



**UNIVERSIDAD DE GUADALAJARA**

**CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS DIVISIÓN DE  
ELECTRÓNICA Y COMPUTACIÓN**

**DEPARTAMENTO DE CIENCIAS COMPUTACIONALES**

**INGENIERÍA EN COMPUTACIÓN**

**INTELIGENCIA ARTIFICIAL I**

**SECCIÓN: D01**

**CLAVE: I7038**

**Nombre: Elizalde- Loera Felipe de Jesus**

**Codigo: 211715281**

**Actividad 2: Búsqueda en el mundo fantasma(BMF)**

## Contenido

Introducción.....	1
HyperLink.....	1
Explicación del Mapa y Agentes.....	2
Mapa.....	2
Agentes.....	3
Red Creciente.....	3
Movimientos del agente.....	3
Premios y Castigos.....	4
Recorridos.....	4
Dijkstra.....	4
Elección de lenguaje de programación.....	5
Diagrama de Clases.....	6
Conclusión.....	7
Codigo.....	7

## Introducción

En esta actividad Búsqueda en el mundo Fantasía (BMF) se tendra que modificar el laberinto que creamos en la primer actividad para hacer Búsqueda con Bresenham. Tendremos que crear una red en donde todos las casillas en el laberinto estan conectadas. Tendremos 3 agentes que recorran el laberinto y dependiendo si fue bueno o malo el recorrido se catigara o premiara la ruta. Al ultimo aplicamos dijkstra para encontra el camino mas rapido para cada agente.

## HyperLink

<https://youtu.be/S3xepFXRvQc>

La liga anterior muestra la ejecución del BMF. Nota. En el minuto 1:46 parece que se congela Mombo al momento de ejecutar Dijkstra por primera vez, esto es error del Programa que utilice para grabar el programa. Cuando termina Mombo y empieza Pirolo se quita ese error visual y despues seleccióno otro punto para aplicar Dijkstra y ya podran ver a Mombo recorrer bien con el algoritmo.

# Explicación del Mapa y Agentes

## Mapa

En la siguiente tabla podran ver que significa cada imagen que aparece en el mapa.









Pasto Verde/Plano	
Pasto Seco/Plano	
Agua	
Montaña	
Baranco	
Muro	
Inicio	
Salida	

Figura I: Significado de Imagenes

## Agentes

En la siguiente tabla podran ver las caracteristicas de cada Agente.





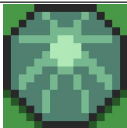










Agentes/Esfuerzo	0.3	1.0	1.5	2.5
<i>Mombo</i> 				
<i>Pirola</i> 				
<i>Lucas</i> 				

Figura II: Caracterisitcas de los agentes

## Red Creciente

La red creciente o growing network que se creo es el laberinto, en donde cada casilla tiene 8 conexiones si es que no esta en una de las 4 esquinas o 4 lados. Si es encuentra en una de las esquinas entonces tiene 3 conexiones y si esta un unos de los lados o (sides) entonces tiene 5 conexiones. Para hacer las conexiones se facilito el trabajo porque utilze una matriz en donde cada casilla tiene un vector dinamico que nos guarda las ponderacioens para cada agente que es  $MAX\_INT/2$ , el tipo de terreno y si esta visitado la casilla. La matriz fue generado con un vector de 3 niveles es decir *Vector<Vector<Vector<Long>>>*.

## Movimientos del agente

Ahora que tenemos nuestra red creciente tenemos que programa los movimientos de los agentes. Para hacer esto validamos si esta en unas de las esquinas, lados o en el centro. Despues de eso el funcionamiento es basicamente el misma para las 3 opciones. Se crea una lista con las opciones disponibles es decir unos de los

vecinos a los que puede ir. Se escoge al azar uno de los vecinos y se puede ir a esa casilla (que no está visitada) entonces va a la casilla y la marca como visitada. Pero que pasa si escoge una casilla que ya está visitada entonces elimina esa opción de la lista y escoge otra opción al azar. Esto lo sigue haciendo hasta que encuentra una opción o si la lista queda vacía. Si queda vacía entonces utilizamos backtracking para ir al último visitada y escogemos otro vecino. Este proceso se repite hasta encontrar la salida.

## Premios y Castigos

El primer recorrido es el **bootstrap** en donde no hay premios o castigos si no que inicializamos el mínimo y máximo que es la sumatoria de esfuerzos para cada agente. Después de eso todos los recorridos tienen un premio o castigo. Si la sumatoria es menor que el promedio del mínimo y máximo es decir  $(\text{mínimo} + \text{máximo})/2$  entonces se premia el recorrido que tenemos guardado que es la lista de visitados. Si es mayor entonces se castiga el recorrido. La fórmula para premiar y castigar es **premio o castigo = sumatoria\_de\_esfuerzo -  $(\text{mínimo} + \text{máximo})/2$** . El resultado a las casillas que están en nuestra lista de visitados. De esta manera las ponderaciones para el agente actual aumentan o disminuyen.

## Recorridos

En el programa se utilizó diferentes variaciones de recorridos para ver cómo se ajustaban las ponderaciones para cada agente y ver cómo afectaba el algoritmo de dijkstra. Si hacíamos 100 recorridos para cada agente no se los agentes a veces rodeaban para encontrar la salida. Si hacía 10,000 recorridos entonces se sobre entrena la red y los agentes rodeaban mucho aunque la salida estuviera a 10 casillas. Es muy interesante porque normalmente Momo y Pirolo rodeaban más y Lucas va directa. Pero después de analizar el mapa, se puede ver que hay más pasto/plano que es su fuerte. Es por eso que va directo.

## Dijkstra

Para aplicar dijkstra se creó otro Vector de 3 niveles  $\text{Vector} < \text{Vector} < \text{Vector} < \text{Long} > > >$  en donde cada casilla contiene las casillas que están al su alrededor. Esto significa que guardamos pares de números que son las coordenadas "X Y". Los pesos que utiliza dijkstra es la ponderación actual de la casilla y se multiplica por el esfuerzo que tiene que ser el agente dependiente del terreno que tenga la casilla. Entonces aplicamos dijkstra 3 veces una para cada agente. En la siguiente figura III podrán ver las tablas de ruteo que genera dijkstra para cada agente

```

Mombos Dijkstra Routing Table
1,1 1,2 -1 1,4 0,5 1,6 -1 1,7 1,7 0,8
-1 1,2 0,3 1,4 0,5 -1 1,7 1,8 -1 -1
1,1 1,2 -1 1,4 1,4 2,6 1,7 -1 1,7 2,8
-1 3,2 2,3 -1 2,3 2,6 2,6 -1 2,8 -1
5,1 -1 5,3 3,2 5,3 -1 5,5 5,7 5,7 4,8
-1 5,2 4,3 4,3 5,3 5,4 -1 4,6 -1 -1
6,1 5,2 -1 5,3 5,3 -1 5,7 -1 5,7 -1
6,1 -1 6,3 6,3 6,3 6,4 -1 6,6 6,8 6,8
9,0 9,0 7,3 -1 -1 7,4 7,7 7,7 7,7 7,9
9,1 8,2 8,2 8,2 -1 -1 -1 8,7 8,7 8,8

Pirollos Dijkstra Routing Table
0,1 1,2 -1 1,4 1,3 1,6 -1 1,8 0,7 0,8
-1 1,2 1,3 1,4 0,5 -1 0,7 0,7 -1 -1
1,1 1,2 -1 1,4 1,3 1,6 1,6 -1 1,7 1,8
-1 4,0 4,3 -1 2,3 4,4 3,5 -1 4,9 -1
5,1 -1 4,3 3,4 3,4 -1 3,5 5,7 5,7 4,8
-1 4,2 4,3 4,3 4,3 4,4 -1 6,6 -1 -1
5,1 6,0 -1 5,3 6,3 -1 7,7 -1 7,7 -1
6,0 -1 6,3 6,3 6,3 8,5 -1 8,6 7,7 7,8
7,0 7,2 7,2 -1 -1 7,4 8,5 7,7 7,7 9,8
8,1 8,1 9,3 8,2 -1 -1 -1 9,8 8,7 9,8

Lucas Dijkstra Routing Table
0,1 1,2 -1 1,4 1,3 1,6 -1 1,8 0,7 0,8
-1 1,2 1,3 1,4 0,5 -1 0,7 0,7 -1 -1
1,1 1,2 -1 1,4 1,3 1,6 1,6 -1 1,7 1,8
-1 4,0 4,3 -1 2,3 4,4 3,5 -1 4,9 -1
5,1 -1 4,3 3,4 3,4 -1 3,5 5,7 5,7 4,8
-1 4,2 4,3 4,3 4,3 4,4 -1 6,6 -1 -1
5,1 6,0 -1 5,3 6,3 -1 7,7 -1 7,7 -1
6,0 -1 6,3 6,3 6,3 8,5 -1 8,6 7,7 7,8
7,0 7,2 7,2 -1 -1 7,4 8,5 7,7 7,7 9,8
8,1 8,1 9,3 8,2 -1 -1 -1 9,8 8,7 9,8

```

Figura III: Dijkstra Routing Table

Cada Par de numeros son las coordenadas para la casilla a la que debe ir el Agente. Los -1 significan que son muros o es la salida. Las coordenadas estan ordenadas en Y,X en vez de X,Y ésta de está manera por la manera que funcionan los vectores. Esta matriz que se genero es un arbol endonde cada conjunto de numeros apunta al padre hasta llegar a la raiz que es la salida.

## Elección de lenguaje de programación

Se eligio programar en el lenguaje de programación Java utilizando Eclipse como IDE porque ya es un lenguaje de programación que domino a cierto grado. Y porque iba a facilitar la creación del interfaz de esta manera no mas me enfocaba en la programación del laberinto y de los algoritmos que iba a manejar. Lo unico que si se complico es de como escalar (scale) las imagenes al tamaño que yo quiera. Investigue y se encontro en el sitio de [stackoverflow](https://stackoverflow.com) como

escalar las imagenes y despues como poner las imagenes en el JLabel.

## Diagrama de Clases

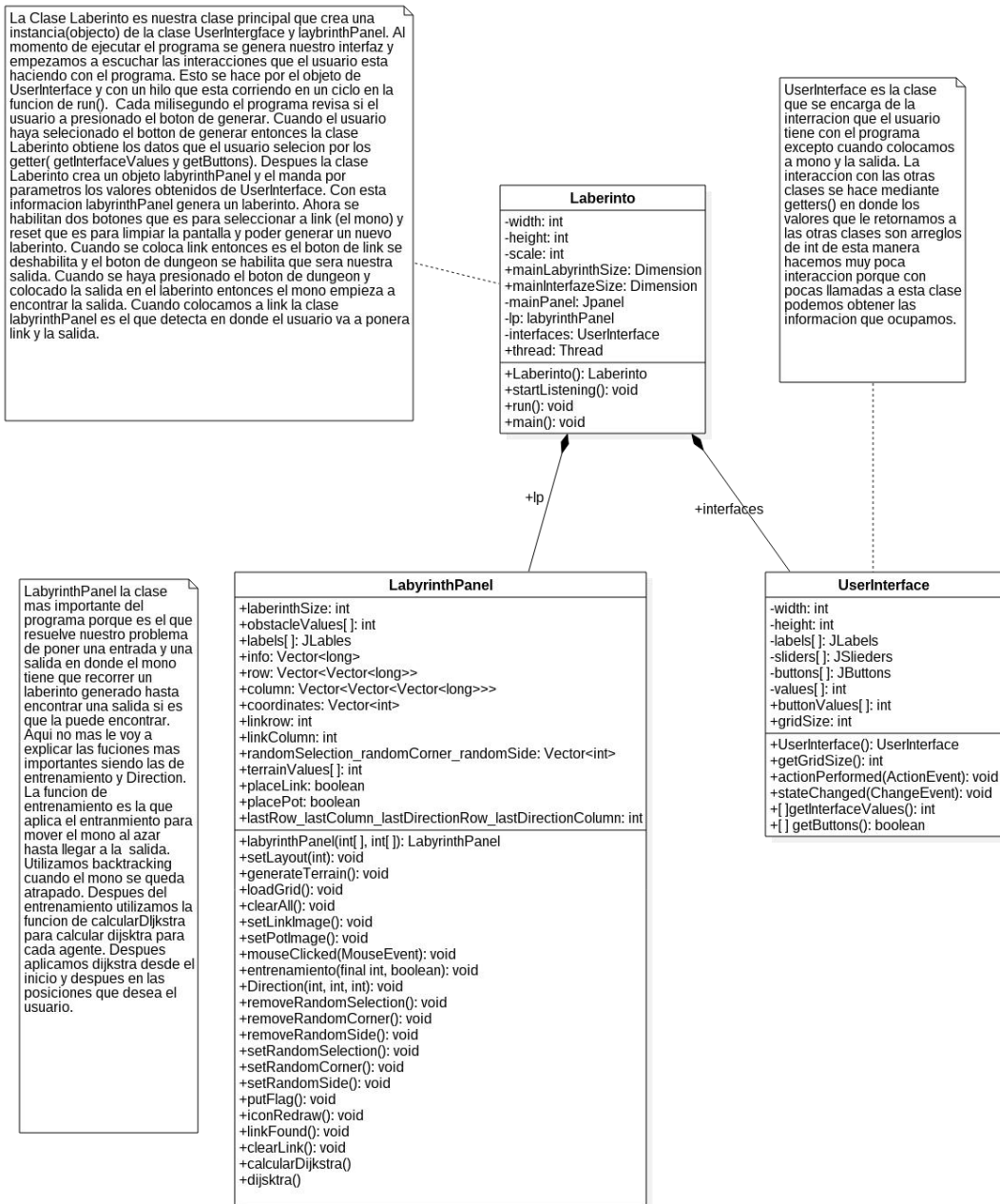


Figura IV: Diagrama de Clases



## Conclusión

Se puede concluir que el algoritmo de dijkstra es buenísimo por que te encuentra la ruta mas corto. Aunque hayamos modificado las ponderaciones del laberinto los agentes tienden a ir a casillas en donde le favorecan el terreno. Esta manera de aprender es muy util porque no siempre se puede saber hacia donde ir entonces dejamos que el agente se mueva al azar para que aprenda por si mismo.

## Codigo

```
public void ponderacion(Vector<Double> mono,int ponderacion)
{
    int divisor = visitedList.size();

    for(int i = 0; i < visitedList.size();i++)
    {
        int y = visitedList.get(i).get(0);
        int x = visitedList.get(i).get(1);

        Long originalPonderacion = column.get(y).get(x).get(ponderacion);
        Long terrain = column.get(y).get(x).get(ids.terrainType.getValue());
        mono.setElementAt(mono.elementAt(0)+(/*originalPonderacion*/mono.elementAt(terrain_to_index(terrain))), 0);
        //System.out.println("Time it took: " + originalPonderacion*mono.elementAt(terrain_to_index(terrain)));
    }

    double esfuerzo = mono.get(0);
    double promedio = (highestValue + lowestValue) / 2;
    if(esfuerzo < promedio && esfuerzo > lowestValue)
    {
        Long ajuste = (long) (esfuerzo - promedio)/100000000.;
        for(int i = 0; i < visitedList.size();i++)
        {
            int y = visitedList.get(i).get(0);
            int x = visitedList.get(i).get(1);

            Long originalPonderacion = column.get(y).get(x).get(ponderacion);
            //System.out.println("Before: " + column.get(y).get(x).get(ponderacion));
            column.get(y).get(x).set(ponderacion, (long) (originalPonderacion + (long)ajuste));
            //System.out.println("After: " +column.get(y).get(x).get(ponderacion));
        }
        System.out.println("Premio: " + ajuste);
    }
}

void bootStrap(Vector<Double> mono,int ponderacion)
{
    for(int i = 0; i < visitedList.size();i++)
    {
        int y = visitedList.get(i).get(0);
        int x = visitedList.get(i).get(1);

        Long originalPonderacion = column.get(y).get(x).get(ponderacion);
        Long terrain = column.get(y).get(x).get(ids.terrainType.getValue());
        mono.setElementAt(mono.elementAt(0)+(/*originalPonderacion*/mono.elementAt(terrain_to_index(terrain))), 0);
    }
    lowestValue = highestValue = media = mono.get(0);
    mono.set(0,(double)0);
}
```

*Figura V y VI:*

### Bootstrap y la funcion para premiar o castigar las ponderaciones.



```

public void dijkstra()
{
    int startC = staticlinkC;
    int startR = staticlinkR;
    int columnBefore = staticlinkC;
    int rowBefore = staticlinkR;
    int tempR;
    int tempC;

    labels[((startC+startR*obstacleValues[0]))].setIcon(blueLinkStartIcon);
    tempR = MomboTreePath.get(staticlinkR).get(staticlinkC).get(0);
    tempC = MomboTreePath.get(staticlinkR).get(staticlinkC).get(1);
    staticlinkR = tempR;
    staticlinkC = tempC;
    sleepFunction();

    while(MomboTreePath.get(staticlinkR).get(staticlinkC).get(0) != -1)
    {
        iconRedraw(columnBefore,rowBefore,staticlinkC,staticlinkR,1);
        columnBefore = staticlinkC;
        rowBefore = staticlinkR;
        tempR = MomboTreePath.get(staticlinkR).get(staticlinkC).get(0);
        tempC = MomboTreePath.get(staticlinkR).get(staticlinkC).get(1);
        staticlinkR = tempR;
        staticlinkC = tempC;
    }
    iconRedraw(columnBefore,rowBefore,staticlinkC,staticlinkR,1);
    linkFound();
}

```

*Figura VII:*

Utilizando Dijkstra para mover el Agente hacia la salida

```

public void calculateDijkstra()
{
    System.out.println("Dijkstra");
    ruta();
    columnDijkstra = new Vector<Vector<Vector<Long>>>();
    for(int x = 0; x < column.size();x++)
    {
        rowDijkstra = new Vector<Vector<Long>>();
        for(int y = 0; y < column.get(x).size();y++)
        {
            if(column.get(x).get(y).get(0) != 10)
            {
                infoDijkstra = new Vector<Long>();
                infoDijkstra.addElement(column.get(x).get(y).get(0));
                infoDijkstra.addElement(column.get(x).get(y).get(1));
                infoDijkstra.addElement(column.get(x).get(y).get(2));
                infoDijkstra.addElement(column.get(x).get(y).get(3));
                addSurroundingPositions(x,y);
            }
            else
            {
                infoDijkstra = new Vector<Long>();
                infoDijkstra.add((long) 10);
            }
            rowDijkstra.addElement(infoDijkstra);
        }
        columnDijkstra.addElement(rowDijkstra);
    }

    MomboQueue = new Vector<Vector<Integer>>();
    PiroloQueue = new Vector<Vector<Integer>>();
    LucasQueue = new Vector<Vector<Integer>>();
    tempInfo = new Vector<Integer>();
}

```

```

while(!MomboQueue.isEmpty())
{
    int currentX = MomboQueue.get(0).get(0);
    int currentY = MomboQueue.get(0).get(1);
    MomboQueue.remove(0);
    if(MomboList[currentX][currentY] == false)
    {
        MomboList[currentX][currentY] = true;
        //Get the number of connections it has
        // Since columnDijkstra has the terrain type and 3 different weights
        // I eliminate subtract those leaving me only the connections
        // The connection are in pairs which is why I divide by 2 to either get
        // connections 8,5,3 IN the Middle , Side or the corner
        int connections = (columnDijkstra.get(currentX).get(currentY).size()-4)/2;
        for(int i = 0; i < connections;i++)
        {
            //Get Peso for the direction selected
            Long newX = columnDijkstra.get(currentX).get(currentY).get(4+2*i);
            Long newY = columnDijkstra.get(currentX).get(currentY).get(5+2*i);
            Long originalPonderacion = column.get(newX.intValue()).get(newY.intValue()).get(1);
            Long terrain = column.get(newX.intValue()).get(newY.intValue()).get(ids.terrainType.getValue());
            Long peso = (long) (originalPonderacion* Mombo.elementAt(terrain_to_index(terrain)));

            if(column.get(newX.intValue()).get(newY.intValue()).get(0) != 10 &&
               MomboList[newX.intValue()][newY.intValue()] == false)
            {
                relajacionMombo(currentX,currentY,newX.intValue(),newY.intValue(),peso);
            }
        }
    }
}
}

```

```

void relajacionMombo(int currentX,int currentY,int newX,int newY,Long peso)
{
    if(distanceMatrixMombo[currentX][currentY] + peso < distanceMatrixMombo[newX][newY]
        || distanceMatrixMombo[newX][newY] == -10)
    {
        distanceMatrixMombo[newX][newY] = (int) (distanceMatrixMombo[currentX][currentY] + peso);
        previousMatrixMombo[newX][newY] = Integer.toString(currentX) + "," + Integer.toString(currentY);
        tempInfo = new Vector<Integer>();
        tempInfo.add(newX);
        tempInfo.add(newY);
        tempInfo.add(distanceMatrixMombo[newX][newY]);
        sortQueue(tempInfo,MomboQueue);
    }
}

```

*Figura IX, X y XI:  
Calculamos dijkstra para todos los agentes*