Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Departamento de Electrónica y Automática

Programación de Sistemas

Práctica 1: Diseño de un experimento

Implementación y comparación de un generador dinámico de laberintos en C++ mediante un algoritmo *recursivo* e *iterativo*

Celia Ramos Ramírez (18295)
Gonzalo Quirós Torres (17353)
Josep María Barberá Civera (17048)

3° GITI

29 de abril de 2021

Resumen

El objetivo del presente trabajo es elaborar y comparar dos algoritmos, uno iterativo y uno recursivo, que realicen una misma acción. Posteriormente se valorará el desempeño de cada uno por separado y se hará una comparación.

Para que la comparación se haga homogéneamente se ha puesto especial cuidado en que los dos algoritmos, tanto el recursivo como el iterativo, sigan procesos análogos. Para lograrlo el esquema funcional de los algoritmos es prácticamente el mismo, los pasos que sigue el proceso son idénticos salvo por las diferencias inherentes a la naturaleza de los algoritmos y el número de llamadas a ficheros externos desde cada fichero es igual en ambos casos.

La elaboración de los dos algoritmos no ha sido simultánea: en primer lugar, se elaboró el algoritmo recursivo pues la naturaleza del problema era abordable de forma directa desde la perspectiva de la recursividad. Después se elaboró el algoritmo iterativo, que planteó gran número de dificultades que poco a poco lograron ser salvadas. En último término se adecuó el algoritmo iterativo para que su estructura y forma coincidiera lo más exactamente con el algoritmo recursivo

Índice

1. Ideación **1**
- 1.1. Primeras propuestas . . . 1
- 1.2. Programación del algoritmo . . . 2
  - 1.2.1. Recursivo . . . 2
  - 1.2.2. Iterativo . . . 3
  - 1.2.3. Comentarios al código . . . 4

2. Experimentación **6**
- 2.1. Toma de medidas . . . 6

3. Análisis **7**

4. Conclusiones **8**

Introducción

Los algoritmos recursivos son una potente herramienta para la algoritmia en la programación. En esta memoria se pretende comparar un algoritmo recursivo con otro iterativo. Dicha comparación se realiza mediante el diseño de un experimento el cual pone en evidencia las ventajas y problemáticas que presenta la programación recursiva frente a la iterativa.

La memoria se estructura en cuatro partes que corresponden a las fases seguidas en el experimento. La primera fase es la de *Ideación*. Es la parte más importante, pues ella condiciona todas las fases posteriores. En ella se plantean las ideas propuestas y se discute la que finalmente ha sido elegida. Además, incluye la implementación sucesiva de los dos algoritmos mediante programación en C++. La segunda fase es la de *Experimentación*. Es aquí donde se exponen y discuten los criterios de comparación utilizados y cómo se han implementado dichas métricas en el código. Por otro lado, se explica la metodología empleada para las pruebas o toma de mediadas. La tercera fase es la de *Análisis*. Esta parte resulta de vital importancia y es en ella en la que se pone de manifiesto de forma gráfica los resultado obtenidos en las fases precedentes. Para la realización del análisis heurístico, se ha procedido con ayuda de la herramienta RStudio, a un análisis gráfico. En la cuarta y última fase de *Conclusiones* se recapitula y enuncian las principales conclusiones obtenidas del experimento realizado.

FASE 1

Ideación

1.1 Primeras propuestas

Inicialmente, se pensó en realizar un experimento con esencia matemática, esto es, abordar la resolución de integrales sencillas mediante sumas de Riemann. La inteción era ir variando las particiones del dominio de integración mientras se comparaba el desempeño de los dos algoritmos. Aunque parecía un experimento adecuado, planteaba también varias problemáticas:

- Las funciones han de ser predeterminadas por el programador, perdiendo así la interacción con el usuario.
- De lo anterior se desprende que la cantidad de funciones que se pueden muestrear es limitada, lo que puede generar un sesgo, pues las funciones elegidas pueden favorecer el desempeño de un algoritmo frente al otro.
- Otra forma de abordar el problema sería mediante aproximaciones polinómicas a las integrales. Esto supone que las problemáticas anteriores siguen estando presentes, pero, además, las fórmulas de aproximación polinómica son sencillas (hasta cierto grado) y por ende los tiempos de ejecución no serían suficientemente largos para que la comparación resultare significativa.

Como segunda posibilidad se valoró la implementación de *juegos de estrategia* como el tres en raya, cuatro en raya, etc. Pero se dedujo que en un tablero acotado como en el tres en raya las iteraciones iban a ser siempre las mismas y al mismo tiempo no tenía sentido diseñar un tablero más grande pues el problema se iba a concentrar en un espacio reducido.

Finalmente se encontró otros algoritmos que diseñaban laberintos a partir de una malla rellena por carácteres. De entre los distintos métodos que abordaban el problema, se eligió aquel que consistía en visitar una por una las casillas, comprobando si eran válidas e ir abriéndose camino hasta rellenar toda la malla con pasillos. Esta metodología es conocida como *recursive backtracker*[2]. El cual es una versión aleatoriazada del algoritmo *depht first-first search*. En la siguiente ilustración (ver FIG. 1) puede verse un ejemplo del algoritmo implementado.

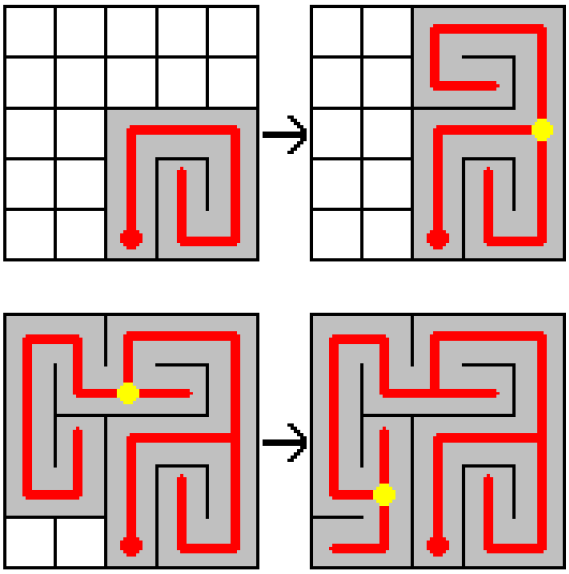

Figura 1: *Ejemplo de un algoritmo recursive backtracker [1]*

1.2 Programación del algoritmo

1.2.1. Recursivo

Desde el punto de vista recursivo, esta forma de contruir un laberinto no presenta grandes problemas. El algoritmo crea una rama del laberinto hasta que le es imposible seguir porque, por ejemplo, el extremo ha quedado "encerrado" entre el muro exterior (el cual marca el límite y evita que el programa se salga de la memoria reservada para el laberinto evitando que el S.O. interrumpa la ejecución) y la propia rama. La naturaleza de la recursividad hace que, una vez llegado a ese punto, al no haber posibilidad de movimiento, resuelva por fin esa llamada a la función recursiva y devuelva el resultado a las casillas anteriores que aún estaban pendientes de resolución. Esto continúa hasta que encuentra una casilla desde la que puede continuar camino, entonces abrirá desde ahí una nueva subrama del laberinto que continuará hasta que quede "encerrada" de nuevo y empiece a resolver hacia las casillas anteriores. Puede verse un ejemplo de dicha iteración en la figura 1. Finalmente, cuando ya no hay más casillas posibles por las que continuar, se volverá hacia

atrás todas las casillas hasta llegar a la primera, en nuestro casi siempre la posición $(1, 1)$. Este proceso asegura que el laberinto discurra por todas las casillas que sean posibles y que nunca quede encajonado"(como ya hemos dicho, la propia naturaleza de la recursividad lo garantiza).

1.2.2. Iterativo

El abordaje iterativo de este problema no es tan sencillo. Esto es debido, al hecho de que crear una rama principal de forma iterativa resulta más complejo que crearla recursivamente. Estableciendo unas condiciones lógicas de control para que la rama no se salga de los bordes y se evite a si misma para no romper el laberinto es suficiente. Pero tarde o temprano, la rama choca consigo misma y no tiene a donde ir. Solucionar este único punto es lo que más esfuerzo supone en la implementación del caso iterativo. La solución más inmedita resulta de crear un nuevo objeto o variable que almacene los lugares por los que la rama (o subrama) ya ha pasado. En esta memoria ha sido la última opción valorada, dado que su implementación significa alejarse del esquema marcado por el algoritmo recursivo. Para ello el equipo se dividió en dos:

- Una persona intentaría resolver el problema mediante la declaración de una variable dinámica local bidimensional. Es decir, una matriz con tantas filas como elementos tuviera la matriz sobre la que se hacía el laberinto y con tantas columnas como las dimensiones del laberinto (2 por ser bidimensional).

- Otra persona intentaría la implementación de una pila dinámica tipo LIFO, que en un principio solo almacenaría los índices de las casillas por las que la rama o subrama había pasado en orden (más tarde se utilizó la pila también para almacenar más datos).

- La tercera persona del equipo ejercería una labor de control y apoyo. Trabajando con las otras dos personas en limpieza del código e implementación de funciones auxiliares. Asimismo, controlaba el desarrollo de las dos vías con el fin de descartar la que menos conveniera para centrar todos los esfuerzos en la más prometedora.

Finalmente, la opción elegida ha sido la pila.

En la primera versión solo almacenaba la situación de las casillas ya ocupadas por la rama. Posteriormente se incluyó una funcionalidad extra en el código: cuando el programa se viera obligado a retroceder por la rama que acababa de construir, comprobaría cada dirección de la casilla en la que estuviera para decidir si era posible continuar en alguna otra dirección, pero de forma que se incuría en algunas ineficiencias, pues se comprobaban direcciones que ya se habían comprobado antes, cuando la rama había pasado por allí por primera vez (o por segunda, o por tercera). Se almacenó entonces en la pila el vector de direcciones aleatorizado, así como por qué iteración iba la comprobación del vector. De esta forma es posible finalizar el laberinto asegurando que todas las direcciones han sido comprobadas y todas en un orden aleatorio.

Se incluye a continuación el diagrama de flujo utilizado para la programación de la función Visit() en el caso iterativo:

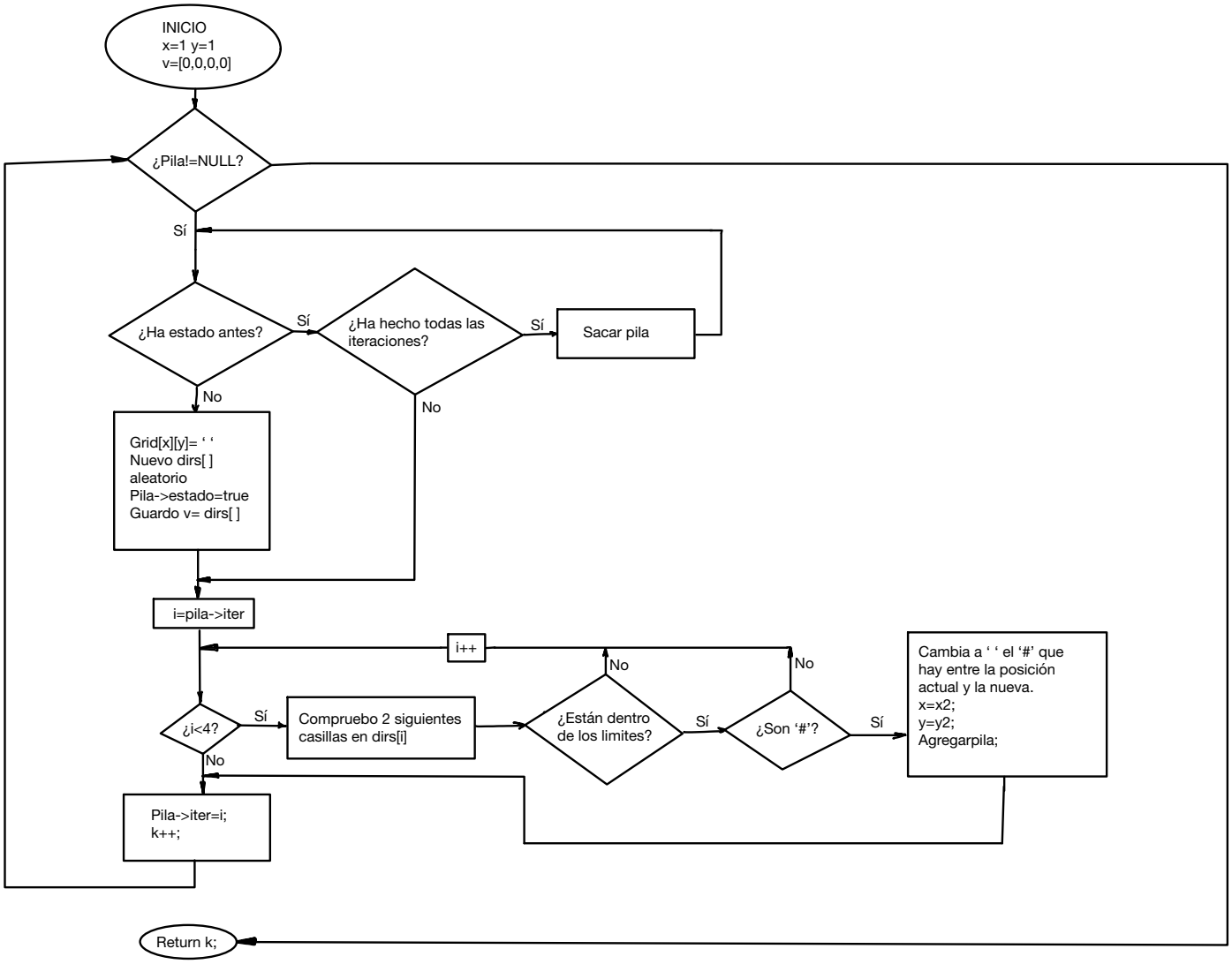

Figura 2: *Diagrama de flujo para la función Visit() en el algoritmo iterativo*

1.2.3. Comentarios al código

Una vez implementados ambos algoritmos, surgen algunos detalles a ajustar.

Lo primero a corregir fue un "doble borde" que aparecía en las paredes derecha e inferior del laberinto. Dichas paredes extras eran debidas a que en la iteración aleatoria por el laberinto la última columna y fila no "caían dentro" de los movimientos del programa. Esto a su vez, era debido a que se comenzaba en la casilla $(1, 1)$ y que las dimensiones fueran pares o impares. Si por ejemplo, las dimensiones fueran $6x6$, al comenzar en la posición $(1, 1)$, y suponiendo que los dos primero movimientos son hacia la derecha, la siguiente casilla a la que moverse sería la $(1, 3)$ que sería válida, pero en el siguiente movimiento la nueva casilla estudiada sería la $(1, 5)$ que forma parte del borde exterior y no forma parte de los límites o casillas válidas. Por lo tanto, la solución elegida ha sido forzar a que las dimensiones sean siempre impares. Esto no supone demasiado problema, pues el usuario puede pedir dimensiones pares y es el programa quien modifica estas para que sean impares. En concreto las siguientes lineas de código:

```
  // Se ajustan las dimensiones para que sean impares siempre.
    (filas%2) ? filas : filas+=1;
    (columnas%2) ? columnas : columnas+=1;
```

Ajuste de dimensiones dentro de la función Pedir()

de forma que es un cambio transparente para el usuario. Por otro lado, el cambio no es perceptible para dimensiones mayores de $10x10$, lo que es la mayoría de los casos.

Las líneas de código anterior fueron planteadas en su inicio como la MACRO siguiente:

```
  #define Arreglar_2D(x,y) (((x%2) ? x : x+=1),((y%2) ? y : y+=1))
```

Ajuste de dimensiones dentro de la función Pedir()

pero finalmente se decidió no utilizar ya que no respondia totalmente a la función de lo que debe ser una MACRO.

Por motivos estéticos, también se buscó algún carácter en UNICODE para que las casillas no vaciadas del laberintos se asemejaran más a paredes y poder visualizar el dibujo más fácilmente. Las siguientes líneas de código muestran la posible variante para la función PringGrid():

```
for (int i=0; i<filas; i++) {
    for (int j=0; j<columnas; j++){
      if(grid[i][j] == '#')
        cout << "\u2B1C";
      else cout << "  ";  // Notese el doble espacio intencionado
    }
    cout << endl;
}
```

Ajuste de dimensiones dentro de la función Pedir()

En las figuras 3 y 4 puede verse la comparación entre la salida por consola mediante el símbolo UNICODE utilizado en el código anterior y el símbolo ASCII #.

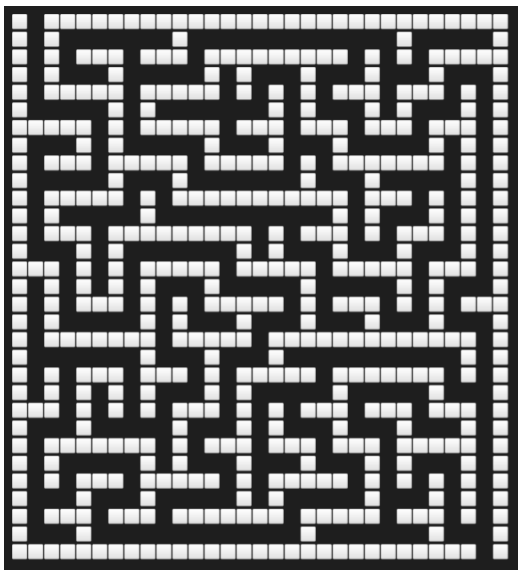

Figura 3: *Salida por consola del programa con carácteres* UNICODE

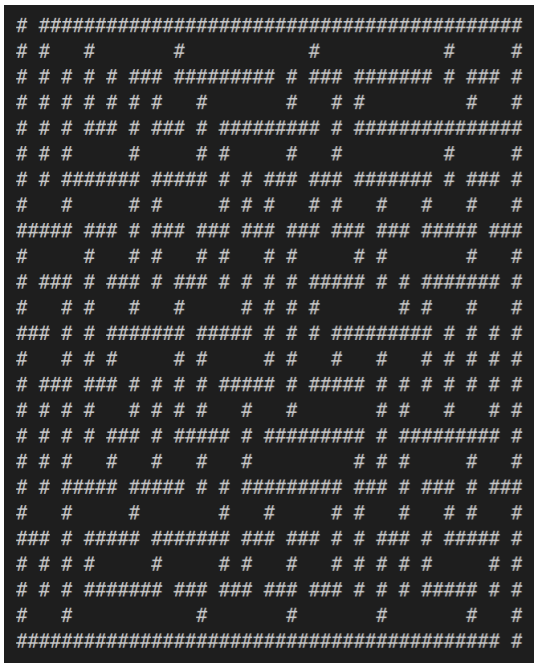

Figura 4: *Salida por consola del programa con carácteres* ASCII

Por motivos de portabilida se ha preferido no incluirlo directamente en el código final. Pero puede ser implementado fácilmente gracias el código anterior.

FASE 2

Experimentación

2.1 Toma de medidas

Para poder estudiar y comparar la ejecución de ambos códigos y algoritmos, ha sido necesaria una fase de toma de datos. Los datos buscados corresponden a las diversas métricas empleadas, estas son: número de iteraciones, velocidad de ejecución y uso de memoria. Para la experimentación, esto es ejecutar el código en diferentes condiciones, se ha elaborado un fichero *shell script* que permite ejecutar cada programa repetidas veces y genera un archivo de salida en el que vuelca los datos de cada ejecución. Más tarde este fichero es convertido a una tabla de datos con formato *.cdv* y dichos datos se han analizado mediante RSTUDIO. Más concretamente, el shell script empieza ejecutando el programa con un valor inicial de 20 tanto para filas y columnas. Y va sumando 50 unidades a filas y columnas con cada ejecución hasta llegar a 10000, realizando cada ejecución 5 veces para poder conseguir una media de los tiempos de ejecución. En cada ejecución, escribe en el fichero *output.csv* (o *output.txt* como se prefiera): un número del 1 al 5 correspondiente con la ejecución en curso, el tamaño de las filas y columnas, el área del laberinto (filas*columnas), el tiempo empleado en la ejecución de la función Visit() y el número de iteraciones realizadas por dicha función.

Se ha observado que en el caso del recursivo el número de iteraciones es siempre el mismo, ya que al ser constantes las dimensiones del laberinto la función Visit() es

llamada el mismo número de veces en cada ejecución con los mismos datos. Pero las 5 ejecuciones nos sirven para obtener la media del tiempo.

En el caso del iterativo, tanto el tiempo como el número de iteraciones es ligeramente diferente en cada ejecución dada la aleatoriedad del recorrido de la pila y, por tanto, cada vez se 'atasca' en un sitio distinto y recorre un camino distinto.

revisar!!!!!

A la hora de realizar pruebas, disponemos de 3 sistemas operativos diferentes: dos UNIX, ambos con diferentes procesadores, y un Windows. Y nuestra intención es extraer datos de la ejecución de los programas en cada uno de los tres. Además, nos aporta una manera de comprobar experimentalmente la portabilidad de nuestro proyecto.

FASE 3

Análisis

FASE 4

Conclusiones

Referencias

[1] M. Foltin. *Automated maze generation and human interaction.* PhD thesis, Masarykova univerzita, Fakulta informatiky, 2011.

[2] Wikipedia. Maze generation algorithm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Maze%20generation%20algorithm&oldid=1008588880, 2021. [Online; accessed 29-April-2021].