PROGRAMACIÓN Y ESTRUCTURAS DE DATOS AVANZADAS

Febrero 2018 (Segunda semana)

Normas de valoración del examen:

- La nota del examen representa el 80% de la valoración final de la asignatura (el 20% restante corresponde a las prácticas).
- Cada cuestión contestada correctamente vale 1 punto.
- Cada cuestión contestada incorrectamente baja la nota en 0.3 puntos.
- Debe obtenerse un mínimo de 3 puntos en las cuestiones para que el problema sea valorado (con 3 cuestiones correctas y alguna incorrecta el examen está suspenso).
- La nota total del examen debe ser al menos de 4.5 para aprobar.
- Las cuestiones se responden en una hoja de lectura óptica.

SOLUCIONES:

Test*:

Tipo A: 1A 2C 3D 4A 5 A/D 6B
Tipo B: 1D 2A 3B 4A/D 5C 6A

*** En la pregunta 5 (modelo A) y la pregunta 4 (modelo B) se han dado por válidas tanto la opción A como la opción D**

Problema (4 puntos).

Teseo se adentra en el laberinto en busca de un minotauro que no sabe dónde está. Se trata de implementar una función *ariadna* que le ayude a encontrar el minotauro y a salir después del laberinto. El laberinto debe representarse como una matriz de entrada a la función cuyas casillas contienen uno de los siguientes tres valores: 0 para “camino libre”, 1 para “pared” (no se puede ocupar) y 2 para “minotauro”. Teseo sale de la casilla (1,1) y debe encontrar la casilla ocupada por el minotauro y salir posteriormente del laberinto deshaciendo el camino recorrido. En cada punto, Teseo puede tomar la dirección Norte, Sur, Este u Oeste siempre que no haya una pared. La función *ariadna* debe devolver la secuencia de casillas que componen el camino de regreso desde la casilla ocupada por el minotauro hasta la casilla (1,1).

La resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado de entre los siguientes: vuelta atrás, divide y vencerás, programación dinámica y ramificación y poda. Escriba la estructura general de dicho esquema e indique cómo se aplica al problema (0,5 puntos).
2. Descripción de las estructuras de datos necesarias (0.5 puntos solo si el punto 1 es correcto).
3. Algoritmo completo a partir del refinamiento del esquema general (2,5 puntos solo si el punto 1 es correcto).
4. Estudio del coste del algoritmo desarrollado (0.5 puntos solo si el punto 1 es correcto).

Solución:

1. **Como no se indica nada al respecto de la distancia entre casillas adyacentes, y ya que se sugiere utilizar únicamente una matriz, es lícito suponer que la distancia entre casillas adyacentes es siempre la misma (1, sin pérdida de generalidad). Por otra parte, no se exige hallar el camino más corto entre la entrada y el minotauro.**

   Tras estas consideraciones previas ya es posible elegir el esquema algorítmico más adecuado. El tablero puede verse como un grafo en el que los nodos son las casillas y en el que como máximo surgen cuatro aristas (N, S, E, O). Todas las aristas tienen el mismo valor asociado (por ejemplo, 1).

   Como no es necesario encontrar el camino más corto, sino encontrar uno cualquiera, una búsqueda en profundidad resulta más adecuada que una búsqueda en anchura.

   Es posible que una casilla no tenga salida por lo que es necesario habilitar un mecanismo de retroceso.

   Por último, es necesario que no se exploren por segunda vez casillas ya exploradas anteriormente.

   Por estos motivos, se ha elegido el esquema de vuelta atrás. El esquema general viene descrito en la página 161 del libro de texto de la asignatura.

2. **Para almacenar las casillas bastará un registro de dos enteros x e y. Vamos a utilizar una lista de casillas para almacenar la solución y otra para las compleciones. Para llevar control de los nodos visitados bastará una matriz de igual tamaño que el laberinto pero de valores booleanos.**

   Será necesario implementar las funciones de lista:

   - **crear_lista**
   - **vacia**
   - **añadir**
   - **primero**

3. **Algoritmo completo: suponemos laberinto inicializado con la configuración del laberinto y visitados inicializado con todas las posiciones a falso.**

```
tipoCasilla = registro
                x,y: entero;
              fregistro

tipoLista

fun vuelta-atrás (laberinto: vector [1..LARGO, 1..ANCHO] de entero;
                  casilla: tipoCasilla
```

```
                    visitados: vector [1..LARGO, 1..ANCHO] de booleano;
                ) dev (es_solución: booleano; solución: tipoLista)

visitados [casilla.x, casilla.y] ← verdadero;
si laberinto[casilla.x, casilla.y] == 2 entonces
        solución ← crear_lista();
        solución ← añadir (solución, casilla);
        devolver (verdadero, solución);
si no
        hijos ← crear_lista();
        hijos ← compleciones (laberinto, casilla)
        es_solución ← falso;
                mientras ¬ es_solución ∧ ¬ vacia (hijos)
                hijo ← primero (hijos)
                si ¬ visitados [hijo.x, hijo.y] entonces
                        (es_solución, solución) ← vuelta-atrás (laberinto,hijo,visitados);
                fsi
        fmientras
        si es_solución entonces
                solución ← añadir (solución, ensayo);
        fsi
        devolver (es_solución, solución);
fsi
ffun
```

En el caso de encontrar al minotauro se detiene la exploración en profundidad y al deshacer las llamadas recursivas se van añadiendo a la solución las casillas que se han recorrido. Como se añaden al final de la lista, la primera será la del minotauro y la última la casilla (1,1), tal como pedía el enunciado.

La función *compleciones* comprobará que la casilla no es una pared y que no está fuera del laberinto

```
fun compleciones (laberinto: vector [1..LARGO, 1..ANCHO] de entero;
                        casilla: tipoCasilla) dev tipoLista
        hijos ← crear_lista();
        si casilla.x+1 <= LARGO entonces
                si laberinto[casilla.x+1,casilla.y] <> 1 entonces
                        casilla_aux.x=casilla.x+1;
                        casilla_aux.y=casilla.y;
                        hijos ← añadir (solución, casilla_aux);
                fsi
        fsi
        si casilla.x-1 >= 1 entonces
                si laberinto[casilla.x-1,casilla.y] <> 1 entonces
                        casilla_aux.x=casilla.x-1;
                        casilla_aux.y=casilla.y;
                        hijos ← añadir (solución, casilla_aux);
                fsi
        fsi
        si casilla.y+1 <= ANCHO entonces
```

```
            si laberinto[casilla.x,casilla.y+1] <> 1 entonces
                casilla_aux.x=casilla.x;
                casilla_aux.y=casilla.y+1;
                hijos ← añadir (solución, casilla_aux);
            fsi
        fsi
        si casilla.y-1 >= 1 entonces
            si laberinto[casilla.x,casilla.y-1] <> 1 entonces
                casilla_aux.x=casilla.x;
                casilla_aux.y=casilla.y-1;
                hijos ← añadir (solución, casilla_aux);
            fsi
        fsi
    ffun
```

4. **Estudio del coste: El espacio de búsqueda del problema es un árbol en el que cada nodo da lugar como máximo a tres ramas correspondientes a las cuatro direcciones de búsqueda, menos la casilla de la que procede el recorrido. El número de niveles del árbol es el número de casillas del tablero, ANCHOxLARGO. Luego una cota superior es $3^{ANCHOxLARGO}$.**