LAPORAN TUGAS BESAR II

Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan Maze Treasure Hunt

Laporan Ini Dibuat Untuk Memenuhi Tugas Perkuliahan Mata Kuliah Strategi Algoritma (IF2211)

Disusun oleh:
Kelompok 5
“Ckptw”

Anggota:
Fahrian Afdholi (13521031)
Fakih Anugerah Pratama (13521091)
Reza Pahlevi Ubaidillah (13521165)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2022/2023

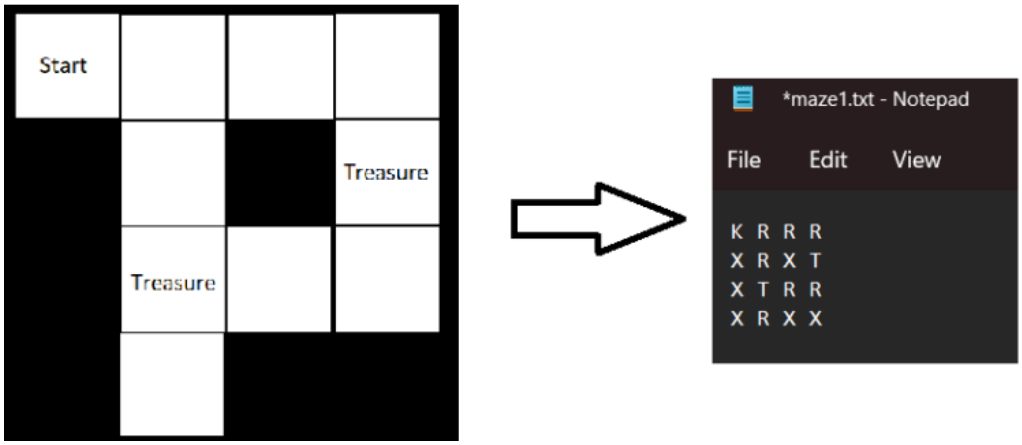
Daftar Isi

BAB 1
Deskripsi Tugas

Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan BFS dan DFS untuk mendapatkan rute memperoleh seluruh treasure atau harta karun yang ada. Program dapat menerima dan membaca input sebuah file txt yang berisi maze yang akan ditemukan solusi rute mendapatkan treasure-nya. Untuk mempermudah, batasan dari input maze cukup berbentuk segi-empat dengan spesifikasi simbol sebagai berikut :

- K : Krusty Krab (Titik awal)
- T : Treasure
- R : Grid yang mungkin diakses / sebuah lintasan
- X : Grid halangan yang tidak dapat diakses

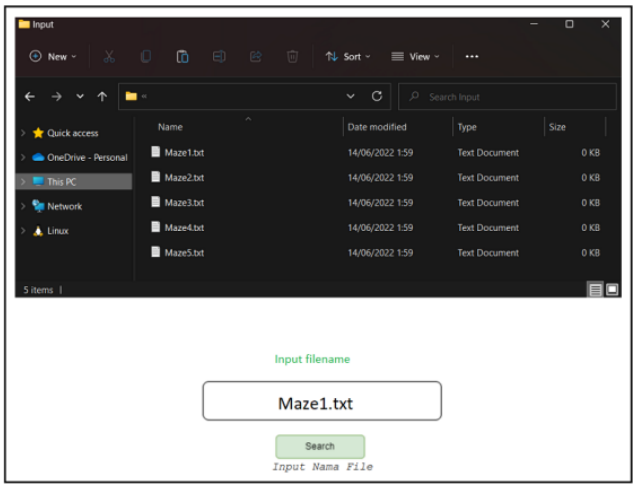
Contoh file input:



Gambar 1. Ilustrasi input file maze

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang memperoleh seluruh treasure pada maze. Perhatikan bahwa rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan asalkan ditulis di laporan ataupun readme, semisal LRUD (left right up down). Tidak ada pergerakan secara diagonal. Anda juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid maze serta hasil pencarian rute solusinya. Cara visualisasi grid dibebaskan, sebagai contoh dalam bentuk matriks yang ditampilkan dalam GUI dengan keterangan berupa teks atau warna. Pemilihan warna dan maknanya dibebaskan ke masing - masing kelompok, asalkan dijelaskan di readme / laporan.

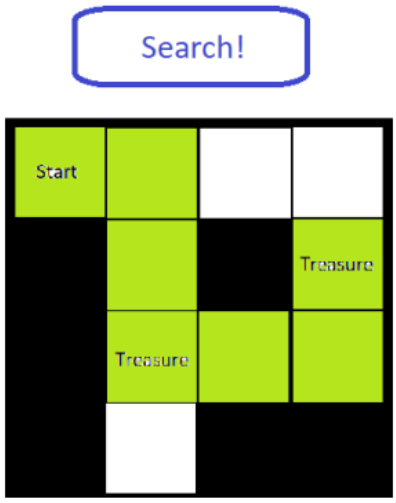
Contoh input aplikasi:



Gambar 2. Contoh input program

Daftar input maze akan dikemas dalam sebuah folder yang dinamakan test dan terkandung dalam repository program. Folder tersebut akan setara kedudukannya dengan folder src dan doc (struktur folder repository akan dijelaskan lebih lanjut di bagian bawah spesifikasi tubes). Cara input maze boleh langsung input file atau dengan textfield sehingga pengguna dapat mengetik nama maze yang diinginkan. Apabila dengan textfield, harus menghandle kasus apabila tidak ditemukan dengan nama file tersebut.

Contoh output aplikasi:



Gambar 3. Contoh output program untuk gambar 1

Setelah program melakukan pembacaan input, program akan memvisualisasikan gridnya terlebih dahulu tanpa pemberian rute solusi. Hal tersebut dilakukan agar pengguna dapat mengerjakan terlebih dahulu treasure hunt secara manual jika diinginkan. Kemudian, program

menyediakan tombol solve untuk mengeksekusi algoritma DFS dan BFS. Setelah tombol diklik, program akan melakukan pemberian warna pada rute solusi.

BAB 2
Landasan Teori

2.1. Dasar Teori

Graf transversal atau juga dikenal sebagai "*cutset*" adalah himpunan simpul pada sebuah graf yang jika dihapus, akan memisahkan graf menjadi dua atau lebih komponen yang tidak terhubung. Dalam kata lain, graf transversal adalah himpunan simpul yang, jika dihapus dari graf, akan memutus jalur atau hubungan antara simpul-simpul tertentu pada graf tersebut. Graf transversal sering digunakan dalam teori graf dan jaringan komputer untuk memodelkan hubungan antara simpul-simpul pada suatu jaringan atau sistem. Selain itu, graf transversal juga digunakan dalam optimasi jaringan untuk menentukan jalur terpendek atau rute terbaik dalam jaringan, serta dalam pengembangan perangkat lunak untuk memecahkan masalah dalam bidang jaringan dan pemrograman. Dalam banyak aplikasi, graf transversal memainkan peran penting dalam pemodelan dan analisis jaringan dan sistem.

Graf DFS (Depth-First Search) adalah metode pencarian jalur pada graf yang dilakukan dengan mengunjungi simpul-simpul graf secara berurutan mulai dari simpul awal hingga simpul terakhir. Metode pencarian ini dilakukan dengan mengeksplorasi setiap simpul secara mendalam terlebih dahulu sebelum bergerak ke simpul yang belum dieksplorasi sebelumnya. Hal ini dilakukan dengan menggunakan teknik rekursi, di mana simpul-simpul graf akan ditelusuri secara terus-menerus hingga tidak ada simpul yang belum dieksplorasi. Metode DFS sangat berguna dalam menemukan jalur terpendek antara simpul-simpul pada graf, serta dalam memeriksa keterhubungan antara simpul-simpul pada graf yang memiliki banyak cabang dan relasi yang kompleks. Selain itu, metode DFS juga sering digunakan dalam pengembangan perangkat lunak dan analisis data untuk memecahkan masalah yang melibatkan graf dan struktur data.

Graf BFS (Breadth-First Search) adalah metode pencarian jalur pada graf yang dilakukan dengan mengunjungi simpul-simpul graf secara berurutan mulai dari simpul awal hingga simpul terakhir, dengan cara mengeksplorasi semua simpul yang terhubung dengan simpul awal terlebih dahulu sebelum bergerak ke simpul yang lebih jauh. Metode pencarian ini dilakukan dengan menggunakan teknik antrian, di mana simpul-simpul yang dieksplorasi akan ditambahkan ke dalam antrian dan diambil satu per satu sesuai dengan urutan kedatangan. Dengan metode BFS, simpul-simpul graf akan ditelusuri secara lebar dan teratur, sehingga metode ini sangat berguna dalam menemukan jalur terpendek antara simpul-simpul pada graf dan memeriksa keterhubungan antara simpul-simpul pada graf dengan banyak cabang dan relasi yang kompleks. Selain itu, metode BFS juga sering digunakan dalam pengembangan perangkat lunak dan analisis data untuk memecahkan masalah yang melibatkan graf dan struktur data.

C# desktop application development adalah proses pembuatan perangkat lunak desktop yang menggunakan bahasa pemrograman C#. C# merupakan bahasa pemrograman yang dikembangkan oleh Microsoft dan dirancang untuk membangun aplikasi berbasis Windows. Dalam pengembangan aplikasi desktop menggunakan C#, developer dapat memanfaatkan berbagai fitur yang disediakan oleh .NET Framework, seperti Windows Forms, WPF, dan lain-lain. Hal ini memungkinkan developer untuk membuat aplikasi desktop yang memiliki antarmuka pengguna yang menarik, serta fitur-fitur seperti manajemen data, pemrosesan gambar, akses ke sumber daya jaringan, dan banyak lagi. Selain itu, dengan adanya Visual Studio, sebuah IDE (Integrated Development Environment) dari Microsoft, developer dapat mempercepat proses pengembangan aplikasi desktop dengan menyediakan fitur-fitur seperti debugging, code completion, dan pengujian secara otomatis. C# desktop application development sangat penting dalam pengembangan perangkat lunak untuk aplikasi bisnis, permainan, dan banyak lagi.

2.2. Cara Kerja Program

Untuk menjalankan program C# yang sudah ada, diperlukan alat-alat berikut agar dapat menjalankan program dengan baik dan benar:

- Visual Studio
  https://visualstudio.microsoft.com/downloads/
- .NET 7
  https://dotnet.microsoft.com/en-us/download/dotnet/7.0

Untuk membangun dan menjalankan program ini diperlukan proses berikut

1. Clone repository yang berada pada lampiran
2. Install .NET 7 dan Visual Studio
3. Buka Visual Studio
4. Pilih file lalu tekan open
5. Pilih Project/Solution
6. Arahkan pada program yang sudah di clone sebelumnya
7. Lalu pilih Tubes2_Ckptw.csproj atau Tubes2_Ckptw.sln
8. Setelah kembali ke Visual Studio tekan tombol play berwarna hijau di samping tulisan Tubes2_Ckptw
9. Program akan berjalan sesuai dengan semestinya

BAB 3
Aplikasi Algoritma BFS dan DFS

3.1. Langkah-langkah Pemecahan Masalah

Pendekatan algoritma BFS adalah sebagai berikut:

1. Mengubah representasi graf dari matriks menjadi dictionary yang menjelaskan hubungan ketetanggaan tiap node.
2. Membuat sebuah *queue* kosong yang menyimpan variabel yang diperlukan, diantaranya path yang sudah ditempuh selama itu, node-node yang sudah dilalui oleh path, harta karun yang sudah diambil, posisi saat ini, stack untuk backtrack, dan node-node yang sudah dikunjungi.
3. Kemudian, akan dilakukan proses pengecekan untuk setiap state yang disimpan ke dalam queue. Jika statenya sudah *goal state*, maka return path. Jika jalan buntu (atau semua tetangga sudah dikunjungi) tetapi baru saja menemukan treasure, maka lakukan backtrack. Jika jalan buntu, lanjutkan ke queue berikutnya. Selain itu, masukkan semua neighbour beserta state saat ini dari node yang sedang diperiksa ke dalam queue dengan urutan UP, LEFT, DOWN, RIGHT.
4. Lakukan *dequeue* terus sampai *queue* kosong (berarti tidak ditemukan solusi), atau sampai mencapai goal state.

Pendekatan algoritma DFS adalah sebagai berikut:

1. Mengambil seluruh matriks terlebih dahulu yang berisikan map yang sudah dibuat dan membuat sebuah variabel yang berisikan char dengan matriks dua dimensi
2. Pada DFS akan membuat sebuah variabel yang berisikan stack kosong yang akan diisi oleh edge yang berisikan treasure atau visit dengan prioritas yang akan di push adalah UP, DOWN, LEFT, RIGHT. Jadi jika ada sebuah edge di sebelah kanan maka edge tersebut akan dikerjakan pertama kali karena edge tersebut berada pada stack paling atas, setelahnya akan dikerjakan edge sesuai dengan antrian prioritasnya masing masing.
3. DFS akan melakukan push dan pop sampai program selesai dikerjakan

3.2. Mapping

Mapping persoalan pada BFS:

1. Matriks visited yang merepresentasikan banyaknya suatu node dikunjungi pada suatu state.
2. Graf dengan representasi dictionary yang menyimpan pair dengan node sebagai key dan tetangga-tetangganya sebagai value.
3. Antrean yang menyimpan state-state saat itu.

4. Stack yang menyimpan node yang masuk ke dalam path, berguna untuk melakukan backtracking.

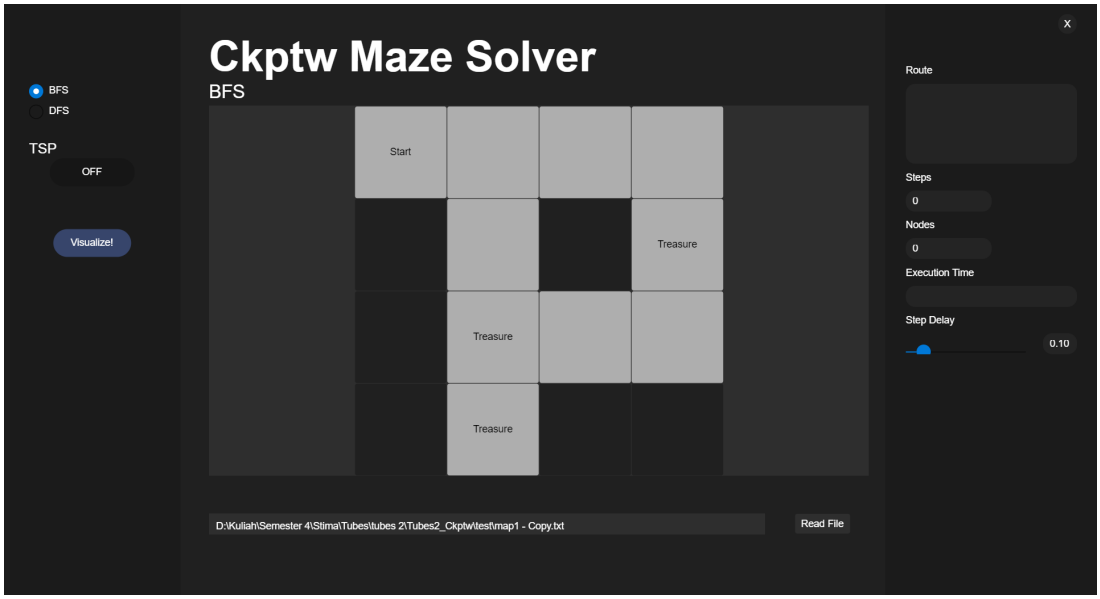
Mapping persoalan DFS:

1. Graf dengan representasi matriks yang menyimpan informasi tiap node.
2. List yang menyimpan node-node yang belum dikunjungi.
3. List yang menyimpan node-node yang menyimpan treasure. Ketika node dengan treasure sudah dikunjungi, maka akan dihapus dari list ini.
4. Stack yang menyimpan node secara runtut. Digunakan untuk melakukan backtracking secara rekursif.

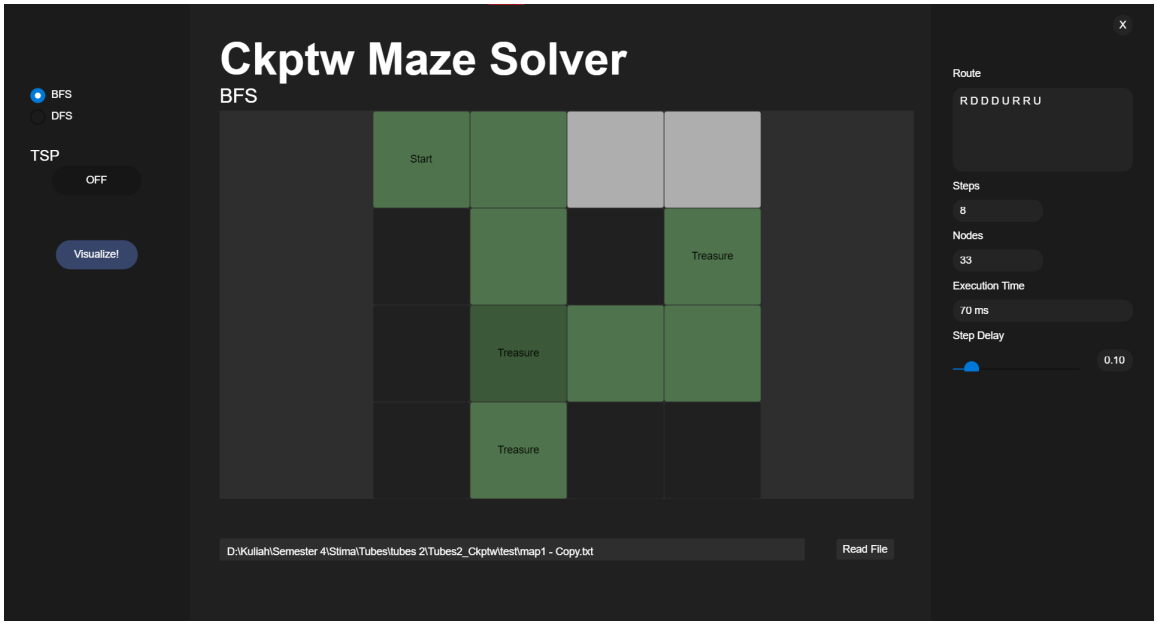
3.3. Kasus Lain

Contoh kasus 1:

a) Contoh masukan:



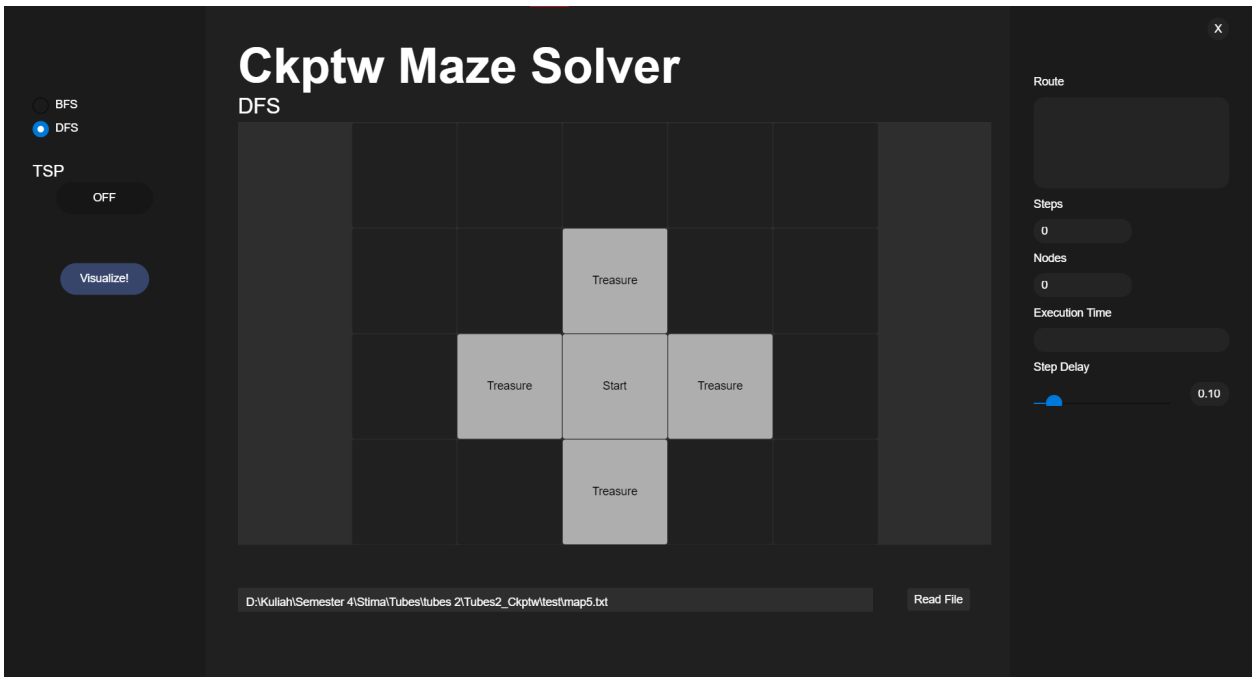
b) Contoh keluaran:



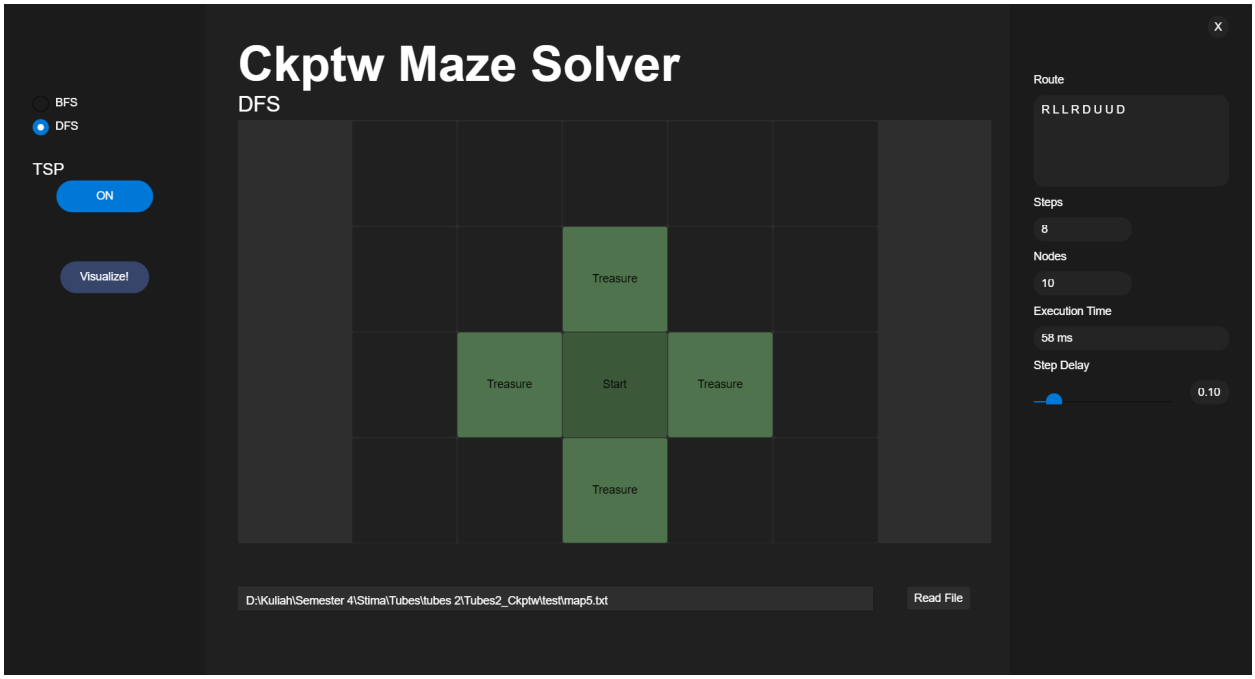
Kasus ini hanyalah modifikasi kecil dari contoh yang diberikan pada spesifikasi, hanya saja kali ini algoritma dipaksa untuk mengunjungi suatu node lebih dari sekali, yaitu karena adanya treasure(2,4). Node berwarna lebih gelap berarti ia dikunjungi lebih dari sekali.

Contoh kasus 2:

a) Contoh masukan:



b) Contoh keluaran:



Pada kasus ini, algoritma akan bergerak menuju semua treasure kemudian kembali ke titik start dengan path R-L-L-R-D-U-U-D. Dalam hal ini, titik start akan dilalui oleh algoritma sebanyak 4 kali.

BAB 4
Analisis Pemecahan Masalah

4.1. Breadth-first Search

Berikut adalah implementasi dari algoritma BFS yang digunakan. Algoritma yang diperlihatkan hanya mencakup bagian-bagian penting program saja dan tidak sama seutuhnya seperti kode program pada source code.

4.1.1. Implementasi Algoritma

```
function toGraph(input maze: list of list of char) -> Dictionary of (int,int), list of (char, list of (int,int), (int,int))
{
        Mengubah representasi graph maze dari matrix menjadi dictionary dengan key sebagai posisi dan valuenya list of neighbour yang mencakup arah, kepemilikan treasure,
dan posisi neighbour itu. Misalnya,
        (1,1): [('D', [], (2,1)),
                ('R', [(1,2)], (1,2))]
Berarti, simpul (1,1) memiliki 2 tetangga, yaitu (2,1) dan (1,2), dengan arah 'D' dan 'R' secara berurutan, serta simpul (2,1) tidak memiliki treasure, sementara (1,2)
memiliki dirinya sendiri sebagai treasure.
}

DEKLARASI
graph: Dictionary of (int,int), list of (char, list of (int,int), (int,int))
height, width, row, col, treasureCount: int
current, down, right: (int,int)

ALGORITMA
height <- maze.GetLength(0)
width <- maze.GetLength(1)
graph()

row traversal [0..height-1]
        col traversal [0..width-1]
                if(maze[row][col] in "KRT") then
                        graph.Add((row,col), [])
                if(maze[row][col]=='T') then
                        treasureCount++
                if(maze[row][col]=='K') then
                        start <- (row,col)

for(current in graph.Keys){
        (row, col) <- current
        down <- (row+1, col)
        right <- (row, col+1)

        if(down in graph) then
                graph[current].Add(('D', maze[row+1][col]=='T' ? [down] : [], down))
                graph[current].Add(('U', maze[row][col]=='T' ? [current] : [], current))
        if(right in graph) then
                graph[current].Add(('R', maze[row][col+1]=='T' ? [right] : [], right))
                graph[current].Add(('L', maze[row][col]=='T' ? [current] : [], current))
}

-> graph
```

```
function solve(input tsp: boolean) -> list of char
{
	Mengembalikan solusi maze dalam bentuk list of character dengan algoritma BFS. Jika tsp true, maka akan dikonkatenasikan dengan langkah menuju ke KrustyKrab dari
posisi BFS terakhir. Dipastikan shortest path.
}

DEKLARASI
type BFS_Tuple: <path: string,
                 path_list: list of (int,int),
                 treasures: HashSet of (int,int),
                 position: (int,int),
                 stack: Stack of ((int,int), int, char),
                 visited: list of list of int>
q: Queue of BFS_Tuple
newPath, pathBack: string
bfs_tuple: BFS_Tuple
current_row, current_col: int
newPathList: list of (int,int)
newTreasures: HashSet of (int,int)
newStack: Stack of ((int,int), int, char)
newVisited: list of list of int
neighbour: (int,int)

ALGORITMA
(start_row, start_col) <- start
q.Enqueue(BFS_Tuple("", [], {}, start, [], [0]))

while(q not empty)
{
        bfs_tuple <- q.Dequeue()
        (current_row, current_col) <- bfs_tuple.position

        { goal state }
        if(bfs_tuple.treasures.Count == treasureCount) then
                if(tsp) then
                        pathBack <- pathToStart(bfs_tuple)
                        -> (bfs_tuple.path + pathBack.path).ToList()
                else
                        -> bfs_tuple.path.ToList()


        { backtrack state }
        if(allNeighboursVisited(bfs_tuple) AND bfs_tuple.stack not empty AND bfs_tuple.treasures.Count > bfs_tuple.stack.Top()[1]) then
                (previousPosition, previousTreasureCount, previousDirection) <- bfs_tuple.stack.Pop()
                bfs_tuple.path_list.Add(bfs_tuple.position)
                newPath <- bfs_tuple.path + reverseDirection()
                newPathList <- bfs_tuple.path_list + bfs_tuple.position
                newTreasures <- bfs_tuple.treasures
                newVisited <- [0]
                for(path in newPathList) do
                        newVisited[path[0]][path[1]]++
                q.Enqueue(BFS_Tuple(newPath, newPathList, newTreasures, previousPosition, bfs_tuple.stack, newVisited))
                continue

        { deadend state }
        if(bfs_tuple.visited[current_row][current_col]>0 AND NOT (bfs_tuple is not empy AND bfs_tuple.treasures.Count > bfs_tuple.Top()[1])) then
                continue

        { search next state}
        bfs_tuple.visited[current_row, current_col]++
        for(node in graph[bfs_tuple.position]) do
                direction <- node[0]
                newTreasures <- HashSet(bfs_tuple.treasures)
                if(node[1].Count!=0) then
                        newTreasures.Add(node[1][0])
                neighbour <- node[2]
                newPathList <- bfs_tuple.path_list + neighbour
                newStack <- bfs_tuple.stack.copy()
                if(NOT allNeighboursVisited(bfs_tuple)) then
                        newStack.Push((bfs_tuple.position, bfs_tuple.treasures.Count, direction))
                newVisited <- [0]
                if(newTreasures.Count > bfs_tuple.treasures.Count) then
                        for(path in newPathList) do
                                newVisited[path[0]][path[1]]++
                else
                        newVisited <- bfs_tuple.visited

                q.Enqueue(BFS_Tuple(bfs_tuple.path+direction, newPathList, newTreasures, neighbour, newStack, newVisited))

        -> "".ToList()

}
```

4.1.2. Penjelasan Struktur Data

a. BFS.cs

Struktur Data dan Kelas	Penjelasan Singkat
BFS_Tuple	Deklarasi struktur data yang menyimpan properti path, path_list, treasures, position, stack, dan visited. Berguna untuk melakukan queue pada setiap langkah BFS.
BFS	Kelas utama algoritma BFS yang berisi metode-metode yang berguna untuk mencari solusi path suatu maze dengan pendekatan BFS, mencari solusi path untuk

	kembali ke start, membalikkan arah suatu edge, serta mengecek apakah semua neighbours suatu node sudah dikunjungi.

4.2. Depth-first Search

4.2.1. Implementasi Algoritma

```
internal class DFS
   {
      private Stopwatch sw <- new Stopwatch();
      private int sizeStep;
      private int lengthNode;
      private char[,] mapMaze;
      private int row;
      private int col;
      private int treasure;
      private List<Tuple<int, int>> liveTreasure <- new List<Tuple<int, int>>();
      private List<Tuple<int, int>> liveNode <- new List<Tuple<int, int>>();
      private Stack<Tuple<int, int>> stack <- new Stack<Tuple<int, int>>();


      public DFS(char[,] mapMaze)
      {
         this.mapMaze <- mapMaze;
         this.row <- mapMaze.GetLength(0);
         this.col <- mapMaze.GetLength(1);

         for (int i <- 0; i < this.row; i++)
         {
            for (int j <- 0; j < this.col; j++)
            {
               if (this.mapMaze[i, j] = 'T') this.treasure++;
            }
         }
      }

      public List<char> getMovementTreasure()
      {
         sw.Reset();
         sw.Start();
         List<char> list <- new List<char>();
         List<Tuple<int, int>> direction <- getDirectionTreasure();

         for (int i <- 0; i < direction.Count - 1; i++)
         {
```

```
            if (direction[i + 1].Item1 - direction[i].Item1 = 1) list.Add('D');
            if (direction[i + 1].Item1 - direction[i].Item1 = -1) list.Add('U');
            if (direction[i + 1].Item2 - direction[i].Item2 = 1) list.Add('R');
            if (direction[i + 1].Item2 - direction[i].Item2 = -1) list.Add('L');
        }
        sw.Stop();
        this.sizeStep <- list.Count;
        return list;
    }
    public List<char> getMovementTSP()
    {
        sw.Reset();
        sw.Start();
        List<char> list <- new List<char>();
        List<Tuple<int, int>> direction <- getDirectionTSP();

        for (int i <- 0; i < direction.Count - 1; i++)
        {
            if (direction[i + 1].Item1 - direction[i].Item1 = 1) list.Add('D');
            if (direction[i + 1].Item1 - direction[i].Item1 = -1) list.Add('U');
            if (direction[i + 1].Item2 - direction[i].Item2 = 1) list.Add('R');
            if (direction[i + 1].Item2 - direction[i].Item2 = -1) list.Add('L');
        }
        sw.Stop();
        this.sizeStep <- list.Count;
        return list;
    }
    public string getTimeExec()
    {
        return this.sw.ElapsedMilliseconds.ToString();
    }
    public int getStep()
    {
        return this.sizeStep;
    }
    public int getNode()
    {
        return this.lengthNode;
    }

    private List<Tuple<int, int>> getDirectionTreasure()
    {
        for (int i <- 0; i < this.row; i++)
        {
            for (int j <- 0; j < this.col; j++)
            {
                if (this.mapMaze[i, j] = 'K') this.stack.Push(new Tuple<int, int>(i, j));
            }
        }
        List<Tuple<int, int>> direction <- new List<Tuple<int, int>>();
```

```
Tuple<int, int> currentDir <- this.stack.Pop();
Tuple<int, int> prevDir <- new Tuple<int, int>(-1, -1);
direction.Add(currentDir);
this.liveNode.Add(currentDir);
this.lengthNode <- 0;



while (this.treasure > 0)
{
   int check <- 0;

   try
   {
      if ((this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] = 'T' ||
         this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] = 'R' ||
         this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] = 'K') && (
         isLiveNode(new Tuple<int, int>(currentDir.Item1 - 1, currentDir.Item2)) &&
         this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] != 'X')
         )
      {
         this.stack.Push(new Tuple<int, int>(currentDir.Item1 - 1, currentDir.Item2));
         check <- 1;
      }
   }
   catch (ArgumentOutOfRangeException e) { }
   catch (IndexOutOfRangeException e) { }


   try
   {
      if ((this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] = 'T' ||
         this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] = 'R' ||
         this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] = 'K') && (
         isLiveNode(new Tuple<int, int>(currentDir.Item1 + 1, currentDir.Item2)) &&
         this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] != 'X')
         )
      {
         this.stack.Push(new Tuple<int, int>(currentDir.Item1 + 1, currentDir.Item2));
         check <- 1;
      }
   }
   catch (ArgumentOutOfRangeException e) { }
   catch (IndexOutOfRangeException e) { }

   try
   {
      if ((this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] = 'T' ||
         this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] = 'R' ||
```

```
            this.mapMaze[currentDir.Item1 , currentDir.Item2-1] = 'K') && (
            isLiveNode(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 - 1)) &&
            this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] != 'X')
            )
        {
            this.stack.Push(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 - 1));
            check <- 1;
        }
    }
    catch (ArgumentOutOfRangeException e) { }
    catch (IndexOutOfRangeException e) { }

    try
    {
        if ((this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] = 'T' ||
            this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] = 'R' ||
            this.mapMaze[currentDir.Item1, currentDir.Item2+1] = 'K') && (
            isLiveNode(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 + 1)) &&
            this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] != 'X')
            )
        {

            this.stack.Push(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 + 1));
            check = 1;
        }
    }
    catch (ArgumentOutOfRangeException e) { }
    catch (IndexOutOfRangeException e) { }



    prevDir <- currentDir;


    currentDir <- this.stack.Pop();

    if (this.mapMaze[currentDir.Item1, currentDir.Item2] = 'T')
    {
        if (!isTreasureLive(currentDir))
        {
            this.treasure--;
            this.lengthNode += this.liveNode.Count;
            this.liveNode.Clear();
            this.liveTreasure.Add(currentDir);
        }
    }

    if (check = 0)
    {
        int lebi <- Math.Abs(Math.Abs(currentDir.Item1 - prevDir.Item1) -
```

```
Math.Abs(currentDir.Item2 - prevDir.Item2));
                int jumlah <- direction[direction.Count - 1].Item1 + direction[direction.Count -
1].Item2 - (lebi);
                int calculate <- Math.Abs(jumlah - (currentDir.Item1 + currentDir.Item2 - 1));

                while (direction[direction.Count - 1].Item1 != currentDir.Item1 &&
                    direction[direction.Count - 1].Item2 != currentDir.Item2 ||
                    ((Math.Abs(direction[direction.Count - 1].Item1 - currentDir.Item1) +
Math.Abs(direction[direction.Count - 1].Item2 - currentDir.Item2)) != 1
                    ))
                {
                    direction.RemoveAt(direction.Count - 1);
                }
                prevDir <- new Tuple<int, int>(direction[direction.Count - 1].Item1,
direction[direction.Count - 1].Item2);
            }

            direction.Add(currentDir);
            this.liveNode.Add(currentDir);

        }
        this.stack.Clear();
        this.liveNode.Clear();
        return direction;
    }

    private List<Tuple<int, int>> getDirectionTSP()
    {
        List<Tuple<int, int>> direction <- getDirectionTreasure();
        Tuple<int, int> currentDir <- direction[direction.Count - 1];
        Tuple<int, int> prevDir <- new Tuple<int, int>(-1, -1);
        this.liveNode.Add(currentDir);
        this.stack.Push(currentDir);
        this.lengthNode <- direction.Count;

        int start <- 1;



        while (start > 0)
        {
            int check <- 0;

            try
            {
                if ((this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] = 'T' ||
                    this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] = 'R' ||
                    this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] = 'K') && (
                    isLiveNode(new Tuple<int, int>(currentDir.Item1 - 1, currentDir.Item2)) &&
                    this.mapMaze[currentDir.Item1 - 1, currentDir.Item2] != 'X')
```

```
            )
        {
            this.stack.Push(new Tuple<int, int>(currentDir.Item1 - 1, currentDir.Item2));
            check <- 1;
        }
    }
    catch (ArgumentOutOfRangeException e) { }
    catch (IndexOutOfRangeException e) { }


    try
    {
        if ((this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] = 'T' ||
            this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] = 'R' ||
            this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] = 'K') && (
            isLiveNode(new Tuple<int, int>(currentDir.Item1 + 1, currentDir.Item2)) &&
            this.mapMaze[currentDir.Item1 + 1, currentDir.Item2] != 'X')
            )
        {
            this.stack.Push(new Tuple<int, int>(currentDir.Item1 + 1, currentDir.Item2));
            check <- 1;
        }
    }
    catch (ArgumentOutOfRangeException e) { }
    catch (IndexOutOfRangeException e) { }

    try
    {
        if ((this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] = 'T' ||
            this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] = 'R' ||
            this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] = 'K') && (
            isLiveNode(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 - 1)) &&
            this.mapMaze[currentDir.Item1, currentDir.Item2 - 1] != 'X')
            )
        {
            this.stack.Push(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 - 1));
            check <- 1;
        }
    }
    catch (ArgumentOutOfRangeException e) { }
    catch (IndexOutOfRangeException e) { }

    try
    {
        if ((this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] = 'T' ||
            this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] = 'R' ||
            this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] = 'K') && (
            isLiveNode(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 + 1)) &&
            this.mapMaze[currentDir.Item1, currentDir.Item2 + 1] != 'X')
            )
```

```
                {
                    this.stack.Push(new Tuple<int, int>(currentDir.Item1, currentDir.Item2 + 1));
                    check <- 1;
                }
            }
            catch (ArgumentOutOfRangeException e) { }
            catch (IndexOutOfRangeException e) { }



            prevDir <- currentDir;

            currentDir <- this.stack.Pop();

            if (this.mapMaze[currentDir.Item1, currentDir.Item2] = 'K') start--;

            if (check = 0)
            {
                int lebi = Math.Abs(Math.Abs(currentDir.Item1 - prevDir.Item1) -
Math.Abs(currentDir.Item2 - prevDir.Item2));
                int jumlah = direction[direction.Count - 1].Item1 + direction[direction.Count -
1].Item2 - (lebi);
                int calculate = Math.Abs(jumlah - (currentDir.Item1 + currentDir.Item2 - 1));

                while (direction[direction.Count - 1].Item1 != currentDir.Item1 &&
                    direction[direction.Count - 1].Item2 != currentDir.Item2 ||
                    ((Math.Abs(direction[direction.Count - 1].Item1 - currentDir.Item1) +
Math.Abs(direction[direction.Count - 1].Item2 - currentDir.Item2)) != 1
                    ))
                {
                    direction.RemoveAt(direction.Count - 1);
                }
                prevDir <- new Tuple<int, int>(direction[direction.Count - 1].Item1,
direction[direction.Count - 1].Item2);

            }
            direction.Add(currentDir);
            this.liveNode.Add(currentDir);

        }
        this.lengthNode <- this.lengthNode+this.liveNode.Count;
        this.stack.Clear();
        this.liveNode.Clear();
        return direction;
    }

    private bool isLiveNode(Tuple<int, int> sample)
    {
        bool itIs <- true;
```

```
            for (int i <- 0; i < this.liveNode.Count; i++)
            {
                if (sample.Item1 = this.liveNode[i].Item1 && sample.Item2 = this.liveNode[i].Item2)
itIs <- false;
            }

            return itIs;
        }
        private bool isTreasureLive(Tuple<int, int> sample)
        {
            if(this.liveTreasure.Count=0) return false;

            for (int i <- 0; i < this.liveTreasure.Count; i++) if (sample.Item1 =
this.liveTreasure[i].Item1 && sample.Item2 = this.liveTreasure[i].Item2) return true;

            return false;
        }
```

4.2.2. Penjelasan Struktur Data

DFS	DFS adalah kelas utama yang memiliki parameter
getMovementTreasure	Fungsi yang akan me-return char matriks dua dimensi yang berisikan U,L,R,D pada pencarian treasure dan dicari sesuai dengan algoritma DFS
getMovementTSP	Fungsi yang akan me-return char matriks dua dimensi yang berisikan U,L,R,D pada pencarian treasure dan edge start
getDirectionTreasure	Fungsi yang akan me-return tuple dua integer matriks dua dimensi yang berisikan koordinat dari edge pada pencarian treasure dan dicari sesuai dengan algoritma DFS
getDirectionTSP	Fungsi yang akan me-return tuple dua integer matriks dua dimensi yang berisikan koordinat dari edge pada pencarian treasure dan edge start
isLiveNode	Fungsi yang berisikan tuple yang berisi koordinat dari edge yang mengecek apakah tuple sama dengan tuple Node yang sudah

	dihidupkan sebelumnya
isLiveTreasure	Fungsi yang berisikan tuple yang berisi koordinat dari edge yang mengecek apakah tuple sama dengan tuple treasure yang sudah dihidupkan sebelumnya
getTimeExec	Fungsi yang mereturn waktu eksekusi
getStep	Fungsi yang mereturn jumlah step
getNode	Fungsi yang mereturn jumlah node

4.3. Struktur Data Lain

4.3.1. Maze

Maze	Maze adalah sebuah kelas yang mewakili sebuah *maze* dengan properti berupa lebar dan panjang *maze* dan petak-petak *maze* yang disimpan dalam sebuah array dua dimensi
Print	Method yang memudahkan dalam mengecek isi Maze, biasa digunakan saat pengembangan.
IsMazeValid	Fungsi yang mengembalikan *bool* apakah *maze* yang diwakili kelas ini valid
UpdateSolutionState	Method yang digunakan untuk memperbarui *state* petak-petak yang ada di dalam Maze ini berdasarkan solusi yang dimasukkan. Digunakan untuk mewarnai petak yang merupakan solusi akhir dari *maze*.
AnimateSolutionState	Method yang digunakan untuk memperbarui *state* petak-petak yang ada di dalam Maze ini dengan pewarnaan yang menandai pencarian solusi. Digunakan dalam animasi pencarian solusi
ResetSolutionState	Method yang digunakan untuk mengembalikan *state* semua petak yang ada dalam *maze* ini ke *state* semula (Untravelled).

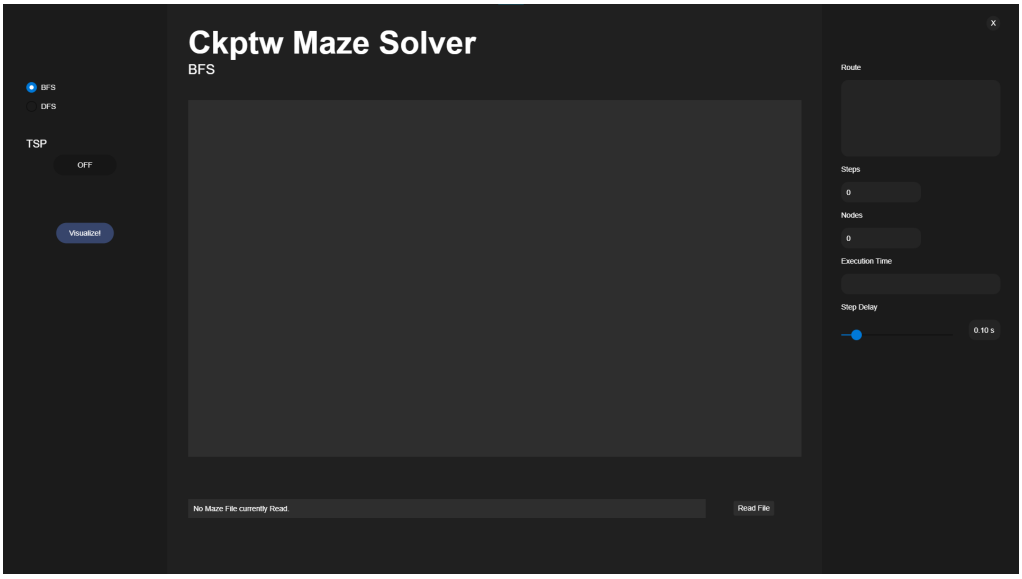
4.3.2. Maze Path

MazePath	Maze Path adalah sebuah kelas yang merepresentasikan sebuah petak yang dimiliki oleh suatu *maze*. Kelas ini mendeskripsikan petak dengan menggunakan *pathState* dan *pathSymbol* yang mewakili *state* dari masing-masing petak dan *simbol* yang diwakilinya.
PathState	Atribut kelas yang mewakili *pathState* yang dimiliki oleh petak tersebut saat ini.
PathSymbol	Atribut kelas yang mewakili *pathSymbol* yang dimiliki oleh petak tersebut.
ToString	Method *override* yang digunakan dalam method Print() Maze, memudahkan dalam pengembangan.

4.3.3. File Reader

FileReader	FileReader adalah sebuah kelas yang bertugas untuk membaca file masukan pengguna dan menyajikannya menjadi sebuah informasi yang dapat diolah untuk membangun sebuah Maze
getMapMaze	Fungsi yang digunakan untuk mendapatkan hasil pembacaan FileReader terhadap berkas Maze dalam sebuah array dua dimensi.
_getPath	Fungsi *async* yang bertugas membuat dialog yang digunakan untuk mengambil berkas
BrowseFile	Wrapper Method yang bertugas menjalankan prosedur pembacaan berkas

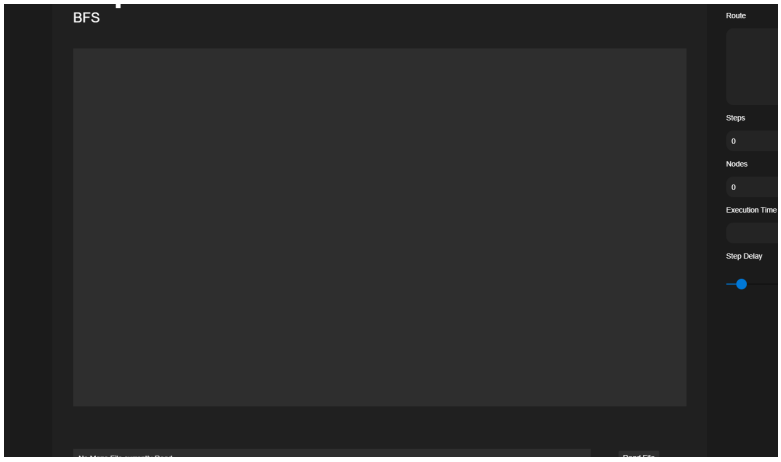
4.4. Interface Program



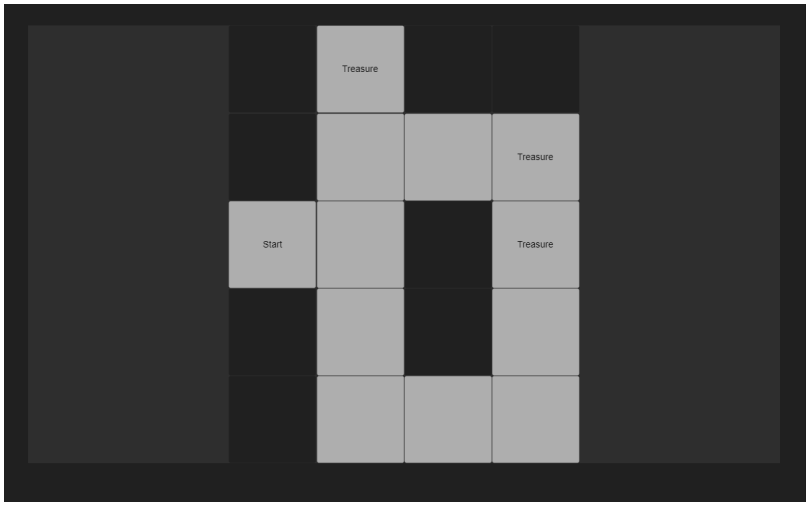
Antarmuka program

Program terdiri dari beberapa panel dan sub-panel yang memiliki fungsi dan kegunaan masing-masing.

4.4.1. Display Maze



Maze Display sebelum *maze* dimuat

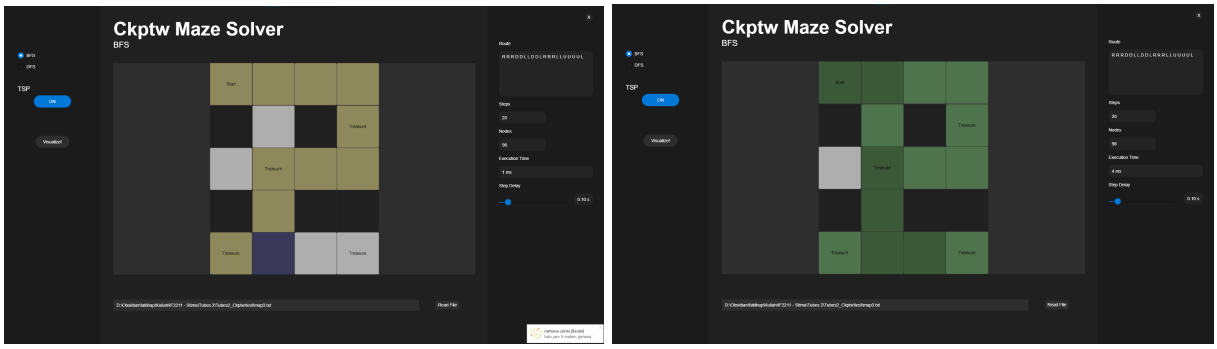


Maze Display setelah *maze* sampel-1 dimuat program

Panel *Maze Display*, akan menampilkan sebuah peta yang merepresentasikan *maze* yang telah dimuat dalam bentuk petak-petak persegi. Petak yang dapat dilewati akan diberi warna putih dan yang tidak akan diberi warna hitam. Selain itu, petak yang mewakili posisi Krusty Krab (*start*) akan diberi penanda berupa teks “Start”. Hal yang sama juga berlaku untuk posisi *Treasure*.

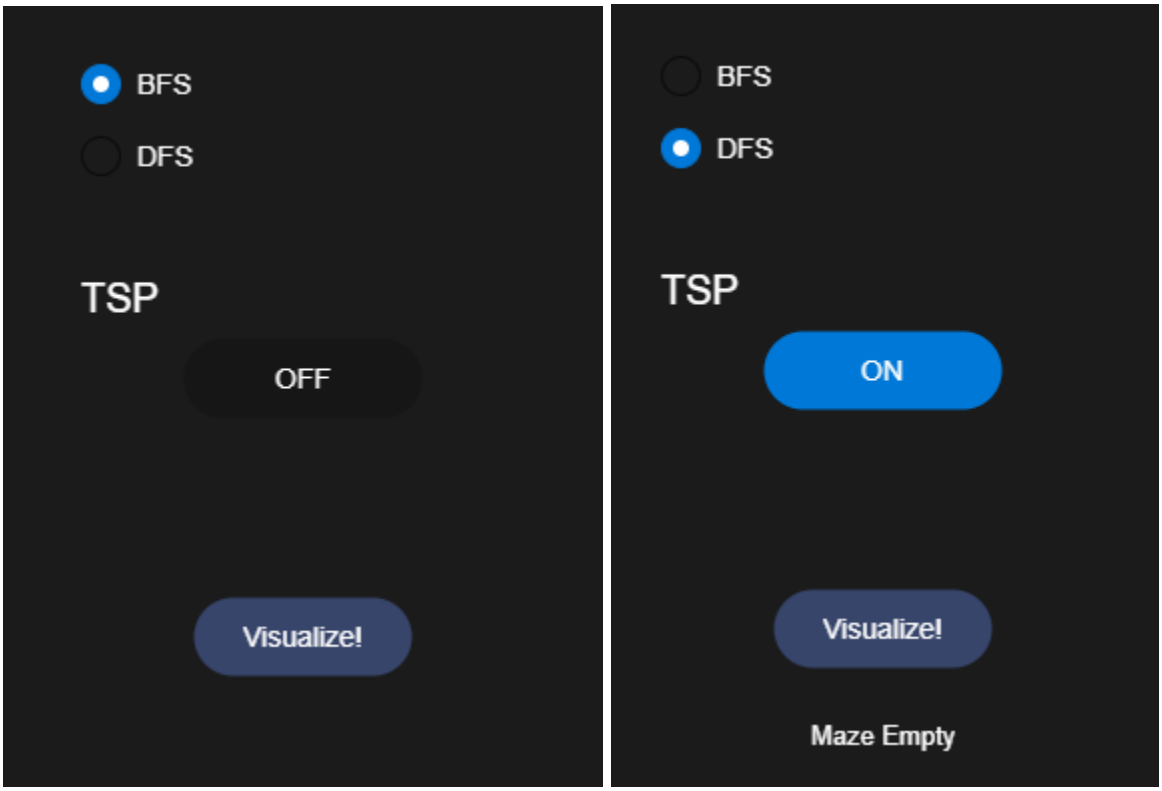
Maze Display setelah *maze* sampel-3 dimuat program

Sedangkan untuk kasus petak yang simbolnya tidak dapat dimengerti oleh program (Selain K, R, T, dan X), program akan secara otomatis melabelinya dengan “Undefined” dan program akan menolak untuk menyelesaikan *maze* ini.

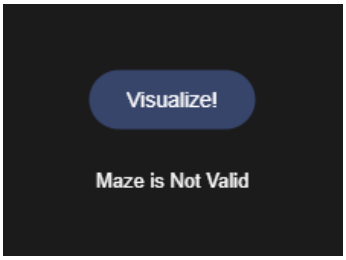


Setelah pengguna memilih *mode* pencarian dan menekan tombol “Visualize”, program akan melakukan visualisasi terhadap proses pencarian solusi dan menampilkan hasilnya. Saat dilakukan pencarian, petak yang telah dilewati akan berwarna kuning sedangkan petak yang sedang dicek akan memiliki warna biru. Saat pencarian telah selesai, program akan menampilkan hasil akhir berupa petak-petak dengan warna hijau yang mewakili petak yang dipilih/dilewati dalam solusi akhir. Petak yang berwarna hijau tua memiliki arti solusi akhir melewatinya lebih dari satu kali.

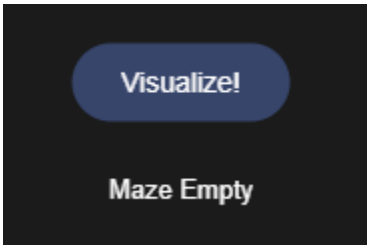
4.4.2. Maze Solver



Panel kiri program berperan sebagai tempat pengguna mengatur penyelesaian *maze* yang akan dilakukan, apakah akan menggunakan BFS atau DFS serta apakah akan menerapkan permasalahan TSP. Selain itu, pengguna bisa menekan tombol “Visualize!” untuk memulai pencarian solusi *maze* dan mendapatkan hasilnya.



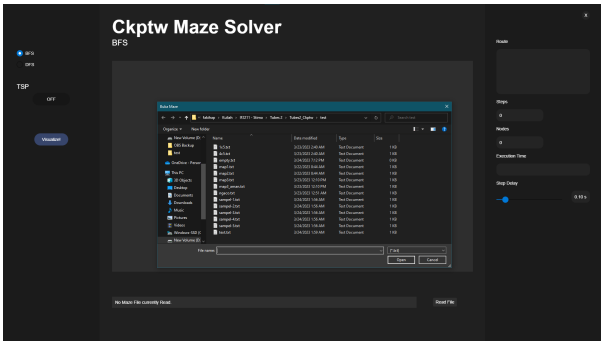
Ketika pengguna memuat *maze* yang dianggap tidak valid, program akan menolak perintah "Visualize" yang diminta pengguna dan menampilkan pesan "Maze is Not Valid".

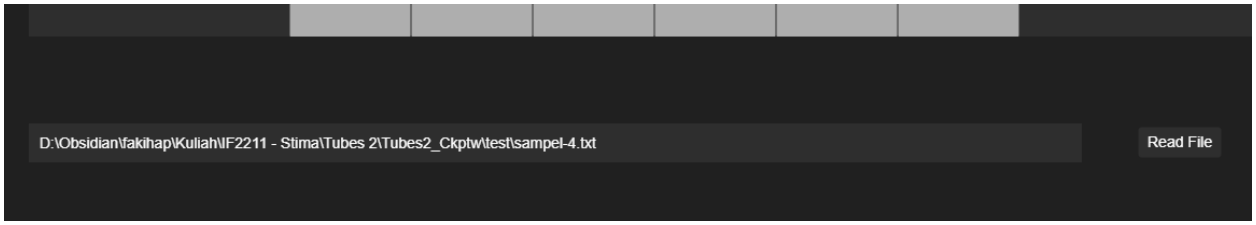


Sedangkan ketika pengguna belum memuat *maze* sama sekali, program akan menampilkan pesan "Maze Empty" ketika tombol perintah "Visualize" ditekan.

4.4.3. File Reader

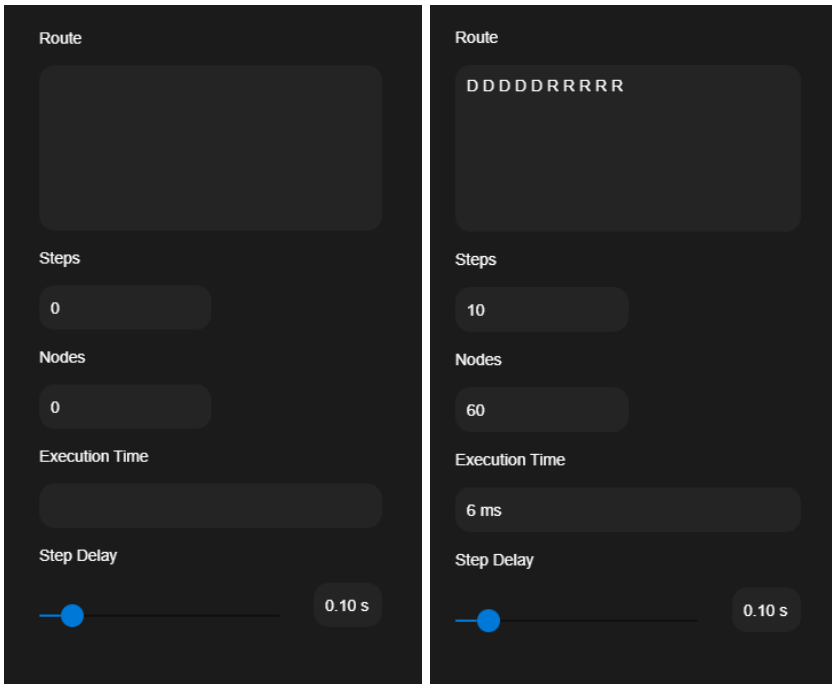
Untuk memuat sebuah *maze* dari file lokal, pengguna dapat menekan tombol "Read File". Setelah tombol ditekan, akan muncul sebuah *dialog* yang akan meminta pengguna memilih sebuah file berformat *.txt* berisi peta *maze*.





Setelah pengguna mengkonfirmasi pemilihan file, program akan memuat berkas yang dimaksud lalu membuat maze yang bersesuaian dan menampilkannya dalam *display*. Dapat dilihat juga *absolute path* dari file yang telah dipilih sebelumnya.

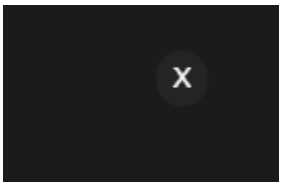
4.4.4. Maze Solution



Panel program yang berada di sebelah kanan berisi deskripsi tentang solusi hasil pencarian baik menggunakan pendekatan DFS maupun BFS. Setelah pengguna memuat *maze* dan menekan tombol "Visualize", program akan secara otomatis meng-*update* keluaran nilai yang bersesuaian dengan solusi yang ditemukan.

Slider Step Display memberikan pilihan bagi pengguna untuk mengatur jeda tiap langkah dalam visualisasi *progress* pencarian petak baik menggunakan algoritma BFS maupun DFS.

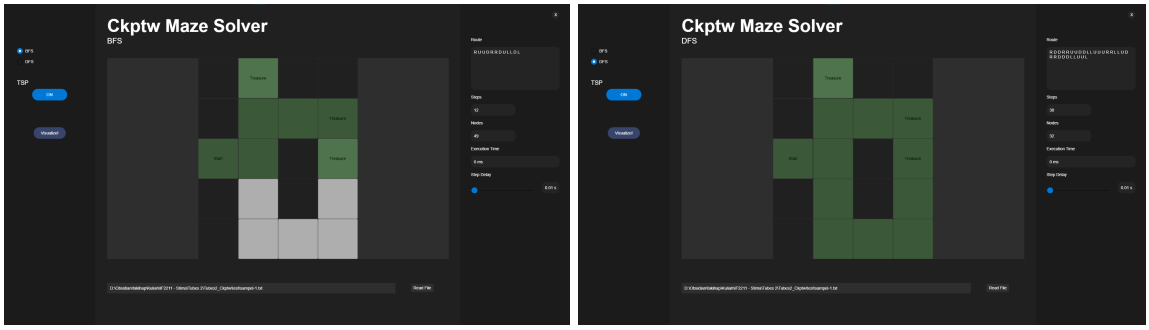
4.4.5. Close



Program juga menyediakan tombol *close* di pojok kanan atas untuk memudahkan pengguna menutup aplikasi.

4.5. Hasil Pengujian

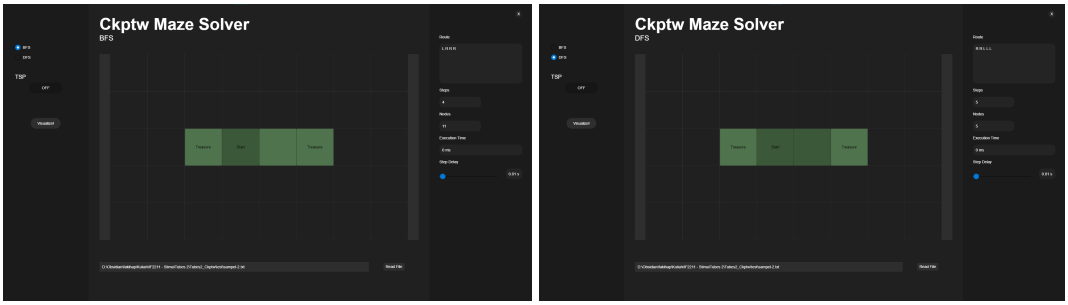
4.5.1. Kasus 1



Menggunakan penyelesaian BFS-Travelling Salesman Problem
Steps : 12
Nodes : 49
Exec Time : 0 ms

Menggunakan penyelesaian DFS-Travelling Salesman Problem
Steps : 30
Nodes : 32
Exec Time : 0 ms

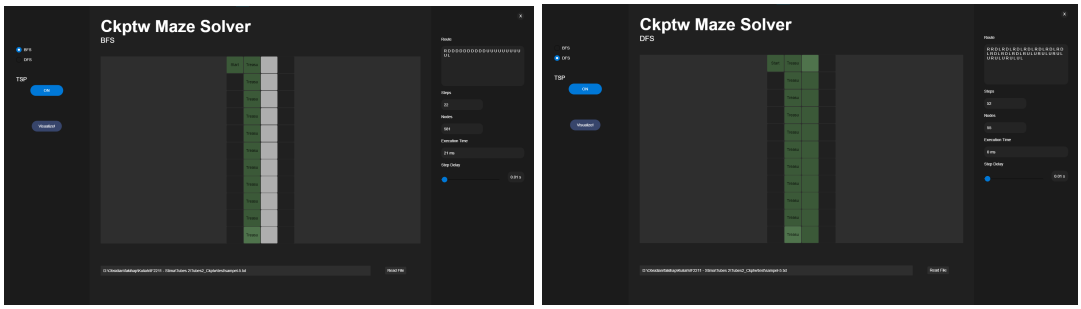
4.5.2. Kasus 2



Menggunakan penyelesaian BFS
Steps : 4
Nodes : 11
Exec Time : 0 ms

Menggunakan penyelesaian DFS
Steps : 5
Nodes : 5
Exec Time : 0 ms

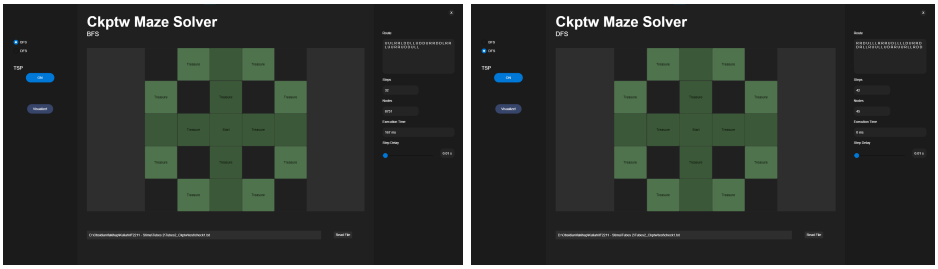
4.5.3. Kasus 3



Menggunakan penyelesaian BFS-Travelling Salesman Problem
Steps : 22
Nodes : 581
Exec Time : 21 ms

Menggunakan penyelesaian DFS-Travelling Salesman Problem
Steps : 52
Nodes : 55
Exec Time : 0 ms

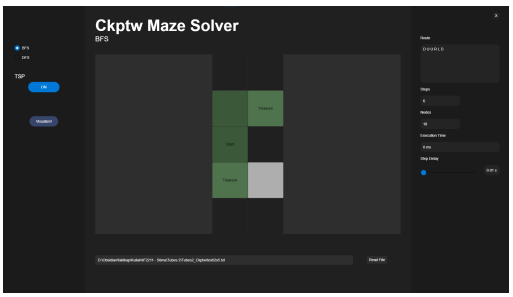
4.5.4. Kasus 4

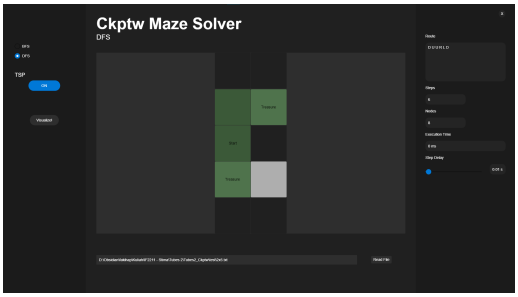


Menggunakan penyelesaian BFS-Travelling Salesman Problem
Steps : 32
Nodes : 8751
Exec Time : 167 ms

Menggunakan penyelesaian DFS-Travelling Salesman Problem
Steps : 42
Nodes : 45
Exec Time : 0 ms

4.5.5. Kasus 5





Menggunakan penyelesaian BFS-Travelling Salesman Problem
Steps : 6
Nodes : 18
Exec Time : 0 ms

Menggunakan penyelesaian DFS-Travelling Salesman Problem
Steps : 6
Nodes : 8
Exec Time : 0 ms

4.6. Analisis Desain Solusi Algoritma BFS dan DFS

4.6.1. Kasus 1

Dapat dilihat, solusi hasil pencarian menggunakan pendekatan BFS memiliki jumlah langkah diperlukan yang lebih sedikit daripada pendekatan DFS. Sebagai gantinya, kompleksitas ruang dan waktu yang digunakan oleh algoritma BFS jauh lebih banyak daripada kompleksitas yang dihasilkan DFS untuk *maze* yang sama.

4.6.2. Kasus 2

Meskipun kasus yang diberikan tergolong kecil dan memiliki kompleksitas rendah, kedua algoritma dapat memberikan keluaran yang berbeda, baik dari solusi maupun banyak langkah dan pengecekan node yang dilakukan. Pemilihan prioritas juga mungkin berpengaruh dalam penyelesaian kasus ini.

4.6.3. Kasus 3

Perbedaan pemilihan prioritas petak yang diambil (seperti URDL) juga memiliki dampak yang signifikan dalam pencarian solusi. Untuk *maze* yang sesuai, urutan prioritas tertentu akan memiliki performa yang lebih baik daripada urutan prioritas petak yang lain.

Dapat dilihat juga, jumlah simpul yang saling bertetanggaan dan jumlah tetangga untuk setiap petak yang semakin banyak memiliki dampak yang lebih besar terhadap pendekatan pencarian solusi BFS daripada algoritma DFS.

4.6.4. Kasus 4

Dapat dilihat melalui kasus ini, perbedaan kompleksitas *maze* sangat berpengaruh pada pencarian solusi menggunakan kedua pendekatan. Terlihat pula bahwa *maze* yang memiliki lebih banyak cabang untuk setiap petaknya akan menambah kompleksitas ruang dan waktu pendekatan BFS jauh lebih banyak daripada pendekatan DFS. Meskipun begitu, pendekatan BFS berhasil memberikan solusi yang memiliki langkah lebih sedikit daripada solusi DFS.

4.6.5. Kasus 5

Dapat dilihat, pengujian pada kasus kecil ini menghasilkan solusi yang sama untuk kedua algoritma namun dengan jumlah pengecekan simpul yang berbeda. Dapat diambil kesimpulan bahwa algoritma BFS akan selalu melakukan pengecekan sama dengan atau lebih banyak dari pengecekan yang dilakukan melalui pendekatan DFS, meskipun di kasus yang lebih kecil pun.

BAB 5
Kesimpulan dan Saran

5.1. Kesimpulan

Dari Tugas Besar 2 IF2211 Strategi Algoritma Semester II 2022/2023 ini kami menyimpulkan bahwa dalam menyelesaikan suatu maze BFS dan DFS memiliki karakteristiknya masing masing dalam memecahkan maze yang sudah diberikan dimana DFS menggunakan sistem stack dalam antrian yang digunakan untuk mencari treasure sedangkan BFS yang menggunakan queue dalam antrian yang digunakan dalam mencari treasure. BFS dapat lebih baik mencari jalan yang paling pendek dibanding dengan DFS karena BFS menggunakan queue dalam antriannya yang dimana BFS akan mengambil jalan yang pertama kali ditemukan dalam mencari seluruh treasure sedangkan dalam DFS hanya mengikuti antrian stack saja. Tetapi dalam kecepatan eksekusi dari program yang kami buat DFS lebih cepat ketimbang BFS karena BFS mencari segala arah dengan menggunakan antrian queue dan mencari jalan yang pertama ditemukan.

5.2. Saran

Kami memiliki beberapa saran untuk disampaikan berkaitan dengan Tugas Besar 2 IF2211 Strategi Algoritma Semester II 2022/2023, yaitu:

- Sepertinya perlu tutorial sedikit dalam membuat sebuah UI agar orang yang masih awam dalam membuat GUI bisa paham cara menggunakannya

5.3. Refleksi

Walau dari luar terlihat sederhana dan mudah, ternyata setelah dikerjakan baru terasa seberapa melatih pemahaman tentang algoritma yang digunakan dan menguras kreativitas dalam menerapkan pendekatan yang diambil, terutama berbagai pendekatan heuristik yang sangat memungkinkan program berjalan lebih efisien lagi. Sebaiknya jangan meremehkan apapun sebelum dicoba.

5.4. Tanggapan

Tugas besar ini bukan hanya menguji pemahaman algoritma BFS dan DFS, tetapi juga meminta mahasiswa untuk mengajukan solusi kreatif mereka untuk menyelesaikan masalah yang ditemui. Terlebih dari itu, tugas ini juga mendorong mahasiswa untuk melakukan eksplorasi lebih lanjut terutama mengenai pengembangan GUI dalam lingkungan C#.

Daftar Pustaka

- https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf
- https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf
- http://bryukh.com/labyrinth-algorithms/
- https://reference.avaloniaui.net/api/

Lampiran

- Link repository: https://github.com/fchrgrib/Tubes2_Ckptw
- Link YouTube: https://youtu.be/CUL83StiE98