MEMORIA PRÁCTICA 2
Movimiento de un agente en un mundo bidimensional

UNIVERSIDAD DE GRANADA

Integrantes:
Ángela María Garrido Ruiz
Adrián Romero Vilchez
Álvaro Ruiz Luzón
Francisco de Asís Rivas Fernández

Grupo: A3

Asignatura: DBA

ÍNDICE

1. Introducción

La resolución de está práctica está centrada en la realización de un agente reactivo que sea capaz de moverse por un mundo bidimensional, evitando obstáculos y alcanzando un objetivo.

Para el desarrollo de la misma hemos considerado la creación de las siguientes clases:

- **Clase mundo** - el mapa se representará mediante una matriz 2D formada por números enteros:
    - 0 para indicar que está la celda libre
    - -1 para indicar que tenemos un obstáculo

- **Clase entorno** - el entorno conoce el mapa del mundo y será el encargado de decirle al agente si las celdas que son percibidas por los sensores están libres o no.

- **Clase agente** - será la que contiene el cálculo de la ruta para llegar al objetivo, además contará con los distintos sensores para la percepción del mapa.

- **Interfaz gráfica** - será la encargada de mostrar la resolución del problema, es decir, nos permitirá ver cómo el agente ha llegado al objetivo.

Además, hemos creado otras clases Auxiliares que nos facilitaron la realización de la práctica, como pueden ser:

- **Clase Coordenadas** - Nos permiten trabajar con un objeto Coordenadas, en lugar de con un arrayList directamente.

- **Clase Direccion** - Es una clase enumerado con las posibles direcciones que puede seguir el agente.

- **Clase DireccionCombinada** - Es una clase que prioriza una dirección u otra en función de las posiciones del agente y del objetivo.

2. Clase Mundo.

La clase mundo cuenta con 3 atributos privados: enteros para las filas y columnas de la matriz y la declaración de la matriz como un array de dos dimensiones.

Para gestionar el mapa hemos creado los siguientes métodos:

- **Constructor por parámetros** - el parámetro que le pasamos es el nombre del archivo, ya que los mapas van a estar almacenados en formato de texto plano. Dentro de este método llamamos al método ***cargarMapa(****filepath****)***, creado por si en algún momento más adelante necesitáramos volver a cargar un mapa.

- **CargarMapa** - el parámetro es el nombre del archivo. Mediante el uso de ***Scanner*** leemos el archivo y sacamos las filas, las columnas y la matriz.

- **Getters** - hay 3 getters:
    - **GetCasilla** - dadas unas coordenadas (x, y) devuelve el valor de la matriz.
    - **GetFila** - devuelve el número de filas.
    - **GetColumna** - devuelve el número de columnas.

3. Clase Entorno.

Esta clase cuenta con cuatro atributos privados: el mapa, posiciones del agente y el objetivo y un vector de coordenadas (nos servirá para saber el camino trazado por el agente en la ejecución).

Para la realización de esta clase hemos montado la clase Coordenadas que es la encargada de contener la posición x e y del mapa, cuenta con un constructor y métodos getters y setters básicos.

Los métodos de los que disponemos en la clase entorno son:

- **Constructor con parámetros** - se encarga de inicializar los atributos de la clase

- **See** - nos devuelve la visión del agente.

- **Booleanos** - son los encargados de determinar si en la visión del agente, es decir, arriba, abajo, izda y dcha, está la celda libre o no. Nos devolverá true si la celda contiene el valor 0. También hay un booleano que nos indica si hemos alcanzado el objetivo.

- **moverAgente** - se encarga de actualizar la posición del agente cuando este haya avanzado en el mapa y guarda esta nueva posición en el vector de coordenadas, además de actualizar la energía utilizada.

- **Getters** - Tenemos los siguientes getters:
    - getEnergia - devuelve la energía consumida.

- getDireccion - devuelve la dirección hacia la que se moverá el agente.
- getAgentPos - devuelve la posición del agente.
- getTargetPos - devuelve la posición del objetivo.

4. Clase Agente.

Esta clase cuenta con los siguientes atributos: entorno, controlador (interfaz gráfica) y posiciones tanto del agente (la inicial y la actual) como del objetivo.

Los métodos que hemos desarrollado son:

- **setup()** - es el método que se llama para la inicialización del agente, obtiene los argumentos, establece las posiciones y añade un comportamiento que se ejecuta cada x ms.

- **MovimientoBehaviour** - extiende a la clase TickerBehaviour por lo que se realizará cada cierto tiempo. En cada tick el agente se intentará dirigir hacia la posición del objetivo, esa distancia la calcula el método ***agentePensar()***.
  - Si detecta que hay un obstáculo mostrará un mensaje por la consola diciendo que está buscando la ruta.
  - Si se alcanza el objetivo se muestra una ventana emergente con la energía que ha consumido para llegar y finaliza.

- **diff_filas** y **diff_columnas** - nos devuelve la diferencia de filas y columnas que hay entre el agente y el objetivo.

- **Funciones heurísticas** - hemos desarrollado dos:
  - **funcionHeuristica** - es la encargada de calcular la distancia entre el agente y el objetivo usando la distancia Manhattan (suma de diferencias en valor absoluto)
  - **funcionHeurísticaConObstaculos()** - como la anterior solo que hace que el agente no repita las casillas por las que ya ha pasado, añadiendo a éstas un valor adicional.

- **agentePensar()** - es la encargada de la toma de la decisión del movimiento del agente, es decir, que dirección toma. Mira todas las decisiones posibles y calcula el coste de realizar cada una de ellas, finalmente nos quedamos con la que tenga el valor más bajo y que sea válida.

- **Funciones auxiliares** - tenemos dos:
  - **obtenerNuevaPosición()** - nos da la posición actualizada del agente
  - **esMovimientoValido()** - comprueba que el movimiento en una dirección concreta es posible mediante una consulta al entorno.

5. Clase Dirección Combinada.

Es una clase en la que ordenaremos las direcciones en función de las posiciones del objetivo y del agente, de forma que prioriza aquellas direcciones que nos permiten acercarnos más al agente, para que a la hora de hacer un switch tengan prioridad.

6. Interfaz Gráfica.

La interfaz gráfica del programa se basa en una serie de componentes de **Swing** que permiten visualizar el entorno del agente en un mapa y proporcionar una experiencia visual para el usuario. A continuación, se detallan las clases que conforman la interfaz gráfica y cómo interactúan con el agente y el mapa.

a. Clase GUI.

La clase GUI es una subclase de JFrame que actúa como el marco principal de la aplicación. Su función principal es mostrar el entorno del agente y los controles necesarios para visualizar su progreso y estado. Está organizada de la siguiente manera:

- **Atributos**:
    - **MapPanel mapPanel**: es el panel donde se representa visualmente el mapa y los elementos de este, como el agente, el objetivo y las celdas ya visitadas. Este panel se crea usando la clase **MapPanel**, que extiende **JPanel** y sobreescribe el método **paintComponent** para personalizar la visualización.
    - **Controlador controlador**: es una referencia al controlador de la aplicación, que se usa para obtener información sobre el estado actual del agente, su posición, y el objetivo. El **Controlador** actúa como intermediario entre el entorno y la interfaz gráfica.
    - **JLabel energiaLabel**: etiqueta que muestra la energía consumida por el agente, la cual se actualiza cada vez que el agente se mueve.

- **Constructor GUI(Controlador controlador, Mapa mapa)**:
    - Configura la ventana principal (**JFrame**) con título, tamaño y propiedades básicas como el comportamiento de cierre.
    - Define el diseño como **BorderLayout** y añade el **MapPanel** al centro para visualizar el mapa.
    - Inicializa la etiqueta **energiaLabel**, que se sitúa en la parte inferior de la ventana (**BorderLayout.SOUTH**) y muestra la energía consumida por el agente. El texto de esta etiqueta se obtiene a partir del controlador.

- **Método actualizarMapa()**:
    - Este método actualiza la interfaz gráfica después de cada movimiento del agente. Se llama cada vez que el agente cambia de posición.
    - Llama a **mapPanel.actualizarAgente**, que establece la nueva posición del agente en el **MapPanel** y actualiza la lista de coordenadas visitadas.

- Actualiza el valor de la etiqueta de energía (**energiaLabel**) para reflejar el nuevo valor de energía obtenido desde el controlador.

La clase **GUI** se encarga de la estructura visual del entorno y permite al usuario observar en tiempo real el movimiento del agente y su progreso en términos de energía. Está diseñada para ser un puente entre el controlador y los elementos visuales de **Swing**, evitando manejar la lógica del entorno directamente.

b. Clase Controlador.

La clase **Controlador** es el intermediario entre la lógica del agente y la interfaz gráfica (**GUI**). Actúa como un controlador clásico en un patrón de diseño MVC (Modelo-Vista-Controlador), centralizando la lógica de movimiento del agente y la actualización de la interfaz gráfica.

- **Atributos**:
  - **Entorno entorno**: instancia de la clase **Entorno** que representa el espacio en el que opera el agente, con métodos para obtener la posición y el estado del agente, verificar obstáculos y mover al agente.
  - **GUI gui**: referencia a la instancia de **GUI**, que se actualiza cada vez que el agente cambia de posición para reflejar los nuevos datos visuales en la interfaz.

- **Constructor Controlador(Entorno entorno, GUI gui)**:
  - Inicializa los atributos **entorno** y **gui** y establece la relación entre ellos para que el controlador pueda actualizar el GUI con los cambios en el entorno.

- **Métodos de movimiento del agente**:
  - La clase **Controlador** define métodos específicos para mover el agente en cada dirección: **moverAgenteArriba**, **moverAgenteAbajo**, **moverAgenteIzquierda**, y **moverAgenteDerecha**.
  - Cada uno de estos métodos verifica si el movimiento es posible en esa dirección llamando a métodos del entorno (**libreArriba**, **libreAbajo**, **librelzda**, **libreDcha**), que devuelven **true** o **false** según si la posición está libre.
  - Si el movimiento es posible, actualiza la posición del agente en el entorno mediante **entorno.actualizarAgente**.
  - Después de cada movimiento exitoso, el controlador llama a **gui.actualizarMapa()** para reflejar la nueva posición del agente en la interfaz.
  - Además, llama a **verificarObjetivo()** para determinar si el agente ha alcanzado el objetivo. Si el agente llega a la posición del objetivo, se muestra un mensaje de éxito con la energía consumida y el programa se cierra.

- **Método verificarObjetivo()**:
  - Compara la posición actual del agente con la posición del objetivo. Si coinciden, abre un **JOptionPane** mostrando un mensaje de éxito junto con la energía consumida.

    - Este mensaje finaliza el programa al cerrar la ventana del diálogo.

- **Método agentePensar()**:
    - Llama a un método de **entorno**, **agentePensar()**, para que el agente decida su siguiente movimiento en función de la lógica del entorno. El método **agentePensar()** devuelve un valor de **Direccion** (ARRIBA, ABAJO, IZQUIERDA, DERECHA).
    - Según la dirección recibida, el controlador ejecuta el método de movimiento correspondiente para desplazar al agente en esa dirección.

- **Método getEnergia()**:
    - Calcula la energía consumida restando uno al tamaño de la lista de coordenadas del recorrido del agente. Cada movimiento agrega una coordenada a esta lista, por lo que el tamaño de la lista menos uno representa la energía consumida.

La clase **Controlador** encapsula la lógica de actualización del estado del agente y de la interfaz gráfica, permitiendo así un desacoplamiento entre el entorno (modelo) y la GUI (vista). Esto facilita la gestión de los eventos y movimientos del agente sin que la lógica de movimiento y los cálculos de energía se mezclen con los elementos de visualización de **Swing**.

7. Conclusión.

Tras la realización de lo explicado anteriormente se consiguió lograr una implementación de un agente inteligente capaz de recorrer un entorno con obstáculos para alcanzar un objetivo definido.

A lo largo de la implementación nos hemos enfrentado a problemas como controlar que el agente no hiciera movimientos en bucle o que se quedará estancado en los límites del mapa.

La interfaz gráfica ha supuesto un valor añadido que nos permite tener una visión más clara del comportamiento del agente, así como la visualización de la energía gastada y restante. Aunque esta implementación supuso algunas complicaciones, en la integración de la interfaz con el agente y sus comportamientos, conseguimos hacerla funcional lo que nos permite tener una visión más completa de la solución aportada.

En conclusión, nuestro trabajo cumple con los requisitos planteados en el guión y ha permitido que nos familiaricemos con la programación de agentes inteligentes en java así como en la realización de interfaces.