Laporan Tugas Besar 2 IF2211 Strategi Algoritma
Semester II Tahun 2022/2023

Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan *Maze Treasure Hunt*

Disusun oleh:

Azmi Hasna Zahrani	13521006
Haikal Ardzi Shofiyyurrohman	13521012
Ditra Rizqa Amadia	13521019

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2023

DAFTAR ISI

DAFTAR ISI **1**
BAB I **2**
DESKRIPSI TUGAS **2**
BAB II **3**
LANDASAN TEORI **3**
2.1. Dasar Teori 3
2.1.1. Graph Traversal 3
2.1.2. Algoritma Breadth First Search 3
2.1.3. Algoritma Depth First Search 4
2.4. C# Desktop Application Development 6
BAB III **7**
APLIKASI STRATEGI BFS DAN DFS **7**
3.1. Langkah-langkah Pemecahan Masalah 7
3.2. Proses Mapping Persoalan Menjadi Elemen-Elemen Algoritma Breadth First Search dan Depth First Search 7
3.3. Contoh Ilustrasi Kasus Lain 8
BAB IV **9**
ANALISIS PEMECAHAN MASALAH **9**
4.1. Implementasi Program 9
4.2. Struktur Kelas yang digunakan 15
4.3. Tata Cara Penggunaan Program 15
4.4. Hasil Pengujian 15
4.5. Analisis Desain Solusi Algoritma BFS dan DFS 18
BAB V **20**
KESIMPULAN DAN SARAN **20**
5.1. Kesimpulan 20
5.2. Saran 20
5.3. Refleksi 20
5.4. Tanggapan Kelompok 20
DAFTAR PUSTAKA **21**
LAMPIRAN **22**

BAB I
DESKRIPSI TUGAS

Tugas Besar 2 IF2211 Strategi Algoritma meminta kelompok untuk membuat sebuah aplikasi *Treasure Hunt Solver* dengan GUI sederhana yang mengimplementasikan algoritma BFS dan DFS untuk mendapatkan rute memperoleh seluruh harta karun yang ada. Program dapat menerima dan membaca input sebuah file berformat txt yang berisi *maz*e dan akan ditemukan solusi rute dari *maze* tersebut untuk mendapat harta karunnya. Batasan input *maze* berbentuk segi empat dengan simbol K sebagai titik awal, T sebagai harta karun, R sebagai lintasan yang dapat diakses, dan X sebagai halangan yang tidak dapat diakses.

Algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS) dimanfaatkan untuk menelusuri lintasan yang mungkin dikunjungi hingga ditemukan rute solusi (rute yang memperoleh seluruh harta karun pada *maze*), baik secara melebar maupun mendalam bergantung alternatif algoritma yang dipilih. Rute yang diperoleh dengan kedua algoritma dapat berbeda dengan pergerakan secara *left right up down* tanpa pergerakan secara diagonal. Kelompok juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid *maze* serta hasil pencarian rute solusinya.

Daftar input *maze* dikemas dalam sebuah folder yang dinamakan *test* dan terkandung dalam *repository* program. Folder tersebut setara kedudukannya dengan folder scr dan doc. Cara input *maze* dapat input file maupun dengan *textfield* sehingga pengguna dapat mengetik nama *maze* yang diinginkan.

Setelah program melakukan pembacaan input, program akan memvisualisasikan grid terlebih dahulu tanpa pemberian rute solusi. Kemudian, program menyediakan tombol *solve* atau *search* untuk mengeksekusi algoritma BFS maupun DFS. Setelah tombol tersebut diklik, program akan melakukan pemberian warna pada rute solusi.

Tugas Besar 2 IF2211 Strategi Algoritma dikerjakan secara berkelompok dengan menggunakan bahasa pemrograman C#.

BAB II
LANDASAN TEORI

2.1. Dasar Teori

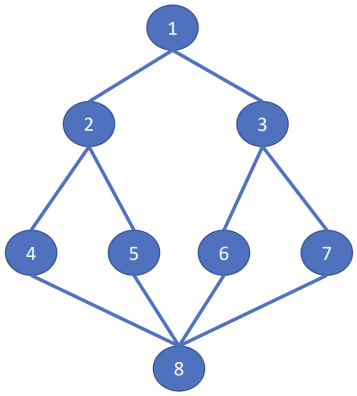
2.1.1. *Graph Traversal*

Graf merupakan struktur data yang digunakan untuk merepresentasikan objek-objek diskrit serta hubungan antara objek-objek tersebut. Graf terdiri atas *vertices* atau simpul dan *edge* atau sisi yang menghubungkan simpul-simpul tersebut. Graf dapat digolongkan menjadi graf sederhana dan tidak sederhana berdasar ada dan tidaknya gelang pada graf tersebut. Graf tidak sederhana dapat dibedakan menjadi dua, yaitu graf ganda (graf yang memiliki sisi ganda) dan graf semu (graf yang memiliki gelang). Sedangkan berdasar orientasi terhadap sisi, graf dapat dibedakan menjadi dua, yaitu graf berarah dan tidak berarah.

Graf banyak dimanfaatkan untuk menyelesaikan beberapa persoalan, seperti *shortest path,* pedagang keliling, tukang pos, dan lain sebagainya. Persoalan pada graf tersebut dapat diselesaikan dengan *graph traversal. Graph traversal* berarti mengunjungi simpul-simpul tepat satu kali secara sistematik yang algoritmanya terbagi menjadi dua, yaitu pencarian melebar atau *Breadth First Search* dan pencarian mendalam atau *Depth First Search.* Dalam proses pencarian solusi, terdapat dua pendekatan yang dapat digunakan, yaitu graf statis dan graf dinamis. Graf statis yaitu graf yang sudah terbentuk sebelum proses pencarian dilakukan. Sedangkan graf dinamis merupakan graf yang terbentuk ketika proses pencarian dilakukan.

2.1.2. Algoritma *Breadth First Search*

Pencarian melebar atau *Breadth First Search* merupakan algoritma traversal yang digunakan untuk melintasi atau mencari semua simpul atau node dari suatu graf. Pencarian dimulai dengan mengunjungi simpul awal lalu mengunjungi semua simpul yang bertetangga dengan simpul awal tersebut. Kemudian, mengunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul yang telah dikunjungi, begitu seterusnya. Misalnya pada contoh graf statis berikut.



Gambar 2.1.2.1. Ilustrasi Graf

Sumber: Bahan Kuliah IF2211 Strategi Algoritma: Breadth First Search dan Depth First Search.

Pada graf tersebut, algoritma *Breadth First Search* akan mengunjungi simpul dengan urutan 1, 2, 3, 4, 5, 6, 7, 8 dengan penjelasan iterasi seperti berikut.

Iterasi	V	Q	dikunjungi							
			1	2	3	4	5	6	7	8
Inisialisasi	1	{1}	T	F	F	F	F	F	F	F
Iterasi 1	1	{2,3}	T	T	T	F	F	F	F	F
Iterasi 2	2	{3,4,5}	T	T	T	T	T	F	F	F
Iterasi 3	3	{4,5,6,7}	T	T	T	T	T	T	T	F
Iterasi 4	4	{5,6,7,8}	T	T	T	T	T	T	T	T
Iterasi 5	5	{6,7,8}	T	T	T	T	T	T	T	T
Iterasi 6	6	{7,8}	T	T	T	T	T	T	T	T
Iterasi 7	7	{8}	T	T	T	T	T	T	T	T
Iterasi 8	8	{}	T	T	T	T	T	T	T	T

Tabel 2.1.2.2. Penjelasan Iterasi *Breadth First Search*
Sumber: Bahan Kuliah IF2211 Strategi Algoritma: Breadth First Search dan Depth First Search.

Pada tabel tersebut, di kolom Q terdapat daftar simpul yang bertetangga dan akan dikunjungi. Algoritma BFS menggunakan konsep *queue* untuk mengakses daftar tersebut dengan prinsip First-In-First-Out (FIFO), yaitu data yang pertama kali dimasukkan akan diproses terlebih dahulu. Sebagai contoh pada iterasi 1, simpul yang dikunjungi adalah simpul 1 yang bertetangga dengan simpul 2 dan 3. Simpul 2 dan 3 masuk ke dalam *queue*, kemudian pada iterasi 2, simpul 2 akan diproses (dikunjungi) dan tetangga dari simpul 2, yaitu simpul 4 dan 5 akan masuk ke queue begitu seterusnya.

Algoritma BFS memiliki beberapa kelebihan, antara lain apabila terdapat solusi maka solusi tersebut pasti akan ditemukan algoritma BFS, tidak akan terjebak di jalur buntu, serta akan menemukan solusi minimal. Sedangkan, kekurangan algoritma BFS adalah kendala memori karena algoritma BFS menyimpan semua node dari level saat ini untuk melanjutkan ke level berikutnya serta apabila solusi jauh akan memerlukan waktu yang lama.

2.1.3. Algoritma *Depth First Search*

Pencarian mendalam atau *Depth First Search* merupakan suatu metode pencarian pada sebuah graf dengan menelusuri satu cabang sebuah graf sampai menemukan solusi. Algoritma ini memiliki prioritas untuk mengunjungi simpul sampai level terdalam

terlebih dahulu. Kemudian, apabila ditemukan jalan buntu, algoritma akan memeriksa simpul sebelumnya yang sudah dikunjungi dan masih bertetangga dengan simpul lain yang belum dikunjungi dan menelusuri simpul tersebut. Sebagai contoh, pada gambar 2.1. Algoritma DFS akan mengunjungi simpul dengan urutan 1, 2, 4, 8, 5, 3, 6, 7 dengan penjelasan sebagai berikut.

E	Simpul Hidup
1	2_1, 3_1
2_1	4_{12},5_{12}, 3_1
4_{12}	8_{124},5_{12}, 3_1
8_{124}	5_{12}, 3_1
5_{12}	3_1
3_1	6_{13},7_{13}
6_{13}	7_{13}
7_{13}	Solusi ditemukan.

Tabel 2.1.3.1 Penjelasan Algoritma DFS

Pada tabel tersebut, simpul hidup merupakan simpul tetangga yang dimiliki oleh simpul yang sedang dikunjungi (E). Berbeda dengan BFS, pada algoritma DFS menggunakan konsep *stack* yang memiliki prinsip Last-In-First-Out. Contohnya pada simpul 1, memiliki tetangga 2 dan 3. Simpul 3 dimasukkan ke dalam stack terlebih dahulu dengan disusul oleh simpul 2. Kemudian, simpul 2 keluar terlebih dahulu (diakses lebih dulu) daripada simpul 3 dan simpul 2 memiliki tetangga yaitu simpul 4 dan 5 yang akan masuk ke dalam stack tersebut secara berurutan. Simpul 4 akan diproses lebih dahulu karena berada di paling luar, begitu seterusnya sampai solusi ditemukan.

Beberapa kelebihan algoritma *Depth First Search* antara lain, dapat menemukan solusi tanpa memeriksa banyak ruang pencarian sama sekali, jauh lebih efisien untuk ruang pencarian dengan banyak cabang karena tidak perlu menelusuri semua cabang yang ada, serta memerlukan memori yang relatif kecil karena node pada lintasan yang aktif saja yang disimpan. Sedangkan, kekurangan algoritma ini adalah sebagai berikut memungkinkan tidak ditemukannya tujuan yang diharapkan dan hanya akan

mendapatkan satu solusi pada setiap pencarian, kemungkinan ditemukan solusi tidak optimal, serta kemungkinan terjebak di jalur pencarian.

2.4. C# *Desktop Application Development*

Pengembangan aplikasi desktop menggunakan teknologi C# merupakan *tools* yang populer di kalangan pengembang aplikasi. C# merupakan salah satu bahasa pemrograman yang digunakan untuk pengembangan aplikasi. Framework yang kerap kali digunakan untuk pengembangan aplikasi desktop menggunakan C# adalah .NET serta Visual Studio. Alasan digunakannya *framework* tersebut adalah karena mudahnya pengembangan aplikasi baik secara desain dan fungsionalitas, kemudahan pengintegrasian dengan aplikasi yang sudah ada, serta kecepatan dan efisiensi proses pengembangan aplikasi.

BAB III
APLIKASI STRATEGI BFS DAN DFS

3.1. Langkah-langkah Pemecahan Masalah

Dalam menyelesaikan tugas besar ini, kelompok melakukan analisis permasalahan sebagai berikut.

1. Membuat *source* program dengan konsep Object-Oriented. Source program terdiri atas program maze yang menginisialisasi maze dengan membaca input dalam bentuk txt, program BFS yang membuat solusi penyelesaian menggunakan algoritma BFS beserta struktur queue, serta program DFS yang membuat solusi penyelesaian menggunakan algoritma DFS beserta struktur stack. Selain itu, terdapat program Route untuk menyimpan list rute yang dilalui, Node untuk menyimpan node yang ada pada rute dalam bentuk array, serta program tambahan yang ditampilkan pada GUI seperti Algorithm berisi penjelasan BFS dan DFS dan Solver yang digunakan untuk memanggil algoritma yang dipilih.
2. Membuat GUI dalam bentuk Windows Form App yang dapat menerima input *maze,* algoritma yang dipilih, serta *maze solver.* Selain itu, GUI akan menampilkan solusi dari *treasure maze,* rute, nodes, steps, dan waktu eksekusi program.
3. Mengintegrasikan GUI dengan program utama. Program perlu diintegrasi agar *source* program dapat tervisualisasi dalam GUI.

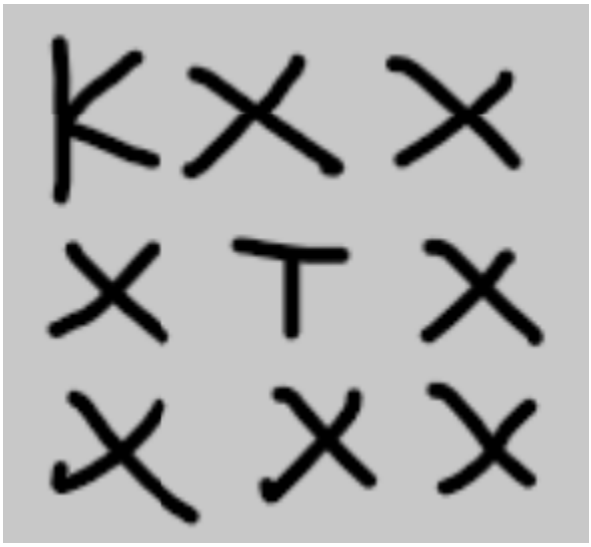
3.2. Proses *Mapping* Persoalan Menjadi Elemen-Elemen Algoritma *Breadth First Search* dan *Depth First Search*

Program *Treasure Hunt Solver* meminta inputan berupa *maze* dengan ukuran segi empat yang memiliki objek T *(treasure)*, K *(krusty krab/titik awal)*, R *(grid)*, dan X *(obstacle grid)* seperti pada deskripsi tugas. *Maze* tersebut dapat ditelusuri gridnya (digambarkan sebagai graf) yang memiliki node-node dengan awal K dan *final* atau *goal* T. Setelah mapping *maze* menjadi graf, *maze* dapat diselesaikan menggunakan algoritma BFS dan DFS.

Node pertama yang pasti dikunjungi dan berperan sebagai awal adalah K, yang kemudian dapat dilakukan aksi berupa arah dengan prioritas LURD *(Left, Up, Right, Down*) sebagai langkah yang akan diambil selanjutnya menuju node tetangga. Pada algoritma BFS digunakan konsep *queue*, yaitu ketika mengunjungi suatu grid, maka akan memeriksa grid tetangganya dan memasukkannya ke dalam *queue*. Kemudian, langkah ke grid selanjutnya akan mengikuti konsep *queue,* yaitu *First-In-First-Out.* Sedangkan, pada algoritma DFS digunakan konsep *stack*, yaitu ketika mengunjungi sebuah grid, maka grid tetangganya akan di-*push* ke dalam *stack* dan langkah ke grid selanjutnya akan mengikuti konsep *stack*, yaitu *Last-In-First-Out.* Ketika telah mencapai ujung grid yang tidak memiliki tetangga yang dapat dikunjungi, maka akan dilakukan *backtrack* ke grid sebelumnya hingga ditemukan grid yang belum dikunjungi dan meneruskan langkah hingga mencapai *goa*l atau *final.*

3.3. Contoh Ilustrasi Kasus Lain

Untuk contoh kasus lain, adalah pada saat terdapat satu atau lebih *treasure* yang tidak dapat dikunjungi karena terhalang oleh *obstacle grid* sehingga tidak bisa mendapatkan semua *treasure* yang ada.



Gambar 3.3.1: Contoh kasus *treasure* tidak bisa didapatkan.
Sumber: Dokumen Pribadi

Dari gambar 3.3.1 dapat dilihat bahwa tidak mungkin untuk mendapatkan treasure tersebut karena sekitar *treasure* tersebut tidak ada *path grid* yang dapat dilewati. Namun, kasus ini telah diatasi dengan exception dari *standard template library C#* dengan menggunakan *library Queue* dan *Stack C#* dengan kasus jika pada saat men-*Dequeue*(pada *Queue*)/ mem-*pop*(pada *Stack*) akan memberikan *exception*.

BAB IV
ANALISIS PEMECAHAN MASALAH

4.1. Implementasi Program

```
Algoritma Breadth First Search
untuk Treasure Hunt Solver
Kelompok a-maze-ing

namespace Amazeing
{
    public class Bfs : Algorithm
    {
        // ATTRIBUTES
        private Queue<Node> nodeQueue;

        // CONSTRUCTOR
        public Bfs() : base("BFS") { }

        // METHODS
        public override Solution Use(Solution solution)
        {
            this.nodeQueue <- new Queue<Node>();
            Solution newSolution <- solution;
            Node currentNode <- newSolution.Maze.StartingNode;
            List<Node> treasureFound <- new List<Node>();
            Console.WriteLine("In AllTreasure");
            List<Node> allTreasures <- getAllTreasure(solution);
            Console.WriteLine("Out AllTreasure");

            this.nodeQueue.Enqueue(solution.Maze.StartingNode);
            while (true)
            {
                currentNode <- this.nodeQueue.Dequeue();
                newSolution.Route.AddNodeToRoute(currentNode);
                if (currentNode.Status != "Visited")
                {
                    newSolution.VisitedNode++;
                }

                if (currentNode.Type = "Treasure" && currentNode.Status !=
"Visited" && !isAlreadyFound(currentNode, treasureFound))
                {
                    newSolution.TreasureFound++;
                    treasureFound.Add(currentNode);

                    // Console.WriteLine("TREASURE FOUND: " +
newSolution.TreasureFound + " id: " + currentNode.X + "," + currentNode.Y);

                    searchPath(currentNode, newSolution);
                    resetPreviousNode(newSolution);
                    resetStatus(newSolution);
                    nodeQueue.Clear();

                    Console.WriteLine("In");
```

```
                    if (isAllTreasuresFound(treasureFound, allTreasures))
                    {
                        for (int i <- 0; i < newSolution.Route.NSelectedRoute;
i++)
                        {
                            if (newSolution.Route.SelectedRouteGraph[i] !=
null)
                            {
                                newSolution.Route.SelectedRouteGraph[i].Status
<- "Visited";
                                Console.WriteLine("Out");
                            }
                        }
                        break;
                    }

                }
                search(newSolution, currentNode);
                currentNode.Status <- "Visited";
            }

            newSolution.Route.InitializeStep();

            return newSolution;
        }

        public void search(Solution solution, Node currentNode)
        {
            currentNode.Status <- "Checking";
            if (currentNode.Left != null && currentNode.Left.Status = "Not
visited" && !nodeQueue.Contains(currentNode.Left))
            {
                currentNode.Left.Previous <- currentNode;
                this.nodeQueue.Enqueue(currentNode.Left);
            }
            if (currentNode.Top != null && currentNode.Top.Status = "Not
visited" && !nodeQueue.Contains(currentNode.Top))
            {
                currentNode.Top.Previous <- currentNode;
                this.nodeQueue.Enqueue(currentNode.Top);
            }
            if (currentNode.Right != null && currentNode.Right.Status = "Not
visited" && !nodeQueue.Contains(currentNode.Right))
            {
                currentNode.Right.Previous <-currentNode;
                this.nodeQueue.Enqueue(currentNode.Right);

            }
            if (currentNode.Bottom != null && currentNode.Bottom.Status = "Not
visited" && !nodeQueue.Contains(currentNode.Bottom))
            {
                currentNode.Bottom.Previous <- currentNode;
                this.nodeQueue.Enqueue(currentNode.Bottom);

            }
        }
```

```
        public bool isAllTreasuresFound(List<Node> treasures, List<Node>
allTreasures)
        {
            if (treasures.Count() = allTreasures.Count())
            {
                int count <- 0;
                for (int j <- 0; j < allTreasures.Count(); j++)
                {
                    if (treasures.Contains(allTreasures[j]))
                    {
                        count++;
                    }
                }
                return count = allTreasures.Count();
            }
            else
            {
                return false;
            }

        }

        public List<Node> getAllTreasure(Solution solution)
        {
            List<Node> allTreasures <- new List<Node>();
            for (int i <- 0; i < solution.Maze.Depth; i++)
            {
                for (int j <- 0; j < solution.Maze.Width; j++)
                {
                    if(solution.Maze.NodeMatrix[i, j] != null)
                    {
                        if (solution.Maze.NodeMatrix[i, j].Type = "Treasure")
                        {
                            allTreasures.Add(solution.Maze.NodeMatrix[i, j]);
                        }
                    }
                }
            }
            return allTreasures;
        }

        public bool isAlreadyFound(Node treasure, List<Node> founded)
        {
            for (int i <- 0; i < founded.Count(); i++)
            {
                if (treasure = founded[i])
                {
                    return true;
                }
            }
            return false;
        }

        public void resetStatus(Solution solution)
        {
```

```
            for (int i <- 0; i < solution.Maze.Depth; i++)
            {
                for (int j <- 0; j < solution.Maze.Width; j++)
                {
                    if (solution.Maze.NodeMatrix[i, j] != null)
                    {
                        solution.Maze.NodeMatrix[i, j].Status = "Not visited";
                    }
                }
            }
        }

        public void resetPreviousNode(Solution solution)
        {
            for (int i <- 0; i < solution.Maze.Depth; i++)
            {
                for (int j <- 0; j < solution.Maze.Width; j++)
                {
                    if (solution.Maze.NodeMatrix[i, j] != null)
                    {
                        if (solution.Maze.NodeMatrix[i, j].Previous != null)
                        {
                            solution.Maze.NodeMatrix[i, j].Previous <- null;
                        }
                    }
                }
            }
        }

        public void searchPath(Node currentNode, Solution solution)
        {
            Stack<Node> pathway <- new Stack<Node>();
            Node tempNode <- currentNode;
            pathway.Push(tempNode);
            while (true)
            {
                if (tempNode.Previous = null)
                {
                    break;
                }
                else
                {
                    pathway.Push(tempNode.Previous);
                    tempNode <- tempNode.Previous;
                }
            }

            while (pathway.Count() != 0)
            {
                solution.Route.AddNodeToSelectedRoute(pathway.Pop());
            }
        }
    }
}
```

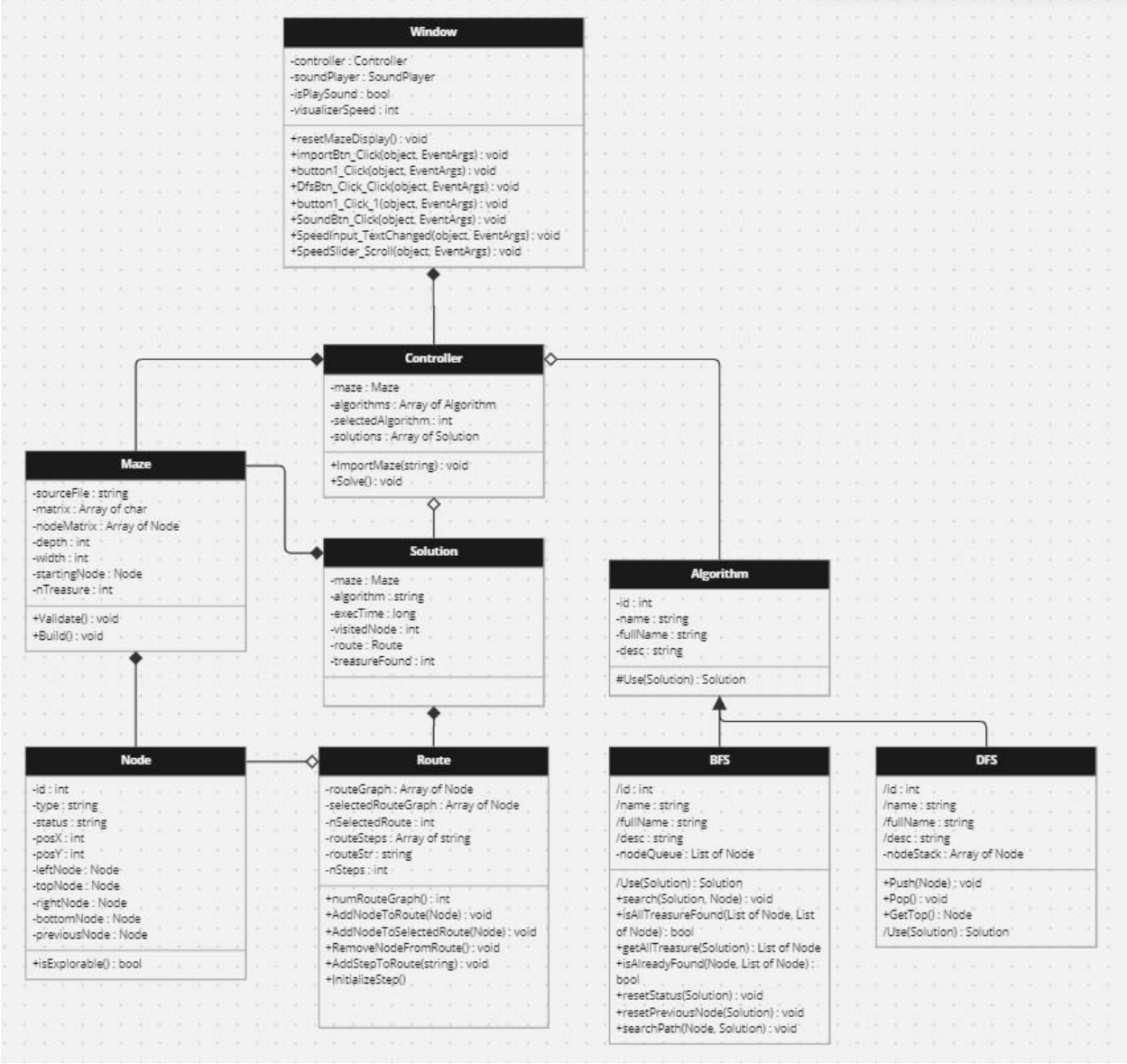
```
Algoritma Depth First Search
untuk Treasure Hunt Solver
Kelompok a-maze-ing

namespace DFSFile
{
    class DFS : Algorithm
    {
        //  ATTRIBUTES
        private Node?[] nodeStack;

        //  CONSTRUCTOR
        public DFS() : base("DFS")
        {
            this.nodeStack <- new Node[100];
        }

        //  METHODS
        public void push(Node newNode)
        {
            for (int i <- 0; i < 100; i++)
            {
                if (this.nodeStack[i] = null)
                {
                    this.nodeStack[i] <- newNode;
                    break;
                }
            }
        }

        public void pop()
        {
            for (int i <- 0; i < 100; i++)
            {
                if (this.nodeStack[i] = null)
                {
                    this.nodeStack[i - 1] <- null;
                    break;
                }
            }
        }

        public Node getTop()
        {
            for (int i <- 0; i < 100; i++)
            {
                if (this.nodeStack[i] = null)
                {
                    return this.nodeStack[i - 1];
                }
            }
            return null;
        }
```

```
        public override Solution use(Solution solution)
        {
            Solution newSolution <- solution;
            this.nodeStack <- new Node[newSolution.getMaze().getMazeDepth()
            *newSolution.getMaze().getMazeWidth()];
            push(newSolution.getMaze().getStartingNode());
            while (true)
            {
                Node currentNode <- getTop();
                newSolution.getRoute().addNodeToRoute(getTop());
                if (currentNode.getStatus() != "Visited")
                {
                    newSolution.setVisitedNode(newSolution.
                    getVisitedNode()+ 1);
                }
                if (currentNode.getType() = "Treasure" &&
                   currentNode.getStatus() != "Visited")
                {
                 newSolution.setTreasureFound(newSolution.
                 getTreasureFound() + 1);
                 Console.WriteLine("TREASURE FOUND: " +
newSolution.getTreasureFound() + " id: " + currentNode.getPosX() + "," +
currentNode.getPosY());
                    if (newSolution.getTreasureFound() =
newSolution.getMaze().getNTreasure())
                    {
                        break;
                    }
                }

                currentNode.setStatus("Checking");
                if (currentNode.getLeftNode() != null &&
currentNode.getLeftNode().getStatus() = "Not visited")
                {
                    push(currentNode.getLeftNode());
                }
                else if (currentNode.getTopNode() != null &&
currentNode.getTopNode().getStatus() == "Not visited")
                {
                    push(currentNode.getTopNode());
                }
                else if (currentNode.getRightNode() != null &&
currentNode.getRightNode().getStatus() = "Not visited")
                {
                    push(currentNode.getRightNode());
                }
                else if (currentNode.getBottomNode() != null &&
currentNode.getBottomNode().getStatus() = "Not visited")
                {
                    push(currentNode.getBottomNode());
                }
                else
                {
                    bool isKeepTrack = false;
                    while (!getTop().isExplorable())
                    {
```

```
                        if (getTop().getType() != "Treasure" && !isKeepTrack)
                        {
                            pop();
                            solution.getRoute().removeNodeFromRoute();
                        }
                        else
                        {
                            pop();
                            isKeepTrack <- true;
                            solution.getRoute().addNodeToRoute(getTop());
                        }
                    }
                }

                currentNode.setStatus("Visited");
            }
            return newSolution;
        }
    }
}
```

4.2. Struktur Kelas yang Digunakan

Berikut adalah diagram struktur kelas yang digunakan pada program *Treasure Hunt Solver* kelompok a-maze-ing.



Gambar 4.2.1 Diagram Kelas

4.3. Tata Cara Penggunaan Program

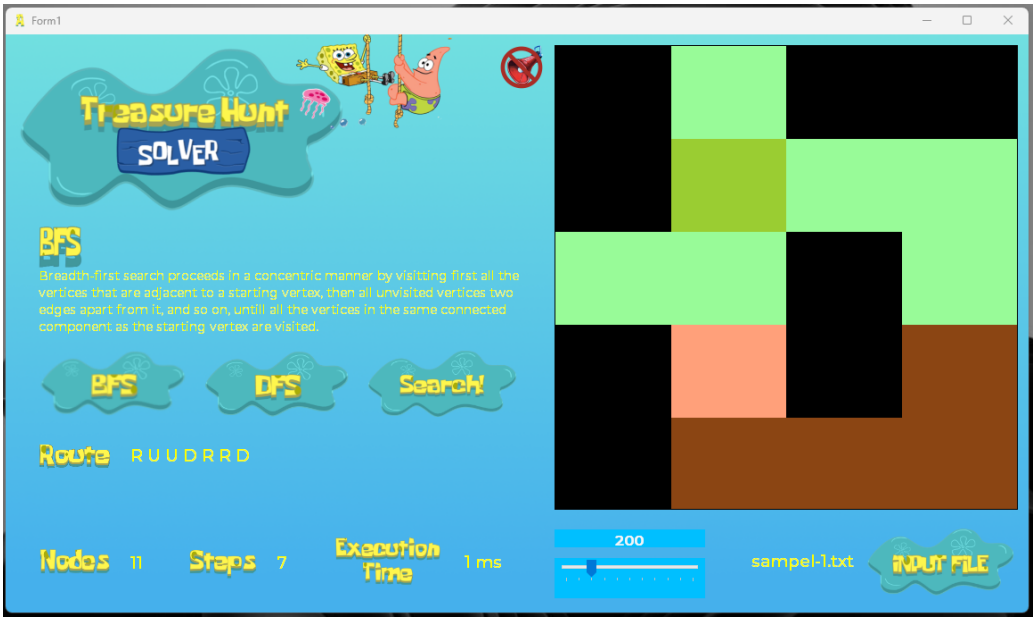
Berikut spesifikasi dan langkah-langkah dalam menjalankan program:

- Spesifikasi:
    1. .NET 7.0 terinstall
- Langkah menjalankan program
    1. Buka file Amazeing(.exe) pada folder bin

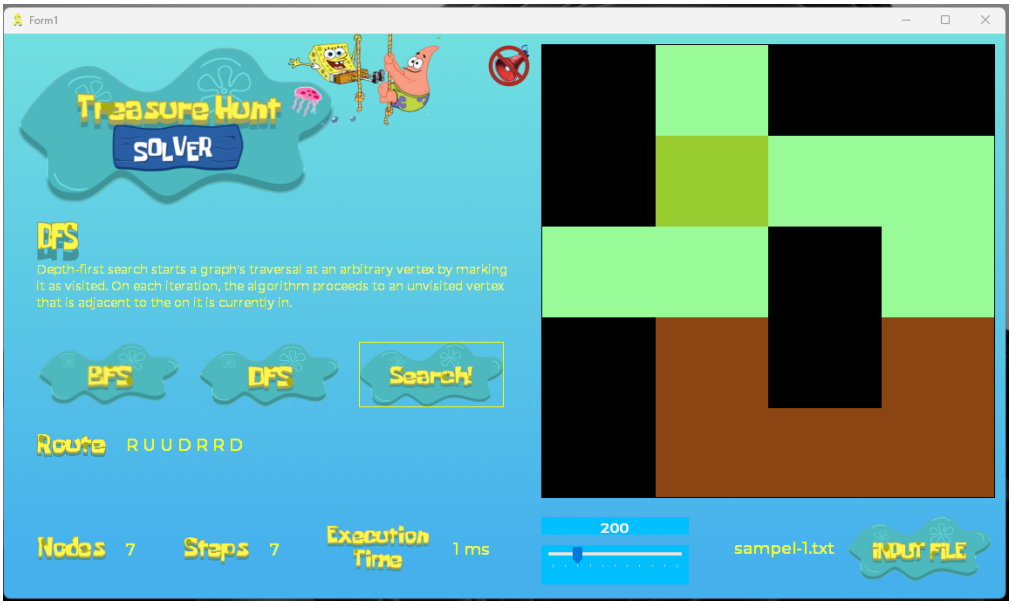
4.4. Hasil Pengujian

4.4.1. BFS dan DFS pada case 1

Gambar 4.4.1.1 Maze *Test Case* 1



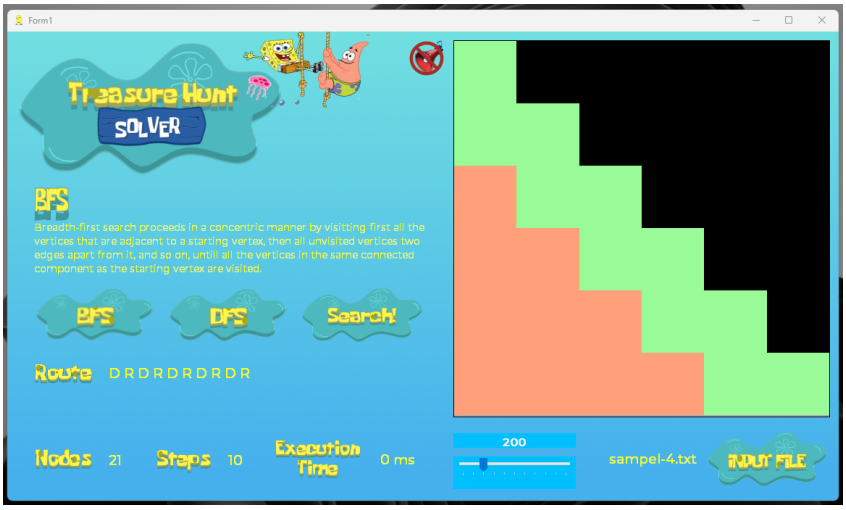
Gambar 4.4.1.2 Hasil Pengujian *Test Case* 1 dengan Algoritma BFS



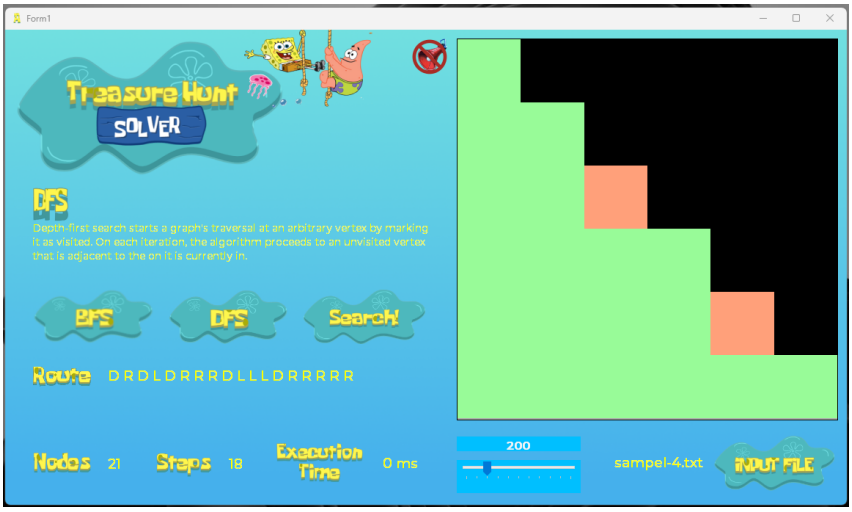
Gambar 4.4.1.2 Hasil Pengujian *Test Case* 1 dengan Algoritma DFS

4.4.2. BFS dan DFS pada case 2

Gambar 4.4.2.1 Maze *Test Case* 2

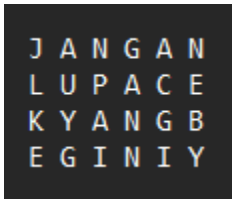


Gambar 4.4.2.1 Hasil Pengujian *Test Case* 2 dengan Algoritma BFS

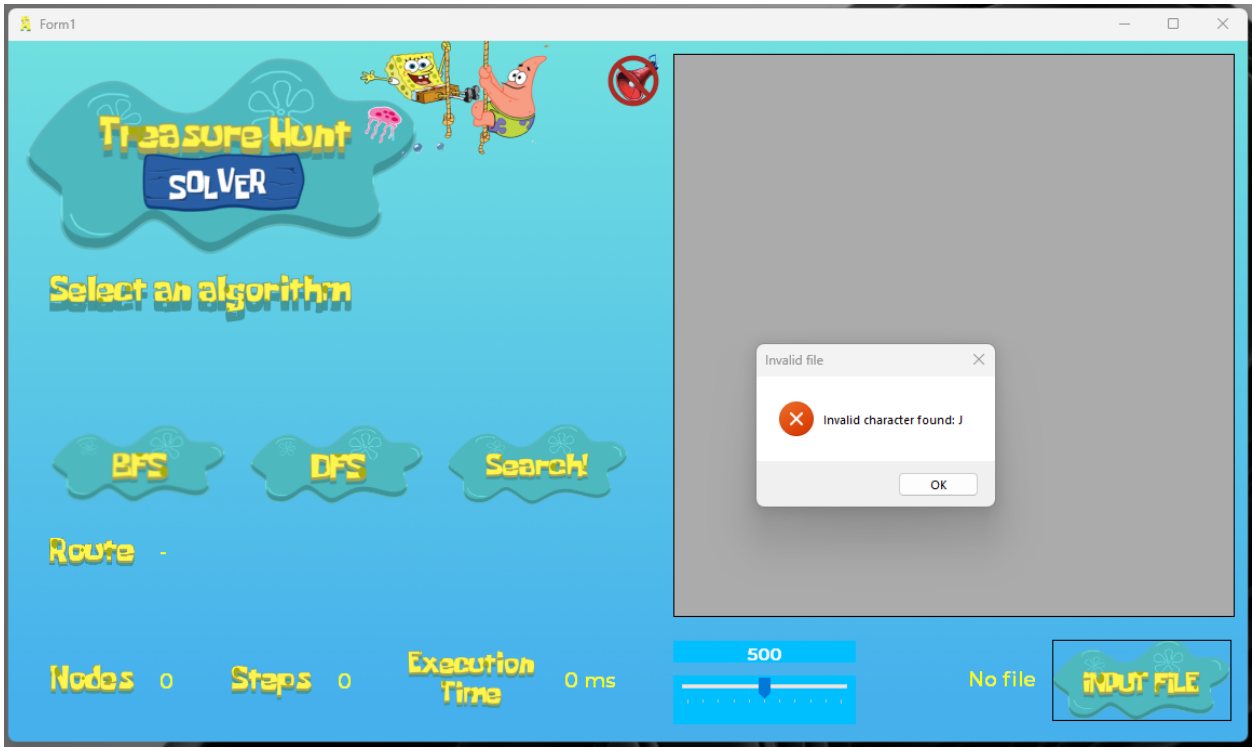


Gambar 4.4.2.2 Hasil Pengujian *Test Case* 2 dengan Algoritma DFS

4.4.3. BFS dan DFS pada case 3



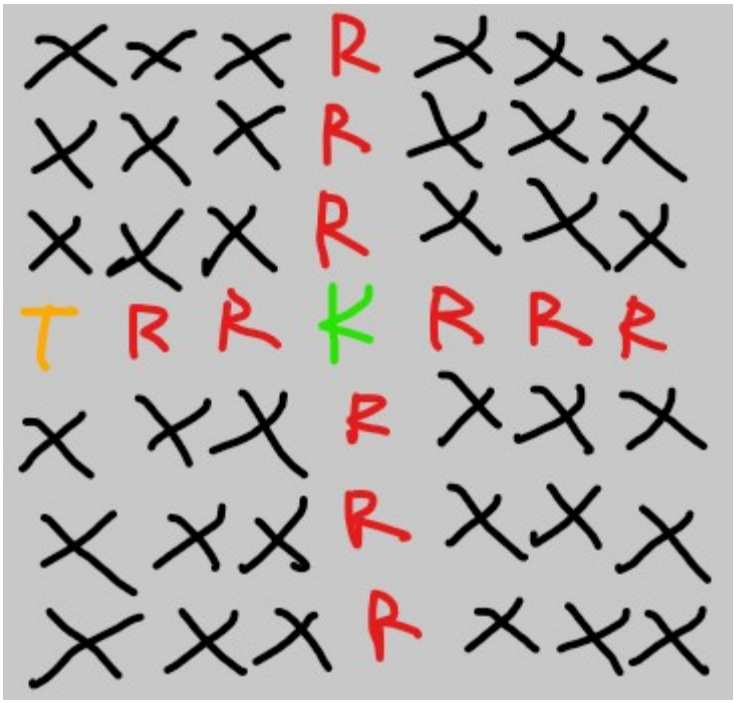
Gambar 4.4.3.1 Maze *Test Case* 3



Gambar 4.4.3.2 Hasil Pengujian *Test Case* 3 Kasus Khusus

4.5. Analisis Desain Solusi Algoritma BFS dan DFS

Desain solusi algoritma *Breadth First Search* dan *Depth First Search* memiliki kelebihan dan kekurangannya masing-masing. Secara umum, algoritma BFS lebih efisien digunakan untuk menyelesaikan permasalahan *Treasure Hunt* ini karena *scope* pencariannya dalam *maze* cenderung lebar sehingga kemungkinan menemukan *Treasure* lebih cepat. Namun, dalam beberapa kasus khusus, BFS memiliki kelemahan untuk pencarian apabila *Treasure* terletak dalam sebuah grid yang dalam sehingga dibutuhkan waktu yang lama untuk mencarinya. Sedangkan, algoritma DFS tidak menguntungkan pada kasus khusus ketika terdapat *Treasure* yang terletak dalam grid yang dangkal tetapi pada cabang yang tidak menjadi prioritas pencarian seperti pada gambar berikut.



Gambar 4.5.1 Contoh Kasus Khusus

Pada gambar 4.5.1. adalah salah satu contoh pada saat DFS akan lebih efisien dibandingkan dengan BFS jika urutan prioritas pencarian adalah *Left, Up, Right, Down* yang membuat DFS hanya perlu mengunjungi 4 *grid* sementara BFS memerlukan 10 *grid*.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Algoritma *Breadth First Search* dan *Depth First Search* merupakan algoritma yang dapat dimanfaatkan dalam kehidupan sehari-hari, khususnya dalam permasalahan pencarian rute. Dengan konsep *queue,* algoritma BFS menyelesaikan sebuah persoalan dengan pencarian melebar. Sedangkan, algoritma DFS menggunakan konsep *stack* untuk melakukan pencarian mendalam. Pada tugas besar 2 IF2211 Strategi Algoritma ini, kelompok telah mengimplementasikan kedua algoritma tersebut dalam program *Treasure Hunt Solver.*

5.2. Saran

Berikut adalah saran yang kami berikan untuk pengerjaan tugas besar ini:

1. Eksplorasi yang mendalam terhadap bahasa pemrograman C# serta C# *desktop application development.*
2. Memerhatikan kebutuhan instalasi aplikasi Visual Studio dalam pengerjaan program.

5.3. Refleksi

Melalui tugas besar ini, kelompok belajar memahami algoritma *Breadth First Search* dan *Depth First Search* dan implementasinya dalam suatu permasalahan. Kelompok belajar untuk bekerja sama dan bekerja keras memikirkan algoritma BFS, DFS, serta desain GUI. Kelompok belajar untuk membagi tugas di tengah kesibukannya masing-masing. Selain itu, melalui batasan yang diberikan pada tugas ini, kelompok belajar untuk mengikuti arahan yang diberikan sesuai spesifikasi.

5.4. Tanggapan Kelompok

Tanggapan kelompok terhadap tugas besar 2 IF2211 Strategi Algoritma adalah sebagai berikut.

NIM	Tanggapan
13521006	Semangat ngerjain design GUI soalnya ada spombob.
13521012	BFS ngeselin.
13521019	Seru seru.

DAFTAR PUSTAKA

1. Munir, Rinaldi. 2022. Bahan Kuliah IF2120 Matematika Diskrit: Graf (Bag. 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf Diakses pada 23 Maret 2023.
2. Munir, Rinaldi., Maulidevi, Nur Ulva. 2021. Bahan Kuliah IF2211 Strategi Algoritma: Breadth First Search (BFS) dan Depth First Search (DFS) (Bag. 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf Diakses pada 23 Maret 2023.

LAMPIRAN

1. Tautan repository
   https://github.com/AlphaThrone/Tubes2_a-maze-ing.git

2. Foto kelompok yang asli