Universidad Zaragoza
1542

Trabajo Fin de Grado

Reorganización y mejoras de una plataforma multirrobot

Reorganization and improvements of a multirobot platform

Autora

Paloma Balmori Elósegui

Directores

Cristian Mahulea
Joaquín Ezpeleta Mateo

Escuela de Ingeniería y Arquitectura

2021-2022

REORGANIZACIÓN Y MEJORAS DE UNA PLATAFORMA MULTIRROBOT

RESUMEN

Este proyecto se enmarca en un proceso de rediseño de una plataforma multirrobot, perteneciente a la Universidad de Zaragoza, con el fin de hacerlo más general y adaptable. La plataforma se encuentra en el Departamento de Informática e Ingeniería de Sistemas.

Entre las posibles mejoras a realizar, este trabajo se ha centrado en implementar una planificación de trayectorias basada en una descomposición en celdas cuadradas.

Además, se han desarrollado métodos que permiten simular la navegación de los robots desde sus puntos de partida hasta sus correspondientes destinos mientras que son supervisados para evitar posibles colisiones.

El lenguaje de programación utilizado en este proyecto ha sido C++. Asimismo, se ha hecho uso de las librerías de OpenCV y Boost C++ para el procesamiento de imágenes y la creación de grafos respectivamente.

ABSTRACT

This project is framed in a redesign process of a multirobot platform, belonging to Universidad de Zaragoza, with the purpose of making it more general and adaptable. The platform is situated in the computing and system engineer department.

Among the possible improvements to be made, this work has focused on implementing a trajectory planning based on a decomposition in square cells.

Furthermore, some methods have been developed that allow simulating the navigation of robots from their starting points to their corresponding targets while they are supervised to avoid possible collisions.

C++ has been the programming language used in this project. Additionally, OpenCV and Boost C++ libraries have been used for image processing and graph creation, respectively.

Contenido

Capítulo 1

1. Introducción

1.1 Motivación y contexto

La navegación de los robots móviles es una de las problemáticas actuales en el mundo de la robótica. Se encuentra con problemas como adaptarse al entorno, saber coexistir con otros robots en el mismo entorno y realizar las tareas de manera eficiente, entre otros. Además, el mundo de la robótica está en constante cambio y avance, por ello es muy importante que el *software* que controla los robots esté preparado para ser modificado. Para lograr avances se necesita un *software* rápido de interpretar y que, en el caso de querer añadir alguna mejoría, no sea necesario destruir código alguno.

El departamento de Informática e Ingeniería de Sistemas de esta universidad tiene distintas líneas de investigación en el ámbito de la robótica. Este trabajo se ha llevado a cabo dentro de este departamento, en un laboratorio de la universidad y con una plataforma de bajo coste.

1.2 Objetivos

Este proyecto se enmarca en un proceso de rediseño del sistema con el fin de hacerlo más general y adaptable.

En concreto, este trabajo se centra en las partes del sistema dedicadas a la planificación de las trayectorias que deben seguir los robots cuando se mueven conjuntamente en la plataforma, basadas en una descomposición del escenario en celas, y en la simulación de la evolución del movimiento de acuerdo a la planificación establecida y al control requerido para evitar colisiones.

En concreto, los objetivos propuestos son los siguientes:

- Discretización del espacio mediante descomposición en celdas.
- Cálculo de trayectorias a partir de la discretización anterior.
- Control de navegación para evitar colisiones.
- Simulaciones y conclusiones

1.3 Alcance y Planificación

Se han creado distintas clases para la discretización del escenario mediante una descomposición en celdas cuadradas. Posteriormente, se hace uso de la librería *The Boost Graph Library* (BGL) [1] para generar un grafo, a partir de la discretización anterior, que permita calcular las trayectorias más cortas entre los robots y sus destinos utilizando el algoritmo de Dijkstra. Para una navegación segura se hace uso del algoritmo del banquero que evita las posibles colisiones. Por último, se han simulado distintas situaciones para evaluar el funcionamiento del programa.

La planificación llevada para realizar este trabajo ha sido la siguiente:

- Familiarizarse con el *hardware* y *software* de la plataforma multirrobot.
- Diseñar las clases necesarias para la planificación de trayectorias mediante la descomposición de celdas.
- Investigar y estudiar librerías que permitan realizar el diseño pensado.
- Implementar las nuevas clases.
- Adaptar el algoritmo del banquero a la nueva planificación.
- Simular distintas situaciones y sacar conclusiones.

1.4 Contenidos

El contenido que se encuentra en el siguiente trabajo se resume en este apartado:

- **Capítulo 2: Estado del Arte.** Explicación de conceptos que han sido utilizados durante el trabajo.
- **Capítulo 3: Reorganización y mejoras de la plataforma multirrobot.** Presentación y explicación de los cambios hechos al programa base.
- **Capítulo 4: Toma de decisiones.** Descripción de las decisiones tomadas respecto al diseño y a la implementación del *software*.
- **Capítulo 5: Conclusiones y líneas futuras.** Resumen de las conclusiones obtenidas y el planteamiento de posibles desarrollos futuros.

Capítulo 2

2. Tecnologías utilizadas

2.1 Programación orientada a objetos

El concepto de la programación orientada a objetos (POO) [2] pretende organizar los programas representando los objetos de la vida real. Para entender la programación orientada a objetos se debe conocer la terminología:

- **Clase:** Una clase es una estructura que define unas características y que trabaja con unas operaciones.

- **Objeto:** Un objeto de una clase es un tipo de dato creado en tiempo de ejecución con las características definidas en la clase.

- **Atributos:** Son las características que son comunes a los objetos de una misma clase. Cada objeto puede tener asignados distintos valores en sus atributos.

- **Métodos:** Son las operaciones que un objeto puede efectuar. Se encuentran creadas dentro de una clase.

Entre las ventajas de la programación orientada a objetos se encuentra la flexibilidad del código. La creación de clases permite a otros usuarios entender el programa y hacer modificaciones de una manera más sencilla y rápida. Otra de las ventajas es la reutilización de código cuando se necesitan varios objetos de la misma clase.

Además, las clases se pueden relacionar entre sí mediante la **herencia** compartiendo así su estructura interna. [3] De esta forma los objetos heredaran las propiedades de todas las clases a las que pertenecen. Esta manera de relacionarse sirve para reducir código cuando hay clases que contienen varios atributos y métodos iguales. Un ejemplo de herencia entre clases se representa en la figura 1.

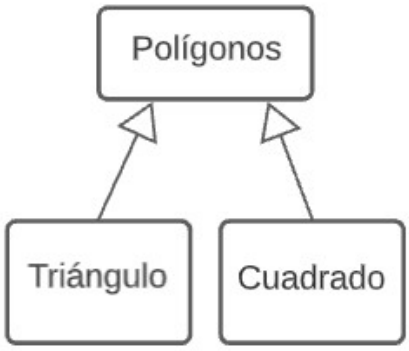


Figura 1. Herencia en POO.

Las clases suelen tener un método llamado *constructor* y *destructor.* Ambos métodos se invocan automáticamente, el *constructor* al crear un objeto de la clase y el destructor al dejar de utilizarlo. Al *constructor* se le pueden introducir datos que se utilizaran para inicializar el objeto [3].

Una manera de representar el sistema de programación orientado a objetos es mediante el Lenguaje de Modelado Unificado (UML). Se utilizará esta representación durante el presente documento para explicar las clases utilizadas y creadas en el *software* de la plataforma. [4]

La representación de las clases incluye el nombre, la lista de atributos y la lista de métodos tal y como se indica en la figura 2.

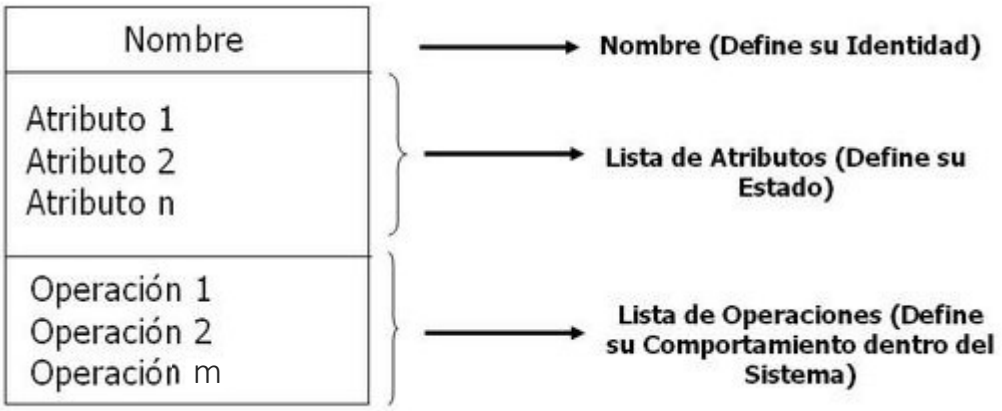


Figura 2. Representación gráfica de una Clase en UML. [19]

Por otro lado, la relación entre clases y objetos se representan con distintos tipos de flechas como se muestra en la figura 3.

- **Uso:** Es la relación entre clases en las que una clase utiliza un objeto de otra clase en algunos de sus métodos.
  Ejemplo: Coche hace uso de la gasolinera para repostar.

- **Agregación:** es la relación en la cual un objeto forma parte de otro.
  Ejemplo: Jugador forma parte de un equipo.

- **Herencia:** representa la relación herencia de POO.

Figura 3. Representación gráfica de las relaciones entre clases y objetos en UML. [3]

2.2 Plataforma multirrobot

El trabajo se realiza en una plataforma multirrobot situada en el laboratorio L0.5 a del edificio Ada Byron en la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza.

En este apartado se explicará tanto el *hardware* de la plataforma como el *software* desarrollado por anteriores estudiantes desde el que se ha comenzado este trabajo.

2.2.1 Hardware

La plataforma multirrobot se compone de robots, regiones de interés, una cámara y un escenario donde además se sitúan los marcadores y los rectángulos que se necesitan para la identificación de la plataforma.

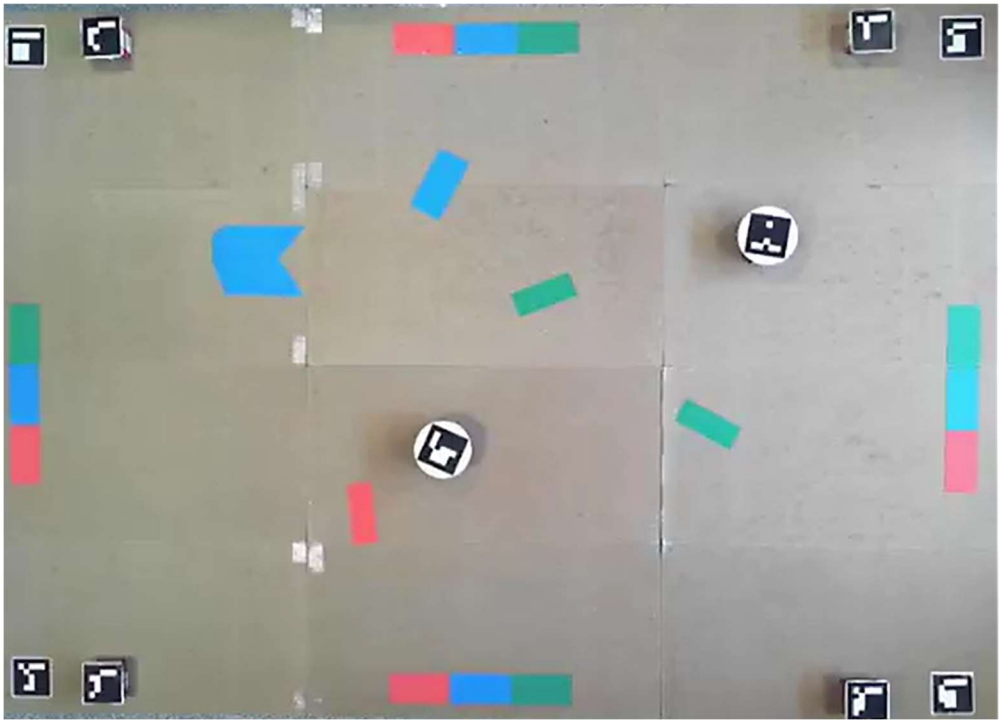


Figura 4. Plataforma multirrobot de bajo coste.

La plataforma tiene unas dimensiones de 3,32 metros de largo y 2,34 metros de ancho. Contiene 8 marcadores distribuidos de a dos por las esquinas, 4 a ras del suelo y 4 elevados a la altura de los robots, que permiten la localización de los robots. Por otro lado, en los 4 lados de la plataforma se encuentran unos rectángulos de tres colores diferentes, rojo, azul y verde, que permiten la identificación de las regiones de interés. Estas regiones de interés son del mismo color que alguno de los rectángulos. Por último, están situados los dos robots con un marcador situado en su cara superior para su posible identificación.

Los robots utilizados funcionan con la placa "DFRobot RoMeo A11 In One Controller V2.2" que se comporta como un Arduino Leonardo. Para la comunicación con los robots desde el PC se utilizan unos dispositivos XBee, pequeños módulos de radio frecuencia, que transmiten y reciben datos a través de señales de radio [5].

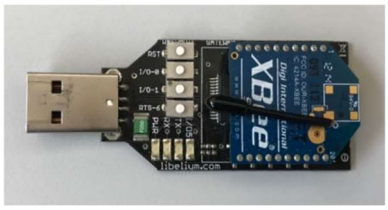


Figura 5. Dispositivo XBee para la comunicación PC-Robot.

La cámara utilizada es Microsoft LifeCam Studio 1080p HD que presenta 3 resoluciones, de las cuales se utiliza la de mayor resolución, 1920x1980 píxeles, para una mayor precisión en la detección del robot [6].

Figura 6. Microsoft LifeCam Studio 1080p. [20]

2.2.2 Software

El *software* utilizado antes de este proyecto estaba implementado por Daniel Roche Garcia [6], y era una adaptación y reorganización de parte del trabajo de fin de máster de Jose.B García Barreto [5]. En este *software* se detectaban tanto los marcadores y los rectángulos, como los robots y las regiones de interés. Esta detección se realizaba utilizando las librerías *OpenCV* [7] y *ArUco* [8] [9]. La primera librería era utilizada para la detección de los marcadores y de las regiones de interés, mientras que la segunda era utilizada para la estimación de la posición de los robots.

El *software* desde el que se partió en este proyecto contenía 6 clases que se relacionaban como se muestra la figura 7. [6] Los atributos de las clases y una explicación más exhaustiva de éstas se encuentran en el anexo 1 de este documento.

- Robot: representa al robot físico real utilizado en la plataforma.

- Camera: es una abstracción del sistema de visión de la plataforma.

- ROI (Region of Interest): modela las regiones de interés. Depende de los colores y lo que el usuario indique, las regiones de interés pueden ser obstáculos, puntos por los que el robot debe pasar, etc. Para este proyecto se ha supuesto que todas las regiones de interés son obstáculos.

- scene: es la representación de la plataforma en su totalidad. En ella se detectan los marcadores, los robots, las regiones de interés... *Robot*, *Camera* y *ROI* son agregaciones de la clase *scene*.

Las clases de *PlanningPath* y *Controller* han sido totalmente redefinidas debido al nuevo diseño que ha adquirido el *software* de la plataforma y que se explica en el apartado 3 de este documento.

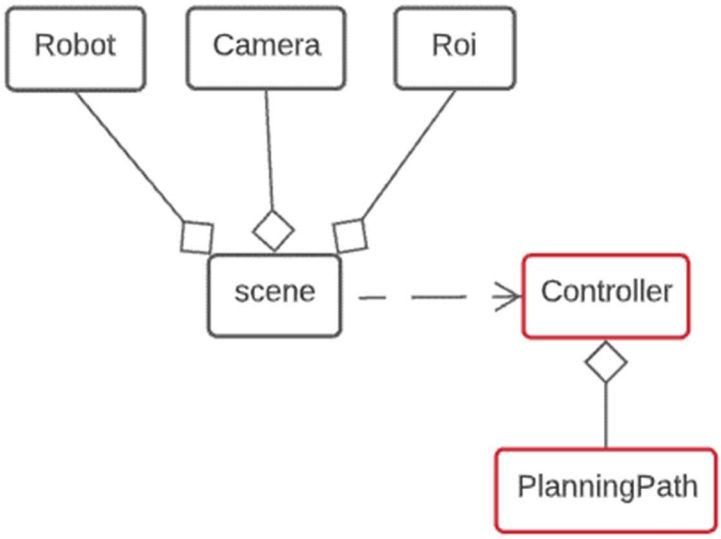


Figura 7. Diseño del software antes de este trabajo. Representación gráfica en UML.

Además, está implementado el uso del archivo JSON (JavaScript Object Notation), para introducir la información necesaria. Un archivo JSON es un formato de texto utilizado para la transferencia de datos compuesto de un conjunto de campos que están asociados con un identificador [10]. Tanto el contenido de este archivo como una explicación más detallada del funcionamiento de un archivo JSON se encuentran en el anexo I.

2.3 Planificación de trayectorias

Para que el robot alcance su objetivo sin colisionar con las regiones de interés consideradas obstáculos, se debe crear una planificación de trayectorias. El área de la planificación de trayectorias se puede dividir en dos grandes categorías: los algoritmos tradicionales de planificación y los algoritmos basados en un muestreo aleatorio. Los algoritmos tradicionales de planificación (algoritmos exactos o combinacionales) discretizan el escenario para encontrar la mejor ruta para el robot, mientras que los algoritmos basados en un muestreo aleatorio buscan la mejor ruta mediante un muestreo pequeño del escenario [5].

2.3.1 Descomposición en celdas

La descomposición en celdas es el algoritmo tradicional de planificación que ha sido utilizado en este trabajo. La descomposición en celdas discretiza el escenario descomponiéndolo en regiones, a las que llamamos celdas. Estas regiones pueden ser de distintas formas dando paso a una descomposición triangular, rectangular, trapezoidal... [11] Una representación de una descomposición en celdas cuadradas se muestra en la figura 8.

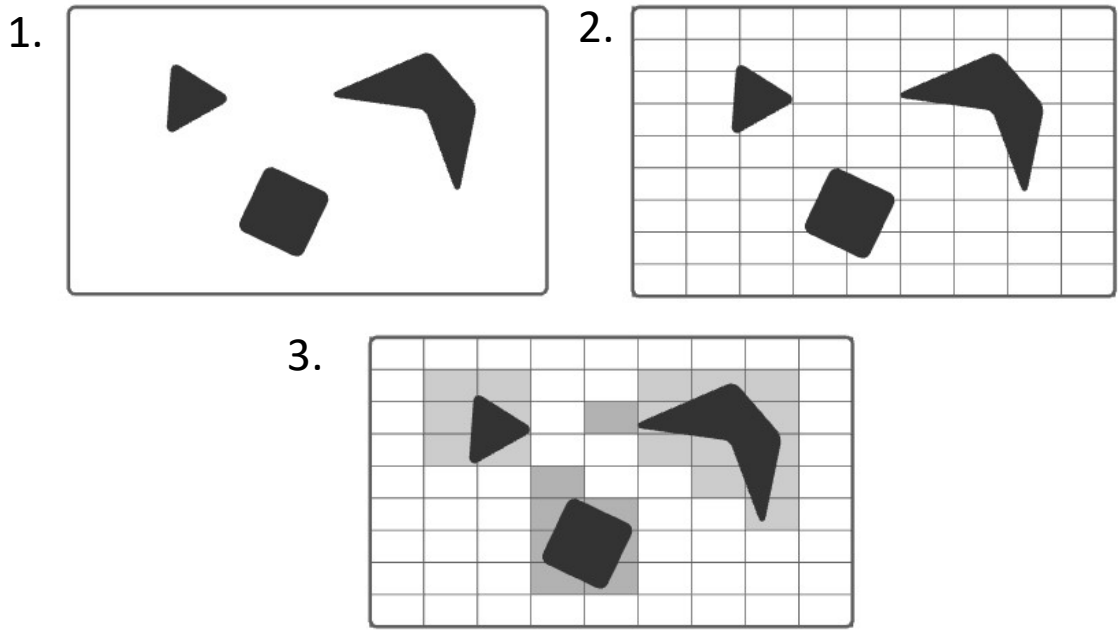


Figura 8. Descomposición en celdas cuadradas. 1. El escenario el cual se quiere discretizar. 2. El escenario se descompone en regiones cuadradas. 3. Se detectan aquellas regiones que están en contacto con alguna región de interés (gris).

2.3.2 Grafos

La Real Academia Española (RAE) define un grafo como: "Diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio" [12]. Están compuestos por objetos llamados vértices y arcos que representan la conexión entre los vértices, tal y como se ve en la Figura 9.

Además, existen distintas maneras de almacenar grafos dependiendo de las características del grafo y del algoritmo que se utiliza para su manipulación. En la figura 9 se muestra un grafo almacenado en una estructura de lista llamada lista de adyacencia. Esta estructura es de dos dimensiones donde la primera dimensión se corresponde con una lista de los vértices que almacenan a su vez en otra lista los vértices adyacentes a ellos [1] [13] .

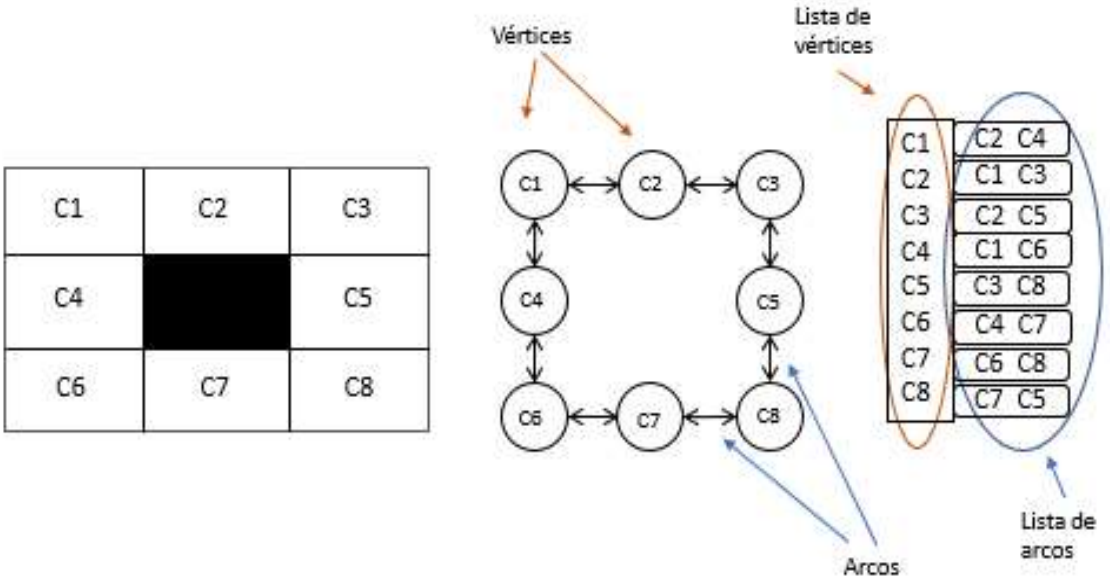


Figura 9. Representación de un grafo dirigido y de la lista de adyacencia de él.

2.3.3 Dijkstra

Dijkstra, también llamado algoritmo de caminos cortos, es un algoritmo de búsqueda que desarrolló Edsger Dijkstra en 1956 [14]. Dicho algoritmo consiste en, a partir de un grafo cuyos arcos tienen pesos de valor positivo, determinar el camino más corto entre un vértice y el resto de los vértices del grafo. El camino cuya suma de los pesos de todos los arcos sea menor, será el camino elegido por el algoritmo [15].

2.3.4 Algoritmo del banquero

El algoritmo del banquero se utiliza en sistemas de procesos que comparten recursos conservativos, y consiste en estudiar con anticipación los recursos que se utilizarán en todos los procesos para conocer si un estado es seguro. El sistema se encuentra en un estado seguro si es posible realizar los procesos en un orden en el cual todas las peticiones de recursos puedan ser concedidas [16] [17].

En el caso de una plataforma multirrobot, dichos procesos son la trayectoria de cada robot desde su punto de partida a su punto de destino, y los recursos son los espacios en los cuales se encuentra dividido el escenario.

Capítulo 3

3. Reorganización y mejoras de la plataforma multirrobot

3.1 Visión general

En este apartado se presentan las clases utilizadas en este proyecto y su relación entre ellas mediante la representación en UML, figura 10. Las clases en color verde son las creadas con anterioridad a este proyecto y que se encuentran explicadas en el apartado 2.2 y el anexo I de este documento.

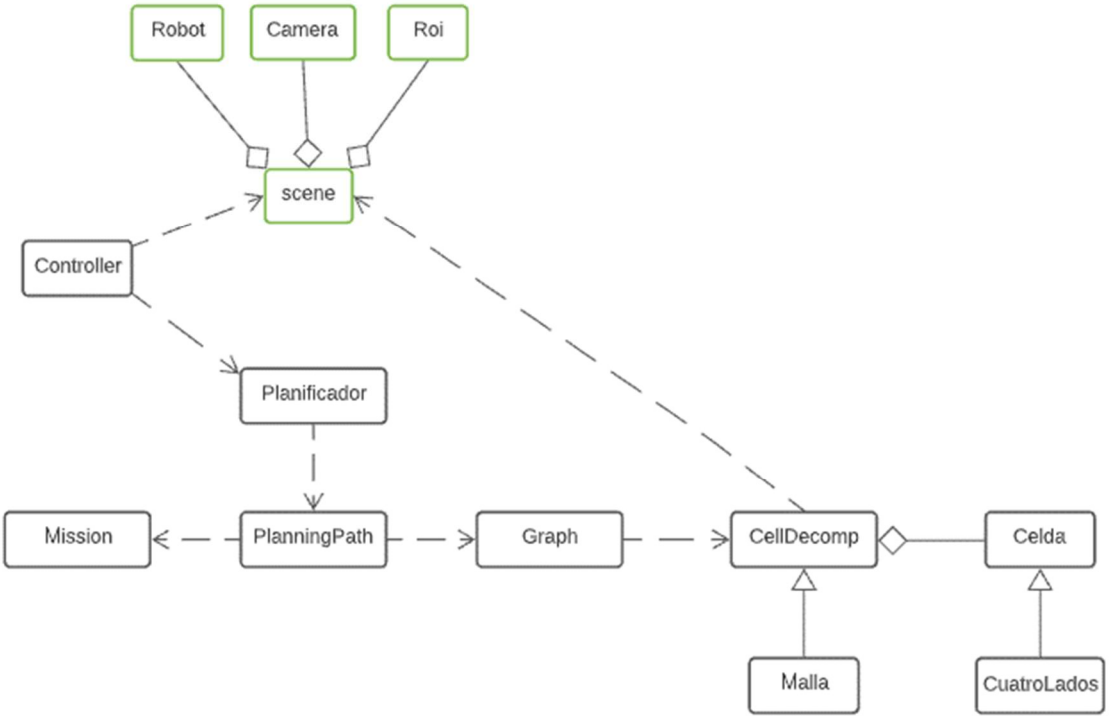


Figura 10. Representación de la relación entre las clases de este proyecto.

En los siguientes apartados se explicarán las características y funciones de las clases creadas en este proyecto. En el anexo II se encuentra una biblioteca con los atributos y métodos de estas clases.

3.2 Clase Mission

La misión de los robots se encuentra almacenada en los atributos de un objeto de la clase Mission. El usuario introduce las coordenadas del destino al que quiere que lleguen los robots a través del archivo JSON. En el Anexo II se encuentra una explicación de los métodos y atributos de esta clase.

3.3 Descomposición en celdas

Para implementar el algoritmo de descomposición en celdas se han creado diferentes clases. La clase *CellDecomp* que representa el conjunto de celdas en el cual se ha discretizado el escenario; la clase *Malla* que guarda las características de una descomposición en celdas cuadradas; la clase *Celda* que representa cada una de estas regiones y la clase *CuatroLados* que guarda las características de los polígonos regulares de 4 lados tales como el cuadrado y el rectángulo. Estas 4 clases mantienen una relación entre ellas:

- CellDecomp y Malla mantienen una relación de herencia.
- CellDecomp y Celda mantienen una relación de agregación donde CellDecomp está compuesta por muchas celdas.
- Celda y CuatroLados mantienen una relación de herencia.

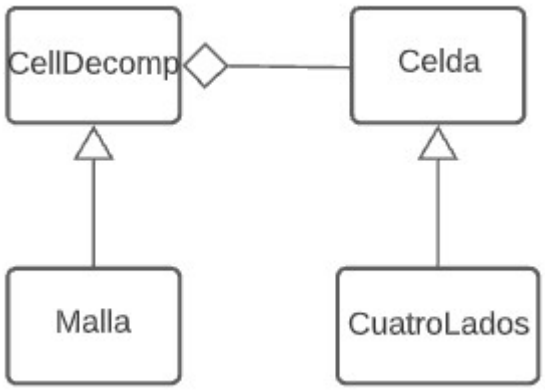


Figura 11. Relación entre las clases para el algoritmo de descomposición en celdas. Representación en UML.

A continuación, se explicará el diseño de las clases *CellDecomp*, *Malla, Celda* y *CuatroLados*. Para entender mejor su funcionamiento, en el anexo II se pueden encontrar las explicaciones tanto de los atributos como de los métodos de las clases.

3.2.2 Clase Celda

La clase *Celda* crea objetos que almacenan la información de cada región en la que se ha dividido el escenario. Esta clase está diseñada de tal forma que no tiene restricciones en el número de esquinas de las regiones, dando la posibilidad en el futuro de que las celdas puedan ser diferentes formas.

Entre los atributos de esta clase se encuentra *state*, un atributo al que se le asigna un dato de tipo entero para indicar si hay situada alguna región de interés en esta celda y de que tipo es. Para identificar el estado de la celda se consultaron funciones de distintas librerías como son OpenCV y Boost C++ que calcularan la intersección entre dos polígonos. Finalmente, se decidió hacer uso de una función llamada *intersects* de la librería Boost C++. Esta función devuelve un valor de tipo bool: true cuando hay una intersección entre los dos polígonos y false en el caso contrario.

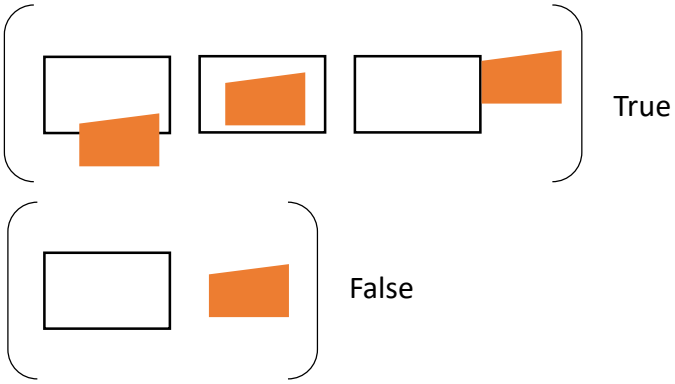


Figura 12. Valor que devuelve la función intersects de Boost. El rectángulo azul representa a una celda y el polígono rosa a una región de interes.

3.2.3 Clase CuatroLados

La clase *CuatroLados* al tener una relación de herencia con la clase *Celda* tendrá los atributos y métodos de ésta además de los suyos propios. En esta clase se crean regiones con forma cuadrada o rectangular.

La razón de crear está clase es facilitar mejorías en el futuro. En vez de crear directamente las celdas cuadradas, se crea una clase con la forma que se desea. De esta forma, en el futuro, si se desea implementar un algoritmo de descomposición en celdas triangulares, se deberá crear una clase triángulo con las características propias de un triángulo y la clase *Celda* no tendrá que ser modificada.

3.2.4 Clase CellDecomp

La clase *CellDecomp* representa el conjunto de celdas en el cual ha sido dividido el escenario. Para comodidad del usuario, se han creado métodos dentro de esta clase que permiten imprimir y mostrar por pantalla la descomposición en celdas realizada.

En este proyecto la descomposición en celdas utilizada ha sido la de celdas cuadradas que se lleva a cabo en la clase Malla. En el futuro, si se desea implementar la descomposición en celdas con otra forma diferente al cuadrado, únicamente se deberá crear una clase, con relación de herencia con *CellDecomp,* que represente la división del escenario en regiones de la forma deseada, el método para imprimir y dibujar dicha descomposición no será necesario crearlo debido a que lo heredara de esta clase.

3.2.5 Clase Malla

La clase *Malla* al tener una relación de herencia con la clase *CellDecomp* tendrá los atributos y métodos de ésta además de los suyos propios. En esta clase se crea una descomposición en celdas cuadradas. Esta clase crea un objeto que representa la división del escenario en cuadrados del tamaño de los robots.

A continuación, se muestran los intentos para dibujar la malla mediante el método creado en la clase CellDecomp y finalmente, el resultado de dicha descomposición correctamente.

En un principio se dibujó la malla introduciendo en la función *line*() de OpenCV las coordenadas de los puntos de las celdas en píxeles. Para ello, primero se cambió el valor de las coordenadas para escribirlas respecto al origen de coordenadas que establece OpenCV. Esta diferencia en el origen de coordenadas de la plataforma y de OpenCV es consecuencia de que OpenCV coje como punto (0,0) la esquina superior izquierda de la imagen, y en el mundo matemático y de la ingeniería este punto está situado siempre en la esquina inferior izquierda, como en la plataforma utilizada en este proyecto. En la figura 13 puede verse donde están situados los orígenes de coordenadas según OpenCV y la plataforma.

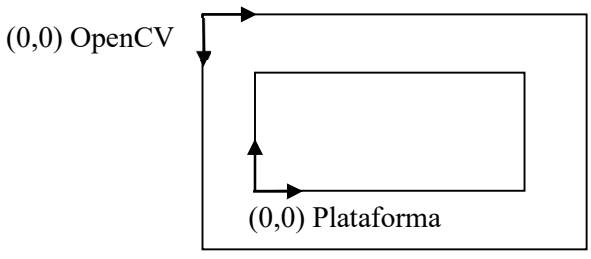


Figura 13. Posición del eje de coordenadas en la plataforma y en OpenCv.

Posteriormente, había que convertir las coordenadas en centímetros a coordenadas en pixeles. Para ello se creó una función que transformaba la distancia en centímetros

entre puntos a pixeles. Para dicha conversión era necesario conocer una relación entre centímetros y pixeles. Tanto el tamaño de los marcadores, como la distancia entre ellos eran conocidos en pixeles y en centímetros. Ambos tenían ventajas e inconvenientes y por ello se realizó un pequeño estudio para conocer cual calculaba la mejor relación centímetro-píxel que se muestra en la figura 14. Para dicha prueba se realizaron 100 mediciones. Los marcadores con altura no se utilizaron en esta prueba debido a que al no estar adheridos a la plataforma la distancia entre ellos puede variar mucho cada vez que se instala la plataforma.

	Ventajas	**Inconvenientes**	**Media (cm/pixel)**	**Desviación**
Tamaño del marcador	Se realiza de forma automática.	Distancia muy pequeña, el error se apreciará mucho.	1.7661	0.3903e-3
Distancia entre marcadores	Requiere tomar e introducir las medidas al montar el escenario.	La distancia es grande, el error será más pequeño.	1.8506	0.1296e-6

Figura 14. Relación centímetro-pixel.
***x**: indica la conversión en el eje x e **y**: indica la conversión en el eje y.*

Teniendo en cuenta los datos, se escogió la conversión calculada por la media de las distancias entre los marcadores. El dibujo de la malla resultante se muestra en la figura 15.

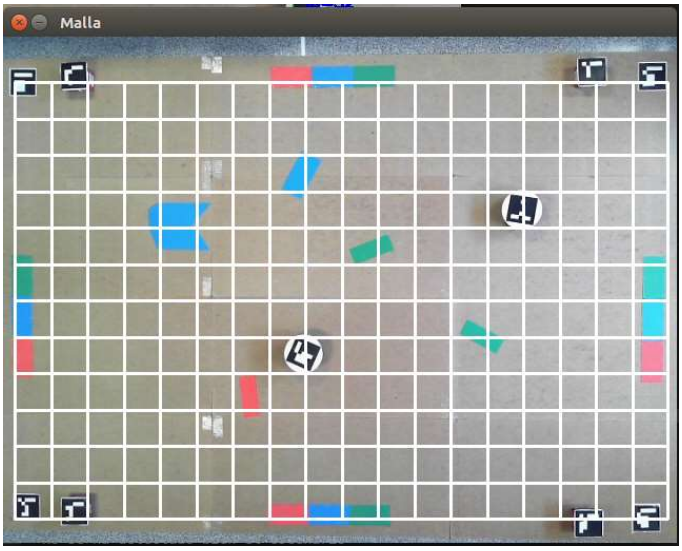


Figura 15. Intento dibujo malla 1 .

En la figura anterior, a pesar de que el origen de coordenadas ya coincide, se puede apreciar como el eje de coordenadas de la malla no coincide con el eje de coordenadas de la plataforma, pues este eje no se sitúa sobre la línea imaginaria que une a los dos

marcadores inferiores que delimitan la plataforma. Esto es debido a que la imagen capturada por la cámara está deformada. Por ello, se necesita convertir la posición de las esquinas de las celdas en la plataforma real a su posición en pixeles en la imagen deformada.

Para poder convertir las coordenadas de los puntos de centímetros de la plataforma a pixeles de la imagen, se debe calcular la matriz de homografía (figura 16). Esta conversión se ha realizado dentro de un método nuevo creado dentro de la clase *scene*, *ConversionPlatformToCV*, que utiliza una función de OpenCV llamada *findHomography*. A esta función, para que pueda calcular la matriz de homografía, hay que introducirle 4 coordenadas en pixeles y sus correspondientes 4 coordenadas en el mundo. [18] [7]

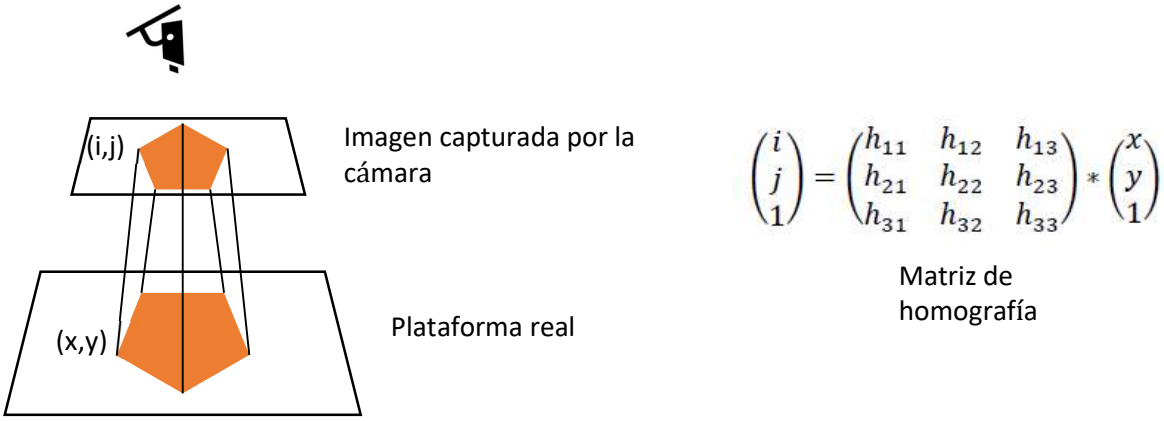


Figura 16. Conversión de puntos del mundo real a la imagen capturada por la cámara.

En la figura 17 se puede ver la malla dibujada sobre el video de la plataforma. En el apartado 4.2 y en el anexo II de este documento, se explicarán las características de esta malla cuadrada como, por ejemplo, porque la malla no llega a la parte superior de la plataforma.

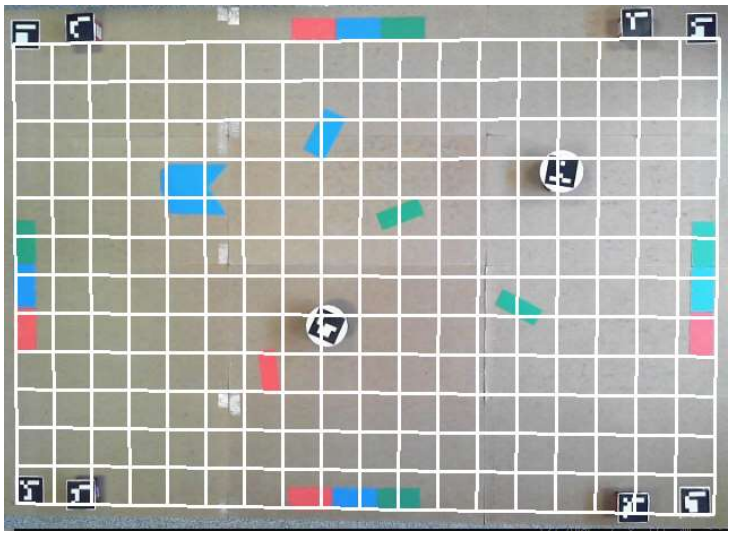


Figura 17. Dibujo de una malla creada por descomposición en celdas cuadradas sobre la plataforma.

3.4 Grafo

Una vez el escenario está discretizado y descompuesto en celdas se crea un grafo donde cada celda es un vértice del grafo que se conecta mediante arcos con sus celdas contiguas. En este proyecto, al considerar todas las regiones de interés como obstáculos, las celdas que contienen una región de interés no están conectadas con sus celdas contiguas pues el robot no puede pasar por ellas. Para diseñar este grafo se ha creado la clase Graph.

3.4.1 Clase Graph

El constructor de la clase *Graph* recorre la malla en busca de celdas contiguaspara relacionarlas mediante un arco, creando así un grafo. Como se ha explicado en el apartado 2.3.2 de este documento, almacena dicho grafo en una lista de adyacencia. Además, para calcular más adelante la trayectoria más corta, se le asigna un peso de valor 1 a los arcos que unen los vértices. En la figura 18 se puede ver una representación gráfica del grafo que se crearía en la clase *Graph* a partir de una malla que contiene un obstáculo en la celda C_5.

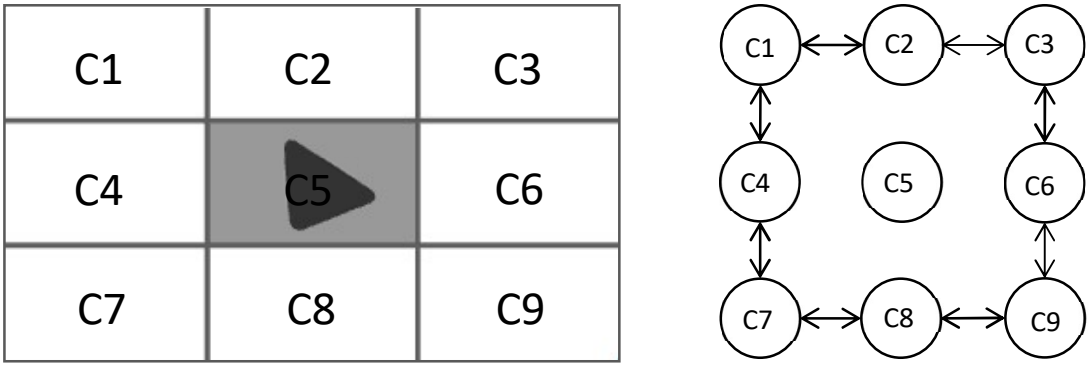


Figura 18. Representación de un grafo dirigido interpretando la región de interés situada en la celda C_5 como un obstáculo.

Para la simulación de la navegación de los robots, las celdas que conforman el borde de la malla se han asignado como celdas de descanso. Estas celdas no se han agregado al grafo, y serán las celdas donde los robots iniciarán y terminarán sus trayectorias.

Adicionalmente, se ha creado un método en la clase *Graph* para guardar en un archivo png el dibujo del grafo calculado. En la figura 19 se puede ver un trozo de un grafo creado a partir de la descomposición en celdas cuadradas de la plataforma utilizada en este proyecto. Debido a que el grafo se compone de 215 nodos y 782 arcos, no se expone la imagen entera.

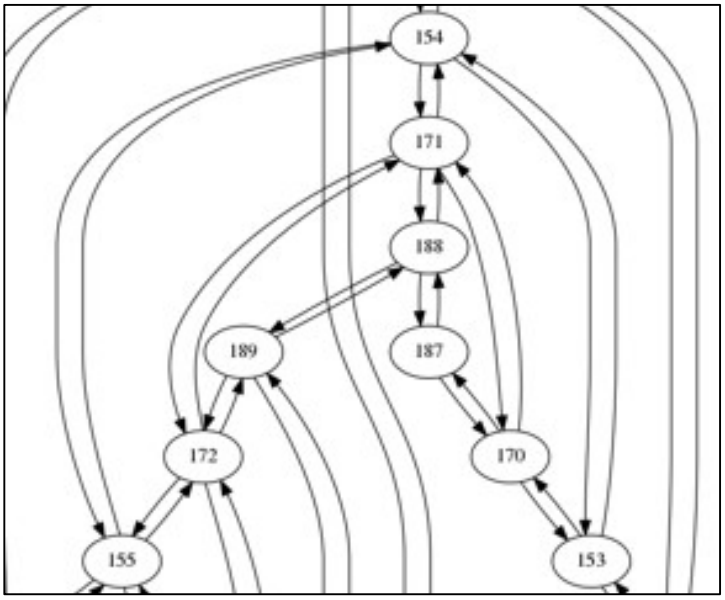


Figura 19. Pequeña parte de un grafo.

3.5 Cálculo de trayectorias

Con el grafo creado, se pueden calcular trayectorias eligiendo un punto de partida y uno de destino. Para ello se ha creado la clase *PlanningPath*, que crea objetos cuyos atributos almacenan las trayectorias que los métodos de esta clase calculan.

En el ejemplo de la figura 18, suponiendo que el robot tiene que ir de la celda C1 a la celda C6, existirían dos trayectorias para alcanzar el destino puesto que desde la celda 1, el robot puede dirigirse hacia la celda 2 o la celda 4 (figura 20):

- Trayectoria 1 (azul): C1 → C2 →C3 →C6
- Trayectoria 2 (naranja): C1 →C4 →C7 →C8 →C9 →C6

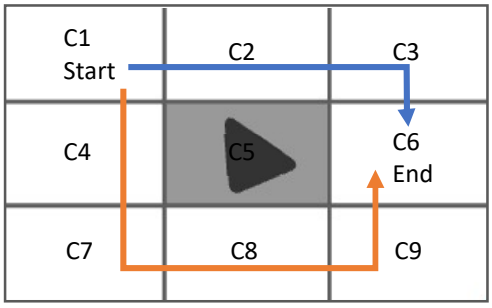


Figura 20. Trayectorias entre dos puntos

3.5.1 Clase PlanningPath

La clase *PlanningPath*, no busca únicamente una trayectoria del robot al destino, sino que se busca la trayectoria posible más corta. Para ello primero se utiliza una función de la librería Boost que implementa el algoritmo Dijkstra. La característica de esta función es que deja introducir el punto de partida, pero no el de destino, y por ello devuelve la trayectoria más corta de la celda inicial a todas las demás celdas del grafo. Siguiendo el ejemplo, esta función nos devolvería las siguientes trayectorias:

- Trayectoria A (de C1 a C2): C1 → C2
- Trayectoria B (de C1 a C3): C1 → C2 → C3
- Trayectoria C (de C1 a C4): C1 → C4
- Trayectoria D (de C1 a C5): No hay trayectoria posible. Obstáculo en C5.
- Trayectoria E (de C1 a C6): C1 → C2→ C3→ C6
- Trayectoria F (de C1 a C7): C1 → C4→ C7
- Trayectoria G (de C1 a C8): C1 → C4→ C7→ C8
- Trayectoria H (de C1 a C9 hay dos trayectorias igual de longitud, devolverá una aleatoriamente): C1 → C2→ C3→ C6→ C9

Una vez se tienen estas trayectorias, solo queda elegir aquella trayectoria que conecta la celda inicial con la celda final. En este ejemplo esa trayectoria sería la E. La azul en la figura 20.

3.6 Cambio de celdas a puntos

La ruta en el grafo obtenida con la clase PlanningPath es un conjunto de celdas. El siguiente paso consiste en transformar esta secuencia de celdas en una secuencia de puntos. Debido a que hay distintas maneras de hacer dicha transformación, como se explicará en el capítulo 4 de este documento, se ha creado la clase *Planificador* que se encarga de convertir la ruta creada en PlanningPath a una secuencia de puntos.

3.6.1 Clase Planificador

La clase Planificador es la encargada de hacer la conversión de una trayectoria de celdas a una secuencia de puntos para poder ser enviada al robot, y dependerá del formato que el usuario habrá introducido en el archivo JSON.

En este trabajo se han calculado las coordenadas del centro de las celdas cuadradas, y serán estas coordenadas las que formen la nueva secuencia de puntos.

Figura 21. Representación de secuencia de celdas y secuencia de puntos.

El resultado de los puntos que debe seguir el robot se ha dibujado por pantalla para que el usuario pueda ver las trayectorias de los robots. Se ha representado la celda de la que parten los robots con un círculo azul, y con uno amarillo la celda de destino.

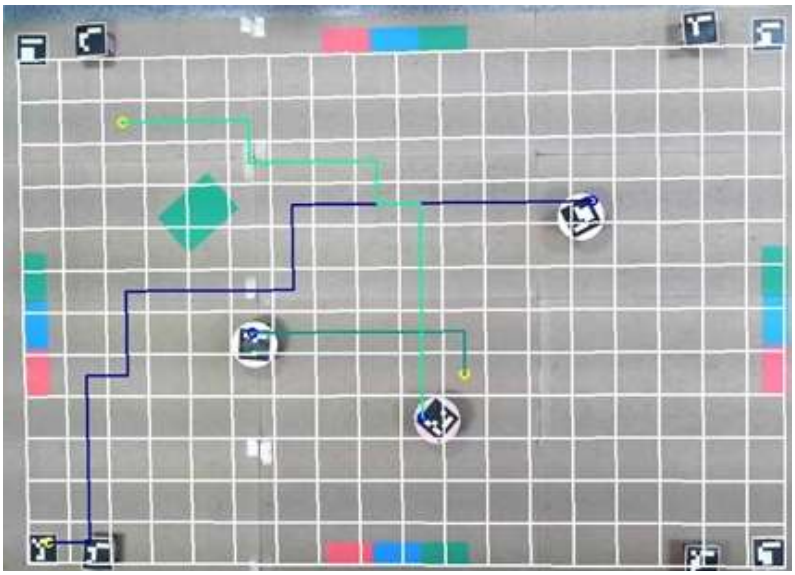


Figura 22. Representación de las trayectorias que deben seguir los robots.

3.7 Evitar colisiones

Para evitar colisiones durante la navegación, se ha adaptado el algoritmo del banquero a la planificación de trayectorias mediante la descomposición en celdas. Este algoritmo se ha implementado en la clase *Controller*.

3.7.1 Clase Controller

La clase *Controller* es la clase creada para generar un supervisor que asegure una navegación sin colisiones. Para ello, se escogerá un robot aleatoriamente y se comprobará, antes de realizar ningún movimiento, si el estado es seguro. Un estado es seguro si, en el hipotético caso de trasladar el robot a una nueva celda, si existe un orden en el movimiento de los robots en el cual todos los robots pueden alcanzar su destino. Si el estado es seguro, este controlador permite mover el robot a su siguiente punto. En caso contrario, el robot no se moverá de su sitio y se pasará a comprobar si es seguro mover a otro robot de su posición. Esta clase cuenta además con un método dibujo, que va mostrando por pantalla las trayectorias que van recorriendo.

La comprobación del funcionamiento de esta clase se ha realizado mediante simulaciones de distintas situaciones que se muestran en el apartado 5 de este documento.

3.8 Mejoras en la clase scene

Por último, en este capítulo se van a comentar las mejoras que se hicieron en la clase *scene*, creada antes de empezar este proyecto, para mejorar la calidad del *software* de la plataforma.

3.8.1 Detección de robots y de ROIs

Durante el proyecto, una vez las trayectorias fueron dibujadas y mostradas por pantalla, se pudo apreciar como las trayectorias no empezaban en la celda donde se situaba el centro del robot, sino en una celda cercana, figura 23. Además, algunas trayectorias pasaban por encima de objetos.

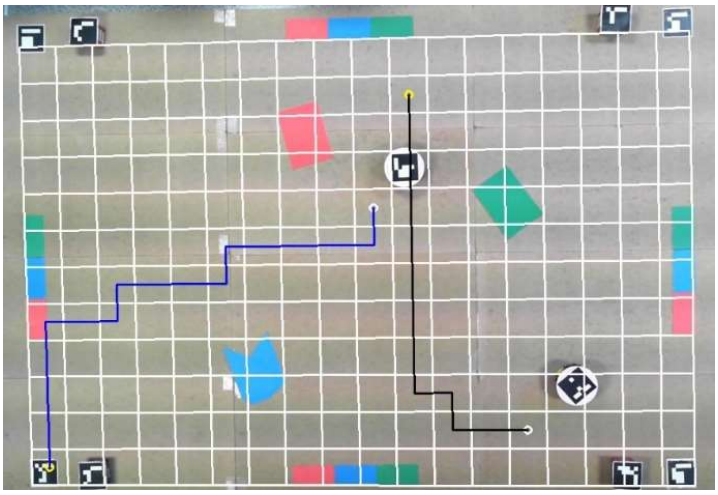


Figura 23. Trayectorias dibujadas con una mala detección de los robots.

Por ello, se decidió mostrar por pantalla el centro de los robots y los contornos de las regiones de interés. En la figura 24, la imagen de la izquierda muestra los contornos de las regiones desplazadas, al igual que en la foto de la derecha se observan desplazados los centros de los robots.

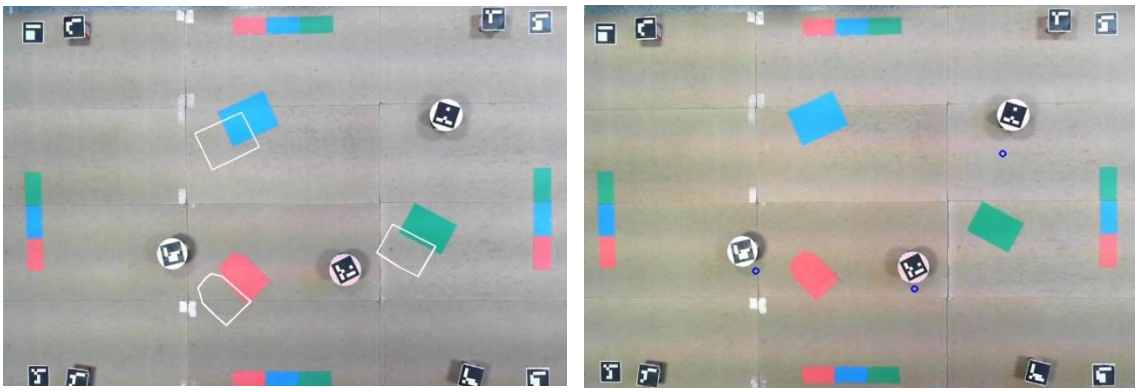


Figura 24. Mejora en la detección de las regiones de interés y de los robots. Parte 1.

El problema por el cual estos objetos eran detectados con mucho margen de error se debía a la matriz de homografía que se utilizaba para convertir los puntos de pixeles de la imagen a puntos en centímetros de la plataforma. Como se ha explicado en el aparatado 3.2.5 de este documento, para calcular la matriz de homografía mediante la función *findHomography* de OpenCV, que relaciona los píxeles de la imagen con los centímetros de la plataforma, se necesitan 4 puntos conocidos tanto en pixeles como en centímetros. Al comienzo de este proyecto, se escogían los centros de los marcadores para encontrar dicha matriz de homografía. El problema de estos puntos es que OpenCV no devuelve los centros de los marcadores y estos deben ser calculados, acumulando errores en el proceso. Esto hacía que el centro de los marcadores no tuviera mucha precisión, y como consecuencia, la matriz de homografía no relacionase correctamente los píxeles de la imagen con los centímetros de la plataforma.

Para corregir este error se utilizaron las esquinas de los marcadores que limitan el espacio. Es decir, la matriz de homografía es la misma que se calcula para dibujar en la imagen, con la diferencia de que esta vez los puntos se multiplican por la matriz inversa de dicha matriz de homografía, para poder pasar de píxeles a centímetros.

$$\begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

De coordenadas en centímetros a coordenadas píxeles.

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}^{-1} * \begin{pmatrix} i \\ j \\ 1 \end{pmatrix}$$

De coordenadas en píxeles a coordenadas centímetros.

Figura 25. Conversión de centímetros a píxeles y viceversa.

El resultado de la detección de los robots y los ROIs después de calcular la matriz de homografía con los puntos comentados anteriormente, se puede ver en la figura 26.

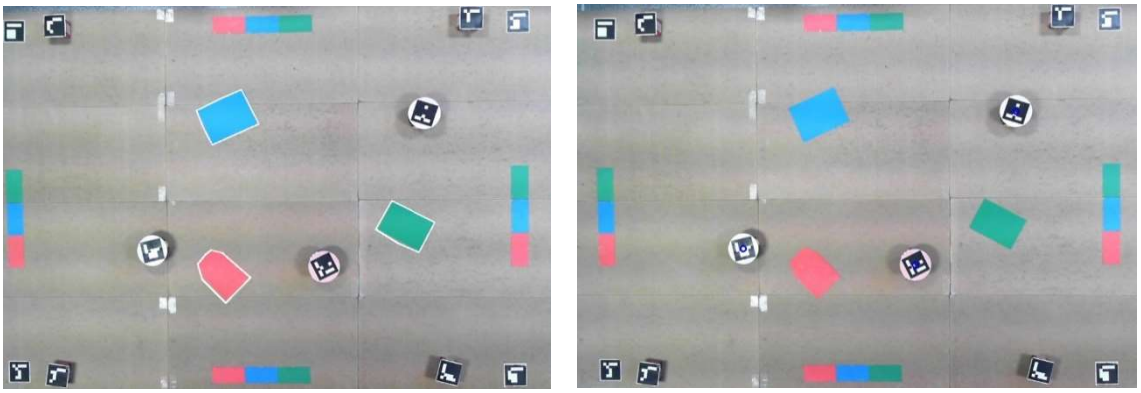


Figura 26. Mejora en la detección de las regiones de interés y de los robots. Parte 2.

3.8.2 JSON

Para que el código pudiera adaptarse a cualquier plataforma se ha decidido introducir más datos a través del archivo JSON. Al comenzar este proyecto, los identificadores de los marcadores y de los robots estaban escritos dentro del código, de tal forma que, si se detectaba otra plataforma con distintos marcadores y robots, el código fallaría durante el proceso de reconocimiento de la plataforma. Como se puede ver en la figura 27, en los marcadores elevados el identificador empieza por 1, en los marcadores a ras del suelo el identificador empieza por 2 y en los robots por 3.

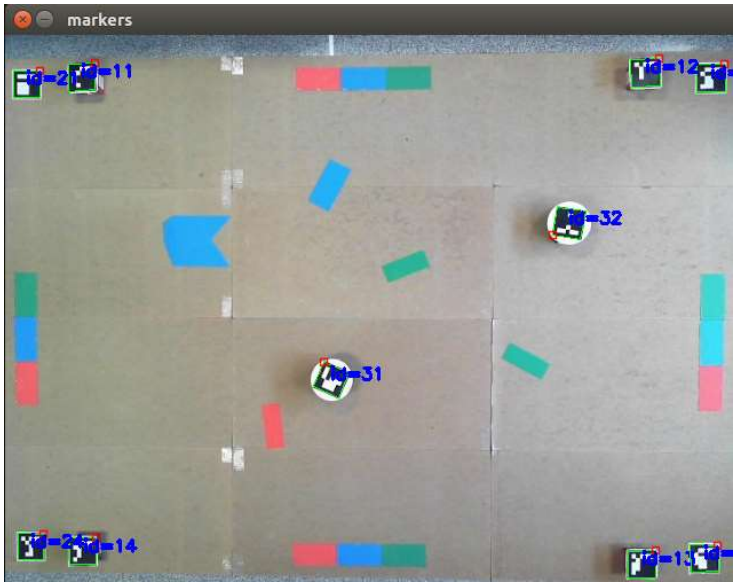


Figura 27. Elementos detectados por la función DetectMarkers de OpenCV.

La identificación de los marcadores y de los robots se utiliza a lo largo del código. Si esta identificación se introduce desde el código mismo, en una plataforma con marcadores diferentes éste no podría funcionar. Para ello se han introducido los identificadores de los marcadores y de los robots a través del archivo JSON.

Además, al haber definido los identificadores dentro del código, únicamente se podían tener 3 robots (ids=31,32,33) y en el caso de utilizar más robots había que modificar el código. Por contrario, al introducir los ids de los robots mediante el archivo JSON, estos serán guardados en un vector y no importara el número de robots siempre y cuando coincida con el número de identificadores introducidos por el usuario.

Capítulo 4

1. Toma de decisiones

Rediseñar el *software* de una plataforma multirrobot puede realizarse de muchas maneras. A ello hay que añadir que a lo largo del rediseño pueden aparecer dificultades que requieran estudiar distintas formas de solucionarlas. Por ello evaluar las diferentes opciones es necesario para poder decidir cuál es la mejor. En este apartado se mostrarán distintas decisiones que se han tenido que tomar a lo largo de este proyecto.

4.1 Asignación de destinos

En un principio se pensó en una situación donde el usuario requería X robots en X destinos, pero no importaba cual era el robot que llegaba a cada destino. Para ello se había pensado en calcular en *PlanningPath* la trayectoria más corta de cada robot a cada destino. De esta forma, suponiendo 2 robots y 2 destinos, deberíamos obtener 4 trayectorias, tal y como muestra la figura 18. Una vez calculadas estas 4 trayectorias se seleccionaría el conjunto más corto: las trayectorias 1 y 2 ó las trayectorias 3 y 4.

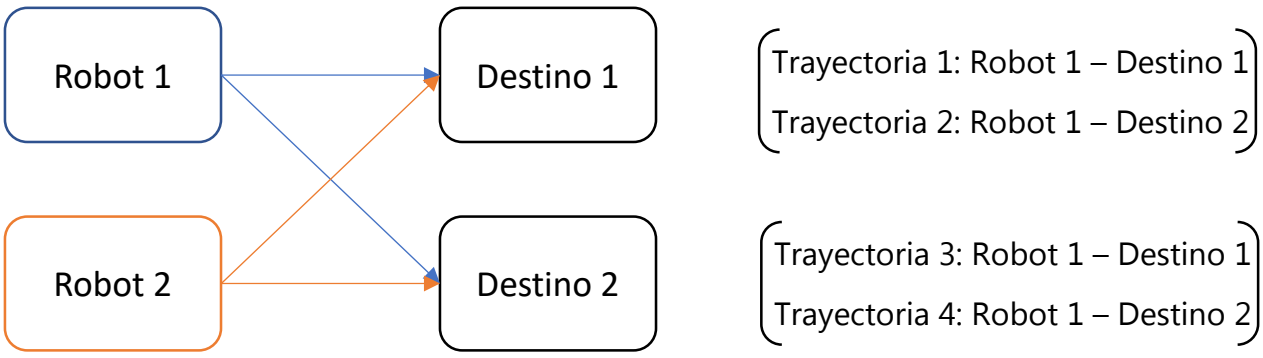


Figura 18. Relación entre robots y destinos.

A la hora de implementar la selección del conjunto más corto nos encontramos con la complejidad que conlleva hacerlo para n número de robots. Por ello se decidió dejar esta opción para un futuro TFG y que en este proyecto fuera el usuario el que asignaba a cada robot un destino.

4.2 Creación de la malla

A la hora de crear la malla la primera decisión que se debe tomar es la forma y el tamaño de las celdas. Para este proyecto se decidió una malla de celdas cuadradas con el tamaño del diámetro del robot.

Cuanto menor es el tamaño de la celda, más optimizada está la trayectoria debido a que no se pierde tanto espacio cuando una celda está ocupada por una parte de un obstáculo, figura 28. Pero, si la celda es menor que el tamaño del robot, al realizar la trayectoria el robot no se situará en una única celda en cada punto, su tamaño abarcará las celdas contiguas. Debido a esto se decidió que el tamaño sería el menor posible para que el robot pueda ocupar únicamente una celda, siendo este tamaño el diámetro del robot.

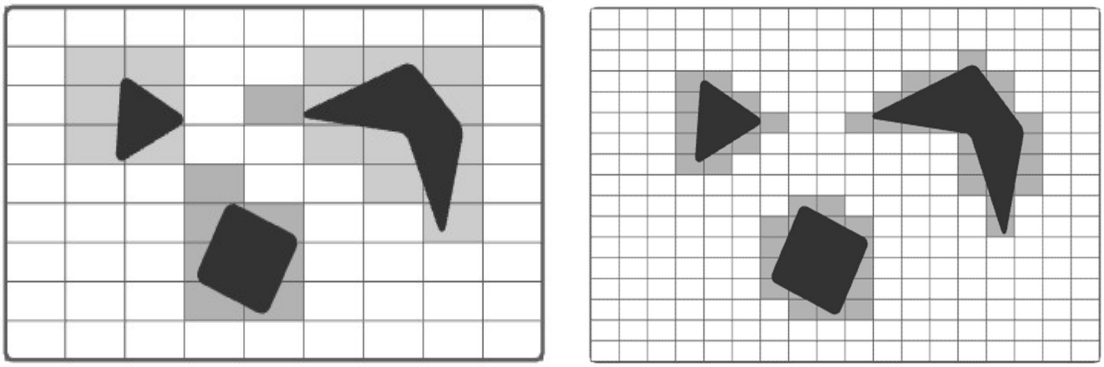


Figura 28. Malla con dos tamaños distintos de celda.

Pero dividir la malla en celdas con el tamaño del robot también tiene sus consecuencias. La división entre el tamaño de la plataforma y el del robot puede no dar exacto dejando así un espacio de la plataforma sin rellenar. En este trabajo se tomó la decisión de incluir unas celdas con forma de rectángulo si el área era mayor del 80% del área del resto de celdas.

La celda se empieza creando desde el punto cero de la plataforma, por ello la malla no alcanza la parte superior de la plataforma, porque el rectángulo tendría menos del 80% del área del resto de celdas. En el caso de la última columna ocurre lo contrario, el rectángulo es de un área mayor al 80% y por ello la última columna está formada con rectángulos.

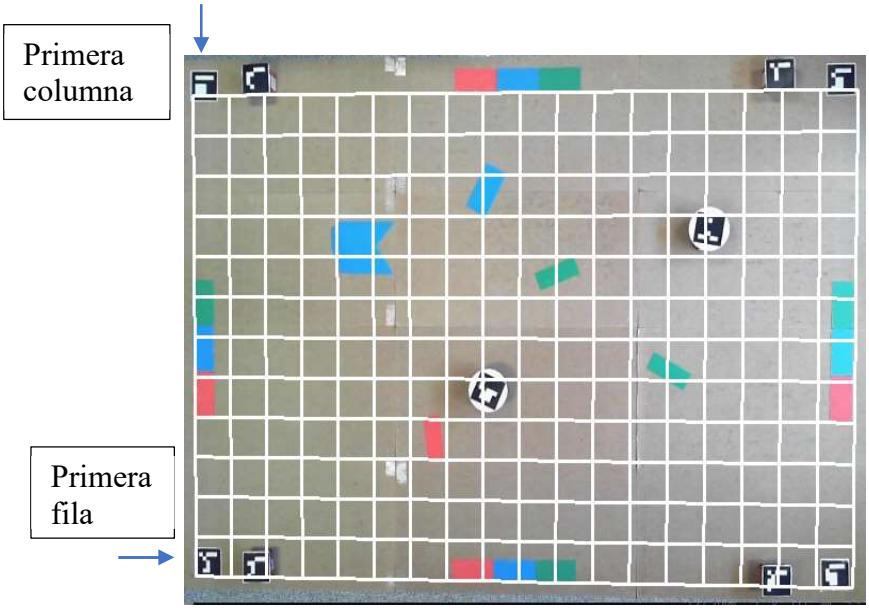


Figura 29. Malla con celdas cuadradas.

4.3 Creación del grafo

4.3.1 Lista de adyacencia

A la hora de crear el grafo se decidió utilizar la librería Boost C++ que posee la opción de crear y trabajar con grafos. Como se explicó en el apartado 2.3.2 de este documento, los grafos pueden almacenarse de distintas maneras. Boost C++ permite almacenar los grafos o bien, en una lista de adyacencia o en una matriz de adyacencia. Una lista de adyacencia, como se explicó anteriormente, es una lista de vectores que almacena a su vez otra lista de vectores. Por otro lado, una matriz de adyacencia es una matriz cuadrada donde se representa con 1 y 0 si hay relación entre vértices o no.

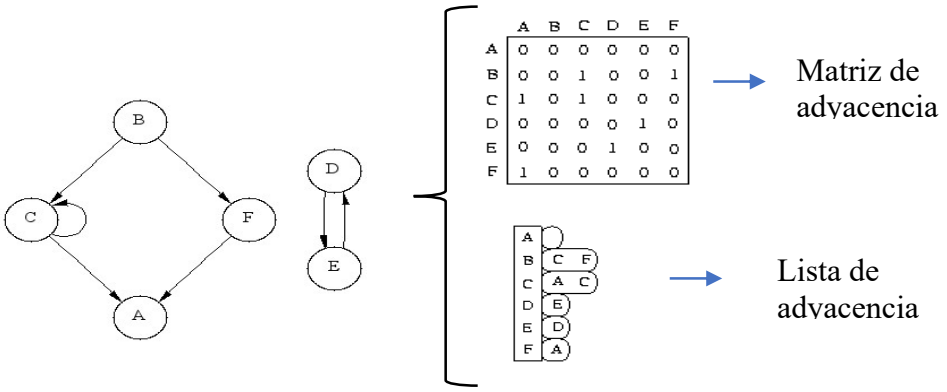


Figura 30. Matriz de adyacencia y lista de adyacencia de un grafo dirigido.

En la siguiente tabla se muestran las ventajas y desventajas de la matriz de adyacencia sobre la lista de adyacencia [1].

Ventajas	Desventajas
Permite introducir y eliminar arcos en tiempo real.	Utiliza mucha memoria.
	El tiempo que utiliza para recorrer los arcos de cada vértice es mucho mayor.

Figura 31. Ventajas y desventajas de la matriz de adyacencia respecto de la lista de adyacencia.

Este trabajo utiliza listas de adyacencia debido a que una vez creado el grafo no se va a modificar los arcos creados y por ello no es necesario utilizar una matriz de adyacencia con las desventajas de espacio y tiempo que estas tienen. Las siguientes decisiones que deben ser tomadas son las características de dicha lista de adyacencia. Lo primero de todo es conocer que parámetros requiere la clase la clase *adjacency*_list de Boost.

Los parámetros utilizados en este proyecto han sido:

- OutEdgeList: Representa la lista de arcos.

- VertexList: Representa la lista de vértices.

- Directed: Indica si el grafo es dirigido, no dirigido o dirigido bidireccional con acceso a los arcos.

- EdgeProperties: Propiedades internas de los arcos. puede trabajar y devolver distintos tipos de datos.

De *OutEdgeList* y *VertexList* puede almacenarse entre distintos tipos de datos que encajan en diferentes situaciones. Entre los tipos de datos que se pueden elegir, la lista y el vector son los más utilizados. En la tabla 2 se muestra en qué condiciones es de mayor utilidad seleccionar el vector y en cuales situaciones la mejor opción es la lista.

	Conozco el número de vértices antes de crear el grafo	**Necesito acceder a los vértices almacenados de forma continua**	**Necesito modificar el grafo una vez creado: añadir o eliminar vértices**
Vector	X	X	
Lista			X

Figura 32. Situaciones en las cuales es más favorable utilizar el vector y en cuales utilizar la lista en OutEdgeList y VertexList, parámetros de la clase Adjacency list de Boost.

En este proyecto el número de vértices del grafo es conocido antes de crearlo (número de celdas que contiene la malla). Además, una vez el grafo esté creado, éste no requiere de modificación alguna, pero en cambio, si se necesita un rápido acceso a los vértices para calcular la trayectoria del robot. Por estas razones, el tipo de dato seleccionado tanto para *OutEdgeList* como para *VertexList* ha sido el vector.

Otro de los parámetros nombrados anteriormente ha sido el parámetro *Directed*. Un grafo dirigido conecta vértices en un único sentido, mientras que en un grafo no dirigido sus arcos tienen ambos sentidos. Boost C++ da una tercera opción que es el grafo dirigido bidireccional con acceso a los arcos, pero requiere mucho espacio para cada arco. En este proyecto se ha creado un grafo direccionado dando la posibilidad en el futuro de que se implemente algún tipo de restricciones como, por ejemplo, carriles de circulación de un único sentido.

4.3.2 Celdas Adyacentes

En un principio se pensó, para deducir si dos celdas eran contiguas, comprobar si compartían dos esquinas. Este método funciona, no solo para la descomposición en celdas cuadradas, sino para la descomposición en celdas de cualquier polígono regular.

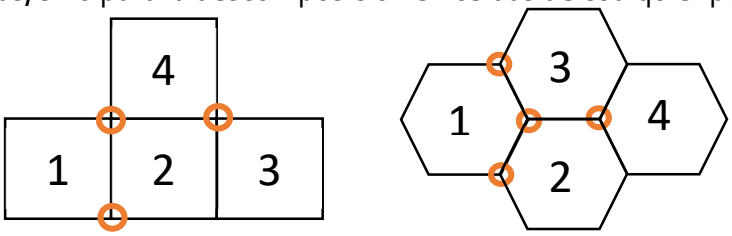


Figura 33. Mallas formadas por polígonos regulares.

El problema de deducir si dos celdas son contiguas de la anterior manera es en la descomposición trapezoidal. La descomposición en celdas trapezoidales detecta las esquinas de los obstáculos en el entorno y traza una línea vertical desde ese punto hacia arriba, hacia abajo o ambas, dependiendo de lo que el obstáculo permita creando trapecios. Como se puede ver en la figura 34, si implementamos el método anterior no detectará la celda 4 y 6 como celdas contiguas, debido a que solo comparten una esquina de sus trapecios.

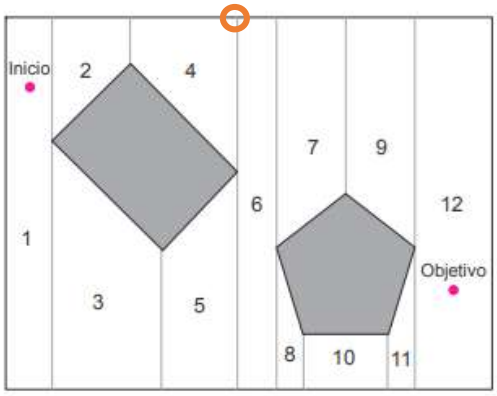


Figura 34. Espacio descompuesto en trapecios.

A pesar de que para la descomposición en celdas cuadradas utilizado en este proyecto este método funciona, como se desea hacer el código lo más general posible y que permita añadir mejorías en el futuro con facilidad, se decidió utilizar otro método.

Para ello se hizo uso de una función de Boost C++ llamada *intersection*. Esta función puede devolver un punto (*Point*), una línea de puntos (*LineString*) o un polígono (*Polygon*), dependiendo del tipo de dato que se elige para el parámetro GeometryOut. Debido a que para este propósito no era necesario calcular el polígono que genera la intersección y la intersección debía de ser de más de un punto, se seleccionó el tipo de dato *LineString* que representa a una secuencia de puntos. En la siguiente tabla se muestra el tamaño de la secuencia de puntos calculada por la función de Boost C++ en las que las celdas se encuentran en distintas situaciones.

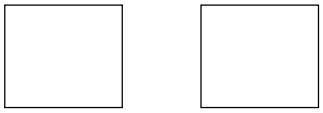
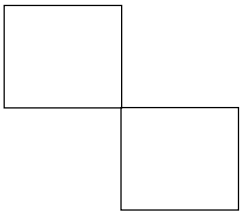
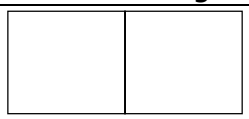
	Situación 1:	Situación 2:	Situación 3:
			
Length (output)	0	1	>1

Figura 35.Output de la función "intersecion" de Boost.

La tabla anterior demuestra que la función *intersection* puede encontrar una intersección entre dos celdas a pesar de que estas no sean contiguas como ocurre en la situación 2. Por ello si únicamente a encontrado un punto, el método creado no las considera celdas adyacentes y en el grafo no se añade un arco entre sus vértices.

4.4 Transformación de celdas a puntos

Como anteriormente se explicó en el apartado 3, se decidió que la transformación de celdas a puntos se creará en una clase diferente a *PlanningPath*. Esto se debe a que hay múltiples opciones para convertir la celda en un punto. Las dos mejores opciones las que se pueden ver en la figura 36: calcular las coordenadas del punto medio de la celda (opción A) o calcular las coordenadas del punto del medio de la línea de intersección entre celdas (opción B). Para este proyecto se optó por la opción A dejando la opción B como posibilidad para mejorar en el futuro y que sea el usuario el que pueda elegir.

Opción A Opción B

Figura 36. Transformación de celda a punto.

Capítulo 5

2. Análisis y conclusiones

El principal objetivo de este proyecto era mejorar un *software* de una plataforma multirrobot añadiendo una planificación de trayectorias basada en la descomposición en celdas. Se realizaron diferentes simulaciones, como la que se muestra a continuación, que permitía comprobar el funcionamiento de la planificación de trayectorias:

1. Se disponía el escenario sin ninguna región de interés situada en él y se calculaban las trayectorias para los robots.

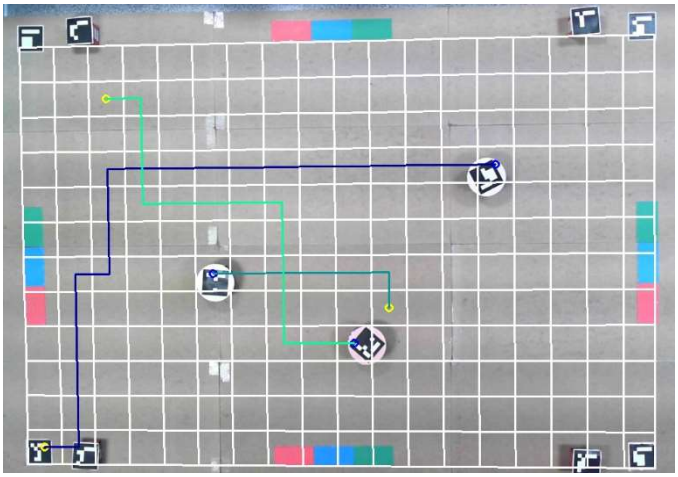


Figura 37. Simulación planificación de trayectorias. Parte 1.

2. A continuación, se colocaba un obstáculo que bloqueaba algunas celdas situadas en alguna de las trayectorias.

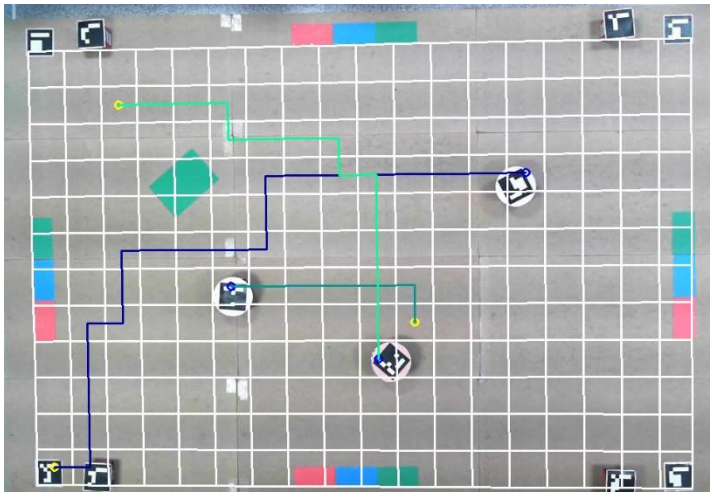


Figura 38. Simulación planificación de trayectorias. Parte 2.

3. Se repetía el paso 2.

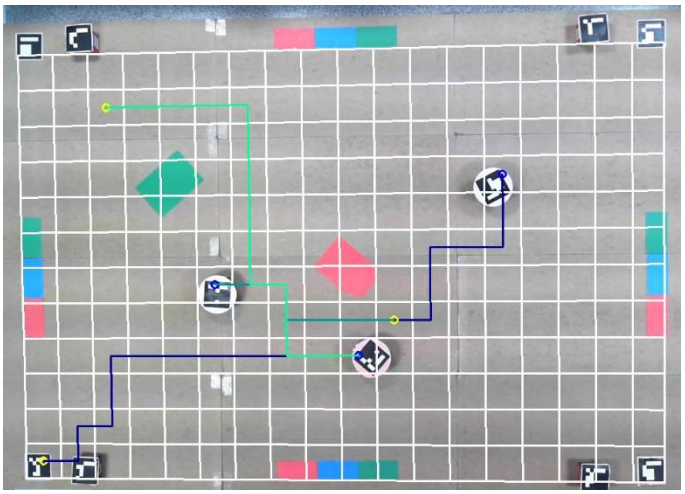


Figura 39. Simulación planificación de trayectorias. Parte 3.

Como podemos comprobar, las trayectorias van cambiando para evitar los obstáculos del camino.

Para analizar el funcionamiento de la clase *controller* que supervisa la navegación y evita colisiones, se ha creado un método que permite crear simulaciones. Para apreciar mejor la simulación, se han dibujado las trayectorias en colores y se ha ido pintando de negro la parte de la trayectoria que el robot ya ha realizado. Además, se ha pintado con un círculo azul el punto inicial de los robots, y con un círculo negro, se indica en que posición estarían los robots después de haberse movido.

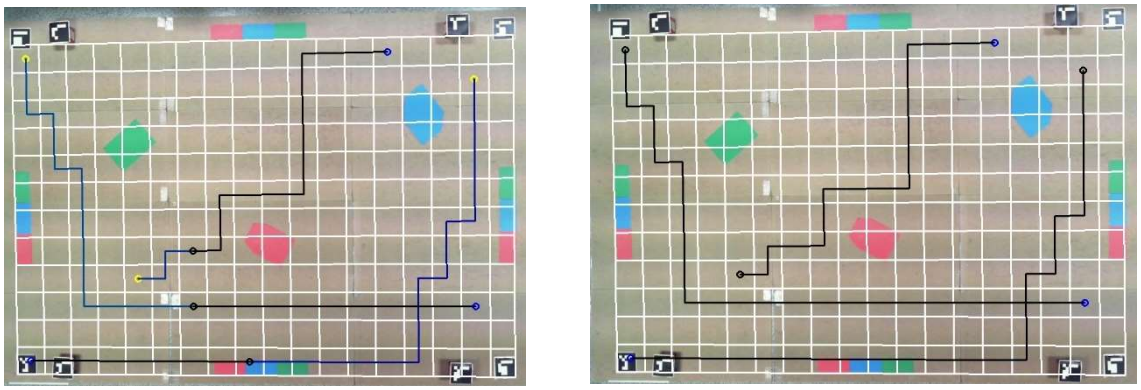


Figura 40. Simulación algoritmo del banquero. Los círculos negros representan la posición actual de los robots, los amarillos el destino y los azules la posición inicial de los robots.

Además, para comprobar si el algoritmo del banquero utiliza mucho tiempo, se ha hecho un estudio del tiempo que le cuesta al programa calcular si una posición es segura o no. Para ello, se ha variado el número de robots y el tamaño de las trayectorias y se han tomado 1000 medidas en cada simulación. Como se puede ver en la figura 41, el tiempo que se tarda en decidir si un estado es seguro o no, no es apreciable para el humano.

Número de robots	Trayectoria más larga (celdas)	Media de tiempo (ms)	Desviación
3	24	0.212	2.1932-13
6	27	0.427	1.269e-13

Figura 41. Estudio del tiempo que le cuesta al algoritmo del banquero decidir si un estado es seguro o no. Número de muestras: 1000.

Dado un estado formado por n pares (r_i,t_i) donde r_i es un robot y t_i es su trayectoria, el coste de determinar si un estado es seguro está acotado superiormente por $n^2 * L$, donde L es la longitud de la trayectoria más larga.

La conclusión general que puede sacarse al terminar este proyecto, y que se ve reflejada sobre todo en el apartado 4 de este documento, es la necesidad de adaptarse a las circunstancias. El diseño de *software* pensado antes de empezar un proyecto puede sufrir cambios debido a las dificultades que se encuentra por el camino o a ideas nuevas que surgen para mejorarlo.

Los conocimientos que he adquirido durante este proyecto respecto a mi formación anterior, grado en ingeniería electrónica y automática, son la planificación de trayectorias y la búsqueda y uso de nuevas librerías. Además, este proyecto me ha permitido reforzar y mejorar mis conocimientos básicos en la programación orientada a objetos y a conocer un lenguaje nuevo para mí, como es C++.

Durante todo el proyecto a la hora de diseñar las clases necesarias para el algoritmo de descomposición en celdas, se ha tenido en cuenta futuras mejoras y la manera de implementarlas sin necesidad de modificar el código escrito en este proyecto. Entre estas mejoras, destaca crear un grafo no constante que permita modificar el estado de las celdas durante la navegación de los robots, por ejemplo, añadir o eliminar regiones de interés, o convertir el robot en obstáculo si este permanece quieto dentro de la plataforma, bien debido a una avería o por requerimiento del usuario. Esto supone una replanificación de las trayectorias antes de hacer uso de la clase *controller*. Para comprobar si esta mejoría para el futuro era factible, se realizaron unas medidas para calcular el tiempo que tarda el algoritmo en planificar una trayectoria. Se realizaron 1000 medidas para ambos casos y se consiguieron unos resultados que permiten replanificar la trayectoria antes de cada movimiento:

Número de celdas	Media de tiempo (ms)	Desviación
7	0.4328	3.4066e-9
11	0.44919	5.1737e-9
24	0.48714	9.8395e-9

Figura 42. Estudio del tiempo que tarda en calcular una trayectoria. Número de muestras: 1000.

Otras líneas futuras que pueden mejorar el *software* de esta plataforma multirrobot son:

- Implementar un algoritmo que permita decidir que robot va a cada destino dependiendo de la posición de éstos como se explica en el apartado 4.1.

- Implementar un algoritmo de descomposición en celdas con otras formas: trapezoidal, triangular…

- Transformar la trayectoria de celdas a trayectorias de puntos utilizando criterios diferentes, tal y como se explica en el apartado 4.4.
- Implementar otras formas de discretizar el escenario como, por ejemplo, los grafos de visibilidad. Dando así la oportunidad al usuario de elegir de qué forma le conviene más discretizarlo.

- Asignar distintas funciones a las regiones de interés dependiendo de su color: obstáculos, destinos, puntos intermedios por donde pasar para ir a un destino…

Capítulo 6

3. Bibliografía

[1] J. Siek, L.-Q. Lee y A. Lumsdaine, boost c++ libraries, Addison-Wesley Professional., 2001.

[2] I. D.Craig, Object-Oriented Programming Languages: Interpretation, London: Springer, 2007.

[3] A. Pertursa, D. Tomás, C. Pérez, J. Aragonés, J. A. Pérez y F. Moreno, Programación 2, Universidad de Alicante.

[4] B. Rumpe, Modeling with UML, Springer, 2016.

[5] J. B. G. Barreto, «Diseño e implementación de un algoritmo que evite colisiones en un sistema multi-robot utilizando el Modified Banker's Algorithm.,» Universidad de Zaragoza, 2020.

[6] D. R. García, «Rediseño de una Plataforma de robots móviles,» Universidad de Zaragoza, 2021.

[7] «OpenCV,» [En línea]. Available: https://opencv.org/.

[8] F. J. Romero Ramirez, R. M. Salinas y R. M. Carnicer, «Speeded Up Detection of Squared Fiducial Markers,» de *Image and Vision Computing 76*, 2018.

[9] R. M. Salinas, F. J. Cuevas y R. M. Carnicer, «Generation of fiducial marker dictionaries using Mixed Integer Linear Programming,» de *Pattern Recognition*, 2015.

[10] J. Pokorný, «JSON Functionally,» de *Advances in Databases and Information Systems*, 2020.

[11] C. Mahulea, M. Kloetze y R. González, Path Planning of Cooperative Mobile Robots, Piscataway: IEEE Press Editorial Board, 2020.

[12] R. A. Española, «Dle,» [En línea]. Available: https://dle.rae.es/grafo.

[13] anonymous, «Universidad de Pamplona,» [En línea]. Available: https://www.unipamplona.edu.co/unipamplona/portalIG/home_23/recursos/general/11072012/grafo3.pdf.

[14] S. Mukherjee, «Dijkstra's Algorithm for Solving the Shortest Path Problem on Networks Under Intuitionistic Fuzzy Environment,» *Journal of Mathematical Modelling and Algorithms,* vol. 11, pp. 345-359, 2012.

[15] E.W.Dijkstra, «A Note on Two Problems in Connexion with Graphs,» *Numerische Mathematik ,* vol. 1, pp. 269-271, 1959.

[16] J.Ezpeleta, F.Tricas, F.Garcia-Vallés y J.M.Colom, «A Banker's solution for deadlock avoidance in FMS with flexible routing and multi-resource states,» de *IEEE Transactions on Robotics and Automation*, vol. 18, 2002, pp. 621-625.

[17] L. Kalinovcic, T. Petrovic, S. Bogdan y V. Bobanac, «Modified Banker's algorithm for scheduling in multi-AGV systems,» de *IEEE International Conference on Automation Science and Engineering*, 2011.

[18] J. B. Arbex, «Localización de múltiples robots móviles mediante una cámara cenital,» Universidad de Zaragoza, 2016.

[19] L. E. Aponte, «El blog de Prof. Luis E. Aponte I,» [En línea]. Available: http://programandoenjava.over-blog.es/article-el-uml-o-lenguaje-de-modelado-unificado-como-herramienta-en-el-modelado-de-objetos-53386438.html.

[20] «https://www.microsoft.com/,» [En línea]. Available: https://www.microsoft.com/es-xl/accessories/business/lifecam-studio-for-business?activetab=overview:primaryr2.

[21] I. f. Cobo, «Software de Simulación del algoritmo Del Banquero,» *Ciencia & Futuro,* vol. 3, nº 3, 2013.

Lista de figuras

ANEXOS

Anexo I

I. *Software* inicial

En este anexo se explican las clases que existían en el *software* antes de empezar este proyecto y el archivo JSON. La explicación de los métodos y atributos de la clase se pueden encontrar en el Trabajo de Fin de Grado de Daniel Roche García [5]

ROBOT

La clase *Robot* crea abstracciones de los robots que se encuentran en la plataforma. Es decir, esta clase crea unos objetos con las características de cada robot. Además, esta clase cuenta con las funciones de comunicación necesarias para la navegación de los robots y con atributos que almacenan la posición y orientación del robot.

CAMERA

La clase *Camera* se utiliza para representar a las cámaras utilizadas para la visualización digital y el reconocimiento de la plataforma. En el caso de este proyecto al utilizar únicamente una cámara, solo se crea un objeto de esta clase.

Los objetos de esta clase almacenan datos como los pixeles de ancho y alto que utiliza y hace uso de funciones que configuran la clase *VideoCapture* de la librería OpenCV.

ROI

La clase *ROI* crea objetos que representan a las regiones de interés que se sitúan en la plataforma. Estos objetos tienen de atributos datos como el contorno de las regiones, el color o el centroide. [6]

SCENE

La clase *scene* representa la plataforma en su totalidad. Es en esta clase donde se lleva lugar la detección de los robots y de las regiones de interés creando así objetos de sus respectivas clases para representarlos.

A la clase ya *scene* ya creada, durante este proyecto se le ha añadido un atributo de tipo vector llamado *limite* que almacena las coordenadas de las 4 esquinas que limitan la plataforma. Además, se ha creado un método llamado *ConversionPlatformToCV*, comentado en el apartado 3.2.1 de este documento para

poder transformar unas coordenadas de la plataforma real en coordenadas en pixeles de la imagen capturada por la cámara.

CONTROLLER

Esta clase representa al controlador que supervisa la navegación de los robots para que no colisionen mediante el algoritmo del banquero. Esta clase fue creada en el Trabajo de Fin de Máster de Jose Benigno García Barreto [5] y Daniel Roche García [6] añadió alguna función para reestructurarla. Para más información sobre los atributos y métodos de esta clase es necesario consultar ambos trabajos.

JSON

Como se definió anteriormente, un archivo JSON es un formato de texto utilizado para la transferencia de datos y que se compone de un conjunto de campos que están asociados con un identificador. Un ejemplo de cómo se introducen datos en el archivo y como se leen desde el programa se ve en la figura 44.

```
"Camera":[
  {
    "NUMBER":0,
    "FPS": 60,
    "WidthRes": 1920,
    "HeightRes": 1080
  }
],
```

```
ifstream datos(nombre);
Json::Reader reader;
Json::Value j;

reader.parse(datos,j);

ncamera = j["Camera"][i]["NUMBER"].asInt();
FPS = j["Camera"][i]["FPS"].asInt();
widthRes = j["Camera"][i]["WidthRes"].asInt();
heightRes = j["Camera"][i]["HeightRes"].asInt();
```

Figura 43. Escribir y leer datos en un archivo JSON. A la izquierda se muestra cómo se introducen datos en el archivo, y a la derecha la forma en la que estos datos se leen en el código.

En este proyecto se introducen los datos necesarios para la configuración de la clase *Camera*, la clase *scene* y las coordenadas de los destinos. Además, el usuario introduce en dicho archivo el algortimo de discretización que desea usar y el método para la transformación de trayectorias de celdas a trayectorias de puntos, figura 45.

```
"Discretize":"CellDescomp",
"CeldasToPoints": "PuntosMedios",
```

Figura 44. Elección de algoritmos y métodos en el archivo JSON.

Anexo II

II. Biblioteca de las clases creadas

Se puede acceder al código utilizado en este proyecto mediante el siguiente enlace:

https://drive.google.com/drive/folders/198XJIeQONzgw29eeCY1kVVqT4grZ0fHA?usp=sharing

MISSION

Clase en la que se crea un objeto misión con las coordenadas de los destinos que han sido introducidas por el usuario.

Atributos:

- **num:** número de misión.
- **destinos:** vector que almacena los destinos a los que deben dirigirse los robots.

Métodos:

- **mission():** constructor de la clase.
- **~mission():** destructor de la clase.
- **mission(string):** constructor de la clase que recibe el nombre del archivo JSON donde se almacenan las coordenadas de los destinos.
- **GetMisiones():** devuelve el atributo destinos.
- **GetNum():** devuelve el atributo número.

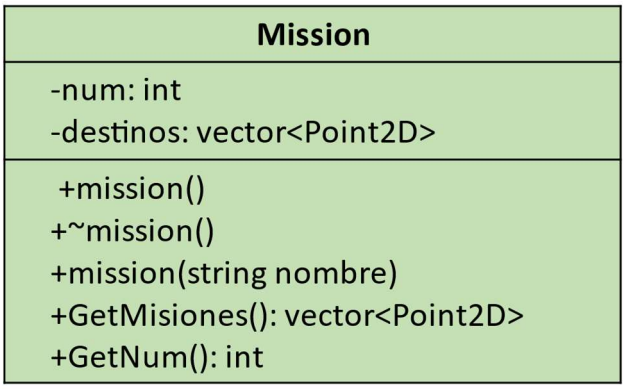


Figura 45. Clase Mission UML.

CELDA

Clase que crea abstracciones de las regiones en las que se divide el escenario al discretizarlo mediante el algoritmo de descomposición de celdas.

Atributos:

- **num:** número de misión.
- **state:** indica el estado de la celda: libre, ocupada por un obstáculo...
- **esquinas:** vector que almacena las coordenadas de las esquinas que forman la celda.
- **centro:** coordenadas del centro de la celda.

Métodos:

- **Celda():** constructor de la clase.
- **~Celda():** destructor de la clase.
- **Celda(int):** constructor de la clase celda al que se le introduce el número de celda.
- **SetState(scene):** Se comprueba si alguna región de interés se encuentra situada en la celda y se le asigna un valor al atributo state.
- **GetState():** devuelve el atributo estado.
- **GetEsquinas():** devuelve el atributo esquinas.

- **GetEsquina(int):** devuelve las coordenadas de la esquina que se le introduce.

- **GetNum():** devuelve el atributo número.

- **GetCentro():** devuelve el atributo centro.

- **print():** imprime por pantalla las coordenadas de las esquinas de la celda, así como el número que la identifica.

- **PuntoDentro(Point2d):** devuelve *true* si el punto introducido se encuentra dentro de la celda y *false* en caso contrario.

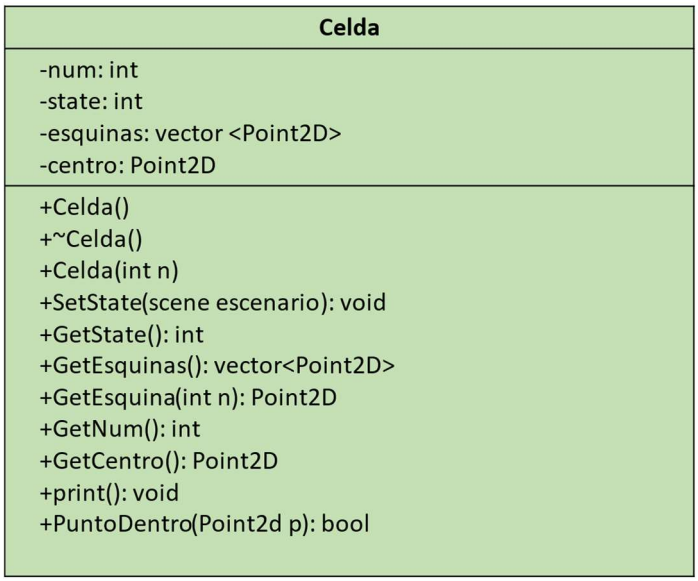


Figura 46. Clase Celda UML.

CUATROLADOS

Clase heredada de la clase *Celda* que genera regiones cuadradas o rectangulares. Los atributos de esta clase son heredados de la clase Celda. Además, todos los métodos de la clase Celda también los contiene la clase *CuatroLados*, por ello únicamente se explican aquí los métodos nuevos que contiene esta clase.

Métodos:

- **CuatroLados(int, Point2D, double, double):** constructor de la clase al que se le introduce: el número de celda, la coordenada de la esquina inferior izquierda, el ancho y el largo de la región.

- **SetEsquinas(Point2D, double, double):** A partir del punto introducido y de la anchura y altura de la región calcula la posición de las coordenadas de las esquinas y las almacena en el atributo esquinas.

- **SetCentro():** calcula las coordenadas del centro del polígono y las almacena en el atributo centro.

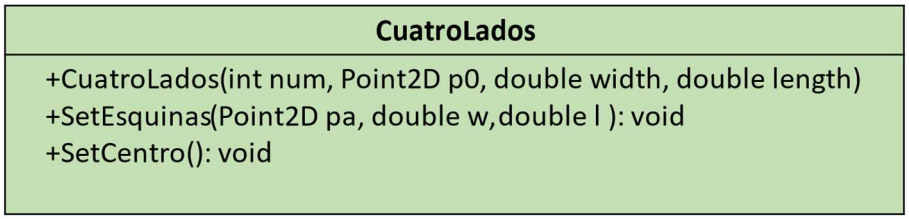


Figura 47. Clase CuatroLados UML.

CELLDECOMP

Clase que representa el conjunto de regiones en las que el escenario se discretiza.

Atributos:

- **num:** número de celdas que forman la malla.

- **cellDecomp:** vector que almacena las celdas que forman la descomposición del escenario.

Métodos:

- **CellDecomp():** constructor de la clase.

  ~CellDecomp(): destructor de la clase.

- **GetSize():** devuelve el atributo num.

- **GetCellDecomp():** devuelve el atributo cellDecomp.

- **GetCelda(int):** devuelve la celda cuyo número es el introducido.

- **print():** imprime por pantalla las celdas que forman la descomposición en celdas.

- **Dibujo(Mat im, scene escenario):** muestra por pantalla la descomposición dibujada. Ver Figura 17.

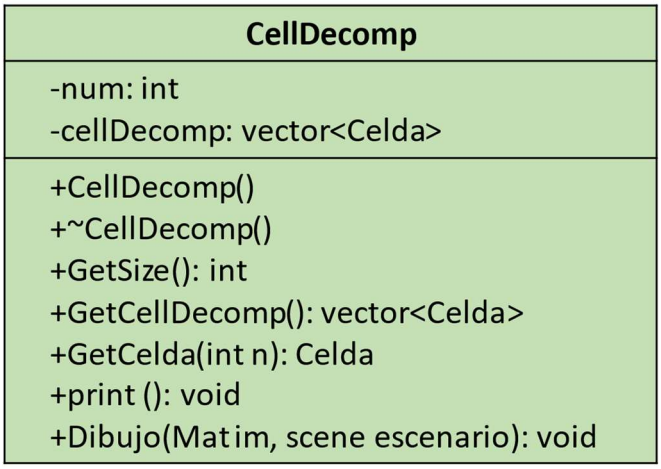


Figura 48. Clase CellDecomp UML.

MALLA

Clase heredada de la clase *CellDecomp* que discretiza el escenario en regiones cuadradas. Los atributos de esta clase son heredados de la clase *CellDecomp*. Además, todos los métodos de la clase *CellDecomp* también los contiene la clase *Malla*, por ello únicamente se explican aquí los métodos nuevos que contiene esta clase.

Métodos:

- **Malla(scene escenario):** constructor de la clase.

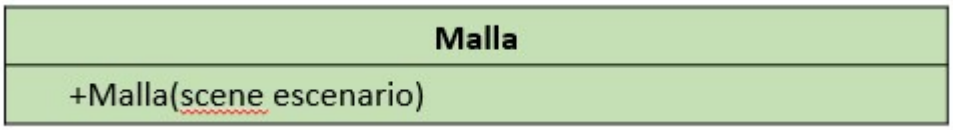


Figura 49. Clase Malla UML.

GRAPH

Clase en la que se crea el grafo que representa las relaciones entre las celdas creadas.

Atributos:

- **graph:** objeto de la clase adjacency_ list de la librería Boost que almacena el grafo.

Métodos:

- **Graph():** constructor de la clase.
- **~Graph():** destructor de la clase
- **Graph(Malla):** constructor de la clase al que se le introduce la malla en la que el escenario ha sido discretizado.
- **vecino(Celda, Celda):** devuelve true si las celdas introducidas son contiguas y false en caso contrario.
- **print():** imprime la lista de adyacencia: cada vértice con la lista de vértices a los que esta unido mediante un arco.
- **GetGraph():** devuelve el atributo graph.
- **DibujarGraph():** Guarda en un archivo png el grafo dibujado.

Graph
-graph: adjacency_list<vecS,vecS,directedS,property<vertex_distance_t, int>,property<edge_weight_t, int>, no_property, listS>
+Graph() +~Graph() +Graph(Malla m) +vecino (Celda c1, Celda c2): bool +print(): void +GetGraph (): typedef adjacency_list<vecS,vecS,directedS,property<vertex_distance_t, int>,property<edge_weight_t, int>, no_property, listS> +DibujarGraph(): void

Figura 50. Clase Graph UML.

PLANNINGPATH

Clase en la que se crea una trayectoria desde el punto de partida de cada robot y su respectivo destino.

Atributos:

- **caminos:** vector que almacena las trayectorias de celdas más corta que une las celdas de partida con las celdas de destino.

- **CeldaInicialRobots:** vector que almacena el número de las celdas donde se encuentran los robots antes de comenzar su navegación.

- **CeldaDestinos:** vector que almacena el número de las celdas donde se encuentran los destinos.

Métodos:

- **PlanningPath():** constructor de la clase.

- **~ PlanningPath ():** destructor de la clase

- **PlanningPath(mission, Graph, Malla, scene):** constructor de la clase al que se le introduce la misión a realizar, el escenario, el grafo y la malla.

- **GetCaminos():** devuelve el atributo caminos.

- **GetRobotQueCelda():** devuelve un vector con las celdas donde están situados los robots.

- **GetDestinoQueCelda():** devuelve un vector con las celdas donde están situados los destinos.

- **RobotQueCeldaSimulador(int num, int celda):** método utilizado para simular las posiciones de los robots.

- **RobotQueCeldaInicialSimulador():** método utilizado para simular las posiciones iniciales de los robots.

- **RobotQueCelda():** Recorre el conjunto de celdas en busca de las celdas donde están situados los robots y las almacena en el atributo CeldaInicialRobots.

  DestinoQueCelda(): Recorre el conjunto de celdas en busca de las celdas donde están situados los destinos y las almacena en el atributo CeldaDestinos.

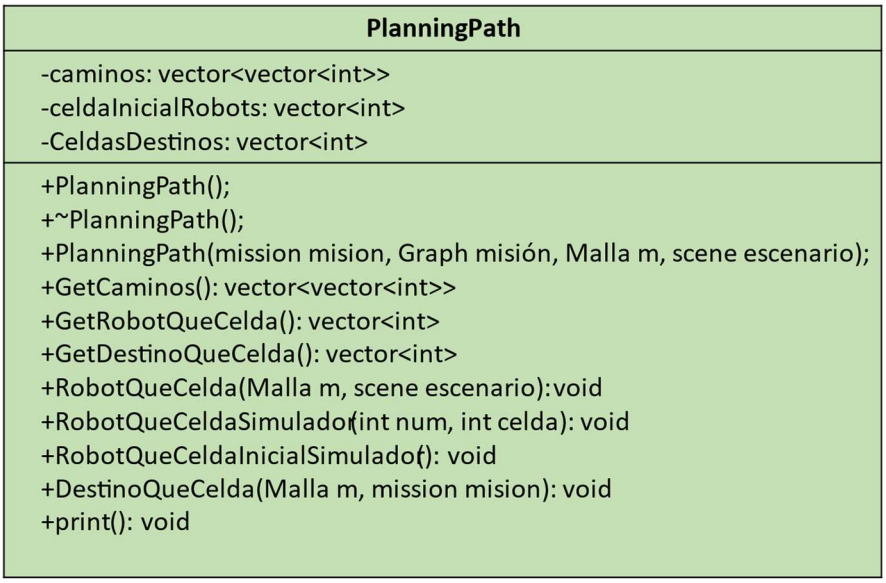


Figura 51. Clase PlanningPath UML.

PLANIFICADOR

Clase que transforma una trayectoria de celdas en una trayectoria de puntos.

Atributos:

- **paths:** vector que almacena las trayectorias de puntos más corta que une los puntos de partida con los puntos de destino.

Métodos:

- **Planificador():** constructor de la clase.

- **~ Planificador ():** destructor de la clase

- **Planificador(string, Malla, PlanningPath):** constructor de la clase que crea una instancia a partir de un string para introducir el archivo JSON que se desea, una malla y el PlanningPath donde se han calculado las trayectorias de celdas.

- **GetPaths():** devuelve el atributo paths.

- **PuntosMedios(Malla, PlanningPath):** transforma las trayectorias de celda a trayectorias de puntos y las almacena en el atributo paths.

- **Dibujartrayectorias(Mat, scene):** Muestra por pantalla las trayectorias dibujadas. Ver figura 22.

- **print():** imprime los puntos que forman la trayectoria.

Planificador
-paths: vector<vector<Point2D>>
+Planificador(); +~Planificador(); +Planificador(string nombre, Malla m, PlanningPath plan) +GetPaths(): vector<vector<Point2D>> +PuntosMedios(Malla m, PlanningPath plan): void +dibujarTrayectorias(Mat im, scene escenario): void +print(): void

Figura 52. Clase Planificador UML.

CONTROLLER

Clase en la que se crea un controlador para supervisar la navegación de los robots y evitar colisiones.

Métodos:

- **Controller():** constructor de la clase.

- **~ Controller ():** destructor de la clase.

- **Controller(CellDecomp m, PlanningPath planning, Planificador plan, Mat im, scene escenario):** constructor de la clase que crea una instancia a partir de un CellDecomp, un planningPath, un planificador, una matriz y un scene.

Controller
+Controller (); +~Controller (); +Controller (CellDecomp m, PlanningPath planning, Planificador plan, Mat im, scene escenario) +Banker(CellDecomp m,PlanningPath planning, int celdaInic, int i, vector<int> robots): bool +Dibujar(PlanningPath planning2, CellDecomp m, scene escenario, vector<int> celdasInics, Planificador plan, Mat im): void

Figura 53. Clase Controller UML.

POINT2D

Clase utilizada durante el proyecto que transforma una trayectoria de celdas en una trayectoria de puntos.

Atributos:

- **x:** coordenada x del punto.
- **y:** coordenada y del punto.

Métodos:

- **Point2D():** constructor de la clase.
- **~ Point2D():** destructor de la clase
- **Point2D(double, double):** constructor de la clase al que se le introducen los valores para las coordenadas x e y.
- **GetX() y GetY():** devuelven los atributos x e y respectivamente.
- **SetX(double)** y **SetY(double):** modifican los atributos x e y respectivamente, asignándoles el valor introducido.
- **Set(double, double):** modifica los dos atributos de la clase asignándoles los valores introducidos.
- **PuntosIguales(Point2D, Point2D):** devuelve *true* si los dos puntos introducidos son el mismo, y *false* en el caso contrario.
- **print():** imprime los valores de los atributos.

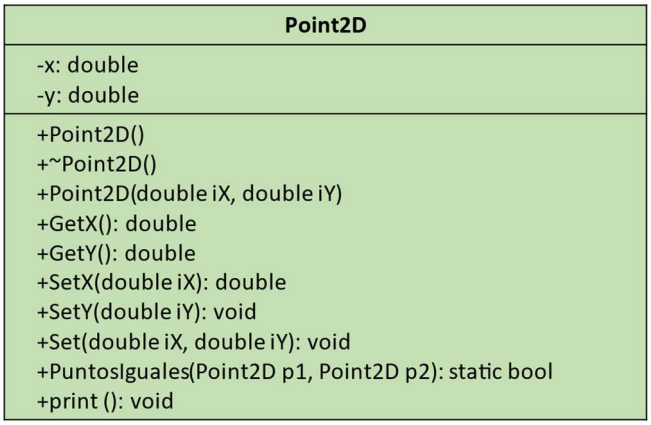


Figura 54. Clase Point2D UML.