Práctica 3: Diseño de algoritmos de búsqueda con retroceso

Programación-II

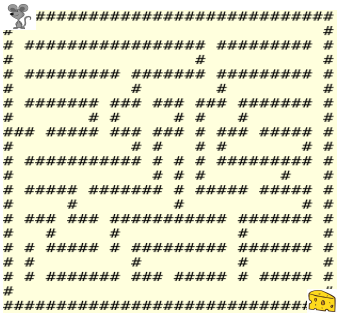
Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivo de la práctica

El objetivo principal de la práctica es aprender a programar un algoritmo de búsqueda con retroceso para encontrar la salida de un laberinto.

2. Problema: Atravesar un laberinto

El problema del laberinto consiste en encontrar un camino desde la posición inicial (esquina superior izquierda) hasta la posición final (esquina inferior derecha), mientras se recorre un entorno con obstáculos (o muros). Es un problema recurrente en robótica, en la que se buscan soluciones (a menudo óptimas) para el empleo de vehículos autónomos en diferentes tareas (por ejemplo, en la industria para transportar productos a través de fábricas u otros lugares).



En esta práctica se desarrollará un programa para ayudar al ratón a encontrar un camino hacia el queso.

Podemos representar el laberinto como una matriz m de dimensión $alto \times ancho$ (donde $alto, ancho \leq$ `MAX_DIM`), en la que cada posición m_{ij} está marcada como `LIBRE` (espacio en blanco `' '` en la figura) o `MURO` (almohadilla `'#'`). Las posiciones de frontera del laberinto delimitan el mismo, así que están marcadas todas como muros. Es posible pasar de una posición a otra moviéndose solamente en vertical u horizontal. La entrada al laberinto corresponde a la posición $(1, 1)$ y la salida corresponde a la posición $(alto - 2, ancho - 2)$.

Se requiere buscar una posible solución (no necesariamente óptima en la longitud del camino).

2.1. Diseño de una solución

Se va a realizar un diseño modular para solucionar el problema planteado. En particular, descomponemos el problema en los siguientes sub-problemas:

- Encontrar un camino en el laberinto.
- Generar el laberinto.
- Visualizar el camino recorrido.
- Gestionar la interacción con el usuario.

Se diseñará un módulo (`laberinto.hpp`, `laberinto.cpp`) que implementará soluciones a los tres primeros sub-problemas. El módulo principal (`mainLab.cpp`) implementará una solución para el último sub-problema.

3. Encontrar un camino en el laberinto

La estructura de datos para almacenar un laberinto se proporciona en el fichero de interfaz `laberinto.hpp` y se muestra en el Listado 1.

```
// Dimensión máxima de los laberintos
const int MAX_DIM = 120;

// Posibles valores de las posiciones del laberinto
const char LIBRE = '␣';  // (espacio en blanco), posición libre
const char MURO = '#';   // (almohadilla), posición con muro
const char CAMINO = '.'; // (punto), camino
const char IMPOSIBLE = 'I'; // posición ya visitada que no llevó a solución

struct Laberinto{
    // Altura (número de filas) del laberinto
    int alto;
    // Anchura (número de columnas) del laberinto
    int ancho;
    // Matriz del laberinto
    // Cada posición puede tomar el valor LIBRE, MURO, CAMINO o IMPOSIBLE
    char mapa[MAX_DIM][MAX_DIM];
};
```

Listado 1: Estructura de datos `Laberinto`.

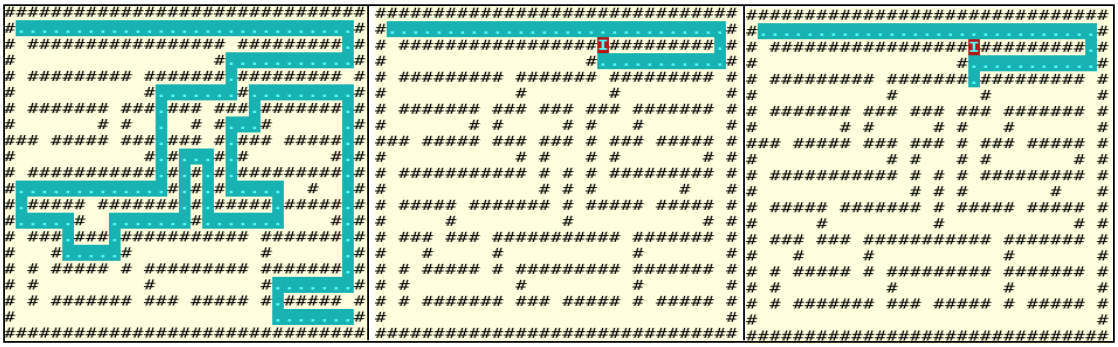
Para resolver el problema planteado, diseñaremos un algoritmo de búsqueda con retroceso (en inglés, *backtracking*) de forma que se marcará en la misma matriz del laberinto un camino solución, si este camino existe. Se debe tener en cuenta que es posible pasar de una posición a otra moviéndose solamente en vertical u horizontal.

Si por un camino recorrido se llega a una posición desde la que es imposible encontrar una solución, hay que volver atrás (*backtracking*) y buscar otro camino. Además hay que marcar las posiciones por donde ya se ha pasado para evitar meterse varias veces en el mismo callejón sin salida, dar vuelta alrededor de obstáculos, etc.

El esquema de la técnica para encontrar una solución es:

1. Si la posición actual es la salida entonces se ha encontrado un camino (devolver éxito).
2. Si la posición actual es `MURO` o ya se visitó entonces no hay que seguir explorándola (devolver fracaso).
3. Si no, marcar la posición como `CAMINO` y:
   a) Si hay camino desde alguna de las posiciones vecinas hasta la salida entonces se ha encontrado un camino (devolver éxito);
   b) si no, no hay camino desde la posición actual (devolver fracaso).

En la Figura 1(a) se marcan las posiciones del camino con un punto (`'.'`). En la Figuras 1(b,c), las posiciones ya visitadas que no llevaron a solución están marcadas con la letra `'I'`.



(a) Camino encontrado (b) Caso retroceso (c) Búsqueda otro camino

Figura 1: Búsqueda de un camino.

Se dice que un laberinto es *correcto* cuando:

- el número de filas y el número de columnas del laberinto es mayor o igual a 3;
- cada posición del laberinto tiene el valor `LIBRE`, `MURO`, `CAMINO` o `IMPOSIBLE`;
- todas las posiciones de la primera y última fila tienen el valor `MURO`; y

- todas las posiciones de la primera y última columna tienen el valor `MURO`.

Del mismo modo, se dice que el laberinto está *vacío* cuando todas las posiciones del laberinto contienen o bien `LIBRE` o bien `MURO`.

Tarea 1: Se pide implementar el algoritmo de búsqueda con una función recursiva `buscarCamino` cuya cabecera se muestra a continuación.

```
// Pre:  "lab" es un laberinto correcto y vacío
// Post: "encontrado" si, y solo si, se cumplen las condiciones siguientes:
//        * en "lab" se ha marcado con CAMINO las casillas de un camino
//          que une las casillas (1,1) y (lab.alto-2,lab.ancho-2)
//        * las casillas visitadas que no llevaban a la salida quedan
//          marcadas como IMPOSIBLE
//        * el resto de casillas no se han modificado
void buscarCamino(Laberinto& lab, bool& encontrado);
```

4. Generar el laberinto

El dato de entrada del algoritmo de búsqueda es el laberinto (estructura con campos de dimensiones y mapa, localizada en el Listado 1). Para verificar la corrección de la solución implementada es posible utilizar un conjunto de pruebas o implementar un algoritmo que genere de forma aleatoria un laberinto.

4.1. Conjunto de pruebas

En Moodle se proporcionan ficheros de texto (incluidos en la carpeta `datos` del código de apoyo) que almacenan laberintos de diferente tamaño.

Tarea 2: Se pide implementar la función de carga de un laberinto desde un fichero de texto, `cargarLaberinto`, cuya especificación se muestra a continuación.

```
// Pre:  "nombFichero" es el nombre de un fichero que almacena un
//       laberinto correcto
// Post: "lab" contiene el laberinto del fichero, almacenado de acuerdo
//       a la especificación dada para el tipo de dato
void cargarLaberinto(const string nombFichero, Laberinto& lab);
```

4.2. Generación aleatoria de laberintos

Se utiliza la función de librería `rand` (visto en la Práctica 1) para generar de forma aleatoria un laberinto. Los parámetros necesarios para generar un laberinto son los siguientes:

- *alto*: altura (en caracteres), es decir número de filas del mapa m;

- *ancho*: anchura (en caracteres), es decir número de columna del mapa m;
- *densidad*: es un valor real $d \in [0, 1]$ que determina la densidad del laberinto. Para cada posición (i, j) del mapa se genera un número aleatorio $x \in [0, 1]$. El valor de la posición (i, j) será `MURO` si $x \leq d$, si no su valor será `LIBRE`.

Tarea 3: Se pide implementar de forma recursiva la función `generarLaberinto` cuya cabecera se muestra a continuación. Los parámetros *alto* y *ancho* son ímplicitos (campos del tipo `Laberinto`).

```
// Pre: 3 ≤ lab.alto, lab.ancho ≤ MAX_DIM ∧
//      0 ≤ fila ≤ lab.alto − 1 ∧ 0 ≤ col ≤ lab.ancho − 1 ∧ 0 ≤ densidad ≤ 1
// Post: "lab" queda inicializado con un laberinto aleatorio
//       correcto y vacío
void generarLaberinto(Laberinto& lab, const double densidad,
                      const int fila, const int col);
```

5. Visualizar el camino recorrido

Se proporciona la función `mostrarLaberinto` en el fichero `laberinto.cpp`, implementada de forma iterativa, como se muestra a continuación.

```
// Pre: "lab" es un laberinto correcto
// Post: Se ha mostrado el laberinto por la salida estándar
// Coms: Versión iterativa
void mostrarLaberinto(const Laberinto& lab) {
    for (int i = 0; i < lab.alto; i++) {
        for (int j = 0; j < lab.ancho; j++) {
            cout << lab.mapa[i][j];
        }
        // Al acabar una fila, se cambia de línea
        cout << endl;
    }
}
```

Tarea 4: Se pide implementar de forma recursiva la función `mostrarLaberinto`.

6. Gestionar la interacción con el usuario

La gestión de la interacción con el usuario se delega al programa principal `mainLab.cpp`. En particular, hay que considerar únicamente dos posibles formas de utilizar el programa.

Primera: Lectura de un fichero de texto que especifica un laberinto. En este caso, el comando para ejecutar el programa será del tipo:

```
$ ./mainLab rutaFichero
```

donde `rutaFichero` es la ruta del fichero (incluido su nombre).

Segunda: Generación de un laberinto de forma aleatoria. En este caso, el comando para ejecutar el programa será del tipo:

```
$ ./mainLab alto ancho densidad
```

Tarea 5: Se pide implementar el programa principal `mainLab.cpp` que verifica los datos proporcionados por el usuario (número de argumentos correcto, valores correctos de altura, anchura y densidad del laberinto a generar de forma aleatoria).

En caso de uso incorrecto, el programa muestra por la salida estándar (pantalla) un aviso de error y termina con un estado de error.

En caso de uso correcto, el programa:

1. lee el fichero o genera el laberinto (según el tipo de comando proporcionado por el usuario);
2. muestra el laberinto leído o generado;
3. busca un camino en el laberinto; y
4. muestra el laberinto con el camino recorrido.

Si no se encuentra un camino hasta la salida, muestra por la salida estándar un aviso y termina el programa.

Tarea 6: Escribir un fichero `Makefile` para generar el programa ejecutable `mainLab`.

7. Código desarrollado en las tres primeras prácticas

Como resultado de las tres primeras prácticas, cada alumno dispondrá en su cuenta de un directorio (carpeta) denominado **`programacion2`** dentro del cual se podrán encontrar los directorios (carpetas) y ficheros que se detallan a continuación.

1. Carpeta **`programacion2/funciones`** con los siguientes ficheros:
   - Ficheros de interfaz y de implementación, **`pilaEnt.hpp`** y **`pilaEnt.cpp`**
2. Carpeta **`programacion2/practica1`**, con los siguientes ficheros fuentes:
   - **`tiempoReaccion.cpp`**, **`generarTabla01.cpp`**, **`generarTabla02.cpp`** y **`medirCoste.cpp`**.
3. Carpeta **`programacion2/practica2`** con los siguientes ficheros:

- Ficheros de interfaz y de implementación, **`calculos.hpp`** y **`calculos.cpp`**.
- Ficheros de interfaz y de implementación, **`funcionesPilaEnt.hpp`** y **`funcionesPilaEnt.cpp`**.
- Ficheros con los programas de prueba (**`pruebas01.cpp`**, **`pruebas02.cpp`**, etc.) que se hayan puesto a punto para realizar pruebas de los desarrollos anteriores.
- Ficheros **`Make_pruebas01`** y **`Make_pruebas02`** para compilar los programas de prueba.

4. Carpeta **`programacion2/practica3`** con los siguientes ficheros:
   - Ficheros de interfaz y de implementación, **`laberinto.hpp`** y **`laberinto.cpp`**.
   - Fichero de implementación **`mainLab.cpp`**.
   - Fichero **Makefile** para obtener el ejecutable **`mainLab`**.

El código de apoyo que se proporciona en Moodle incluye los siguientes ficheros: `laberinto.hpp`, `laberinto.cpp` (incompleto) y los laberintos de prueba en la carpeta `datos`.

La duración de esta práctica es de dos sesiones. Por tanto, estos ficheros deberán estar listos antes de la sexta sesión de prácticas.