Documentación
MazeSolver
V. 1.0.

Ramírez Ortega Angelo
2017080055
D’Ambrosio Soza Guilliano
2017158561

Escuela de Ingeniería en Computación
Estructuras de Datos
Profesor: José Dolores Navas Su
Instituto Tecnológico de Costa Rica
Martes 21 de noviembre del 2017

Índice

Resumen Ejecutivo

A continuación, se abarcaran todos los temas relacionados con el programa MazeSolver. Se propuso encontrar una manera de generar, resolver, graficar, guardar y cargar laberintos de tipo “maze”, los cuales son estructuras intencionalmente complejas que cuentan con caminos con bifurcaciones entre dos puntos, representados mediante la utilización de grafos no dirigidos. Los laberintos contarán con 20 filas y 30 columnas de nodos, donde se generarán laberintos creando arcos entre estos mediante el algoritmo de búsqueda en profundidad modificado. Posteriormente, al tener un laberinto generado se utiliza el algoritmo de búsqueda del camino más corto de Edsger W. Dijkstra para determinar la ruta más corta entre el nodo inicial de la esquina superior izquierda y el nodo final de la esquina inferior derecha. La solución se desarrolló en el ambiente de programación CodeBlocks con la librería wxWidgets en el sistema operativo Ubuntu.

La justificación de la realización de este proyecto es su enfoque pedagógico, consolidando así la profundización de conceptos estudiados en el curso de Estructuras de Datos del Tecnológico de Costa Rica como: grafos, estructuras de datos jerárquicas, estructuras de datos secuenciales, generación de interfaces gráficas en C++ y el manejo de archivos en forma de lectura y escritura para darle permanencia a los laberintos. La solución implementada se detalla en profundidad en este documento. Se logró implementar con éxito cada aspecto requerido para esta programación. Cabe recalcar entre las recomendaciones generales la importancia de investigar las estructuras viables para la solución, para poder determinar cuál resulta más adecuada y así facilitar la solución y evitar sufrir inconvenientes con la fecha de entrega. Además, siempre se pretende utilizar código limpio para una codificación más eficaz.

Introducción

Este proyecto nace a raíz de una tarea programada que plantea un programa con la capacidad de generar, resolver, graficar, guardar y cargar laberintos de tipo “maze” por medio de una interfaz gráfica. Un laberinto de tipo “maze” es una estructura intencionalmente compleja, que contiene caminos entre puntos con bifurcaciones, las cuales pueden llevar a la salida, o no. Provienen de la mitología griega dónde Dédalo diseñó una estructura que llamo *labyrinthos* para mantener preso a Minotauro, una criatura mítica mitad hombre mitad toro. Este proyecto busca fomentar el trabajo en equipo de los participantes, resaltando así la importancia de la comunicación efectiva, la organización grupal y el código limpio, ya que sin estas herramientas el proceso de codificación toma más tiempo del debido y puede que se encuentren problemas con la fecha de entrega.

Adicionalmente, se pretende reforzar los temas estudiados en el curso, juntando diversas herramientas adquiridas y darles así un uso práctico. Se incluyen temas como grafos, estructuras de datos jerárquicas, estructuras de datos lineales, generación de interfaces gráficas en C++ y el manejo de archivos en forma de lectura y escritura para darle permanencia a los laberintos. Se pretende generar laberintos mediante la utilización de una búsqueda a profundidad modificada, resolverlos utilizando el algoritmo de búsqueda del camino más corto propuesta por Edsger W. Dijkstra, graficarlos y guardarlos en memoria para poder cargarlos posteriormente. Al guardarlos en memoria se ideó una manera de acceder a cualquier laberinto, mediante índices. Así es fácil determinar si el índice que está buscándose existe, de lo contrario, se le índica al usuario para que revise y lo intente nuevamente.

Presentación y Análisis del Problema

Problema

El problema principal de del proyecto es determinar cómo generar los laberintos, ya que hay que determinar cómo se le dará forma a la estructura de grafo por utilizar, de manera que sea optima tanto como para encontrar un camino entre nodos, como para poder obtener información para graficar el laberinto y su solución y poder guardarlos en memoria.

- Estructura del grafo por utilizar:

Para poder solucionar el problema presentado primero se tuvo que pensar la información que contendrían los nodos de la solución. Se optaron por coordenadas, de forma que cada nodo tiene una coordenada x y una coordenada y, característica que facilita la graficación, ya que cuando dos nodos contienen un arco entre ellos, basta con obtener las coordenadas para mostrar la ruta en pantalla, mediante un DrawLine facilitado por la librería wxWidgets. Se requiere, entonces, que la estructura de grafo contenga una lista con vértices y una representación de arcos mediante una matriz de adyacencia, así, se puede determinar con facilidad si existe un arco entre dos nodos, obtener su peso y todos los vecinos, facilitando la solución. Además, se requiere tener un mecanismo que permita determinar si se ha visitado un vértice en particular, para esto se utilizó una lista de booleanos paralela a la lista de vértices. Queda así entonces preparada la estructura, que nos da la capacidad de agregar vértices, arcos, visitar un nodo, obtener vecinos y algoritmos complementarios internos que aportan a la solución del problema.

- Generación de un laberinto:

Para poder generar un laberinto se consideraron tres algoritmos propuestos por el profesor, cada uno tiene características propias y necesita modificaciones a la estructura. Se ha de determinar cuál es el óptimo por desarrollar una vez que se determine la estructura de grafo por utilizar.

1- Búsqueda en profundidad modificada: Esta estrategia utiliza el algoritmo de búsqueda en profundidad, pero con un factor aleatorio que permite generar laberintos. Genera un grafo conexo, que contiene pasillos largos, por lo que

resulta fácil de resolver. Puede generar inconvenientes de desbordamiento de pila porque tiene un alto nivel de recursión.

2- Algoritmo de Kruskal modificado: El algoritmo de Kruskal original genera una lista de arcos ordenados de mayor a menor peso, agregándolos al grafo de forma que no se generen ciclos. La modificación que se realiza es que se le añade un factor de aleatoriedad al orden de los arcos, de esta forma se generan patrones relativamente fáciles de resolver. Este algoritmo requiere de una estructura para implementar conjuntos.
3- Algoritmo de Prim modificado: El algoritmo de Prim original va generando un grafo a partir de un punto, seleccionando las aristas más cortas, hasta que todos los vértices se encuentren en el nuevo grafo. La modificación consiste en volver la selección de aristas aleatorias. De esta forma se generan laberintos con un nivel de complejidad similar al de Kruskal.

- Encontrar la librería correcta:

Una parte muy importante de todo el proyecto es escoger una librería o librerías que sean útiles y eficaces para los propósitos que se tienen. Si no se escoge la librería óptima entonces el proyecto puede alargar en duración y volverse más complicado. Una buena librería facilita todo el proceso de graficación e interfaz con el usuario además de facilitar las operaciones necesarias para lograr el fractal.

- Resolviendo el laberinto generado:

Uno de los requisitos principales del proyecto es poder obtener y graficar una solución para los grafos generados. Existen diversos algoritmos para encontrar rutas entre grafos, inclusive utilizando los algoritmos de búsqueda a profundidad o a lo ancho. Pero estás formas resultan muy ineficientes, por lo que se favorece la utilización del algoritmo para conseguir la ruta más corta propuesto por Edsger W. Dijkstra, haciendo backtracking para determinar la ruta una vez se llegue al vértice determinado.

- Graficación de laberintos:

Se requiere poder representar laberintos, para esto se pueden ver los grafos generados cómo los caminos que se pueden seguir, por lo que basta darles coordenadas a los vértices, se debe crear una forma para poder recorrer el grafo de

laberinto y poder unir los puntos, así como crear las paredes del laberinto ya que el recorrido del laberinto lo que deja son las paredes del laberinto.

Solución al Problema

A continuación, se detallará cómo se lograron implementar con éxito soluciones a los problemas dados, con la guía de las especificaciones dadas por el profesor y adaptando los algoritmos al nuevo uso.

- Estructura de grafos:

Se optó por utilizar una estructura de grafos que contiene representación de arcos mediante una matriz de adyacencia, una lista de valores booleanos que determina si un vértice en particular se ha visitado y una lista de vértices que mantiene las referencias a los vértices creados. Contiene la implementación también de los métodos para añadir vértice, añadir arco, determinar si existe un arco entre dos vértices, determinar si un vértice ha sido visitado, resetear las visitas a todos los vértices, obtener lista de vecinos de un vértice en particular y determinar si existe por lo menos algún vértice que no ha sido visitado. Todo esto fue hecho utilizando los vectores estándar de C++ y una clase creada de lista enlazada.

- Generación de laberintos:

El primer paso que se siguió para la generación de laberintos fue la creación de un grafo, el cual contaba con estructura de una cuadricula de 20x30, es decir, los vértices están conectados de forma de que se conecta, de no ser borde, con los vértices de encima, debajo, derecho e izquierdo, así se tiene una plantilla general para los laberintos. El segundo paso por seguir es la utilización del algoritmo de búsqueda en profundidad modificado, dónde se van añadiendo arcos de vértices vecinos a un vértice de manera aleatoria, hasta que todos los vértices se encuentren en el nuevo grafo, éste no cuenta con ciclos, pero es conexo, ya que se puede llegar a cualquier nodo desde cualquier nodo. Paralelo a este proceso se genera una lista de arcos en forma de vértices adyacentes, ésta lista será recorrida, de forma de que se van obteniendo los vértices y con sus coordenadas se van creando líneas usando el DeviceContextManager de WxWidgets con el método DrawLine(xi,yi,xf,yf).

- Encontrar la librería correcta:

Para este proyecto se utilizó la librería wxWidgets. Esta librería permite la creación de ventanas en la cual se despliega la interfaz y también la creación de líneas en dicha ventana. Esta librería facilitó mucho el proceso de interfaz gráfica ya que tenía métodos fáciles, versátiles y completos para cada uno de nuestros propósitos.

- Resolviendo el laberinto:

Para resolver el laberinto se decidió utilizar el algoritmo de Dijkstra, ya que contamos con una estructura de grafos que permite encontrar la solución recorriendo nodos, marcándolos como visitados y buscando los vecinos, de forma que se va buscando la ruta más corta hasta que se consigue el camino correcto. Para esto se crearon dos listas adicionales, una lista de distancias y una lista de vértices anteriores, la lista de distancias se inicializa en un valor pseudoinfinito para cada vértice, para poder saber si se ha encontrado una ruta hacia un vértice en particular. La lista de vértices anteriores nos permite hacer un recorrido en retroceso desde el vértice de salida hasta el vértice de entrada, cuando se consiga la solución. El primer paso del algoritmo es inicializar los valores de las listas de distancias y vértices anteriores, dónde la distancia se coloca en infinito y los vértices anteriores como indefinidos, así como marcar cada uno de los nodos como no visitado. El segundo paso es colocar el valor de la distancia del nodo inicial como 0, ya que esto nos dice que empezamos justamente en este nodo. Posteriormente, mientras exista por lo menos un nodo sin visitar y no se haya encontrado un camino hacia la salida, asignamos un nodo auxiliar, que será el vértice no visitado con la distancia mínima en la lista de distancias. Se buscan todos los vecinos no visitados de este vértice y para cada uno de ellos se hace una evaluación de su distancia que consiste en determinar cuál es el peso de la ruta si se proviene del nodo auxiliar, de ser esta menor a la distancia de la lista que se tiene, entonces se le asigna esta como su nuevo valor y se le asigna como vértice previo el vértice auxiliar. Si el vecino encontrado es igual al objetivo, quiere decir que se encontró una ruta. Al haber encontrado una ruta se procede a realizar el retroceso, Se empieza desde el último nodo y se va empujando a una pila su vértice anterior hasta que se llegue al vértice

inicial. De esta forma se obtiene una pila que al hacerle pop nos da el recorrido preciso de vértices y nos permite graficar una solución.

- Interfaz de usuario:

Para la interfaz se optó por una interfaz simple e intuitiva para el usuario, ya que no era necesario utilizar complejas interfaces debido a las pocas funciones de este proyecto. Se utilizan tres ventanas, una principal en la cual el usuario la modalidad de graficación por utilizar, ya sea carga o generación aleatoria. Dentro de la ventana de generación aleatoria se le permite al usuario crear una cantidad indefinida de laberintos, así como encontrar su solución y guardarlos en un archivo para su futura utilización. Dentro de la ventana de carga, el usuario selecciona un laberinto por índice y este se despliega en pantalla y permite que el usuario vea la solución.

Análisis de Implementación

Todos los requerimientos del proyecto se lograron llevar a cabo. Se lograron generar laberintos aleatorios, graficarlos en pantalla, encontrar soluciones correctas, guardarlos en memoria y cargarlos para desplegarlos nuevamente mediante selección por índices, además de que se le dio la investigación adecuada a cada aspecto de las estructuras por utilizar y los algoritmos adecuados para solucionar los problemas presentados, como, por ejemplo, cómo generar laberintos de expansión aleatorias con bifurcaciones , cómo calcular los puntos en el plano y cómo graficarlos en pantalla de manera de que tengan un aspecto agradable. Se logró reforzar también el trabajo en equipo recalcando la importancia de la buena planeación, la comunicación efectiva, el reparto de tareas y la práctica del código limpio. Entre los aspectos que se pueden mejorar queda darle la opción al usuario de que genere laberintos mediante diferentes algoritmos y darle la opción al usuario del tamaño de laberinto en filas y columnas, se omitieron por falta de tiempo debido a las demandas por parte de cursos adicionales diferentes al presente.

Conclusión

- Los laberintos son estructuras complejas por definición que pueden ser representados y solucionados mediante la utilización de estructuras de redes, ya que facilitan una representación de rutas con una matriz de adyacencia y métodos para determinar si se ha visitado previamente una celda en particular, lo cual permite una solución eficiente para una estructura compleja.
- Es posible generar laberintos de tipo “maze” mediante la utilización de algoritmos de recorrido de grafos modificados, en este caso se utilizó la búsqueda en profundidad con un factor aleatorio que tiende a generar estructuras con rutas relativamente simples de recorrer, se omitió la utilización de recursión para este algoritmo para no incurrir en desbordamientos de pila, para simular ésta se utilizó una estructura de pila.
- La librería utilizada afecta mucho al resultado del programa, cuando se selecciona una librería útil para el propósito se facilita mucho el proceso de creación de código o interfaz. En la situación adversa, al no usar librerías o usar una incorrecta ocasiona que todo el proceso sea más tedioso o no se pueda lograr. La librería wxWidgets resultó indispensable en este proyecto ya que facilitó la escritura del código de la interfaz y la graficación de líneas además de que sus métodos llevaron a un código más limpio y eficiente.
- El trabajo en equipo es una pieza fundamental para cualquier proyecto de naturaleza colectiva, habiendo que favorecer una comunicación clara con términos en común y un planeamiento en conjunto que permita explotar las características individuales de cada participante. Adicionalmente se recalca la necesidad de la utilización de código limpio como una herramienta que favorece la eficiencia de la escritura del código, de forma que no se pierda tiempo valioso por ambigüedad dentro del código, hay que pensar del código como una actividad colaborativa y social para que no se caiga en vicios de pereza de dejar nombres sin significado alguno o la creación de procesos con un propósito difícil de descifrar.

Recomendaciones

- Se recomienda empezar por la búsqueda o escritura de una estructura de redes que contenga todos los métodos y atributos necesarios para la creación y solución de laberintos, ya que esta es la base de la solución a todo problema presentado en este programa.
- Se recomienda el estudio de los algoritmos de búsqueda en profundidad como posible solución al problema propuesto, ya que al agregarse un factor de aleatoriedad se logran crear estructuras de grafos conexas acíclicas, similares a los laberintos de tipo “maze”
- Se recomienda la utilización de vértices que contengan pares ordenados con coordenadas, ya que a la hora de crear el laberinto la graficación se vuelve más sencilla al tener únicamente que acceder a dichos puntos y trazar líneas entre estos para tanto desplegar los laberintos como las soluciones
- Se recomienda inicialmente crear un grafo que contenga forma de cuadricula de nxm, dónde un vértice esté conectado con aquellos que serían adyacentes gráficamente, entonces para poder generar laberintos se utiliza el algoritmo de expansión acíclica seleccionado con modificación aleatoria, así se solventa el problema de cómo crear estructuras de grafos con forma de laberinto.
- Para la creación de archivos que funcionen para dar permanencia a los laberintos se puede generar un algoritmo que obtenga todos los arcos del laberinto y los guarde en una única línea dentro del archivo, en forma de números separados por comas, donde cada número representa una posición de grafo, de esta forma se puede acceder a cualquier laberinto por un índice, ya que todos cuentan con 20 filas y 30 columnas, basta con saber dónde están los arcos para poder desplegarlos nuevamente.
- El código limpio toma un rol altamente relevante cuando se trabaja en grupos, ya que, como hay más de una persona revisando y modificando el código, es imperativo poder entender los cambios que la otra persona ha realizado para no perder el valioso tiempo tratando de entender el contexto de la aplicación.

Referencias

[1] (2011, agosto 16). *Common Dialogs* [Online]. Disponible en https://wiki.wxwidgets.org/Writing_Your_First_Application-Common_Dialogs

[2] (2011, abril 23). *Drawing on a Panel with a DC* [Online]. Disponible en https://wiki.wxwidgets.org/Drawing_on_a_panel_with_a_DC

[3] (2017, febrero 11). *Adjacency Matrix Graph Implementation in C++* [Online]. Disponible en https://gist.github.com/arrayed/4121223743f60e2105cde3b83f26590a

[4] (2009, diciembre 12). *Parsing a comma-delimited std::string* [Online]. Disponible en https://stackoverflow.com/questions/1894886/parsing-a-comma-delimited-stdstring

[5] (2012, diciembre 15). *Determine coordinates for an object* [Online]. Disponible en https://stackoverflow.com/questions/13895237/determine-coordinates-for-an-object-at-a-specific-distance-and-angle-from-a-point