Escola Tècnica Superior d'Enginyeria

Universitat Rovira i Virgili

[DEIM]

Metodologies de la Programació
Pràctica 3

Pràctica avaluable 3
Caminos en un laberinto
curs 2021-22

Estudiantes: Marc Fonseca (GEI)
Joel Lacambra (GEI)
Ismael Ruiz (GEI)
Professor/a: Simeó Reig Pelleja
Grupo: 18

Índice

1. Diseño de estrategias

El programa se divide básicamente en 4 clases:

- **“UsaLaberinto”,** donde se podrá leer el fichero (laberinto) en cuestión y además utilizar los dos algoritmos para intentar resolverlo.
- La clase **“Laberinto**” en la cual se diseñan los dos algoritmos especificados (voraz y la búsqueda exhaustiva).
- La clase **“Celda”** la cual contiene el tipo de operación a ejecutar junto a la puntuación actual.
- Por último, la clase **“CeldaVecina”** la cual actuará como una celda auxiliar para almacenar información.

Método voraz/avid:

Este algoritmo se especializa en encontrar el camino a la salida más óptima.

Su funcionalidad es la siguiente:

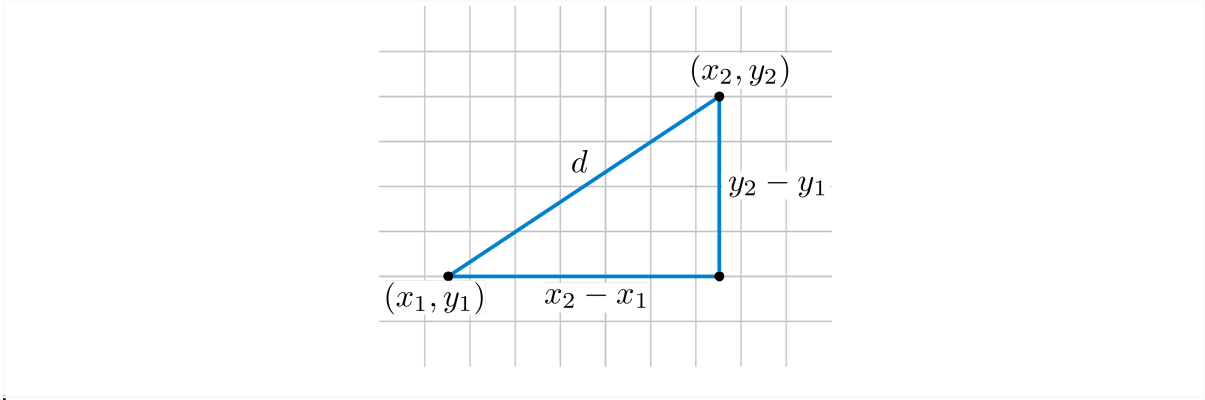
- Se comprueba cada celda vecina de la actual, se guardan en una variable auxiliar las 4 opciones.

- Se calcula la opción más viable para llegar a la salida con más de 0 puntos y con el camino más corto. Esto se consigue haciendo lo siguiente:
    - Comprobando el resultado de la operación (si es más grande que 0).
    - Y finalmente calculando la distancia de la celda actual hasta la final y comparándola con el mismo cálculo de la celda anterior.

- Ya que una vez avanzada a la siguiente posición hay que volver a mirar todos sus vecinos, uno de ellos siempre será la anterior posición de celda.
  Para quitarnos este problema, los atributos de celdas por las cuales hemos pasado (por donde se está creando el camino) se modificarán de tal manera:
    - Un “set” de su operación con el carácter “N”.
    - Un “set” de su valor con el número 0.

- Finalmente, una vez encontrado la mejor celda se guardará en un vector (el cual contendrá el camino hasta el momento) y repetirá este proceso hasta que lleguemos a la solución o no podamos, ya sea porque nos hemos quedado sin puntos o porque nos hallamos en una celda sin salida.

El espacio entre la celda actual y la celda resultante se calculará mediante la distancia euclidiana, que podremos obtener gracias a las coordenadas X e Y.



Distancia en un sistema de coordenadas cartesianas.

Este método se puede encontrar en la misma clase de **“Laberinto”**.

Método de búsqueda exhaustiva/backtracking:

Para este algoritmo se buscará no el camino más óptimo pero el primero que se encuentre.
En este caso, vamos a optar por no utilizar la distancia euclidiana ya que si no siempre irá por el mismo camino.

- Se harán llamadas recursivas para cada celda vecina, es decir, si primero comenzamos por el NORTE y vemos que puede seguir ese camino limitando la puntuación mayor que 0, seguirá por esa vía.

- Si seguimos por ese camino NORTE y no puede proceder, irá a la anterior posición y comprobará la siguiente dirección descrita en el código, que en este caso es ESTE. Hará sus llamadas recursivas hasta que ocurra una situación similar a la anteriormente mencionada.

- Una vez no puede seguir, es decir, que cada vez irá retrocediendo posiciones ya que sus llamadas recursivas no encuentran ningún camino a la salida, acabará en el punto de partida, el cual procederá a comenzar desde el ESTE (en este caso). Y así sucesivamente hasta que encuentre un camino.

2. Implementación en Java

2.1 Algoritmo voraz

```
while(!(salidaX == actualX && salidaY == actualY)){
	puntos = solucio.laberinto[actualX][actualY].operacion(puntos);

	CeldaVecina[] vecinos = new CeldaVecina[4];
	int indiceVecinos = 0;
	// Vecinos N, E, S, O

	// Vecino N
	if((actualX-1) >= 0 &&
	!solucio.laberinto[actualX-1][actualY].getOperacion().equals("N")) {
		vecinos[indiceVecinos]= celda visitada
		indiceVecinos++;
	}
	// Vecino E
	if((actualY+1) < columns &&
	!solucio.laberinto[actualX][actualY+1].getOperacion().equals("N")) {
		vecinos[indiceVecinos]= celda visitada
		indiceVecinos++;
	}
	// Vecino S
	if((actualX+1) < rows &&
	!solucio.laberinto[actualX+1][actualY].getOperacion().equals("N")) {
		vecinos[indiceVecinos]= celda visitada
		indiceVecinos++;
	}
	// Vecino O
	if((actualY-1) >= 0 &&
	!solucio.laberinto[actualX][actualY-1].getOperacion().equals("N")) {
		vecinos[indiceVecinos]= celda visitada
		indiceVecinos++;
	}
```

```
        // Elegir la mejor opcion
        double optim = 0;
        CeldaVecina mejorCelda = null;

        for(int i = 0; i < indiceVecinos;i++) {

              if(vecinos[i].getPosiblePuntuacion()>0) {

                    if (vecinos[i].getPosiblePuntuacion() /
                    vecinos[i].getDistancia() > optim) {

                          optim = vecinos[i].getPosiblePuntuacion() /
                          vecinos[i].getDistancia();
                          mejorCelda = vecinos[i];

                    }
              }
        }

        // Si no puede seguir, retorna ya el laberinto sin solucionar
        if (mejorCelda != null) {
              listaCeldas[indiceListaCeldas] = mejorCelda;
              solucio.laberinto[actualX][actualY].setOperacion("N");
              solucio.laberinto[actualX][actualY].setNumero(0);
              actualX = mejorCelda.getEjeX(); actualY = mejorCelda.getEjeY();
        }
        else {
              return vector2matriz (listaCeldas, indiceListaCeldas);
        }

        indiceListaCeldas++;
}
```

2.2 Búsqueda exhaustiva con ramificación y poda

```
boolean[][] lugaresVisitados = new boolean[rows][columns];

for (int i = 0; i < rows; i++) {
      for (int j = 0; j < columns; j++) {
            lugaresVisitados[i][j] = false;
      }
}

lugaresVisitados[actualX][actualY] = true;

// Vecinos N, E, S, O

// Vecino N
if((actualX-1) >= 0 &&
!laberinto[actualX-1][actualY].getOperacion().equals("N")) {
      encontrado = funcion recursiva
}
// Vecino E
if((actualY+1) < columns &&
!laberinto[actualX][actualY+1].getOperacion().equals("N")) {
      tamCamino = 1;
      encontrado = funcion recursiva
}
// Vecino S
if((actualX+1) < rows &&
!slaberinto[actualX+1][actualY].getOperacion().equals("N")) {
      tamCamino = 1;
      encontrado = funcion recursiva
}
// Vecino O
if((actualY-1) >= 0 &&
!laberinto[actualX][actualY-1].getOperacion().equals("N")) {
      tamCamino = 1;
      encontrado = funcion recursiva
}

return vector2matriz (listaCeldas, this.tamCamino);
```

```
boolean funcion recursiva () {
      boolean encontrado = false;

      // Si hemos llegado al final, se acaba
      if((this.salidaX == actualX) && (this.salidaY == actualY)) {
            opcionActual[this.tamCamino] = celda visitada
            return true;

      }
      else { // Anadir esta posicion al camino
            lugaresVisitados[actualX][actualY] = true;
            puntos = laberinto[actualX][actualY].operacion(puntos);
            opcionActual[this.tamCamino] = celda visitada
      }

      if(puntos > 0) {
            // Vecinos N, E, S, O

            // Vecino N
            if((actualX-1) >= 0 &&
            !laberinto[actualX-1][actualY].getOperacion().equals("N")) {
                  encontrado = funcion recursiva
            }
            // Vecino E
            if((actualY+1) < columns &&
            !laberinto[actualX][actualY+1].getOperacion().equals("N")) {
                  tamCamino = 1;
                  encontrado = funcion recursiva
            }
            // Vecino S
            if((actualX+1) < rows &&
            !slaberinto[actualX+1][actualY].getOperacion().equals("N")) {
                  tamCamino = 1;
                  encontrado = funcion recursiva
            }
            // Vecino O
            if((actualY-1) >= 0 &&
            !laberinto[actualX][actualY-1].getOperacion().equals("N")) {
                  tamCamino = 1;
                  encontrado = funcion recursiva
            }

      }

      if (!encontrado) {
            tamCamino--;
      }

      return encontrado;
}
```

3. Juego de pruebas

Si el tiempo obtenido es de color azul entonces, ha encontrado solución

Si el tiempo obtenido es de color rojo entonces, no ha encontrado solución

Laberinto	**Voraz**	**Búsqueda exhaustiva**
Laberinto PDF	1.2562 ms	0.1435 ms
Entrada: arriba izquierda Salida: abajo derecha	0.1546 ms	0.2089 ms
Entrada: abajo derecha Salida: arriba izquierda	0.876 ms	0.111099 ms
Entrada: abajo izquierda Salida: arriba derecha	0.0815 ms	0.1319 ms
Entrada: arriba derecha Salida: abajo izquierda	0.0848 ms	0.0994 ms
100x100 con solución	3.4951 ms	8.2166 ms

Como podemos observar el algoritmo voraz casi siempre tarda menos que la búsqueda exhaustiva, pero su principal inconveniente es que muy pocas veces encuentra solución. En cambio, la búsqueda exhaustiva tarda mucho más cuanto más grande sea el laberinto, pero SIEMPRE (si la hay) encuentra una solución aunque no sea la más óptima.