Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Programación de Sistemas

Departamento de Electrónica y Automática

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

30 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

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.

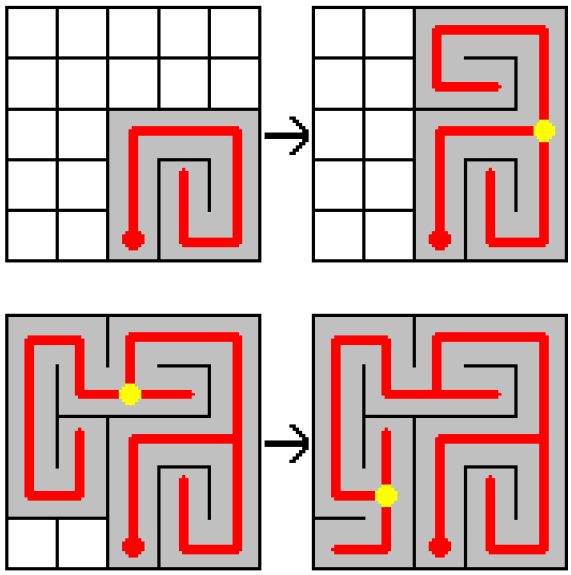

Figura 1: *Ejemplo de un algoritmo recursive backtracker [1]*

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:

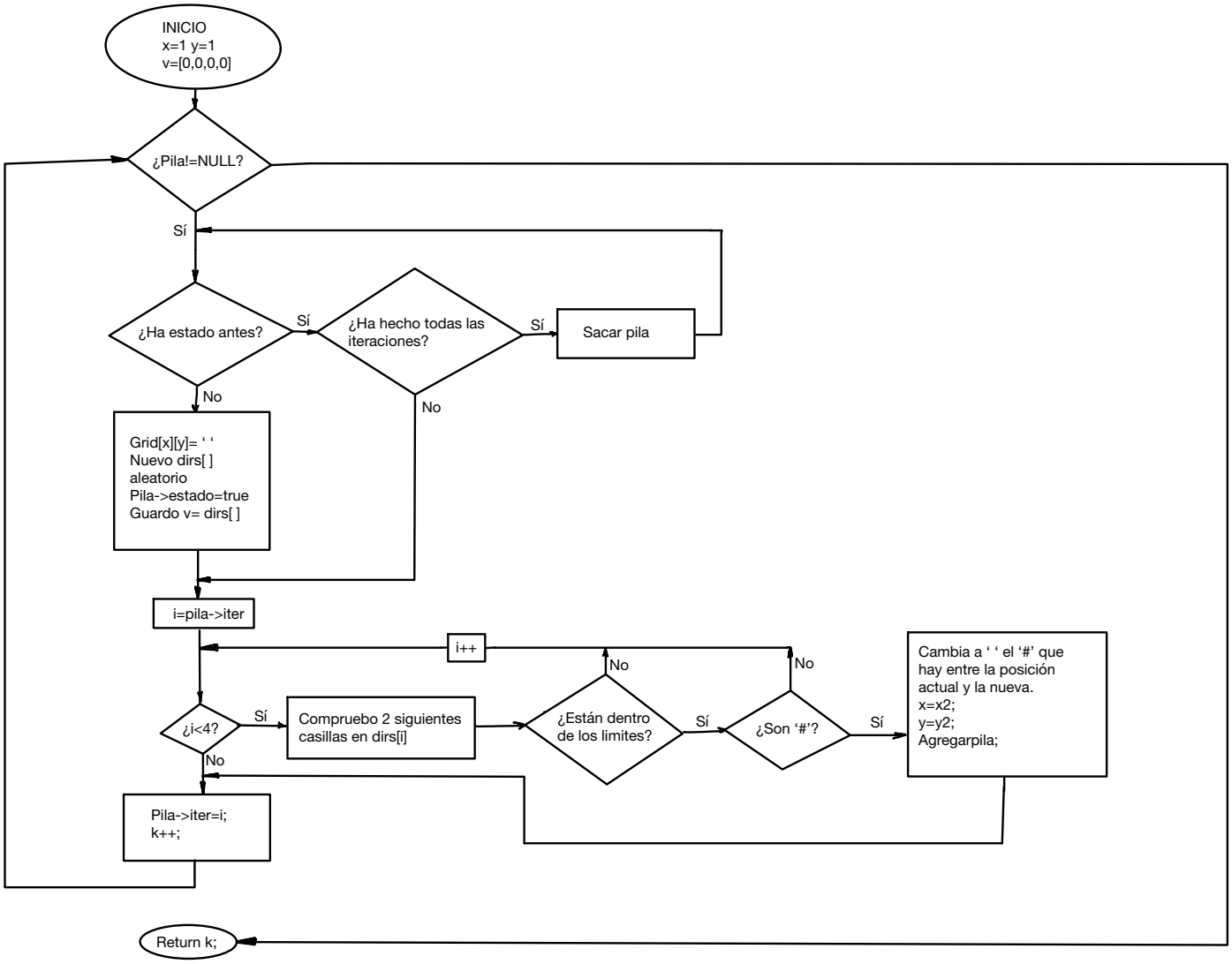

Figura 2: *Diagrama de flujo para la función Visit() en el algoritmo iterativo*

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;
}
```

Variente para la función PrintGrid() con carácteres UNICODE

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 #.

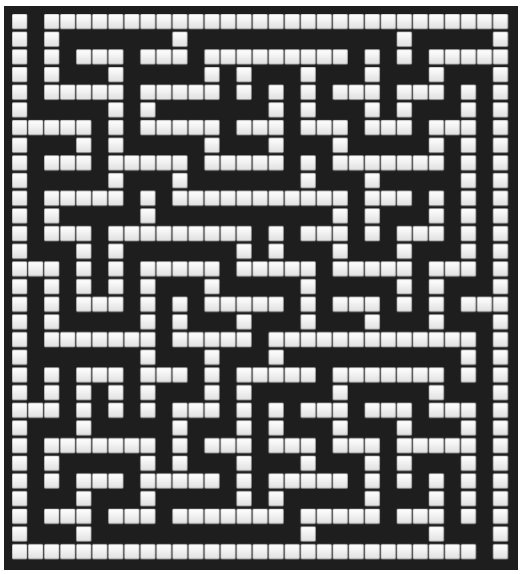

Figura 3: *Salida por consola del programa con carácteres* UNICODE

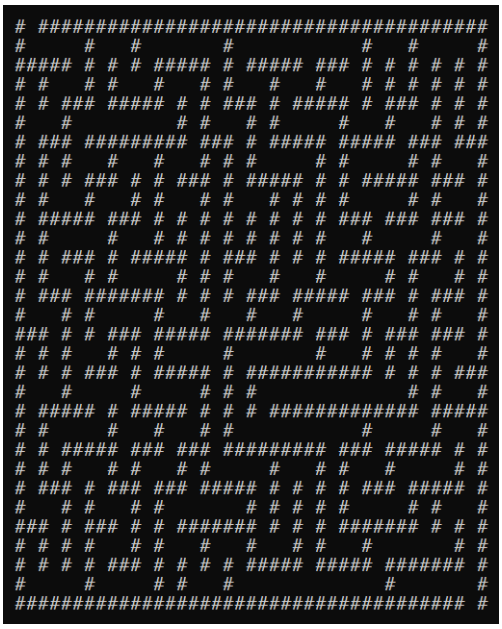

Figura 4: *Salida por consola del programa con carácteres* ASCII

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 medidas. 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 bajo 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 *.csv*. Para su estudio y análisis se ha utilizado la herramienta estadística 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 si se prefiere *output.txt*): 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 incluye a continuación el bucle principal del *shell script*:

```
initial=20
final=10000
step=50
contador=1

while [ $initial -le $final ]
  do
    while [ $contador -le 5 ]
    do
      echo -n $contador\ >> $file
      $mypath/Maze_Recursivo $initial $initial 1\ >> $file
      contador=$(($contador+1))
    done
    initial=$(( $initial + $step ))
    contador=$(($contador-5))
  done
```

Variente para la función PrintGrid() con carácteres UNICODE

Es importante utilizar `ulimit -s unlimited` dentro del script para evitar que el sistema operativo mate el programa cuando las dimensiones sean demasiado grandes, permitiendo que la pila sea más grande de lo habitual y no llegue a llenarse. Para poder ejecutar el mismo *script* en Windows ha sido necesario utilizar un emulador de Unix (*Cygwin* o una consola tipo *Git Bash*) y cambiar el comando anterior por `/STACK:reserve[10000000]` con la misma inteción.

En la toma de medidas se ha observado que en el 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, por otro lado, las 5 ejecuciones sirven para obtener la media del tiempo.

En el caso del iterativo, tanto el tiempo como el número de iteraciones varían en cada ejecución. Se cree que esto es debido a la aleatoriedad del recorrido y el vaciado o llenado de la pila. Pues, en el caso recursivo el "arbol de opciones" es recorrido de manera sistemática.

A la hora de realizar pruebas, se ha dispuesto de 3 computadoras diferentes: dos con sistemas operativos vasados en UNIX, en concreto *Linux*, ambos con diferentes procesadores, y un tercer computador con sistema operativo *Windows 10*. Se ha procedido extrayendo datos de la ejecución de los programas en cada uno de las tres computadores. Lo cual, supone además una manera de comprobar experimentalmente la portabilidad de los programas.

Es importante resaltar en este punto, lo conveniente que ha sido, para la ejecución del *script*, implementar en ambos código el paso de parámetros a la función main(). Se incluye brevemente el código y el paso de parámtros a una función de dichos argumentos:

```
//----FUNCION PRINCIPAL-------------------------------------
int main(int argc, char **argv){
  // Se pide al usuario el tamano del laberinto.
    Pedir(argc, argv);
```

Implementación de paso de argumentos al main()

FASE 3

Análisis

Gracias a los datos obtenido en la fase de experimentación, se ha podido elaborar una base de datos de tamaño suficiente. La depuración y el tratamiento de los datos se ha llevado a cabo utilizando el software R en el entorno de desarrollo RStudio. Se ha realizado en varias pasos:

- En primer lugar, se importaban los datos a RStudio desde el propio documento de salida, en formato *.csv*, que elaboraba automáticamente el *shell script*.
- Los documentos se inspeccionaban rápidamente mediante gráficos en busca de posibles datos atípicos. Al pasar a través de distintos sistemas operativos era común que las comas decimales se desplazaran, lo que provocaba grandes errores que debían ser detectados.
- Solucionado este problema, había que hacer la media de los tiempos, para lo que se ha utilizado la función tapply() de la librería "tidyvers", añadir un factor que recogiera el microprocesador utilizado y el sistema operativo en cada ejecución y por último volcar cada tabla de datos a un documento *.csv* de nuevo.
- Una vez los datos están depurado y tratados y volcado en sus respectivas tablas, se han unificado en unico archivo. Para poder hacer los gráficos posteriores adecuadamente, ha habido que volver a transformar el dato del microprocesador y el sistema operativo a tipo factor, pues por defecto, R no asigna variables procedentes de una lectura externa a un tipo factor.
- Con la tabla final, se procede a elaborar los gráficos pertinentes. Para ello se hace uso del paquete *ggplot* de la librería "tidyvers".

En el siguiente gráfico (ver Fig. 5) se ha representado la evolución del numero de veces que se llama a la función Visit() en función del numero de filas (o columnas) de un laberinto de matriz cuadrada. Como se aprecia, para dimensiones pequeñas (número de filas no mayor que 2500), el algoritmo iterativo y el recursivo presentan un número de iteraciones parecido. Cuando el número de filas empieza a ser elevado, las diferencias entre ambos son notables. A priori, con este gráfico podría decirse que el método iterativo es "mejor", pero son necesarios otros gráficos con menos sesgado para realizar conclusiones definitivas.

El siguiente gráfico (ver Fig. 6) es en esencia similar al primero. La principal diferencia radica en que ahora varía el número de elementos del laberinto, no la dimensión de sus filas o columnas. Con este cambiio podemos inferir más adecuadamente el número de llamadas a la función Visit() en el caso de que el laberinto no sea cuadrado. Como puede observarse, la relación entre el número de llamadas y el "área" de la matriz es absolutamente lineal de pendiente aproximada 1/5 para el algoritmo iterativo y 1/2 para el recursivo. La conclusión que se desprende de este gráfico y del anterior es la misma: el algoritmo iterativo es mejor. Pero esta afirmación es completamente errónea, como demostramos con los dos gráficos siguientes.

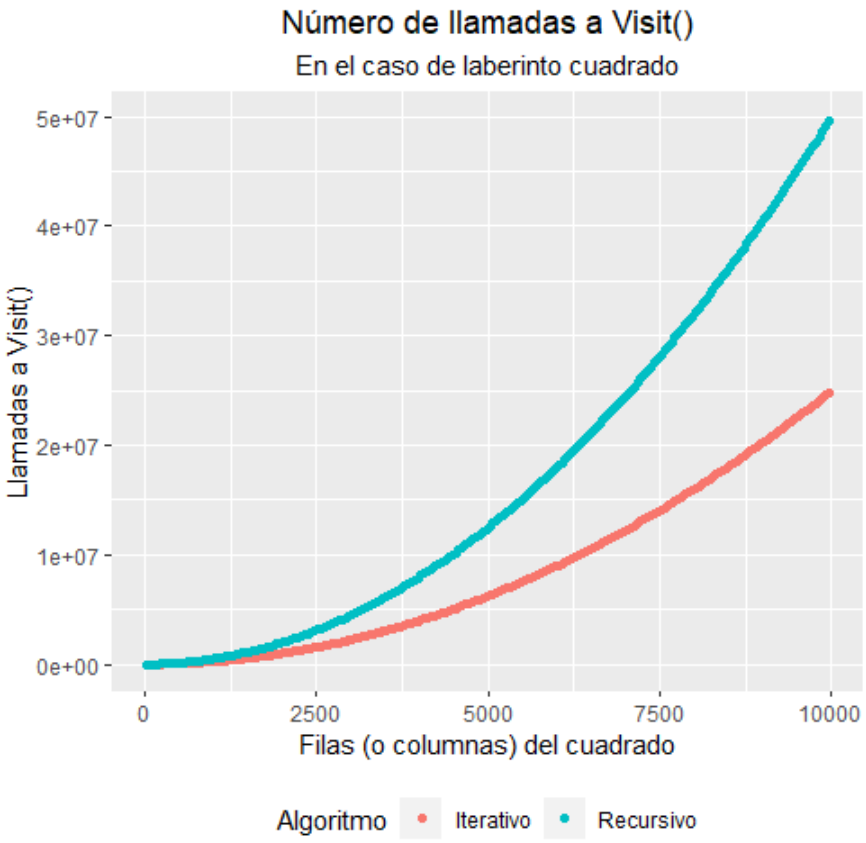

Figura 5: *Comparativa de número de iteraciones frente a la dimensión de un lado (supuesto el otro igual) para el caso recursivo e iterativo*

En los gráficos de la siguiente página (o de donde estén), se han comparado los tiempos reales de ejecución completa del programa para los dos algoritmos, para laberintos cuyo número de elementos es inferior a 100 millones, en tres equipos distintos, de distintas gamas y con distintos años de uso a fin de mostrar la relevancia de estas características en el desempeño de los programas. Por cuestiones de tiempo no se tienen datos del desempeño del algoritmo iterativo en un equipo con Windows 10 de gama alta (con procesador Intel core I7). Se observa que el algoritmo recursivo es claramente mejor que el iterativo, llegando a recortar en más de dos segundos la ejecución del iterativo en el caso del laberinto más grande evaluado (9971 x 9971 elementos). Lo cual equivale a una mejora sustancial de entre el 31 % y el 33 % respecto al iterativo. Como vemos, el desempeño real es radicalmente opuesto al que pudiera predecirse a partir de los gráficos de las llamadas a la función Visit().

Estas notables diferencias son debidas a que los dos primeros gráficos están muy sesgados, pues solo miden la llamada a la función principal de ambos algoritmos, la función Visit(). Si bien esto puede no implicar que su medida incurra en un sesgo importante para determinar cuan bueno es el algoritmo recursivo, sí se incurre en sesgos mayores cuando se trata del algoritmo iterativo. Esto es así porque el mayor peso del recursivo recae en la función Visit(), y la propia naturaleza de la recursividad impone que Visit() se llame una y otra vez a sí misma. Por lo tanto, midiendo la cantidad de veces que se llama a Visit() en el algoritmo recursivo, proporciona una medida útil. Sin embargo, los pesos computacionales del algoritmo iterativo están más repartidos (Visit() llama a otras funciones que no se están contabilizando) y

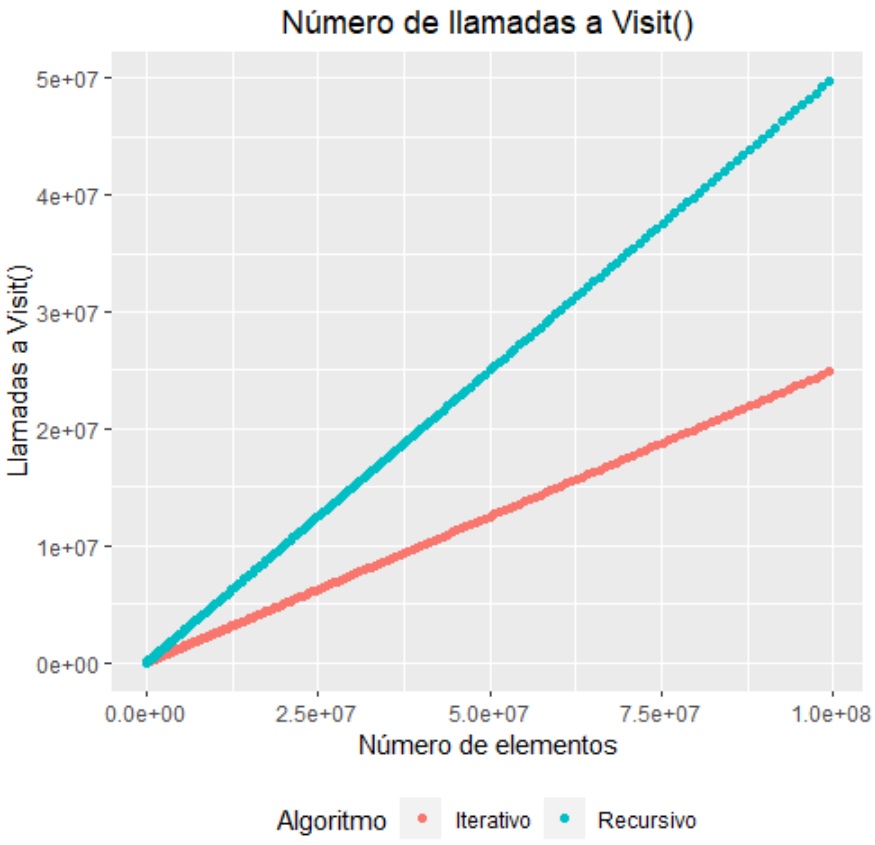

Figura 6: *Comparativa de número de iteraciones frente a la dimensión de un lado (supuesto el otro igual) para el caso recursivo e iterativo*

por ende, la medida de la cantidad de veces que se llama a Visit() en este caso, es menos útil, porque es menos extrapolable y comparable, que la misma medida en el recursivo.

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].