I

ÍNDICE

1. Objetivo de la práctica.

Esta práctica consiste en hacer que Bender, nuestro robot representado con una X, consiga llegar hasta la meta, representada con un $.

Para poder llegar tenemos dos opciones. El método run() el cual consiste en seguir unas direcciones en un orden concreto en un orden (S,E,N,W) o usar el método bestRun(), este método consiste en encontrar el camino más corto para llegar hasta la meta.

El robot y la meta están representados en un mapa, el cual tendrá Teletransportadores los cuales cambian la posición del robot en el mapa, e inversores, los cuales cambian el orden de prioridad de movimientos al robot.

GitHub: https://github.com/bcastanerc/Bender.
Diagrama ULM completo de Bender: diagramaClaseBender.

2. Objeto Cell.

El objeto Cell representa cada celda del mapa, esta celda tiene los atributos:

- posY: define la posición en el eje Y (altura) dentro del array multidimensional.
- posX: define la posición en el eje X (diagonal) dentro del array multidimensional.
- character: define el carácter en la posición de la celda.
- goneBy: indica cuantas veces se ha pasado por esa celda para saber si el robot está en un bucle infinito.
- distanceFromRobotl: Este atributo solo se usa en el método bestRun(), define a qué distancia se encuentra del Robot.

```
// Cell run() attibutes.
private char character;
private int posY;
private int posX;
private int goneBy = 0;

//Cell bestRun() attibutes.
private int distanceFromRobot = -1;
```

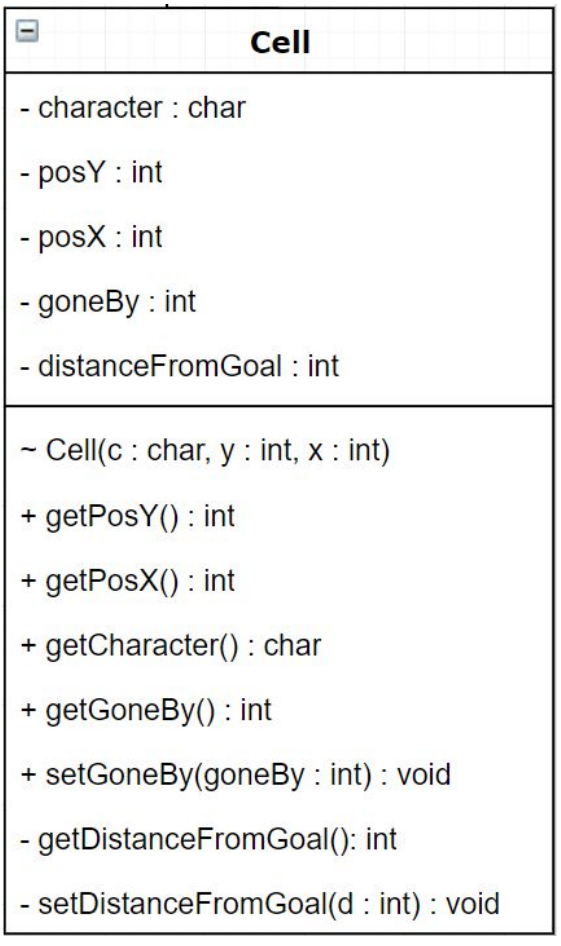
El objeto Cell dispone de los métodos:

- Cell(char c, int y, int x): Constructor que asigna los valores de la celda.
- getPosY(): Devuelve la posición Y.
- getPosX(): Devuelve la posicion X.
- getCharacter(): Devuelve el caracter de la celda.
- setGoneBy(int goneBy): Define las veces que se ha pasado por la celda.
- getGoneBy(): Devuelve las veces que se ha pasado por la celda.
- getDistanceFromRobotl: Devuelve la distancia hasta la meta.
- setDistanceFromRobotl: Define la distancia hasta la meta.

```
Cell(char c, int y, int x){
    this.character = c;
    this.posX = x;
    this.posY = y;
}

public int getPosY() { return posY; }
public int getPosX() { return posX; }

public char getCharacter() { return character; }

public void setGoneBy(int goneBy) { this.goneBy = goneBy; }
public int getGoneBy() { return goneBy; }

public int getDistanceFromGoal() { return distanceFromGoal; }
public void setDistanceFromGoal(int distanceFromGoal) { this.distanceFromGoal = distanceFromGoal; }
```

2.1 Diagrama UML del Objeto.

Diagrama UML de Cell, atributos y métodos.



3. Objeto Robot.

El objeto Robot simula a Bender moviéndose por el mapa, para lograr que simula el recorrido dispone de estos atributos.

- robotY: Posición del Robot en el eje Y.
- robotX: Posición del Robot en el eje X
- directions: Un array de caracteres con las direcciones en un orden específico S, E, N, W.
- invertedDirections: Un array de caracteres con las direcciones invertidas N, W, S, E.
- inverted: Determina el estado del Robot y que array de direcciones usará.

```
// Robot run() atributes.
private int robotX;
private int robotY;
private char[] directions = {'S','E','N','W'};
private char[] ivertedDirections = {'N','W','S','E'};
private boolean inverted;
```

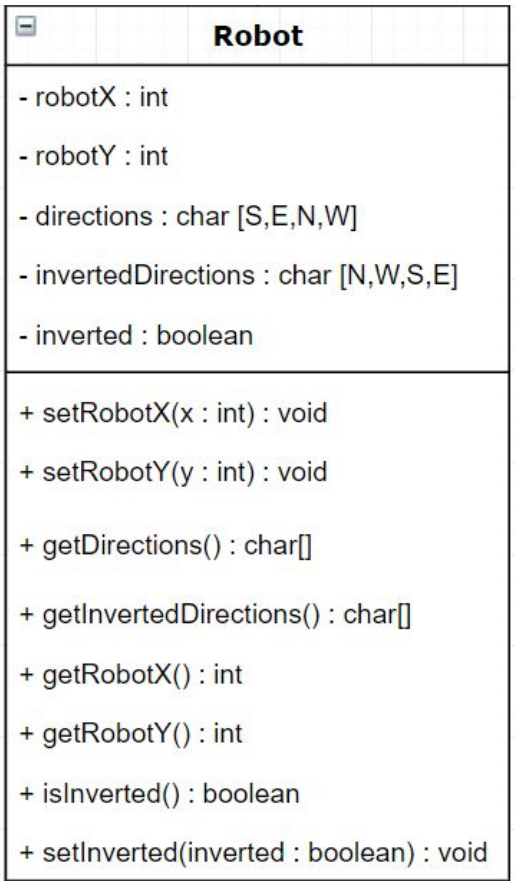
Para poder interaccionar con el objeto tenemos ciertos getters y setters los cuales nos dan acceso a los atributos privados, no dispone ningún método con lógica.

- setRobotY: Asigna la posición del robot en el eje de las Y.
- setRobotX: Asigna la posición del robot en el eje de las X.
- getDirections: Devuelve el array de direcciones.
- getInvertedDirections: Devuelve el array de direcciones invertidas.
- getRobotY: Devuelve la posición del Robot en el eje Y.
- getRobotY: Devuelve la posición del Robot en el eje X.
- isInverted: Devuelve true or false según si está invertido o no.
- setInverted: Asigna true or false al atributo inverted.

```
Robot(){}

public void setRobotX(int x) { this.robotX = x; }
public void setRobotY(int y) { this.robotY = y; }

public char[] getDirections() { return directions; }
public char[] getIvertedDirections() { return ivertedDirections; }

public int getRobotX() { return this.robotX; }
public int getRobotY() { return this.robotY; }

public boolean isInverted() { return inverted; }
public void setInverted(boolean inverted) { this.inverted = inverted; }
```

3.1 Diagrama UML del Objeto.

Diagrama UML de Robot, atributos y métodos.



4. Objeto Teleport.

El objeto Teleport cambia de posición al Robot en cuanto lo pisa, al Robot se le asignará una nueva posición dependiendo de cual sea el teletransportador más cercano a el teletransportador actual.

Atributos del Objeto Teleport:

- posY: Posición del Teleport en el eje vertical.
- posX: Posición del Teleport en eje horizontal.
- nearTpY: Posición del Teleport más cercano en el eje vertical.
- nearTpX: Posición del Teleport más cercano en el eje horizontal.

```
private int posY;
private int posX;
private int nearTpY;
private int nearTpX;
```

Métodos de los que dispone el teletransportador:

- getPosX: Devuelve la posición del Teletransportador en el eje horizontal.
- getPosY: Devuelve la posición del Teletransportador en el eje vertical.

- getNearPosY: Devuelve la posición del Teletransportador más cercano en el eje vertical.
- getNearPosX: Devuelve la posición del Teletransportador más cercano en el eje horizontal.
- setNearPosY: Asigna la posición del Teletransportador más cercano en el eje vertical.
- setNearPosX: Asigna la posición del Teletransportador más cercano en el eje horizontal.
- defineNearestTp: Calcula cual será el Teletransportador más cercano.
- setAngles: Calcula a qué ángulo se encuentra un teletransportador en un plano cartesiano con relación al origen.

```
Teleport(int y, int x){
    this.posX = x;
    this.posY = y;
}

public int getNearTpX() { return nearTpX; }
public int getNearTpY() { return nearTpY; }
public void setNearTpX(int nearTpX) { this.nearTpX = nearTpX; }
public void setNearTpY(int nearTpY) { this.nearTpY = nearTpY; }

public int getPosY() { return posY; }
public int getPosX() { return posX; }

public void defineNearestTp(LinkedList<Teleport> teleportMapList){...}
public double setAngles(Teleport tOrigen, double y, double x){...}
```

4.1. Función defineNearest.

Para ahorrarnos tiempo de proceso y mejorar el rendimiento calcularemos cual es el más cercano y guardamos la posición de este como atributo del teletransportador de entrada.

Esto nos ahorra tener que hacer el cálculo cada vez que entramos en un Teleport, aunque si no usamos ninguno haremos un cálculo innecesario, sin embargo si tenemos un mapa muy grande en el que saltemos de teletransporte a teletransporte como lo tenemos calculado será más óptimo a la larga. Este cálculo se realiza una vez tenemos la lista de teleports completa y se llama a esta función desde el constructor de Bender.

Para hacer el cálculo de la distancia entre dos vectores usaremos la formula de Pitagoras, la distancia es la hipotenusa entre nuestros dos vectores. Si el tamaño de la hipotenusa es menor quiere decir que el Teletransportador está más cerca.

```
double x = tActual.getPosX()-this.getPosX();
double y = tActual.getPosY()-this.getPosY();
double auxDistance = Math.sqrt((Math.pow(y,2))+(Math.pow(x,2)));
```

Si la distancia entre dos teletransportadores es la misma tendremos que calcular el ángulo con el que se encuentra respecto al plano cartesiano, para calcular el ángulo necesitamos llamar a la función setAngles(). Si el ángulo del actual es menor que el anterior asignamos como teletransportador más cercano al actual.

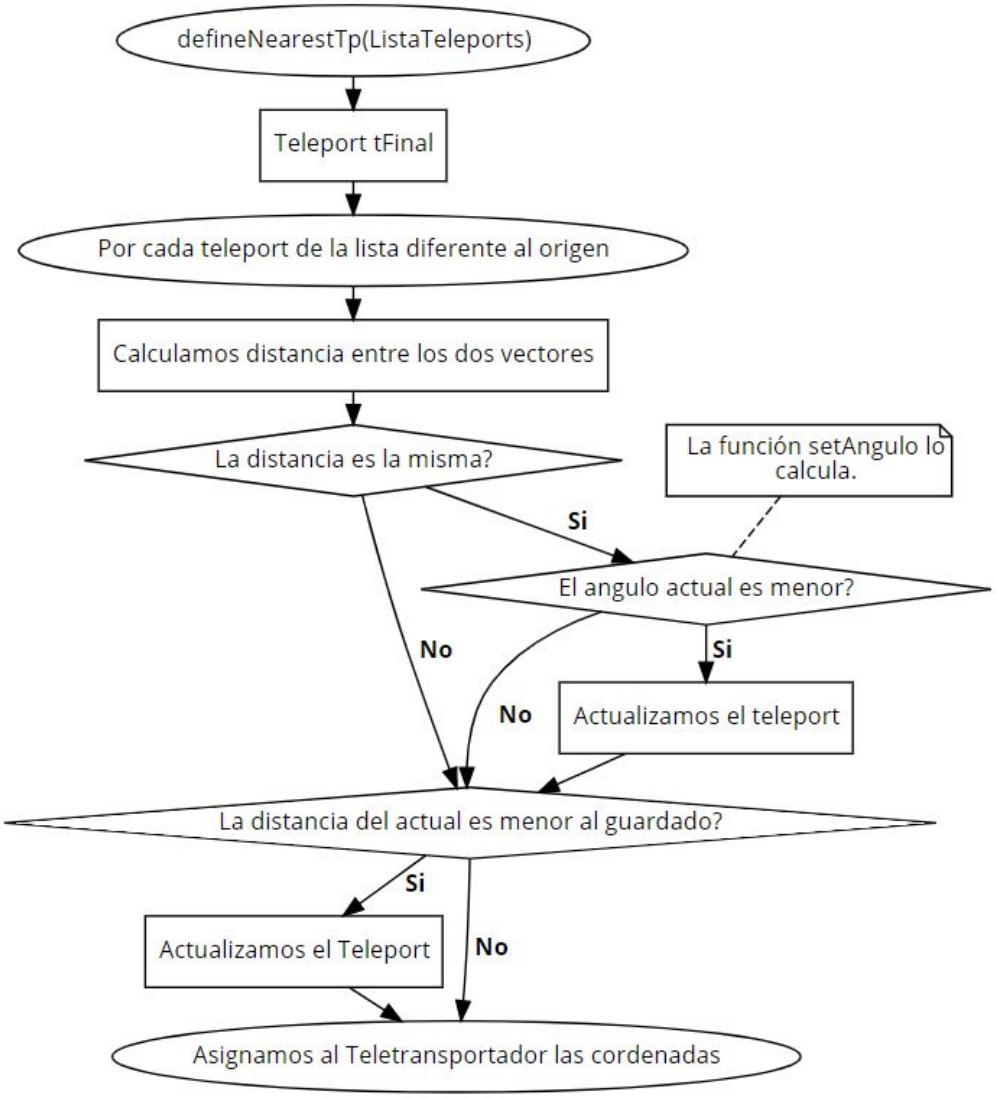
```
if (distancia == auxDistance){
    if (setAngles( tOrigen: this,tFinal.getPosY(),tFinal.getPosX())
            > setAngles( tOrigen: this,tActual.getPosY(),tActual.getPosX())){
        tFinal = tActual;
        distancia = auxDistance;
    }
```

Una vez hemos comparado con todos los teletransportadores de la lista asignamos al teletransportador origen los atributos NearPosY y NearPosX según las coordenadas del teletransportador final.

```
this.setNearTpY(tFinal.getPosY());
this.setNearTpX(tFinal.getPosX());
```

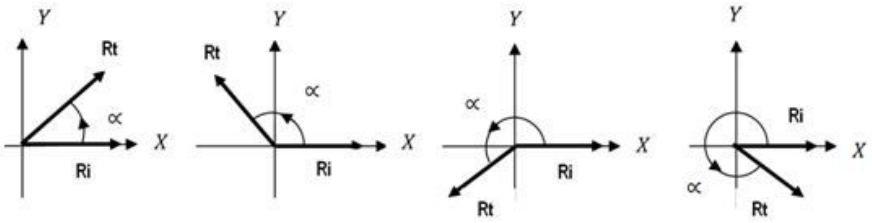
4.1.1 Diagrama de flujo de la función defineNearest.

Este diagrama de flujo representa a la función defineNearest.



4.2. Función setAngles.

Esta función va en conjunto con defineNearest(). Cuando defineNearest calcula la distancia a la que se encuentran los teleports, si hay dos a la misma distancia usará esta función para determinar cuál definir como más cercano. Nos tenemos que quedar con el más cercano a los 0 grados.

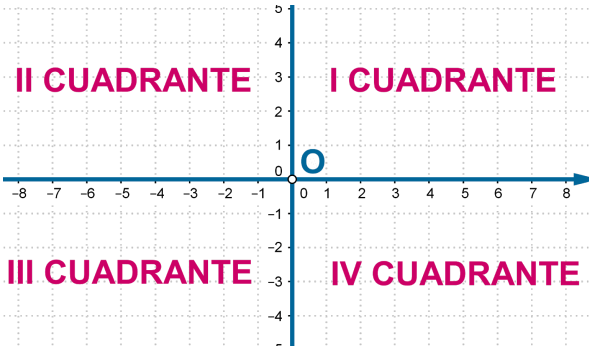


Si el vector con el cual se calcula está en el cuadrante 2 o 4 tendrá la x o la y en negativo, si multiplicamos x*y y el número es negativo sabemos que se encuentran en estos cuadrantes y se calculan y/x, si estan en los otros cuadrantes x/y.

```
if ((xActual*yActual)  < 0){
    anguloActual = Math.abs(Math.toDegrees(Math.atan(yActual/xActual)));
}else{
    // Si da negativo x/y.
    anguloActual = Math.abs(Math.toDegrees(Math.atan(xActual/yActual)));
}
```

Ahora tenemos que situar este punto en un cuadrante para calcular el ángulo. El cuadrante 1 va desde los 0º - 90º, el cuadrante 4 va desde los 90º - 180º, el 3 cuadrante va desde 180º - 270º y el cuadrante 4 va desde 270º - 360º. Estos grados determinan la prioridad que tendrá el teletransportador, cuanto menores sean los ángulos más prioridad.

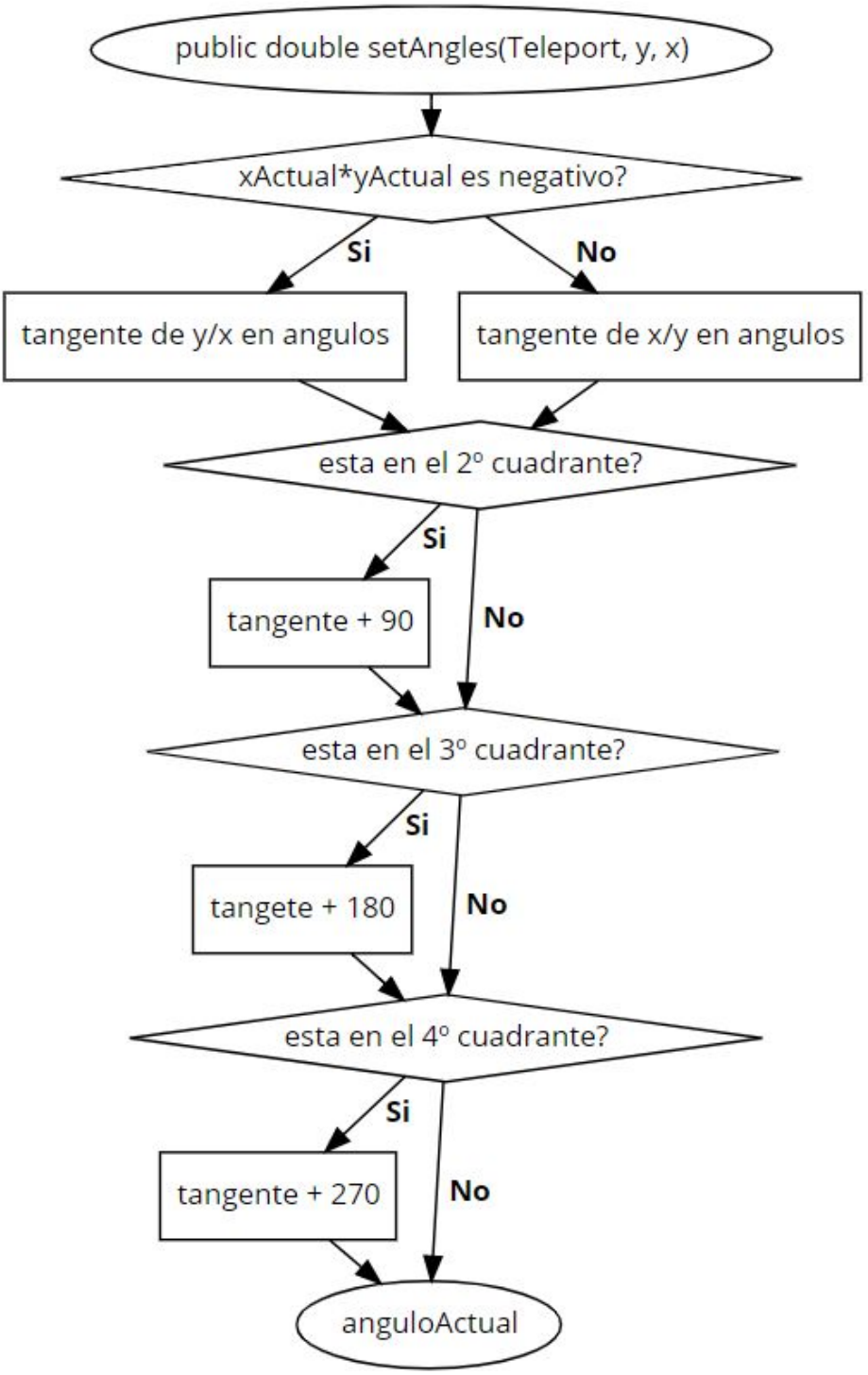
Los vectores en el cuadrante 1 tiene x e y positivos, el cuadrante 4 tiene y negativa y la x positiva, el cuadrante 3 tiene x e y negativa, el cuadrante 2 tiene la y positiva y la x negativa.




```
if(xActual >= 0 && yActual < 0)anguloActual+=90;
if (xActual < 0 && yActual < 0) anguloActual+= 180;
if (xActual < 0 && yActual >= 0) anguloActual+=270;
return anguloActual;
```

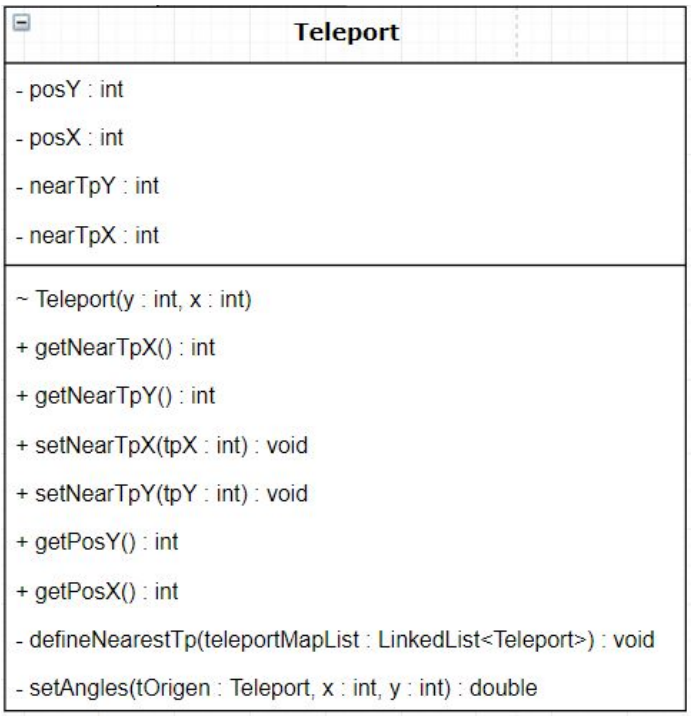
4.2.1 Diagrama de flujo de la función setAngles.

Este diagrama de flujo representa a la función setAngles.



4.3. Diagrama ULM del Objeto.

Diagrama UML de Teleport, atributos y métodos.



5. Objeto Goal.

El objeto Goal sólo es necesario si implementamos el método bestRun(), sino no hace falta guardar las coordenadas de la meta en el run() ya que comprobamos el carácter en la posición que se encuentra el Robot actualmente, atributos de Goal:

- posY: Posición de la meta en el eje vertical.
- posX: Posición de la meta en el eje horizontal.

```
private int posY;
private int posX;
```

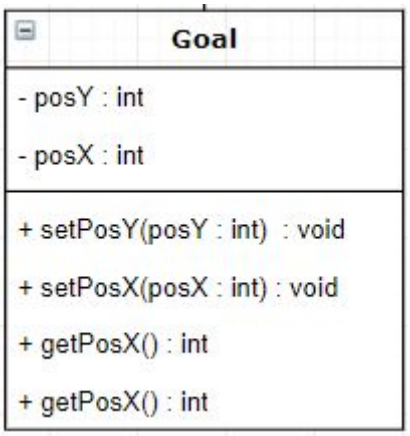
Métodos del Objeto Goal:

- getPosY: Devuelve la posición de la meta en el eje vertical.
- getPosX: Devuelve la posición de la meta en el eje horizontal.
- setPosY: Define la posición de la meta en el eje vertical.
- setPosX: Definela posición de la meta en el eje horizontal

```
public int getPosY() { return posY; }
public int getPosX() { return posX; }
public void setPosY(int posY) { this.posY = posY; }
public void setPosX(int posX) { this.posX = posX; }
```

5.1 Diagrama ULM del objeto Goal.

Diagrama UML de Goal, atributos y métodos.



6. Objeto Map.

El objeto Map es el encargado de crear el mapa, está compuesto por un array multidimensional de objetos tipo Cell, los atributos del objeto son:

- Cell [][]: Es un Array multidimensional de Celdas para representar cada coordenada de mapa.
- validMap: Determina si el mapa cumple las características necesarias para ser considerado un mapa válido.
-

```
private Cell[][] formedMap;
private boolean validMap = true;
```

Métodos del Objeto Map:

- Map: Constructor del Objeto.
- rightPaddingMap: Este método añade las paredes que faltan al mapa.
- mapIsValid: Determina si el mapa es válido.
- countColumns: Cuenta cuantos caracteres hay en la fila más larga del mapa.
- toString: Override a toString de Object, devuelve el mapa en String.
- getFormedMap: Devuelve el Array multidimensional de Celdas.
- isValidMap: Devuelve un boolean, true si el mapa es valido, false inválido.

```
Map(@NotNull String map, Robot rb, LinkedList<Teleport> teleportMap, LinkedList<Cell> cellMap, Goal goal){...}
@NotNull
private String rightPaddingMap(String line, int totalCharacters){...}
private boolean mapIsValid(@NotNull String map){...}
private int countColumns(@NotNull String strMap){...}
@Override
public String toString(){...}
public Cell[][] getFormedMap() { return formedMap; }
public boolean isValidMap() { return validMap; }
```

6.1. Constructor del objeto Map.

En el constructor de Map es donde se crea el mapa entero, para poder formarlo necesitamos el mapa en forma de String, el Robot, la meta, una lista para guardar Teleports y otra lista para guardar las celdas del Teleport.

Lo primero que hace el constructor es dividir el mapa en forma de String por los \n que separan cada fila y meterlos en un Array.

```
String[] arr = map.split( regex: "\n");
```

A continuación calculamos cual es la fila más larga mediante la función countColumns y si hay filas más cortas que la más larga llamamos a la función rightPaddingMap para rellenar con paredes los huecos vacíos.

```
int maxColumnes = countColumns(map);
for (int i = 0; i < arr.length; i++) {
    if (arr[i].length() < maxColumnes){
        arr[i] =  rightPaddingMap(arr[i], totalCharacters: maxColumnes-arr[i].length());
    }
}
```

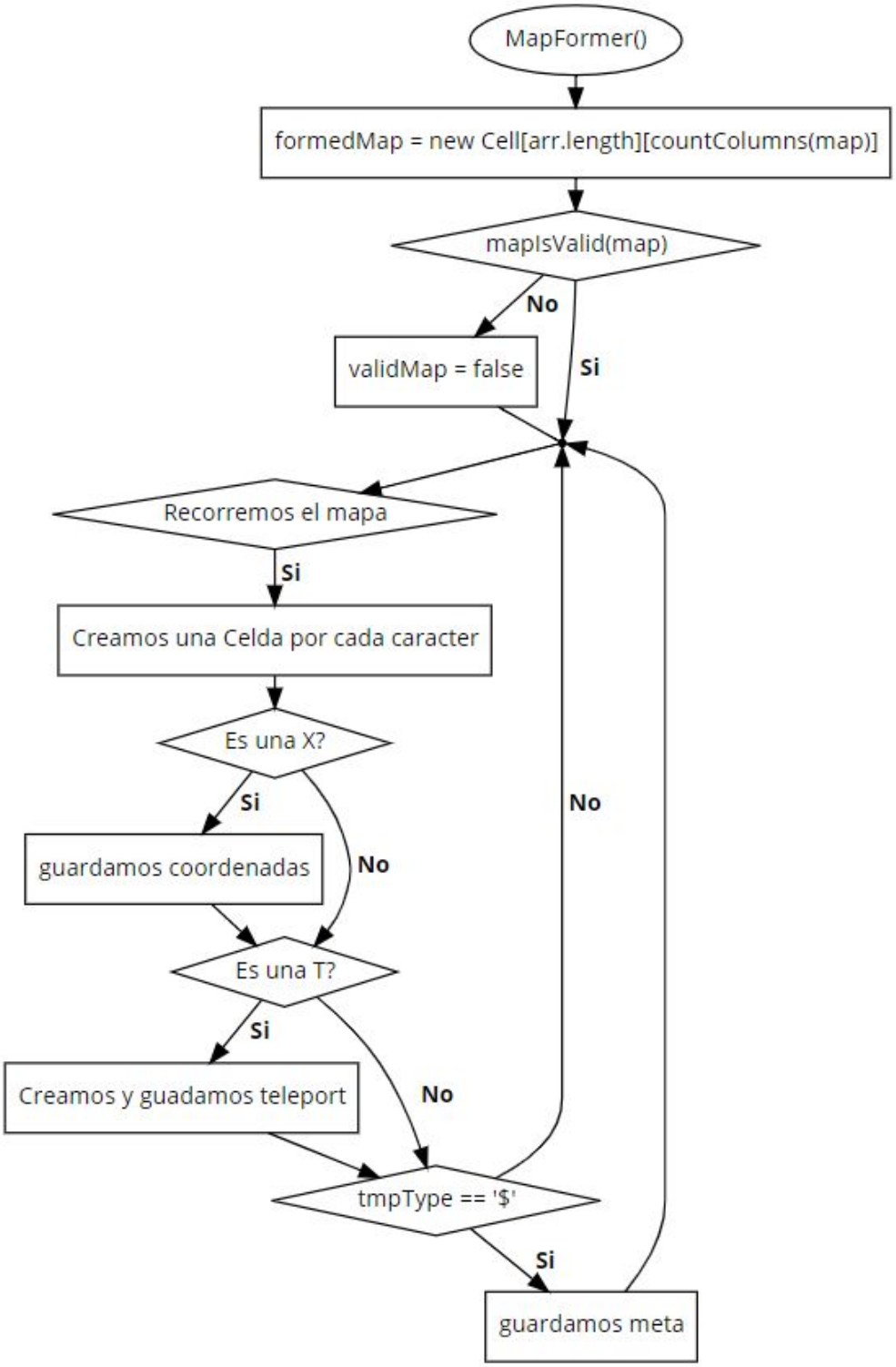
Una vez tenemos el String listo creamos el Array multidimensional y si el mapIsValid la recorremos a la vez que el String y creamos una celda por cada caracter del String, a esta celda se le asigna la posición y el carácter de esta. Si el carácter que se encuentra es una X se le asigna la posición al Robot, si es una T, se le asignan las coordenadas y se guarda en la lista de Teleports y si es un $ se le asigna la posición.

En caso de que el mapa no sea válido simplemente se le asignará false al atributo isValid y no hace falta recorrer el String ya que no usaremos el mapa.

```
    formedMap = new Cell[arr.length][countColumns(map)];

    if(mapIsValid(map)){
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length(); j++) {
                char tmpType = arr[i].charAt(j);
                formedMap[i][j] = new Cell(arr[i].charAt(j),i ,j);

                if (tmpType == 'X'){...}

                if (tmpType == 'T'){...}

                if (tmpType == '$'){...}
            }
        }
    }else validMap = false;
}
```

6.1.1. Diagrama de flujo del constructor del Map.

Este diagrama de flujo representa el constructor de Map.



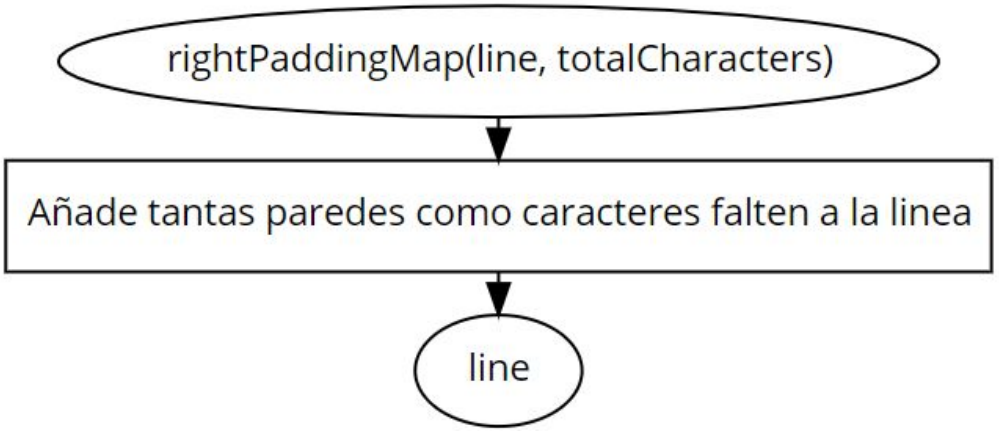
6.2. función rightPaddingMap.

La función rightPaddingMap añade paredes a la línea que entra por parámetro según el número de entrada. Esta función la he implementado para evitar posibles errores al generar un mapa con filas de longitudes diferentes, no es estrictamente necesaria.

```
private String rightPaddingMap(String line, int totalCharacters){
    line = line + "#".repeat(Math.max(0, totalCharacters));
    return line;
}
```

6.2.1. Diagrama de flujo de la función rightPaddingMap.

Este diagrama de flujo representa a la función rightPaddingMap.



6.3. función mapIsValid.

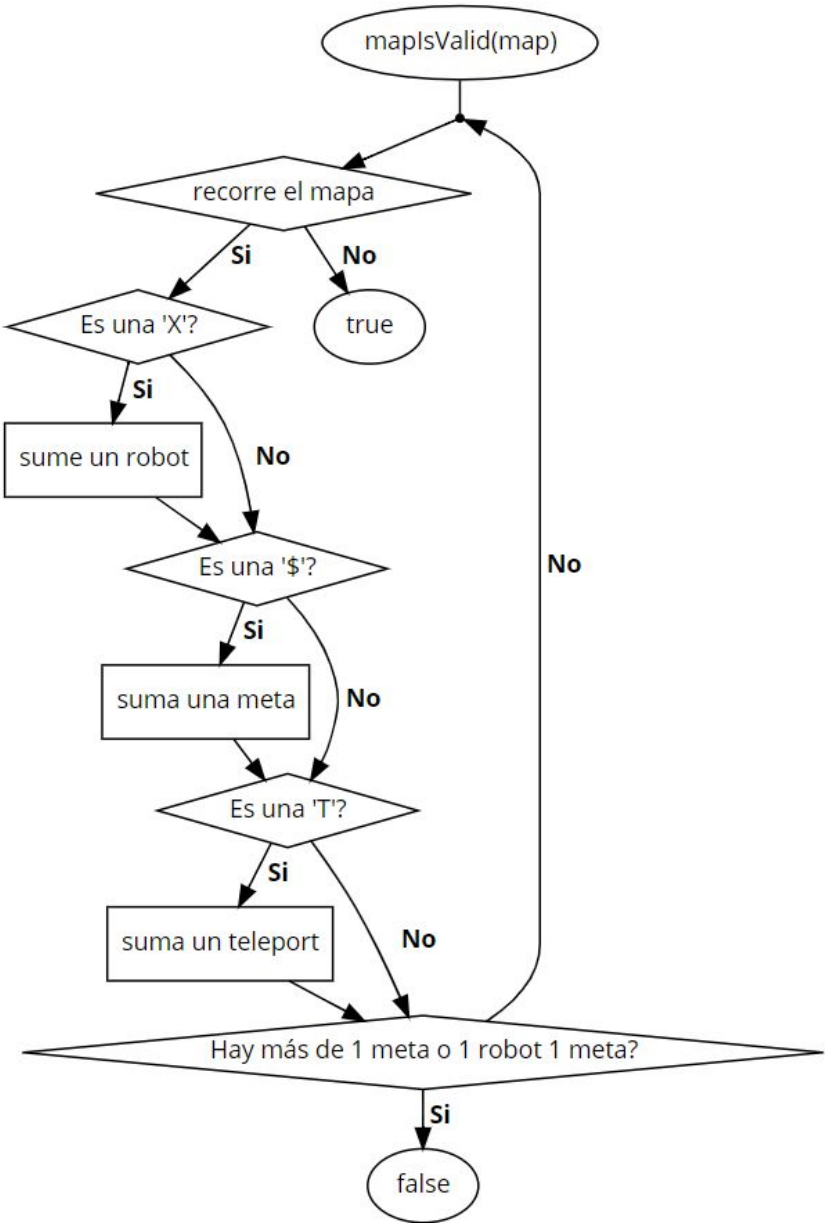
Esta función comprueba que el mapa sea considerado válido, el mapa es considerado válido si tiene solo 1 Robot, 1 Goal y 0 o más Teleports, no puede tener un número diferente del especificado aquí.

Para comprobarlo simplemente recorremos el mapa y contamos cuántos vemos de cada tipo hay.

```
for (int i = 0; i < map.length(); i++) {

    if (map.charAt(i) == 'X') robotCounter++;
    if (map.charAt(i) == '$') goalCounter++;
    if (map.charAt(i) == 'T') teleportCounter++;
    if (robotCounter > 1|| goalCounter > 1) return false;
}
return robotCounter == 1 && goalCounter == 1 && teleportCounter != 1;
```

6.3.1. Diagrama de flujo de la función mapIsValid.

Este diagrama de flujo representa a la función mapIsValid.



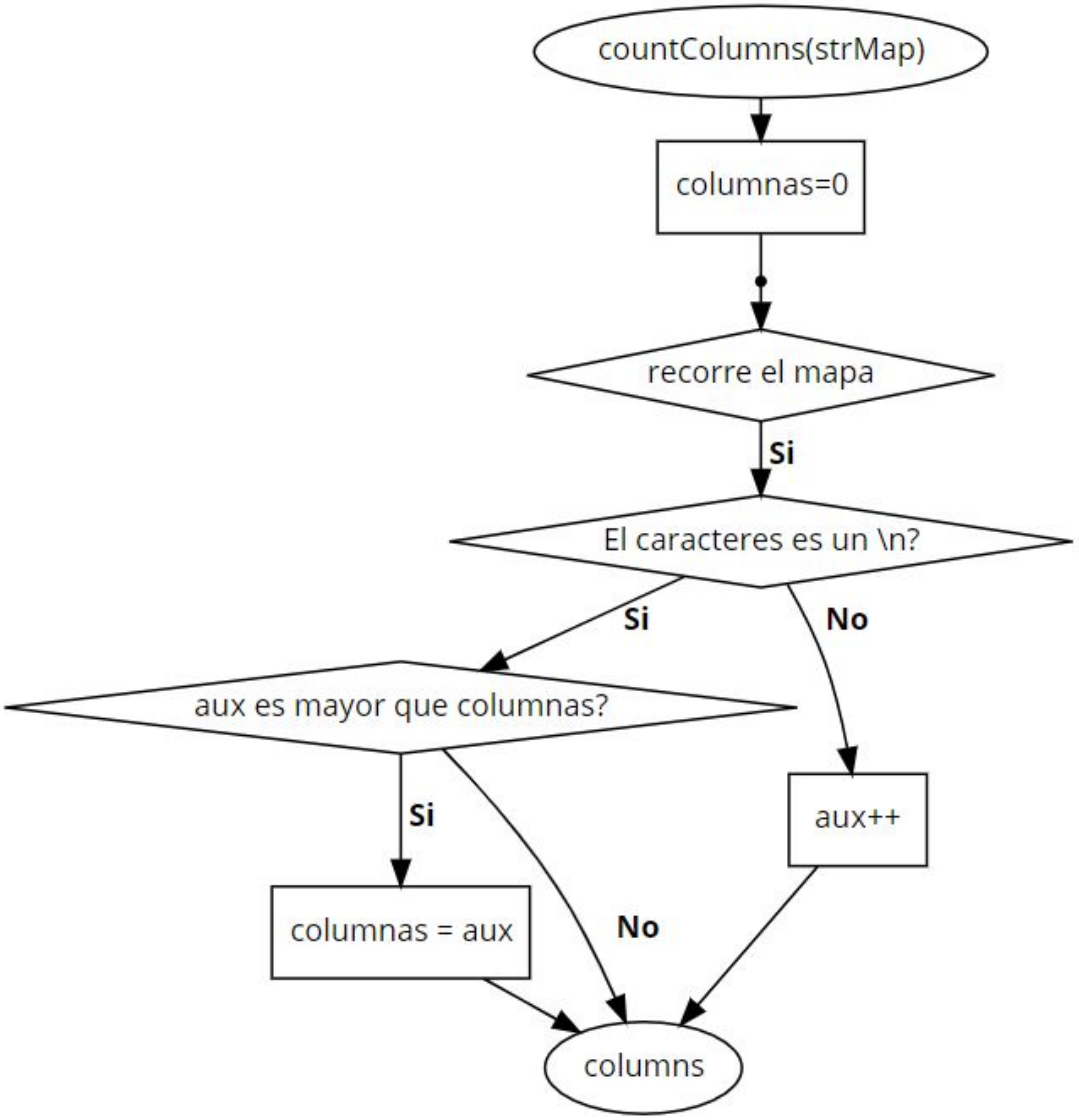
6.4. función countColumns.

La función countColums recorre el mapa y cuenta los caracteres que hay entre cada \n y guarda el mayor número de caracteres de entre todas las líneas del Array.

```
for (int i = 0, aux = 0; i < strMap.length(); i++) {
    if (strMap.charAt(i) != '\n') aux++;
    else{
        if (aux > columns){
            columns = aux;
        }
        aux=0;
    }
}
return columns;
```

6.4.1. Diagrama de flujo de la función countColumns.

Este diagrama de flujo representa a la función countColumns.



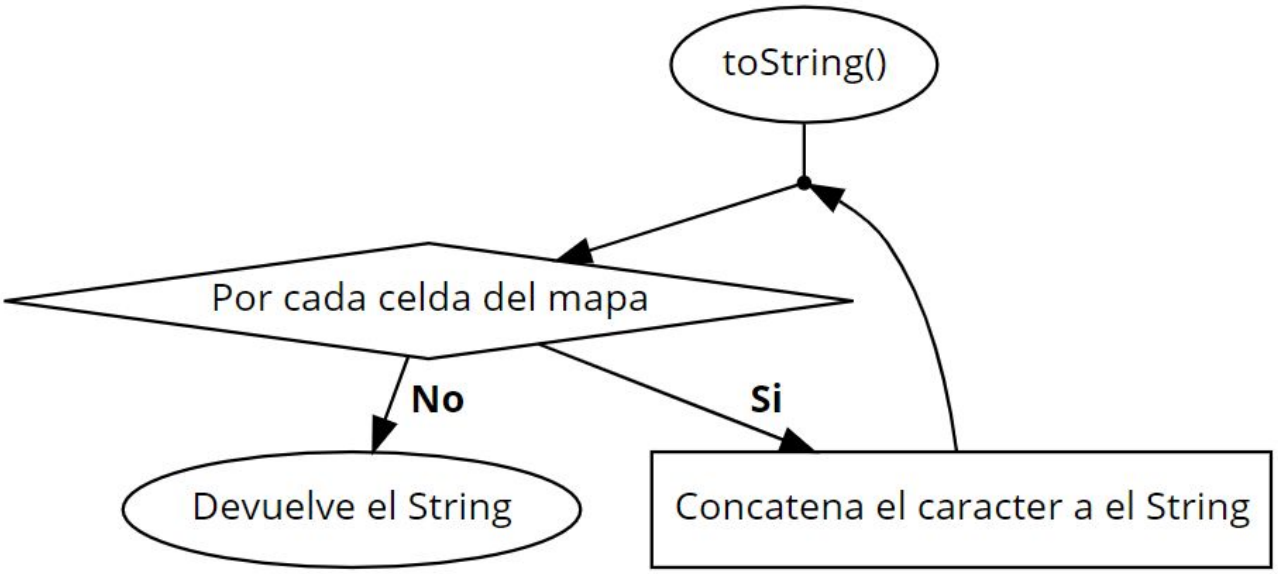
6.5. función toString.

Esta función hace un Override al metodo toString del Objeto de Java Object. Recorre cada Celda del mapa, por cada Celda guardará el carácter y lo concatena a un String el cual formará el mapa final.

```
StringBuilder strMap = new StringBuilder();
for (Cell[] cells : this.formedMap) {
    for (int j = 0; j < this.formedMap[0].length; j++) {
        strMap.append(cells[j].getCharacter());
    }
    strMap.append('\n');
}
return strMap.toString();
```

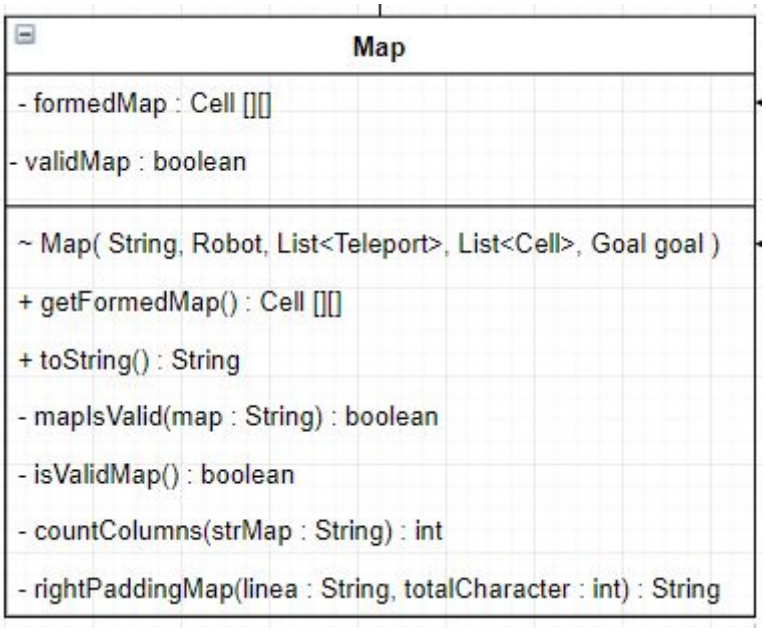
6.5.1. Diagrama de flujo de la función toString.

Este diagrama de flujo representa a la función toString.



6.6 Diagrama ULM del Objeto Map.

Este diagrama ULM representa los atributos y métodos del objeto Map.



7. Constructor de la clase Bender.

La clase Bender dispone de diferentes atributos, estos són: Map, Robot, LinkedList<Teleport> y LinkedList<Cell>.

```
private Map map;
private Robot rb = new Robot();
private LinkedList<Teleport> teleportMapList = new LinkedList<>();
private LinkedList<Cell> cellMapList = new LinkedList<>();
private Goal goal = new Goal();
```

A este constructor le entra por parámetro un String con el mapa, este String lo usaremos en el constructor para crear el mapa llamando a el constructor del Objeto Map. A este constructor le pasamos los objetos sin inicializar para asignarle los atributos.

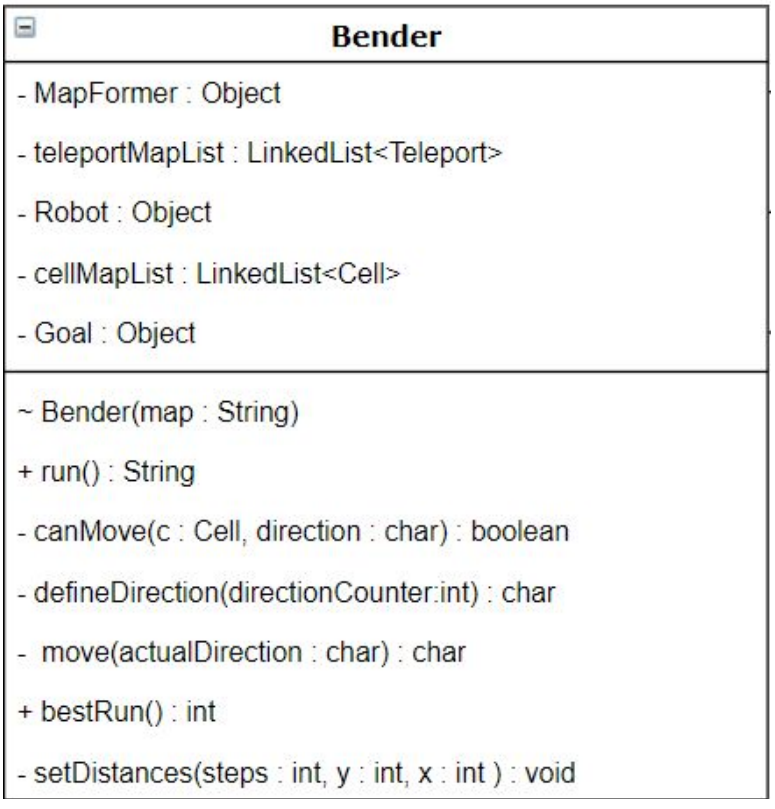
```
this.map = new Map(map,rb, teleportMapList, cellMapList, goal);
```

Una vez se ha creado el mapa tendremos los atributos del mapa, el robot, la meta una lista de Teletransportadores y de Celdas. Con la lista de Teletransportadores la recorreremos para asignar el teletransportador más cercano a cada uno de la lista.

```
for (Teleport actualTeleport : teleportMapList) {
    actualTeleport.defineNearestTp(teleportMapList);
}
```

7.1 Diragrama ULM de Bender.

Diagrama UML de Bender, atributos y métodos.



8. Método run.

El método run consiste en hacer que el Robot representado con una X, consiga llegar hasta la meta, representada con un $, hasta que no llegue o se defina que el mapa es infinito seguirá este método.

Cuenta cuantas veces se ha pasado por la misma celda, si el Robot pasa más de 8 veces por la misma casilla el Robot está en un bucle infinito.

```
// Suma obtiene la cantidad de veces que se ha pasado por la celda actual.
int goneByActualCell = this.map.getFormedMap()[this.rb.getRobotY()][this.rb.getRobotX()].getGoneBy();
this.map.getFormedMap()[this.rb.getRobotY()][this.rb.getRobotX()].setGoneBy(goneByActualCell+1);
```

```
if (this.map.getFormedMap()[this.rb.getRobotY()][this.rb.getRobotX()].getGoneBy() > 8 ) return null;
```

Para poder llegar tiene que seguir unas direcciones en un orden concreto (S,E,N,W) o (N,W,S,E) estas direcciones las define la función defineDirection, una vez se ha definido la dirección la función move mueve al Robot a la posición.

#	#	#	#	#	#	#	#	#	#
#	X								#
#	S								#
#	S								#
#	S			#	#	#		#	#
#	S			#	W	W	W	T	#
#	S			#	S	#	#	#	#
#	S	E	T	#	S	E	E	$	#
#	#	#	#	#	#	#	#	#	#

Al moverse el Robot actualiza en qué celda está, dependiendo de sobre qué celda este, Si el Robot está sobre un inversor se actualiza el estado.

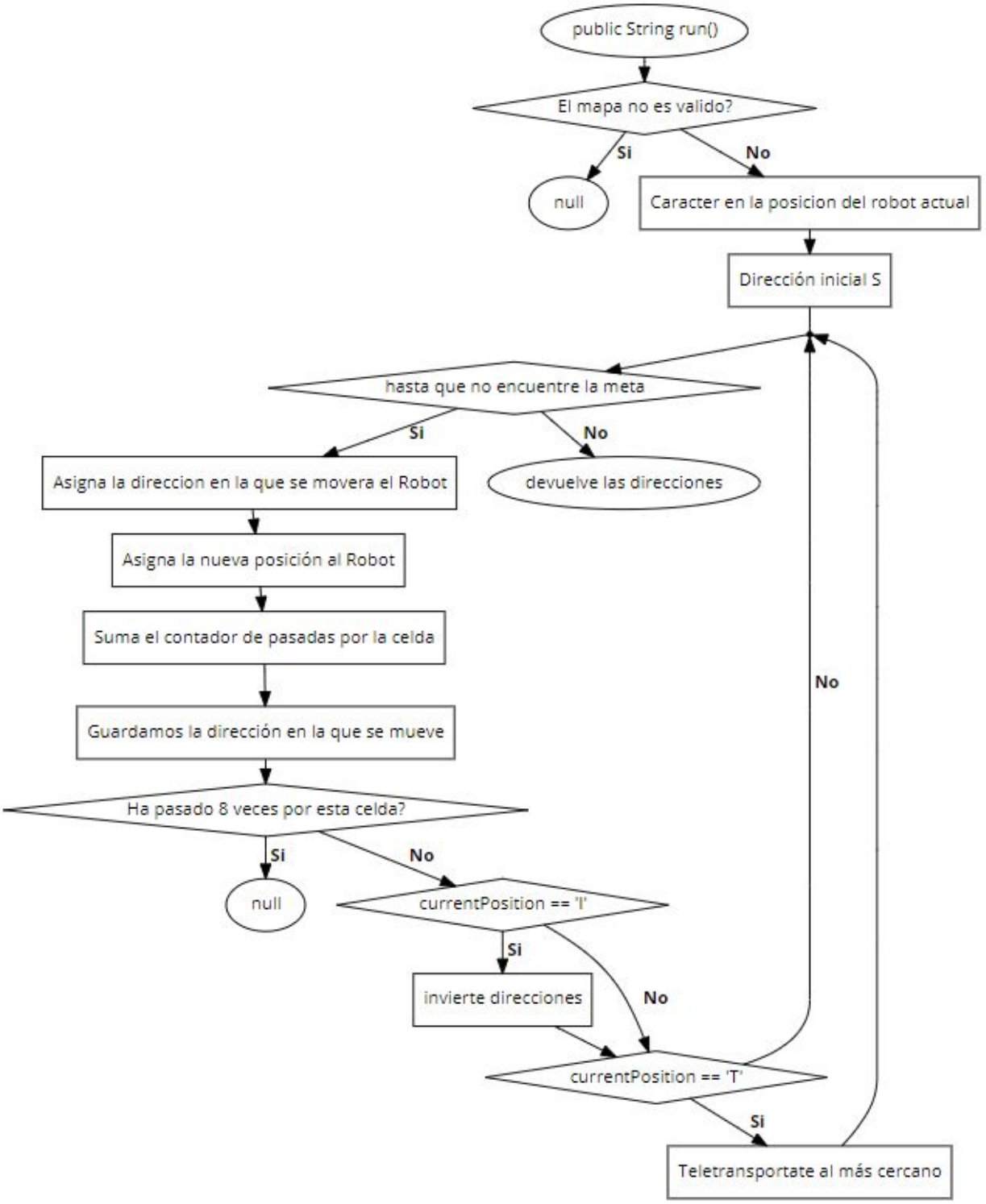
```
if (currentPosition == 'I') this.rb.setInverted(!this.rb.isInverted());
```

Si el caracter es un Teletransportador la posición del Robot se actualiza a las coordenadas nearTpY y nearTpX del Teletransportador en la celda actual.

```
if (currentPosition == 'T'){
    Teleport tpOut = teleportMapList.get(cellMapList.indexOf(this.map.getFormedMap()[this.rb.getRobotY()][this.rb.getRobotX()]));
    this.rb.setRobotX(tpOut.getNearTpX());
    this.rb.setRobotY(tpOut.getNearTpY());
}
```

8.1. Diagrama de flujo del método run.

Este diagrama de flujo representa el método run.



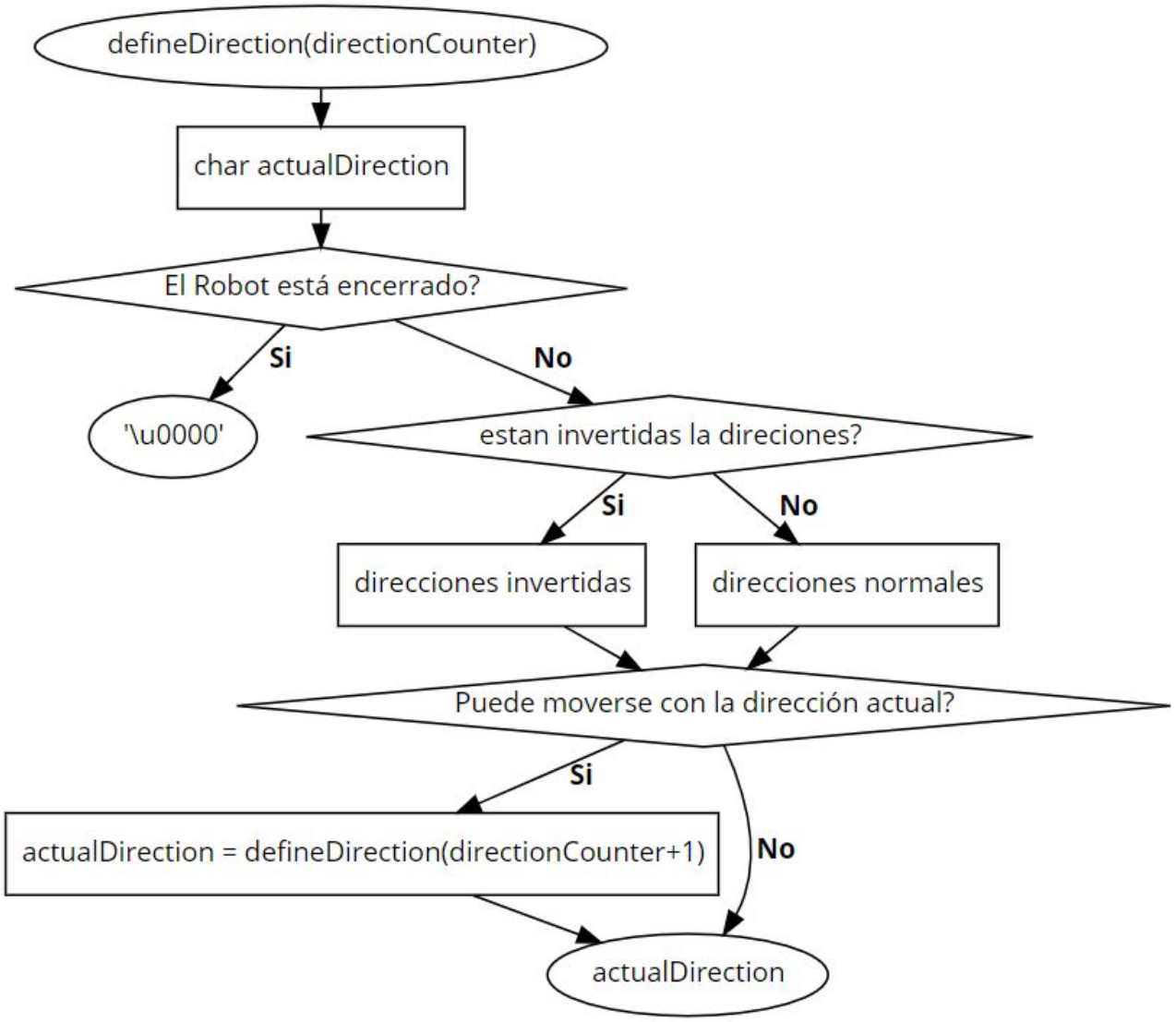
8.2. función defineDirection.

Esta función es la encargada de asignar las direcciones del Robot. Para esto tiene en cuenta si el estado del Robot está invertido, al estar invertido usará el atributo de Robot invertedDirections, sino usará directions. Esta función es recursiva ya que va probando por orden las direcciones hasta encontrar una que no le esté bloqueando el paso una pared.

```
private char defineDirection(int directionCounter){
    char actualDirection;
    if (directionCounter > 3) return '\u0000';
    if (this.rb.isInverted()){
        actualDirection = this.rb.getIvertedDirections()[directionCounter];
    }else{
        actualDirection = this.rb.getDirections()[directionCounter];
    }

    if (cantMove(this.map.getFormedMap()[this.rb.getRobotY()][this.rb.getRobotX()], actualDirection)){
        actualDirection = defineDirection( directionCounter: directionCounter+1);
    }

    return actualDirection;
}
```

8.2.1. Diagrama de flujo de la función defineDirection.

Este diagrama de flujo representa a la función defineDirection.



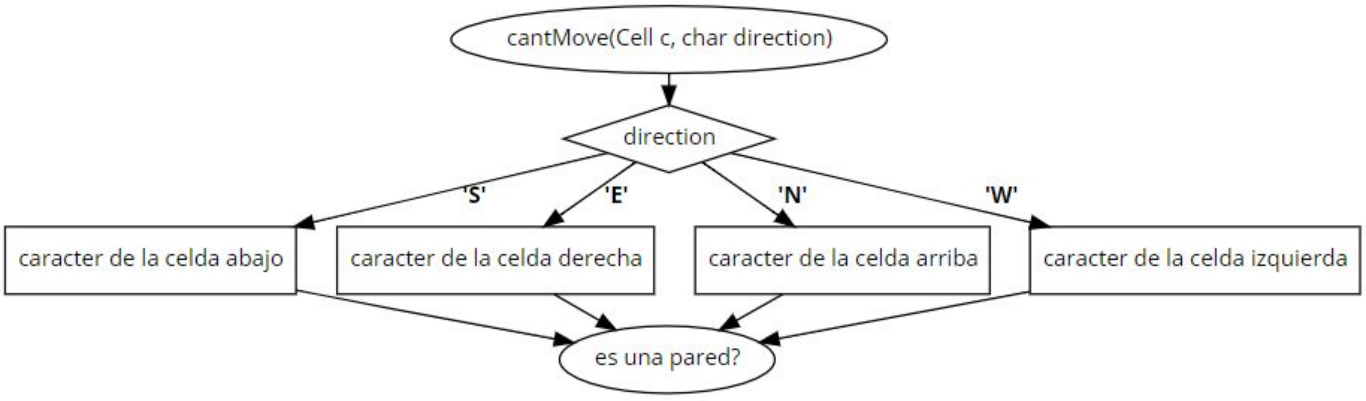
8.3. función cantMove.

La función cantMove funciona conjuntamente con la función defineDirection, esta prueba si la dirección que le ha asignado la función es válida, para determinar si esta función es válida comprueba que la posición una vez se mueva no sea una pared. Si el carácter es una pared devuelve true, si está despejado false.

```
switch (direction){
    case 'S':
        actual = this.map.getFormedMap()[c.getPosY()+1][c.getPosX()].getCharacter();
        break;
    case 'E':
        actual = this.map.getFormedMap()[c.getPosY()][c.getPosX()+1].getCharacter();
        break;
    case 'N':
        actual = this.map.getFormedMap()[c.getPosY()-1][c.getPosX()].getCharacter();
        break;
    case 'W':
        actual = this.map.getFormedMap()[c.getPosY()][c.getPosX()-1].getCharacter();
        break;
}
return actual == '#';
```

8.3.1. Diagrama de flujo de la función cantMove.

Este diagrama de flujo representa a la función cantMove.



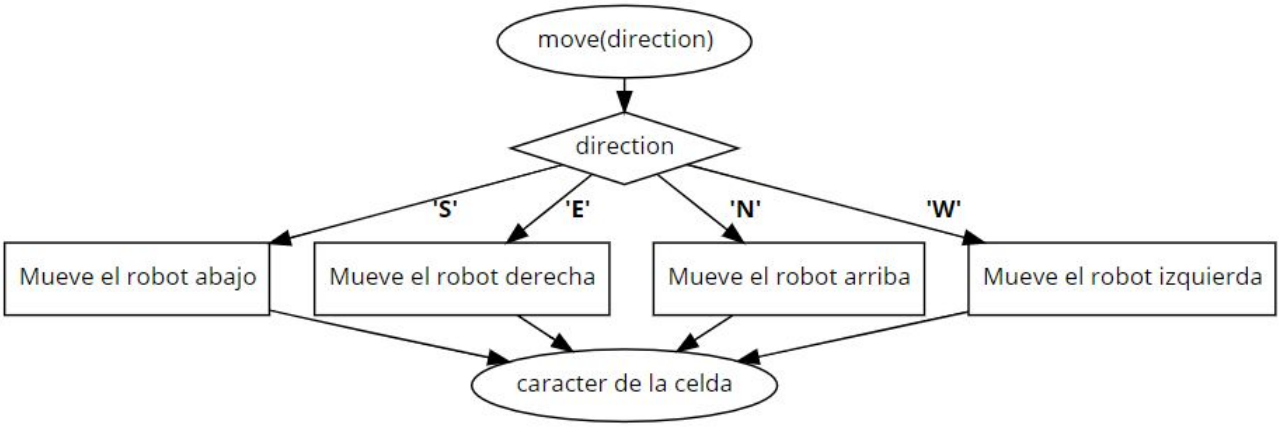
8.4. función move.

La función move actualiza la posición del Robot simulando que se mueve, la posición que adquiere esta determinada por la posición asignada por defineDirection. Devuelve el carácter de la nueva posición del Robot para saber si esta en un inversor, un Teleport o la meta.

```
switch (actualDirection){
    case 'S':
        this.rb.setRobotY(this.rb.getRobotY()+1);
        this.rb.setRobotX(this.rb.getRobotX());
        break;

    case 'E':
        this.rb.setRobotY(this.rb.getRobotY());
        this.rb.setRobotX(this.rb.getRobotX()+1);
        break;

    case 'N':
        this.rb.setRobotY(this.rb.getRobotY()-1);
        this.rb.setRobotX(this.rb.getRobotX());
        break;

    case 'W':
        this.rb.setRobotY(this.rb.getRobotY());
        this.rb.setRobotX(this.rb.getRobotX()-1);
        break;
}
return this.map.getFormedMap()[this.rb.getRobotY()][this.rb.getRobotX()].getCharacter();
```

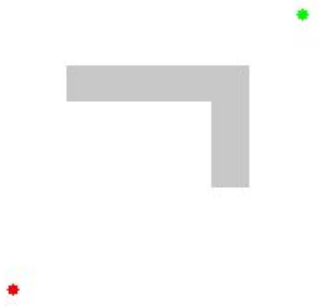
8.4.1. Diagrama de flujo de la función move.

Este diagrama de flujo representa a la función move.



9. Método bestRun (Opcional).

El método bestRun consiste en encontrar a cuántos pasos se encuentra el Robot respecto a la meta siguiendo el camino más corto. Para calcular a cuanto se encuentra el Robot de la meta. Para poder encontrar el camino más corto usamos el algoritmo de Dijkstra el cual consiste en desde el punto de salida simular la propagación de una onda.



Para simular esto en nuestro mapa simularemos una onda de números la cual se expande por el mapa asignando el valor a las casillas. Para empezar tenemos que asignar a la casilla en la que se encuentra el Robot 0 de distancia, por defecto todas las casillas se encuentran a -1 de distancia, esto quiere decir que no se les ha asignado una.

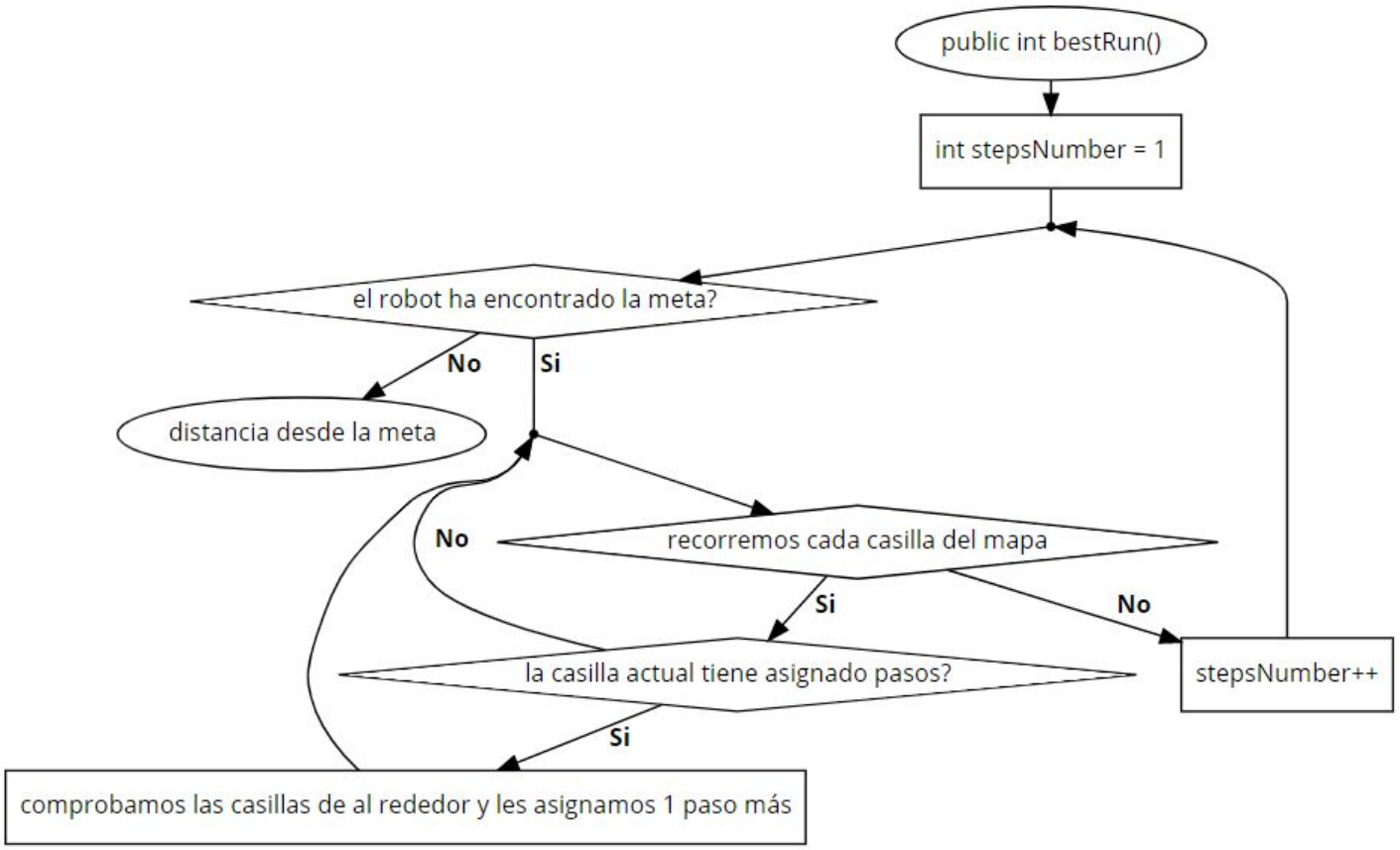
#	#	#	#	#	#	#	#	#	#
#	X	1							#
#	1								#
#									#
#				#	#	#		#	#
#				#				T	#
#				#		#	#	#	#
#			T	#				$	#
#	#	#	#	#	#	#	#	#	#

Una vez hemos puesto un 0 en la casilla del Robot recorremos el mapa buscando celdas la cuales tengan el número actual de pasos asignados, cuando se encuentra una celda con los pasos actuales llamamos a la función setDistances comprobando las casillas de alrededor de esta, solo se le asignará la distancia a las celdas que no sean paredes y que no tengan ya un número asignado. Esto se realiza hasta que se le asigna la distancia a la casilla de la meta. Por cada pasada se incrementan los pasos para simular los caminos.

#	#	#	#	#	#	#	#	#	#
#	X	1	2	3	4	5	6	7	#
#	1	2	3	4	5	6	7	8	#
#	2	3	4	5	6	7	8	9	#
#	3	4	5	#	#	#	9	#	#
#	4	5	6	#	11	10	9	T	#
#	5	6	7	#	12	#	#	#	#
#	6	7	T	#	13	14	15	$	#
#	#	#	#	#	#	#	#	#	#

9.1. Diagrama de flujo de la función bestRun.

Este diagrama de flujo representa a la función bestRun.



9.2. Función setDistances del método bestRun.

Esta función funciona complementariamente con el método bestRun, a esta se le enviará los pasos actuales que lleva el Robot y las coordenadas de la celda la cual se comprueba que no sea una pared y que no tenga ya un número asignado. Una vez se cumplen estas condiciones se comprueba si es un Teletransportador, en el caso de que lo sea el número de pasos se le asigna a la entrada del Teletransportador y a la salida de este.

```
private void setDistances(int steps, int y, int x){

    if ((this.map.getFormedMap()[y][x].getCharacter() != '#') && (this.map.getFormedMap()[y][x].getDistanceFromGoal() == 0)){

        if (this.map.getFormedMap()[y][x].getCharacter() == 'T'){
            Teleport tpOut = teleportMapList.get(cellMapList.indexOf(this.map.getFormedMap()[y][x]));
            this.map.getFormedMap()[tpOut.getNearTpY()][tpOut.getNearTpX()].setDistanceFromGoal(steps);
        }
        this.map.getFormedMap()[y][x].setDistanceFromGoal(steps);
    }
}
```

9.2.1. Diagrama de flujo de la función setDistances.

Este diagrama de flujo representa a la función setDistances.