Universidad de Santiago de Chile

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE INGENIERÍA EN INFORMÁTICA

TAREA 1: ALGORITMO A*

TALLER DE PROGRAMACIÓN

Sección: 13315-0-A-1

Profesor: Pablo Roman Asenjo

Ayudante: Ricardo Hasbun

Autor: John Serrano Carrasco

Abril 2022

Contents

1 El problema

Se dispone de un laberinto donde se tiene marcada una entrada y una salida. Se busca generar un programa que dado un laberinto encuentre un camino desde la entrada hasta la salida (secuencia de pasos) de la manera mas eficiente posible. En caso de que no encuentre solución debe indicarlo. Utilizando el generador se debe ejecutar varias veces el programa y medir el tiempo promedio utilizado. El laberinto se representa por una matriz de nxn n umeros enteros 0, 1, 2, 3. Donde 0 es un lugar vacío, 1 es una pared, 2 es la entrada y 3 es la salida. El programa a construir debe estar codificado en C++, implementando el algoritmo A* para resolver además de las estructuras de datos auxiliares. Dichas estructuras de datos auxiliares deben ser implementadas en base a arreglos. Los datos deben estar dispuestos de manera contigua en memoria. El programa principal debe calcular el tiempo promedio de varios laberintos aleatorios. Se entregan además uno o varios programas de test por clase.

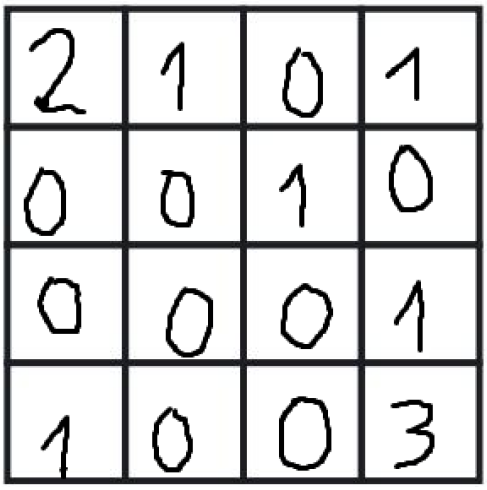

Figura 1: Un dibujo de un laberinto 4x4, representado por 0s, 1s, 2 y 3.

2 La solución

Para resolver este problema y crear el algoritmo A*, necesitamos cuatro clases importantes:

- Clase laberinto
- Clase Nodo
- Clase Heap
- Clase Container

La solución es la siguiente: Primero, se debe generar el laberinto. Por ordenes de la pauta, tenemos que la dimensión del laberinto es de 1000 x 1000 (esto puede ser modificado como se desee), con un porcentaje de vacio de 80. De esta forma, tendremos un laberinto generado gracias al metodo generate(). Luego, viene el metodo solve(). Primero se parte creando nuestra Heap, que será de tamaño (dimension * dimension). De esta forma, la Heap, la cual recordemos que corresponde a nuestra lista de nodos por visitar, tendra suficiente espacio en caso de cualquier camino. Lo mismo haremos nuestro container, lo cual inicialiamos con la dimension, ya que es una matriz.

Posterior a eso, crearemos dos nodos: El primero, llamado "current", será el nodo que tiene el menor valor. Inicialmente, corresponde al nodo de la entrada, cuyas coordenadas (i,j) corresponde a (0,0), su valor es 0 y al ser la entrada, es una raiz, por lo que no tiene padre (NULL). Luego dejamos creado otro nodo al cual llamamos "neighbor". Agregamos a current a la Heap, y se entra dentro de un ciclo. Mientras la heap, no este vacia, primero, sacaremos a Current de la Heap y utilizando nuestra matriz (Container) que corresponde a nuestra lista de ya visitados, marcamos el nodo correspondiente a las coordenadas, colocando un 1.
Una vez realizado eso, verificaremos si el nodo corresponde a la salida, pero al ser el comienzo del programa claramente no lo es, por lo que continuamos esta vez verificando los vecinos del nodo Current.

Para ello, debemos verificar si las coordenadas de sus vecinos son validas (es decir, que esten dentro del laberinto) y que correspondan a un camino o bien, a la salida. Una vez hecho eso y si se cumple la condición anterior, se calcula el valor del nodo, a traves de la distancia entre dos puntos utilizando las coordenadas del vecino y las coordenadas de la salida (dimension -1, dimension -1). Dejamos todo esto en el Nodo "Neightbor" y dejamos que su padre sea current. Una vez realizado eso, vamos a verificar si el nodo ya fue agregado a la closelist (si su posicion de la matriz es 1). Si es asi, simplemente ese vecino se ignora y se sigue con el siguiente vecino. Caso contrario, se agrega el vecino a la Heap. Se realizará el mismo proceso con los 4 posibles vecinos (Norte, Sur, Este, Oeste). Una vez tengamos listo los vecinos, el ciclo se repetirá, y gracias a que la Heap ordena automaticamente los nodos colocando en el top al elemento de menor valor, podemos ir moviendo de a poco el camino a recorrer. Finalmente, cuando se llegue a la salida, se aplicará un break para escapar del ciclo y se imprimirá la solución, el laberinto y el camino en color. Si no se encontró una solución, lo cual implica que la heap se vació ya que no habia camino por recorrer hasta la salida, entonces se imprimirá el laberinto junto con un mensaje de que no fue posible encontrar un camino.

En el main del programa, se hace un ciclo for con N repeticiones del laberinto para poder calcular el tiempo de resolución de los laberintos y el tiempo promedio.

El proceso de creación de la solución fue basada en lo enseñado en clases, como también en algunos tutoriales encontrados en la Web e informacion de paginas como GeeksforGeeks. Una mencion honrosa es al video "A* Pathfinding (E01: Algorithm explanation)" de Sebastian Lague en Youtube, el cual fue uno de los pocos tutoriales que explicaba la idea de utilizar los nodos de los padres para devolverse en el camino correcto, algo que al principio costó entender pero finalmente se logró y fue de gran ayuda.

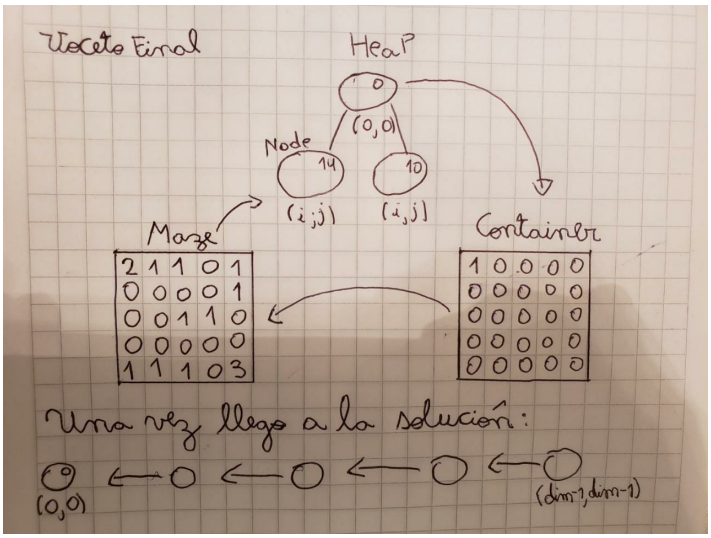

Figura 2: Boceto final de la solución

2.1 Pruebas y errores

Hubieron multiples pruebas y multiples errores durante el proceso de creación. Una de ellas fue que por alguna razón se repetian vecinos y se generaba un bucle infinito, por lo que se decidió implementar una busqueda para la Heap (este metodo aún esta en el archivo). Sin embargo, esto aumentaba considerablemente la complejidad del algoritmo, y el uso de memoria, ya que con laberintos muy grandes habian casos donde se quedaba infinitamente buscando en la Heap. Por lo tanto, se decidio mejorar la Heuristica y las verificaciones del camino a recorrer. Una vez que se arreglaron esos problemas, se eliminó la necesidad de buscar en la Heap.

Otro problema, fue que inicialmente, se tenia planeado utilizar un Arreglo de tamaño 10000 para la closelist, pero nuevamente la busqueda implicaba complejidad y memoria. Sin embargo, considerando una de los conocimientos aprendidos en el curso de Estructura de Datos, se decidio replicar uno de los metodos para "marcar" posiciones, la cual consiste en utilizar una matriz de 0s y simplemente ir accediendo a una posicion especifico para la verificación, lo cual conlleva coste de O(1) y no implica grandes tiempos de espera.

Con ambas implementaciones mencionadas anteriormente, el tiempo de espera para un laberinto 1000 x 1000, era de aproximadamente 35-36 segundos. Ahora, con las mejoras implementadas, resolver un laberinto 1000 x 1000 puede llevar a lo mas 3 segundos, pero usualmente toma menos de 1 segundo.

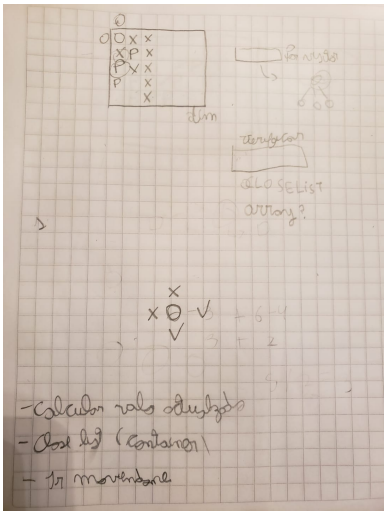

Figura 3:Ideas durante el proceso de ideación de la solución

3 Complejidad y eficiencia

A pesar de que hubieron muchos problemas, pruebas y errores con el proceso de creación, resulta que si se tiene un Algoritmo bastante eficiente, ya que es capaz de recorrer laberintos de 1000 x 1000 en menos de 1 segundo y puede terminar de recorrer 100 laberintos de 100 x 100 en menos de 0,5 segundos, lo cual demuestra de que se tiene una complejidad bastante buena. Si ignoramos las creaciones del laberinto y de la closeList, es probable que tengamos una complejidad de O(Log(n)), gracias a que procesos que tomaban complejidad de O(n) fueron reemplazados por procesos de O(1), como por ejemplo, la busqueda en la closelist. Si hacemos pruebas de 100 de 1000 x 1000, puede demorarse desde 13- 27 segundos, lo cual de todas formas es bastante eficiente, especialmente si consideramos que el estandar por laberinto es de 10 - 25 segundos, lo cual implicaria que en terminar los 100 de 1000 x 1000 podria demorar desde 16,6 minutos hasta 41,6 minutos, por lo que claramente se superó el estandar.

4 Conclusiones

Finalmente, se logró completar la tarea cumpliendo los objetivos propuestos, donde se tiene un algoritmo que es capaz de resolver un gran número de laberintos de diferentes dimensiones, encontrando el camino mas eficiente si es que este existe.

A pesar de las complicaciones, se logró programar la tarea en C++ aprendiendo nuevas cosas en el proceso y se espera que este conocimiento sirva de experiencia para las futuras entregas de la Asignatura.

5 ANEXOS

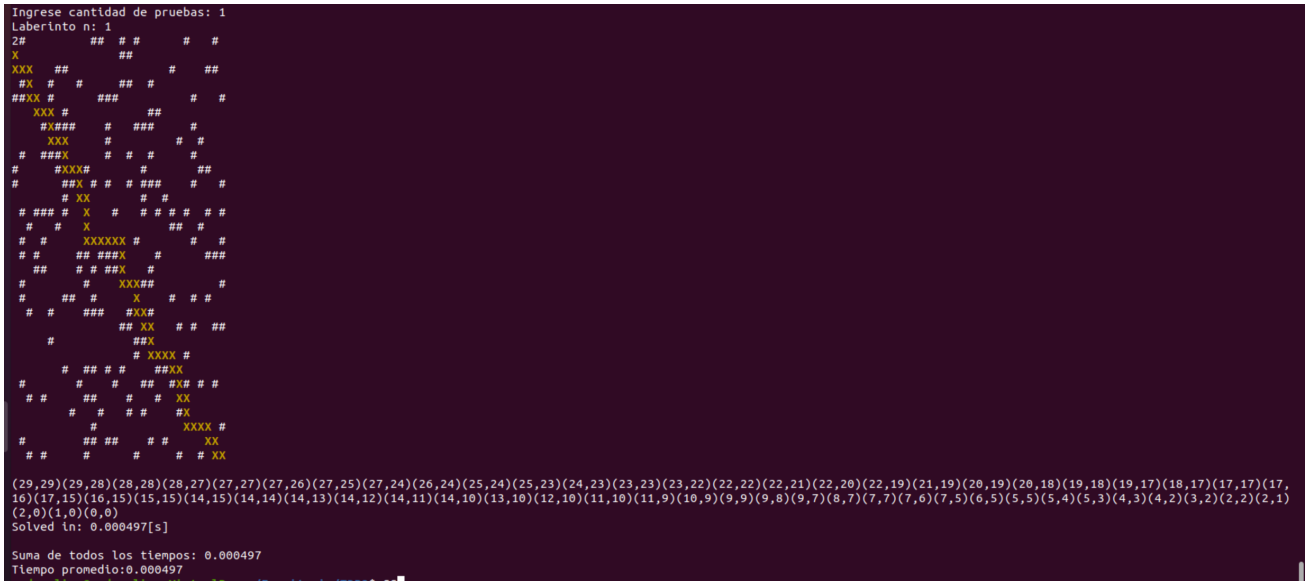

Figura 4: Solución de laberinto de 30 x 30

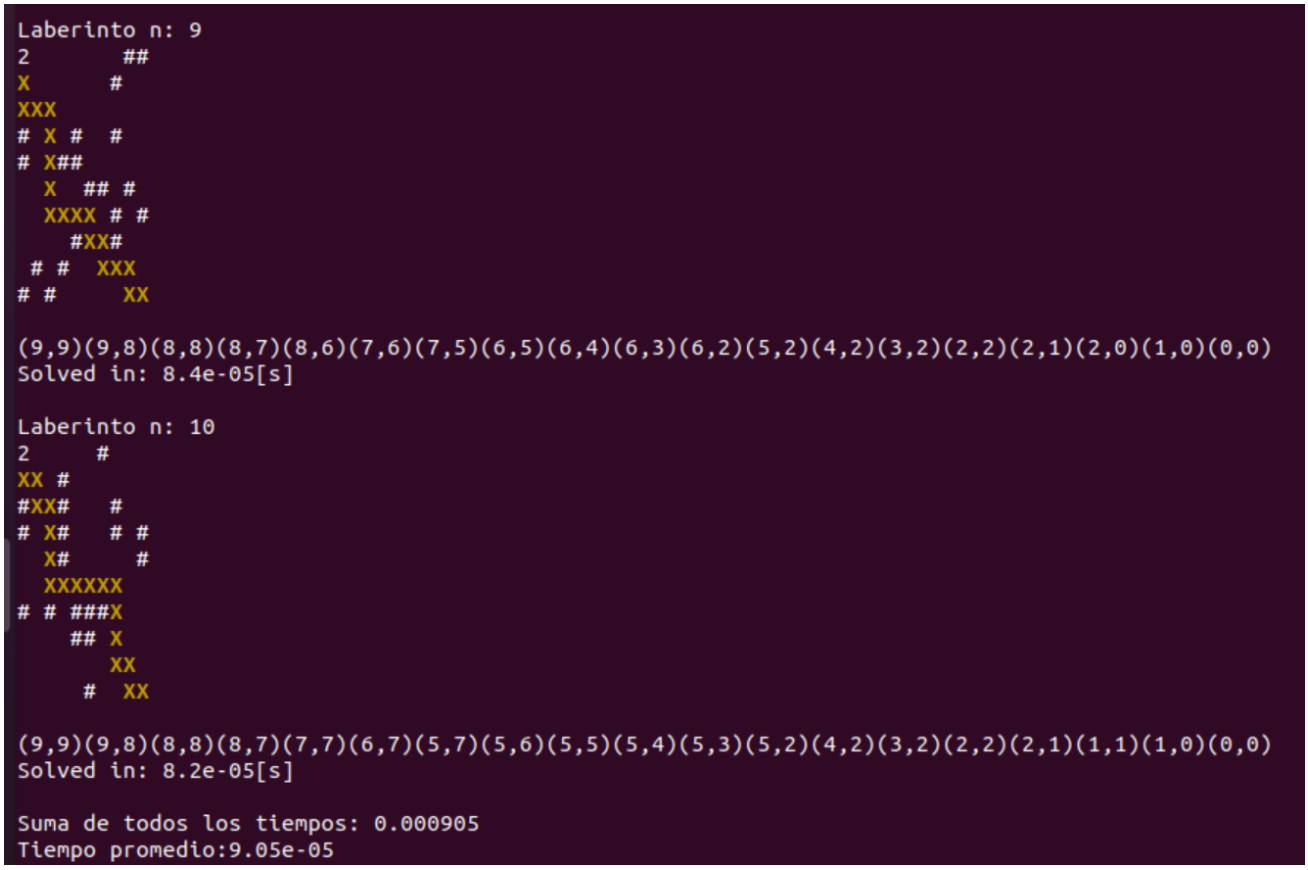

Figura 5: Solución de laberinto 2 laberintos de 10 x 10

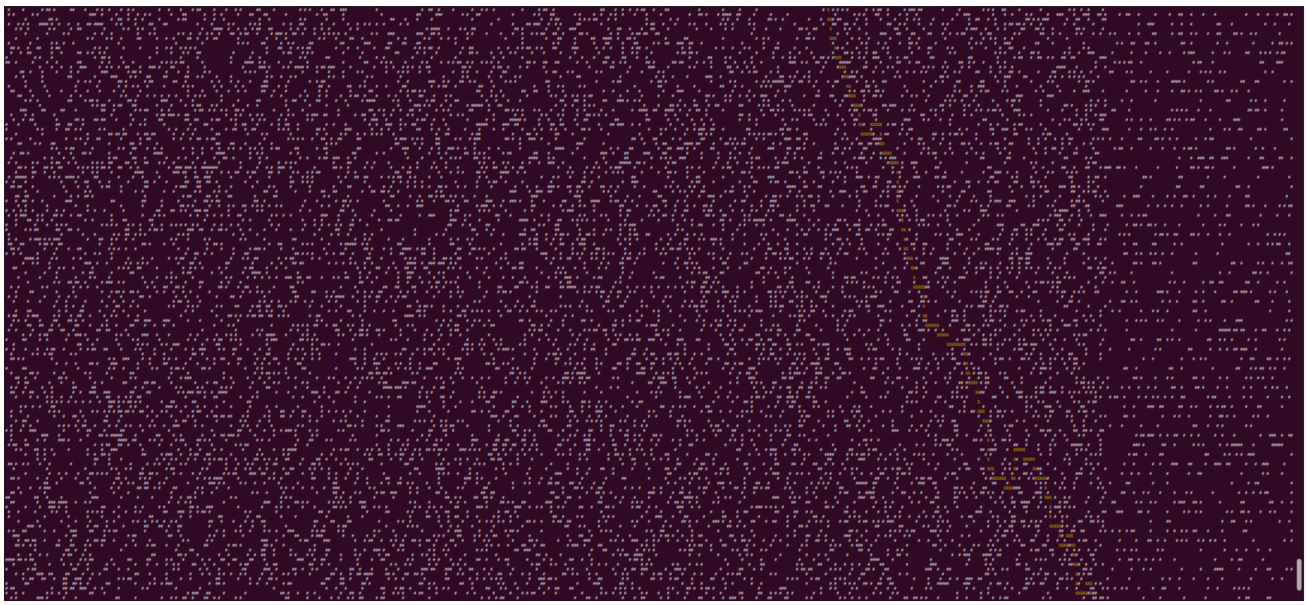

Figura 6: Solución de laberinto 1000 x 1000 (Es tan grande que no cabe todo en la pantalla)