Instituto Tecnológico de Costa Rica - ITCR

Sede de Alajuela

Curso:
Lenguajes de programación IC4700

Profesora:
Samantha Ramijan Carmiol

I Semestre del 2021

Integrantes del grupo:

Jose Alexander Artavia Quesada	2015098028
Bryan Andrey Díaz Barrientos	2019264426
Josué Gerardo Gutiérrez Mora	2018300436

Diseño de la solución

Datos y estructuras utilizadas

Dada la naturaleza del lenguaje (es funcional) la utilización de variables establecidas para que funcionen a lo largo del código no fue implementada, sin embargo, cabe mencionar aquellos elementos que se pueden observar muy repetidamente a lo largo del código con el fin de orientar mejor en cómo es que está diseñado todo el algoritmo de este trabajo:

N: se usa para el tamaño de la matriz del laberinto,el tamaño del laberinto será de 6x5 por ejemplo.

Maze: es el laberinto del juego, el cual es una matriz de tamaño n*n (6x6 para efectos del trabajo elaborado en este caso), tiene como valores 1 si el camino es válido, un 0 si el espacio es un obstáculo. El espacio de llegada tiene un número 2, mientras que el camino recorrido se muestra con un 4.

list: A lo largo del algoritmo se podrá observar cómo se trabaja con demasiadas variaciones de listas, ya sea para crear matrices, modificar matrices, encontrar caminos, etc. de aquí que muchas funciones lleven “list” entre su nombre.

Dikjstra: Es sobre la estructura de datos que conlleva al algoritmo de Dijkstra que recae la responsabilidad del funcionamiento de este proyecto que es encontrar el camino más corto entre un punto de inicio de un laberinto y su meta final (en caso de que sí exista un camino que conduzca a ello). Hay que recordar en que se basa dicho algoritmo y qué mejor definición que la que brinda Melanie Sclar (2016):

> “El algoritmo calcula las distancias mínimas desde un nodo inicial a todos los demás. Para hacerlo, en cada paso se toma el nodo más cercano al inicial que aún no fue visitado (…) Luego, se recalcula todas los caminos mínimos (...) así, en cada paso tendremos un subconjunto de nodos que ya tienen calculada su mínima distancia y los demás tienen calculada su mínima distancia si solo puedo usar los nodos del conjunto como nodos intermedios

> (...) Con cada iteración agregaremos un nodo más a nuestro conjunto, hasta resolver el problema en su totalidad.

En pocas palabras, se evidencian 2 cosas para la realización de este proyecto:

- La utilización de grafos (se hace uso de una de las bibliotecas que brinda Racket para poder llevar a cabo su implementación).
- La necesidad de acotar dicho algoritmo al trabajo actual, ya que, si bien se dicho este algoritmo se encarga de encontrar rutas cortas, esto puede ser implementado en mapas, problemas matemáticos o en este caso, para un laberinto (que se puede ver cómo una versión un tanto más compleja de un mapa pero cabe mencionarlo en un punto aparte cómo un juego).

Descripción de los algoritmos utilizados

Dada la forma en que fue programado el proyecto, se pueden dividir sus funciones en 3 partes primordiales:

Matriz lógica

get: Obtiene un elemento de la lista

set: Mete un elemento en la matriz y retorna la matriz con ese elemento ya metido

make-maze: Función encarga de crear una matriz NxN (6x6 para el caso de este proyecto), esta función genera dicha matriz de manera binaria donde los 1's representan obstáculos en el laberinto y los 0's el camino que se puede recorrer, sólo funcionará para llevar a cabo la lógica del programa, de forma que es sobre esta matriz que se ejecuta la solución de los del proyecto.

list-with: Recibe una lista, una posición y el elemento a insertar dentro de esta.

set-element-matrix: Función encargada de editar los elementos de una lista, cambiando así los elementos de la lista inicial que se generó.

random-xy: Devuelve un número random entre entre 0 y 6 incluyendo a estos 2.

randomObs: Se encarga de colocar los obstáculos en la matriz, esto haciéndola de forma aleatoria, creando de esta manera un laberinto de 6x6 el cual puede incurrir en situaciones donde existan varios caminos para poder llegar o incluso no tener solución, en caso de ser así el propio sistema se encargará de decirlo al usuario.

Creación e implementación de grafos

graph-solve: Se encarga de crear un grafo único y sin dirección alguna.

add-list-to-graph: Le otorga a un grafo una lista.

related-graph: Se encarga de ver si los elementos del grafo están relacionados, hace uso de una función auxiliar llamada que lleva a cabo todo el proceso de verificación.

change-matrix-to-graph: Cómo su nombre lo indica, toma una matriz y la transforma en una ruta de grafos. Luego mediante **related-graph** los relaciona para mediante **get-in-list-solve-path** retornar la lista lista con la solución de cual es el camino más corto para llegar a la meta final.

Apartado gráfico dentro de una ventana

path-gui: Se usa para insertar mediante *bitmap%* la imagen del camino que se puede recorrer según la matriz lógica que se haya generado antes de iniciar a dibujar en la ventana. lo mismo ocurre con **path2-gui** sólo que dibuja el camino con otro tipo de imagen.

obs1-gui: Usado para la inserción de la imagen que representa un obstáculo, lo mismo ocurre con **obs2-gui.**

sol-gui: Inserta la imagen del camino que se va dibujando cuando Dijkstra devuelve el camino más corto.

get-x-start: Cuadro de inserción de texto donde se pondrá en que posición (en el eje x) se va iniciar en el laberinto. Lo mismo ocurre con **get-y-start** sólo que sucede en el eje y.

input-button: Dibuja un botón sobre la ventana, además, al presionarlo, manda a llamar la solución del laberinto (en caso de no insertar bien los datos de entrada, dispara un mensaje de error, lo mismo en caso de que el laberinto no tenga solución, todo esto se implementa en otra función denominada **status-message** encargada de dibujar los mensajes en la ventana).

input-save: de la mano con el elemento anterior, es la función encargada de realizar las validaciones de las entradas del usuario.

select-image: Es la función que se encarga de recorrer la matriz y ver, dependiendo de su contenido, si lo que se lee es camino o un obstáculo, dependiendo del caso, colocará la imagen adecuada para mostrar en el laberinto de la ventana gráfica.

my-canvas: Dibuja el espacio sobre el que se van a dibujar dentro de este las imágenes que representarán al laberinto.

draw-maze: Dibuja imagen a imagen las imágenes que se utilizarán para representar el laberinto.

update: Cuando se pide la solución del laberinto, esta función se dispara con la intención de ir redibujando el camino hasta encontrar la meta.

solve-maze-gui: Es la función que se manda a llamar una vez que los datos ingresados por el usuario son correctos, se encarga de hacer correr todo la estructura de grafos con el fin de que el algoritmo de Dijkstra cobre vida y así pueda resolver el laberinto, ya sea que esto termine en mostrar el camino desde el punto de inicio hasta la meta o devolver el mensaje de que no hay camino para llegar hasta dicha meta.

Lecciones aprendidas

- Descubrimiento de la opción *map* que proporciona el lenguaje Racket para ayudar así al rápido manejo de elementos en listas sobretodo cuando se debe de utilizar varias de estas simultáneamente.

- Utilización de un lenguaje completamente funcional, saliendo así del típico método de programación donde se va paso a paso solucionando las ideas con muchas variables, elementos globales, clases, objetos…

- Uso de la documentación de Racket, lo cual es bastante prometedor debido a que por lo general las investigaciones que se llevan a cabo para sopesar las ideas sobre un lenguaje se realizan a lo largo de internet, ya sea con tutoriales de YouTube o en alguna página web, pocas veces recurriendo a la documentación oficial, sin embargo, debido a la poca información y pocos ejemplos que se pueden encontrar sobre Racket pues la utilización de su documentación oficial fue altamente necesaria para entender muchos conceptos.

- Manejo de grafos y la librería que proporciona Racket para su uso, haciendo más factible y sencillo llegar a la solución.

- Implementación de una interfaz gráfica funcional, es decir, siendo capaz de cambiar de estado (redibujarse) cada vez que se va mostrando el camino a recorrer. Lo anterior era sencillo de hacer en la terminal, pero ejecutarlo en una ventana aparte fue todo un reto que terminó siendo nada más que un conjunto de pasos a seguir.

Bibliografía

1. McCarthy, J. (2021). 2013-04-15: A* Search in Racket. Retrieved 15 April 2021, from https://jeapostrophe.github.io/2013-04-15-astar-post.html

2. Miller, M. (2021). Learn Racket by Example: GUI Programming. Retrieved 18 April 2021, from https://blog.matthewdmiller.net/learn-racket-by-example-gui-programming

3. Sclar, M. (2021). Camino mínimo en grafos. Retrieved 25 April 2021, from http://www.oia.unsam.edu.ar/wp-content/uploads/2017/11/dijkstra-prim.pdf

4. Chang, S. (2021). Racket Generic Graph Library. Retrieved 25 April 2021, from https://docs.racket-lang.org/graph/index.html

5. Racket Documentation. (2021). Retrieved 25 April 2021, from https://docs.racket-lang.org/

6. Popa, B. (2021). racket/gui saves the day. Retrieved 25 April 2021, from https://defn.io/2019/06/17/racket-gui-saves/