BÚSQUEDA DEL TESORO

Integrantes:

Josué Toledo Castro
Noé Campos Delgado
María Nayra Rodríguez Pérez

ÍNDICE

1. OBJETIVO DE LA PRÁCTICA

El objetivo de este apartado es el diseño de un sistema basado en agentes utilizando diferentes robots humanoides. Para ello, habrá que definir los elementos del sistema basado en agentes (percepciones, acciones, objetivos y entorno), así como la arquitectura software indicando qué funcionalidades realizaría cada agente.

Esta práctica está basada en la utilización de estrategias de búsqueda como propuesta de resolución en la determinación de la planificación de trayectorias para la navegación de un robot.

2. DESCRIPCIÓN DEL PROBLEMA

El entorno del robot es un panel rectangular de dimensiones MxN. Debe estar constituido por celdas donde el agente puede realizar acciones de movimiento, una cada vez, desde la casilla actual a una de las cuatro vecinas, puesto que este movimiento sólo puede efectuarse en horizontal y vertical (norte, sur, este y oeste).

En este problema, el robot puede encontrar casillas ocupadas por obstáculos, que se crearán tanto manualmente como aleatoriamente. En ese caso, nuestro agente debe encontrar el mejor camino por las casillas libres.

Para encontrar el mejor camino, será necesario utilizar una estrategia de búsqueda para encontrar la solución a nuestro problema de planificación de trayectoria del robot a través de los obstáculos.

2.1. Escenario

El escenario planteado tendrá como agente principal a un pirata que busca un tesoro en un mapa. De este modo, el pirata tendrá el rol de robot humanoide (agente), el tesoro y el mapa serán elementos del entorno. El primero será el premio u objeto que buscará el pirata y el mapa será la matriz MxN. Los obstáculos dispuestos en el mapa serán rocas, árboles y palmeras entre otros.

2.2. Agentes

Un agente es un sistema que interactúa con su entorno, es decir, que percibe de él a través de sensores y actúa sobre él a través de efectores, realizando una cierta tarea, es decir, cumpliendo unos objetivos.

Como agentes podemos destacar los siguientes:

1. Robot humanoide (pirata). Éste se moverá automáticamente por una matriz de MxN con el objetivo de encontrar un objeto determinado. En nuestro caso, el robot tendrá el rol de pirata y el objetivo perdido a encontrar será un tesoro.

No se considerarán como agentes los obstáculos o los otros elementos descritos en el apartado de entorno, puesto que no interactúan con los otros elementos más que por su mera presencia en el proyecto.

2.3. Diseño

- **Percepciones**

El Robot dispone de cuatro movimientos, para cada uno de los posibles direcciones (norte, sur, este y oeste). Debe detectar si existe un obstáculo y encontrar un camino óptimo hasta llegar a su destino. A través de estos movimientos, el robot humanoide establecerá un camino y casilla por casilla tendrá que comprobar la presencia o no de un obstáculo.

- **Acciones**
    - El agente puede moverse sobre la matriz MxN en dirección horizontal y vertical (norte, sur, este y oeste).
    - Detectar la presencia de un obstáculo.
    - En el caso de que en el siguiente movimiento no exista un obstáculo, el robot deberá continuar por el mejor camino.
    - En el caso de que el agente detecte un obstáculo, deberá evitar el mismo y recalcular el camino para llegar al objetivo.
    - En el caso de que no haya camino posible debido a la presencia excesiva de obstáculos que no posibiliten al agente o robot llegar al objetivo deseado, éste no se moverá.
    - En el caso de que el robot humanoide pueda moverse pero no exista camino posible para llegar al premio u objetivo debido, por ejemplo, a la presencia de obstáculos alrededor del mismo en las casillas cercanas.

- **Objetivos**

El objetivo del robot humanoide es conseguir llegar a la casilla establecida mediante una referencia, sorteando todos los obstáculos. Tomando el mejor camino para él.

- **Entorno**

Como principales elementos del entorno cabe destacar:

- **Obstáculos**: La presencia de obstáculos en la matriz de MxN tendrán por función principal imposibilitar el movimiento del robot. No se considerarán agentes como consecuencia de que no cambian o modifican su comportamiento en función de unos estímulos o elementos presentes en el entorno, como sí hace el robot humanoide. En cuanto a la caracterización, los obstáculos serán árboles, rocas, agua,...

-Presencia de **otros agentes** en el caso de que los hubiese.

- **Objetivo del agente**: El tesoro o premio que encontrará el agente principal o robot. Este elemento será estático, es decir, no se moverá por la matriz ni interactuará con otros elementos del entorno en función de las acciones de estos últimos.

- **Mapa**: Este elemento será la matriz MxN. Se dispone encasillado a modo de poder determinar la posición en función de los ejes X e Y de los otros elementos del entorno.

- **Arquitectura Software:** Agente basado en objetivos.

Para decidir que hay que hacer, aparte de la información acerca del estado que prevalece en el ambiente, se necesita: cierto tipo de información sobre su meta, inteligencia artificial y agentes inteligentes. El agente así diseñado es más flexible.

2.4. Implementación

El lenguaje utilizado para este caso práctico ha sido: Java.

Este lenguaje presenta más recursos para programar una interfaz gráfica, puesto que está más orientado a este tipo de programación. Además como principal ventaja, lo hemos elegido por tener una mayor potencia de cálculo a la hora de realizar la heurística.

La herramienta para utilizar este lenguaje ha sido: Eclipse.

Eclipse es un software de desarrollo de código abierto basado en Java. Por si mismo, es un marco de trabajo y un conjunto de servicios para la construcción del entorno de desarrollo de los componentes de entrada. Éste tiene un conjunto de complementos, incluidos las Herramientas de Desarrollo de Java (JDT). Este software presenta una mayor usabilidad frente a otros softwares similares, al igual que da más complementos/plugins para el desarrollo de Java.

Hemos utilizado el plugin JFrame y la biblioteca de funciones javax.swing.*, las cuales nos permiten diseñar una aplicación con ventanas y elementos gráficos.

3. ESTRATEGIA HEURÍSTICA

Se pretende realizar una planificación del camino usando estrategias heurísticas. Con nuestro agente definido anteriormente, se deberá determinar la trayectoria óptima partiendo desde una posición inicial hasta alcanzar una posición final. Ambas posiciones son definidas por el usuario, en el entorno de simulación desarrollado previamente. Se puede disponer de un mapa topológico que contenga la descripción del entorno.

La exploración heurística realiza la exploración utilizando algún conocimiento específico o heurístico del problema.

En la implementación de nuestro caso práctico se utilizará: Estrategia en Escalada.

La estrategia en escalada se basa en continuar por el mejor de los hijos del nodo actual, basándonos en alguna evidencia que nos permita ordenar los nodos hijos, de tal forma que se exploren primero aquellos que presentan mayor expectativa de éxito.

Implementación:

1. Introducir en la lista una trayectoria inicial que contiene únicamente el nodo raíz.
2. Hasta que la lista esté vacía o se encuentre el objetivo, examinar la primera trayectoria de la lista:
    2.1. Si el primer nodo es el objetivo, entonces salir del bucle.
    2.2. Si el primer nodo no es el objetivo, eliminar esta trayectoria de la lista y añadir sus trayectorias descendientes, en el caso de que existan, al principio de la lista ordenada en base a sus expectativas de éxito, colocando en primer lugar a la trayectoria más prometedora.
3. Si el objetivo se ha encontrado finaliza el procedimiento con éxito, siendo la solución la primera trayectoria de la lista, en caso contrario el problema no tiene solución.

Ventajas:

- Usa muy poca memoria
- Puede encontrar soluciones razonables en espacios de estado grandes.

Dificultades:

- Existencia de máximos locales: lo cual sólo asegura una optimización local pero no global.
- Existencia de áreas planas: lo que implica la no existencia de direcciones privilegiadas que conduzca con seguridad al máximo.
- Existencias de aristas: o discontinuidades en la pendiente, provocando una optimización muy local.

3.1. Distancias

- **D. Euclídea**: D(a,b) = $\sqrt{(X1 - X2)^2 + (Y1 - Y2)^2}$

En nuestra práctica se ha optado por utilizar la Distancia Manhattan en primer lugar. Pero después de realizar varias trazas de nuestro algoritmo (escalada), hemos decidido finalmente utilizar la Distancia Euclídea. Esto es debido a que si se obtiene en dos casillas adyacentes a la posición del pirata el mismo valor, habría que definir un orden de prioridad, y esto dependería siempre de la posición del tesoro. Por ello hemos usado finalmente esta distancia, puesto que siempre nos dará decimales, nunca valores iguales.

4. DESARROLLO DE LA PRÁCTICA

4.1. Interfaz

4.1.1. Imágenes

- Añadir imágenes al botón.

A través del siguiente método se inserta la imagen para cada una de las casillas de la matriz de botones. Inicializamos la matriz a botones y la recorremos. Para cada casilla se redimensiona el tamaño de la imagen obteniendo 'ancho' y 'alto', así se escala con las dimensiones especificadas.

```
mCasillas = new JButton[numero_filas][numero_columnas];
  for(int i=0;i<numero_filas;i++)
  {
     for(int j=0;j<numero_columnas;j++)
     {
        mCasillas[i][j] = new JButton();
        ancho = 900/numero_columnas;
        alto = 600/numero_filas;
```

Para escalar las imágenes con el tamaño ya calculado, utilizamos la función 'getScaledInstance' añadiendo el ancho y alto deseado para que la imagen encaje correctamente en el botón. Después de obtener la imagen escalada, la introducimos en la matriz de botones.

```
        ImageIcon imagen_arena = new ImageIcon(getClass().getResource("arena.jpg"));

        ImageIcon imagen_escalada= new
        ImageIcon(arena.getImage().getScaledInstance(ancho, alto,
        java.awt.Image.SCALE_FAST));

        mCasillas[i][j].setIcon(imagen_escalada);
```

A continuación, se vincula al botón con su posición (i, j) de la matriz.

```
        String posicion = new Integer(i).toString();
        posicion += ","+new Integer(j).toString();

        mCasillas[i][j].setActionCommand(posicion);
        panel.add(mCasillas[i][j]);
  }
}
```

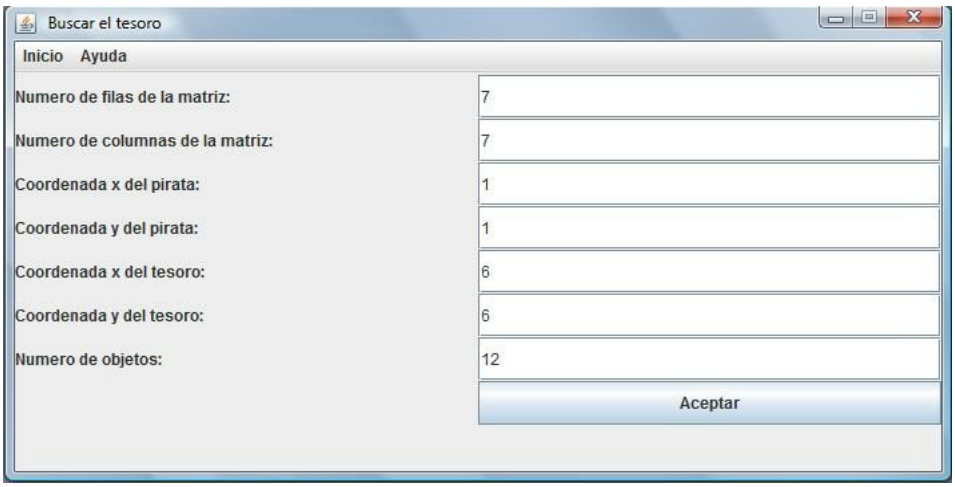
4.1.2. Paneles

- Ventana inicial.

Menú principal en el que se le da la opción al usuario de elegir entre panel con datos, el usuario introduce manualmente los elementos, o panel con ratón, el usuario introduce con el ratón los elementos en el tablero.

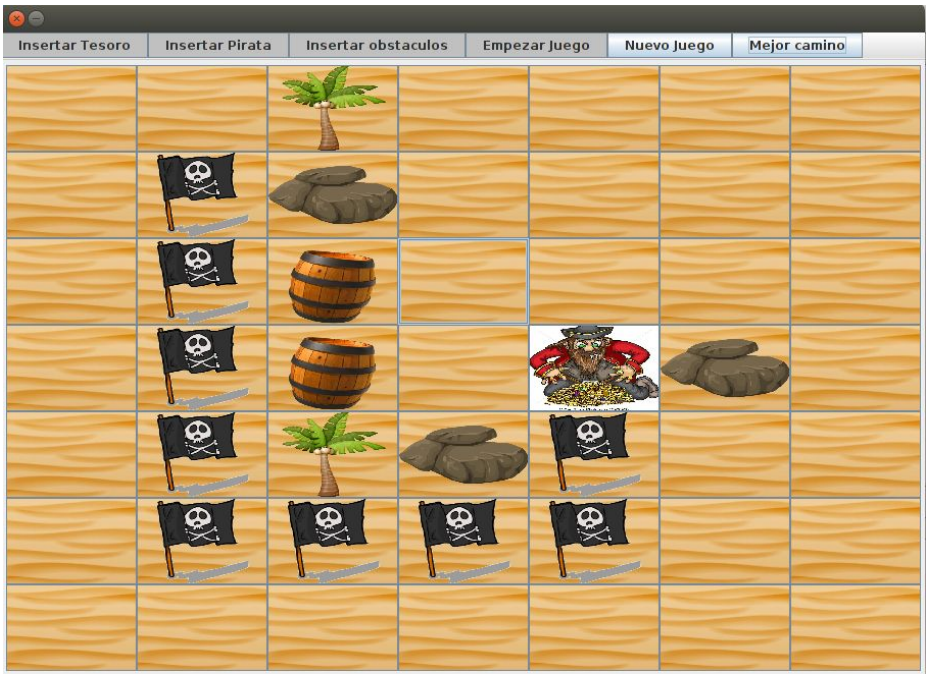
- Panel de datos.

Menú en el que el usuario introduce por teclado las coordenadas de cada uno de los elementos a introducir en el tablero.



- Panel con ratón.

Con esta opción el usuario visualiza directamente el tablero y pulsando el botón de cada elemento específico puede introducirlo con el ratón en la coordenada que desee.



4.2. Planificación del camino

4.2.1. Nomenclatura

A continuación se muestra la nomenclatura utilizada en nuestro código para el cálculo de los posibles movimientos que tendrá el pirata.

	X - 1	
Y - 1	**Posición Pirata**	**Y + 1**
	X + 1	

4.2.2. Heurística - Distancia

- Distancia Euclídea: D(a,b) = $\sqrt{(X1 - X2)^2 + (Y1 - Y2)^2}$

Para el cálculo de la heurística hemos creado una variable 'distancia', la cual hemos inicializado con un valor alto (10000) para que al comparar siempre escoja uno de los posibles caminos.

Mediante el uso de condicionales, comparamos la menor distancia entre las cuatro casillas adyacentes.

```
private int calcular_heuristica() {

if(distancia > (Math.sqrt(Math.pow((pirataX-1)-tesoroX,2)+Math.pow(pirataY-tesoroY,2)))
{
        if((matriz_visitados[posicion_x_pirata_aux-1][posicion_y_pirata_aux]==false) &&
        (matriz_Obstaculos[posicion_x_pirata_aux-1][posicion_y_pirata_aux]==false))
        {
        distancia = Math.sqrt(Math.pow((Xpirata-1)-Xtesoro,2)+Math.pow(Ypirata-Ytesoro,2))
        movimiento = 1;                 → Movimiento arriba
        }
}
…
```

Realizamos este código para cada uno de los movimientos posibles (abajo, derecha e izquierda). Si la distancia calculada anteriormente es mayor que la actual, cambiamos el valor 'distancia' por el actual. Así obtendremos el mejor movimiento para el pirata en cada una de las casillas.

En el caso que no obtenga ningún movimiento, es decir, no entramos en ninguna de las condiciones anteriores, el pirata no obtendrá movimiento.

```
if (control == false){
        movimiento = 0;
}
```

4.2.3. Estrategia en Escalada - Algoritmo

El algoritmo comienza siempre y cuando el pirata y el tesoro se encuentren en posiciones diferentes, para ello comprobamos: mientras la posición 'x' o la posición 'y' de pirata y tesoro no coincidan, calculamos la heurística y determinamos el mejor movimiento posible.

```
while (pirataX != tesoroX) || (pirataY != tesoroY){
        int mov = calcular_heuristica();

switch (mov){
```

Han de contemplarse los diferentes casos posibles en base a los movimientos del pirata, pueden ser:

- Caso 0: No hay camino posible. El pirata se encuentra en una situación en la cual está rodeado de obstáculos o bien todas sus casillas adyacentes están visitadas.
- Caso 1: Movimiento arriba (X-1).
- Caso 2: Movimiento abajo (X+1).
- Caso 3: Movimiento izquierda (Y-1).
- Caso 4: Movimiento derecha (Y+1).

En primer lugar, contemplamos el caso en el que el pirata no tenga un camino posible. Retrocedemos a la casilla anterior para calcular un nuevo camino. Para ello utilizamos una pila que tendrá almacenada cada posición por la que ha pasado el pirata, por lo tanto en el top de la pila estará almacenada la última posición visitada.

```
case 0:
        if(p.empty()!=true)
        {
                coordenadas aux1 = new coordenadas();
                aux1 = p.pop();
                int posX_pila = (int)aux1.get_x();
                int posY_pila = (int)aux1.get_y();
```

A continuación, comparamos la posición extraída de la pila con cada una de las adyacentes a la posición del pirata para marcar el camino recorrido con una nueva imagen.

```
if ((posX_pila == pirataX-1) && (posY_pila == pirataY))
{
        mCasillas[pirataX][pirataY].setIcon(imagen);
}
if((posX_pila == pirataX+1) && (posY_pila == pirataY))
{
        mCasillas[pirataX][pirataY].setIcon(imagen);
}
if((posX_pila == pirataX) && (posY_pila==posicion_y_pirata_aux-1))
{
        mCasillas[pirataX][pirataY].setIcon(imagen);
}
if((posX_pila == pirataX) && (posY_pila == pirataY+1))
{
        mCasillas[pirataX][pirataY].setIcon(imagen);
}
```

La posición escogida será actualizada en la matriz de visitados marcándose como 'true' y por lo tanto también será introducida en el top de la pila.

```
mCasillas_visitados[pirataX][pirataY] = true;
pirataX = posX_pila;
pirataY = posY_pila;
```

Para el caso en el que el mejor movimiento calculado sea 'movimiento=1', accedemos al primer case y el pirata se movería hacia arriba. En esta situación, realizamos lo siguiente:

- Marcamos la posición del pirata con la imagen que hemos asignado para marcar el camino.
- Actualizamos dicha posición en la matriz de visitados.
- Almacenamos en la pila la posición actual del pirata.
- Por último avanzamos en el panel y actualizamos la posición del pirata a la mejor posición calculada anteriormente.

```
case 1:
        mCasillas[pirataX][pirataY].setIcon(imagen);
    mCasillas_visitados[pirataX][pirataY]=true;
    p.push(pirataX, pirataY);
    pirataX = pirataX - 1;
```

Para los siguientes movimientos posibles (abajo, izquierda y derecha), realizamos las mismas acciones en cada caso, salvo al actualizar la posición del pirata. Dependiendo de cada movimiento, actualizamos dicha posición con sus coordenadas respectivas.

A continuación se muestran los tres movimientos restantes que puede realizar nuestro agente.

```
case 2:
        mCasillas[pirataX][pirataY].setIcon(imagen);
        mCasillas_visitados[pirataX][pirataY]=true;
        p.push(pirataX, pirataY);
        pirataX = pirataX + 1;

case 3:
        mCasillas[pirataX][pirataY].setIcon(imagen);
        mCasillas_visitados[pirataX][pirataY]=true;
        p.push(pirataX, pirataY);
        pirataY = pirataY - 1;

case 4:
        mCasillas[pirataX][pirataY].setIcon(imagen);
        mCasillas_visitados[pirataX][pirataY]=true;
        p.push(pirataX, pirataY);
        pirataY = pirataY + 1;
```

Finalmente, se realizan las comprobaciones en los casos en los cuales pueda finalizar la búsqueda del camino.

- No existe camino posible.

```
if(((pirataX != tesoroX) ||(pirataY != tesoroY))
{
    if(calcular_heuristica() == 0)
        JOptionPane.showMessageDialog(null, "No hay camino posible", "Fin del
        juego", JOptionPane.WARNING_MESSAGE);
        break;
```

- El pirata ya ha encontrado el tesoro.

```
if((pirataX==tesoroX)&&(pirataY==tesoroY))
    ImageIcon imagen = new ImageIcon(getClass().getResource("/../premio.jpg"));
    mCasillas[pirataX][pirataY].setIcon(imagen);

    JOptionPane.showMessageDialog(null, "Tesoro encontrado", "Fin del juego",
    JOptionPane.WARNING_MESSAGE);
```

4.3. Problemas encontrados

Al implementar el algoritmo de escalada, se ha detectado un problema a la hora de recorrer la matriz y el agente quedarse encerrado entre obstáculos y casillas visitadas.

Cada vez que el pirata avance de casilla, se marcará como visitada. Así cuando se de la situación en la que el pirata no encuentre una casilla posible por la que continuar el camino, puesto que está rodeado de obstáculos o casillas visitadas, pararía la ejecución y devuelve que no encuentra solución. Esto ocurre porque en la condición está establecido que no contemple las casillas con obstáculos, y también las casillas visitadas para evitar que se cicle y entre en un bucle.

Como solución a este problema se ha implementado una pila que va almacenando cada posición que visita el pirata. En el caso en el que no encuentre casilla vacía por la que continuar, se extrae (pop) de la pila la posición anterior y marcamos la actual como visitada. Así permitimos que el pirata pueda retroceder el camino ya visitado y seguir realizando la búsqueda.

4.4. Mejoras

Al implementar las mejoras hemos introducido un botón “Camino mínimo”, donde mostramos el camino mínimo realizado por el agente.
Este camino se debe a que las posiciones del agente quedan almacenadas en la pila una vez que encuentra su objetivo, las cuales marcamos con una bandera para poder diferenciarlas.

```
while((pila.!empty))
{
        Int pos_x_min= (int)aux.get_x();
        Int pos_y_min= (int)aux.get_y();
        mCasillas[pos_x_min][pos_y_min].setIcon(null);
        mCasillas[pos_x_min][pos_y_min].setBackground(Color.BLUE);
}
```

5. METODOLOGÍA DE TRABAJO

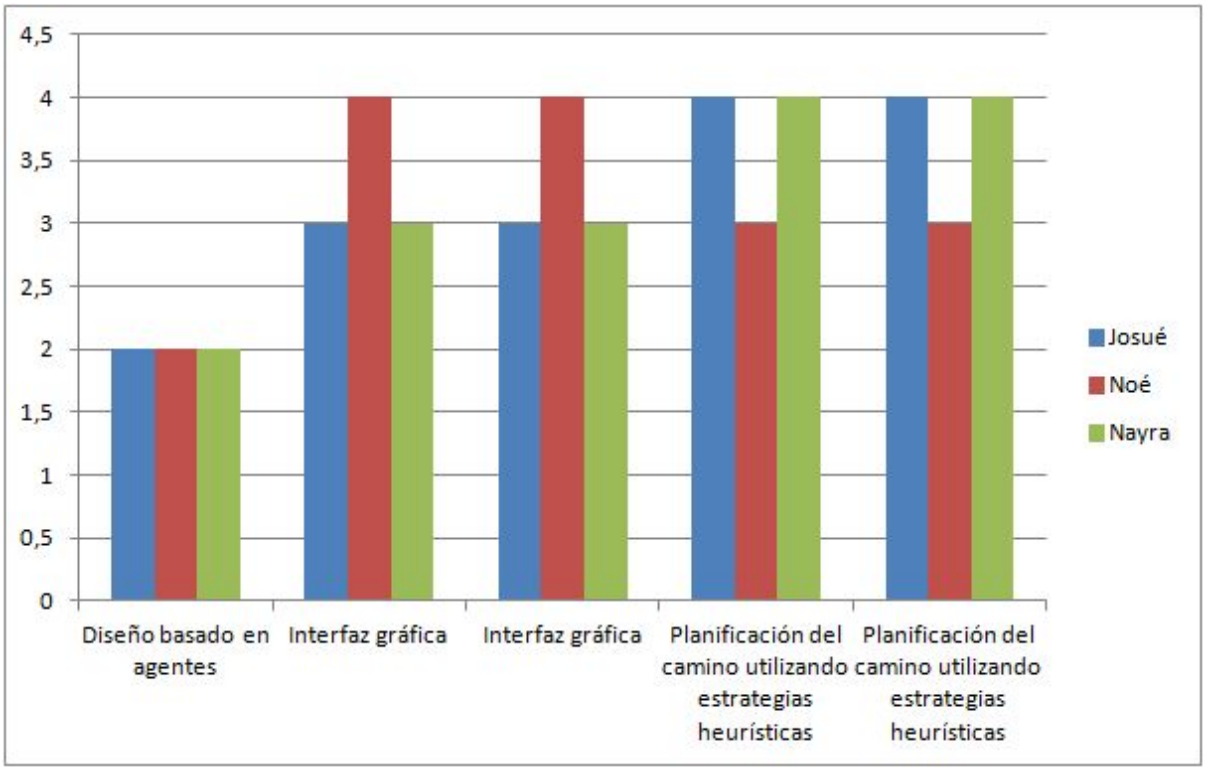
Integrantes

- Josué Toledo Castro
    - Correo: **alu0100763492@ull.edu.es**

- Noé Campos Delgado
    - Correo: **alu0100622492@ull.edu.es**

- María Nayra Rodríguez Pérez
    - Correo: **alu0100406122@ull.edu.es**

Diagrama de trabajo



Enlace al repositorio

- Github: http:/github.com/JosueTC94/PracticaIA_Pirata

6. BIBLIOGRAFÍA

- Tutoriales Java:
  - http://www.javaya.com.ar/
  - https://www.youtube.com/playlist?list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk
  - https://codigofacilito.com/courses/JAVA

- Algoritmos:
  - https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*
  - http://www.ecured.cu/index.php/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*
  - http://inteligencia7b.blogspot.com.es/2010/11/hill-climbing-search.html
  - http://oscar-sandoval.blogspot.com.es/2012/08/vendedor-viajero-algoritmos-hill.html