20 DE ABRIL DE 2021

IMPLEMENTACIÓN DEL ALGORITMO A*

PRÁCTICA 1 – INGENIERÍA DEL CONOCIMIENTO

VITALIY SAVCHENKO
CARLOS LUENGO HERAS
VICTOR VELASCO ARJONA

ÍNDICE

1. Algoritmo: En esta primera parte de la memoria trataremos de exponer la lógica detrás de nuestro algoritmo, como lo hemos implementado y que hemos utilizado para para ello.

Nuestro algoritmo parte del concepto de **nodo**, actúa sobre un conjunto de nodos al que llamaremos **tablero**, que realmente es una matriz de nodos.

Un nodo guarda información como la posición (i,j) de nuestro mapa, un id, único para cada nodo, un coste para llegar a ese nodo y un booleano que nos sirve para saber si es alcanzable o no.

```
public class Nodo implements Comparable<Nodo> {

    private int i;
    private int j;
    private int id;
    private Integer coste;
    private boolean descartado;
    private Nodo anterior;
```

Una vez entendido que como es nuestro tablero, podemos hablar de la lógica que hemos implementado para hacer funcionar nuestra versión del algoritmo A*:

Tenemos una lista abierta donde guardamos los nodos alcanzables, de tipo Set para evitar que haya nodos repetidos, y una lista cerrada de tipo HashMap donde priorizamos una eficiencia y optimización del coste del algoritmo.

Además, tenemos un ini y un fin donde registramos el punto de partida y el punto destino para el algoritmo.

Los costes diagonal y recto se usan para la evaluación heurística.

```
public class AlgoritmoA {

    public static final int costediagonal = 15;
    public static final int costerecto = 10;
    private Nodo[][] tablero;
    private Set<Nodo> listaAbierta;
    private HashMap<Integer, Nodo> listaCerrada;
    private Pair ini;
    private Pair fin;
```

Los obstáculos se guardan directamente en la lista cerrada, puesto que no podemos acceder a ellos. En la sección 2.c) es donde tratamos las ampliaciones realizadas, ahí hablaremos de como hemos implementado los way points y las celdas peligrosas.

Una vez entendida la estructura lógica el algoritmo es muy sencillo, tras dar dimensiones al tablero, valores al inicio y al final, además de posibles obstáculos, way points y celdas peligrosas, comienza el algoritmo (pulsando Empezar). Esto desencadena una serie de llamadas (patrón modelo-vista-controlador) que acaba ejecutando el algoritmo, añadiendo a la lista abierta las celdas accesibles y calculando mediante la función heurística la coordenada accesible de menor coste. Cada celda recorrida se mandará a la lista cerrada y se registrará en una lista las celdas componentes del camino solución, que se mostrarán al usuario gráficamente al acabar el algoritmo mediante pintado del tablero.

2. Metodología aplicada: En esta segunda parte de la memoria trataremos de describir los detalles propios de nuestra implementación, empezando por el lenguaje que hemos utilizado, continuando con el diseño del nuestro código y otros patrones utilizados para estructurar su funcionamiento, y finalmente revisando las ampliaciones realizadas.

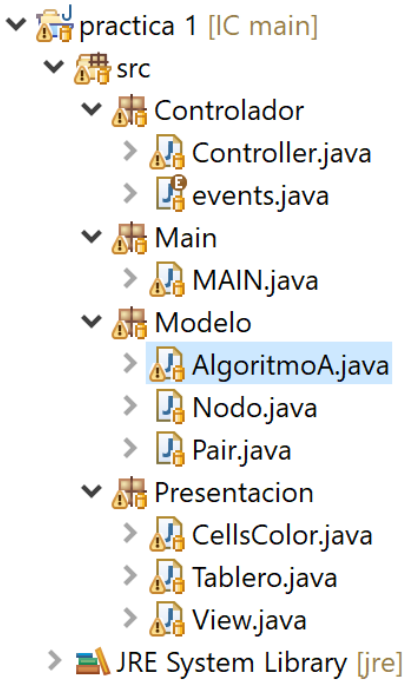
a) Lenguaje utilizado:

Para realizar esta práctica hemos decidido que la mejor opción sería utilizar Java. Nos ha parecido la mejor opción al ser una aplicación de escritorio, aunque estuvimos tanteando otras opciones como HTML-CSS y hacer una aplicación web.

Por lo tanto, el código se ha desarrollado utilizando un repositorio de GitHub y de editor la herramienta Eclipse. Además, aprovechando la versatilidad de Java y los paquetes de sus proyectos, hemos decidido utilizar el patrón modelo vista controlador, aunque ya lo explicaremos en detalle en el punto siguiente.

b) Diseño del código:

Comenzamos con el mítico patrón modelo-vista-controlador, de esta forma tenemos 3 paquetes principales:



La aplicación se estructurará así:

Controlador, con el propio Controller que se crea desde el Main y genera la primera instancia de la aplicación, la vista.

A su vez la vista, a través de los botones, llama al controlador que llama a Modelo. Modelo realizará el algoritmo a través de los datos obtenidos en la vista.

Modelo devuelve la solución a la vista, que a través de CellsColor pinta la solución de Tablero.

Otro patrón que utilizamos es el patrón builder. Al comenzar la aplicación, mediante la vista, llamamos al controlador que llama al modelo para cada inserción de datos. Es decir, le vamos pasando al Algoritmo dato a dato el Ini, el fin, obstáculos... construye el algoritmo modularmente.

c) Ampliaciones realizadas:

A parte de la funcionalidad básica de “dado un Inicio, un Fin, y unos obstáculos, hacemos el recorrido óptimo...” hemos realizado un par de ampliaciones más:

1. *Realizar un recorrido por distintas posiciones predeterminadas denominadas “way points”*

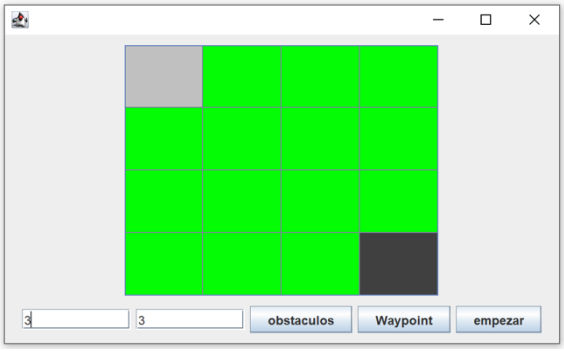
Para realizar esta parte hemos creado un nuevo botón en la vista que permita añadir puntos por los que queremos que obligatoriamente pase nuestro camino solución.

Además, este nuevo botón solo aparece cuando hemos establecido el Fin.

En cuanto a la lógica seguimos utilizando el mismo algoritmo pero lo hacemos secuencialmente. Tenemos una nueva lista, la lista de las “Metas”, en la que se añade en primer lugar el Fin y luego 1 a 1 los way points. Por lo tanto, ahora se va realizando el algoritmo recorriendo la lista de metas desde el final (último way point añadido) hasta el principio (el Fin original). Cuando termina 1 de las múltiples iteraciones o secuencias de way point introducidos, reinicia los costes.

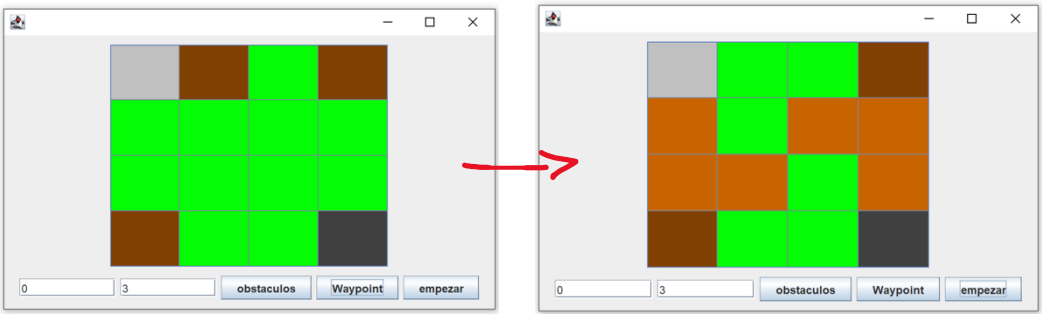
```
public class AlgoritmoA {

    public static final int costediagonal = 15;
    public static final int costerecto = 10;
    private Nodo[][] tablero;
    private Set<Nodo> listaAbierta;
    private HashMap<Integer, Nodo> listaCerrada;
    private Pair ini;
    private List<Pair> metas;
    private List<Nodo> solucion;
```

Ejemplo:



Aquí tenemos el campo de césped con el inicio, en gris, y el fin, en negro, ya añadidos. Ahora nos deja añadir way points.

Añado, por ejemplo, que quiero que pase por el (0,3) y por el (3,0). Así hará un camino marrón en N inversa.

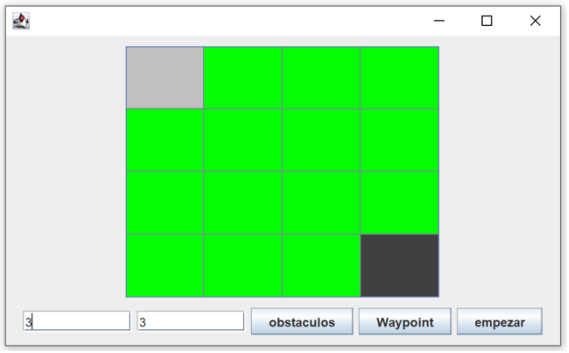


2. ***Suponer que determinadas celdas son relativamente peligrosas, pero que pueden atravesarse asumiendo un cierto riesgo. En este caso es necesario introducir un factor de corrección en la función de evaluación heurística.***

Para realizar esta parte hemos creado un nuevo botón en la vista que permita seleccionar celdas transitables, pero con un mayor coste que otras celdas que no serán celdas peligrosas.

En cuanto a la lógica, hemos aumentado el coste de la función heurísitca para que solo pase por estas celdas cuando sea necesario, es decir, la lógica de la aplicación sigue siendo la misma en lo referente a listas abiertas y cerradas, simplemente el coste de pasar a una de estas celdas es mayor que el de pasar a una celda no peligrosa.

Ejemplo:



Aquí tenemos el campo de césped con el inicio, en gris, y el fin, en negro, ya añadidos. Ahora nos deja añadir zonas peligrosas.

Añado, por ejemplo, que quiero que entre la ruta óptima haya 2 zonas peligrosas, para que el coste ya no sea el mejor en el camino diagonal.

En amarillo quedan reflejadas las nuevas celdas peligrosas, es decir, las diagonales: (1,1) y (2,2).

Ahora, sale barato ir por otro lado.

El **resultado** es el esperado:

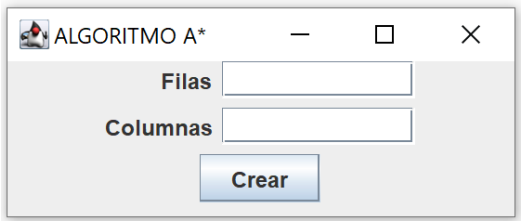
Ha elegido el camino marrón por el lado izquierdo, aunque habría sido también el mismo coste el camino del lado derecho.

Funciona correctamente.

3. Manual de usuario: En esta parte de la memoria haremos un pequeño recorrido por la funcionalidad de la aplicación, a modo introducción a la aplicación.

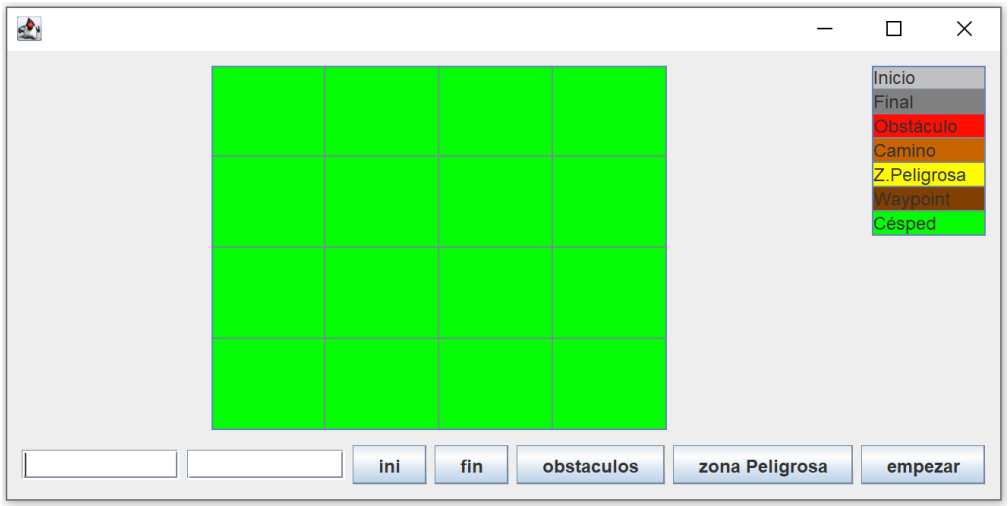
1- Creación del tablero:

Nada más compilar la aplicación por primera vez nos preguntará las dimensiones del tablero.



2- Interfaz de usuario:

Aquí podremos ver el tablero virgen, sin ningún posicionamiento previo.
El color verde muestra un suelo ficticio de, por ejemplo, plantas, evocando a un sendero o bosque sin aún recorrer.

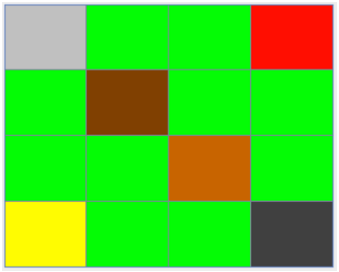


Debemos rellenar las dos componentes (columna,fila) y luego pulsar el botón de posicionado que queremos. Debemos de tener en cuenta que Ini y Fin deben ser únicos, por lo que desaparecerá el botón una vez establecidos. Al pulsar Fin dejará seleccionar “way points”

3- Coloreado

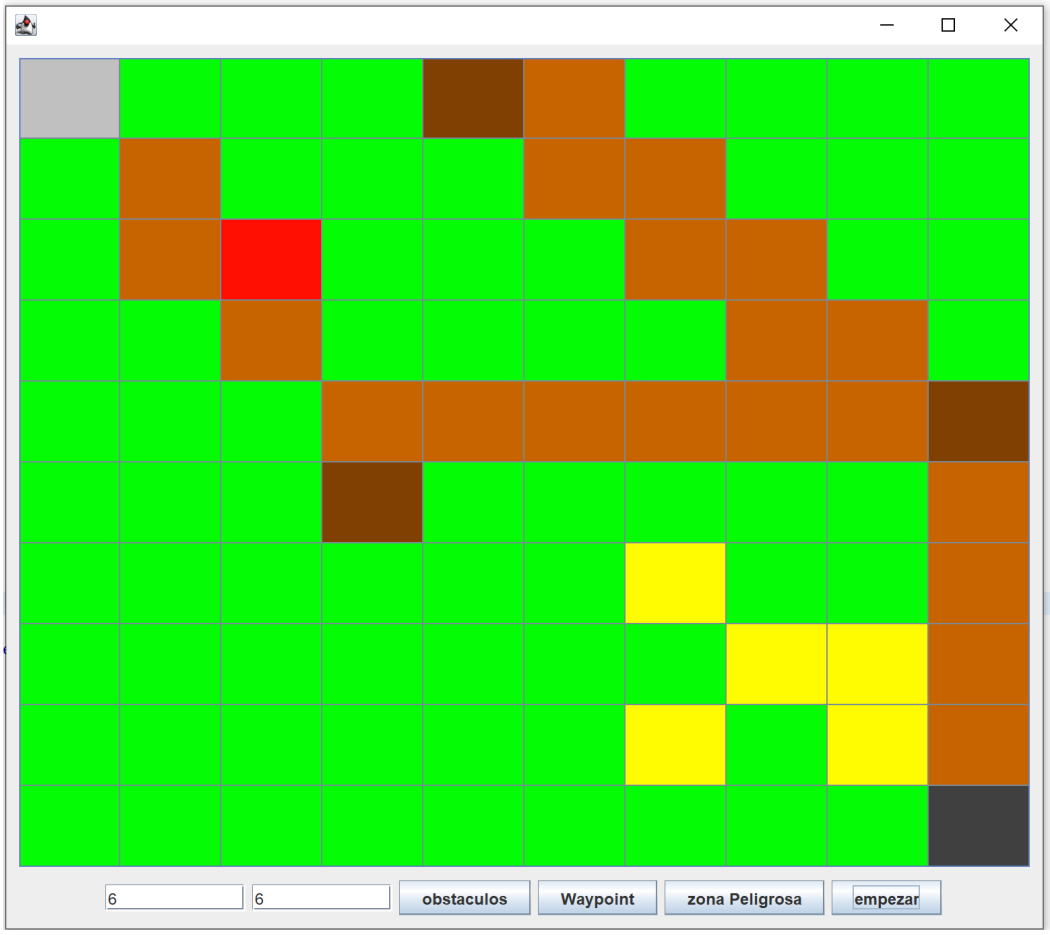
Según vamos añadiendo puntos podemos ver que se irá coloreando el tablero, con distintos colores para el Inicio, Fin, Obstaculos, Waypoints y Zonas peligrosas:

Verde significa que no hay nada, es el sendero virgen.
Gris indica el Inicio.
Negro indicia el Fin, el destino.
Marrón será el camino finalmente recorrido.
Rojo serán obstáculos por los que no se pueden pasar.
Amarillo serán Celdas Peligrosas, con mayor coste.
Por último, un marrón, aunque más oscuro, serán los “way points”.



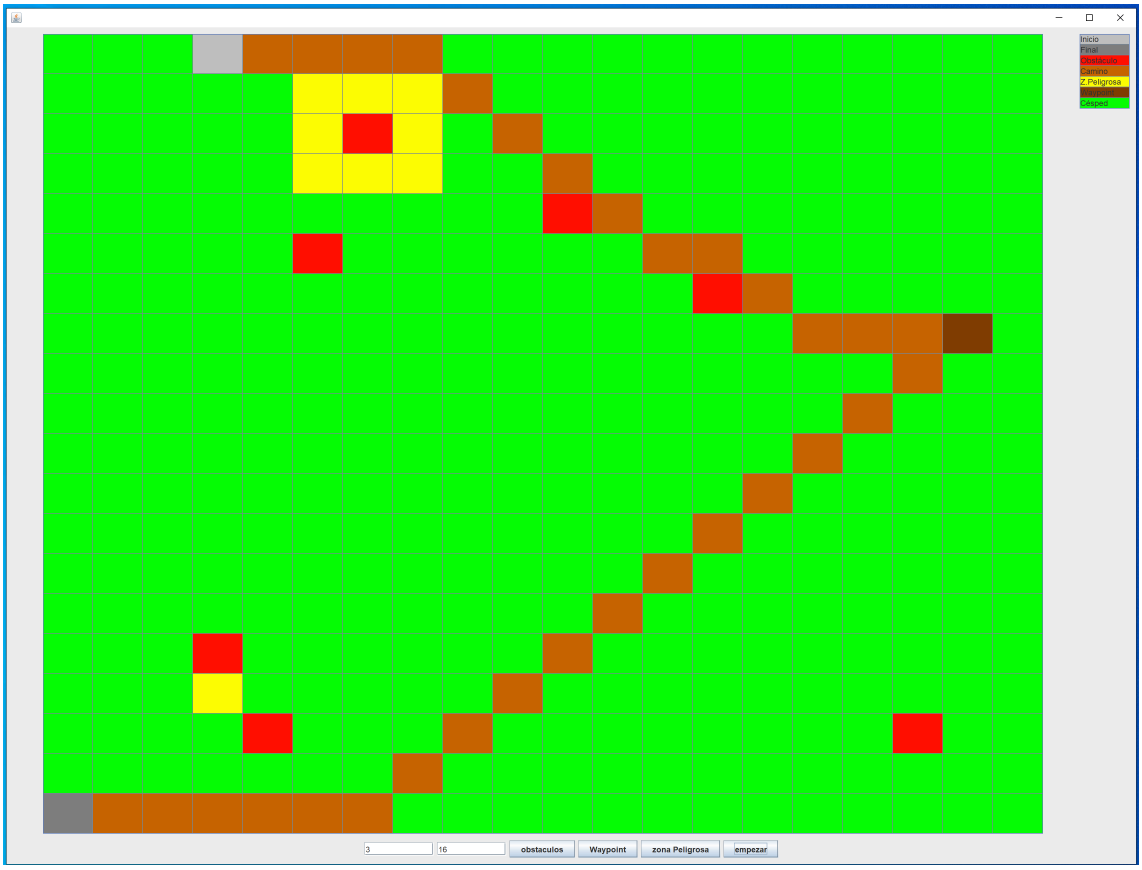
4- Resultado

Finalmente, al pulsar a Empezar, nuestro algoritmo hará su cálculo y devolverá la solución gráficamente pintando todas las celdas por las que pasa la solución.

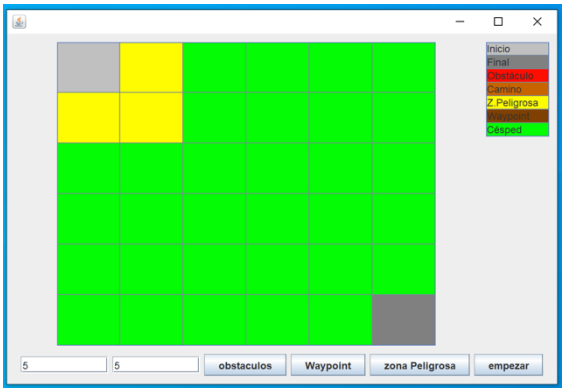


4. Ejemplos de compilación: En esta parte de la memoria hemos recopilado ejemplos de casos de uso para demostrar que la aplicación funciona correctamente sin necesidad de hacer pruebas recurrentes.

1- Un ejemplo básico donde obligamos que vaya de un extremo a otro pasando por un way point entre medias, pero enfrentándose a obstáculos que evita con zonas peligrosas.

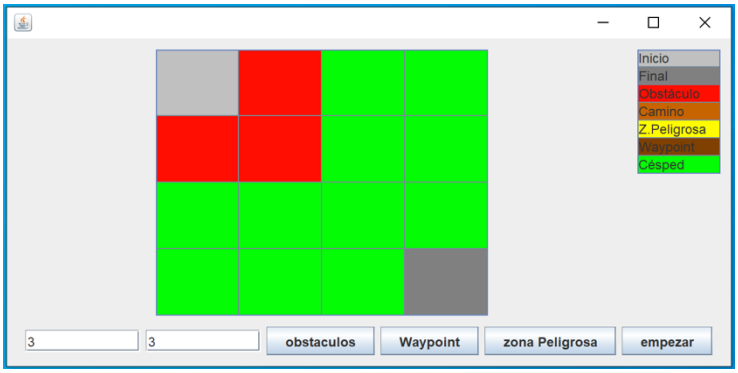


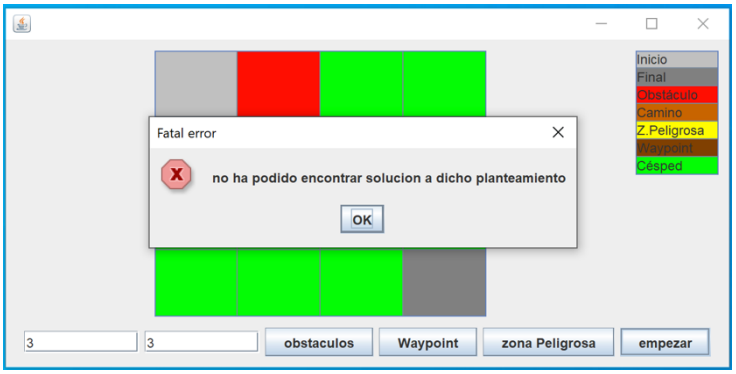
2- Por aquí, otro ejemplo. En este caso, el camino queda bloqueado por zonas peligrosas pero necesarias para llegar al Fin.



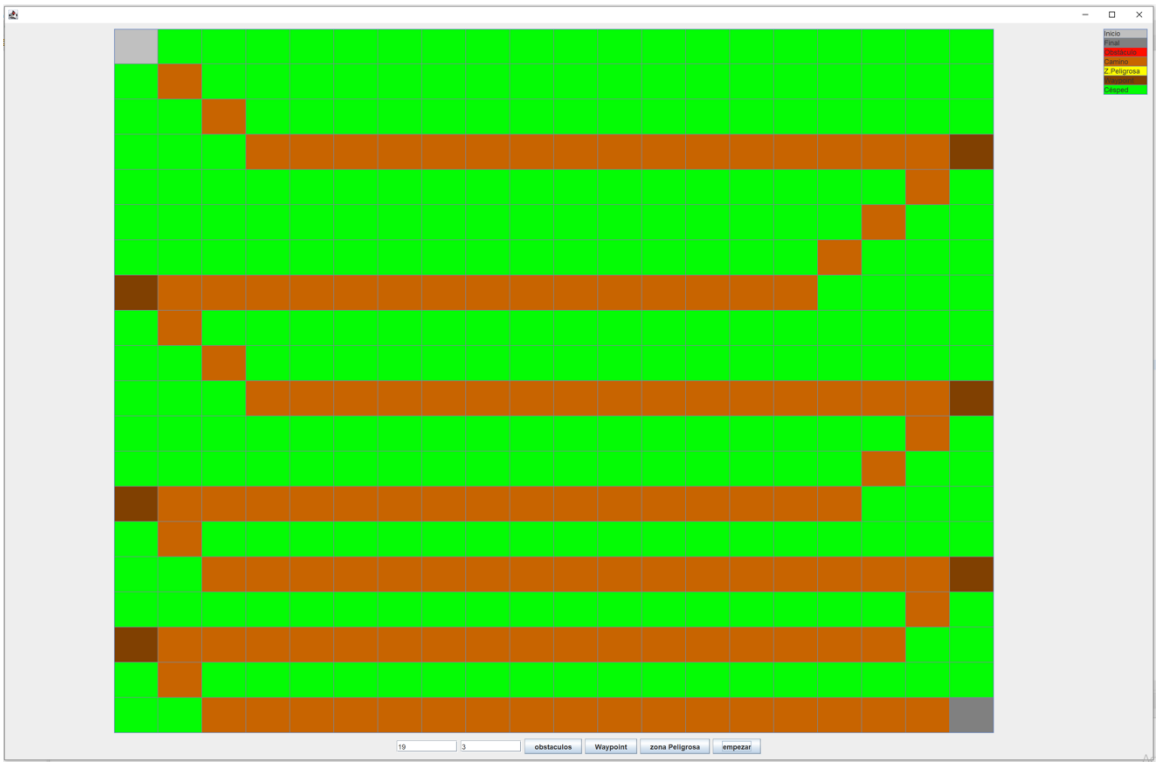
En esta situación, verificamos que ante necesidad sí que puede pasar por una zona peligrosa.

3- Otro ejemplo para verificar que en ocasiones nuestro programa no puede encontrar un camino válido entre dos puntos Ini y Fin. En este caso, rodeamos de obstáculos el Ini de tal manera que no haya manera de sortearlos.



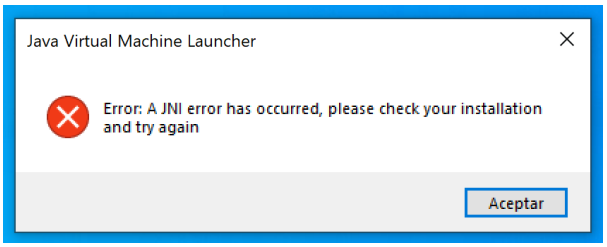


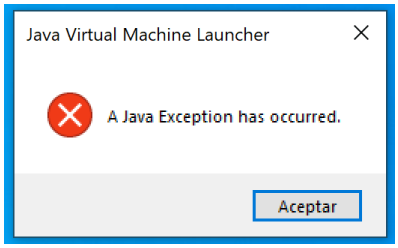
4- En este ejemplo probaremos a colocar varios way points por los que obligar a pasar nuestro camino solución.



5. Resolución de problemas: En esta parte final de la memoria hemos recopilado una pequeña guía para solucionar un problema con la versión de Java al inicializar nuestro ejecutable. El problema se puede presentar tanto al abrir el .jar.

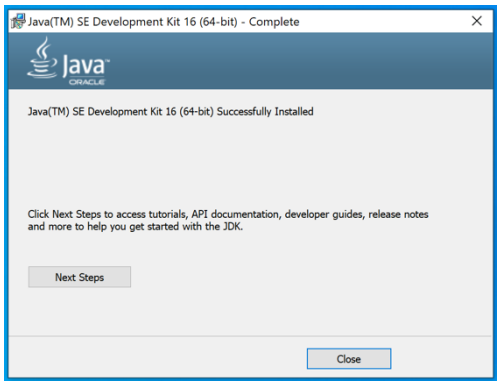
El problema se origina al intentar ejecutar el archivo. Debería salir algo así:





Para solucionarlo, debemos instalar una versión de Java más reciente.
En este caso, la podremos descargar desde el siguiente enlace de la web oficial de Oracle:

https://www.oracle.com/java/technologies/javase-jdk16-downloads.html



Una vez instalado, podemos ejecutar el .jar correctamente.