UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Análisis y Procesamiento de Planos de Arquitectura para Estudios Estructurales

Informe de Proyecto de Grado presentado por

Eduardo Nogueira & Matías Manrique

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Ignacio Ramirez
Federico Rivero

Montevideo, 28 de octubre de 2025

Agradecimientos

A nuestros familiares y amigos por su incondicional apoyo, soporte e incentivo para llevar este trabajo a su fin. Sin su comprensión y palabras de motivación no hubiésemos podido superar los obstáculos que enfrentamos durante estos años para darle cierre a este proyecto. Gracias.

Resumen

Este informe presenta el desarrollo de un proyecto de grado de la carrera de Ingeniería en Computación, cuyo objetivo principal es automatizar la identificación de elementos estructurales en planos arquitectónicos. El proyecto surge a petición de la empresa Ingenium, con la intención de reducir el tiempo que sus ingenieros dedican a la "limpieza" manual de planos recibidos de arquitectos.

A partir del estudio del estado del arte, se encontró que este problema se ha investigado en mayor profundidad para imágenes rasterizadas que para imágenes vectoriales o archivos CAD.

Los principales desafíos que enfrentó el proyecto fueron la falta de conjuntos de datos clasificados y la complejidad inherente a la identificación de objetos en imágenes vectoriales.

Como parte del proyecto se desarrolló un programa prototipo que procesa los planos de forma automática para detectar paredes y puertas (cuyo código se puede obtener siguiendo el instructivo del apéndice B.1), el cual se evaluó procesando planos de distintas características, y tomando varias métricas a partir de los resultados. Los mismos planos se procesaron utilizando distintas configuraciones para determinar los parámetros que permitieron obtener mejores resultados.

Los resultados obtenidos demuestran que es viable utilizar reglas para identificar elementos estructurales en planos. El prototipo desarrollado logra buenos resultados para ser una primera aproximación, sin embargo tienen margen de mejora en términos de precisión y usabilidad. Se destaca particularmente la arquitectura flexible desarrollada, que permite extender y configurar el sistema para futuros refinamientos y mejoras.

Se sugieren líneas de trabajo futuras, como la generación de conjuntos de datos de planos clasificados, la exploración de técnicas de aprendizaje automático para imágenes vectoriales, y la combinación de técnicas de visión por computadora en imágenes raster con reglas rígidas para el procesamiento vectorial.

Palabras clave: AutoCAD, Planos de Arquitectura, Automatización, Clasificación, DXF

Índice general

Capítulo 1

Introducción

La forma usual de organizar un proyecto de construcción edilicia consta de tres etapas: identificación de la necesidad, diseño y construcción (Richard H. Clough, 2008). Durante la etapa de diseño, arquitectos e ingenieros civiles colaboran en la generación de los planos de construcción.

Los ingenieros civiles realizan el análisis estructural de los planos para determinar si la estructura diseñada soportará la carga esperada desde un punto de vista teórico (Hibbeler, 2012). Para esto, se inicia con la limpieza del plano, descartando los elementos que no son de interés, como el mobiliario o la vegetación.

La empresa Ingenium se dedica (entre otras cosas) a realizar análisis estructurales para proyectos de construcción. Sus ingenieros realizan de forma manual la limpieza de los planos que reciben de los arquitectos, lo que puede ser un proceso lento (según mencionó el cliente, unos 20 minutos por plano). Más aún, este proceso deben repetirlo en cada iteración del plano hasta llegar al diseño final.

Con la intención de aprovechar mejor el tiempo de los ingenieros, la empresa propone el presente proyecto para explorar la posibilidad de automatizar la identificación de elementos estructurales de un plano utilizando un programa de computadora.

Los planos que reciben los ingenieros son archivos digitales en formato DWG o DXF, utilizados comúnmente en *AutoCAD*. Ambos formatos representan la información gráfica de manera vectorial y preservan atributos geométricos y topológicos que se necesitan conservar para el análisis estructural y para la edición posterior en CAD.

La principal necesidad que justifica este proyecto es la de trabajar directamente sobre representaciones vectoriales (DWG/DXF) y no sobre imágenes ráster (formada por una matriz regular de celdas llamadas píxeles, dispuestas

en filas y columnas). Esto es importante para el análisis estructural, ya que cualquier conversión o alteración del dibujo (por ejemplo, cambios en grosores de las paredes) puede modificar la geometría relevante y afectar los resultados.

Se partió del conocimiento del cliente sobre este tipo de software, quien no había encontrado una solución a su problema. Se realizó una búsqueda en la web de programas que realizaran análisis de planos arquitectónicos para detectar elementos estructurales; sin embargo, no se encontró ninguno de fácil acceso que permitiera procesar archivos *DXF*. A continuación, se mencionan algunos artículos sobre detección de ambientes y figuras en planos rasterizados.

1.1. Antecedentes

La búsqueda de antecedentes se enfocó en artículos y soluciones de software que abordaran problemas similares. Se observó que la gran mayoría de trabajos publicados procesan planos como imágenes rasterizadas; se encontró un artículo que trabajaba sobre datos vectoriales que se describe al final de esta sección.

En cada trabajo mencionado a continuación se aclara, cuando corresponde, que el enfoque opera sobre ráster y, en consecuencia, no es aplicable de forma directa a archivos *DXF*/*DWG*.

En "Improved Automatic Analysis of Architectural Floor Plans" (Ahmed, Liwicki, Weber, y Dengel, 2011) se describe una técnica para el análisis de planos arquitectónicos desde una visión de planta. A grandes rasgos, se proponen técnicas para realizar la detección de elementos y espacios en planos almacenados como imágenes ráster. Se basa en trabajos previos como (Bay, Ess, Tuytelaars, y Van Gool, 2008) para detección de símbolos.

El procesamiento de las imágenes se realiza en 3 etapas, la primera para segmentar la información de la imagen, separando el texto de las figuras, detectando los símbolos que representan las paredes y clasificándolas según su grosor; la segunda busca identificar los bordes de las paredes y el contorno del plano; y en la tercera se utiliza la información acumulada en las etapas previas para detectar los espacios del plano (por ejemplo, usando el texto extraído se intenta determinar dónde está la sala, el baño, etc.). El preprocesamiento que se realiza en la primera etapa parece mejorar el resultado respecto a otros trabajos previos donde se atacó este mismo problema. La técnica estudiada en este artículo no puede ser aplicada a imágenes vectoriales.

"Automatic Room Detection and Room Labeling from Architectural Floor Plans" (Ahmed, Liwicki, Weber, y Dengel, 2012), si bien no se menciona explícitamente, es la continuación del artículo anterior, realizado por los mismos autores. Al igual que en el trabajo anterior, se basa en el uso de técnicas de procesamiento de imágenes como segmentación de líneas y detección de contornos. A estas, se les suma reconocimiento óptico de caracteres (OCR) en la capa de texto para obtener etiquetas semánticas de las habitaciones, como pueden

ser "cocina", "baño", etc. La técnica OCR consiste en la extracción de texto presente en una imagen ráster. Un aspecto novedoso del sistema es su etapa de pos-procesamiento, en la que un espacio físico que no está separado por ninguna pared, se separa semánticamente según las etiquetas del plano (por ejemplo, un espacio grande etiquetado como "living/estudio" puede ser separado en dos habitaciones distintas). A su vez, habitaciones sin etiquetar pueden ser unidas a espacios contiguos si su semántica es similar, permitiendo así una mejor agrupación de los espacios de acuerdo a sus funciones. Al igual que en el trabajo precedente, el enfoque exclusivo en imágenes ráster lo vuelve inaplicable a este proyecto.

El artículo "Statistical segmentation and structural recognition for floor plan interpretation" (Heras, Ahmed, Liwicki, Valveny, y Sánchez, 2013) explica un método que combina la segmentación estadística a nivel de píxeles y el análisis estructural para identificar y categorizar los elementos constructivos principales: muros, puertas y ventanas, y luego extraer la información de las habitaciones. La primera fase es la división de la imagen en pequeñas regiones llamadas parches. Cada parche se describe usando Blurred Shape Model (Escalera, Fornés, Pujol, Lladós, y Radeva, 2007), que calcula la densidad de píxeles de cada celda, teniendo en cuenta también las celdas vecinas. Los descriptores generados se agrupan con el algoritmo k-means para crear un vocabulario visual usando el modelo Bag of Visual Words (Csurka, Dance, Fan, Willamowski, y Bray, 2004).

En este caso cada "palabra visual" representa un patrón gráfico aprendido de los datos y se le relacionan las probabilidades de pertenecer a alguna de las categorías: muro, puerta, ventana o fondo. Los parches se asocian con las palabras visuales más cercanas, y las probabilidades se suman para clasificar los píxeles. El resultado de esta segmentación y agrupación es un grafo que se procesa usando algoritmos como A* (Heart, Nilsson, y Raphael, 1968) para refinar la detección de puertas, ventanas y paredes (mediante la detección de ciclos en el grafo). El método fue evaluado con cuatro conjuntos de planos reales con notaciones diferentes, con un nivel de precisión de hasta el 95 %. La técnica descrita se basa en segmentar una imagen ráster, y no se puede aplicar directamente en imágenes vectoriales.

"Parsing Floor Plan Images" (Dodge, Xu, y Stenger, 2017) explora el uso de visión por computadora para extraer información estructural y semántica de los planos de planta, como la forma y tamaño de las habitaciones, la ubicación de puertas, ventanas, elementos de cocina y baño. El método combina tres enfoques principales:

- Segmentación de paredes: Utilizando un enfoque de aprendizaje automático basado en redes neuronales (más precisamente FCN (Long, Shelhamer, y Darrell, 2015)) para segmentar las paredes en los planos de planta.
- Detección de objetos: Se utiliza un modelo Faster R-CNN (Ren, He, Girshick, y Sun, 2016) para detectar objetos específicos en los planos de planta, como puertas, ventanas, estufas, bañeras, lavabos y re-

tretes. Este tipo de modelos consiste en una arquitectura de CNN (Lecun, Bengio, y Haffner, 1998) que permite identificar objetos y su ubicación dentro de una imagen. Un subconjunto de imágenes se anotó manualmente para entrenar la red neuronal.

- Reconocimiento óptico de caracteres (OCR): Se utiliza OCR para leer los tamaños de las habitaciones en los planos de planta. En algunos casos, no todas las habitaciones tienen etiquetas de tamaño. En estos casos, se propagan los tamaños de las habitaciones conocidas a las habitaciones de tamaño desconocido, utilizando sus tamaños relativos.

Estos métodos permiten convertir la imagen de un plano de planta en un modelo paramétrico completo que se puede utilizar para crear visualizaciones en 3D, recorridos virtuales e incluso remodelar propiedades. Además, la lectura de los tamaños de las habitaciones en la imagen permite una visualización precisa a escala del mobiliario en la propiedad.

Las técnicas utilizadas no pueden aplicarse sobre imágenes vectoriales, por lo que no fueron utilizadas en el proyecto.

El artículo "Automatic analysis and simplification of architectural floor plans" (Alexander Wyss and Florian Bruggisser, 2020) describe una forma de procesar imágenes ráster de planos para detectar habitaciones separadas por paredes. El mismo combina varios algoritmos para limpiar y clasificar elementos del plano, entre los que destacan *Morphological cleaning* para reducir el ruido de la imagen, *Aprendizaje Automático* y *Convex Hull Closing* para detectar habitaciones y definir su contorno y *Connected Component Analysis* para mejorar la detección de habitaciones. En la portada del proyecto se argumenta que la precisión de este es mejor que la de otros de años anteriores.

Lo destacable de este trabajo es que su código fuente está disponible públicamente en GitHub, lo que permite realizar pruebas para evaluar su usabilidad y desempeño (figura 1.1). La herramienta permite configurar parámetros del flujo de trabajo para mejorar la precisión del resultado, pero dichos parámetros están íntimamente relacionados con el funcionamiento interno de la aplicación y de los modelos que utiliza para procesar la imagen, lo que los vuelve poco intuitivos para el usuario final.

Al igual que en casos anteriores, las técnicas utilizadas se enfocan en procesamiento de imágenes ráster, volviéndolas inviables para resolver la necesidad del cliente.

"Leveraging Large Language Models for Scalable Vector Graphics-Driven Image Understanding" (Cai, Huang, Li, Wang, y Lee, 2023) busca responder la pregunta: *¿puede un LLM (Large Language Model), que nunca ha visto información visual, entender y razonar sobre imágenes?*. Los LLMs son modelos capaces de procesar y generar texto para realizar tareas de distinta índole, como responder preguntas, hacer resúmenes, comprensión semántica y razonar (Naveed y cols., 2024). El artículo detalla un estudio sobre distintos modelos de

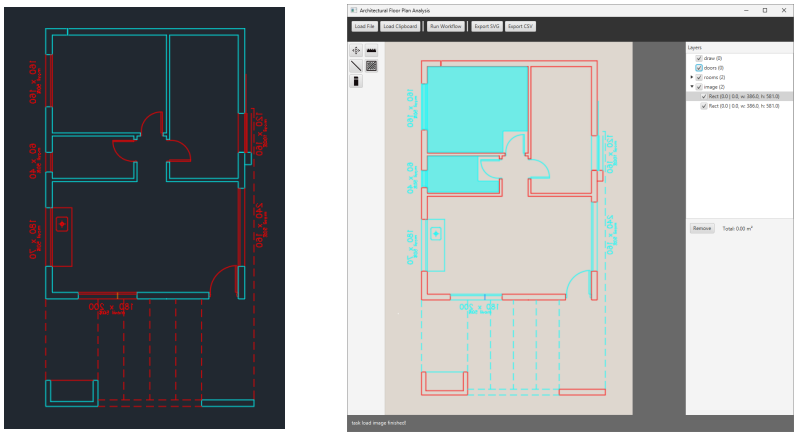


Figura 1.1: A la izquierda, plano original. A la derecha, plano procesado con la implementación de "Automatic analysis and simplification of architectural floor plans", mostrando en cyan la detección de dos de las 4 habitaciones.

aprendizaje de máquina y los compara en base a sus habilidades para contestar preguntas sobre un conjunto de imágenes. Lo novedoso del estudio es que utiliza imágenes en formato SVG (formato vectorial expresado en XML), y se prueban las capacidades de comprensión semántica de varios modelos LLM. Las pruebas se separan en 3 categorías: razonamiento visual y responder preguntas, clasificación de imágenes frente a cambios de distribución y con ejemplos limitados, y la generación de imágenes a partir de comandos visuales. El artículo concluye que los LLM muestran resultados prometedores para interpretar y generar imágenes visuales en formato SVG, y se presenta como un primer paso exploratorio en este campo. Esta técnica, en principio prometedora para ser utilizada con archivos *DXF*, no fue explorada en el presente proyecto ya que el mismo estaba avanzado cuando el artículo fue publicado.

Se encontró un sitio con un volumen suficiente de planos como para armar un conjunto de datos (Alcequiez, s.f.), sin embargo, los mismos no están marcados como para poder utilizarse con técnicas de aprendizaje automático.

Finalizado el proyecto, se entró en conocimiento de algunos conjunto de datos que podrían utilizarse para explorar técnicas de aprendizaje automático para el problema planteado, como son FloorPlanCAD (Fan, Zhu, Li, Zhu, y Tan, 2021) y Deep Floor Plan (ZENG, LI, Yu, y Fu, 2019) y Floor-SP (Chen, Liu, Wu, y Furukawa, 2019).

1.2. Contribuciones y objetivos del proyecto

En este contexto, se plantean los siguientes objetivos para el presente proyecto de grado:

- Estudiar las herramientas actuales para el análisis automático de planos de arquitectura.
- Implementar un prototipo que permita realizar de forma automática la detección de elementos estructurales en un plano de planta de una edificación. El prototipo debe dar como resultado un archivo en formato *DWG* o *DXF* con las mismas figuras que contenía el archivo original, clasificadas según su semántica (por ejemplo, todas las figuras que representan puertas deben estar en el mismo conjunto).

Los elementos principales para realizar el análisis estructural son las paredes, las columnas y las aberturas (como puertas y ventanas). Este proyecto se centra en la detección de paredes y puertas.

Por la naturaleza del problema y en base a la experiencia de los tutores, se planteó inicialmente utilizar herramientas y técnicas de aprendizaje automático y/o visión por computadora. Por motivos que se detallan a lo largo de este informe, se decantó por hacer el análisis en base a reglas.

La estrategia utilizada comienza con el estudio de artículos y herramientas relacionados al problema planteado, seguido de la implementación de un programa que adapte alguna de las soluciones encontradas, o explore un camino distinto en el caso de que ninguna aplique.

En el presente proyecto se explora y documenta una forma de clasificar elementos de un plano de arquitectura a partir de sus representaciones vectoriales almacenadas en archivos *DXF*, haciendo especial foco en mantener las dimensiones y proporciones originales.

1.3. Organización del documento

En el capítulo 2 se detalla el análisis realizado para entender la estructura de los archivos *DXF* y cómo se pueden procesar para clasificar las figuras del plano. El capítulo 3 describe las decisiones de tecnología y diseño tomadas durante la implementación del prototipo. El capítulo 4 explica detalles de implementación del prototipo, entre los que se encuentran los algoritmos, las estructuras de datos utilizadas y los problemas que surgieron durante el desarrollo. El capítulo 5 detalla las pruebas realizadas sobre el prototipo. Finalmente, en el capítulo 6 se expresan las conclusiones y pensamientos finales de los estudiantes sobre el proyecto, y se detallan ideas para posibles trabajos futuros que pueden basarse en este o extenderlo.

Capítulo 2

Análisis

En este capítulo se describen conceptos necesarios para comprender el resto del proyecto. Principalmente se destacan: la descripción general de un plano de planta, el formato de archivos *DXF*, la introducción a las herramientas de análisis geoespacial, el análisis de las figuras que componen un plano de arquitectura, y cómo se pueden definir algunas reglas, que combinadas, ayudan a encontrar puertas y paredes.

2.1. Plano de planta

"El plano de planta es una convención abstracta ortogonal al plano de suelo, situado a la altura del punto de vista humano, que representa gráficamente los elementos definidores del espacio: estructura, cerramientos, programa, y su relación con el entorno próximo" (González de la Cal y Blanco de Paz, 2017). Geométricamente, es una proyección sobre el plano horizontal de los elementos que forman un piso de una construcción. En una casa de dos pisos, lo normal es tener dos diagramas o planos de planta, uno por cada piso. A modo de ejemplo, en la figura 2.1 se puede observar el plano de un baño.

En un documento que contiene el plano de una edificación pueden existir otros diagramas además de los planos de planta, como pueden ser fachadas, vistas laterales, diagramas de instalación eléctrica y de cañería, etc. Sin embargo, en el presente proyecto solo se trabaja con planos de planta ya que así lo dictan los requerimientos establecidos por el cliente.

2.2. AutoCAD

AutoCAD (Autodesk, 2025) es un sistema de diseño asistido por computadora (CAD) ampliamente utilizado para la elaboración y edición de planos de arquitectura e ingeniería. En el contexto de este trabajo resulta relevante

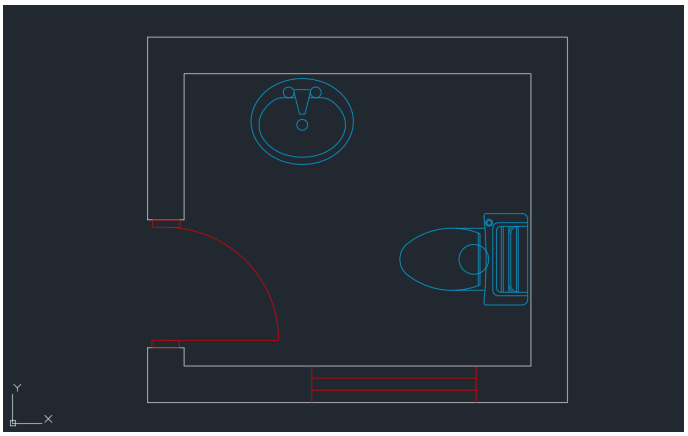


Figura 2.1: Plano de planta de un baño. Se observa la puerta a la izquierda, una ventana en la pared de abajo, el inodoro hacia la derecha, y el lavamanos arriba. En el contorno se observan las cuatro paredes que lo encierran.

por su uso extendido en la industria. En particular, es el software utilizado por Ingenium para procesar los planos previo al análisis estructural.

En este proyecto se utilizaron principalmente las herramientas de edición de geometrías 2D, en particular la visualización de planos y la consola de comandos para tareas puntuales de reparación y limpieza de los archivos, de las que se darán más detalles en la sección 5.1.

A continuación se describen algunos de los elementos principales presentes en un plano elaborado con *AutoCAD*:

Capas

Las figuras del diagrama se agrupan en capas (figura 2.2). Se pueden crear tantas capas como sean necesarias, las que a su vez, pueden tener todas las entidades (o figuras) que haga falta. Cada entidad puede pertenecer a una única capa.

Figura 2.2: Listado de capas de un plano. Captura tomada de la interfaz de AutoCAD.

Las capas contienen atributos que pueden usarse para la visualización de

las entidades que la componen. Estos atributos pueden ser el color base de la capa, el tipo de línea (punteada, continua, etc.), entre otros, pero no limita la posibilidad de cambiarlo manualmente para una entidad en particular. Por ejemplo, al agregar una línea a una capa cuyo color base es verde, la línea se visualizará de color verde, luego, se puede cambiar el color de esa línea a anaranjado sin afectar el color de toda la capa.

En el ejemplo de la figura 2.1 tenemos 3 capas: *Paredes*, *Sanitaria* y *Puertas y ventanas*.

Las capas se pueden ocultar, lo que permite despejar el diagrama para centrar la atención en algún aspecto particular del mismo.

Las capas ofrecen un mecanismo para agrupar las entidades de un plano según distintos criterios. Para la necesidad de este proyecto, es deseable que cada capa agrupe entidades con una misma función semántica. En la práctica no existe un estándar ni una convención de facto para nombrar las capas, ni para decidir qué entidades incluir en cada una. Como resultado, es frecuente encontrar en una misma capa elementos heterogéneos (por ejemplo, muros, cotas, símbolos y texto) y, a la vez, entidades equivalentes distribuidas en capas distintas. Además, muchas veces las capas se utilizan con fines operativos (impresión, visibilidad, colores o grosores) más que semánticos. Esta variabilidad impide asumir una correspondencia directa *capa* $\rightarrow$ *categoría*, lo que complica cualquier flujo de limpieza y clasificación.

Layouts

El espacio de trabajo donde se crea y edita el plano se conoce como *Model layout*. Consiste en una superficie infinita con un sistema de coordenadas donde se ubican todas las entidades que conforman el plano. Para cada plano existe un único *Model layout*.

Por otro lado, los *Viewport Layouts* representan vistas del plano y pueden ser muchos. Una forma de pensar en un *Viewport Layout* es como la "foto" de una porción del plano. A diferencia del Model layout, un Viewport layout tiene un área finita, comúnmente asociada a un tamaño de papel. Dentro de sus atributos se incluyen un nombre, que lo identifica, el tamaño de la imagen (o del papel donde se va a imprimir), el color de fondo, la escala, la orientación, etc. Cuando se imprime un plano, se hace a través de un *Viewport layout*. A modo de ejemplo, se pueden encontrar Viewports de cada planta de la construcción, de la fachada, o de la instalación de cañerías.

Los Layouts ofrecen otro mecanismo para agrupar las entidades de un plano. Sin embargo, al igual que sucede con las capas, no existen estándares o convenciones que aseguren contar con un layout que agrupe los elementos estructurales de un plano, por lo que tampoco se puede asegurar que faciliten la limpieza y clasificación.

Entidades

Los elementos que vemos en un diagrama se conocen como *entidades*, es decir, una entidad representa un elemento visual del diagrama. Existen decenas de tipos de entidades, como pueden ser texto, líneas, círculos, medidas, reglas, referencias, etcétera.

Los atributos de las entidades determinan cómo se visualizan las mismas, indicando entre otras cosas la posición, el color, el tamaño, el tipo de línea, etc.

A continuación se describen algunas entidades sobre las que se enfoca el trabajo de este proyecto.

LINE

Una entidad de tipo *LINE* representa un segmento de recta. Está definida por un punto de inicio y uno de fin.

POLYLINE

Consiste en una secuencia ordenada de puntos y se visualiza como segmentos de recta que conectan dichos puntos en orden (figura 2.3).

Los polylines también se utilizan para representar figuras cerradas, como un rectángulo.

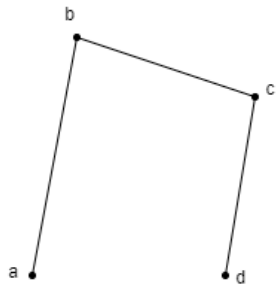


Figura 2.3: *POLYLINE* definido por la secuencia de puntos $[a, b, c, d]$ se visualiza como 3 segmentos de recta contiguos, $[a, b]$, $[b, c]$ y $[c, d]$.

ARC

Un arco representa una porción continua de una circunferencia (curva azul de la figura 2.4).

BLOCK & INSERT

Un *BLOCK* representa un conjunto de entidades reutilizables, que se pueden insertar en múltiples lugares.

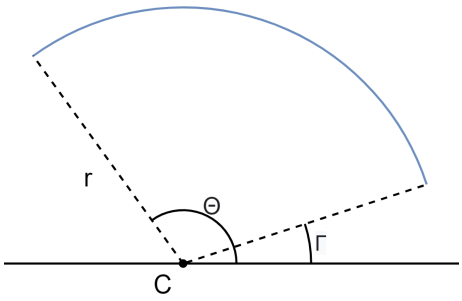


Figura 2.4: ARC definido por un centro C, un radio r, un ángulo de inicio Γ y un ángulo de fin Θ. Todos los ángulos se miden en sentido antihorario a partir del eje de coordenadas x.

Cuando una misma figura se repite muchas veces en un diagrama, es común guardar esa figura como un bloque e insertarla en todos los lugares donde se necesite. Los muebles, vehículos, plantas o elementos sanitarios se suelen almacenar como bloques.

Por ejemplo, en lugar de tener múltiples copias de las figuras que representan un inodoro, se almacenan estas figuras como el bloque llamado "Inodoro", el cual se inserta en cada baño del diagrama.

Para insertar un bloque en el diagrama, se utiliza una entidad de tipo *INSERT*, que indica la posición, la rotación y la escala de las figuras del bloque que se está insertando.

Atributos globales de un plano

Dentro de los atributos globales de un plano, se destacan las unidades de medida, y los sistemas de coordenadas.

Unidades de medida

Las unidades de medida determinan la escala del diagrama, y permiten transportar las coordenadas del diagrama a distancias físicas reales.

Sistemas de coordenadas

Existen tres tipos de sistemas de coordenadas en los planos creados en *AutoCAD*:

- **WCS** (World Coordinates System) hace referencia al sistema de coordenadas "de referencia". Las coordenadas de las entidades expresadas en este sistema no se modifican mientras no se realicen movimientos sobre las mismas.

- **UCS** (User Coordinates System) hace referencia a sistemas de coordenadas definidos por el usuario. Se utilizan para cambiar la vista (o la perspectiva) del Model layout. Por ejemplo, en lugar de alinear el plano con los puntos cardinales, el usuario puede elegir alinearlo con una calle lindera al edificio. Las coordenadas de las figuras pueden cambiar dependiendo del UCS que se esté usando.

- **ECS** (Entity Coordinates System) es un sistema de coordenadas relativo a la propia entidad. Se utiliza por ejemplo en los bloques, donde las coordenadas de las figuras que componen el bloque se expresan en ECS.

En este proyecto se utiliza siempre el **WCS**, ya que los valores medidos en este sistema de coordenadas son invariantes ante cambios en los otros sistemas.

2.3. Formato de archivos

DWG

DWG es el formato por defecto para los planos creados en *AutoCAD*, cuenta con mayor soporte por parte de dicho software y suele ser el favorecido por Autodesk. Sin embargo, es un formato propietario de la empresa (Adobe, 2024b), y es necesario pagar una licencia para desarrollar software que genere o modifique este tipo de archivos. Autodesk ofrece un SDK pago llamado RealDWG (Autodesk, 2024b) para trabajar con archivos DWG.

Otras bibliotecas capaces de trabajar con archivos DWG son:

- *LibreDWG* (GNU, 2024) es una biblioteca para el lenguaje de programación *C*. Se encuentra en desarrollo, por lo que no soporta muchas versiones del formato DWG. Está patrocinada por la fundación GNU.

- *com.1spatial:dwg-lib*, para Java (1spatial, 2024). La última versión fue publicada en 2017, lo que indica que está sin mantenimiento.

- *Teigha* (OpenDesign, 2024) es una biblioteca de la Open Design Alliance (ODA). Está disponible para C++, .NET y Java. Para poder acceder a la misma es necesario pertenecer a la alianza pagando una suscripción anual.

Durante el análisis inicial (Abril de 2021) se realizó una búsqueda de la especificación de este formato de archivos. En ese momento se concluyó que no existía una especificación pública del formato. Una nueva búsqueda realizada recientemente (Junio de 2024) arrojó un resultado no oficial (Open Design Alliance, 2018).

Debido a la falta de herramientas estables y libres para trabajar con este tipo de archivos, el foco del proyecto estuvo en trabajar con archivos de formato *DXF*.

DXF

DXF es la sigla de "Drawing Exchange Format" o "Drawing Interchange Format" (formato de intercambio de dibujos) y es un tipo de archivo vectorial que permite trabajar con diagramas en 2D y 3D (Adobe, 2024a).

A diferencia de *DWG*, es un formato abierto, por lo que existen múltiples herramientas y bibliotecas gratuitas que facilitan trabajar con el mismo. *DXF* está pensado para que los usuarios puedan compartir archivos y trabajar de forma colaborativa, sin necesidad de que todos usen las mismas herramientas de diseño.

2.4. Sistemas de Información Geoespacial (GIS)

En este proyecto se utilizan herramientas y técnicas GIS de bajo nivel, principalmente para determinar y comparar dimensiones (como distancias y áreas), y detectar intersecciones entre figuras.

Existen muchas definiciones de GIS, y dependiendo del contexto, algunas son más relevantes que otras. En este proyecto, resultan particularmente acertado pensar en GIS como "una herramienta para revelar lo que de otra forma es invisible en la información geográfica", y "una herramienta para realizar operaciones sobre información geográfica que es muy tediosa, costosa o imprecisa si se realizan a mano" (Longley, Goodchild, Maguire, y Rhind, 2005a).

En la práctica, los sistemas de información geográfica (SIG) abarcan un amplio abanico de problemas relacionados con información geoespacial (referida al espacio cercano a la superficie terrestre) y espacial (referida a cualquier tipo de espacio). Suelen estar compuestos por una combinación de software, hardware y datos, y permiten responder preguntas que involucran información espacial o geográfica. Su estudio conduce al desarrollo de numerosas técnicas y métodos, que pueden adaptarse a problemas más allá de los espacios geográficos, incluidos, por ejemplo, las superficies de otros planetas, el espacio cósmico y el cuerpo humano (imágenes médicas). Técnicas de SIG se utilizan incluso en el análisis de secuencias de ADN (Longley, Goodchild, Maguire, y Rhind, 2005b).

Muchos sistemas GIS diseñados para trabajar con mapas utilizan datos vectoriales muy similares a los usados en *AutoCAD*, y en particular en los archivos *DXF*. Esto permite utilizar herramientas y técnicas GIS para analizar los planos de arquitectura. Es posible encontrar analogías entre planos y mapas, por ejemplo, entre las habitaciones que forman una edificación y los países que integran un continente; determinar si dos habitaciones son contiguas podemos verlo como determinar si dos países son limítrofes.

A continuación se describen algunas de las herramientas o conceptos utilizados en este proyecto.

Buffer

Un buffer, desde el punto de vista geoespacial, es el área formada por todos los puntos del plano que están a menos de cierta distancia de la figura.

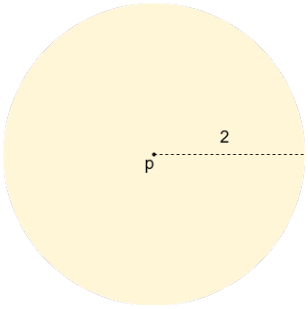


Figura 2.5: Buffer de distancia 2 del punto p

(a) (b)

Figura 2.6: En (a) se muestra en amarillo un buffer de tipo **round cap** de una línea. En (b) se muestra en celeste un buffer de tipo **flat cap** de una línea.

2.5. Sobre el uso de Aprendizaje Automático

La descripción original del proyecto indicaba el interés de usar técnicas de aprendizaje automático para resolver el problema planteado. Los primeros análisis fueron enfocados en estudios del estado del arte por un lado, y en la viabilidad de usar aprendizaje automático por otro.

Algo que las técnicas de aprendizaje automático comparten es su necesidad de datos para construir un modelo que resuelva el problema relacionados a estos. Una forma de clasificar el aprendizaje automático es de acuerdo a si la información que necesita para entrenar un modelo debe estar clasificada o no. Esto se conoce como aprendizaje automático supervisado o no supervisado. El aprendizaje supervisado se divide principalmente en clasificación y regresión (aproximación de funciones) mientras que el no supervisado tiene como objetivo aprender estructuras inherentes en los datos.

El aprendizaje automático supervisado parece ser una técnica prometedora para el problema de este proyecto, sin embargo, no fue posible conseguir un conjunto de datos que permitiese entrenar un modelo.

2.5.1. Visión por Computadora y su Aplicación en el Análisis de Planos Arquitectónicos

La visión por computadora, dota a los sistemas computacionales de la capacidad de extraer información significativa a partir de imágenes digitales, vídeos y otras entradas visuales.

Uno de los objetivos de la visión por computadora es entrenar a las máquinas para realizar tareas de análisis visual con mayor eficiencia y precisión que los seres humanos.

Entre las técnicas más utilizadas en este campo, destacan:

- **Clasificación de imágenes:** Consiste en asignar etiquetas categóricas a imágenes basándose en su contenido. Por ejemplo, clasificar imágenes según colores predominantes como "rojo", "amarillo" o "azul", permitiendo al sistema diferenciar entre estas categorías.
- **Detección de objetos:** Esta técnica identifica y localiza objetos específicos dentro de una imagen o vídeo. Se aplica en diversos campos, desde la detección de peatones en sistemas de seguridad vial hasta la identificación de anomalías en imágenes médicas.
- **Seguimiento de objetos:** Implica el rastreo del movimiento de objetos previamente detectados, ya sea en vídeos, secuencias de imágenes o en tiempo real. Esta técnica es crucial en aplicaciones como el seguimiento de vehículos en sistemas de transporte inteligente o el análisis de movimientos en eventos deportivos.
- **Recuperación de imágenes basada en contenido:** Se utiliza para navegar, buscar y recuperar imágenes de grandes bases de datos, basándose en el contenido visual de las imágenes en lugar de en metadatos asociados. Esta técnica es particularmente útil cuando la información requerida no se encuentra en los metadatos disponibles.

Como se detalló en el estudio del estado del arte (Ahmed y cols., 2011), (Ahmed y cols., 2012), (Heras y cols., 2013), (Dodge y cols., 2017), (Alexander Wyss and Florian Bruggisser, 2020) y (Cai y cols., 2023), la falta de un volumen suficiente de planos correctamente etiquetados para entrenar un modelo de aprendizaje automático, vuelve inviable el uso de esta técnica. Por este motivo, se exploran otras estrategias.

2.6. Detección en base a reglas

Al desestimar el uso de aprendizaje automático, surge la idea de generar una serie de reglas, que combinadas convenientemente, nos aproximan a la clasificación deseada. Para esto, se buscaron patrones y características geométricas que identifiquen ciertos elementos de los planos de planta. Las primeras pruebas con esta estrategia fueron prometedoras, por lo que se decidió avanzar con esta.

Las definiciones de las reglas se apoyaron en la experiencia y conocimientos de los clientes sobre cómo se representan los distintos elementos de una construcción en un plano.

Las reglas se redactan formalmente para que exista un paralelismo entre éstas y la forma en que funciona el programa.

Previo a ejecutar las reglas, es necesario filtrar y descomponer las entidades para quedarse solo con las que resultan de interés para el análisis. Para esto, se realizan las siguientes acciones:

- Sustituir los **INSERT**s por las entidades que componen cada **BLOCK** asociado.
- Descartar **BLOCK**s formados por muchas entidades pequeñas, ya que se observó que comúnmente representan vegetación o texturas irrelevantes para el análisis estructural.
- Descomponer los **POLYLINE**s en **LINE**s individuales.
- Filtrar entidades, dejando solo **LINE**s y **ARC**s, ya que las reglas definidas trabajan solo sobre este tipo de entidades.

2.6.1. Detección de puertas

Las puertas se dibujan con un arco de aproximadamente 90^{o} que indica la dirección de apertura de la puerta, y una línea o rectángulo que nos muestra una vista superior de la hoja de la puerta (Ching, 2015). Además pueden tener representaciones de los marcos. Esta definición también fue validada con los clientes.

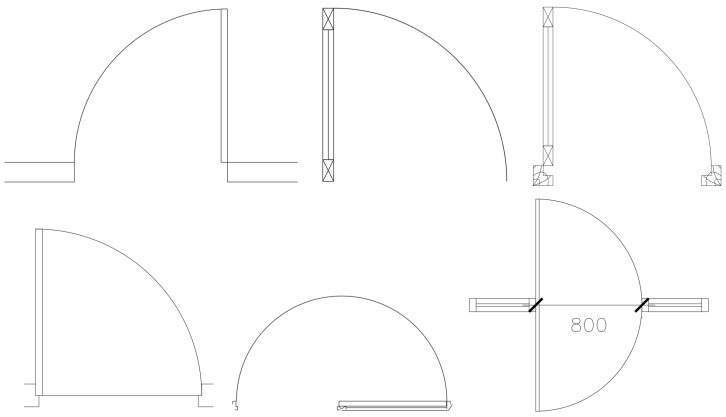


Figura 2.7: Ejemplos de representaciones de puertas en un plano de planta.

Dadas estas características, se define la regla 2.1 que permite identificar la mayoría de las puertas.

Tomando la figura 2.8 como referencia, una puerta se representa por una arco de centro B, ángulo α y radio r, tal que existe una línea (B, C).

Regla 2.1: Puertas

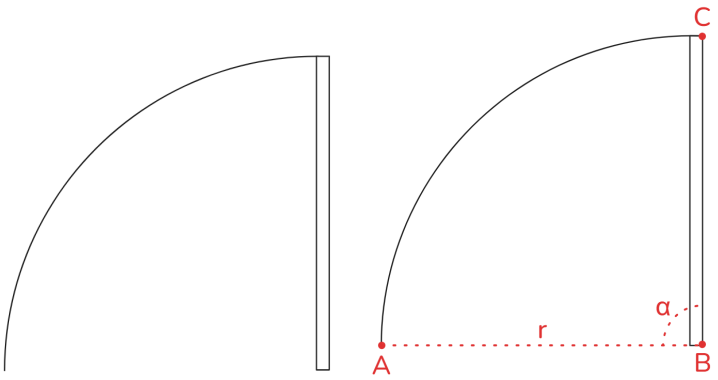


Figura 2.8: A la izquierda se observa una representación típica de una puerta en un plano. A la derecha, la misma representación anotada con los nombres de los elementos clave utilizados para definir la regla 2.1.

Cabe destacar que A y B son los puntos de anclaje de las puertas a las paredes, o dicho de otra forma, se corresponden con la ubicación aproximada de los marcos de las puertas en el plano. Esta idea se utiliza en la sección 2.6.3, por lo que se define de forma temprana el conjunto de puntos A y B de todas las puertas del plano como Π.

2.6.2. Líneas paralelas

La detección de paredes es uno de los focos principales del trabajo realizado en este proyecto. Estas se representan con dos segmentos de recta paralelos, enfrentadas, separadas por una distancia dada (equivalente al grosor de la pared).

Dos segmentos paralelos están *enfrentados* cuando sus proyecciones ortogonales se solapan; es decir, existe una línea perpendicular que corta a ambos ((a), (b) y (c) en la figura 2.9). Con esa noción, se define la regla 2.2.

En la práctica, Ψ contiene todas o la gran mayoría de las líneas que representan paredes. Sin embargo, en un plano también aparecen pares de paralelas que no son muros (mobiliario, texturas, etc.). La figura 2.10 muestra en color anaranjado todas las paralelas enfrentadas del plano; allí conviven paredes con otros elementos no estructurales. Las siguientes secciones introducen reglas adicionales para filtrar estos falsos positivos.

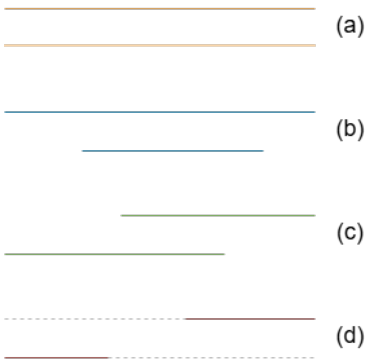


Figura 2.9: (a), (b) y (c) representan pares de líneas paralelas enfrentadas mientras que (d) muestra un par de líneas paralelas no enfrentadas.

Dado un conjunto S de segmentos de recta, y dos cardinales c y d con $c < d$, se define Ψ como el conjunto de segmentos $s \in S$ para los que existe $r \in S$ tal que:

I) r es paralelo a s,

II) la distancia entre las rectas que contienen a s y r pertenece a $[c, d]$,

III) s y r están *enfrentados* (sus proyecciones ortogonales se solapan).

Regla 2.2: Paralelas

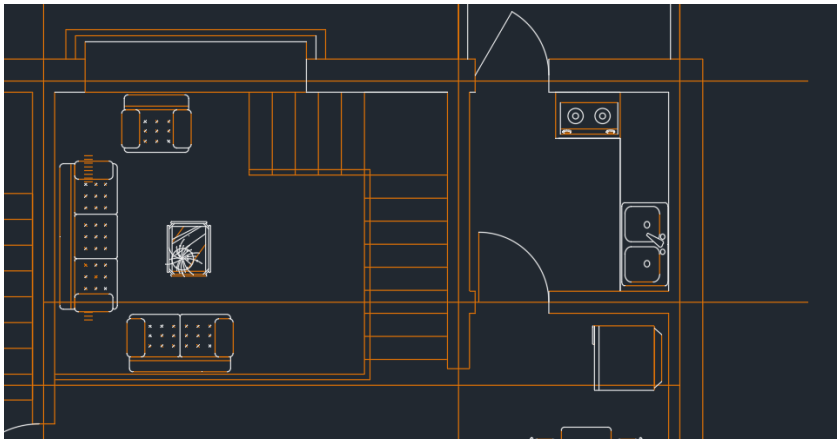


Figura 2.10: Se muestra la detección de las paralelas enfrentadas entre 15 y 35 cm en color anaranjado. En blanco el resto de las figuras del plano.

2.6.3. Paredes adyacentes a puertas

Una forma de buscar líneas paralelas que tienen altas probabilidades de representar paredes es utilizar la idea de que las puertas están amuradas a estas. Para ello, se toma como base el conjunto de líneas paralelas Ψ (regla 2.2) y se obtienen aquellas cercanas a los marcos de las puertas (conjunto Π definido al final de la sección 2.6.1). En la figura 2.11 se aprecia esta idea de manera gráfica.

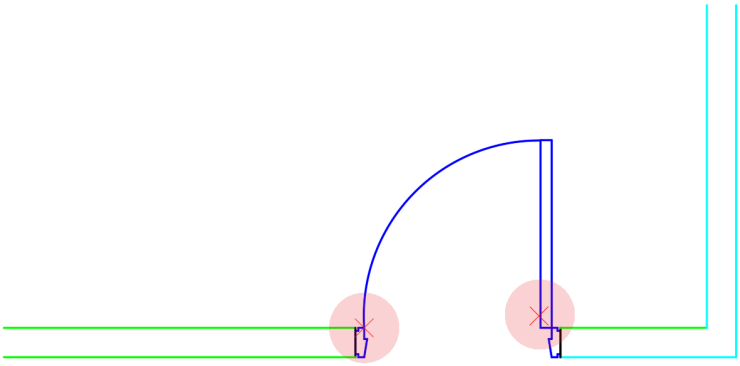


Figura 2.11: En azul se observa una puerta, en rojo los puntos que representan la ubicación aproximada de los marcos, y en verde las líneas paralelas que están a menos de d de dichos puntos. Por último, en cian se ven las líneas paralelas que no coinciden con la regla.

Más formalmente:

Dado un conjunto de puntos Π, un conjunto de segmentos de recta Ψ (como se definieron en las reglas 2.1 y 2.2), y un escalar d, se define Ω como el conjunto de segmentos de recta que pertenecen a Ψ y distan menos de d de algún punto de Π.

Regla 2.3: Paralelas adyacentes a puertas

Nótese que, dependiendo del parámetro d de la regla 2.3, puede ocurrir que algunas líneas que forman parte de las paredes no se detecten como tales. Esto se observa en la figura 2.11, donde una de las líneas paralelas que forman la pared horizontal no se detecta porque su distancia a la ubicación aproximada del marco de la puerta es mayor que d. Es posible tomar un valor de d mayor para detectar más líneas, pero al hacerlo, aumenta el riesgo de falsos positivos.

2.6.4. Paredes a continuación de otra pared

En la sección anterior, se describió una forma de encontrar líneas paralelas con mayor probabilidades de representar paredes, debido a que las puertas suelen estar unidas a las paredes.

Otra conclusión lógica es que las paredes suelen estar conectadas entre sí. Es decir, si tenemos líneas que representan paredes, es probable que las líneas paralelas unidas a estas (como las definimos en la regla 2.2) también representen paredes. Por ejemplo, en la figura 2.11 se observa en verde un tramo de pared unido a la puerta y en cian líneas paralelas que claramente continúan la pared.

Los conjuntos de figuras sobre los que opera la regla que se formaliza son Ψ (segmentos de recta paralelos) y Ω (paralelas cercanas a las puertas), como se definen en las reglas 2.2 y 2.3, respectivamente. Para la noción de «unión» de tramos de paredes, se consideran aquellos segmentos cuyos extremos son cercanos (según la definición de cercanía utilizada hasta ahora).

Dado un conjunto de segmentos de recta Ψ, un conjunto de segmentos de recta Ω (como se definieron en las reglas 2.2 y 2.3, respectivamente), y un escalar d, se define Δ inductivamente como:

- Todo elemento de Ω pertenece a Δ.
- Dado un segmento $(a, b) \in \Psi$, si existe un segmento $(h, i) \in \Delta$ tal que $min(|a, h|, |a, i|, |b, h|, |b, i|) < d$, entonces $(a, b) \in \Delta$.

Donde $|a, b|$ es la distancia entre los puntos a y b.

Regla 2.4: Paralelas adyacentes a paredes

2.6.5. Bloques densos

Como se explica en la sección 2.2, una herramienta disponible en *AutoCAD* para facilitar el dibujo de planos es el uso de bloques, que permite reutilizar dibujos. Un ejemplo muy común es el de las puertas, ya que un mismo plano suele incluir varias puertas iguales. Otro ejemplo habitual es el mobiliario, que permite identificar fácilmente la intención del arquitecto al distinguir un estar con sillones y mesas de un dormitorio con cama y mesas de luz.

En las pruebas iniciales se encontraron algunos planos donde la cantidad de entidades que formaban parte de elementos decorativos superaba ampliamente la cantidad de entidades que representaban elementos estructurales. Esto causa errores en la detección (se encontraban muchas líneas paralelas que formaban parte de muebles, plantas o vehículos) y perjudicaba la velocidad de procesamiento del programa, ya que se debían procesar individualmente más entidades de las que se procesarían si el plano no tuviese las decoraciones mencionadas. Por este motivo, se buscó identificar estos bloques para descartarlos del análisis, y así mejorar la precisión y acelerar el procesamiento.

Una característica compartida por los bloques decorativos es que concentran una gran cantidad de entidades en un área reducida (figura 2.12).

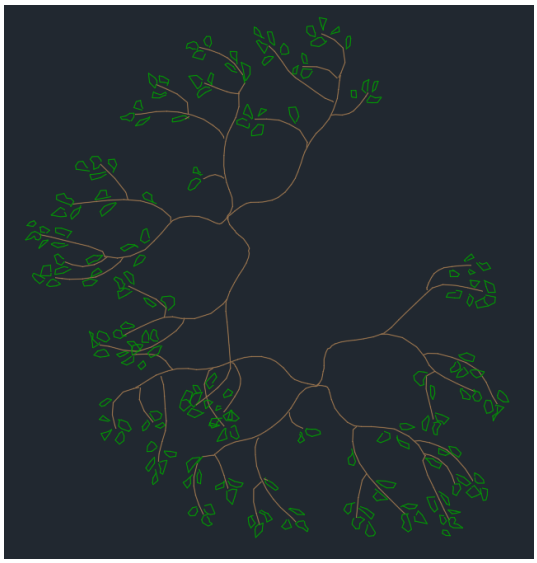


Figura 2.12: Captura de un arbusto tomada de un plano de ejemplo. El área es de unos $0{,}3m^2$ y contiene 1272 entidades (en su mayoría de tipo LINE y POLYLINE)

La propuesta para este caso fue definir una medida de densidad de bloques, basada en la cantidad de figuras que lo componen y el área que ocupa. En base a criterio de densidad, se define la regla 2.5 que permite identificar bloques demasiado densos.

Sea n la cantidad de figuras que componen un bloque, y a el área del polígono convexo de menor área que contiene dicho bloque. Definimos la densidad del bloque como $D = n/a$.
Dados N un escalar positivo y M un entero positivo, diremos que un bloque compuesto por n figuras es *denso cuando* $D > N$ *y* $n > M$.

Regla 2.5: Bloque densos

El entero M cumple la función de reducir los falsos negativos, evitando descartar bloques que contienen pocas entidades.

Capítulo 3

Diseño

En el capítulo anterior se describieron una serie de reglas que permiten identificar las figuras que cumplen ciertas características dentro de un plano. Al combinar estas reglas de distinta manera, se busca evaluar la capacidad de esta técnica para clasificar las figuras en distintos conjuntos que representen la intención del arquitecto para estas figuras.

3.1. Tecnología

3.1.1. Lenguaje, bibliotecas y software

De acuerdo con los requerimientos del proyecto y la experiencia previa de los integrantes del equipo, se priorizó el uso de los lenguajes de programación Python y C#. De esta manera, los esfuerzos se enfocaron en abordar el problema central del proyecto, en lugar de enfrentar la curva de aprendizaje de un nuevo lenguaje de programación. La decisión final fue usar *Python* por ser bueno para la creación de prototipos, y por la disponibilidad de bibliotecas compatibles entre sí para el manejo de archivos *DXF* y operaciones geométricas.

Algunas bibliotecas *Python* que permiten trabajar con archivos *DXF* son:

- `dxfgrabber` cuenta con funcionalidades para manipular archivos *DXF*. Está bajo la licencia MIT. No tiene mantenimiento desde 2018. En la documentación se sugiere utilizar `ezdxf` en su lugar. (Mozman, 2024)
- `pyautocad` está pensado para permitir la extensión de *AutoCAD* usando scripts *Python* y *ActiveX*. Se distribuye bajo licencia BSD. No tiene mantenimiento desde 2016. (reclosedev, 2024)
- `ezdxf` es un módulo *Python* que permite leer, modificar y escribir documentos *DXF*, soportando múltiples versiones del formato. Cuenta con

licencia MIT y recibe actualizaciones frecuentemente. (Manfred Moitzi, 2024).

Se optó por utilizar `ezdxf` por ser la única con mantenimiento activo. Además era compatible con `Shapely`, biblioteca de la que se habla a continuación.

Las herramientas de análisis geoespacial trabajaban con objetos muy similares a los que se encuentran en un plano de arquitectura (líneas, multilineas, polígonos, puntos, arcos, etc.) lo que las hacía particularmente útiles para implementar las reglas definidas en la sección 2.6. En la tabla 3.1 se evaluaron 4 bibliotecas populares que permiten realizar operaciones geométricas. Se compararon en base al tipo de licencia, mantenimiento del proyecto, facilidad de integración con `ezdxf` y funcionalidad. Para estas características, se consideró que `Shapely` (Shapely, 2024) era la más adecuada por tener operaciones geométricas puras (buffers, interesección, etc.), permitir integración nativa con `ezdxf` y ser de código abierto; `GeoPandas` (NumFOCUS, 2024) añade operaciones de consulta tabulares innecesarias para el prototipo, `GDAL/OGR` (Open Source Geospatial Foundation, 2024) y `PyProj` (Pyproj, 2024) se orientan a transformaciones y sistemas de referencia.

Existen numerosas bibliotecas y frameworks que ayudan a construir interfaces gráficas en *Python* (Comunity, 2024). En este proyecto se consideraron tres alternativas para la interfaz gráfica: `tkinter` (Python Software Fundation, 2024), `PyQt` (Fitzpatrick, 2020) y `FreeSimpleGUI` (FreeSympleGUI, 2025). La tabla 3.2 resume sus diferencias en términos de licencia, modelo de programación, curva de aprendizaje, ecosistema y adecuación al prototipo. Con base en estos criterios se optó por `FreeSimpleGUI`, decisión findamentada por su equilibrio entre funcionalidades, más que suficientes para el simple diseño de la interfaz del prototipo creado en este proyecto, y por su baja curva de aprendizaje.

Se utilizó `Pyinstaller` (Cortesi, 2024) para generar ejecutables autocontenidos para Windows, los cuales se compartieron con los clientes para que realizaran pruebas del prototipo. También se utilizó ocasionalmente `auto-py-to-exe` (Vollebregt, 2024) que proporciona una interfaz gráfica para `Pyinstaller`.

Finalmente, se utilizó *AutoCAD* (Autodesk, 2024a) para visualizar los planos en lugar de otras opciones código abierto debido a su amplia adopción en el sector, además de ser la herramienta utilizada en Ingenium.

Característica	GDAL/OGR	Pyproj	Shapely	GeoPandas
Licencia	MIT/X11 (Open Source)	MIT (Open Source)	BSD 3-Clause (Open Source)	BSD 3-Clause (Open Source)
Mantenimiento del proyecto	Muy activo, OSGeo Foundation. Releases regulares. Comunidad grande.	Muy activo. Mantenido por pyproj4. Actualizaciones frecuentes.	Muy activo. Parte del ecosistema consolidado. Desarrollo constante.	Muy activo, PyData ecosystem. Releases regulares. Gran adopción en ciencia de datos. Integración con pandas.
Compatibilidad con ezdxf	Sin conversión directa con ezdxf.	Sin conversión directa con ezdxf.	Integración nativa con ezdxf.	No tiene soporte nativo para DXF ni conversión directa con ezdxf.
Facilidad para prototipar	API compleja pero potente. Curva de aprendizaje moderada. Documentación extensa.	API muy simple. Fácil de usar. Sintaxis intuitiva.	Muy fácil de usar. API pythónica. Ideal para prototipos rápidos.	Sintaxis familiar (pandas). Análisis exploratorio intuitivo. Visualización integrada. Operaciones vectorizadas.
Soporte	Comunidad muy grande. Stack Overflow activo. Documentación oficial completa. Mailing lists activas.	Buena comunidad. GitHub Issues activo. Documentación clara. Ejemplos abundantes.	Comunidad sólida. Documentación excelente. Muchos tutoriales. GitHub Issues activo.	Comunidad muy grande. Documentación completa. Integración Jupyter notebooks. Soporte Stack Overflow activo. Abundantes tutoriales.
Features principales	**Lectura/escritura:** • Múltiples formatos • Operaciones ráster **Capacidades avanzadas:** • Filtros espaciales • Consultas SQL	**Transformaciones CRS (Coordinate Reference System):** • Pipeline transformations **Capacidades específicas:** • CRS validation • EPSG database access	**Operaciones geométricas:** • Creación de geometrías • Análisis espacial • Operaciones topológicas • Buffer, intersection, union • Predicados espaciales **Tipos de geometría:** • Point, LineString, Polygon • GeometryCollection	**Análisis espacial avanzado:** • DataFrames geoespaciales • Joins espaciales • Agregación espacial • Operaciones de superposición **I/O y visualización:** • Lectura/escritura de múltiples formatos • Plotting integrado • Integración con matplotlib

Tabla 3.1: Comparación de bibliotecas para manipulación y análisis geométrico

Característica	tkinter	PyQt	FreeSimpleGUI
Licencia	PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2	Dual: GPL / licencia comercial	GNU Lesser General Public License (LGPL 3) +
Enfoque	Toolkit estándar de *Python*; widgets y bucle de eventos.	*binding* para *Python* de la biblioteca Qt de C++; señales/slots y conjunto amplio de componentes.	Capa de abstracción sobre toolkits (p. ej., Tkinter/Qt) con API simplificada.
Complejidad	Moderada; API más detallada.	Alta; potente pero más compleja.	Baja; ideal para prototipos rápidos.
Funcionalidades relevantes	Conjunto base de widgets para formularios y diálogos. Compatibilidad con macOS, Windows y UNIX.	Amplio set de widgets, vistas avanzadas y capacidades ricas de UI. Compatibilidad con macOS, Windows y UNIX.	Componentes como cuadros de texto, botones, etc. Layouts simples. API basada en un event-loop y en eventos generados por el usuario.
Ecosistema	Documentación oficial extensa; ampliamente disponible.	Ecosistema Qt; comunidad grande y madura.	Documentación y ejemplos abundantes; comunidad activa.
Adecuación al prototipo	Requiere más código para layouts y manejo de eventos.	Mayor complejidad y esfuerzo de empaquetado.	Balance entre simplicidad y funcionalidades.

Tabla 3.2: Comparativa de bibliotecas para la GUI consideradas en el proyecto.

3.2. Arquitectura

3.2.1. Primera versión

En las primeras etapas del proyecto, la solución se construyó utilizando una arquitectura basada en scripts. Esta decisión se tomó con el objetivo de facilitar la exploración de ideas y estrategias de manera ágil, permitiendo realizar pruebas y validaciones de forma iterativa.

El programa no tenía interfaz gráfica, y la ejecución de los scripts se realizaba desde la consola de comandos.

Esta aproximación brindó flexibilidad durante la fase exploratoria, pero presentó algunas limitaciones. Una de las principales desventajas fue la duplicación de código, ya que las funciones se encontraban repetidas en múltiples scripts, lo que dificultaba el mantenimiento y la reutilización. Esto evidenció la necesidad de evolucionar hacia una arquitectura más estructurada y modular a medida que el proyecto se volvía más complejo.

3.2.2. Segunda versión

Luego de haber hecho algunas pruebas y confirmar que se podían clasificar con cierta certeza las figuras de un plano usando reglas, y tener algunas estra-

tegias distintas para clasificar las figuras, se pasó a una siguiente etapa en el desarrollo con el objetivo de generar un prototipo con interfaz gráfica que los usuarios pudiesen usar fácilmente, y que se pudiera iterar y corregir con más facilidad.

Se pasó a un paradigma orientado a objetos y una arquitectura modular (figura 3.1), donde cada clase existía en un único archivo *Python*, que se correspondía a su vez con un único módulo. Estos módulos y clases eran importados y reutilizados desde otros archivos *Python*. De esta forma, se extrajeron las funciones que se repetían en los scripts de la primera fase de desarrollo.

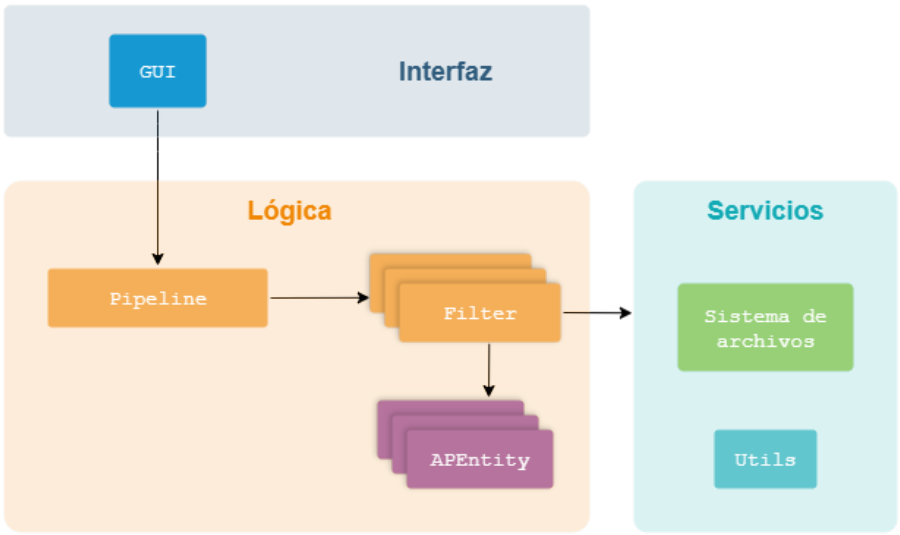


Figura 3.1: Diagrama de arquitectura (segunda versión).

Esta modularización, permitía que las mejoras y correcciones hechas en un módulo, quedaran inmediatamente disponibles en los scripts que usaban esos módulos.

La interfaz gráfica inicial permitió al usuario seleccionar el archivo que deseaba procesar, la estrategia con la que deseaba procesarlo, y la ubicación para almacenar el archivo con las figuras clasificadas.

Cada pipeline usa una combinación de filtros, los cuales toman conjuntos de figuras y las clasifican en subconjuntos más específicos. Todo esto se apoya en varias estructuras de datos, módulos con funciones utilitarias, y las ya mencionadas bibliotecas `ezdxf` y `Shapely`.

3.2.3. Versión final

Una limitante de la arquitectura anterior era que las estrategias se encontraban directamente en el código fuente. Esto implicaba que para crear una

nueva estrategia era necesario escribir un nuevo pipeline en *Python* y regenerar el ejecutable, lo cual no era amigable para lo usuarios.

En esta etapa, se exploró la posibilidad de crear estrategias a partir de archivos de configuración (figura 3.2). La motivación detrás de esto fue la idea de permitir que los propios usuarios pudiesen extender y mejorar la funcionalidad del programa creando sus propias estrategias (o pipelines).

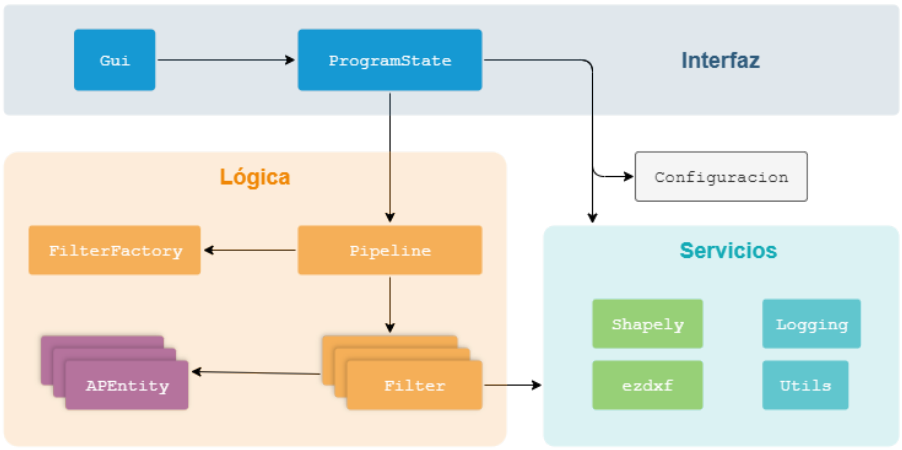


Figura 3.2: Diagrama de arquitectura final. El componente `ProgramState` orquesta la carga de configuraciones (incluyendo las configuraciones *YAML* de los pipelines), la carga de archivos *DXF*, la construcción y ejecución del pipeline seleccionado por el usuario en la GUI, y la persistencia del resultado. El `Pipeline` utiliza el componente `FilterFactory` para instanciar los filtros según la configuración.

En esta iteración se consolidó la interfaz gráfica, introduciendo técnicas de multithreading para mantener la fluidez mientras se ejecutaba un pipeline.

3.3. Pipes and Filters

Pipes and Filters (Woolf y Hohpe, 2003) es un estilo de arquitectura de software que propone descomponer una tarea compleja en una secuencia de tareas más pequeñas e independientes, llamadas Filters, las cuales se interconectan a través de canales de comunicación denominados Pipes.

Por ejemplo, consideremos un sistema de detección de fraude para un e-commerce, que pretende reaccionar a eventos particulares (como la realización una de compra o la apertura de un reclamo), detectar comportamientos maliciosos de los usuarios y tomar acciones automáticas (como bloquear usuarios o cancelar órdenes de compra). Esta tarea compleja se puede separar en tareas más específicas, como obtener información de las entidades relevantes (usuarios,

órdenes, productos, etc.), ejecutar reglas o modelos de aprendizaje automático para clasificar dichas entidades (usuarios de riesgo, productos de alto valor, etc), decidir si es necesario tomar alguna acción (por ejemplo, anular la compra) y por último ejecutar dichas acciones. Cada una de las tareas específicas se pueden implementar como componentes desacoplados (Filters), comunicados entre sí usando colas de mensajes (figura 3.3).

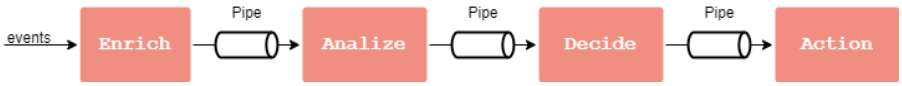


Figura 3.3: Ejemplo de pipeline de detección de fraude, formado por una secuencia de filtros que se comunican a través de tuberías.

Los *Pipes* son canales de comunicación, que transportan los mensajes de salida de un filtro hacia el siguiente. Dado que todos los componentes *Filter* utilizan la misma interfaz de entrada y de salida, se pueden combinar de distintas maneras, lo que permite construir soluciones flexibles y adaptables.

Otra ventaja que proporciona este estilo de arquitectura radica en el desacoplamiento que proporcionan los Pipes. Cada Filter puede enviar mensajes a su Pipe de salida, sin conocer el proceso que consumirá dicho mensaje posteriormente. Esta separación permite distribuir los pasos de procesamiento en diferentes instancias, lo que combinado con técnicas de multiprocesamiento y/o procesamiento distribuido puede reducir el tiempo requerido para resolver un problema complejo.

Finalmente, esta separación en componentes desacoplados facilita las pruebas y depuración del sistema. Cada Filter se puede probar por separado, simplificando significativamente la búsqueda de errores, al hacer foco en una funcionalidad específica en lugar de tener que probar todo el pipeline de forma integral.

3.3.1. Adaptación de la arquitectura

En el capítulo 2 se detallaron una serie de reglas que permiten clasificar y definir conjuntos con las figuras del plano. Dichas reglas, fueron pensadas para cumplir dos características principales:

- ser independientes y compatibles entre sí, de modo que se puedan combinar para formar reglas más complejas. Esto las convierte en excelentes candidatas para ser implementadas como Filters en un arquitectura de Pipes and Filters
- ser compatibles con operaciones espaciales, como las que ofrece `Shapely`

Cada una de estas reglas se implementa como un Filter, tal que recibe como entrada uno o más conjuntos de figuras, y devuelve también uno o más conjuntos de figuras.

A diferencia de la versión más simple de esta arquitectura, los filtros implementados en el proyecto reciben varias entradas, dependiendo del tipo de filtro. Por ejemplo, el filtro que detecta las rectas paralelas que estaban cerca de las puertas necesita un conjunto de puntos con la ubicación aproximada de los marcos de las puertas, y un conjunto de líneas sobre las que se realiza la búsqueda.

Además, cada filtro puede tener varias salidas, una por cada conjunto generado, como se muestra en la figura 3.4.

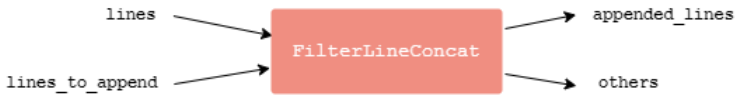


Figura 3.4: Filtro con varias entradas y varias salidas

Para proporcionar una interfaz común para los filtros, se implementa una colección (llamada APCollection), que representa un conjunto genérico de figuras. Luego, cada filtro utiliza solo las figuras relevantes para cumplir su función. Cada mensaje que se transmite entre dos filtros, es en realidad un objeto de tipo APCollection.

Cada Filter, puede tener varios Pipes conectados al mismo puerto de salida. Esto quiere decir que varios filtros pueden utilizar como input el mismo conjunto de figuras sin tener que ejecutar varias veces el filtro que produce dicho conjunto, como se observa en la figura 3.5. Por ejemplo, el filtro "FilterToApEntity" recibe como entrada un documento *DXF*, y produce un conjunto que contiene todas sus figuras. Este suele ser el filtro inicial de todos los pipelines, ya que traduce la información almacenada en el documento *DXF* a una estructura de datos que todos los demás filtros pueden procesar.

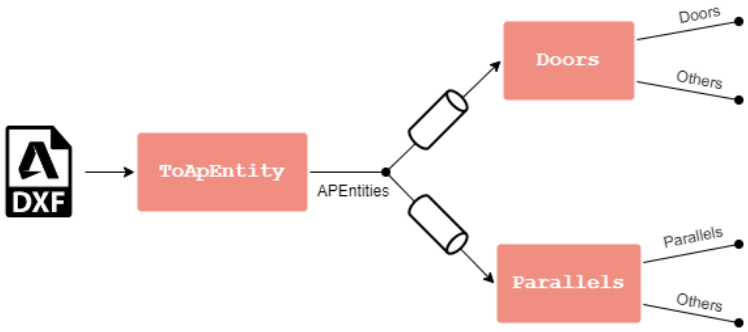


Figura 3.5: Filtro con dos Pipes en el mismo puerto de salida

Los filtros están implementados de tal forma que no modifican las colecciones que reciben como entradas. Esto es importante debido a que varios filtros pueden

utilizar las mismas colecciones, como se muestra en la figura 3.5.

Otra característica importante de los filtros implementados es que los mismos necesitan que todas sus entradas tengan información antes de comenzar a procesar. Por ejemplo, el filtro "FilterBufferIntersect" toma un conjunto de figuras y un conjunto de puntos, y retorna un conjunto de aquellas figuras cuya distancia a alguno de los puntos es menor a cierto valor configurado, y otro conjunto con las figuras cuya distancia a todos los puntos dados es mayor al valor antes mencionado. Dicho de otra manera, es una implementación generalizada de la regla 2.3. Cuando el pipeline se esté ejecutando, este filtro permanece en espera hasta haber recibido ambos conjuntos de figuras. Además, envía los resultados por el pipeline una vez que termine de clasificar todas las figuras. Esto quiere decir que los pipelines no funcionan procesando las figuras como un stream, sino como un sistema reactivo, donde un filtro reacciona cuando recibe un mensaje (un conjunto de figuras) en sus puertos de entrada.

El mecanismo de comunicación entre filtros es el Pipe, el cual se implementa usando el patrón de diseño Observer (Gamma, Helm, Johnson, y Vlissides, 1994). Cuando se construye el pipeline, este establece las relaciones entre los distintos puertos de los Filtros. Por cada filtro que recibe cierto resultado del filtro A, se crea un objeto de tipo Pipe, el cual se suscribe al puerto de salida de A, y conoce a qué puerto de entrada del siguiente filtro debe entregar ese resultado. Cuando el filtro A produce su resultado, notifica dicho resultado a todos los Pipes suscritos a ese puerto. Luego, cada Pipe inserta el resultado en el puerto de entrada correspondiente del filtro de destino.

Esta implementación del Pipe permite abstraer la comunicación entre filtros de forma liviana, ya que no requiere realizar transformaciones o almacenar los resultados de los filtros en buffers intermedios, sino que simplemente se le pasa una referencia de la colección generada a los filtros de destino.

3.4. APEntities

Hasta ahora, se ha hablado de "figuras" cuando se han mencionado los distintos elementos visuales que conforman un plano, los cuales pueden ser puntos, líneas, arcos, círculos, entre otros. En el contexto de *AutoCAD* y de los archivos *DXF*, estos elementos se denominan "Entities". La biblioteca `ezdxf` refleja a su vez el concepto de "Entity", ya que todas las clases específicas de figuras extienden de la clase `DXFEntity`.

Como se mencionó en la sección 2.3, cada tipo de figura del plano tiene una serie de atributos que describen con detalle cómo debe ser dibujada por los programas CAD. Esto incluye desde la posición, la escala, el sistema de coordenadas, hasta el color, el grosor, el trazo de las líneas, la capa a la que pertenece, etc.

En los análisis y pruebas realizados en este proyecto, el foco está en las

características espaciales de las figuras. Sin embargo, es de interés que se puedan perpetuar el resto de las características en el archivo generado. Es decir, si una línea tiene un trazo punteado en el archivo original, se desea que también sea puntuada en el archivo resultado. La motivación para mantener las características originales de las figuras es que éstas le dan una semántica particular a las mismas (una línea punteada puede ser una referencias de alineación, o también puede indicar que está por detrás de otro objeto proyectado, un punto dibujado con trazo grueso puede estar representando un pivot de una pieza con movimiento).

Al cargar el archivo, los datos de las entidades están en un formato definido por la biblioteca `ezdxf`, la cual utiliza programación orientada a objetos. Cada tipo de entidad *DXF* está representada por una clase, y los atributos de la entidad *DXF*, son atributos de la clase correspondiente.

Por otro lado, las reglas usadas para clasificar las entidades están definidas por operaciones espaciales, las cuales se resuelven utilizando `Shapely`.

`ezdxf` incluye un componente utilitario llamado `geo` que permite exportar las figuras representadas por clases propias a objetos `Shapely`. De esta forma, se logra obtener las figuras del plano en un formato manipulable por las herramientas de análisis de `Shapely`.

Hasta ahora, se tiene objetos de `ezdxf` por un lado, que mantienen todos los atributos de las figuras, pero no pueden usarse directamente con herramientas de análisis espacial, y objetos de `Shapely` por otro, que se pueden manipular con las operaciones de `Shapely`, pero que no tienen todos los atributos que es necesario persistir. Si se utilizan directamente los objetos `Shapely` en los filtros, al momento de guardar el resultado de la clasificación se pierden los atributos no espaciales de las figuras (figura 3.6). Para resolver este problema se evalúa utilizar composición o herencia múltiple.

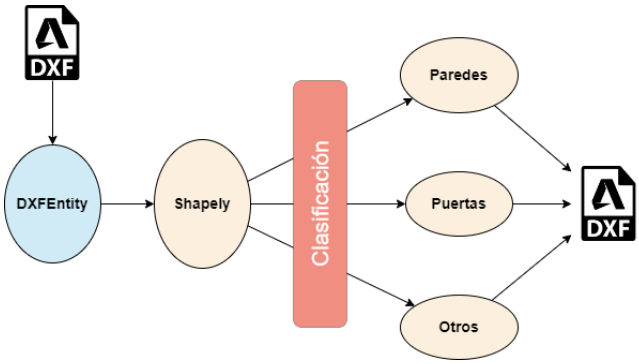


Figura 3.6: Idea de clasificación de figuras en conjuntos. Se convierten las entidades del archivo *DXF* en objetos de `Shapely`, se clasifican, y por último se genera a partir de estos un nuevo archivo *DXF*.

3.4.1. Composición

La primera opción evaluada consiste en crear una clase compuesta por un objeto *DXF*, y el correspondiente objeto `Shapely` (figura 3.7). Esta opción parece ser viable, pero en la práctica, dificulta la utilización de una funcionalidad clave de `Shapely`, que es la capacidad de realizar operaciones espaciales sobre colecciones de figuras, ya que al crear una `GeometryCollection` que contenga las figuras sobre las que se realiza la operación, se pierde la trazabilidad con los correspondientes objetos de `ezdxf`.

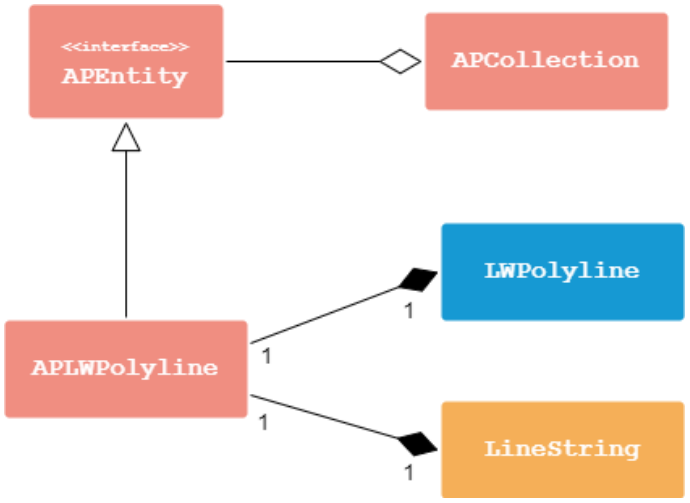


Figura 3.7: Clase `APLWPolyline` perteneciente al prototipo, que contiene una referencia a un objeto de clase `LWPolyline` (perteneciente a `ezdxf`) y una referencia a un objeto de clase `LineString` perteneciente a `Shapely`. También se observa la interfaz `APEntity` que se puede utilizar para almacenar todas las figuras en una colección, la cual se utiliza como entradas y salidas de los filtros.

3.4.2. Herencia

La segunda opción evaluada consiste en aprovechar la herencia múltiple de *Python*, la cual permite crear una clase que herede las características de múltiples clases. De esta forma, es posible crear clases capaces de comportarse simultáneamente como objetos de `ezdxf` y de `Shapely`, como se puede apreciar en la figura 3.8.

De esta forma, se puede usar el mismo objeto para:

- transmitir conjuntos de figuras entre los filtros
- ejecutar operaciones espaciales utilizando las funciones de `Shapely`
- construir el documento *DXF* resultado del procesamiento del pipeline usando `ezdxf`

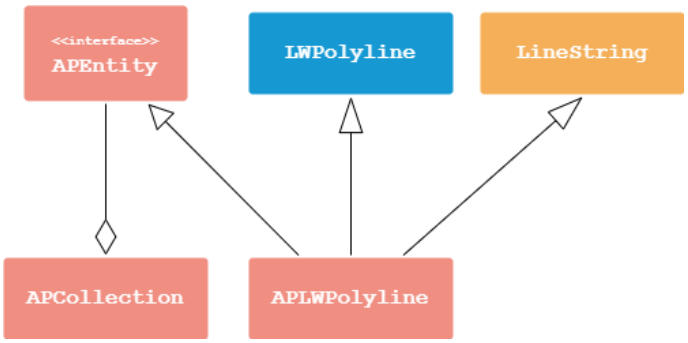


Figura 3.8: Ejemplo de herencia múltiple. La clase `APLWPolyline` hereda de `LWPolyline` (`ezdxf`), `LineString` (`Shapely`) y `APEntity` (definido en nuestro prototipo). Esto último permite tener una colección de elementos que se comportan tanto como entidades *DXF* como objetos `Shapely`.

Esta fue la segunda opción evaluada, y la que finalmente se utilizó en el prototipo.

3.4.3. Herencia y referencia

A continuación, se menciona una tercera opción que puede considerarse más elegante y menos propensa a ciertos problemas que se asocian con la herencia múltiple. De haber pensado antes en este diseño, es probable que hubiese sido el utilizado en el prototipo.

La idea es heredar de las clases de `Shapely`, como en la opción anterior, pero en lugar de heredar también de las clases de `ezdxf`, mantener una referencia a las mismas, como se muestra en la figura 3.9.

De esta forma es posible utilizar las operaciones de `Shapely` directamente sobre los objetos de tipo "AP*" para realizar la clasificación de las figuras del plano, y mantener la trazabilidad (o las referencias) de los objetos `ezdxf` que permiten luego generar el documento resultado del procesamiento del pipeline.

La ventaja principal de esta opción es que evita posibles colisiones de nombres de métodos y atributos que se pueden dar al utilizar herencia múltiple.

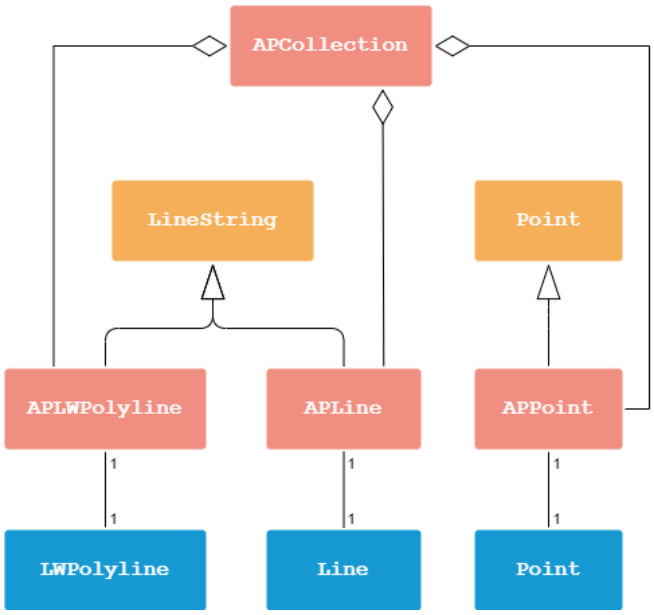


Figura 3.9: Combinación de herencia y referencia. En rojo, las clases de la aplicación que heredan de las clases de `Shapely` (en anaranjado) y contienen una referencia a las clases de `ezdxf` (en azul).

Capítulo 4

Desarrollo

4.1. Entorno de desarrollo

AutoCAD está disponible solo para Windows y macOS (Autodesk, 2024c). Se utilizó Visual Studio Code como entorno de desarrollo integrado, el cual provee entre otras funciones, resaltado de código y autocompletado.

Para el control de versiones del código se utilizó Git, con un repositorio remoto en GitLab.

4.2. Implementación de filtros

Se describe la implementación de los distintos filtros que fueron utilizados, destacándose en cada uno de estos los problemas encontrados y las soluciones aplicadas.

4.2.1. BaseFilter

`BaseFilter` define el contrato y el ciclo de ejecución común para todas las etapas de procesamiento del sistema. Cada subclase implementa el método `execute`, mientras que la clase base gestiona las entradas, la verificación de requisitos, la ejecución y la publicación de resultados. De esta forma se facilita la composición de filtros.

Cada implementación de `BaseFilter` registra las entradas requeridas cuando se construye, es decir, cada filtro conoce las entradas necesarias para poder ejecutar.

Los filtros reciben datos con el método `feed`, que almacena la entrada. Se comprueba si tienen las entradas requeridas (required_inputs) y se invoca el

método `execute`. Por último, los resultados se depositan en *`output_buffers`* y se difunden mediante el método `finish`.

Cuando se construye el pipeline, se conectan las salidas de un filtro con las entradas del siguiente usando el método `set_output_pipe`, que asocia una salida de un filtro a una o varias entradas de otros filtros.

`BaseFilter` implementa los siguientes patrones de diseño:

Template method

Establece el esqueleto del proceso de una etapa:

1. Ingesta
2. Verificación de disponibilidad de entradas
3. Ejecución
4. Publicación de resultados

La variación se da en el método `execute`, implementado por la subclase. Este patrón aporta consistencia y reduce la duplicación de lógica en cada filtro.

Observer

Cada salida mantiene una colección de destinos registrados mediante `set_output_pipe`. Al finalizar, el método `finish` notifica a todos los suscriptores de cada salida enviándoles los datos correspondientes. Aunque la suscripción se da en la construcción del pipeline y no en tiempo de ejecución, la relación "sujeto con múltiples observadores" se mantiene: el filtro no necesita conocer quién consume sus resultados.

4.2.2. Líneas paralelas

Se presentan 4 aspectos a solucionar en la implementación de este filtro:

1. determinar si dos segmentos de línea dados son paralelos.
2. determinar si la distancia entre éstos está entre dos valores dados c y d, con $c < d$.
3. determinar si los segmentos están "enfrentados" (ver figura 2.9).
4. buscar todos los segmentos de línea de un conjunto que cumplen las tres condiciones anteriores.

Para determinar si dos segmentos de recta son paralelos en un plano cartesiano, alcanza con determinar si las pendientes de las rectas que los contienen son iguales.

En lugar de comprobar explícitamente una recta perpendicular y calcular distancias segmento a segmento, se implementa la noción de "distancia en $[c, d]$"

y “enfrentadas” como se describe en la sección 2.6.2, utilizando dos buffers alrededor de cada segmento s: uno *exterior* a distancia d y un *interior* a distancia c, ambos con extremos tipo *flat cap* (figura 4.1). Un segundo segmento r se considera válido si interseca el buffer exterior y no interseca el interior. Así, en un único test se verifican simultáneamente distancia y enfrentamiento.

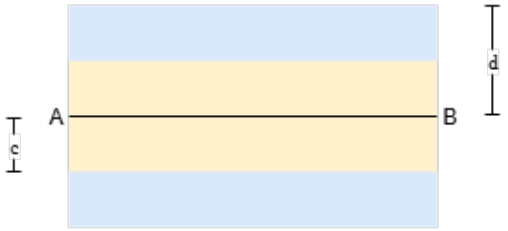


Figura 4.1: En celeste, un buffer de tipo *flat cap* y distancia d. En amarillo, un buffer de tipo *flat cap* y distancia c. Una paralela enfrentada es aquella que interseca la zona celeste y no interseca la zona amarilla.

Por último, para resolver la búsqueda de todos los segmentos de recta que cumplen las condiciones 1, 2 y 3 se implementaron tres algoritmos distintos, con la intención de encontrar el de mejor rendimiento para esta tarea.

Fuerza bruta

La solución más intuitiva consistía en comparar todo par de segmentos de recta, y quedarse con los que cumplían las condiciones 1, 2 y 3. Un pseudocódigo de este algoritmo se presentó en el bloque de código 4.1.

Listing 4.1: Paralelas: fuerza bruta

```
d_min = distancia_min
d_max = distancia_max
segmentos

paralelas = nuevo conjunto
para cada s en segmentos
    min_buffer(s, d_min)
    max_buffer(s, d_max)
    para cada r en segmentos
        si paralelos(s, r)
                y intersecta(max_buffer, r)
                y no intersecta(min_buffer, r)
            paralelas.agregar(s)
            terminar para
        fin si
   fin para
fin para
```

A pesar de que se pueden hacer algunas optimizaciones a este algoritmo, evitando comparar más de una vez el mismo par de rectas, el orden de ejecución es $O(n^2)$ (siendo n la cantidad de líneas del plano), lo que hace que el tiempo de ejecución sea demasiado largo cuando el plano es muy complejo.

Ordenamiento

El siguiente algoritmo consiste en definir un criterio de orden entre los segmentos de recta tal que se pueda hacer una búsqueda mucho más eficiente sobre el conjunto ordenado. Para esto, se definió un criterio de orden ajustado para las condiciones 1 y 2 definidas anteriormente.

Tomando como referencia la figura 4.2, diremos que $(a, b) < (g, h)$ si

- $\Gamma < \Theta$ o bien
- $\Gamma = \Theta$ y $x_1 < x_2$ (si existen) o bien
- $\Gamma = \Theta$ e $y_1 < y_2$

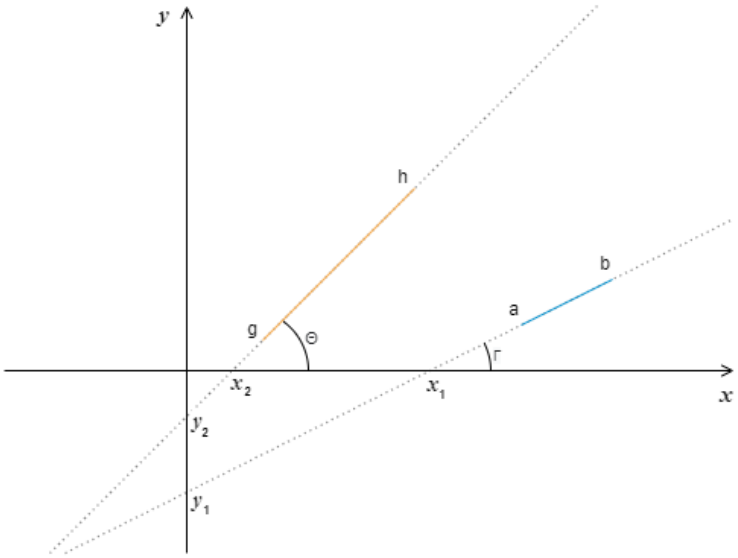


Figura 4.2: Se consideran dos segmentos, (a, b) y (g, h), tal que Γ es el ángulo que forma (a, b) con el eje x, Θ es el ángulo que forma (g, h) con el eje x. x_1 e y_1 son los puntos de corte de la recta que contiene (a, b) con los ejes x e y respectivamente. x_2 e y_2 son los puntos de corte de la recta que contiene (g, h) con los ejes x e y respectivamente.

Las líneas se ordenan en primer lugar según su ángulo, esto hace que todas las líneas con el mismo ángulo (paralelas) queden contiguas en la lista. Luego

se ordenan según su corte con el eje X o el eje Y (posición especial donde se encuentra), lo que hace que las líneas cercanas queden contiguas.

Una vez implementado el criterio de orden antes mencionado, se utilizó la función `sorted` de *Python* para ordenar la lista. El algoritmo utilizado por esta función es *Timsort* (*Python 2.3 Changelog*, s.f.), el cual tiene un orden de ejecución de $O(n \log(n))$ (Peters, s.f.).

Por último, luego de tener la lista de segmentos de línea ordenados, alcanza con comparar cada segmento con los que le siguen en la lista, hasta encontrar uno que ya no es paralelo (con una tolerancia dada) o cuya distancia es mayor que d.

Por ejemplo, tomando la lista de líneas ordenadas de la figura 4.3, alcanza con realizar 4 comparaciones para determinar los pares de paralelas enfrentadas ($[(o, l), (k, p), (k, m), (p, m)]$), lo cual es una mejora importante respecto a las 15 comparaciones que se harían usando fuerza bruta (evitando comparar más de una vez el mismo par de segmentos).

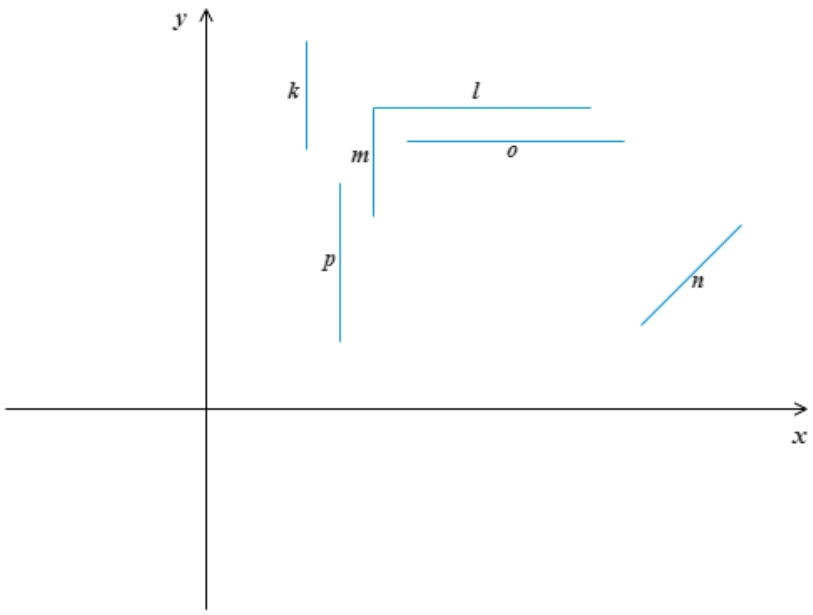


Figura 4.3: Conjunto de líneas en un plano. Al ordenarlas según el criterio anterior se obtiene la lista $[o, l, n, k, p, m]$.

Dado que se utiliza una tolerancia para determinar si dos segmentos son paralelos, puede ocurrir que un segmento con ángulo 0° y otro con $360° - (tolerancia/2)$ se consideren paralelos y queden en extremos opuestos de la lista. Para sortear este problema se utilizó una lista circular.

En el peor caso, todos los segmentos de recta son paralelos y la distancia entre ellos es menor a d. En este caso, el orden de ejecución es $O(n\log(n)+n^2)$ ($n\log(n)$ por el algoritmo *Timsort* y n^2 por la necesidad de comparar todos los segmentos entre sí). Sin embargo, en la práctica las ejecuciones se acercan más a un orden de $O(n\log(n))$.

QuadTree

QuadTree es una estructura de datos que permite almacenar información para luego recuperarla utilizando claves compuestas (Finkel y Bentley, 1974). Consiste en almacenar objetos en una estructura de árbol, lo que ayuda a acelerar las búsquedas al evitar comparaciones entre objetos lejanos. Aplicada a los planos de planta, permite responder de forma eficiente a consultas sobre un conjunto de entidades, como pueden ser "¿cuál es la entidad más cercana a cierto punto dado?" o "¿cuáles son todas las líneas del plano contenidas en cierta área?".

La idea es limitar el trabajo a un área rectangular del plano, donde se encuentran todos los objetos sobre los que se deseaban hacer consultas. Dicha área se divide sistemáticamente en subregiones rectangulares de igual tamaño a medida que se agregan elementos al QuadTree. La estructura de datos toma una forma de árbol, donde cada nodo contiene un subconjunto de objetos del plano, y además puede tener 0 o 4 nodos hijos.

Un QuadTree cumple con las siguientes características:

- El nodo raíz del árbol se corresponde con el rectángulo de mayor tamaño, que representa la totalidad del área cubierta por el árbol. Para que un objeto se pueda almacenar en el árbol debe estar completamente contenido en esta área.
- Cuando un área rectangular contiene más de $MIN\_ELEMENTS$ objetos, esta se divide en cuatro rectángulos de igual tamaño (que se corresponden con cuatro nodos hijos en el árbol), y los objetos contenidos en dicha área se reparten entre estos cuatro rectángulos.
- Todo objeto perteneciente a un nodo debe estar completamente contenido en el área correspondiente a ese nodo.
- Todo objeto perteneciente al árbol debe almacenarse en el nodo de menor tamaño que lo puede contener.

La figura 4.4 muestra a la izquierda una región del plano que contiene 8 figuras, además muestra las subregiones correspondientes a los distintos nodos del QuadTree. A la derecha, se puede ver el QuadTree que almacena dichos objetos.

Tomando de ejemplo el QuadTree de la figura 4.4, si se desea obtener todos los objetos que colisionan con el punto E, se realizan las comparaciones

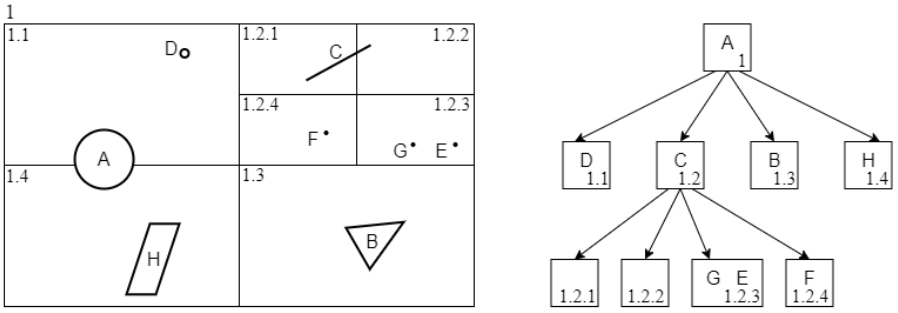


Figura 4.4: A la izquierda se observa un área rectangular del plano que contiene varias figuras. Dicha área está dividida en subregiones. Cada región está numerada para poder referenciarla. A la derecha, el QuadTree generado a partir de dicho plano donde $MIN\_ELEMENTS \geq 2$. Los números de cada nodo indican a qué región del plano corresponde. El círculo A se almacenaba en el nodo 1, ya que es la única región que puede contenerlo completamente; el triángulo B se almacena en el nodo 1.3, ya que es la región de menor tamaño que puede contenerlo; el segmento de recta C se almacena en el nodo 1.2, ya que es el de menor tamaño que puede contenerlo completamente, y así sucesivamente.

$[(E, G), (E, C), (E, A)]$, correspondientes a comparar E con todos los elementos almacenados en su mismo nodo y en los nodos ancestros.

El orden de ejecución de insertar un elemento en esta estructura es $O(\log(n))$, de modo que el orden de ejecución para crear un árbol con n elementos es $O(n \log(n))$.

En el caso de las búsquedas, el orden de ejecución depende del tamaño y distribución de los objetos que el árbol contiene, y del tipo de búsqueda que se desea realizar. Por ejemplo, para buscar todos los elementos del árbol que intersecan un elemento dado X, se necesita comparar X con todos los elementos de todos los nodos cuyos rectángulos lo intersecan.

El QuadTree se implementó como una estructura auxiliar de la clase "APCollection", la cual se genera solo cuando se realiza la primera consulta sobre los elementos de la colección. De esta forma, se evita el costo de crear el QuadTree cuando los filtros que utilizan la colección no lo necesitan. La principal ventaja de agregar esta estructura en "APCollection", es que todos los filtros se pueden beneficiar de la eficiencia que esta proporciona en las consultas.

Ordenamiento vs. QuadTree

Se realizaron pruebas para comparar QuadTree con el algoritmo de ordenamiento descrito en la sección 4.2.2 utilizando como referencia el filtro de paralelas. Se contaron la cantidad de comparaciones entre líneas. Los resultados indican que no hay un algoritmo que sea mejor en todos los casos (figura 4.5).

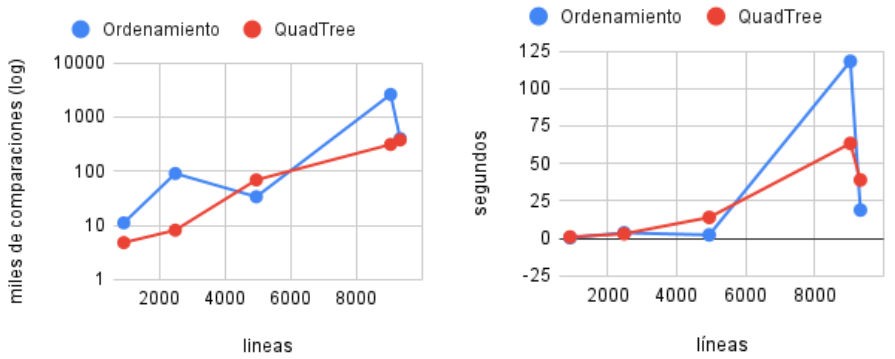


Figura 4.5: A la izquierda, gráfico de cantidad de comparaciones entre líneas (en escala logarítmica) por cantidad de líneas del plano. A la derecha, gráfico del tiempo de procesamiento por cantidad de líneas.

Llama la atención que el plano que requirió más comparaciones y demoró más, no es el que tiene la mayor cantidad de líneas. Al observar el resultado del procesamiento de dicho plano (figura 4.6), se vio que el mobiliario del mismo no fue filtrado por el filtro de bloques densos. Como resultado de esto, ambos algoritmos debieron procesar una mayor cantidad de líneas paralelas cercanas entre sí. En particular, el conjunto de sillas resaltadas en la imagen está formado por más de 7500 líneas individuales, que, a pesar de no ser paralelas entre sí, debieron ser comparadas por los algoritmos.

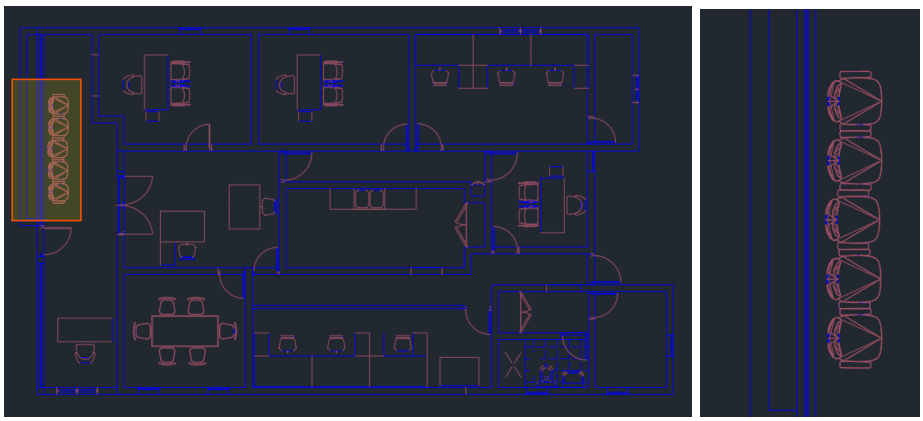


Figura 4.6: A la izquierda, imagen de un plano utilizado para probar los algoritmos de Ordenación y QuadTree. A la derecha, recorte del mismo plano destacando un conjunto de sillas.

4.3. Interfaz gráfica

Para simplificar el uso del prototipo se creó una interfaz gráfica sencilla que pretende guiar al usuario en los 3 pasos necesarios para procesar un plano (figura 4.7). La interfaz permite seleccionar un archivo *DXF* e intenta obtener las unidades de medida del plano. En caso de que las unidades detectadas no sean correctas, el usuario puede cambiarlas. Luego, se selecciona y ejecuta el pipeline que se usará para procesar el archivo. Por último, una vez que termina la ejecución del pipeline, permite guardar el archivo generado.

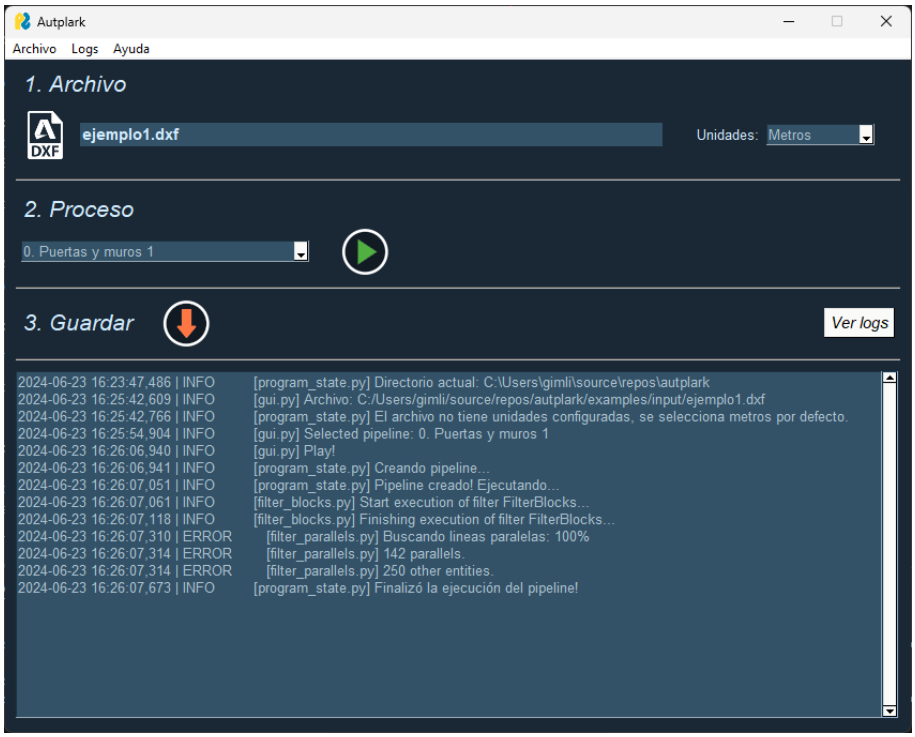


Figura 4.7: Interfaz gráfica

Opcionalmente, el usuario puede optar por ver los logs que se producen mientras el pipeline estaba corriendo.

Para permitir que la interfaz responda al usuario mientras se ejecutan distintas tareas de la aplicación, el event-loop se ejecuta en un hilo separado al hilo usado para ejecutar otras tareas (como la ejecución del pipeline).

4.4. Configuración

En la sección 3.3 se mencionó que la arquitectura final permite crear pipelines (o estrategias) para procesar un plano a partir de archivos de configuración en formato YAML. En esta sección se explica más en detalle el formato general de la configuración de un pipeline.

Listing 4.2: Pipeline Básico

```yaml
---
name: Pipeline Ejemplo

filters:
  - type: FilterBlocks
    name: blocks
    inputs:
      drawing: DRAWING
    outputs:
      high_density_blocks: high_density_blocks
    config:
      max_density: MAX_DENSITY
      min_entity_count: MIN_ENTITY_COUNT
      UNITS: UNITS

  - type: FilterToAPEntity
    name: ap
    inputs:
      drawing: DRAWING
      high_density_blocks: high_density_blocks
    outputs:
      ap_entities: supported
      unsupported_entities: unsupported

outputs:
  - output: supported
    layer: Supported Entities
  - output: unsupported
    layer: Unsupported Entities
```

En el bloque de código 4.2 se puede ver un ejemplo básico de configuración de un pipeline. Se compone de 3 partes principales:

- *name*: es el nombre del pipeline, tal como se muestra en la interfaz gráfica para que el usuario pueda seleccionarlo.
- *filters*: es una lista de configuraciones de filtros. Puede contener tantos filtros como sea necesario.
- *outputs*: contiene la información necesaria para crear el archivo resultado de la ejecución del pipeline. Bajo este atributo, se indica una lista de capas en las que se guardan las salidas particulares de los filtros. Cada elemento

de la lista está compuesto por un atributo *output* que indica el nombre de un pipe, y un atributo *layer* que indica el nombre de la capa del archivo donde se guardarán las figuras que se envíen por dicho pipe. En el ejemplo anterior, las figuras enviadas por el pipe "supported" del filtro de tipo "FilterToAPEntity" se guardarán en la capa "Supported Entities", y las figuras enviadas por el pipe "unsupported" se guardarán en la capa "Unsupported Entities". Pueden haber varios outputs que guarden figuras en la misma capa.

Las configuraciones de los filtros contenían los siguientes atributos:

- *type*: es el tipo de filtro que se va a utilizar. Cada tipo se corresponde con un filtro de los que se muestran en el apéndice E.
- *name*: es el nombre de la instancia del filtro dentro del pipeline. Se utiliza para distinguir los filtros, en caso que existan varios del mismo tipo. No puede haber dos filtros con el mismo nombre.
- *inputs*: es un mapa que indica los pipes que se conectan a los puertos de entrada del filtro. Cada puerto se corresponde con un input necesario para la ejecución del filtro. Las claves del mapa representan el nombre del puerto, mientras que los valores representan el pipe que se conecta con ese puerto. Los nombres de los puertos (es decir, las claves del mapa) están definidos para cada tipo de filtro, mientras que los nombres de los pipes los puede definir el usuario. En las reglas definidas en el capítulo 2, los inputs se corresponden con los conjuntos sobre los que trabaja la regla.
- *outputs*: es un mapa que indica los pipes que se conectan a los puertos de salida del filtro. Se cumplen las mismas reglas que para los inputs. Las claves se corresponden con los nombres de los puertos de salida del filtro, mientras que los valores son los nombres de lo pipes conectados a esos puertos. En las reglas definidas en el capítulo 2, los outputs se corresponden con los conjuntos que se definen en la regla.
- *config*: son configuraciones particulares de cada filtro. En las reglas definidas en el capítulo 2, las configs se corresponden con los parámetros de la regla. Por ejemplo, un filtro de tipo FilterBlocks es la implementación de la regla 2.5, tal que el parámetro de configuración `max_density` se corresponde con el escalar N definido en la regla, y `min_entity_count` se corresponde con el entero M. *config* puede no agregarse en caso de que el tipo de filtro no requiera ninguna configuración, como sucede con los filtros de tipo "FilterToAPEntity".

Los filtros de tipo "FilterToAPEntity" se encargan de convertir las figuras del plano (almacenadas dentro de un objeto de tipo Drawing de `ezdxf`) en objetos que heredan de "APEntity", que luego son transmitidos por los pipes. Este filtro en particular necesita acceder directamente al documento *DXF*. Para representar esto en las configuraciones, se utiliza el input especial *drawing*, al que se le asigna el valor *DRAWING*.

Además de las configuraciones de los pipelines, se utilizaba un archivo de configuración que contenía valores por defecto para todas las configuraciones de los filtros, lo que permitía definir los filtros dentro de un pipeline sin necesidad de definir valores para todos sus parámetros.

4.5. Limitaciones

El prototipo generado tiene algunas limitaciones en cuanto a los archivos que puede procesar y algunas características indeseadas en los resultados los cuales se detallan a continuación.

Cantidad de figuras en el plano En la práctica, no es difícil encontrar planos con varias decenas de miles de figuras. Además, los algoritmos utilizados en los filtros no siempre son óptimos para los datos del plano (por ejemplo, las líneas "grandes" de un plano se devuelven en la mayoría de las consultas que se le hacen a un QuadTree, lo que hace que estas se comparen incluso con figuras lejanas). Al combinar ambas condiciones, se puede llegar a que los tiempos de ejecución se vuelvan elevados cuando la cantidad de segmentos de líneas del plano supera las 4 cifras.

Dicho esto, en las pruebas realizadas por los estudiantes para evaluar el prototipo (capítulo 5), los tiempos de procesamiento estuvieron por debajo de los 20 segundos por plano.

"Limpieza" del plano El prototipo no era capaz de distinguir entre diagramas de planta y otro tipo de diagramas (como fachadas). Esto hacía que todos los diagramas dentro del mismo archivo se procesaran con las mismas reglas.

Una forma de mitigar esto era crear una copia del archivo y borrar los diagramas que no correspondían a planos de planta usando *AutoCAD* u otra herramienta CAD.

Configuración Poder configurar distintos aspectos del pipeline y sus filtros era algo positivo, sin embargo, para cada plano podía ser necesario hacer ajustes a la configuración por defecto para mejorar la precisión. Por ejemplo, la configuración por defecto busca puertas usando arcos de entre 80^{o} y 90^{o}, sin embargo, algunos planos dibujan las puertas con ángulos de 45^{o}.

Dependencia de las puertas La lógica principal para encontrar paredes se basa en la búsqueda de puertas, por lo que si una planta no tiene puertas, no se detecta tampoco ninguna pared. Una mitigación para este caso, es usar solo el filtro de paralelas y luego realizar una clasificación manual de las líneas que no representan paredes.

Figuras repetidas Algunos filtros generan figuras nuevas durante su procesamiento. Por ejemplo, el filtro "ExplodePolylines" toma una figura de tipo Polyline (o LineString en términos de `Shapely`) y genera a partir de esta múltiples figuras de tipo "Line" (que representa un solo segmento de recta). En la etapa final del pipeline (cuando se genera el nuevo archivo *DXF*) pueden guardarse tanto la Polyline original como las Lines generadas por el filtro.

Este comportamiento puede hacer que sea necesaria una limpieza manual luego del procesamiento del plano.

Se puede mitigar configurando el pipeline para que no guarde las figuras originales, seleccionando solo las salidas adecuadas de los filtros (en el ejemplo anterior, se mitiga evitando guardar las figuras de tipo Polyline).

Capítulo 5

Experimentación

La validación del programa se realizó utilizando algunos planos de ejemplos disponibles públicamente en la web, de los cuales se seleccionaron **10** que fueran diferentes entre si y representaran los ejemplos conocidos.

Estos planos de complejidad variable se procesaron con el prototipo, ajustando manualmente los parámetros para mejorar la precisión tanto como fuera posible.

Luego de realizar la clasificación de las figuras de forma automática con el programa se hizo un posprocesamiento manual, reclasificando parte de las figuras. El foco estuvo nuevamente en las paredes. El objetivo fue determinar cuáles y cuántas figuras se detectaron como paredes sin serlo (falsos positivos), y cuántas no se detectaron como paredes cuando lo eran (falsos negativos).

En los apéndices A.2 y A.3 se pueden ver las configuraciones utilizadas para procesar cada plano.

5.1. Preprocesamiento

Previo al procesamiento de cada plano, se realiza un trabajo de "limpieza" del mismo usando *AutoCAD*.

- Se borran las partes que no corresponden a planos de planta, como pueden ser las fachadas, diagramas de instalación eléctrica, referencias, etc. Esto mejora el tiempo de procesamiento.
- Se borran elementos del plano que no se usan con el comando "PURGE" (como layers que están definidas pero no tienen ningún elemento, bloques que no se insertan en ningún lado del dibujo, etc.). Al igual que el caso anterior, ayuda a reducir el tiempo de procesamiento.

- Se solucionan problemas con el comando “AUDIT” (por ejemplo, referencias rotas o fuentes que se usan pero no se incluyen en el archivo). Este paso evita errores de ejecución al momento de leer los archivos *DXF*.

5.2. Posprocesamiento

Luego de procesar cada plano con el programa, se evaluó el resultado “anotando” manualmente las figuras, identificando las paredes detectadas correctamente, los falsos positivos (FP) y los falsos negativos (FN). En la figura 5.1 se puede ver el resultado del posprocesamiento de un plano. Las líneas identificadas como FP y FN se movieron de capa para poder contarlas fácilmente y realizar el análisis de los resultados. En este plano en particular, se encontraron 40 FN, 13 FP y 97 detecciones correctas.

5.3. Validación de resultados

La tabla 5.1 contiene el registro de las pruebas realizada. Para cada uno se tiene el número total de figuras del plano, la cantidad de figuras detectadas como paredes correctamente, la cantidad de figuras detectadas como paredes pero que no lo eran (Falsos Positivos), y la cantidad de figuras que no se detectaron como paredes y lo eran (Falsos Negativos).

Plano	Figuras total	Paredes encontradas	Falsos Positivos	Falsos Negativos
1	295	147	57	0
2	1294	110	13	40
3	9534	217	77	25
4	8644	360	67	34
5	3331	327	127	8
6	3515	588	287	6
7	3365	469	213	10
8	8616	344	139	24
9	2215	636	284	304
10	4208	733	387	35

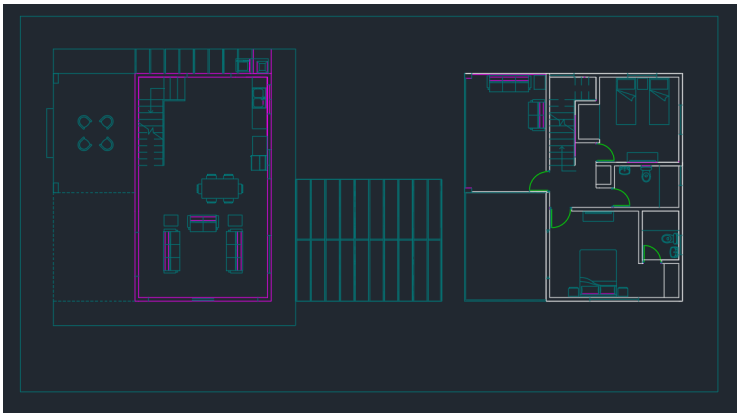
Tabla 5.1: Registro de resultados

En Aprendizaje Automático existen varias métricas que ayudan a entender mejor la performance de los modelos de clasificación. En este proyecto se usaron 4 de estas métricas, que ayudaron a determinar las conclusiones y las oportunidades de mejoras del prototipo.

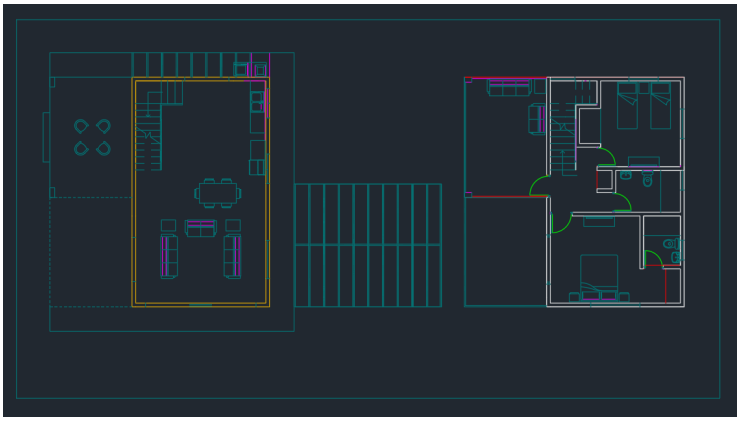
La métrica ***accuracy*** da una noción de qué tan seguido un algoritmo o modelo clasifica correctamente las figuras. Se calcula como el cociente entre los

elementos detectados correctamente y el total de elementos detectados (Zheng, 2015).

$$accuracy = \frac{\#\, detecciones\ correctas}{\#\, elementos\ evaluados}$$



(a) Resultado crudo.



(b) Resultado corregido a mano.

Figura 5.1: Plano luego de ser procesado. Puertas en verde claro. Paredes detectadas en blanco. Paredes erróneamente detectadas en rojo (falsos positivos). Paredes no detectadas en anaranjado (falsos negativos). Líneas paralelas (no paredes) en rosa. El resto de las líneas en verde oscuro.

Una mayor *accuracy* indica que el modelo comete menos errores. Sin embargo, la *accuracy* suele ser alta cuando la cantidad de elementos relevantes es mucho menor a la cantidad total de elementos, por lo que conviene utilizar otras métricas además de ésta.

Precision, ***recall*** y ***F1*** son métricas que hacen foco en los elementos relevantes. Dado que no toman en cuenta la cantidad de elementos totales clasificados, se prestan bien para medir la performance de un modelo mirando un único grupo de elementos relevantes (que en este trabajo son las paredes).

La *precision* se calcula como el ratio entre la cantidad de elementos relevantes clasificados correctamente y la cantidad de elementos clasificados en total.

$$precision = \frac{\#\ detecciones\ correctas}{\#\ detecciones\ hechas}$$

Una *precision* mayor indica una menor cantidad de falsos positivos.

El *recall* se calcula como el ratio entre la cantidad de elementos relevantes clasificados correctamente y la cantidad real de elementos relevantes.

$$recall = \frac{\#\ detecciones\ correctas}{\#\ elementos\ relevantes}$$

Un mayor *recall* indica una menor cantidad de falsos negativos.

Se puede calcular la cantidad de $clasificaciones\ correctas$ de paredes como la resta entre la cantidad de paredes encontradas y la cantidad de falsos positivos. La cantidad real de *elementos relevantes* (paredes) es la suma de las clasificaciones correctas más los falsos negativos (figura 5.2).

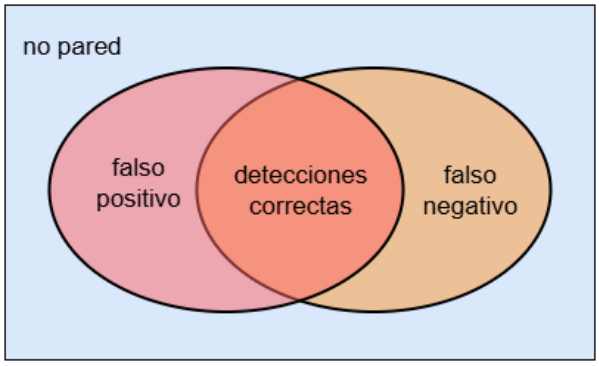


Figura 5.2: El conjunto de la izquierda representa las predicciones del prototipo y el de la derecha las paredes reales. Mientras que el conjunto de la intersección son las predicciones correctas.

Por último, *F1* es la media armónica entre la *precision* y el *recall*. Es un tipo de promedio entre varios números, y se calcula con la siguiente ecuación.

$$F1 = 2\frac{presicion\ .\ recall}{presicion\ +\ recall}$$

Es un promedio que se acerca más al menor de los números, por lo que disminuye tanto cuando la cantidad de falsos positivos aumenta, como cuando la cantidad de falsos negativos aumenta.

En la tabla 5.2 se puede ver el valor de las métricas anteriores para cada plano.

Plano	Accuracy	Precision	Recall	F1
1	0,806	0,612	1,0	0,759
2	0,959	0,882	0,708	0,785
3	0,989	0,645	0,848	0,733
4	0,988	0,814	0,896	0,853
5	0,959	0,612	0,961	0,748
6	0,916	0,512	0,980	0,673
7	0,933	0,546	0,962	0,697
8	0,981	0,596	0,895	0,716
9	0,734	0,553	0,537	0,545
10	0,900	0,472	0,908	0,621

Tabla 5.2: Accuracy, Precision, Recall y F1

Se puede medir la variabilidad de cada métrica calculando su desviación estándar (tabla 5.3). Una desviación estándar mayor indica que el prototipo presenta una mayor variabilidad para esa métrica. En las pruebas realizadas, *recall* fue la métrica con mayor variabilidad, mientras que *accuracy* fue la de menor variabilidad.

Métrica	Promedio	Desviación estándar
Accuracy	0,917	0,084
Precision	0,625	0,128
Recall	0,870	0,144
F1	0,714	0,085

Tabla 5.3: Promedio y desviación estándar

Se puede observar que la métrica con menor promedio es la *precision*, lo que indica un alto ratio de falsos positivos. Esto mismo se observa en la tabla 5.1.

5.3.1. Análisis de la *precision*

En el gráfico de la figura 5.3 se observa una alta variabilidad en la relación *Precicion-Recall*. Esto puede deberse a que las características de cada plano (cada punto del gráfico) varían considerablemente, haciendo que un mismo pipeline no se ajuste bien a todos los casos.

Según se mencionó al inicio de este capítulo, la evaluación se realizó ajustando los distintos parámetros del pipeline buscando un balance entre los FP y los

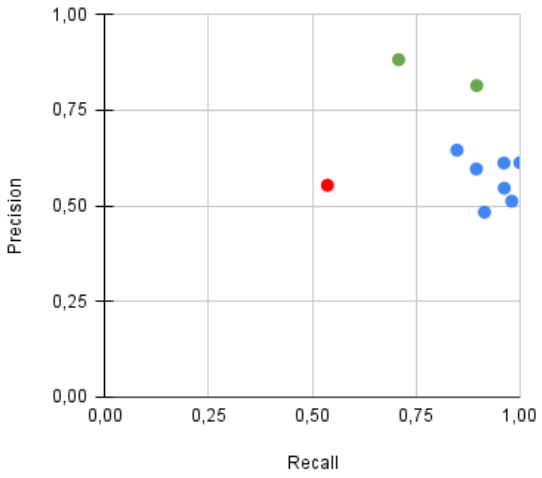


Figura 5.3: *Precision* en relación al *Recall* para cada uno de los 10 planos evaluados.

FN. Los puntos se concentran principalmente en el cuadrante superior derecho. Tres resultados se centran en torno al *recall* 0,9 con una *presicion* que varía entre 0,48 y 0,85, lo que indica que diferentes configuraciones y planos pueden dar la misma proporción de falsos negativos, con un variación significativa en la proporción de falsos positivos. Se observan además resultados con *recall* cercano a 1 y *precision* superior a 0,5, lo que sugiere la posibilidad de alcanzar una alta cobertura con un nivel aceptable de exactitud.

Valores atípicos

En el gráfico de la figura 5.3 destacan tres resultados: el plano 9 (punto rojo) por ser el de menor *precision* y *recall*; los planos 2 y 4 (puntos verdes) por alcanzar *precision* y *recall* superiores a la media.

En la figura 5.4 se muestra una sección del plano con una concentración de falsos negativos. Según se observa en *(a)*, las puertas no son representadas con un arco, por lo tanto no son detectadas por la regla 2.1. Esto hace que las paralelas cercanas a las puertas no sean consideradas como paredes (líneas anaranjadas). Por otro lado, *(b)* muestra una línea detectada erróneamente como pared (rojo) que le permite al filtro que concatena líneas detectar parte de las paredes (líneas blancas). Cabe destacar que la línea roja conecta dos paredes, según se observa en la figura 5.5.

En la figura 5.6 se muestra un caso de falso positivo. El mismo es causado por la combinación de la detección de paralelas (regla 2.2) y una tolerancia de *1 cm* en la concatenación de líneas (regla 2.4):

1. *(c)* la línea roja se detectó como paralela de una de las líneas blancas más alejadas.
2. *(d)* La tolerancia permitió que el filtro que concatena líneas incluyera la línea roja al conjunto de paredes, siendo la misma parte de un mueble empotrado en la pared.

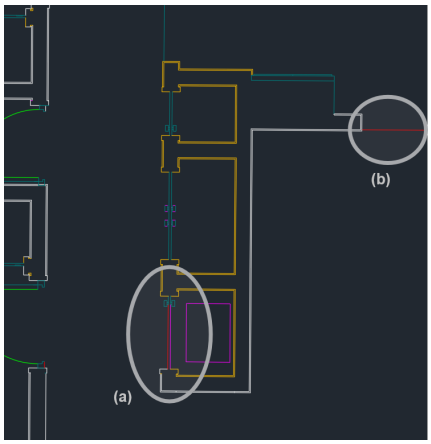


Figura 5.4: Recorte del plano 9. Sección de paredes interiores sin puertas detectadas.

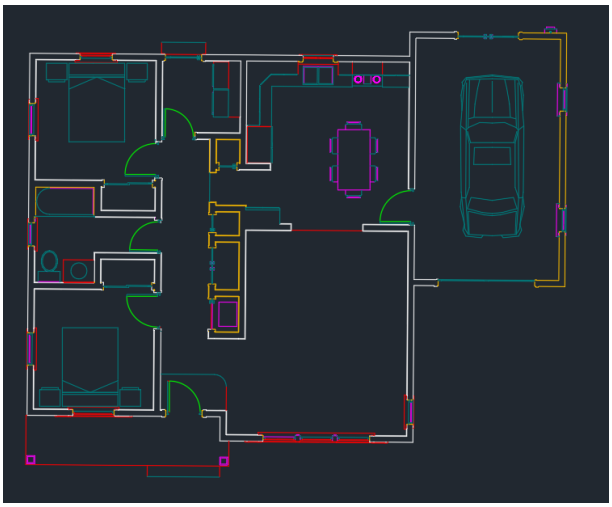


Figura 5.5: Plano 9 procesado y marcado. Puertas en verde claro. Paredes detectadas en blanco. Paredes erróneamente detectadas en rojo (falsos positivos). Paredes no detectadas en anaranjado (falsos negativos). Líneas paralelas (no paredes) en rosa. El resto de las líneas en verde oscuro.

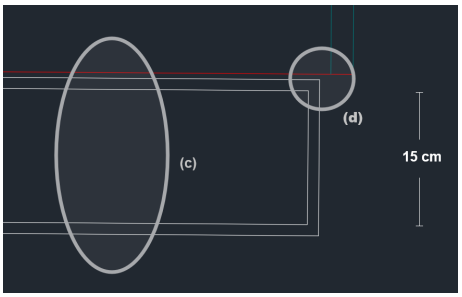


Figura 5.6: Recorte del plano 9. Se observa el extremo de una pared (blanco) con una línea detectada erróneamente como pared (rojo).

En la figura 5.7 se observan 4 líneas numeradas. Las líneas *(2)*, *(3)* y *(4)* son falsos positivos. A continuación se describen los pasos que ejecutó el algoritmo y explican este resultado:

1. Las líneas *(1)*, *(2)* y *(4)* se agregaron al conjunto de líneas paralelas.
2. La línea *(3)* se agregó al conjunto de posibles paredes por estar conectada a 2 líneas paralelas.
3. El filtro que concatena líneas detectó la línea *(1)* como una pared, y continuó detectando las líneas *(3)*, *(2)* y *(4)* (en ese orden). La línea *(4)* se ve punteada por un atributo de la entidad. Dicho atributo no se toma en cuenta durante la detección.

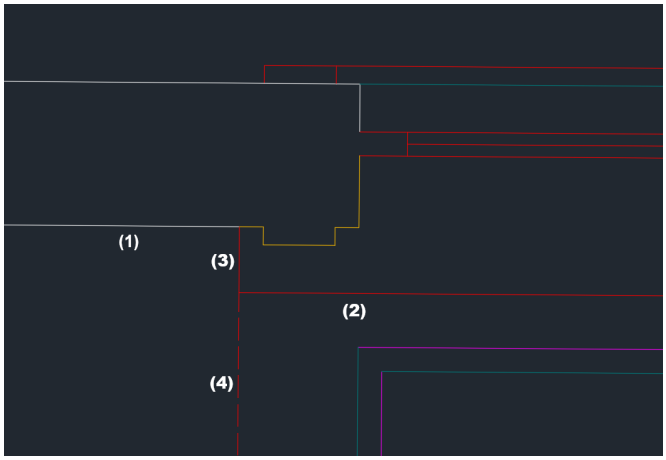


Figura 5.7: Recorte del plano 9 donde se muestra la unión de una pared y una ventana. En blanco se observan la pared de 15 cm de espesor. En rojo, las líneas detectadas como pared pero no lo son (en su mayoría forman parte de la ventana). En anaranjado las líneas que representan paredes no detectadas. En rosa las líneas paralelas, y en verde oscuro el resto de las líneas.

El resto de los falsos positivos (líneas rojas) de la figura 5.7 son resultado de este mismo mecanismo.

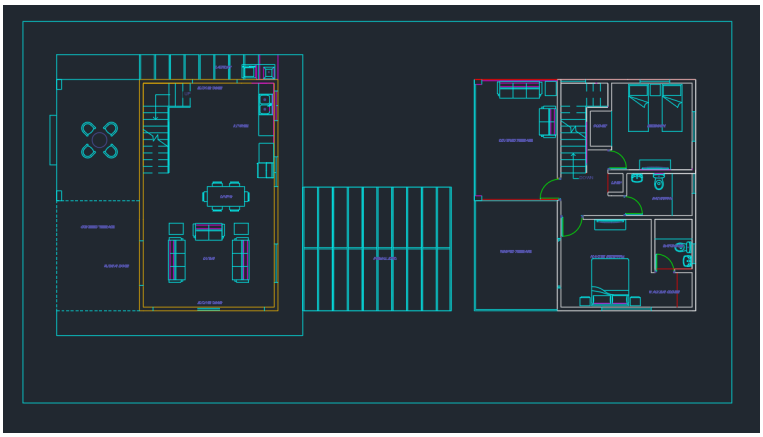


Figura 5.8: Resultado del procesamiento del plano 2. El mismo está formado por dos plantas. A la derecha se observa la planta baja, con pocos falsos positivos (rojo) y falsos negativos (anaranjado). A la izquierda la segunda planta, cuyas paredes no se detectaron.

En la figura 5.9 se observa una sección del plano con solo 1 falso negativo y pocos falsos positivos. Es de destacar en este plano que el grosor de todas las paredes es el mismo, lo que permite ajustar el filtro de paralelas con un rango de distancias de apenas 2 mm, disminuyendo así los falsos positivos.

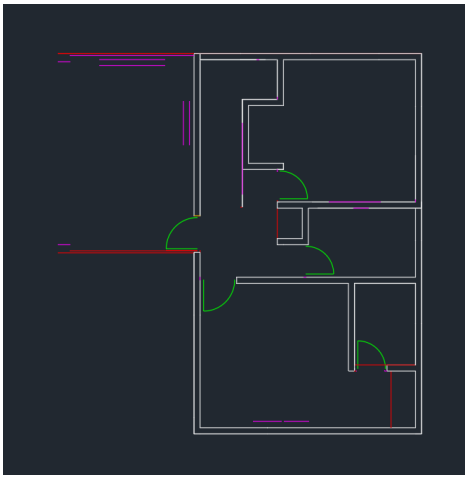


Figura 5.9: Sección del plano 2 con foco en la primera planta. Puertas en verde claro. Paredes detectadas en blanco. Paredes erróneamente detectadas en rojo (falsos positivos). Líneas paralelas (no paredes) en rosa.

La figura 5.10 muestra la segunda planta del plano 2. La misma no tiene puertas, lo que impide el reconocimiento de paredes; sin embargo, dado que esta planta solo tiene 4 paredes, la cantidad de falsos negativos no es elevada, evitando así una disminución significativa en el *recall*.

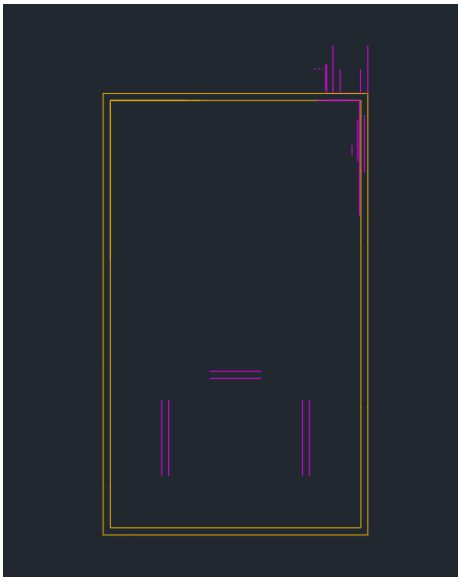


Figura 5.10: Sección del plano 2 con foco en la segunda planta. En anaranjado se muestran las paredes no detectadas. Líneas paralelas (no paredes) en rosa.

La figura 5.11 muestra un plano de planta de dos apartamentos. Destaca de nuevo el hecho de que todas las paredes tienen el mismo grosor. Como se menciona en el caso del plano 2, esto permite configurar el filtro de paralelas con un rango de distancias de 2 mm, disminuyendo así los falsos positivos. Por otro lado, todas las paredes se conectan a alguna puerta, lo que disminuye los falsos negativos, y por ende aumenta el *recall*.

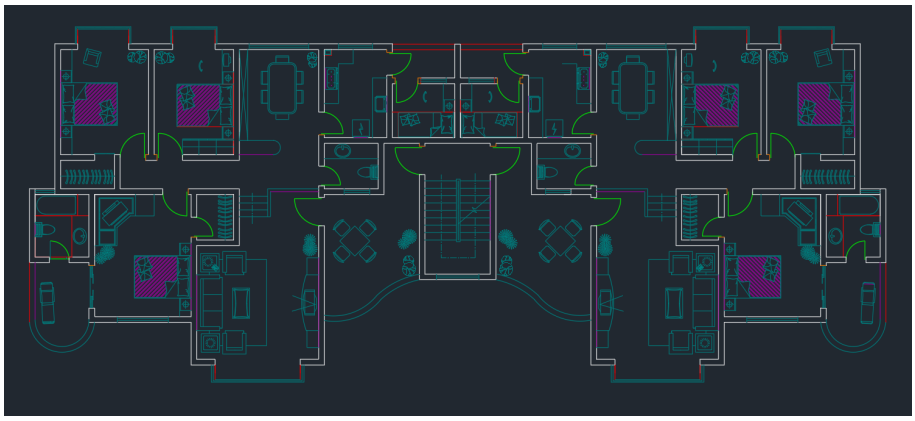


Figura 5.11: Resultado del procesamiento del plano 4. Puertas en verde claro. Paredes detectadas en blanco. Paredes erróneamente detectadas en rojo (falsos positivos). Paredes no detectadas en anaranjado (falsos negativos). Líneas paralelas (no paredes) en rosa. El resto de las líneas en verde oscuro.

Mejora de *métricas*

Al observar el funcionamiento del filtro de lineas paralelas, se tiene que este se basa en la distancia mínima y máxima entre las líneas (*rango de distancia entre paralelas*). El pipeline usado para procesar los planos tiene un único filtro de paralelas, el cual se configuró para identificar todas las rectas paralelas cuyas distancias estaban entre el grosor mínimo y máximo de las paredes del plano. Es decir, si el plano tenía paredes de 15 cm y de 25 cm de grosor, el *rango* configurado fue de 10 cm. En los planos con paredes de un único grosor, el *rango* utilizado fue de 0,2 cm (grosor de la pared ±0,1 cm). En la figura figura 5.12 se observa que la *precision* disminuye a medida que este rango aumenta.

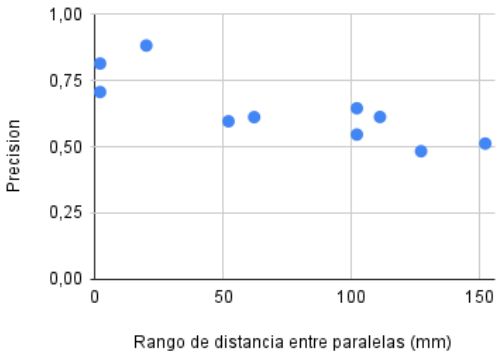


Figura 5.12: Comparativa entre el valor de la precision y el rango de distancia entre paralelas configurado en el pipeline, calculado para los 10 planos evaluado.

En base a las observaciones hechas en la sección anterior, se procesaron nuevamente los 5 planos con menor *precision* utilizando pipelines con más de un filtro de paralelas (uno por cada grosor de pared) y ajustando el rango de distancia entre estas, logrando disminuir la cantidad de falsos positivos y en algunos casos disminuyendo los falsos negativos (tabla 5.4).

Plano	Figuras total	Paredes encontradas	Falsos Positivos	Falsos Negativos
1	295	147	57	0
2	1294	110	13	40
3	9534	217	77	25
4	8644	360	67	34
5	3331	327	127	8
6	3515	527	225	6
7	3365	453	150	8
8	8616	358	24	25
9	2215	505	121	322
10	4208	616	265	34

Tabla 5.4: Resultados mejorados

La tabla 5.5 muestra las métricas actualizadas, las filas sombreadas corresponden a los planos que se volvieron a procesar.

Plano	Accuracy	Precision	Recall	F1
1	0,806	0,612	1,0	0,759
2	0,959	0,882	0,708	0,785
3	0,989	0,645	0,848	0,733
4	0,988	0,814	0,896	0,853
5	0,959	0,612	0,961	0,748
6	0,934	0,573	0,980	0,723
7	0,953	0,668	0,974	0,793
8	0,994	0,932	0,930	0,931
9	0,8	0,760	0,543	0,634
10	0,928	0,569	0,911	0,701

Tabla 5.5: Accuracy, Precision, Recall y F1 mejorados

La gráfica de la tabla 5.6 permite comparar el promedio de las métricas de las primeras pruebas con el promedio luego de ajustar los pipelines. Se observa una mejora importante en la *precision* y *F1*, con una mejora moderada en el *recall*.

En la figura 5.13 se evidencia que es posible mejorar el *recall* y la *precision* de manera simultanea al modificar el pipeline.

Métrica	Promedio		Desviación estándar		Mejora
	inicial	ajustado	inicial	ajustado	
Accuracy	0,919	0,931	0,085	0,071	1,3 %
Precision	0,641	0,707	0,128	0,132	10,3 %
Recall	0,830	0,875	0,258	0,144	5,42 %
F1	0,682	0,766	0,171	0,082	12,3 %

Tabla 5.6: Comparativa de promedio y desviación estándar

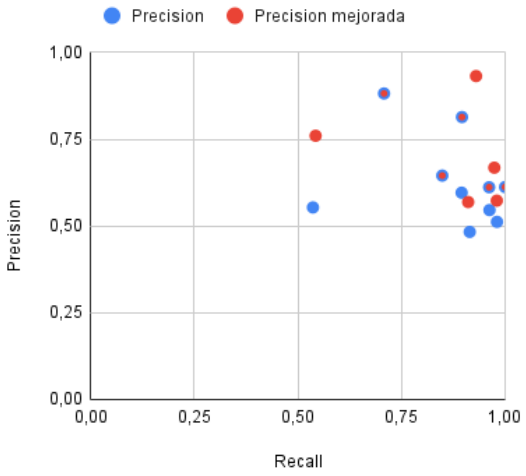


Figura 5.13: En azul, los valores de *recall* y *precision* iniciales. En rojo, los valores de *recall* y *precision* utilizando un pipeline mejorado para los 5 planos con menor *precision* en las pruebas iniciales.

Por último, es importante destacar que estas métricas pueden tener una relevancia distinta desde el punto de vista operativo o funcional del programa. Durante el posprocesamiento se percibió que encontrar manualmente los FP toma menos tiempo que encontrar los FN. Si el posprocesamiento manual con muchos falsos positivos es menos costoso que el posprocesamiento con muchos falsos negativos, es mejor priorizar un mayor *recall*, por más que la *precision* disminuya.

Capítulo 6

Conclusiones

6.1. Estado del arte

Mientras que la identificación de objetos en imágenes ráster ha sido ampliamente estudiada, destacándose las soluciones basadas en aprendizaje automático con resultados altamente precisos, la identificación de objetos o figuras en imágenes vectoriales sigue siendo un área relativamente nueva y poco explorada.

A la falta de literatura científica que aborde este problema se suma la escasez de conjuntos de datos públicos de imágenes vectoriales clasificadas que puedan usarse para entrenar modelos de aprendizaje automático. Más aún si buscamos conjuntos de datos específicos de planos arquitectónicos de planta.

6.2. Prototipo

El uso de reglas para identificar elementos estructurales dentro de un plano es viable, aunque puede requerir combinaciones complejas de reglas para lograr una buena precisión.

La precisión del resultado depende en gran medida de las características del plano, de la estructura del *pipeline* usado para procesarlo y de la configuración de sus parámetros. Dicho de otra forma, no es posible obtener una precisión uniforme para todos los planos usando siempre la misma configuración.

Por otro lado, es posible obtener resultados aceptables al configurar solo el grosor de las paredes, para lo cual puede resultar útil tener algunos *pipelines* predefinidos para seleccionar según la cantidad de grosores distintos que tengan las paredes del plano a procesar.

6.3. Gestión

Se lograron resultados razonables al problema planteado, sin embargo, es innegable que la gestión y el desarrollo del proyecto se vieron afectados negativamente por diversos factores.

La situación personal de los estudiantes generó momentos de inactividad, que llevaron a la desmotivación del equipo, y al posterior abandono del proyecto por parte de uno de los estudiantes, lo que dificultó aún más el progreso.

La priorización de tareas no siempre favoreció el avance del proyecto en la dirección adecuada. Un ejemplo de esto fue el foco que se puso en diseñar un programa altamente configurable, en lugar de dedicar más tiempo a idear y probar reglas que mejoraran la precisión en la clasificación de figuras o a mejorar la interfaz gráfica para facilitar la configuración de los *pipelines*.

Por otro lado, la comunicación con los *stakeholders* no siempre tuvo la fluidez necesaria. Esto derivó en pocas instancias de validación con los ingenieros de Ingenium, lo que dificultó que el proyecto avanzara en la dirección más adecuada.

Los problemas enfrentados y errores cometidos durante el transcurso del proyecto ayudaron a reafirmar aprendizajes muy importantes en los estudiantes, los cuales se pueden resumir en los siguientes puntos.

- Una correcta investigación respecto al estado del arte de un problema paga con creces el tiempo invertido, y ayuda a encauzar correctamente la dirección de un proyecto.
- Una comunicación fluida y frecuente con los stakeholders es crucial para corregir desvíos de forma temprana.
- Los factores humanos y otros imprevistos pueden afectan la velocidad con la que avanza un proyecto. Reconocer riesgos tempranamente puede ayudar a estimar mejor el trabajo y el tiempo necesario para completarlo.
- Es vital no perder de vista el objetivo principal del proyecto para priorizar correctamente las tareas y evitar poner mucho esfuerzo en atacar problemas secundarios.

6.4. Trabajo futuro

Una de las principales dificultades que se presentaron al comienzo de este proyecto fue la falta de planos cuyas figuras estuvieran correctamente clasificadas, que sirvieran tanto de guía como de herramienta de validación. Por lo tanto, la generación de un conjunto de datos con estas características puede aportar un valor significativo a trabajos posteriores que se enfoquen en este problema. El prototipo creado durante este proyecto puede ayudar para generar dicho conjunto de datos, al proporcionar una clasificación inicial que puede mejorarse luego de forma manual.

Por otro lado, si se cuenta con un conjunto de datos adecuado, explorar técnicas de aprendizaje automático para el reconocimiento de imágenes vectoriales parece ser un camino prometedor, no solo por los posibles aportes al problema que se planteó en este proyecto, sino que al ser un campo poco explorado, las técnicas que se desarrollen podrían enriquecer el campo del aprendizaje automático.

Otra opción que podría aportar valor es combinar técnicas de visión por computadora usando imágenes ráster con reglas como las utilizadas en el prototipo desarrollado en este proyecto. Para ello, se podría generar una imagen ráster a partir de los datos vectoriales del plano, esta imagen se procesaría con algún algoritmo o modelo que identifique los elementos del plano y devuelva las zonas de la imagen donde se los detecte. Posteriormente se traducirían las coordenadas de los píxeles a coordenadas del plano, y por último, se utilizaría esta información para filtrar los elementos del plano mediante reglas.

Esta última idea permitiría aprovechar los resultados y herramientas existentes para identificación de objetos en imágenes ráster, mientras que mantendría las coordenadas de la representación vectorial.

Referencias

1spatial. (2024). *dwg-lib.* https://mvnrepository.com/artifact/com.1spatial/dwg-lib. (Accessed: 2024-04-25)

Adobe. (2024a). *Archivos dxf.* https://www.adobe.com/creativecloud/file-types/image/vector/dxf-file.html. (Accessed: 2024-4-27)

Adobe. (2024b). *dwg-file.* https://www.adobe.com/creativecloud/file-types/image/vector/dwg-file.html. (Accessed: 2024-04-25)

Ahmed, S., Liwicki, M., Weber, M., y Dengel, A. (2011). Improved automatic analysis of architectural floor plans. En *2011 international conference on document analysis and recognition.* P.O. Box 3049, 67653 Kaiserslautern,Germany: German Research Center for AI.

Ahmed, S., Liwicki, M., Weber, M., y Dengel, A. (2012). Automatic room detection and room labeling from architectural floor plans. En *2011 international conference on document analysis and recognition.* P.O. Box 3049, 67653 Kaiserslautern,Germany: German Research Center for AI.

Alcequiez, E. (s.f.). *Free cad floor plans.* https://freecadfloorplans.com/. (Accessed: 2025-10-23)

Alexander Wyss and Florian Bruggisser. (2020, 7). *Architectural floor plan analysis.* https://github.com/cansik/architectural-floor-plan. (Accessed: 2024-04-27)

Autodesk, . (2025). *Autocad.* https://www.autodesk.com/mx/products/autocad/overview#:~:text=%C2%BFPara%20qu%C3%A9%20se%20utiliza%20AutoCAD,%2C%20caracter%C3%ADsticas%20de%20documentaci%C3%B3n%2C%20etc. (Accessed: 2025-06-23)

Autodesk. (2013). *Dxf format.* https://images.autodesk.com/adsk/files/autocad_2014_pdf_dxf_reference_enu.pdf. (Accessed: 2024-06-19)

Autodesk. (2024a). *Autocad preguntas frecuentes.* https://latinoamerica.autodesk.com/products/autocad/overview?term=1-YEAR&tab=subscription&plc=ACDIST#faqs. (Accessed: 2024-01-25)

Autodesk. (2024b). *realdwg.* https://aps.autodesk.com/developer/overview/realdwg. (Accessed: 2024-01-25)

Autodesk. (2024c). *Requistos de autocad.* https://www.autodesk.es/support/technical/article/caas/sfdcarticles/sfdcarticles/ESP/System-requirements-for-AutoCAD-2024-including-Specialized-Toolsets.html. (Accessed: 2024-06-01)

Bay, H., Ess, A., Tuytelaars, T., y Van Gool, L. (2008). Speeded-up robust features (surf). *Computer Vision and Image Understanding*, *110*(3), 346-359. Descargado de https://www.sciencedirect.com/science/article/pii/S1077314207001555 (Similarity Matching in Computer Vision and Multimedia) doi: https://doi.org/10.1016/j.cviu.2007.09.014

Cai, M., Huang, Z., Li, Y., Wang, H., y Lee, Y. J. (2023). *Leveraging large language models for scalable vector graphics-driven image understanding.*

Chen, J., Liu, C., Wu, J., y Furukawa, Y. (2019). Floor-sp: Inverse cad for floorplans by sequential room-wise shortest path. En *The ieee international conference on computer vision (iccv).*

Ching, F. D. K. (2015). *Architectural graphics, 6th ed.* United States of America: JOHN WILEY X SONS, INC.

Comunity. (2024). *Python gui.* https://wiki.python.org/moin/GuiProgramming#Cross-Platform_Frameworks. (Accessed: 2024-06-23)

Cortesi, D. (2024). *Pyinstaller.* https://pyinstaller.org/en/stable/. (Accessed: 2024-07-20)

Csurka, G., Dance, C. R., Fan, L., Willamowski, J., y Bray, C. (2004). *Visual categorization with bags of keypoints.* 6, chemin de Maupertuis, 38240 Meylan, France: Xerox Research Centre Europe.

Dodge, S., Xu, J., y Stenger, B. (2017, 05). Parsing floor plan images. En (p. 358-361). doi: 10.23919/MVA.2017.7986875

Escalera, S., Fornés, A., Pujol, O., Lladós, J., y Radeva, P. (2007). *Multi-class binary object categorization using blurred shape models.* Verlag Berlin: Springer.

Fan, Z., Zhu, L., Li, H., Zhu, S., y Tan, P. (2021, October). Floorplancad: A large-scale cad drawing dataset for panoptic symbol. En *Proceedings of the ieee/cvf international conference on computer vision (iccv).*

Finkel, R. A., y Bentley, J. L. (1974). *Quad trees: A data structure for retrieval on compsite keys.* Descargado de https://rdcu.be/eG9yX

Fitzpatrick, M. (2020). *Create gui applications with python & qt5* (4.ª ed.). Autor.

FreeSympleGUI. (2025). *Freesimplegui.* https://freesimplegui.readthedocs.io/en/latest/. (Accessed: 2025-09-03)

Gamma, E., Helm, R., Johnson, R., y Vlissides, J. M. (1994). *Design patterns: Elements of reusable object-oriented software* (1.ª ed.). Addison-Wesley Professional. Descargado de http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

GNU. (2024). *libredwg.* https://www.gnu.org/software/libredwg/. (Accessed: 2024-01-25)

González de la Cal, J. R., y Blanco de Paz, J. (2017, ene.). La planta, partitura escrita. *REIA - Revista Europea de Investigación en Arquitectura*(07-08). Descargado de https://erevistas.universidadeuropea.com/index.php/reia/article/view/199 doi: 10.64197/REIA.07-08.199

Heart, P. E., Nilsson, N. J., y Raphael, B. (1968). *A formal basis for the heuristic determination of minimum cost paths.* IEEE.

Heras, L.-P., Ahmed, S., Liwicki, M., Valveny, E., y Sánchez, G. (2013, 09). Statistical segmentation and structural recognition for floor plan interpretation: Notation invariant structural element recognition. *International Journal on Document Analysis and Recognition (IJDAR)*, *17*. doi: 10.1007/s10032-013-0215-2

Hibbeler, R. C. (2012). *Análisis estructural* (8.ª ed.). 53519, Naucalpan de Juárez, Estado de México, México: Parson Educacion.

Lecun, Y., Bengio, Y., y Haffner, P. (1998). *Gradient-based learning applied to document recognition.* IEEE. doi: 10.1109/5.726791

Long, J., Shelhamer, E., y Darrell, T. (2015). *Fully convolutional networks for semantic segmentation.* Descargado de https://arxiv.org/abs/1411.4038

Longley, P. A., Goodchild, M. F., Maguire, D. J., y Rhind, D. W. (2005a). *Geographical information systems and science.* Chichester, West Sussex PO19 8SQ, England: John Wiley and Sons, Ltd.

Longley, P. A., Goodchild, M. F., Maguire, D. J., y Rhind, D. W. (2005b). *Geographical information systems and science.* Chichester, West Sussex PO19 8SQ, England: John Wiley and Sons, Ltd.

Manfred Moitzi. (2024). *ezdxf.* https://ezdxf.readthedocs.io/en/stable/. (Accessed: 2024-01-25)

Mozman. (2024). *Dxfgrabber.* https://dxfgrabber.readthedocs.io/en/latest/. (Accessed: 2024-4-27)

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., ... Mian, A. (2024). *A comprehensive overview of large language models.* Descargado de https://arxiv.org/abs/2307.06435

NumFOCUS. (2024). *Geopandas.* https://geopandas.org/en/stable/index.html. (Accessed: 2024-4-27)

Open Design Alliance. (2018). *Open design specification for .dwg files.* https://www.opendesign.com/files/guestdownloads/OpenDesign_Specification_for_.dwg_files.pdf. (Accessed: 2024-06-18)

Open Source Geospatial Foundation. (2024). *Gdal.* https://gdal.org. (Accessed: 2024-4-27)

OpenDesign. (2024). *oda-releases-teigha-431.* https://www.opendesign.com/blog/2017/august/oda-releases-teigha-431. (Accessed: 2024-04-25)

Peters, T. (s.f.). *Timsort.* https://mail.python.org/pipermail/python-dev/2002-July/026837.html. (Accessed:2024-06-09)

Pyproj. (2024). *Pyproj.* https://pyproj4.github.io/pyproj/stable/. (Accessed: 2024-4-27)

Python Software Fundation. (2024). *tkinter — python interface to tcl/tk.* https://docs.python.org/3.9/library/tkinter.html. (Accessed: 2024-06-23)

Python 2.3 changelog. (s.f.). https://docs.python.org/release/2.3/whatsnew/node17.html. (Accessed:2024-06-09)

reclosedev. (2024). *Pyautocad.* https://pypi.org/project/pyautocad/. (Accessed: 2024-4-27)

Ren, S., He, K., Girshick, R., y Sun, J. (2016). *Faster r-cnn: Towards real-time object detection with region proposal networks.* Descargado de https://arxiv.org/abs/1506.01497

Richard H. Clough, S. K. S., Glenn A. Sears. (2008). *Construction project management, a practical guide to field construction management.* Hoboken, New Jersey: John Wiley & Sons, Inc.

Shapely. (2024). *Shapely.* https://pypi.org/project/shapely/. (Accessed: 2024-4-27)

Vollebregt, B. (2024). *Auto py to exe.* https://pypi.org/project/auto-py-to-exe/. (Accessed: 2024-5-8)

Woolf, B., y Hohpe, G. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Livery Place, 35 Livery Street, Birmingham B3 2PB, UK: Addison-Wesley Professional.

ZENG, Z., LI, X., Yu, Y. K., y Fu, C.-W. (2019). Deep floor plan recognition using a multi-task network with room-boundary-guided attention. En *Ieee international conference on computer vision (iccv).*

Zheng, A. (2015). *Evaluating machine learning models* (1.ª ed.). O'Reilly Media, Inc.

Anexo A

Configuraciones

A.1. Configuración por defecto

El siguiente es un conjunto de configuraciones por defecto utilizadas por el prototipo.

Listing A.1: Default configs

```
# Door detection settings #

# DOOR_ARC_TOLERANCE_METERS represents the radius of
# small circles centered at the start, end, and
# center of an arc. Shapes intersecting this circles
# are analysed in search for a rectangle representing
# a door.
DOOR_ARC_TOLERANCE_METERS: 0.04

# The maximum and minimum thickness of the door, or
# the maximum and minumum width of the rectangle
# which represents said door.
DOOR_MAX_THICKNESS_METERS: 0.08
DOOR_MIN_THICKNESS_METERS: 0.02

# The maximum and minimum width of the door, or the
# maximum and minumum lenght of the rectangle which
# represents said door. Also is the maximum and
# minimum radius of the arc of the door.
DOOR_MIN_WIDTH_METERS: 0.5
DOOR_MAX_WIDTH_METERS: 1.5

# Doors are represented with an arc of aproximated
# 90 degrees, but drawings are not that precise.
DOOR_ARC_MIN_ANGLE: 80
```

```
DOOR_ARC_MAX_ANGLE: 100

# When looking for walls, this setting is used to
# draw a circle around the frame of the door.
# Figures intersecting this circles are likely to
# contain walls.
DOOR_FRAME_BUFFER_METERS: 0.20


# Wall detection settings #

# The minimun and maximun thickness of a wall in
# meters.
MIN_WALL_THICKNESS_METERS: 0.14
MAX_WALL_THICKNESS_METERS: 0.35

# The tolerace between line extremes to concatenate
# them
CONCAT_TOLERANCE_METERS: 0.01


# Dense Block Filtering #

# The maximum density to filter blocks, if density
# is higher then this, blocks are discarded. The
# density es calculated as the amount of entities
# over the area of the minimum square containing the
# block.
MAX_DENSITY: 40
# The minimum amount of entities in the block for it
# to be discarded. If the block has less than this
# quantity of entities, it won't be discarded.
MIN_ENTITY_COUNT: 100


# Complementary lines detection #

# When enriching a set of lines with other lines
# whose bouth extremes touche extremes of lines from
# the original set.

# The minimum amount of lines from the original set
# of lines that must be touched for a given
# supplementary line to be added as enrichment.
ENRICHER_MIN_TOUCHES: 1
# The maximum amount of lines from the original set
# of lines that could be touched for a given
# supplementary line to be added as enrichment.
ENRICHER_MAX_TOUCHES: 10
```

```
# Line filtering #

# When filtering lines by lenght: The minumum
# accepted lenght
MIN_LINE_LENGHT_METERS: 0.1
MAX_LINE_LENGHT_METERS: 100


# A default value for the units. 6 means 'meters'.
DEFAULT_UNIT: 6
```

A.2. Pipeline básico

Esta sección muestra la configuración en formato YAML del pipeline básico utilizado para las pruebas que se detallan en el capítulo 5.

Los valores expresados como un `string` en mayúsculas se corresponden con parámetros tomados del archivo de configuraciones por defecto.

Listing A.2: Basic Pipeline

```
name: Basic Pipeline

filters:
  - type: FilterBlocks
    name: blocks
    inputs:
      drawing: DRAWING
    outputs:
      high_density_blocks: high_density_blocks
    config:
      max_density: MAX_DENSITY
      min_entity_count: MIN_ENTITY_COUNT
      UNITS: UNITS

  - type: FilterToAPEntity
    name: ap
    inputs:
      drawing: DRAWING
      high_density_blocks: high_density_blocks
    outputs:
      ap_entities: ap_entities
      unsupported_entities: unsupported_entities

  - type: FilterDoors
    name: doors
    inputs:
      ap_entities: ap_entities
```

```
    outputs:
      door_frame_points: frame_points
      doors: doors
      others: not_doors
    config:
      door_arc_tolerance_meters: DOOR_ARC_TOLERANCE_METERS
      door_max_thickness_meters: DOOR_MAX_THICKNESS_METERS
      door_min_thickness_meters: DOOR_MIN_THICKNESS_METERS
      door_min_width_meters: DOOR_MIN_WIDTH_METERS
      door_max_width_meters: DOOR_MAX_WIDTH_METERS
      door_frame_buffer_meters: DOOR_FRAME_BUFFER_METERS
      door_arc_min_angle: DOOR_ARC_MIN_ANGLE
      door_arc_max_angle: DOOR_ARC_MAX_ANGLE
      UNITS: UNITS

  - type: FilterExplodePolylines
    name: polylines
    inputs:
      ap_entities: ap_entities
    outputs:
      ap_entities: no_polylines

  - type: FilterParallelsQTree
    name: parallels
    inputs:
      ap_entities: no_polylines
    outputs:
      parallels: parallels
      others: no_parallels
    config:
      min_wall_thickness_meters: MIN_WALL_THICKNESS_METERS
      max_wall_thickness_meters: MAX_WALL_THICKNESS_METERS
      UNITS: UNITS

  - type: FilterLinesByLenght
    name: line_lenghts
    inputs:
      ap_entities: parallels
    outputs:
      filtered_lines: adecuated_parallels
      others: non_adecuated_parallels
    config:
      min_lenght_meters: MIN_LINE_LENGHT_METERS
      max_lenght_meters: MAX_LINE_LENGHT_METERS
      UNITS: UNITS

  - type: FilterLineEnricher
    name: line_enricher
    inputs:
      lines: no_polylines
```

```
      lines_to_enrich: adecuated_parallels
    outputs:
      enriched_lines: enriched_parallels
      others: others
    config:
      tolerance_meters: CONCAT_TOLERANCE_METERS
      min_touches: ENRICHER_MIN_TOUCHES
      max_touches: ENRICHER_MAX_TOUCHES
      UNITS: UNITS

  - type: FilterBufferIntersect
    name: buffer_intersect
    inputs:
      ap_entities: adecuated_parallels
      points: frame_points
    outputs:
      intersection: entities_near_doors
      others: entities_far_from_doors
    config:
      UNITS: UNITS
      radius_meters: DOOR_FRAME_BUFFER_METERS

  - type: FilterLineConcatQTree
    name: walls
    inputs:
      lines: entities_near_doors
      lines_to_append: enriched_parallels
    outputs:
      appended_lines: walls
      others: not_walls
    config:
      UNITS: UNITS
      tolerance_meters: CONCAT_TOLERANCE_METERS

outputs:
  - output: doors
    layer: Doors
  - output: walls
    layer: Walls
  - output: parallels
    layer: Parallels Not Walls
  - output: no_polylines
    layer: Other Lines
  - output: ap_entities
    layer: Other Entites
  - output: unsupported_entities
    layer: Unsupported Entities
```

A.3. Cambios en configuración para la experimentación

En esta sección se detallan los cambios de configuración realizados en el pipeline básico (apéndice A.2) para realizar la etapa de validación de prototipo.

Solo se expresan los valores de las configuraciones que se modificaron respecto a la configuración original del *pipeline*.

Listing A.3: Config tunnings

```
---
# Plano 1
- type: FilterParallelsQTree
  config:
    min_wall_thickness_meters: 0.09
    max_wall_thickness_meters: 0.201

---
# Plano 2
- type: FilterParallelsQTree
  config:
    min_wall_thickness_meters: 0.14
    max_wall_thickness_meters: 0.16
- type: FilterLinesByLenght
  config:
    min_lenght_meters: 0.14

---
# Plano 3
- type: FilterParallelsQTree
  config:
    min_lenght_meters: 0.099
    max_wall_thickness_meters: 0.201
- type: FilterLinesByLenght
  config:
      min_lenght_meters: 0.099

---
# Plano 4
- type: FilterParallelsQTree
  config:
    min_lenght_meters: 0.199
    max_wall_thickness_meters: 0.201
- type: FilterLinesByLenght
  config:
      min_lenght_meters: 0.199

---
# Plano 5
```

```yaml
- type: FilterDoors
  config:
      door_min_width_meters: 0.35
- type: FilterParallelsQTree
  config:
    min_lenght_meters: 0.119
    max_wall_thickness_meters: 0.181
- type: FilterLinesByLenght
  config:
      min_lenght_meters: 0.059

---
# Plano 6
- type: FilterParallelsQTree
  config:
      min_lenght_meters: 0.0999
      max_wall_thickness_meters: 0.251
- type: FilterLinesByLenght
  config:
      min_lenght_meters: 0.0999
      max_lenght_meters: 21

---
# Plano 7
- type: FilterParallelsQTree
  config:
      min_lenght_meters: 0.149
      max_wall_thickness_meters: 0.251
- type: FilterLinesByLenght
  config:
      min_lenght_meters: 0.149
      max_lenght_meters: 22

---
# Plano 8
- type: FilterDoors
  config:
      door_min_width_meters: 0.6
      door_max_width_meters: 0.9
- type: FilterParallelsQTree
  config:
      min_lenght_meters: 0.099
      max_wall_thickness_meters: 0.151
- type: FilterLinesByLenght
  config:
      min_lenght_meters: 0.099
      max_lenght_meters: 17

---
# Plano 9
```

```yaml
- type: FilterParallelsQTree
  config:
    min_wall_thickness_meters: 0.149
    max_wall_thickness_meters: 0.151
- type: FilterLinesByLenght
  config:
    min_lenght_meters: 0.049

---
# Plano 10
- type: FilterParallelsQTree
  config:
    min_wall_thickness_meters: 0.124
    max_wall_thickness_meters: 0.251
- type: FilterLinesByLenght
  config:
    min_lenght_meters: 0.099
```

Anexo B

Desarrollo

B.1. Compilación del proyecto

El proyecto se desarrolló usando Windows 11 y Python 3.9.

- ***Instalación de Python***

  Descargar e instalar Python 3.9 desde el sitio oficial `https://www.python.com/downloads/`.

- ***Obtención del código fuente***

  El código fuente de la aplicación se encuentra disponible en el GitLab de la Facultad de Ingeniería.

  `https://gitlab.fing.edu.uy/proy-mati-edu/autplark`

- ***Crear entorno virtual (opcional)***

  - Abrir Power Shell en la ubicación del código fuente
  - Crear un entorno virtual de python

    ```
    > python –m venv venv
    ```

  - Activar el entorno virtual

    ```
    > .\venv\Scripts\Activate.ps1
    ```

- ***Instalar las dependencias del proyecto***

  ```
  > pip install –r requirements.txt
  ```

- ***Crear el ejecutable***

  ```
  > pyinstaller autplark.spec
  ```

El ejecutable se genera en el directorio '<directorio del proyecto>/dist'.

Anexo C

Manual de usuario

C.1. Instrucciones

La siguiente sección indica los pasos a seguir de un usuario final para poder utilizar el programa en su computadora (con Windows 10 en adelante).

Consta de las siguientes etapas:

- Descargar el archivo comprimido autplark.zip.
- Descomprimir el contenido del archivo en una carpeta.
- Dentro de la carpeta veremos los siguientes archivos:

  El archivo de texto *log.txt* donde se almacena un registro de la ejecución del programa; el ejecutable *autplark.exe*; un archivo de configuración llamado *config.yml* el cual se describe en el apéndice A.1; un directorio *pipelines* el cual contiene los distintos procesos que se pueden ejecutar para procesar un plano.

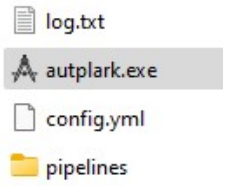


Figura C.1: Directrio del programa

- Una vez ejecutado muestra una ventana con tres etapas para procesar un plano.

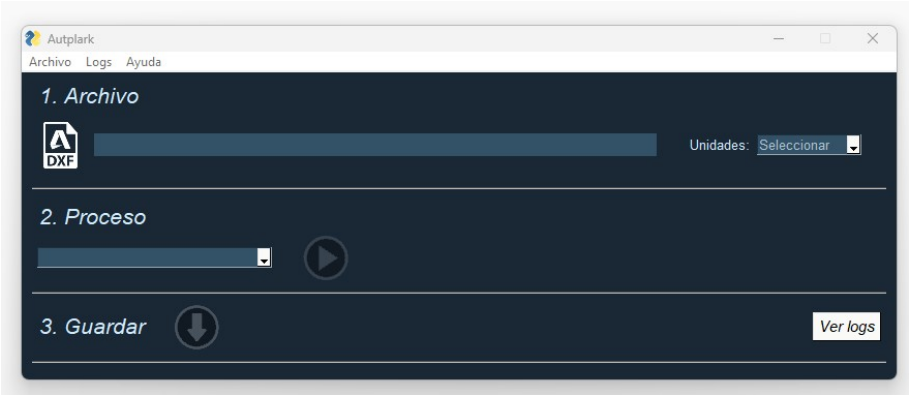


Figura C.2: Pantalla principal del programa

1. Archivo: Se debe seleccionar el archivo del plano que se quiere procesar. El programa intentará obtener las unidades automáticamente, mostrándolas en el campo **Unidades**, pero si no lo logra o las identifica de manera errónea se deberá seleccionar de la lista desplegable. Para seleccionar el archivo se despliega una ventana como se muestra en la figura C.3, donde se puede buscar el archivo y una vez seleccionado se presiona **OK**.

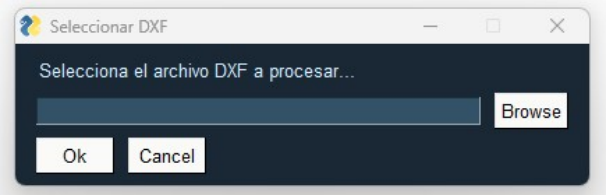


Figura C.3: Pantalla desplegada para buscar archivo

2. Proceso: Se debe elegir un tipo de procesamiento a realizar sobre el plano. Cada procesamiento aplica las reglas geométricas en base a las cuales se clasifican las distintas figuras del plano (por más detalles ver la sección Procesos). Luego de seleccionar el tipo de proceso, se habilita el botón **Play**, al presionarlo el programa comienza a procesar el plano. Dependiendo de la complejidad y tamaño, varia el tiempo de ejecución.

   Mientras se encuentra procesando el botón cambia y mantiene una animación para que se pueda verificar fácilmente que el programa continua ejecutando.

   Una vez finalizado se habilita la siguiente sección para guardar el archivo.

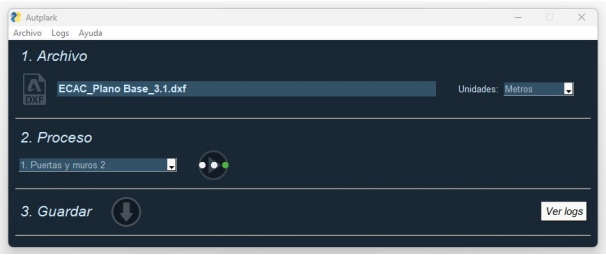


Figura C.4: Procesando archivo

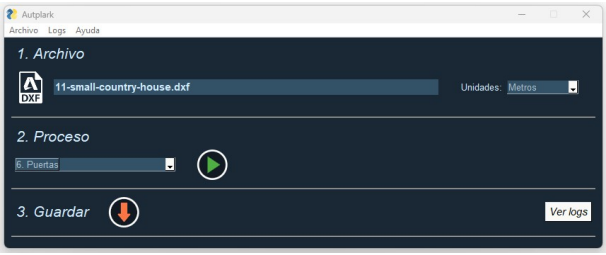


Figura C.5: Habilitado el botón de guardar

3. Guardar: Al presionar el botón **Guardar** se despliega una ventana (figura C.6) donde se elige la ubicación para guardar el arhivo, el nombre con el que se desea guardar y por último se presiona **OK**.

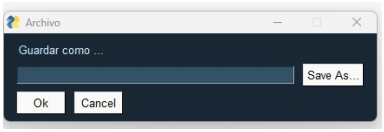


Figura C.6: Pantalla desplegada para guardar el archivo

Una vez finalizado, el programa indica que el archivo se guardó (figura C.7) y se retoma desde el comienzo con un nuevo plano o se realiza alguna modificación, como cambiar el proceso o volver a guardar el resultado con otro nombre.

- Algunas consideraciones a tener en cuenta:
  - Para visualizar los planos se debe usar un programa CAD, de preferencia *AutoCAD*.
  - El programa solo soporta archivos DXF.
  - El resultado del programa es un nuevo archivo DXF con las figuras del plano categorizadas por capas (por ejemplo, las puertas se guardan en la capa “Puertas”).

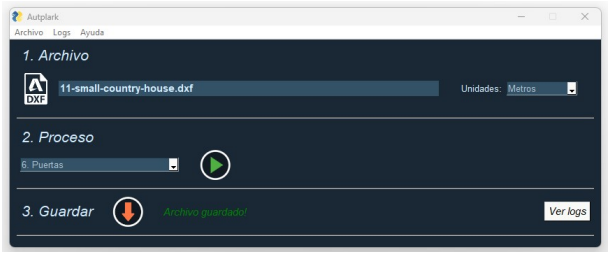


Figura C.7: Archivo guardado correctamente

- Puede que algunas figuras se "pierdan" después del procesamiento, dependiendo del tipo de procesamiento aplicado (por ejemplo, los bloques densos).

C.2. Procesos

C.2.1. Puertas y muros 1, 2, 3, 4 y 5

Estos procesos son muy similares entre sí, variando unicamente el grosor de los muros detectados. Las puertas son detectadas buscando arcos de entre 80^{o} y 100^{o}, cuyo radio sea de 0.5 a 1.5 metros. La siguiente tabla muestra los grosores de muros detectados por cada uno de estos procesamientos

Proceso	Grosor del muro
Puertas y muros 1	entre 0.099 y 0.35 metros
Puertas y muros 2	entre 0.149 y 0.35 metros
Puertas y muros 3	entre 0.199 y 0.35 metros
Puertas y muros 4	entre 0.099 y 0.4 metros
Puertas y muros 5	entre 0.199 y 0.4 metros

Tabla C.1: Diferencias entre los procesos de puerta y muro

C.2.2. Puertas y muros 6

Este proceso es similar a los anteriores, pero intenta separar la búsqueda de paredes en 2 rangos de grosores distintos: muros de entre 0.099 y 0.14 metros de grosor, y muros de entre 0.14 y 0.35 metros de grosor.

Anexo D

Estructura de un archivo DXF

En esta sección, se da una descripción de los principales conceptos presentes en los archivos DXF, sin entrar en el detalle de la codificación de la información.

Los archivos DXF agrupan la información en 7 secciones (Autodesk, 2013), como se aprecia en la figura D.1

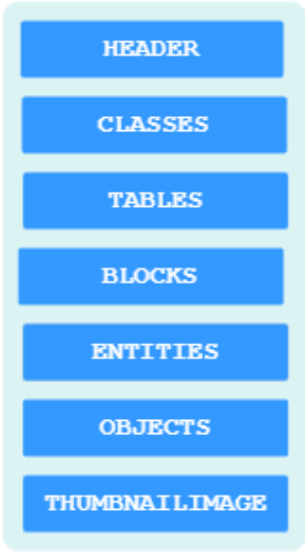


Figura D.1: Secciones de un archivo DXF

HEADER

Contiene información y atributos globales del documento. En particular, se destacan las unidades de medida que se usan en el diagrama, tanto para las distancias como para los ángulos.

CLASSES

Contiene información de las clases definidas por la aplicació,n que luego aparecen instanciadas como bloques, entidades y objetos. En este proyecto no se hace uso de los elementos que aparecen en esta sección del archivo.

TABLES

Contiene varias tablas de información. Algunas de estas no impactan la visualización del diagrama, como por ejemplo, algunos contadores que solo mantienen información estadística utilizada por algunas aplicaciones.

De esta sección se destaca la tabla *LAYER*, donde se almacena información referente a las capas del diagrama donde se agrupan las figuras (o entidades). Además, la tabla *UCS* almacena los sistemas de coordenadas definidos por el usuario, descritos en la sección 2.2.

BLOCKS

Contiene la definición de los bloques que se utilizan en el diagrama, según se describe en la sección 2.2.

ENTITIES

En esta sección están las figuras que componen el *Model layout*, como se describe en la sección 2.2.

OBJECTS

Contiene objetos no gráficos, como pueden ser diccionarios clave-valor, objetos definidos por el usuario, estilos, etc.

THUMBNAILIMAGE

Es una sección opcional que puede contener una vista previa del diagrama.

Anexo E

Tipos de filtros

base_filter

- **Tipo:** Clase abstracta base.
- **Propósito:** Clase padre de la que heredan todos los demás filtros.
- **Uso:** No se utiliza directamente, sino como base para implementar nuevos filtros.

filter_to_ap_entity

- **Tipo:** Filtro inicial.
- **Entradas requeridas:** `drawing`, `high_density_blocks`.
- **Salida:** Colección de entidades APEntity.
- **Propósito:** Convierte un archivo DXF en entidades internas de AutPlark.
- **Parámetros de configuración:**
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Este debe ser el primer filtro en cualquier cadena de procesamiento.

filter_explode_polylines

- **Entradas requeridas:** `ap_entities`.
- **Salida:** Entidades APArc y APLine.
- **Propósito:** Descompone polilíneas complejas en elementos básicos (líneas y arcos).
- **Parámetros de configuración:**

- `UNITS`: Unidades de medida del documento.

- **Uso:** Útil para simplificar geometrías complejas antes de aplicar otros filtros.

filter_parallels

- **Entradas requeridas:** `ap_entities`.
- **Salida:** APLine filtradas.
- **Propósito:** Identifica y selecciona líneas paralelas que están enfrentadas y a cierta distancia.
- **Parámetros de configuración:**
  - `min_wall_thickness_meters`: Grosor mínimo de pared en metros.
  - `max_wall_thickness_meters`: Grosor máximo de pared en metros.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Ideal para detectar paredes paralelas en planos arquitectónicos.

filter_parallels_qtree

- **Entradas requeridas:** `ap_entities`.
- **Salida:** APLine filtradas.
- **Propósito:** Versión optimizada de `filter_parallels` usando QuadTree.
- **Parámetros de configuración:**
  - `min_wall_thickness_meters`: Grosor mínimo de pared en metros.
  - `max_wall_thickness_meters`: Grosor máximo de pared en metros.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Recomendado para planos grandes donde el rendimiento es importante.

filter_line_concat

- **Entradas requeridas:** `lines`, `lines_to_append`.
- **Salida:** APLine concatenadas.
- **Propósito:** Une líneas consecutivas que forman una línea continua.
- **Parámetros de configuración:**
  - `tolerance_meters`: Tolerancia en metros para considerar líneas como consecutivas.

- `UNITS`: Unidades de medida del documento.

- **Uso:** Útil para simplificar líneas fragmentadas.

filter_line_concat_qtree

- **Entradas requeridas:** `lines`, `lines_to_append`.
- **Salida:** APLine concatenadas.
- **Propósito:** Versión optimizada de `filter_line_concat` usando QuadTree.
- **Parámetros de configuración:**
  - `tolerance_meters`: Tolerancia en metros para considerar líneas como consecutivas.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Recomendado para planos grandes.

filter_line_enricher

- **Entradas requeridas:** `lines`, `lines_to_enrich`.
- **Salida:** APLine enriquecidas.
- **Propósito:** Añade información adicional a las líneas (propiedades, metadatos).
- **Parámetros de configuración:**
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Prepara las líneas para análisis más avanzados.

filter_lines_by_length

- **Entradas requeridas:** `ap_entities`.
- **Salida:** APLine filtradas.
- **Propósito:** Filtra líneas por longitud mínima o máxima.
- **Parámetros de configuración:**
  - `min_lenght_meters`: Longitud mínima en metros.
  - `max_lenght_meters`: Longitud máxima en metros.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Elimina líneas muy cortas (ruido) o muy largas.

filter_doors

- **Entradas requeridas:** `ap_entities`.
- **Salida:** Entidades de puertas identificadas.
- **Propósito:** Detecta y marca puertas en el plano.
- **Parámetros de configuración:**
  - `door_arc_tolerance_meters`: Tolerancia en metros para detectar arcos de puertas.
  - `door_max_thickness_meters`: Grosor máximo de puerta en metros.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Identifica aberturas que representan puertas en la arquitectura.

filter_blocks

- **Entradas requeridas:** `drawing`.
- **Salida:** Lista de nombres de bloques de alta densidad.
- **Propósito:** Identifica y descarta bloques que contienen muchas entidades en un área pequeña.
- **Parámetros de configuración:**
  - `max_density`: Densidad máxima de entidades por unidad de área.
  - `min_entity_count`: Cantidad mínima de entidades para considerar un bloque.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Filtra bloques densos que no representan estructuras relevantes del edificio.

filter_buffer_intersect

- **Entradas requeridas:** `ap_entities`, `points`.
- **Salida:** Entidades que intersectan con los buffers de los puntos.
- **Propósito:** Identifica entidades que están dentro de cierta distancia de puntos específicos.
- **Parámetros de configuración:**
  - `radius_meters`: Radio del buffer en metros alrededor de cada punto.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Detecta entidades cercanas a puntos de interés específicos.

filter_union

- **Entradas requeridas:** `a`, `b`.
- **Salida:** Conjunto unificado (`a_union_b`).
- **Propósito:** Combina resultados de múltiples filtros.
- **Parámetros de configuración:** No requiere parámetros específicos.
- **Uso:** Útil para fusionar resultados de diferentes ramas de procesamiento.

filter_to_dxf

- **Entradas requeridas:** Variables (se configuran dinámicamente con `add_required_input`).
- **Salida:** Archivo DXF.
- **Propósito:** Convierte entidades internas de vuelta a formato DXF.
- **Parámetros de configuración:**
  - `file_name`: Nombre del archivo DXF resultante.
  - `callback`: Función de callback para notificaciones.
  - `UNITS`: Unidades de medida del documento.
- **Uso:** Exporta los resultados procesados a un archivo DXF estándar.