Robótica

Representación y recorrido de Mapas Internos

Trabajo Práctico Final

Fecha de Entrega: 17/11/2016

—

Realizado por:

Fernando Ares

Diego Tabares

Índice

Problema a resolver

Se debe crear un sistema que permita representar mapas reales del interior de un edificio. Este sistema debe contemplar la creación, edición y exportación de estas representaciones. La solución debe poder representar los elementos más significativos que pueden encontrarse en un mapa interior tal como recintos entre los cuales podemos encontrar aulas, baños, laboratorios, etc., obstáculos como mesas, sillas, pizarrones, estufas, proyectores, etc.

El sistema debe permitir que cada recinto tenga una “Grilla” de tamaño variable, la cual representa la posición de las baldosas en el suelo.

Debe existir la posibilidad de definir trayectorias dentro del mapa, pudiendo iniciar en un recinto y terminar en otro, evitando los obstáculos definidos. Las mismas deben poder guardarse, cargarse y exportarse a formato texto.

Solución Propuesta

Para el desarrollo de esta solución se optó por trabajar en un lenguaje orientado a objetos debido a que el problema a resolver es complejo y este tipo de lenguajes facilita la estrategia y el diseño a aplicar.
El lenguaje elegido será Java, principalmente por la experiencia desarrollada con el mismo por parte de los integrantes del equipo de trabajo y además por ser un lenguaje de alto nivel que podría facilitar el manejo de estructuras de datos complejas y por su integración con diversas librerías externas.

Debido a que este desarrollo requiere de una contribución constante por parte de los integrantes del grupo, se optó por trabajar con Eclipse como entorno de desarrollo y GIT cómo versionador de código. Además, se utilizara GitHub como repositorio central, todas estas herramientas nos simplificarán el trabajo en equipo.

Otro factor determinante para la elección de Java fue la necesidad de utilización de una librería gráfica para poder representar el mapa. En lo que respecta a la misma, se evaluaron diferentes opciones existentes en el mercado actual como AWT, SWING y JAVAFX, resultando JAVAFX la librería seleccionada por su ventaja por sobre el resto para el manejo de interfaces gráficas a través de su capacidad de manipulación de archivos xml que nos permitirá optimizar el tiempo de desarrollo enfocándonos en el desarrollo de la solución y no en la interfaz en sí. Esto es posible a través de una herramienta integrada con Eclipse y JavaFX denominada Scene Builder.

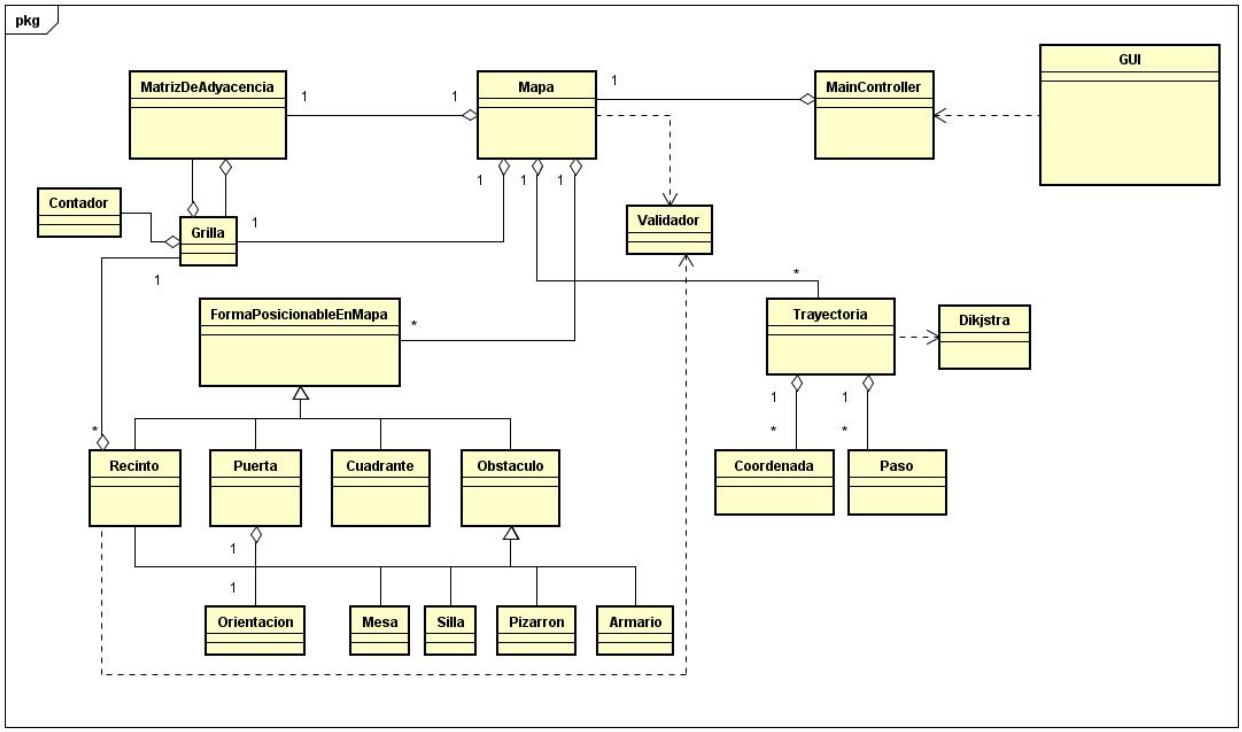
Para desarrollar esta solución, se creará un sistema con una interfaz gráfica sencilla e intuitiva que contenga diversos menus que permitan:

- Crear Mapas
- Borrar Mapas
- Editar mapas
    - Recintos
    - Obstáculos
    - Grillas
    - Puertas
- Calcular Trayectorias
- Guardar y cargar Mapas
- Guardar y cargar Trayectorias asociadas a un mapa
- Exportar el mapa a una imagen
- Exportar Trayectorias a texto

Para poder encarar el desarrollo de este proyecto se trabajó en aspectos de diseño con un diagrama simplificado de clases, lo que permitió poder pensar la solución y comenzar el desarrollo de manera iterativa.

Además se utilizaron herramientas de licenciamiento gratuito que faciliten aplicar metodologías ágiles para el desarrollo como Trello para el manejo de backlog de tareas.

Diagrama de clases



Solución Realizada

Alcance

El sistema fue diseñado para generar mapas interiores de un edificio convencional, esto quiere decir que no contempla las distintas situaciones que podrían generarse en un mapa exterior como calles, tránsito, o cualquier otro factor dado fuera de un edificio.
El sistema además fue pensado para poder generar trayectorias que permitan ser utilizadas por un sistema externo como por ejemplo un robot, pero no contempla como actuar frente a un evento como podría ser abrir una puerta.
Finalmente la solución se limita a generar mapas en 2 dimensiones, cualquier representación que implique una dimensión adicional como una pared, no fue contemplada en la misma. Sin embargo el diseño apuntó a que la solución permita agregar este tipo de requerimientos en el futuro.

Límites

Las grillas contemplan un offset en 2 paredes.
No será posible calcular trayectorias sin tener grillas definidas para cada uno de los recintos ya que la trayectoria depende de la generación de una matriz de adyacencia.
El sistema mostrará por pantalla sólo una trayectoria a la vez.
Las trayectorias generadas solo se calculan utilizando el camino más corto de Dikjstra, no existen métodos alternativos para generar trayectorias.
La forma de los recintos (y de los Mapas) siempre es rectangular, no se soporta otro tipo de formas al momento.
Para poder guardar y cargar trayectorias será necesario guardar previamente el mapa.

El código de la solución puede ser encontrado en github (https://github.com/dtabares/tpRobotica) el mismo puede ser descargado, redistribuído y modificado libremente ya que se encuentra bajo licencia MIT.

Elegimos una solución Orientada a objetos para resolver el problema. La clase **Mapa** contiene referencias a los **recintos**, que a su vez, contienen referencias a los **obstáculos**, **grillas** y **puertas**.
Todas estas colecciones son listas ordenadas (LinkedList) ya que necesitamos que se respete el orden en el que se fueron agregando para poder dibujar correctamente el mapa.

Todo objeto que se pueda posicionar en el mapa (**Cuadrante**, **Obstáculo**, **Recinto**, etc) hereda de la clase "**FormaPosicionableEnMapa**" la cual setea comportamientos y atributos comunes a todas ellas.

Los **Recintos**, representados mediante rectángulos (Clase Rectangle de JavaFX), tienen un nombre para poder ser identificados luego en distintos menus de la interfaz gráfica. Es importante destacar, que cuando se crea un nuevo recinto, una serie de validaciones (a cargo de la clase estática **Validador**) son realizadas, por ejemplo, que el recinto esté dentro del mapa y no se pise con otro recinto ya existente.

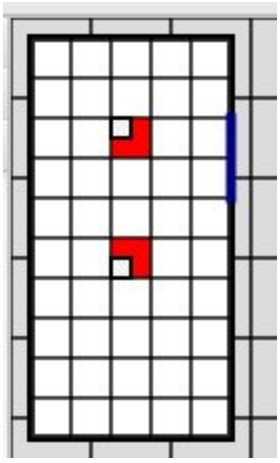
La **Grilla** se representa como una matriz de **Cuadrantes**. Para hacer que la **Grilla** sea “adaptable” se toma en consideración el tamaño deseado por el usuario para la misma y el tamaño (alto y ancho) del **Recinto**, adaptando de ser necesario alguna fila y/o columna dependiendo de la selección del vértice realizado por el usuario en la interfaz gráfica.
Al crear la Grilla se puede seleccionar el desplazamiento respecto de alguno de los 4 vértices, como se puede ver en la siguiente imagen:

Los **Cuadrantes** también se representan gráficamente mediante rectángulos , los cuales pueden cambiar de color dependiendo de la necesidad, por ejemplo: rojo para mostrar que el cuadrante no se encuentra disponible, verde para mostrar el destino en una trayectoria, etc.
Cada Cuadrante tiene un identificador local, que lo identifica dentro de la **Grilla** solamente, y otro global, que lo identifica unívocamente dentro del **Mapa** (este número no se repite).

El **Contador** se utiliza para llevar la cuenta de cuántos **Cuadrantes** fueron creados hasta el momento, ya que esto es importante para calcular la matriz de adyacencia global, es decir, la que representa todo el mapa, incluyendo los recintos.
El mismo está modelado como un Singleton, para que pueda llevar la cuenta global, sea quien sea la clase que lo invoque.
Cada vez que el usuario modifica una **Grilla** existente, este contador se resetea, volviendo a cero (0), y se reasignan los identificadores globales a cada cuadrante, ya que al realizar esa modificación se modificó también la Matriz de Adyacencia.

Los **Obstáculos** se representan también mediante rectángulos, y su forma es calculada en base a la selección del tipo de Obstáculo elegido por el usuario en la paleta de obstáculos de la interfaz gráfica.
Cuando un obstáculo se posiciona sobre uno o más **cuadrantes**, estos se marcan como no disponibles en la **Grilla**, por lo que el robot no podrá utilizar ese cuadrante para pasar.

Las **Puertas** fueron representadas como líneas, las mismas tienen más grosor que las paredes de los **recintos** y un color azul para poder distinguirlas fácilmente. Las puertas permiten entrar y salir de los recintos, siempre y cuando, no haya un obstáculo bloqueando el cuadrante inmediato a la puerta.


Ejemplo de recinto con 2 obstáculos y una puerta.

Para poder guardar y cargar mapas, hicimos que la clase **Mapa** y todos sus clases referenciadas implementen la interfaz "**Serializable**" que Java provee. Fue aquí donde nos encontramos con un problema difícil de solucionar, la clase **Shape** de JavaFX y todas las que heredan de ella (como por ejemplo **Rectangle**) no son serializables.
Sabiendo que mediante la información guardada en los atributos de cada una de nuestras clases, contábamos con todos los datos requeridos para poder volver a crear estas "**Shapes**", decidimos declarar todo atributo que fuera del tipo **Shape** como "**transient**" lo cual hace que este atributo no se serialize al momento de guardar el mapa. Al cargar un mapa, generamos una lógica que recorre todos los elementos del mismo, y regenera sus componentes gráficos, solucionando así el problema expuesto.

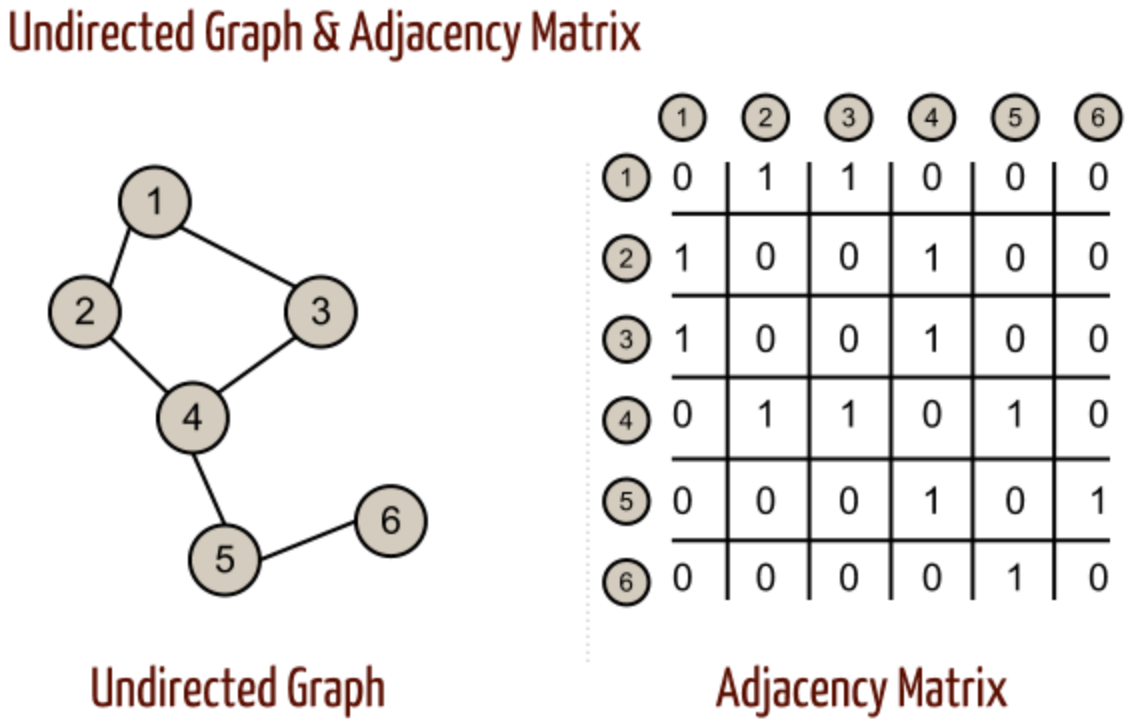
Definimos las **Trayectorias** como una secuencia de **Pasos**, los cuales guardan qué cantidad de baldosas se deben mover y en qué dirección. Para calcular la dirección se toma en cuenta en qué baldosa se encuentra actualmente, a cuál desea ir y la desviación respecto del norte que se setea inicialmente cuando se crea el **Mapa**. La **trayectoria** se representa coloreando de azul al cuadrante inicial, los intermedios de violeta, y el final de color verde.

Para calcular la **Trayectoria**, primero generamos la **matriz de adyacencia** de cada recinto a partir de la matriz que de cuadrantes de cada recinto, la cual contiene qué recintos están disponibles y cuales no. Llamaremos a esta matriz, "**Matriz de Adyacencia Local**". Cabe aclarar que también generamos una para la **Grilla** del **Mapa**, ya que debemos saber cuáles cuadrantes se encuentran libres en esa **Grilla** también.
Para poder generar la **matriz de adyacencia local**, se verifica la disponibilidad de la cada **Grilla**, marcando como no disponibles los **cuadrantes** ocupados por **obstáculos**.
La **matriz de adyacencia local**, es una matriz booleana cuadrada, y tiene como dimensiones n x n siendo n la cantidad de **cuadrantes** que posea la **Grilla** de ese **recinto**. Esta matriz se inicializa en falso y se marcan los cuadrantes conexos disponibles para cada **cuadrante** generando, por ejemplo:

	0	1	2	3
0	FALSE	TRUE	TRUE	TRUE
1	TRUE	FALSE	TRUE	TRUE
2	TRUE	TRUE	FALSE	TRUE
3	TRUE	TRUE	TRUE	FALSE

Nota: La diagonal principal siempre contiene “FALSE” ya que queremos evitar bucles en los nodos.

Gracias a la **matriz de adyacencia**, pudimos representar como **grafo** la **grilla** de cada **recinto**.



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Una vez generada las **matrices de adyacencia** de cada **recinto** y la **matriz de adyacencia** de la **Grilla** del **Mapa**, todas se insertan en una “**Matriz de Adyacencia Global**” que representa todo el universo recorrible. Esta matriz global tiene una dimensión de m x m, siendo m la cantidad global de **cuadrantes**, es decir, la suma de todos los **cuadrantes** de cada **recinto** más la cantidad de **cuadrantes** de la **grilla** del **mapa**. Finalmente, recorremos todas las **puertas**, y si tanto el **cuadrante** interior como exterior a la **puerta** se encuentran disponibles, los marcamos como conexos en la matriz global, no importa si estos dos **cuadrantes** pertenecen uno a un **recinto** y el otro a un segundo **recinto**, o bien, a la **grilla** del **mapa**.

Esta matriz global es utilizada por una clase que implementa el algoritmo de Dijkstra para devolvernos el camino más corto entre los puntos de inicio y fin seleccionados por el usuario. La salida del algoritmo (una lista de identificadores de cuadrantes) es procesada para transformar la lista de identificadores en una lista de pasos que puedan ser comprendidos por un robot, indicando la cantidad de baldosas a avanzar y en qué dirección en grados realizar el avance.

Cálculo de camino más corto para trayectorias

Para realizar este cálculo se evaluaron diferentes opciones entre las cuales se eligió implementar el algoritmo de Dijkstra para calcular el camino más corto. Gracias a la implementación de la matriz de

adyacencia global como representación de un grafo con todos los caminos posibles, el algoritmo se implementó de la siguiente manera:

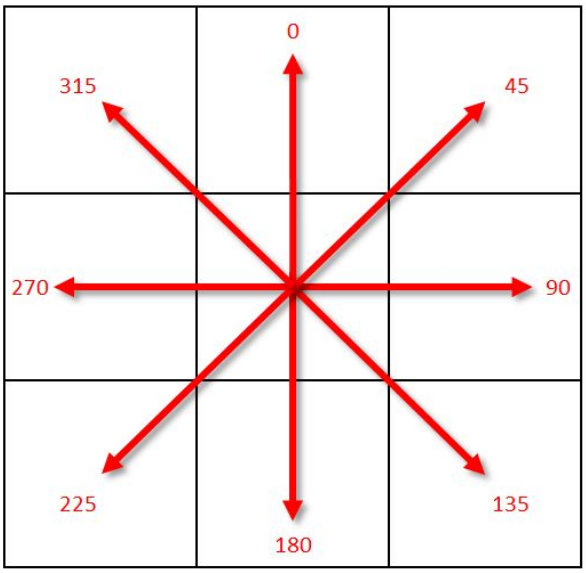
Se recorre la matriz de adyacencia comenzando por el id del cuadrante de origen y como objetivo el cuadrante destino. El algoritmo comienza a recorrer las columnas de esta matriz buscando los cuadrantes conexos al cuadrante posicionado, a medida que los encuentra los agrega a una cola y los marca como visitados, además guarda un registro para saber desde qué cuadrante se llegó a cada uno de ellos.
El algoritmo repite este procedimiento desencolando cada elemento previamente encolado, hasta encontrar el id del cuadrante destino. Una vez encontrado el destino se genera una lista partiendo del registro anteriormente mencionado que guarda cada movimiento utilizado, de esta manera se obtiene el camino más corto.

Un problema que encontramos con esta implementación es que por diseño el costo de moverse a cualquier cuadrante lindero al que estemos posicionados, tiene el mismo costo de movimiento, es por ello que en las trayectorias se puede visualizar que en determinados casos el camino, no es el más óptimo en distancia, aunque siempre es el más corto en cantidad de cuadrantes. Una posible solución a este problema sería implementar un cálculo de distancias durante la ejecución de este algoritmo, de manera que los elementos encolados estén ordenados por distancia. No se llegó a implementar esta mejora por cuestiones de tiempo.

Generación de Trayectorias paso a paso

Para poder generar trayectorias y exportarlas de manera de poder ser utilizadas con un sistema externo, se generó una representación del movimiento en ángulos estandarizados al movimiento de una baldosa, esto quiere decir que para poder avanzar debemos primero ajustar el rumbo en grados. Esta forma de calcular trayectorias, contempla la orientación del propio mapa la cual parte de una referencia al norte magnético.

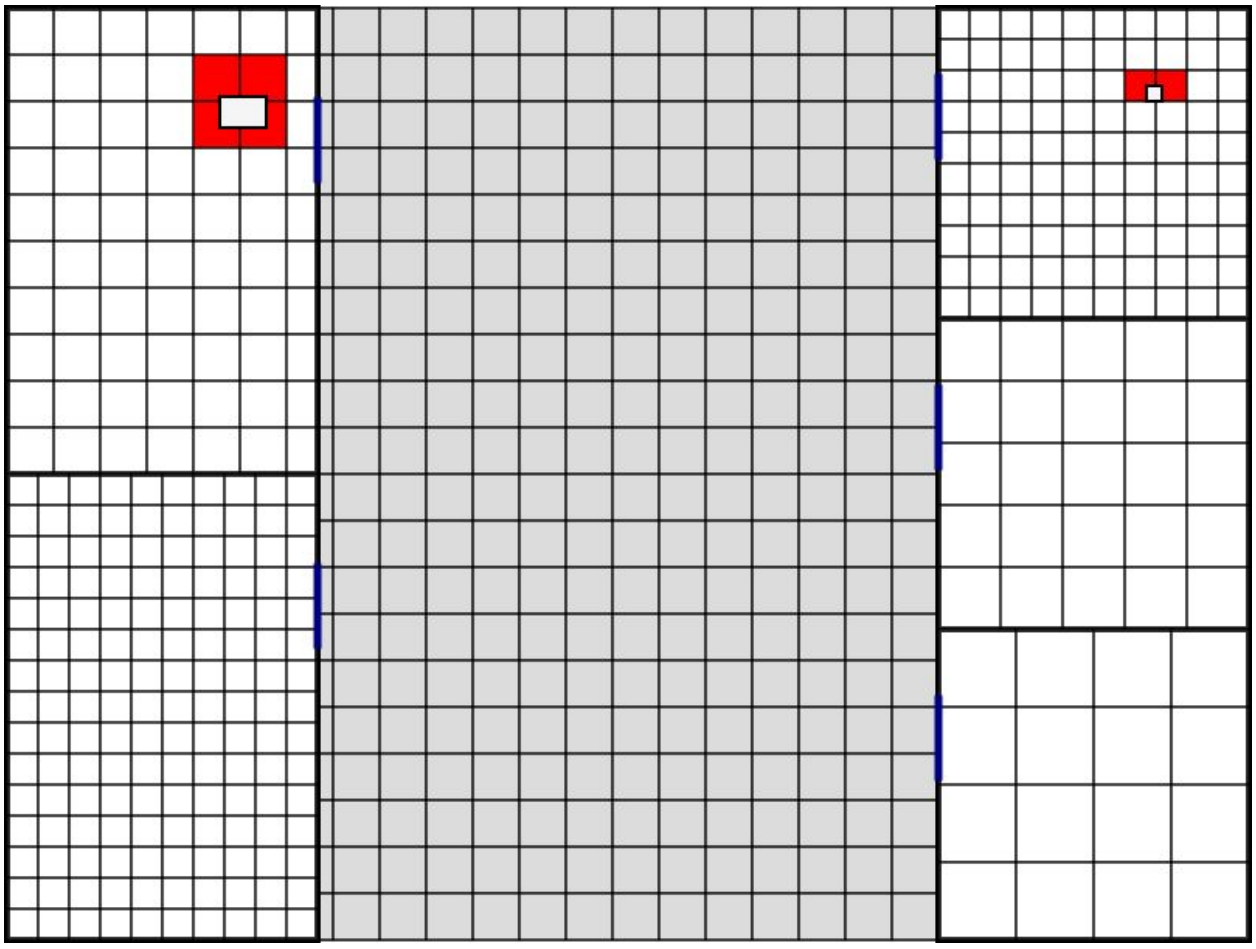
La siguiente figura muestra los ángulos de movimiento estándar con una desviación de cero grados respecto del norte magnético:



Pruebas Realizadas

1. Como prueba integral se diseñó un ejemplo similar al del piso 2 de la sede Caseros II de Untref. El mismo cuenta con los siguientes recintos:

Nombre de recinto, posición en (x,y), dimensiones, tamaño de grilla
Mapa: 800x600, 30
Taller de Sonido, (0,0), 200x300, 30
Taller de Informatica, (0,300), 200x300, 20
Taller de Aula 201, (600,0), 200x200, 20
Taller de Aula 202, (600,200), 200x200, 40
Taller de Aula 203, (600,400), 200x200, 50



2. Se agregó una puerta en cada recinto, y algunos obstáculos también

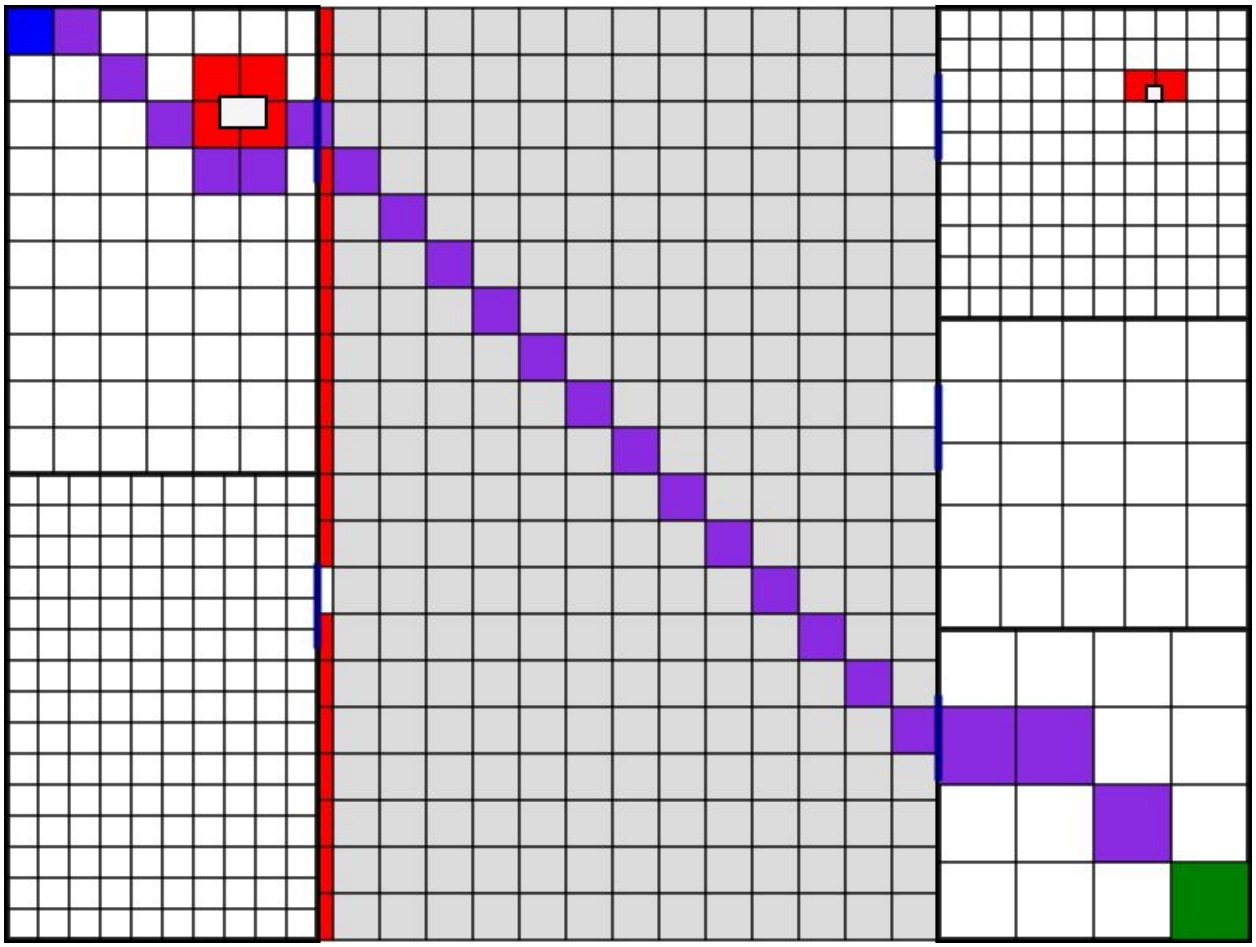
Puertas:
Taller de Sonido: posición (200,50), ancho 50
Taller de Informática: posición (200,350), ancho 50
Aula 201: posición (600,45), ancho 50

Aula 202: posición (600,245), ancho 50
Aula 203: posición (600,445), ancho 50

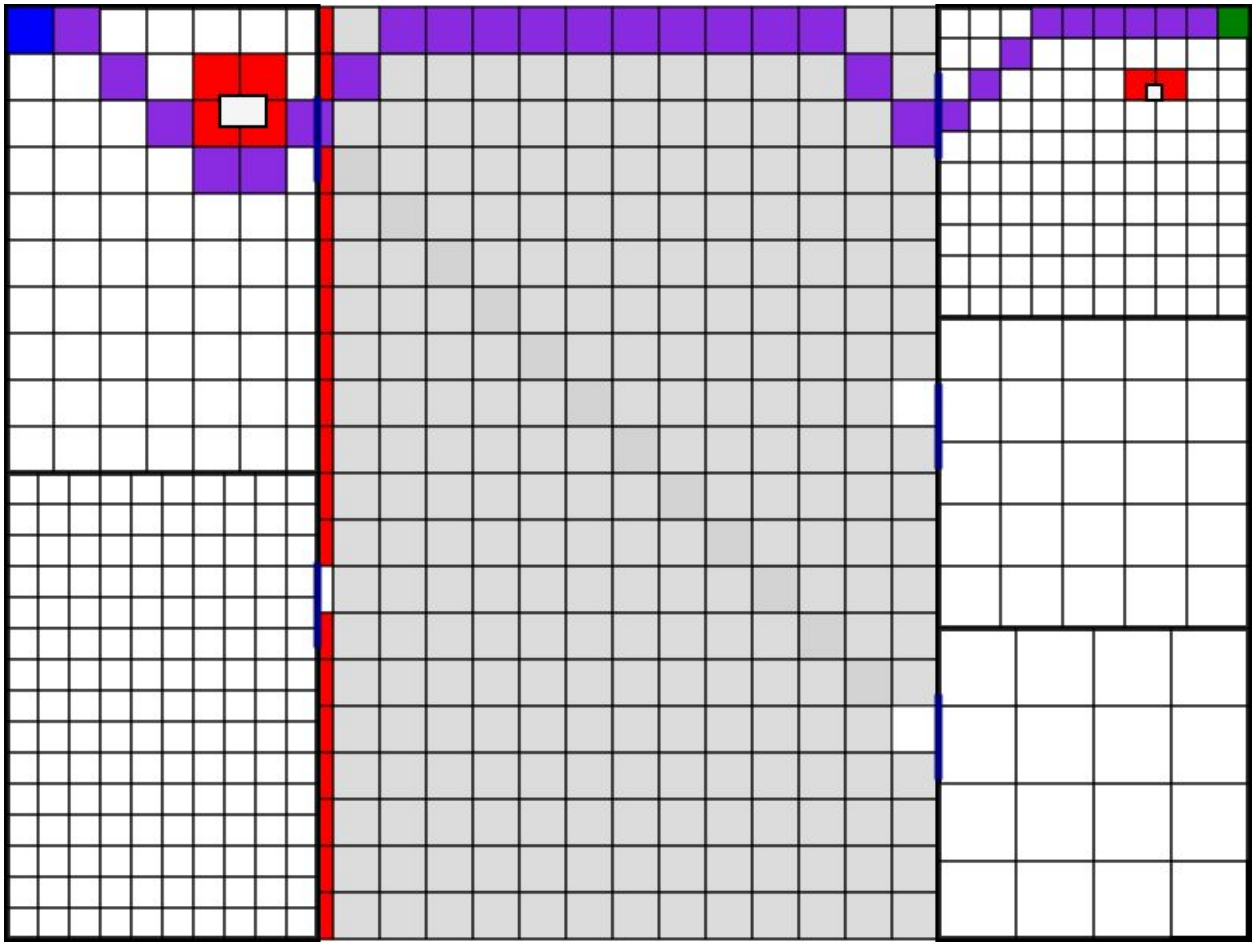
Obstaculos:
Mesa, recinto Taller de Sonido, posición (137,57)
Silla, recinto Aula 201, posición (734,50)

3. Se generaron distintas trayectorias entre recintos y dentro del mapa detalladas a continuacion:

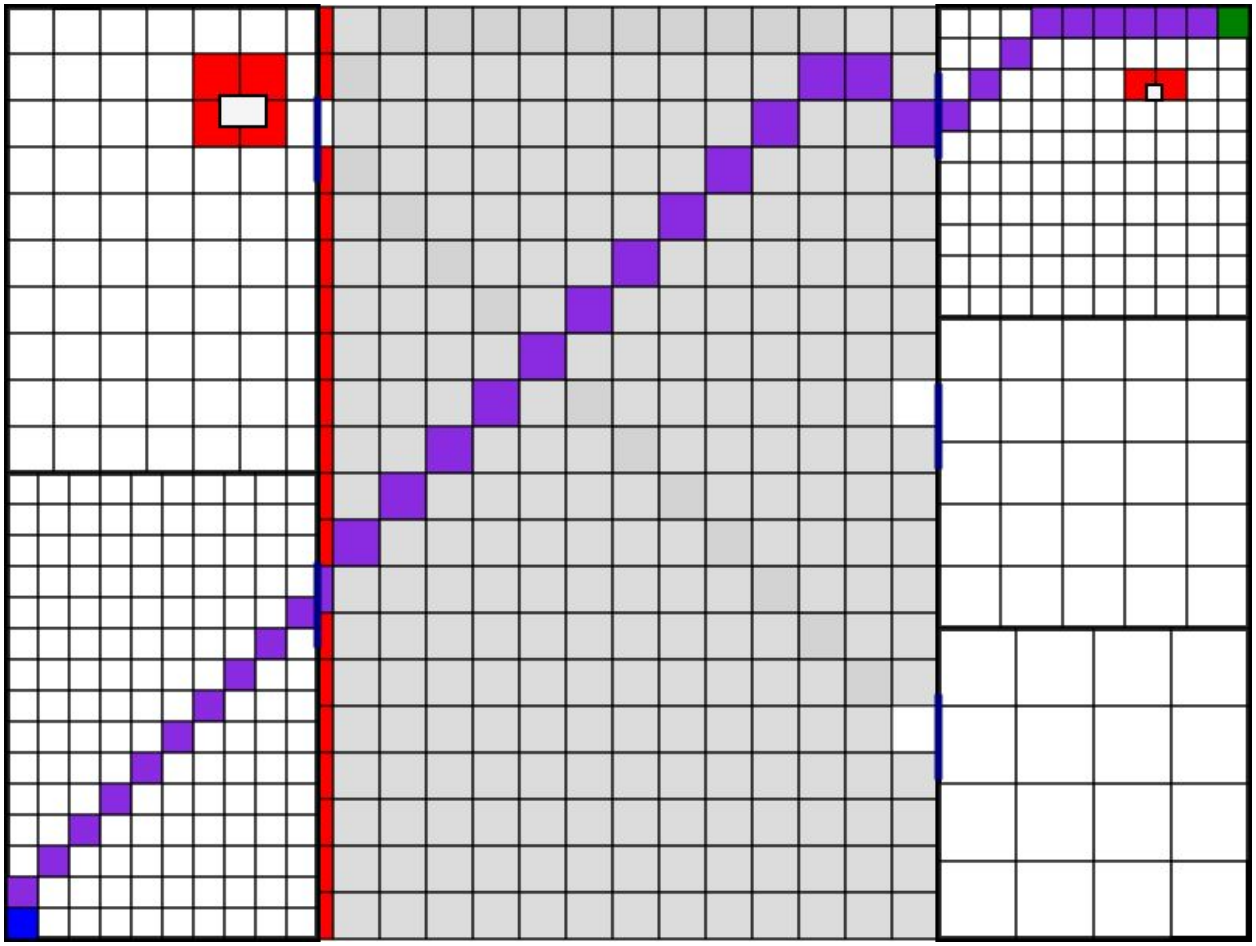
T1: (1,1) al (799,599)



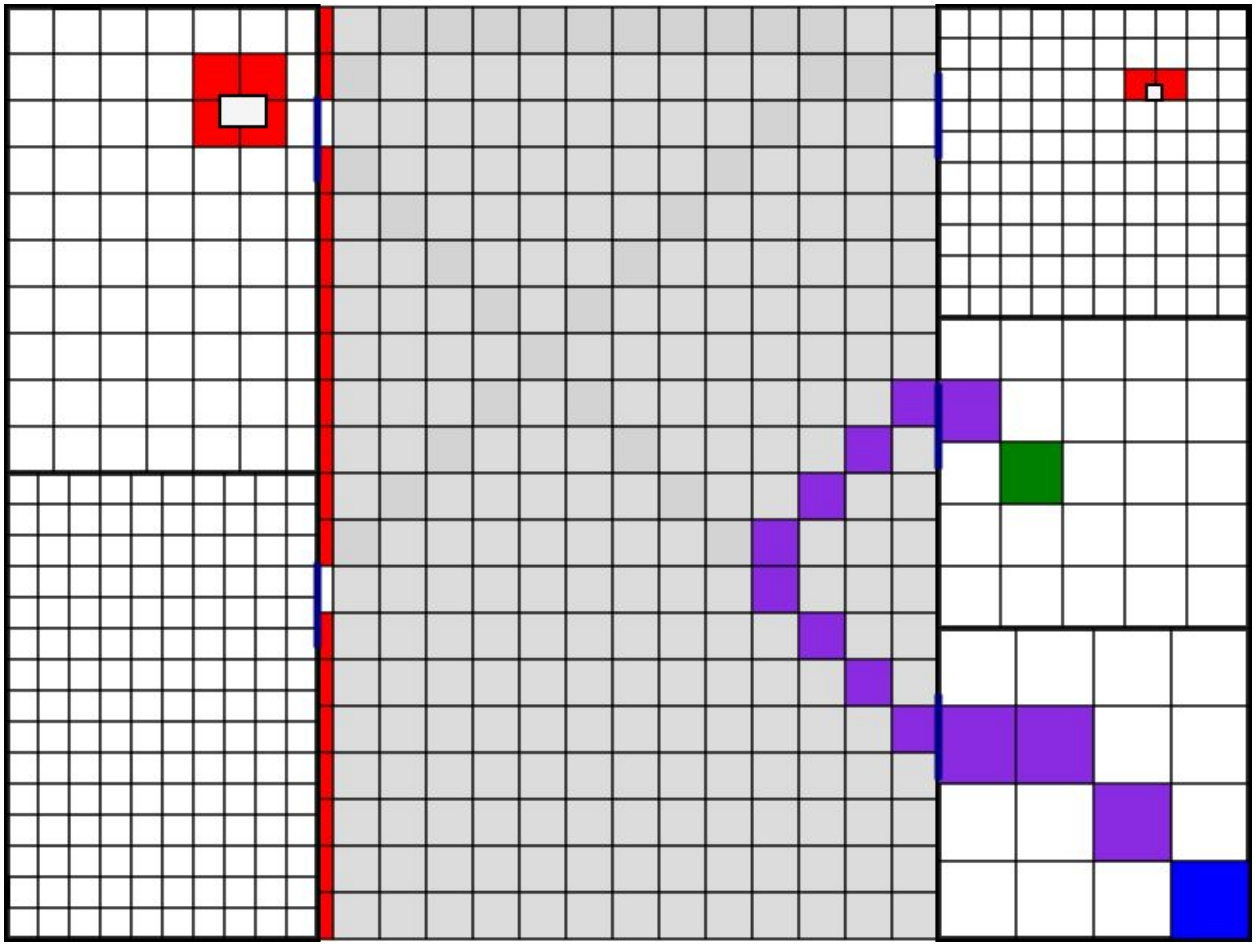
T2: (10,10) al (799,1)



T3: (20,599) al (799,1)



T4: (799,599), (650,300)



Conclusiones

El trabajo realizado fue de gran utilizad para entender diferentes alternativas y estrategias para encarar este tipo de problemas. Si bien el desarrollo fue enfocado en una aplicación puntual, interpretamos que la finalidad del mismo era la creación de un sistema base para poder representar y recorrer mapas que permitan reflejar la realidad de modo que un robot u otro sistema externo pueda utilizarlos para moverse en estos espacios.

Como hemos mencionado en la sección “Solución Realizada” uno de los principales desafíos que encontramos fue en diseñar una solución que contemple futuras funcionalidades, ya que en principio la aplicación soporta únicamente dos de las tres dimensiones que conforman el espacio del mundo real. Además, es difícil poder predecir qué tipo de sistema puede ser desarrollado para recorrer este mapa, ya sea por el amplio espectro de tecnología que puede llegar a aplicarse desde sensores de ultrasonido hasta reconocimiento de imágenes.

Lo que intentamos lograr con el trabajo fue abstraernos de este tipo de cuestiones y desarrollar una solución que no esté limitada al plano de dos dimensiones, sino que a través de la programación orientada a objetos, pueden ser suficientemente flexible para incorporar nuevas funcionalidades.

Otro de los desafíos con los cuales nos encontramos fueron las distintas estrategias aplicables para generar las trayectorias, existen variados tipos de algoritmos que pueden solucionar este problema y que dependen de ciertas situaciones con las que un sistema externo podría llegar a encontrarse, algo que nos resultó difícil de predecir pero que podría ser ajustado sin problema en un futuro.

También nos encontramos con distintos problemas estrictamente de diseño que hemos tenido que ajustar a lo largo del proyecto, una de ellas tiene que ver con la estrategia aplicada para encontrar el camino más corto, ya que, como se puede distinguir en la sección “Pruebas Realizadas”, nuestra estrategia fue orientada en cantidad de pasos y no en distancia, con las trayectorias más cortas en cantidad de pasos no siempre son las más cortas en distancia. Este tipo de limitaciones puede ser ajustada dependiendo de la estrategia requerida por quien vaya a utilizar esta solución.

Para terminar, el trabajo contribuyó a familiarizarse con distintas herramientas que no habían sido utilizadas hasta el momento como es el caso de librerías gráficas e interfaces de usuario.

Bibliografía:

- Java 8 (Documentación Oficial): http://docs.oracle.com/javase/8/javase-clienttechnologies.htm
- JavaFX (Documentación Oficial): https://docs.oracle.com/javase/8/docs/
- Scene Builder (Guía de Usuario): http://docs.oracle.com/javafx/scenebuilder/1/user_guide/jsbpub-user_guide.htm
- Shortest-Path Algorithms: https://www.mcs.anl.gov/~itf/dbpp/text/node35.html