert(*p); //insertamos todos los puntos de la primera CSU
        en los totales
    porcion.insert(*it); //Ponemos esa CSU en el grupo que se va a mirar
    CSU ini = *it; //Almacenamos la primera CSU
    lista_ini.erase(it); //Borramos la CSU de la lista total
    jt = lista_ini.begin(); //jt es el principio de la lista de CSUs
    // Seleccionamos todos aquellos que colisionan con el anteriormente
    seleccionado
    while(!lista_ini.empty() && jt != lista_ini.end()) { //comprobamos
        cada uno con el primero de la lista
        if (ini.colision(*jt) || jt->colision(ini)) {
            porcion.insert(*jt); //Ponemos esa CSU en el grupo que se va a
            mirar
            p_parcial = jt->getPoints(); //p_parcial son los puntos de esa
            CSU
            for (p = p_parcial.begin(); p != p_parcial.end(); p++)
                puntos.insert(*p); //Aniadimos los puntos al total
            static set<CSU>::iterator borrar;
            borrar = jt; //Guardamos la posicion que se va a eliminar
            jt++; //pasamos a mirar la siguiente
            lista_ini.erase(borrar); //
        }
        else {
            jt++;
        }
    }
}

```

```
static bool continuar;
continuar = true;
// Si la colision no es necesaria teoricamente, se intenta reducir
if (puntos.size()/NUM_BARRAS < porcion.size()) {
    static bool izq_menor;
    static CSU final;
    static set<CSU>::iterator fin;
    fin = porcion.end();
    fin--;
    final = *fin;
    izq_menor = (ini.gety() < final.gety())? true : false;
    set<CSU> nuevos = sacar_apuntados(puntos);
    sort(nuevos, TAMPUN);
    //Si se ha conseguido mejorar, volvemos a insertarlos para
    //remirarlos
    if ((nuevos.size() < porcion.size()) || (nuevos.size() == porcion.
        size() && nuevos.begin()->size() > porcion.begin()->size())) {
        continuar = false;
        porcion = nuevos;
        for (it = porcion.begin(); it != porcion.end(); it++)
            lista_ini.insert(*it);
    }
}
// Si no se ha conseguido mejorar o la colision en necesaria,
// quitamos la primera CSU
if (continuar) {
    it = porcion.begin();
    lista_fin.insert(*it);
    p_parcial = it->getPoints();
    for (p = p_parcial.begin(); p != p_parcial.end(); p++) {
        eliminados.insert(*p);
    }
    porcion.erase(it);
    for (it = porcion.begin(); it != porcion.end(); it++)
        lista_ini.insert(*it);
}
lista_ini = lista_fin;
}
return lista_fin;
}
```

Listado A.1: Código de la función que reduce las colisiones aleatorias

## A.2. grasp

```

set<CSU> grasp (set<CSU> resultado, set<Element> puntos) {
    int num_sol = 5;
    int tam_entrada = resultado.size();
    sort(resultado, TAMPUN);
    set<CSU> posibles, lista[num_sol], final;
    static set<CSU>::iterator it, jt, borrar;
    set<Element> p_finales;
    //PASO 1: generar N soluciones
    for (int i = 0; i < num_sol; i++)
        lista[i] = apuntadosRandom(puntos, i);
    //PASO 2: buscar mejor solucion
    for (int i = 0; i < num_sol; i++)
        if (lista[i].size() < resultado.size())
            resultado = lista[i];
    set<CSU> resultado_previo = resultado;
    //inicializar la solucion final
    it = resultado.begin();
    while (it != resultado.end()) {
        progreso();
        jt = it;
        jt++;
        static bool col;
        col = false;
        while (jt != resultado.end() && !col) {
            progreso();
            if (it->colision(*jt) || jt->colision(*it))
                col = true;
            jt++;
        }
        if (!col) {
            final.insert(*it);
            static set<Element> contar;
            contar = it->getPoints();
            for (set<Element>::iterator cc = contar.begin(); cc != contar.end(); cc++)
                p_finales.insert(*cc);
            borrar = it;
            it++;
            resultado.erase(borrar);
        }
        else
            it++;
    }
    //PASO 3: Aniadir los restantes de S1 a la RCL
    for (it = resultado.begin(); it != resultado.end(); it++)
        posibles.insert(*it);
    for (int i = 0; i < num_sol; i++)
        for(it = lista[i].begin(); it != lista[i].end(); it++)

```

```

    posibles.insert(*it);
//PASO 4: Seleccionar elementos de la RCL hasta que se cubran todos los
    puntos
while (p_finales.size() != puntos.size()) {
    progreso();
    static int tam_point_max, limite;
    it = jt = posibles.begin();
    tam_point_max = it->size();
    limite = 0;
    it++;
    while (it->size() == tam_point_max) {
        it++;
        limite++;
    }
    if (limite != 0)
        limite = rand() % limite;
    for (int i = 0; i < limite; i++)
        jt++;
    final.insert(*jt);
    static set<Element> contar;
    contar = it->getPoints();
    for (set<Element>::iterator cc = contar.begin(); cc != contar.end();
         cc++)
        p_finales.insert(*cc);
    posibles.erase(jt);
}
//PASO 5: Intentar mejorar el resultado obtenido.
posibles.clear();
posibles = final;
posibles = colisiones(posibles, puntos.size());
if (posibles.size() < resultado_previo.size()) {
    cout << "Mejorado con " << posibles.size() << endl;
    return posibles;
}
else {
    cout << "GRASP inutil" << endl;
    return resultado_previo;
}
}

```

Listado A.2: Método heurístico GRASP

### A.3. Salida exacto.xml

1	<Apuntado>
2	<X>479.1</X>

```
3 <Y>598.787</Y>
4 <PA>0</PA>
5 <Configuracion>
6   <Barra>93.9</Barra>
7   <Barra>115.9</Barra>
8   <Barra>-21.1</Barra>
9   <Barra>-70.1</Barra>
10  <Barra>119.9</Barra>
11  <Barra>-76.1</Barra>
12  <Barra>94.9</Barra>
13  <Barra>55.9</Barra>
14  <Barra>-52.1</Barra>
15  <Barra>-17.1</Barra>
16  <Barra>65.9</Barra>
17  <Barra>72.9</Barra>
18  <Barra>-25.1</Barra>
19  <Barra>-71.1</Barra>
20  <Barra>89.9</Barra>
21  <Barra>83.9</Barra>
22  <Barra>98.9</Barra>
23  <Barra>-23.1</Barra>
24  <Barra>112.9</Barra>
25  <Barra>-27.1</Barra>
26  <Barra>7.9</Barra>
27  <Barra>35.9</Barra>
28  <Barra>-67.1</Barra>
29  <Barra>29.9</Barra>
30  <Barra>46.9</Barra>
31  <Barra>109.9</Barra>
32  <Barra>-42.1</Barra>
33  <Barra>30.9</Barra>
34  <Barra>-25.1</Barra>
35  <Barra>-30.1</Barra>
36  <Barra>-37.1</Barra>
37  <Barra>-58.1</Barra>
38  <Barra>-3.1</Barra>
39  <Barra>6.9</Barra>
40  <Barra>24.9</Barra>
41  <Barra>8.9</Barra>
42  <Barra>-59.1</Barra>
43  <Barra>68.9</Barra>
44  <Barra>75.9</Barra>
45  <Barra>51.9</Barra>
46  <Barra>-60.1</Barra>
47  <Barra>11.9</Barra>
48  <Barra>21.9</Barra>
49  <Barra>-28.1</Barra>
50  <Barra>102.9</Barra>
51  <Barra>35.9</Barra>
```

```
52 <Barra>107.9</Barra>
53 <Barra>52.9</Barra>
54 <Barra>-36.1</Barra>
55 <Barra>-20.1</Barra>
56 <Barra>112.9</Barra>
57 <Barra>-9.1</Barra>
58 <Barra>117.9</Barra>
59 <Barra>86.9</Barra>
60 <Barra>-70.1</Barra>
61 </Configuracion>
62 </Apuntado>
```



# Bibliografía

- [1] Instituto de Astrofísica de Canarias: **Página web del proyecto EMIR** 2013, [<http://guaix.fis.ucm.es/emir/>]. 15
- [2] Instituto de Astrofísica de Canarias: **El Proyecto GOYA** 2013, [<http://www.iac.es/proyecto/GOYAiac/GOYAiac.html>]. 15
- [3] **Instituto de Astrofísica de Canarias** [<http://www.iac.es/>]. 15
- [4] Instituto de Astrofísica de Canarias: **Gran Telescopio CANARIAS** 2013, [<http://www.gtc.iac.es/>]. 15
- [5] Wikipedia: **Analizador sintáctico — Wikipedia, La enciclopedia libre** 2013, [[http://es.wikipedia.org/w/index.php?title=Analizador\\_sint%C3%A1ctico&oldid=64519288](http://es.wikipedia.org/w/index.php?title=Analizador_sint%C3%A1ctico&oldid=64519288)]. 29
- [6] **Xerces-C++ XML Parser** [<http://xerces.apache.org/xerces-c/>]. 30
- [7] Sweet, Michael: **Mini-XML: Lightweight XML Library** [<http://www.msweet.org/projects.php?Z3>]. 31
- [8] **Página web oficial de TinyXML** [<http://www.grinninglizard.com/tinyxml/index.html>]. 31
- [9] **Web del proyecto pugixml** [<http://pugixml.org/>]. 31
- [10] **Página web del proyecto RapidXML** [<http://rapidxml.sourceforge.net/>]. 32
- [11] Cumming M: **libxml++ - An XML Parser for C++** 2012, [<http://libxmlplusplus.sourceforge.net/>]. 32
- [12] Hargreaves S: **Allegro 5** 2010, [<http://alleg.sourceforge.net/>]. 33

- [13] Hartigan, J A and Wong, M A: **A K-Means Clustering Algorithm.** *Applied Statistics* 1979, **28**:100–108. 35
- [14] Ester M, Kriegel HP, Sander J, Xu X: **A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise.** In *Second International Conference on Knowledge Discovery and Data Mining*. Edited by Simoudis E, Han J, Fayyad UM, AAAI Press 1996:226–231. 36
- [15] Wikipedia: **DBSCAN — Wikipedia, The Free Encyclopedia** 2013, [<http://en.wikipedia.org/w/index.php?title=DBSCAN&oldid=553183512>]. 36
- [16] Mladenović, N and Hansen, P: **Variable neighborhood search.** *Comput. Oper. Res.* 1997, **24**(11):1097–1100, [[http://dx.doi.org/10.1016/S0305-0548\(97\)00031-2](http://dx.doi.org/10.1016/S0305-0548(97)00031-2)]. 40
- [17] Kanwar Bhajneek: **DBScan-Data clustering algorithm in Java (with Gui)** 2012, [<https://github.com/KanwarBhajneek/DBSCAN>]. 43
- [18] Atlassian Software: **Bitbucket** 2013, [<https://bitbucket.org/>]. 54
- [19] Matt Mackall: **Mercurial source control management** 2013, [<http://mercurial.selenic.com/>]. 54
- [20] Dimitri van Heesch: **Doxxygen** 2013, [<http://www.stack.nl/~dimitri/doxygen/index.html>]. 55
- [21] Leslie Lamport: **LaTeX** 2013, [<http://www.latex-project.org/>]. 56

# Índice de figuras

1.1.	Unidad de rendijas configurables de EMIR . . . . .	23
1.2.	Representación gráfica de una CSU . . . . .	24
1.3.	Apuntado con 4 barras ocupadas . . . . .	25
1.4.	Muestra de funcionamiento de <code>beam switching</code> . . . . .	25
3.1.	Ejemplos de la potencia gráfica de la librería Allegro . . . . .	34
3.2.	Ejemplo de cómo funciona el K-means . . . . .	36
3.3.	Ejemplo del resultado del DBSCAN. Los puntos en gris son ruido . . . . .	38
4.1.	Resultado de aplicar DBSCAN al <code>real1.xml</code> . . . . .	45
4.2.	Movimientos de una CSU en el proceso de mejora . . . . .	49
5.1.	Vértices de la CSU . . . . .	62
5.2.	Disponibilidad de las barras según las opciones de entrada. a) Barra normal. b) Barra con <code>--noborder</code> . c) Barra con <code>--beams</code> . d) Barra con ambas opciones . . . . .	78

6.1. Ejemplo <code>exacto.xml</code> . 55 objetos que se ajustan perfectamente a una única CSU . . . . .	83
6.2. Ejemplo <code>4_0solapes.xml</code> . 4 apuntados distanciados entre si . . . . .	84
6.3. Ejemplo <code>4_2solapes.xml</code> . Ejemplo de “colisión” de dos apuntados . . . . .	85
6.4. Ejemplo <code>denso.xml</code> . Gran número de objetos distribuidos homogéneamente . . . . .	86
6.5. Ejemplo <code>denso2000.xml</code> . 2000 objetos generados aleatoriamente y distribuidos de forma homogénea . . . . .	87
6.6. Ejemplo <code>disperso.xml</code> . 200 objetos distribuidos en un campo de un grado cuadrado . . . . .	88
6.7. Ejemplo <code>clusters.xml</code> . Pequeño grupo de objetos agrupados en tres clústers independientes . . . . .	89
6.8. Ejemplo <code>real1.xml</code> . Caso real de 2000 objetos que siguen una distribución uniforme . . . . .	90
6.9. Ejemplo <code>real2.xml</code> . Aproximadamente 300 objetos reales . . . . .	91
6.10. Ejemplo <code>cluster1.xml</code> . Caso real cuyos objetos muestran una distribución concéntrica. . . . .	92
6.11. Ejemplo <code>cluster2.xml</code> . Segundo ejemplo cuyos objetos muestran una distribución concéntrica. . . . .	93
6.12. Ejemplo <code>cluster3.xml</code> . Tercer caso real cuyos objetos muestran una distribución concéntrica. . . . .	94

# Índice de tablas

1.1. Especificaciones de alto nivel en EMIR . . . . .	21
5.1. Posibles valores del método <code>p_potencial</code> . . . . .	65
6.1. Resultados del ejemplo <code>exacto.xml</code> . . . . .	82
6.2. Resultados del ejemplo <code>4_0solapes.xml</code> . . . . .	83
6.3. Resultados del ejemplo <code>4_2solapes.xml</code> . . . . .	84
6.4. Resultados del ejemplo <code>denso.xml</code> . . . . .	86
6.5. Resultados del ejemplo <code>denso2000.xml</code> . . . . .	86
6.6. Resultados del ejemplo <code>disperso.xml</code> . . . . .	87
6.7. Resultados del ejemplo <code>clusters.xml</code> . . . . .	89
6.8. Resultados del ejemplo <code>real1.xml</code> . . . . .	90
6.9. Resultados del ejemplo <code>real2.xml</code> . . . . .	91
6.10. Resultados del ejemplo <code>cluster1.xml</code> . . . . .	92
6.11. Resultados del ejemplo <code>cluster2.xml</code> . . . . .	93
6.12. Resultados del ejemplo <code>cluster3.xml</code> . . . . .	94



# Índice de listados

5.1.	Ejemplo básico de uso de la librería Allegro . . . . .	57
5.2.	Ejemplo de dibujo con Allegro . . . . .	57
5.3.	Constantes de tamaño de la CSU . . . . .	60
5.4.	Código del ray casting simplificado . . . . .	63
5.5.	Función que comprueba si un objeto está en la CSU . . . . .	64
5.6.	Función que comprueba si un objeto podría estar en la CSU . .	66
5.7.	Método que añade un punto que requiere de ciertos ajustes . .	67
5.8.	Código del movimiento de mejora vertical hacia arriba . . . .	69
5.9.	Código del movimiento de mejora horizontal izquierdo . . . .	70
5.10.	Código del método que comprueba las colisiones . . . . .	71
5.11.	Estructura de los ficheros de entrada para <code>CSUOptimizer</code> . . .	72
5.12.	Ejemplo de entrada para <code>CSUOptimizer</code> , extraído de <code>exacto.xml</code>	73
5.13.	Primera fase del algoritmo constructivo: generar la solución inicial . . . . .	75
5.14.	Código de la función que genera soluciones aleatorias . . . .	76
5.15.	Estructura del modo de resultado <code>.dat</code> . . . . .	78

5.16. Estructura de entrada de los ficheros .dat . . . . .	80
A.1. Código de la función que reduce las colisiones aleatorias . . . . .	99
A.2. Método heurístico GRASP . . . . .	102