Universidad de Costa Rica
Escuela de Ingeniería Eléctrica
IE0-117 Programación bajo plataformas abiertas

Proyecto

Corredor del laberinto

Maycol Sáenz Jarquín
A95616

Julio 2022

Índice

Introducción

El algoritmo aplicado para la solución del laberinto se basa en tres pasos: la construcción del laberinto como matriz de números enteros a partir del archivo de texto, la identificación de todos los puntos de entrada en los bordes del mismo y su almacenamiento en una lista enlazada, y finalmente, la búsqueda recursiva del número 2 a partir de dichas entradas.

El propósito de este documento es explicar la relación entre las funciones creadas en los tres archivos de código fuente, y principalmente, describir las secciones que componen el archivo *conductor* del algoritmo (*main.c*), ya que sus llamados a las funciones contenidas en las dos bibliotecas son la clave para entender la lógica de la resolución del problema.

El análisis individual de las líneas de código de cada función y biblioteca, sin embargo, se deja a consideración del lector. Esto debido, en primera instancia, a que no se busca replicar aquí el repositorio de acceso público. En segundo término, porque el uso de la sintaxis es una cuestión de técnica que varía según las preferencias de cada desarrollador.

Al final de la lectura de este informe, el usuario será capaz de contribuir con sus propias mejoras a la solución presentada, e incluso, tomar ésta como base para encontrar la solución aplicando otros algoritmos de resolución de laberintos, ya que dispondrá de las funciones utilitarias, gracias a la modularidad otorgada por las bibliotecas.

Diseño general

Principios para crear el algoritmo

El algoritmo para solucionar el laberinto se basa en el razonamiento de 3 pasos, como se puede ver en la Figura 1. En el primer paso, tras la lectura del archivo, se convierte la matriz de caracteres tipo *char* en una matriz de tipo *integer* con la idea de hacer el manejo de las funciones de forma más sencilla.

El segundo paso consiste en encontrar y almacenar cuantos puntos de entrada existan en los bordes del laberinto, para posteriormente ser evaluados por la función que buscará la solución. Finalmente, el tercer paso es imprimir el camino encontrado con el primero punto que lleve a una respuesta válida.

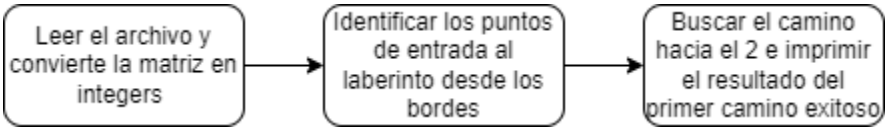


Figura 1: Las 3 ideas principales para crear el algoritmo

Algoritmo expandido

El vector de caracteres y la matriz de *integer* creada a partir de la misma, son guardadas en el HEAP para mantener al mínimo la memoria utilizada para resolver el problema.

La Figura 2 muestra el diagrama de flujo que explica el algoritmo implementado. Note la matriz de ceros creada con la finalidad de ser impresa al final del programa, reemplazando los *ceros* con *unos* en el camino válido hacia la solución.

La función recursiva es la parte central del algoritmo, ya que evalúa cada uno de los puntos de entrada al laberinto, y se detiene ante uno de las siguientes dos condiciones:

- Si encuentra el número 2, incluso si no es el camino más corto. La función devuelve el par ordenado de entrada evaluado e imprime el camino encontrado.
- Si llega al final del laberinto sin una respuesta. La función devuelve ***No hay solución***.

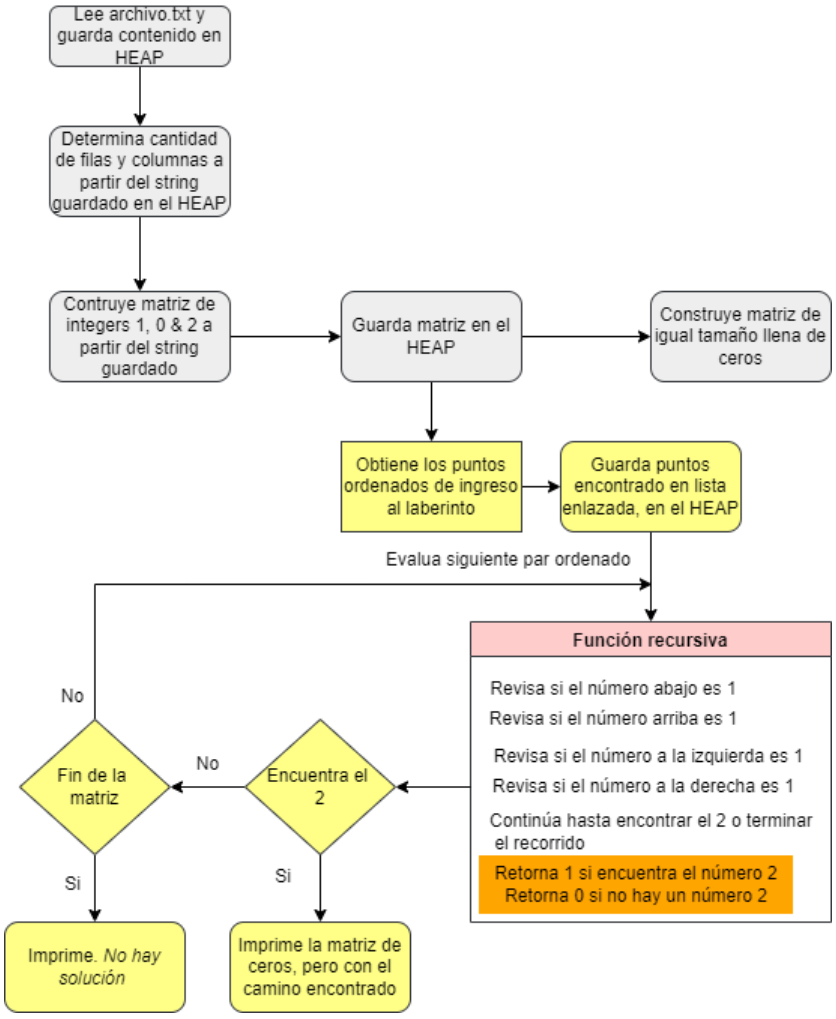


Figura 2:diagrama de flujo del algoritmo

Bibliotecas con las funciones

El programa se divide en 6 archivos de código fuente. Los primeros 3 corresponden a la función *main()*, y dos bibliotecas que contiene las funciones implementadas para cada paso descrito en la Figura 1. Los otros 3 archivos corresponden a los encabezados (*headers*) para la implementación de las bibliotecas.

La Figura 3 explica la relación entre los archivos y los *headers* utilizados para especificarlos como bibliotecas.

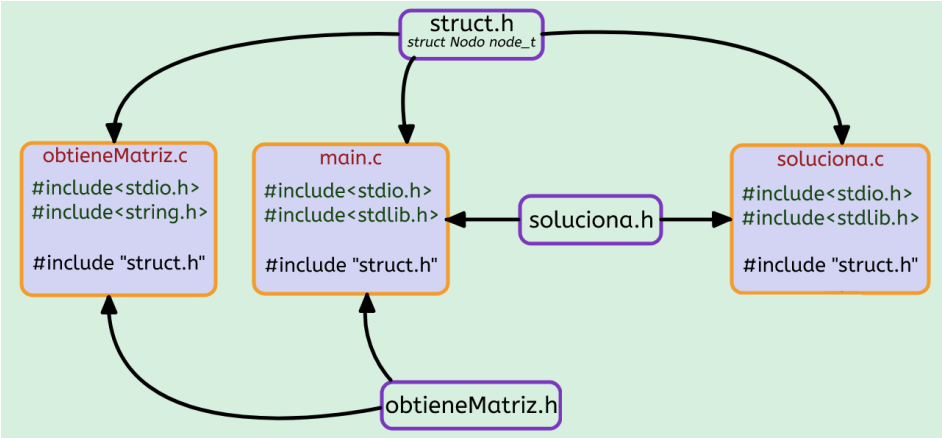


Figura 3: Diagrama general de relación entre archivos de código fuente y las bibliotecas

Implementación de la función *main()*

La función *main()* dirige el algoritmo mediante la llamada a las funciones en las bibliotecas. Este llamado puedo verse en las primeras líneas del código en la Figura 4, donde se llama a las tres bibliotecas.

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "struct.h"
4  #include "obtieneMatriz.h"
5  #include "soluciona.h"
6
7  int main(){
8    //Seccion 1: lectura del archivo laberinto.txt
```

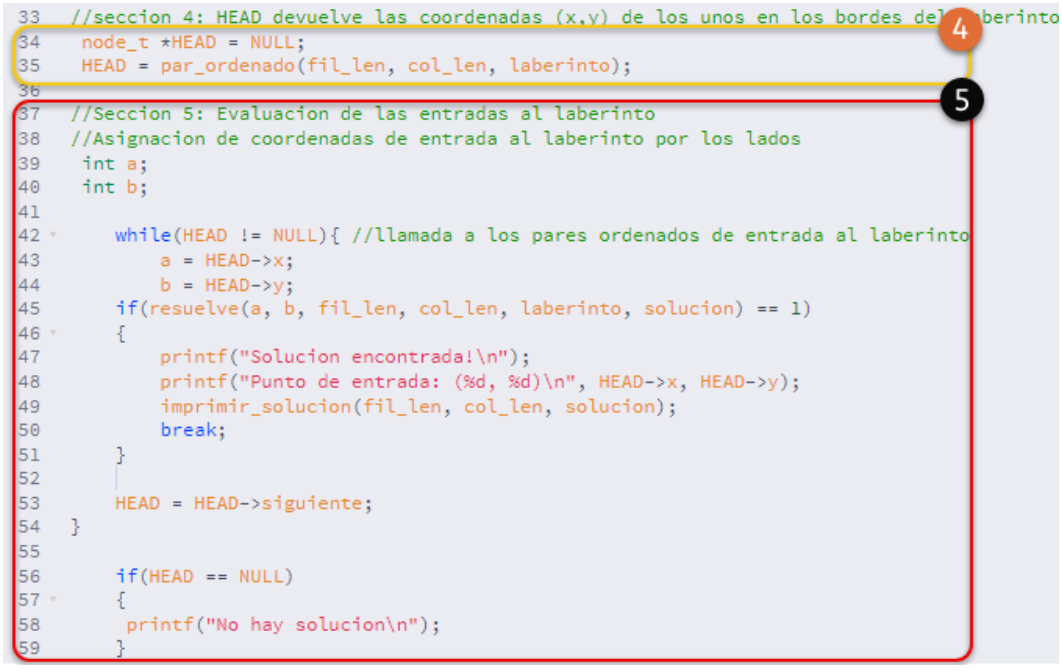
Figura 4: Llamada a las bibliotecas desde la función main()

La función *main()* puede ser analizada en 6 secciones señaladas por los comentarios. Las primeras tres secciones, mostradas en la Figura 5, tienen la finalidad de procesar y preparar la información de forma que sea utilizada por la función recursiva para la determinación de un camino solución. Para ello, se calcula la cantidad de filas y columnas a partir de los datos leídos, y se convierte la matriz de tipo *string* en una matriz de tipo *integer*.


```c
int main(){
  //Seccion 1: lectura del archivo laberinto.txt

  char *informacion = malloc(10000*sizeof(char));
  FILE *archivo = fopen("./laberinto.txt", "r");

  fread(informacion, 10000*(sizeof *informacion), sizeof informacion, archivo);

  int col_len = cuenta_columnas(informacion);
  int fil_len = cuenta_filas(informacion, col_len);

//Seccion 2: construccion del laberinto para analizar. Se guardara en el heap.
    int (*laberinto)[fil_len][col_len] = malloc(sizeof*laberinto);
    for(int m = 0; m < fil_len; m++){
     for(int n = 0; n < col_len; n++){
      (*laberinto)[m][n] = Contruye_matrix(fil_len, col_len, informacion, m, n);
    }
  }
//Seccion 3: se rellena la matrix solucion con ceros
  int (*solucion)[fil_len][col_len] = malloc(sizeof*solucion);
  for(int i = 0; i < fil_len; i++){
    for(int j = 0; j < col_len; j++){
      (*solucion)[i][j] = 0;
    }
  }
```

Figura 5: Secciones de la función main() para preparar la información

Posteriormente, en la sección 4 mostrada en la Figura 6, se utiliza la función *par_ordenado()* de la biblioteca *soluciona.h* para guardar todos los puntos iguales a 1 en los bordes del laberinto, su implementación usa el concepto de listas enlazadas, ya que permite guardar una cantidad indeterminada de entradas.




```c
//seccion 4: HEAD devuelve las coordenadas (x,y) de los unos en los bordes del laberinto
 node_t *HEAD = NULL;
 HEAD = par_ordenado(fil_len, col_len, laberinto);

//Seccion 5: Evaluacion de las entradas al laberinto
//Asignacion de coordenadas de entrada al laberinto por los lados
 int a;
 int b;

    while(HEAD != NULL){ //llamada a los pares ordenados de entrada al laberinto
        a = HEAD->x;
        b = HEAD->y;
    if(resuelve(a, b, fil_len, col_len, laberinto, solucion) == 1)
    {
        printf("Solucion encontrada!\n");
        printf("Punto de entrada: (%d, %d)\n", HEAD->x, HEAD->y);
        imprimir_solucion(fil_len, col_len, solucion);
        break;
    }

    HEAD = HEAD->siguiente;
}

    if(HEAD == NULL)
    {
     printf("No hay solucion\n");
    }
```

Figura 6: La sección 4 muestra el uso de las funciones que resuelven el problema

La resolución del problema se da en la sección 5, donde se usa el valor devuelto por la función *resuelve()*. Esta función recursiva utiliza el concepto de *backtracking*, el cual consiste en analizar el siguiente punto a partir de una entrada, y seguir el camino realizando esa misma evaluación hasta llegar al objetivo, en este caso, encontrar un número 2 al final del recorrido.

La segunda parte de la sección 5 consiste en el recorrido de los puntos de entrada al laberinto para entonces imprimir la matriz solución con el camino encontrado o para imprimir que no hay solución.

Finalmente, la sección 6 mostrada en la Figura 7, hace la llamada a la función de liberación de los espacios de memorias del *Heap* usados para guardas las variables durante la ejecución del programa.

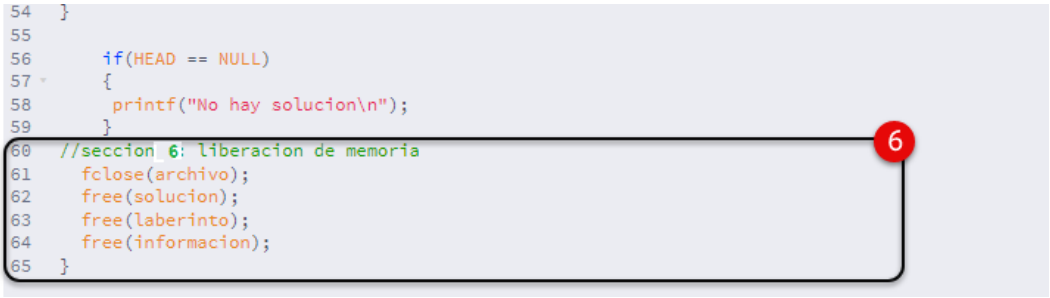


Figura 7: Liberación de los espacios de memorias utilizados en el Heap.

Principales retos

1. Entender el funcionamiento de los punteros: el comportamiento de los punteros, apuntando al Stack o al Heap resulta impredecible en algunas ocasiones. Varias veces fue necesario recurrir al *debugger* para conocer el valor de las variables llamadas mediante un puntero.
2. Entender como está compuesta la *Memoria de aplicación*. Se infiere que parte del proyecto fue investigar sobre su uso, ya que es necesaria para entender el comportamiento de los punteros.
3. La división del archivo fuente en varios archivos siguiendo el concepto de bibliotecas. El proyecto se llevó hasta casi su terminación en un solo archivo. La separación consumió mucho tiempo. Especialmente la biblioteca para definir la estructura *typedef struct Nodo node_t*, ya que, tras prueba y error, se determinó que sólo se ocupa el archivo *struct.h* con su definición para funcionar.
4. Investigar y comprender los métodos de resolución de laberintos. Lo cual implicó entender el funcionamiento del algoritmo de *backtracking*. El cual se eligió como solución para encontrar el objetivo, ya que es un método recursivo fácil de entender e implementar.
5. Adaptar soluciones encontradas en internet a los propósitos del proyecto. Es complicado, pero instructivo, poder convertir ejemplos útiles en otros lenguajes de programación como Java, C#, C++ y Python, al language C. En la mayoría de los casos es necesario entender la lógica aplicada detrás de la sintaxis de esos lenguajes.

Conclusiones

La experiencia de crear el proyecto requiere mucha persistencia y tiempo de investigación y dedicación frente a la computadora. Sobre todo, experimentar mucho con el lenguaje hasta comprender como funciona su sintaxis y el manejo de la memoria.

Lecciones aprendidas

1. El 90% del tiempo invertido en el proyecto consistió en la investigación sobre temas de programación en c, herramientas de desarrollo y lenguajes de programación. En resumen, para completar el proyecto fue necesario investigar y aprender por cuenta propias los siguientes aspectos:
   i) El uso de *header guards* en los archivos *header* de las bibliotecas, con el propósito de evitar errores de compilación por llamadas múltiples de las funciones.
   ii) Profundizar en el concepto de *memoria de aplicación*, ya que sólo así se entiende el funcionamiento de los punteros y de su uso junto con el espacio de memoria llamado Heap.
   iii) El aprendizaje del funcionamiento de las *listas enlazadas*. Este consiste en llevar una estructura de datos común al siguiente nivel usando un puntero hacia sí mima. En el proyecto se usó para guardar la cantidad de las entradas al laberinto, ya que la cantidad es diferente para cada caso, por lo que almacenar los pares ordenados en un vector no es práctico.
   iv) La creación de archivos tipo *Makefile* para realizar la compilación de todos los archivos con un solo comando.
   v) El funcionamiento de Git, y GitHub. En ese último, el aprendizaje del lenguaje de programación *Markdown* para la edición del contenido de archivos Readme.md
2. Se aprendió la diferencia de las funciones de lectura de archivos, fread() y fscanf(), donde la primera permite leer todo un archivo texto a la vez, mientras que la segunda está enfocada en la lectura de un carácter por llamada.
3. Es mejor separar las funciones en bibliotecas desde el principio, que separarlas una vez el programa completa está en un solo código fuente, ya

que después hay que resolver problemas de referencia a punteros que quitan mucho tiempo.

Preguntar sin responder

1. Una mejora importante para poder procesar cualquier tamaño de laberinto (tamaño del archivo de texto de entrada), es poder expandir la memoria inicialmente asignada en el *Heap*. Hay mucha documentación referente al uso de la función *calloc* para la reasignación de más espacio de memoria, pero no fue posible implementarla junto con la función *fread()*.
2. Qué tan modular debe ser un proyecto según las buenas prácticas de programación. Esta pregunta surge ante la posibilidad de crear una biblioteca por función o tener funciones que cumplen un trabajo relacionado dentro de una misma biblioteca. Esta última opción fue la escogida aquí, pero sin certeza si esa es la mejor práctica.
3. La solución presentada se detiene con la primera entrada que permite encontrar el objetivo dentro del laberinto. Es decir, no se da cuenta si hay un camino que sea más corto. Queda la duda de si es posible utilizar la función recursiva para contar la cantidad de pasos de las soluciones e imprimir la más corta, o si hay que reescribir la función totalmente. Queda como un buen reto para el futuro.