Estructuras de Datos y Algoritmos (EDA)
Tarea 2: Laberinto

Profesor: José Manuel Saavedra R.

Integrantes: Tomás Rodríguez A.

Manuel Tagle A.

2 de octubre de 2023

Resumen

En el siguiente informe se revisarán distintos tipos de algoritmos de búsqueda y luego serán comparados con respecto a su tiempo de ejecución. Se mostrará el código de cada uno de los algoritmos. Se estudiaron dos algoritmos de búsqueda diferentes, DFS y BFS. El algoritmo más eficiente fue DFS basado en la ADT Stack, el uso de laberintos de prueba con dimensiones pequeñas ($11x11$, $21x21$, $31x31$ y $41x41$) lo sitúan como ganador. Los objetivos planteados en la introducción fueron logrados satisfactoriamente.

Índice

1. Introducción

Al momento de llevar al mundo cotidiano la aplicación de distintos algoritmos de búsqueda de datos, pueden ser distintos los beneficios que cada uno conlleve en relación con lo que se quiere lograr. Si es que se quiere analizar una situación para poder aplicar uno de estos métodos de búsqueda, primero se tiene que poder estudiar el caso, reduciendo su abstracción hasta tal punto de lograr analizar el caso en su forma más simple.
Siguiendo esta línea de pensamiento podemos estudiar el caso de un laberinto. Este consta de una estructura rectangular la cual está compuesta de muros y caminos, habiendo una (o más) solución posible, solución la cual consiste en poder llegar, ininterrumpidamente, de inicio a fin.
Como convención de este estudio, los caminos y muros de un laberinto se podrán ver representados como ceros (0) y unos (1), para así poder trabajar de mejor manera con este al momento de buscar una solución implementada en código escrito en C++.

En este estudio, se analizará la solución de un laberinto a partir del uso de dos métodos distintos, los cuales se basan en la utilización de dos estructuras de datos abstractas (*ADT*'s por sus siglas en inglés) distintas; pila (*stack*), y cola (*queue*).

1.1. ADT's utilizadas

1.1.1. Stack

Un stack es una estructura de tipo LIFO (Last-In, First-Out). Se puede entender como una pila de libros, de la cual si queremos sacar alguno, se deben retirar todos los que estén sobre este.

1.1.2. Queue

Un queue, a diferencia del stack, es una estructura de tipo FIFO (First-In, First-Out). Se puede entender como la fila de una tienda, ya que la gente es atendida por orden de llegada a la fila; se tiene que esperar a que atiendan a todos los que están por delante de uno para poder ser atendido.

1.2. Algoritmos de búsqueda

1.2.1. DFS

DFS (Depth First Search) es un algoritmo que implementa el uso de *stacks*. Imaginando que tenemos una ramificación, lo que hace DFS es simplemente ir rama por rama, verificando si se llegó al target. Se trabaja por la profundidad de la estructura bajo estudio.

1.2.2. BFS

A su vez, BFS (Breadth First Search) utiliza *queues*. Imaginando la misma estructura anterior (ramificaciones), va buscando el target por niveles, es decir por todas las ramas posibles a la par. Se trabaja por el ancho de la estructura estudiada.

1.3. Objetivos principales

- Aplicar las ADT's especificadas para resolver problemas cotidianos.
- Aprender los distintos algoritmos de búsqueda derivados de las ADT's estudiadas (DFS y BFS).

1.4. Objetivos generales

- Comprender el funcionamiento de las ADT's estudiadas.
- Implementar de manera ordenada un algoritmo de búsqueda en C++.

2. Desarrollo

La implementación de los algoritmos fue realizada en el lenguaje C++. También se tomó inspiración del repositorio del profesor, el cual contaba con el desarrollo del generador de laberintos y los ADT. Los códigos de las estructuras mencionadas (*queue* y *stack*) se pueden encontrar en el libro guía del curso *"Estructura de Datos y Algoritmos"* (Saavedra y Chang 2022)

2.1. solve_pila

Para el desarrollo de *solve_pila* se inspiró en el algoritmo de búsqueda mencionado anteriormente, DFS.

2.1.1. Depth First Search

DFS es un algoritmo de búsqueda ampliamente utilizado para estructuras ramificadas, como laberintos o árboles. Consiste en un buscar por caminos, es decir, revisa cada rama hasta llegar al final de esta, si no se encuentra el target en dicha rama revisa la rama adyacente.

Listing 1: solve_pila

```
1         while (!finished) {
2                 if (getBox(i,j)){
3                         stackX.push(i);
4                         stackY.push(j);
5                         options = Split(i, j);
6
7                         if (options == 0) {
8                                 Return(i, j, stackX, stackY, stack_splitX.
                                     top() -> getData(), stack_splitY.top()
                                     -> getData());
9                                 stack_splitX.pop();
10                                stack_splitY.pop();
11                                setWall(iAux, jAux, 5);
12                        }
13
14                        if (options > 1) {
15                                iAux = i;
16                                jAux = j;
17                                stack_splitX.push(i);
18                                stack_splitY.push(j);
19                        }
20
21                        shuffle(i, j);
22
23                        if(i == i1 && j == j1) {
24                                finished = true;
25                                setWall(i, j, 3);
26                        }
27                }
28        }
```

2.2. solve_cola

A su vez, para desarrollar *solve_cola*, se utilizó el algoritmo BFS

2.2.1. Breath First Search

BFS es un algoritmo de búsqueda ampliamente utilizado para estructuras ramificadas, como laberintos o árboles. Consiste en un buscar por niveles en vez de ramas, símil al movimiento de un líquido a través de ramificaciones.

Listing 2: solve_cola

```
1  int** Maze::solveQueue(int i0, int j0, int i1, int j1) {
2          eda::Queue queueX;
3          eda::Queue queueY;
4          bool finished = false;
5          int i = i0;
6          int j = j0;
7          int iAux;
8          int jAux;
9          int options;
10         int** arr = new int*[YMAX];
11
12         if(grid[i0][j0] == 1 || grid[i1][j1] == 1) {
13                 std::cout << "No␣hay␣camino␣posible..." << std::endl;
14                 exit(0);
15         }
16
17         queueX.push(i);
18         queueY.push(j);
19
20         while (!finished) {
21                 options = Split(i, j);
22
23                 if (options > 1) {
24                         iAux = i; jAux = j;
```

```
25                          queueX.push(i);
26                          queueY.push(j);
27                          shuffle(i, j);
28                          queueX.push(i);
29                          queueY.push(j);
30                          i = iAux; j = jAux;
31                  }
32                  shuffle(i, j);
33
34                  queueX.push(i);
35                  queueY.push(j);
36
37                  queueX.pop();
38                  queueY.pop();
39
40                  i = queueX.top()->getData();
41                  j = queueY.top()->getData();
42
43                  if (i == i1 && j == j1) {
44                          finished = true;
45                          setWall(i, j, 3);
46                  }
47          }
48  }
```

3. Resultados y discusiones

Para poder analizar y contrastar empíricamente las diferencias entre la aplicación de *stacks* y *queues* en problemas cotidianos, a favor de mantener coherencia con los objetivos planteados, se midió el tiempo, en milisegundos (ms), que tardó cada ADT en resolver un mismo laberinto de dimensiones *nxn*, dando la posibilidad de llevar a cabo un análisis más significativo sobre los resultados obtenidos.

Las pruebas se realizaron en la máquina virtual del curso (*Ubuntu*), y las características de la computadora en la cual se realizó el trabajo se encuentran en detalle más adelante (Subsección 3.3).

3.1. Resultados

A continuación se presentan los resultados obtenidos en una tabla, contrastando los tiempos de ejecución con respecto a la dimensión del laberinto resuelto.

Tabla 1: Tiempos de ejecución v/s Dimensión.

Dimensiones	**Tiempo de Ejecución (ms)**	
	Stack	**Queue**
11x11	0.0286898	0.0746388
21x21	0.092572	0.827569
31x31	0.184296	3.33926
41x41	0.312642	13.4296

Fuente: Elaboración propia.

A partir de los resultados expuestos en la tabla anterior (Tabla 1), se puede obtener más información sobre qué es lo que estos representan. Esto se logra obteniendo la diferencia porcentual de rendimiento entre ambas estructuras de datos, en este caso analizando cuanto más se demora una solución utilizando *queues* en lugar de *stacks*, con una muestra de 100 pruebas con laberintos distintos por cada dimensión, resultando en la siguiente tabla.

Tabla 2: Porcentaje de Diferencia Queue sobre Stack.

Dimensiones	Diferencia (%)
11x11	%160
21x21	%794
31x31	%1.712
41x41	%4.195

Fuente: Elaboración propia.

3.2. Discusiones

Después de haberle dado una interpretación más comparativa a los distintos rendimientos que se presentaron, se pueden observar distintas cosas.

En primer lugar, se puede ver en la Tabla 1 cómo la utilización de *stacks* para resolver un laberinto resulta en un tiempo de ejecución de nivel bajo en todas las pruebas hechas, pudiendo incluso asimilarse a un crecimiento lineal, el cual es bastante óptimo. A diferencia de la aplicación de *queues*, cuyo comportamiento se ve caracterizado por el crecimiento notorio de su tiempo de ejecución, demostrando que este se podría decir que es de carácter cuadrático, o incluso exponencial. Esto se puede ver reflejado de manera más clara en la Tabla 2, donde basta ver cómo, y a que ritmo, va aumentando la diferencia porcentual con respecto al tiempo de ejecución de la solución de un laberinto que implementa el uso de *queues*, en comparación a la que usa *stacks*, dando cuenta a la gran ventaja en temas de rendimiento que la segunda demuestra.

Es importante destacar que dichos tiempos de ejecución están sujetos al desarrollo de los algoritmos de búsqueda (Subsección 1.2) que se implementaron en esta experiencia, ya que se pueden hacer ciertas modificaciones que alteren el comportamiento de estos, resultando, posiblemente, en tiempos de ejecución distintos. **Por lo que no se deben tomar como referencias extrapolables los resultados expuestos en el presente informe**

3.3. Características

En esta sección del informe se revisarán las características de la computadora en la cual se realizó la ejecución de los algoritmos de búsqueda para resolver los laberintos. Cabe mencionar que, al haberse realizado las pruebas dentro de una máquina virtual con recursos limitados, es muy probable que esto haya afectado en el rendimiento de los distintos algoritmos.
Las características del dispositivo que son de importancia para considerar sobre el tiempo de ejecución son:

- **Procesador:** Intel(R) Core(TM) i7-1065G7 CPU @ 1.30 GHz
- **# Núcleos:** 4
- **RAM:** 12 GB

Mientras que, de los recién mencionados, estos son los recursos que la máquina virtual tuvo disponible al momento de realizar las pruebas:

- **Procesador:** Intel(R) Core(TM) i7-1065G7 CPU @ 1.30 GHz
- **# Núcleos:** 2
- **RAM:** 6 GB

4. Conclusiones

A modo de cierre, luego de haber introducido como objetivos principales tanto reconocer los distintos tipos de búsquedas, como compararlos entre sí para determinar la eficacia de estos, y una vez analizados los distintos tiempos de ejecución, se puede concluir que existe cierta jerarquía al momento de estudiar el rendimiento de los algoritmos expuestos en este trabajo.

Se pudo determinar que el DFS es mucho más eficiente con laberintos de pequeñas dimensiones. A su vez, el algoritmo con cola presentó problemas de rendimiento, pues el entorno y las características del computador no fueron las óptimas.

Una vez analizado el tiempo de ejecución de los dos algoritmos previamente mencionados, y expuestos en la Tabla, se puede determinar que el algoritmo que usa stack es el más eficiente con respecto a su tiempo de ejecución.

5. Referencias bibliográficas

Referencias

Saavedra, José M. y Violeta Chang (2022). *Estructura de Datos y Algoritmos*. Primera edición. Universidad de los Andes.