



UNIVERSIDAD DE GRANADA

GRADO EN INGENIERÍA INFORMÁTICA

DESARROLLO BASADO EN AGENTES
CURSO 2023 / 2024

Memoria – Práctica 2

Movimiento de una agente en un mundo bidimensional

Cano Flores, Julia María
Muñoz Gómez, Sergio
Reyes García, Teresa Fernanda
Rincón Otero, Marta
Velázquez Ortuño, Diego

21 de noviembre, 2023

ÍNDICE

1. Planificación del proyecto.	2
2. Estructura del proyecto.	4
Diagrama de clases	5
3. Algoritmo seguido por el agente (RTA*)	6
4. Interfaz Gráfica	7
Capturas de pantalla de la interfaz del programa.	8

Para el desarrollo del proyecto hemos hecho uso de GitHub como plataforma para la gestión grupal de nuestro código ya que es una plataforma muy cómoda para la colaboración en equipo. Hemos hecho uso de diferentes ramas para las funcionalidades del proyecto como es el entorno, mapa, agente, etc. Además, hemos usado un README para hacer una descripción breve del proyecto.

- **GitHub:** <https://github.com/teresitarg22/Desarrollo-Agentes>
- **ID_GRUPO:** 206

1. Planificación del proyecto.

El **Product Backlog** es una lista priorizada que contiene todas las funcionalidades y tareas que necesitamos para la creación de nuestro producto final. Esta lista nos permite organizar y planificar de manera eficiente todo lo que debemos implementar en el proyecto, lo que resulta fundamental para una gestión efectiva del tiempo que disponemos.

PRODUCT BACKLOG			
1	Diseño inicial del proyecto.		
	División en tareas.		
	ID	Descripción	Estimación
	1.1	Diseño conceptual de la estructura de clases.	2h
2	1.2	Reparto y asignación de tareas.	30min
	Todos		
	Clase 'Mapa'.		
	División en tareas.		
	ID	Descripción	Estimación
3	2.1	Creación de 'Mapa' que carga y gestiona el mapa.	30min
	2.2	Implementar la lectura del mapa desde un archivo texto.	1.5h
	2.3	Implementación de los métodos de la clase 'Mapa'	1.5h
	Marta		
	Clase 'Entorno' y sensores.		
	División en tareas.		
	ID	Descripción	Estimación
	3.1	Creación de 'Entorno' para la información del entorno.	30min
	3.2	Implementar los sensores.	1h
	3.3	Implementación de los métodos de la clase 'Entorno'	2h
	3.4	Corrección de errores y mejora de implementación.	1.5h
Todos			

4	Clase 'Agente' y comportamientos.		
	División en tareas.		
	ID	Descripción	Estimación
	4.1	Creación de la clase 'Agente' que maneja las acciones.	40min
	4.2	Creación de las clases de los comportamientos, interfaces y enumerado.	2h
	4.3	Implementación del comportamiento 'Visión del entorno'.	30min
	4.4	Implementación del comportamiento 'Toma de decisión'.	3h
	4.5	Implementación del comportamiento 'Mover'.	1h
5	4.6	Corrección de errores y mejora de implementación.	2h
	Interfaz de visualización del mapa y agente.		
	División en tareas.		
	ID	Descripción	Estimación
	5.1	Boceto de diseño de la interfaz.	2h
	5.2	Creación e implementación de interfaz de visualización.	4h
	5.3	Corrección de errores y mejora de implementación.	2h
	Pruebas y documentación.		
6	División en tareas.		
	ID	Descripción	Estimación
	6.1	Realización de pruebas de funcionamiento y mejora.	4h
	6.2	Redacción de la memoria.	2.5h
	6.3	Creación de la presentación del proyecto.	3h

2. Estructura del proyecto.

Contamos con tres elementos principales en nuestro proyecto: el agente, el entorno y la vista. Para la clase **agente** hemos implementado tres comportamientos específicos que son la visión del entorno, la toma de decisión del siguiente movimiento y la acción de moverse según el movimiento calculado anteriormente. Por otro lado, la clase **entorno** proporciona una representación del entorno en el que el agente realiza sus acciones en el mundo bidimensional. Cuenta con variables como los sensores, que almacenan información sobre las celdas adyacentes al agente, el mapa o las posiciones del agente y el objetivo. Finalmente, hemos implementado una **interfaz gráfica** que muestra el comportamiento del agente ante los diferentes mapas.

Hemos hecho uso de **interfaces** para escuchar eventos relacionados con la interfaz gráfica. Contamos con un listener para el entorno y otro para el main. Por otro lado, para la gestión de los posibles movimientos disponibles para el agente (contando con diagonales), en un entorno bidimensional, hemos hecho uso de un **enumerado**. El **directorio de mapas** contiene archivos de texto que representan diferentes configuraciones de mapas, desde entornos sin obstáculos hasta entornos con obstáculos complejos.

- **Agente**
 - main.java
 - AgenteMundo2D.java
 - Entorno.java
 - Mapa.java
 - MapaGUI.java

 - EntornoListener.java
 - MainListener.java
 - PosiblesMovimientos.java
- **Comportamientos**
 - Vision.java
 - TomaDecision.java
 - Mover.java
- **Img**
 - agente.png
 - objetivo.png
 - punto.png
 - seleccion.png
- **Mapas**
 - mapWithComplexObstacle1.txt
 - mapWithComplexObstacle2.txt
 - mapWithComplexObstacle3.txt
 - mapWithDiagonalWall.txt
 - mapWithHorizontalWall.txt
 - mapWithoutObstacle.txt
 - mapWithVerticalWall.txt

El **diagrama de clases** es una herramienta que nos va a permitir representar visualmente la estructura de clases que hemos seguido en nuestro proyecto, identificando sus clases (y cómo las clases se relacionan entre ellas), métodos y atributos.



3. Algoritmo seguido por el agente (RTA*)

Al encontrarnos en un entorno en el que **el agente es reactivo** y desconoce el mapa, necesitaremos usar un algoritmo de búsqueda en tiempo real basado en A* (*Real-Time A**). El algoritmo A* se basa en tener una función de evaluación que devuelva la heurística desde una posición dada para usarla al tomar la decisión del trayecto a seguir.

En nuestro caso, el agente no conocerá ese trayecto, pues es reactivo, sino que calculará para su campo de visión la heurística en cada uno de los posibles movimientos que puede hacer, y elegirá el mejor siendo el de menos coste y cumpliendo con condiciones como no poder traspasar muros.

Sin embargo, si sólo tenemos en cuenta la heurística en cada momento, el agente entra en bucle en la elección de la mejor heurística, siendo entonces necesario almacenar información sobre las posiciones y, en general, sobre las zonas que ya ha visitado. En este punto entra en acción el algoritmo RTA*, que usa un sistema de pesos almacenados para cada posición.

El almacenamiento de los pesos funciona de manera que, una vez visitada una posición, en ésta se almacene el segundo mejor peso de los hijos que había generado, es decir, el segundo mínimo peso de todos los posibles movimientos que había estudiado. De esta manera, al estudiar estos posibles movimientos, se comprueba si esa posición no tenía guardado un peso, en cuyo caso se lo asigna con **distancia Manhattan** y lo usa. En caso de que sí tuviese un peso asignado, usa éste para comprobar cuál es el menor.

Dado que el enfoque de nuestro agente es llegar al objetivo sin ser necesario explorar al máximo el mapa, usamos el segundo mínimo como cota, puesto que queremos acortar el espacio de búsqueda (se consigue con RTA*). En el supuesto de que se quisiera explorar la mayor parte del mapa posible se podría usar el primer mínimo, que lo que hace es expandir el campo de búsqueda a coste de mayor tiempo (este algoritmo pasaría a ser LRTA* *Learning Real-Time A**).



* El agente cuenta con los movimientos arriba, abajo, derecha, izquierda y diagonales.

¿Por qué hemos usado la distancia Manhattan?

Hemos hecho uso de la **distancia Manhattan** mejorada, teniendo en cuenta los movimientos diagonales. Hemos recurrido a esta herramienta fundamentalmente por lo simple y lo directa que es. La distancia Manhattan calcula la distancia entre dos puntos de forma rápida, haciendo la suma de las diferencias absolutas de las coordenadas X e Y de cada uno de ellos.

```
// -----  
// Calculamos la distancia Manhattan entre dos puntos.  
private int distanciaManhattan(SimpleEntry<Integer,Integer> puntoA, SimpleEntry<Integer,Integer> puntoB){  
    return Math.abs(puntoB.getKey()-puntoA.getKey()) + Math.abs(puntoB.getValue()-puntoA.getValue());  
}
```

Sin embargo, como hemos mencionado anteriormente, hemos implementado una versión que tiene en cuenta también los movimientos diagonales en la distancia.

```
// -----  
// Calculamos la distancia Manhattan entre dos puntos teniendo en cuenta las diagonales.  
private int distanciaManhattanDiagonal(SimpleEntry<Integer,Integer> puntoA, SimpleEntry<Integer,Integer> puntoB){  
    if (Math.abs(puntoB.getKey() - puntoA.getKey()) > Math.abs(puntoB.getValue() - puntoA.getValue()))  
        return Math.abs(puntoB.getKey() - puntoA.getKey());  
    else  
        return Math.abs(puntoB.getValue() - puntoA.getValue());  
}
```

En un entorno reactivo necesitamos que la velocidad del cálculo sea lo más eficaz posible para que nuestro agente, sin necesidad de explorar el mapa completamente, tome las mejores decisiones en tiempo real y encuentre el objetivo.

4. Interfaz Gráfica

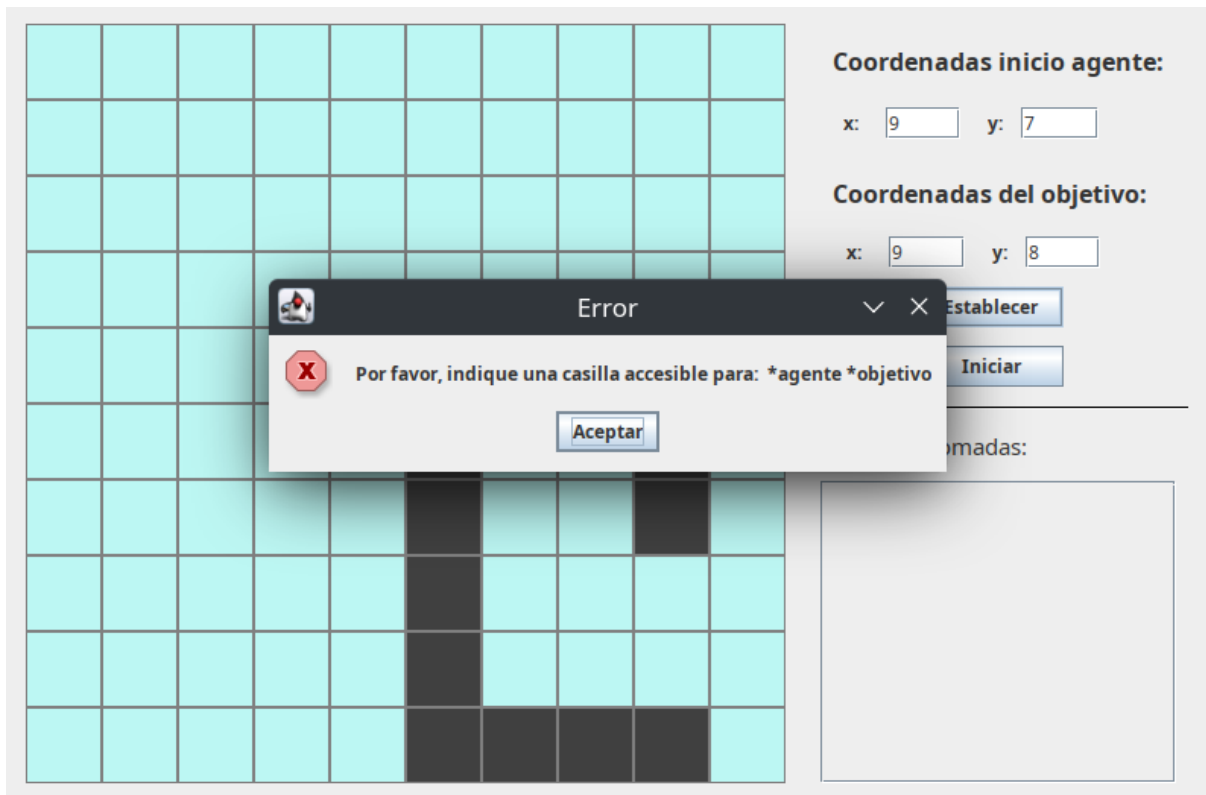
Hemos realizado una **interfaz gráfica** para ver visualmente los pasos que va dando nuestro agente y las decisiones que va tomando durante el trayecto. Esta interfaz proporciona una representación visual del mapa (mostrando las celdas accesibles y no accesibles), la posición actualizada del agente y la posición del objetivo.

Además, en nuestra interfaz permitimos que el usuario introduzca manualmente las coordenadas tanto del agente como del objetivo. En caso de no introducir ningún valor, se establecen las posiciones por defecto: el agente en la celda (0,0) y el objetivo en la celda (1,1).

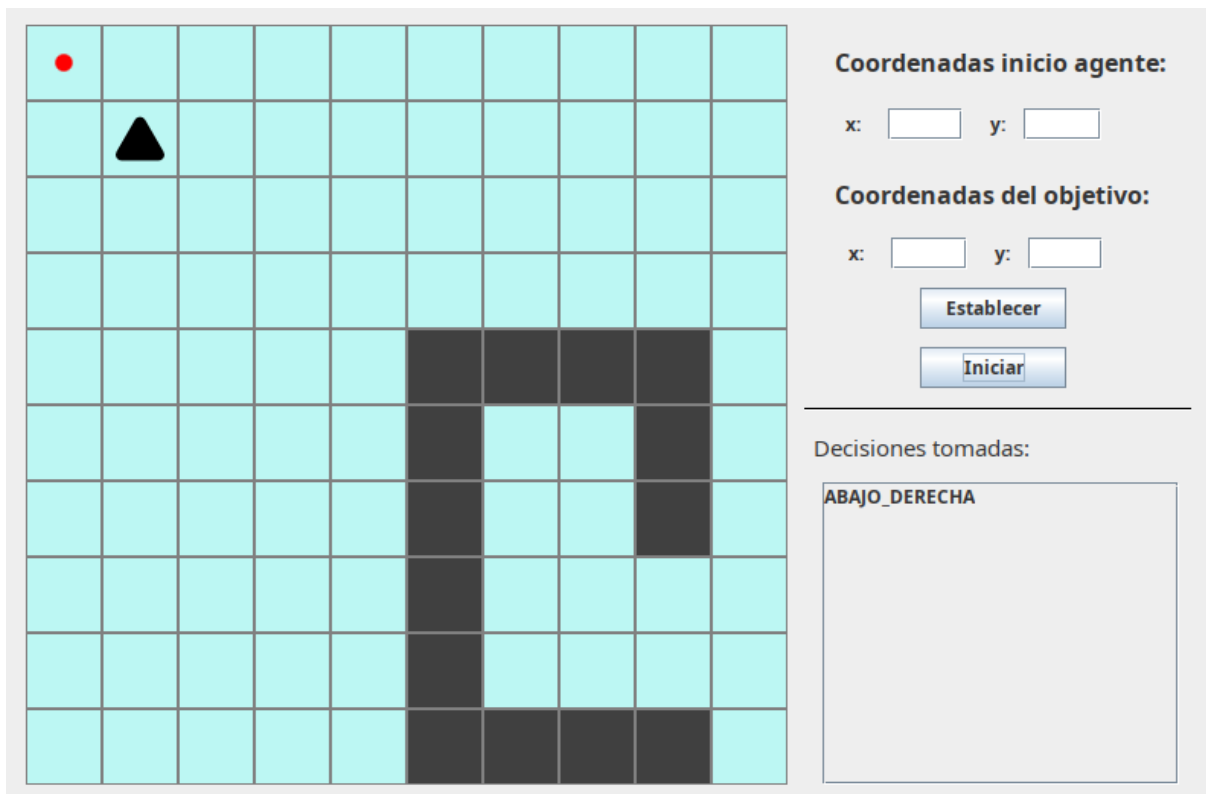
Finalmente, como parte de nuestra interfaz hemos incluido un pequeño apartado donde se van anotando las decisiones que va tomando el agente durante el trayecto, además de marcarse visualmente en el mapa con un pequeño punto rojo.

En la interfaz, las casillas azules son las accesibles y las negras las no accesibles.

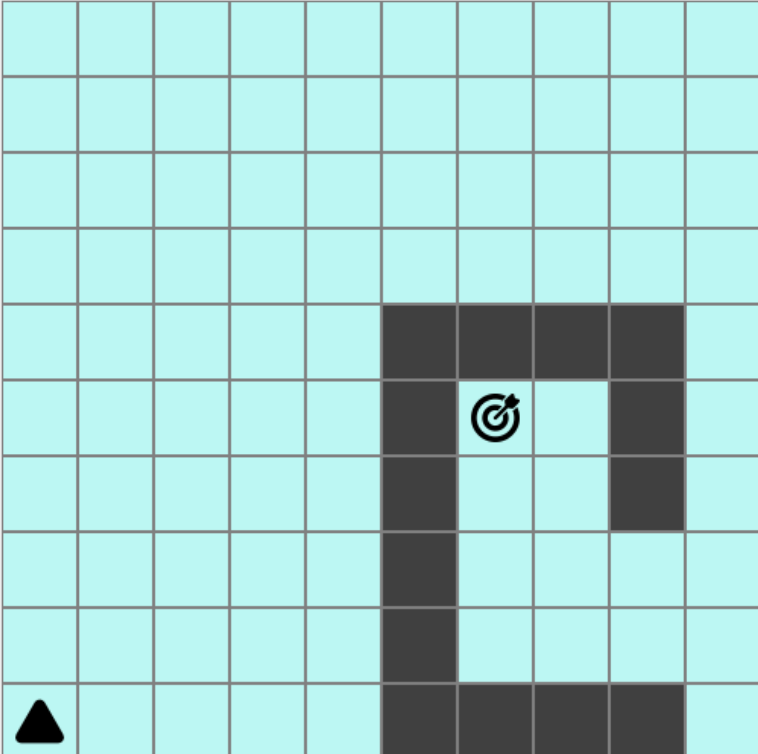
Capturas de pantalla de la interfaz del programa.



** Si colocas el agente y/o objetivo en una celda no accesible salta un error.*



** Si no introduces valores de posición, se colocan en posiciones por defecto.*



Coordenadas inicio agente:

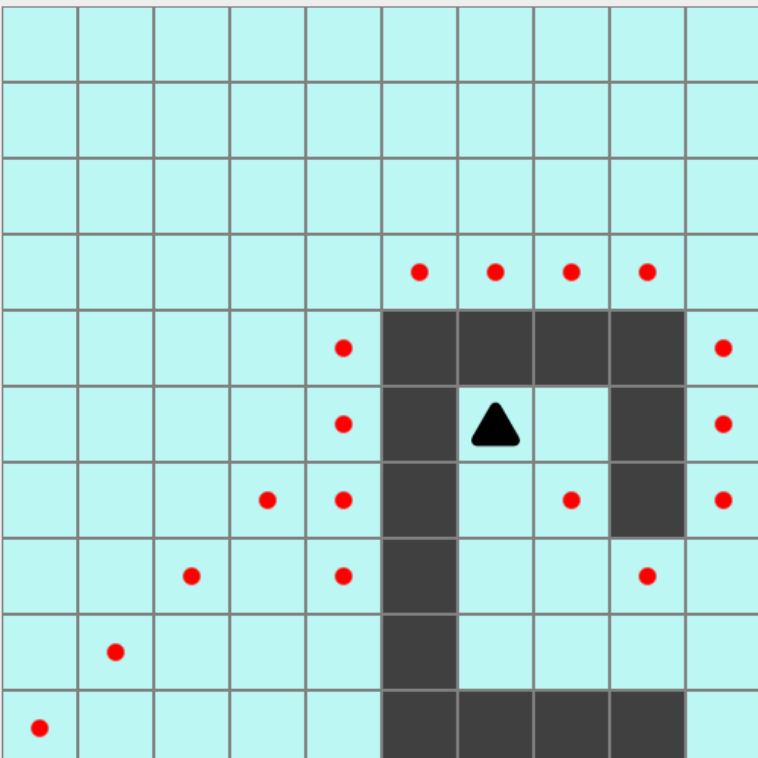
x: y:

Coordenadas del objetivo:

x: y:

Decisiones tomadas:

** Ejemplo de ejecución del programa.*



Coordenadas inicio agente:

x: y:

Coordenadas del objetivo:

x: y:

Decisiones tomadas:

ARRIBA_DERECHA
ARRIBA
ARRIBA_DERECHA
DERECHA
DERECHA
DERECHA
ABAJO_DERECHA
ABAJO
ABAJO
ABAJO_IZQUIERDA
ARRIBA_IZQUIERDA
ARRIBA_IZQUIERDA

** Ejemplo de finalización de ejecución del programa.*