

IF2211 Strategi Algoritma

**IMPLEMENTASI ALGORITMA BFS DAN DFS DALAM
APLIKASI PEMECAH LABIRIN**

Laporan Tugas Besar 2

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma
pada Semester 2 (satu) Tahun Akademik 2022/2023



Oleh

Raditya Naufal Abiyu **13521022**

Muhammad Aji Wibisono **13521095**

Arsa Izdihar Islam **13521101**

Kelompok Spongbob

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

DAFTAR ISI

DAFTAR ISI	2
DAFTAR GAMBAR	4
DAFTAR TABEL	5
BAB 1	
DESKRIPSI TUGAS	1
1.1 Deskripsi Tugas	1
1.2 Spesifikasi Tugas	3
1.2.1 Spesifikasi GUI	4
1.2.2 Spesifikasi Wajib	5
BAB II	
LANDASAN TEORI	6
2.1 Algoritma BFS	6
2.2 Algoritma DFS	7
2.3 Travelling Salesman Problem	8
2.3 C# Desktop Application development	9
2.4 Avalonia UI framework	9
BAB III	
APLIKASI ALGORITMA	10
3.1 Pemecahan Masalah	10
3.2 Aplikasi Algoritma BFS/DFS	11
3.3 Kasus lain	13
BAB IV	
ANALISIS PEMECAHAN MASALAH	15
4.1 Implementasi Program	15
4.1.1 Loop Utama	15
4.1.2 Langkah BFS	16
4.1.2 Langkah DFS	18
4.2 Struktur Data	20
4.2.1 Algorithm.cs	21
4.2.2 Map.cs	23
4.2.3 Graph.cs	23
4.2.4 Parser.cs	24
4.2.5 Result.cs	24
4.2.6 BFS.cs	25
4.2.7 DFS.cs	25
4.3 Pengujian Program	26
4.3.1 sampel-1.txt	26
4.3.2 sampel-2.txt	27
4.3.3 sampel-3.txt	29

4.3.4 sampel-4.txt	29
4.3.5 sampel-5.txt	31
4.4 Analisis Desain	32
BAB 5	
KESIMPULAN DAN SARAN	33
5.1 Kesimpulan	33
5.2 Saran	33
5.3 Refleksi	34
5.4 Tanggapan	34
DAFTAR PUSTAKA	35
LAMPIRAN	36

DAFTAR GAMBAR

Gambar 1.1.1 Contoh pembacaan file	1
Gambar 1.1.2 Contoh input program	2
Gambar 1.1.3 Contoh output program	3
Gambar 1.2.1 Tampilan program sebelum dicari solusinya	4
Gambar 1.2.2 Tampilan program setelah dicari solusinya	4
Gambar 2.1.1 Ilustrasi pencarian menggunakan BFS	6
Gambar 2.2.1 Ilustrasi pencarian menggunakan DFS	7
Gambar 3.3.1 Ilustrasi kasus lain dengan map berukuran kecil	13
Gambar 3.3.2 Ilustrasi kasus lain dengan map berukuran sedang	13
Gambar 3.3.3 Ilustrasi kasus lain dengan map berukuran besar	14

DAFTAR TABEL

Tabel 3.1.1 Elemen pada BFS/DFS	11
Tabel 4.2.1.1 Atribut dan kelas pada file Algorithm.cs	21
Tabel 4.2.2.1 Atribut dan kelas pada file Map.cs	23
Tabel 4.2.3.1 Atribut dan kelas pada file Graph.cs	23
Tabel 4.2.4.1 Atribut dan kelas pada file Parser.cs	24
Tabel 4.2.5.1 Atribut dan kelas pada file Result.cs	24
Tabel 4.3.1.1 Hasil pengujian program pada file sampel-1.txt	26
Tabel 4.3.2.1 Hasil pengujian program pada file sampel-2.txt	27
Tabel 4.3.3.1 Hasil pengujian program pada file sampel-3.txt	29
Tabel 4.3.4.1 Hasil pengujian program pada file sampel-4.txt	29
Tabel 4.3.5.1 Hasil pengujian program pada file sampel-5.txt	31

BAB 1

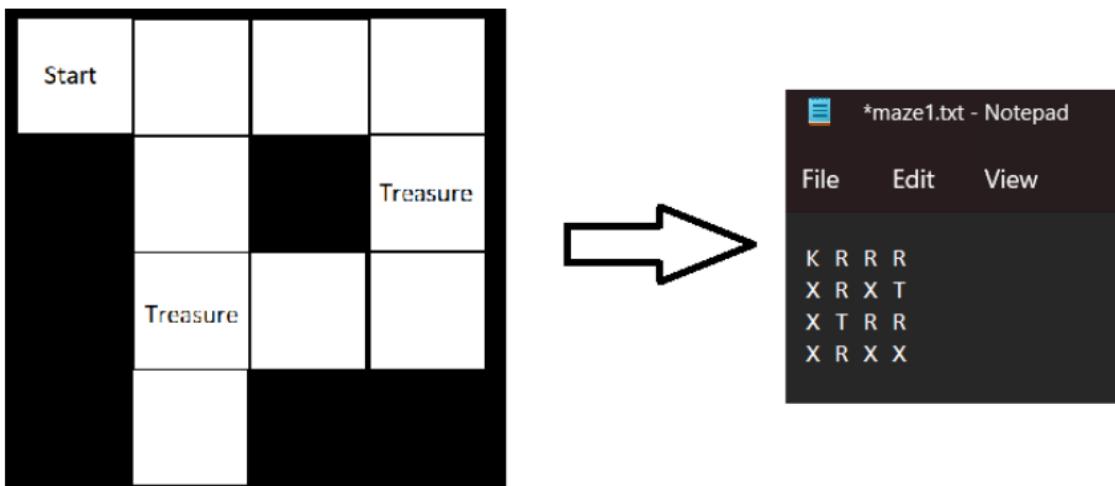
DESKRIPSI TUGAS

1.1 Deskripsi Tugas

Dalam tugas besar ini, diminta untuk membuat 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. 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 pembacaan file:

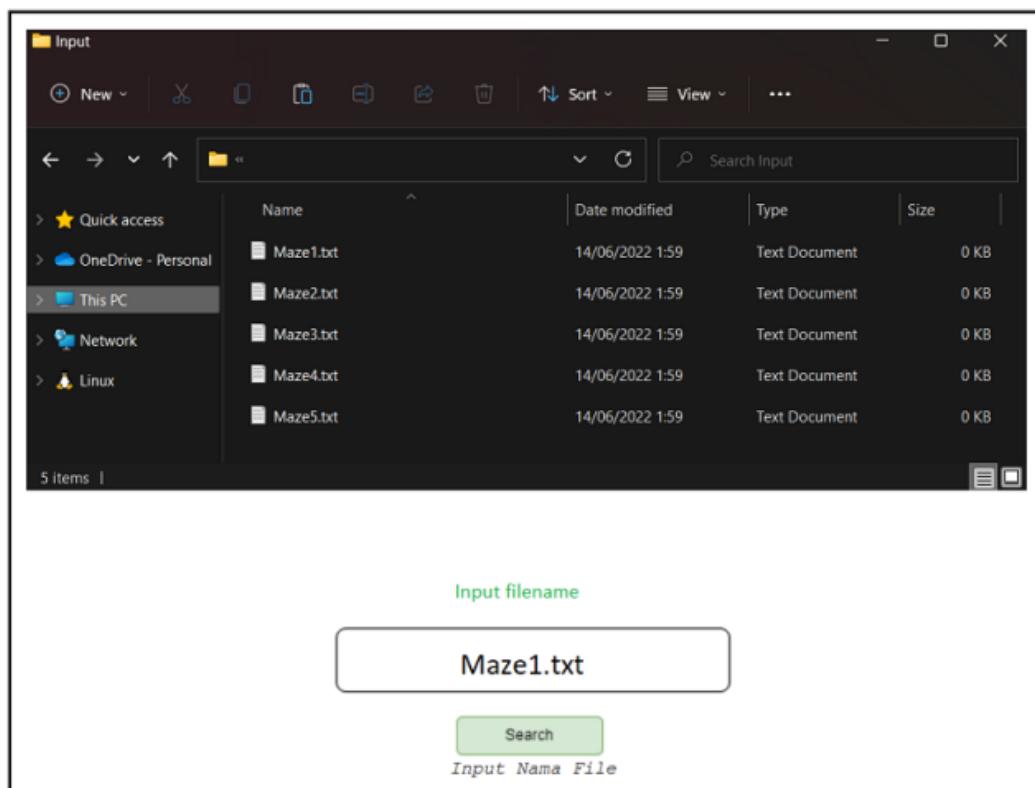


Gambar 1.1.1 Contoh pembacaan file

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), program 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. Program juga

perlu memvisualisasikan input file 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.

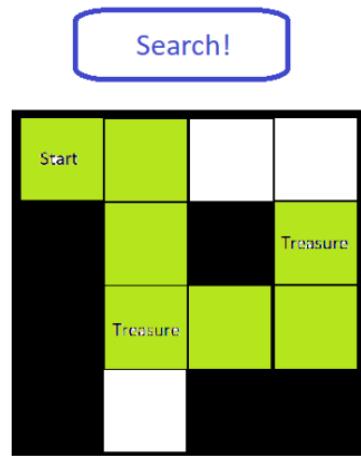
Contoh input program:



Gambar 1.1.2 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. 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 program:

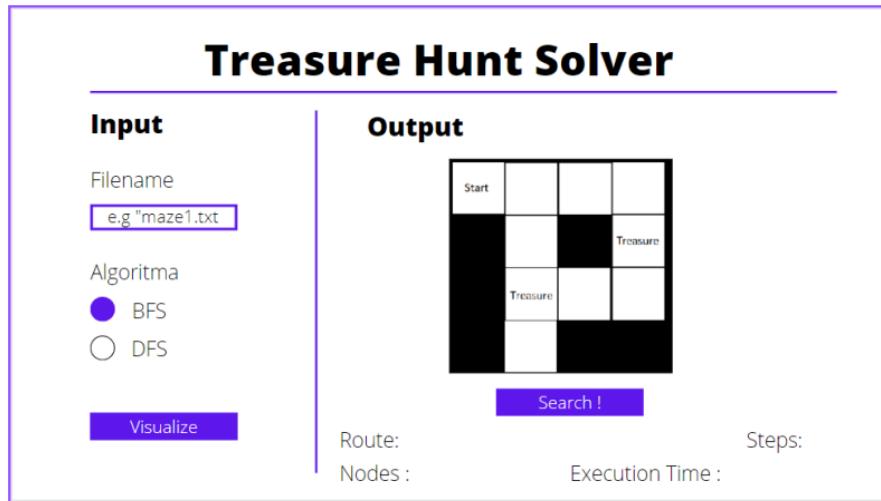


Gambar 1.1.3 Contoh output program

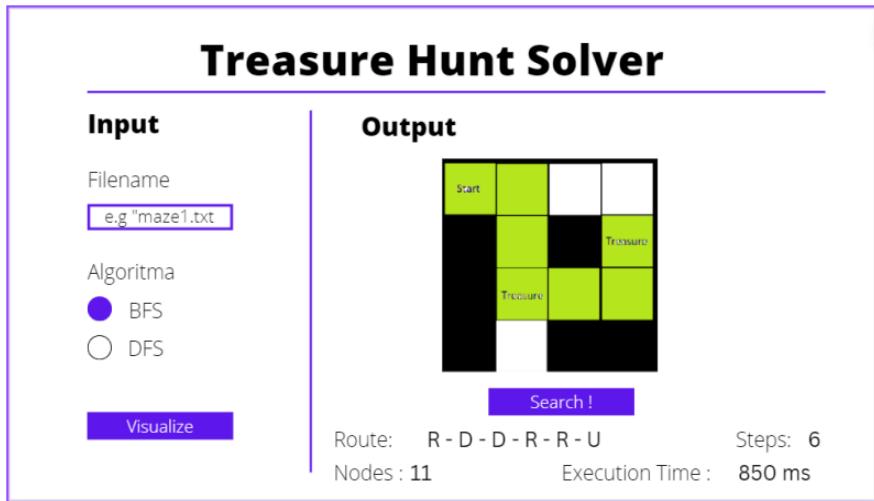
Setelah program melakukan pembacaan input, program akan memvisualisasikan gridnya terlebih dahulu tanpa pemberian rute solusi. Hal tersebut dilakukan agar pengguna dapat mengerjakan terlebih dulu *treasure hunt* secara manual jika diinginkan. Kemudian, program menyediakan tombol solve untuk mengeksekusi algoritma DFS dan BFS. Setelah tombol diklik, program akan melakukan pemberian warna pada rute solusi.

1.2 Spesifikasi Tugas

Aplikasi yang akan dibangun dibuat berbasis GUI. Berikut ini adalah contoh tampilan dari aplikasi GUI yang akan dibangun:



Gambar 1.2.1 Tampilan program sebelum dicari solusinya



Gambar 1.2.2 Tampilan program setelah dicari solusinya

1.2.1 Spesifikasi GUI

Design layout program dibebaskan dengan syarat mengandung seluruh input dan output yang terdapat pada spesifikasi, yakni

1. Masukan program adalah file maze treasure hunt tersebut atau nama filenya.
2. Program dapat menampilkan visualisasi dari input file maze dalam bentuk grid dan pewarnaan sesuai deskripsi tugas.
3. Program memiliki toggle untuk menggunakan alternatif algoritma BFS ataupun DFS.

4. Program memiliki tombol search yang dapat mengeksekusi pencarian rute dengan algoritma yang bersesuaian, kemudian memberikan warna kepada rute solusi output.
5. Luaran program adalah banyaknya node (grid) yang diperiksa, banyaknya langkah, rute solusinya, dan waktu eksekusi algoritma.
6. (Bonus) Program dapat menampilkan progress pencarian grid dengan algoritma yang bersesuaian. Hal tersebut dilakukan dengan memberikan slider / input box untuk menerima durasi jeda tiap step, kemudian memberikan warna kuning untuk tiap grid yang sudah diperiksa dan biru untuk grid yang sedang diperiksa.
7. (Bonus) Program membuat toggle tambahan untuk persoalan TSP. Jadi apabila toggle dinyalakan, rute solusi yang diperoleh juga harus kembali ke titik awal setelah menemukan segala harta karunnya (Tetap dengan algoritma BFS atau DFS).
8. GUI dapat dibuat sekreatif mungkin asalkan memuat 5 (7 jika mengerjakan bonus) spesifikasi di atas.

1.2.2 Spesifikasi Wajib

Ada pula program yang dibuat harus memenuhi spesifikasi wajib sebagai berikut:

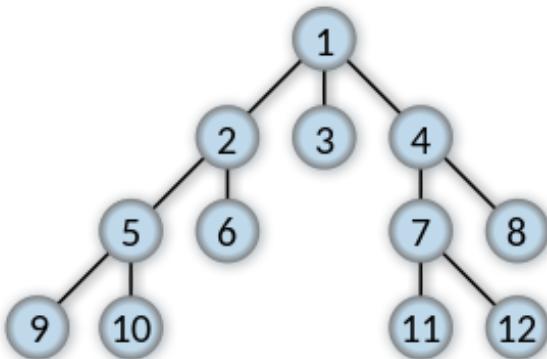
1. Buatlah program dalam bahasa C# untuk mengimplementasi Treasure Hunt Solver sehingga diperoleh output yang diinginkan. Penelusuran harus memanfaatkan algoritma BFS dan DFS.
2. Awalnya program menerima file atau nama file maze hunt.
3. Apabila filename tersebut ada, Program akan melakukan validasi dari file input tersebut. Validasi dilakukan dengan memeriksa apakah tiap komponen input hanya berupa K, T, R, X. Apabila validasi gagal, program akan memunculkan pesan bahwa file tidak valid. Apabila validasi berhasil, program akan menampilkan visualisasi awal dari maze treasure hunt.
4. Pengguna memilih algoritma yang digunakan menggunakan toggle yang tersedia.
5. Program kemudian dapat menampilkan visualisasi akhir dari maze (dengan pewarnaan rute solusi).
6. Program menampilkan luaran berupa durasi eksekusi, rute solusi, banyaknya langkah, serta banyaknya node yang diperiksa.

BAB II

LANDASAN TEORI

2.1 Algoritma BFS

Algoritma *Breadth First Search* (BFS) merupakan algoritma traversal yang digunakan untuk melintasi atau mencari semua simpul atau node dari suatu struktur pohon atau graf. Algoritma ini melakukan pencarian dengan melintasi graf mulai pada simpul yang paling dekat ke akar lalu merambat ke simpul - simpul yang memiliki kedalaman lebih tinggi.



Gambar 2.1.1 Ilustrasi pencarian menggunakan BFS

sumber: https://en.wikipedia.org/wiki/Breadth-first_search

Diakses pada 22 maret 2023 pukul 17.34 WIB

Cara kerja algoritma Breadth First Search adalah dengan memasukan node akar ke dalam sebuah antrian. Kemudian mengambil simpul pertama pada level paling dekat ke akar, jika simpul merupakan solusi pencarian selesai dan hasil dikembalikan. Jika simpul bukan merupakan solusi, masukan seluruh simpul yang bertetangga dengan simpul tersebut ke dalam antrian. Apabila semua simpul telah diperiksa dan antrian kosong, pencarian selesai dengan mengembalikan hasil solusi tidak ditemukan.

Algoritma *Breadth First Search* memiliki kelebihan sebagai berikut:

- BFS menjamin menemukan solusi terpendek karena mencari jalur terpendek dari simpul awal ke simpul tujuan.
- BFS lebih cocok digunakan pada graf dengan banyak jalur bercabang dan memiliki solusi.
- BFS dapat diterapkan pada graf dengan kedalaman yang sangat besar tanpa risiko terjebak dalam loop tak terbatas.

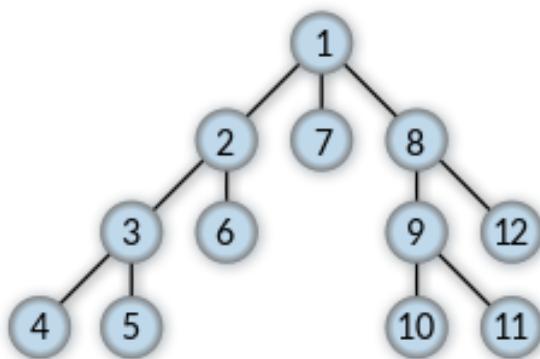
- Solusi yang ditemukan oleh BFS akan selalu optimal dan optimalitas solusi dapat dihitung secara matematis.

Algoritma *Breadth First Search* memiliki kekurangan sebagai berikut:

- BFS membutuhkan lebih banyak memori daripada DFS karena membutuhkan struktur data queue.
- BFS dapat menjadi lambat jika graf memiliki kedalaman yang sangat besar atau memiliki banyak jalur bercabang.
- BFS tidak cocok untuk graf yang memiliki jalur panjang yang tidak memiliki solusi, karena BFS akan menelusuri seluruh jalur hingga mencapai simpul tujuan.

2.2 Algoritma DFS

Algoritma *Depth First Search* (DFS) merupakan algoritma traversal yang digunakan untuk melintasi atau mencari semua simpul atau node dari suatu struktur pohon atau graf. Algoritma ini memiliki perbedaan terhadap algoritma *Breadth First Search* pada prioritas urutan pencarian. Algoritma *Depth First Search* melakukan pencarian dengan mendahulukan titik yang diketahui oleh pencari yang memiliki kedalaman paling tinggi terlebih dahulu.



Gambar 2.2.1 Ilustrasi pencarian menggunakan DFS

sumber: https://en.wikipedia.org/wiki/Depth-first_search

Diakses pada 22 maret 2023 pukul 17.35 WIB

Cara kerja algoritma Depth First Search adalah dengan memasukan node akar ke dalam sebuah tumpukan. Kemudian mengambil simpul pertama pada level paling dekat ke akar, jika simpul merupakan solusi pencarian selesai dan hasil dikembalikan. Jika simpul bukan merupakan solusi, masukan seluruh

simpul yang bertetangga dengan simpul tersebut ke dalam tumpukan. Apabila semua simpul telah diperiksa dan tumpukan kosong, pencarian selesai dengan mengembalikan hasil solusi tidak ditemukan.

Algoritma *Depth First Search* memiliki kelebihan sebagai berikut:

- DFS lebih sederhana dan mudah dipahami dibandingkan dengan algoritma pencarian lainnya.
- DFS cocok untuk memecahkan masalah dengan solusi yang bersifat rekursif.
- DFS dapat menemukan solusi secara cepat jika solusi terletak pada cabang pertama dari simpul awal.
- DFS cocok digunakan pada graf dengan ukuran yang relatif kecil dan cabang yang dalam.

Algoritma *Depth First Search* memiliki kekurangan sebagai berikut:

- DFS tidak menjamin menemukan solusi terpendek, karena algoritma cenderung menelusuri cabang terdalam terlebih dahulu.
- DFS dapat mengalami masalah stack overflow jika mencoba menelusuri graf dengan kedalaman yang sangat besar.
- DFS dapat menemukan jalur yang tidak ditemukan oleh BFS, terutama pada graf yang memiliki banyak jalur bercabang dan tidak memiliki solusi.
- DFS dapat terjebak dalam loop tak terbatas jika graf memiliki siklus.

2.3 Travelling Salesman Problem

Travelling Salesman Problem (TSP) adalah sebuah permasalahan yang ada pada optimasi. Secara umum berikut adalah karakteristik dari permasalahan TSP:

1. Perjalanan berawal dan berakhir dari dan ke kota awal
2. Ada sejumlah kota yang semuanya harus dikunjungi tepat satu kali
3. Perjalanan tidak boleh kembali ke kota awal sebelum semua kota tujuan dikunjungi
4. Tujuan dari permasalahan ini adalah meminimumkan total jarak yang ditempuh salesman dengan mengatur urut-urutan kota yang harus dikunjungi

Pada program ini dibuat sebuah implementasi yang menyelesaikan permasalahan TSP namun dengan beberapa penyederhanaan. Pada implementasi program ini, TSP hanya diimplementasikan untuk berawal dan berakhir pada petak yang sama, aspek lain dari TSP tidak diimplementasikan.

2.3 C# Desktop Application development

C# Desktop Application development adalah proses membuat aplikasi desktop menggunakan bahasa pemrograman C# (C Sharp) dan framework .NET. C# adalah bahasa pemrograman yang modern, aman, dan efisien yang digunakan untuk membuat aplikasi desktop, aplikasi web, dan aplikasi seluler. .NET Framework adalah platform yang menyediakan berbagai fitur dan library yang diperlukan untuk membangun aplikasi desktop, seperti Windows Forms dan WPF (Windows Presentation Foundation).

Dalam proses C# Desktop Application development, seorang pengembang biasanya akan menggunakan IDE (Integrated Development Environment) seperti Visual Studio untuk membangun aplikasi desktop mereka. IDE menyediakan berbagai alat untuk membantu pengembang dalam proses pengkodean, debugging, dan deployment aplikasi desktop mereka. C# Desktop Application development dapat mencakup berbagai macam fitur, seperti database, networking, multimedia, dan UI (user interface) yang menarik. Dengan C# Desktop Application development, pengembang dapat membuat aplikasi desktop yang kuat dan mudah dioperasikan, yang dapat digunakan untuk berbagai macam tujuan, seperti bisnis, pendidikan, dan hiburan.

2.4 Avalonia UI framework

Avalonia UI Framework adalah sebuah framework untuk membangun aplikasi desktop cross-platform yang modern dan responsif dengan menggunakan bahasa pemrograman C# dan XAML. Framework ini dibangun di atas teknologi .NET dan mendukung sistem operasi Windows, Linux, dan macOS. Dibandingkan dengan framework UI desktop lainnya, Avalonia UI Framework menawarkan performa yang lebih baik dan tampilan yang lebih cantik.

Dalam penggunaannya, Avalonia UI Framework menyediakan berbagai macam kontrol UI yang dapat digunakan untuk membuat aplikasi desktop seperti tombol, kotak teks, panel, dan banyak lagi. Pengembang dapat menggunakan XAML untuk mendefinisikan antarmuka pengguna dan C# untuk logika aplikasi. Framework ini juga menyediakan dukungan untuk animasi, pengaturan tata letak responsif, dan integrasi dengan sistem operasi, seperti notifikasi dan integrasi dengan file explorer. Dengan Avalonia UI Framework, pengembang dapat membuat aplikasi desktop yang modern dan responsif dengan mudah dan cepat.

BAB III

APLIKASI ALGORITMA

3.1 Pemecahan Masalah

Definisi Dari *breadth first search* (BFS) dan *depth first search* (DFS) pada bab 2 menggambarkan kedua algoritma tersebut dengan sangat akurat. Namun dalam kasus seperti tugas besar ini, tidak dapat digunakan algoritma BFS dan DFS secara sederhana. Dalam tugas besar ini, objektifnya adalah mendapatkan semua harta karun yang tersebar di peta tidak seperti kegunaan BFS dan DFS yang awalnya digunakan untuk mencari rute dari titik awal ke titik akhir yang sudah ditentukan.

Algoritma pemecahan masalah yang kami gunakan memiliki sedikit modifikasi dari algoritma BFS dan DFS tradisional. Modifikasi ini disebabkan karena titik akhir dari rute yang diambil tidak statis melainkan bisa berubah sesuai urutan pengambilan harta karun. Dalam kasus ini, titik akhir yang digunakan adalah harta karun yang terakhir dikunjungi.

Proses pencarian secara BFS dan DFS, keduanya dimulai pada simbol K pada peta yang menandakan “start point” kemudian pencarian dilakukan secara BFS atau DFS hingga mencapai harta karun pertama. Setelah mendapatkan harta karun, apabila jumlah harta karun tidak sama dengan total harta karun yang ada di peta maka pencarian akan dilanjutkan dengan membuat harta karun yang baru saja ditemukan menjadi titik awal pencarian dan titik akhir adalah harta karun lain. Proses pencarian ini akan terus dilakukan selama jumlah harta karun yang didapatkan tidak sama dengan harta karun total atau hingga semua petak yang bisa dijelajahi sudah habis.

Dalam implementasinya, berikut adalah langkah - langkah pemecahan masalah yang dilakukan untuk menyelesaikan tugas ini:

1. Memahami konsep dari algoritma DFS dan BFS
2. Mempelajari syntax bahasa pemrograman C# dan cara kerja Avalonia UI Framework
3. Memetakan input menjadi elemen - elemen pada BFS dan DFS
4. Mendesain kelas dan algoritma yang diperlukan untuk menyelesaikan permasalahan
5. Mengimplementasikan algoritma BFS dan DFS pada pencarian solusi di setiap elemen
6. Membentuk UI design menggunakan figma
7. Membuat Graphical User Interface (GUI) menggunakan Avalonia UI Framework pada Visual Studio
8. Menghubungkan seluruh program dengan GUI yang sudah dibuat

Pada kasus ini, input permasalahan dipetakan menjadi elemen - elemen pada BFS dan DFS sebagai berikut:

Tabel 3.1.1 Elemen pada BFS/DFS

Nodes	Petak yang tidak berisi X
Edges	Petak kosong yang bertetanggaan dengan petak yang ditempati saat ini
Goal State	Semua <i>treasure</i> diambil untuk non-TSP, semua <i>treasure</i> diambil dan sudah kembali ke start untuk TSP

Program kami memiliki karakteristik pencarian yang memiliki prioritas “up”, “right”, “down”, dan terakhir “left”. Setelah proses pencarian selesai maka akan ditunjukkan rute yang telah didapatkan. Rute tersebut disimbolkan dengan warna pink di dalam grid peta yang terbentuk. Apabila program memvisualisasikan proses pencarinya, maka petak berwarna pink merupakan *current route* dan petak berwarna ungu gelap merupakan *visited tile* sedangkan petak berwarna kuning adalah harta karun.

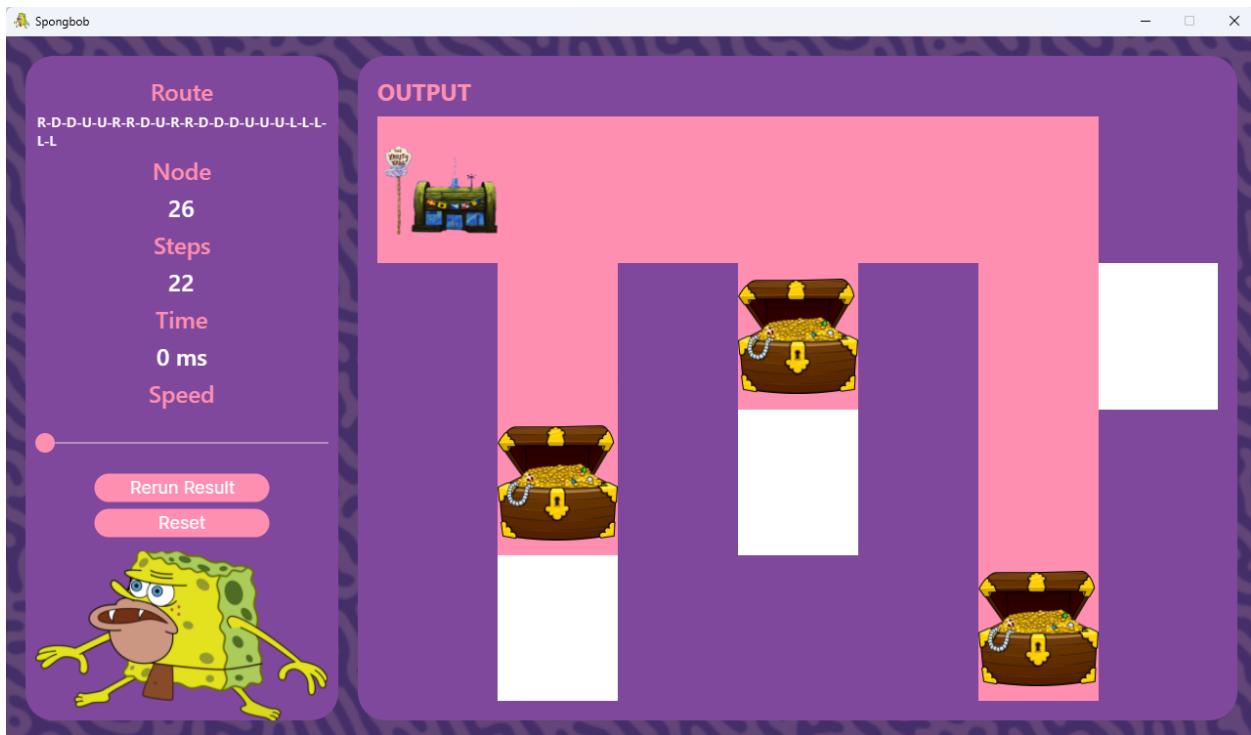
3.2 Aplikasi Algoritma BFS/DFS

Pada implementasi program, algoritma BFS/DFS diaplikasikan dengan beberapa tahapan sebagai berikut:

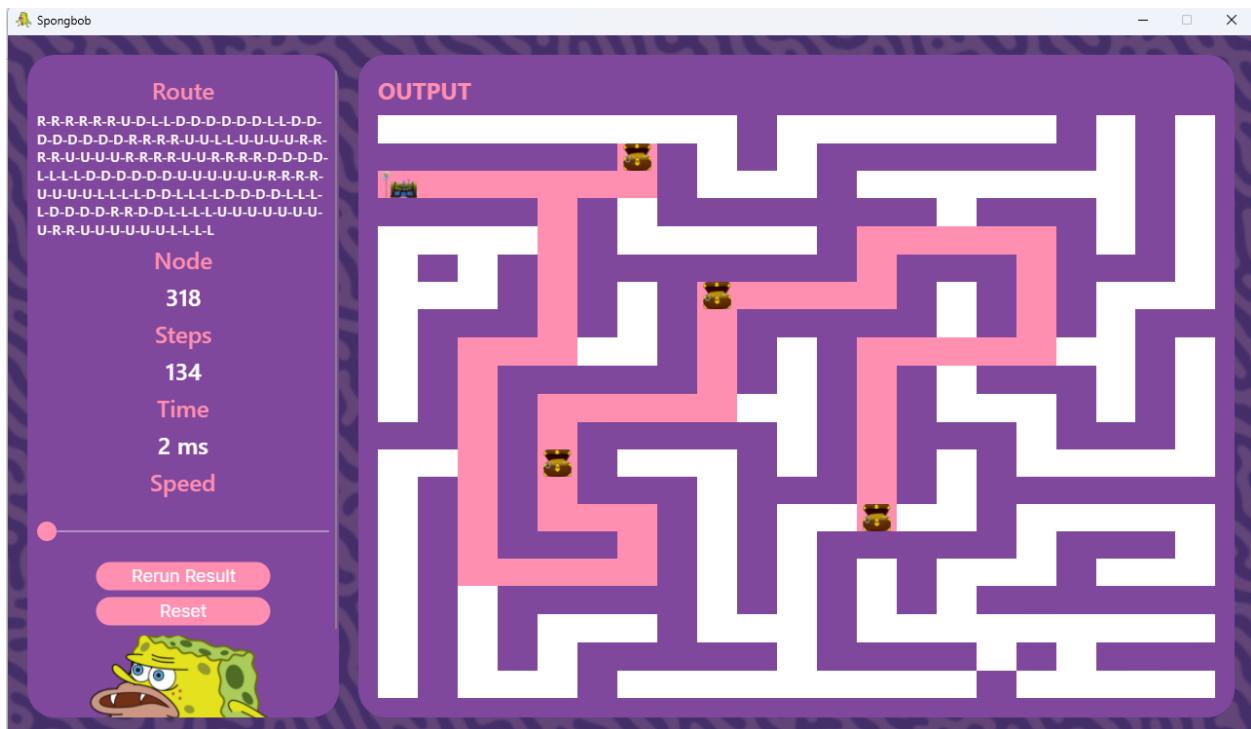
1. Peta pada file txt dibaca dengan tiap petak yang bukan X dijadikan sebagai simpul (*Node*) dengan tipe data graf yang memiliki atribut berupa graf yang bertetanggaan dengannya. Petak yang memiliki tanda T dan K akan memiliki atribut penanda *treasure* dan *Start*. Pada file juga akan dibaca jumlah *treasure* yang terdapat pada peta tersebut.
2. Program akan melakukan traversal dimulai dari petak K, lalu rute menuju petak - petak yang bertetanggaan akan dimasukkan sebagai sudut (*Edge*) ke queue dan stack tergantung pada pencarian BFS atau DFS. Rute tersebut ditandai dengan menambahkan karakter ke string id dengan 0 bernilai atas, 1 bernilai kanan, 2 bernilai bawah, dan 3 bernilai kiri
3. Perpindahan antar petak dilakukan dengan menambahkan karakter ke sebuah string id yang menyimpan rute perjalanan. Perpindahan dilakukan dengan prioritas atas, kanan, bawah, lalu kiri. Perpindahan akan mengecek terlebih dahulu apakah petak tersebut sudah dikunjungi atau belum. Jika petak tersebut sudah dikunjungi, program akan melewati petak tersebut dan tidak melakukan perpindahan.
4. Setelah dilakukan perpindahan, program akan menandai petak tersebut dengan status terkunjungi.

5. Jika ditemukan *treasure* maka penghitung jumlah *treasure* akan ditambahkan dengan nilai satu lalu semua rute tujuan yang terdapat pada stack atau queue akan direfaktor dengan ditambahkan rute tambahan dari *treasure* sampai ke percabangan.
6. Program akan memprioritaskan rute dengan pemutaran balik sesedikit mungkin sehingga semua rute yang didapatkan sebelumnya akan dipindahkan ke queue atau stack lain dengan prioritas lebih rendah. Queue atau stack dengan prioritas lebih rendah ini tidak akan diakses sampai queue atau stack utama kosong (pencarian buntu). Ketika pencarian buntu, maka queue atau stack akan dipindahkan seluruhnya ke queue atau stack utama.
7. Ketika petak selanjutnya yang berada pada stack atau queue tidak bersebelahan dengan letak pencarian saat ini, akan dilakukan backtracking dengan menyesuaikan id dengan rute yang dituju dan melakukan perubahan per langkah sampai didapat petak tersebut bersebelahan.
8. Visualisasi dilakukan dengan melihat state TileView yang dimiliki petak dan perubahan berdasarkan perubahan tempat dari letak pencarian saat ini menggunakan fungsi callback untuk setiap perpindahan petak.
9. Program akan mengetahui kelengkapan *treasure* dengan membandingkan jumlah *treasure* yang sudah didapatkan dengan jumlah *treasure* yang didapatkan ketika membaca file txt.
10. Algoritma tersebut akan diulangi terus menerus sampai kondisi sukses (*Goal*) dipenuhi, yakni semua *treasure* sudah didapatkan untuk non-TSP dan semua *treasure* sudah didapatkan dan pencari petak sudah kembali ke petak *start*, atau jika semua petak yang dapat dijelajahi sudah dikunjungi namun *treasure* belum terpenuhi.

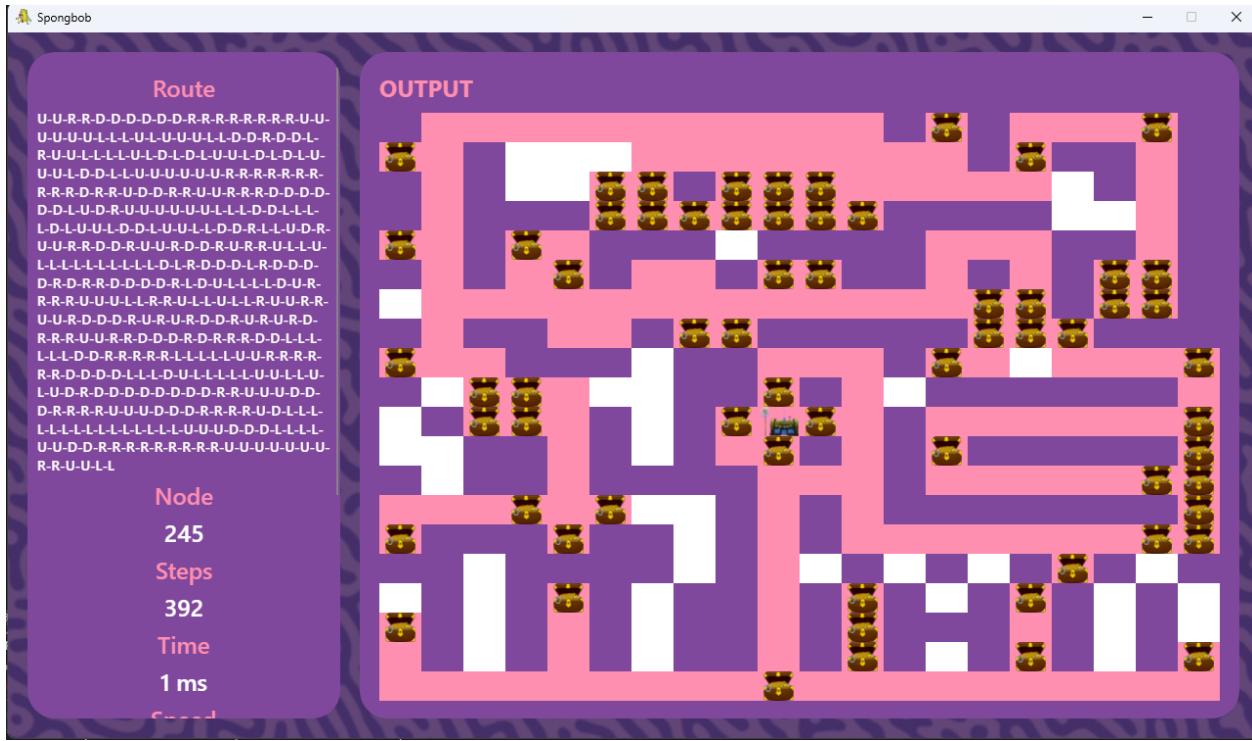
3.3 Kasus lain



Gambar 3.3.1 Ilustrasi kasus lain dengan map berukuran kecil (BFS TSP)



Gambar 3.3.2 Ilustrasi kasus lain dengan map berukuran sedang (DFS TSP)



Gambar 3.3.3 Ilustrasi kasus lain dengan map berukuran besar (DFS TSP)

BAB IV

ANALISIS PEMECAHAN MASALAH

4.1 Implementasi Program

4.1.1 Loop Utama

Run() pada Algoritma.cs

```
function Run() -> Result:
    res <- new Result(map.Width, map.Height)
    id <- ""
    watch <- new Stopwatch()
    watch.Start()
    initialize()
    step <- Next(id)
    id <- step.Item1

    while not IsDone:
        try:
            step <- Next(id)
        except:
            watch.Stop()
            res.Time <- watch.ElapsedMilliseconds
            res.Found <- false
            return res

        id <- step.Item1

        if step.Item2:
            res.NodesCount <- res.NodesCount + 1

        watch.Stop()
        res.Time <- watch.ElapsedMilliseconds
        GetResult(res, id)

    return res
```

4.1.2 Langkah BFS

Next(string) pada BFS.cs

```
function Next(previous: string) -> Tuple<string, bool>:  
    int loc <- 1  
    graphsprio1.TryPeek(out var el)  
    if (el = null):  
        int n <- graphsprio2.Count()  
        for (int i <- 0; i < n; i++):  
            graphsprio1.Enqueue(graphsprio2.ElementAt(i))  
        graphsprio2.Clear()  
        graphsprio1.TryPeek(out el)  
  
        if (el = null):  
            loc <- 2  
            stucks.TryPeek(out el)  
        if (el = null):  
            throw new Exception("No solution")  
  
        if (IsBack ? el.Item2.BackStates = TileState.Visited : el.Item2.States =  
TileState.Visited):  
            switch (loc):  
                case 1:  
                    graphsprio1.TryDequeue(out el)  
                    break  
                case 2:  
                    stucks.TryDequeue(out el)  
                    break  
            return Next(previous)  
  
        string id <- el.Item1  
  
        switch (loc):  
            case 1:  
                graphsprio1.TryDequeue(out el)  
                break  
            case 2:  
                stucks.TryDequeue(out el)  
                break  
  
        Graph tile <- el!.Item2  
  
        if (IsBack):  
            tile.BackStates <- TileState.Visited  
        else:  
            tile.States <- TileState.Visited
```

```

if (!IsBack && tile.IsTreasure):
    int len1 <- graphsprio1.Count()
    int len2 <- graphsprio2.Count()

    for (int i <- 0; i < len2; i++):
        graphsprio2.TryDequeue(out var duplicate)
        graphsprio2.Enqueue(RefactorRoute(duplicate!, id))
    for (int i <- 0; i < len1; i++):
        graphsprio2.Enqueue(RefactorRoute(graphsprio1.ElementAt(i), id))
    graphsprio1.Clear()

    treasureCounts++

    if (treasureCounts = map.TreasuresCount):
        isTreasureDone <- true
        if (isTSP):
            graphsprio1.Clear()
            graphsprio2.Clear()
            graphsprio1.Enqueue(new Tuple<string, Graph>(id, tile))
        return new Tuple<string, bool>(id, true)

    else if (IsBack && tile = map.Start):
        isTSPDone <- true
        return new Tuple<string, bool>(id, true)

List<Tuple<Graph?, int>> neighborsData <- new()
for (int i <- 0; i < tile.Neighbors.Length; i++):
    Graph? candidate <- tile.Neighbors[i]
    if (candidate = null):
        continue
    if (IsBack && candidate?.BackStates = TileState.Visited):
        continue
    if (IsBack && candidate = map.Start):
        neighborsData.Add(new Tuple<Graph?, int>(candidate, i))
    if (candidate?.States != TileState.Visited):
        neighborsData.Add(new Tuple<Graph?, int>(candidate, i))

    int neighborsCount <- neighborsData.Count

    if (IsBack):
        List<Tuple<Graph?, int>> neighborsStuckData <- new()
        for (int i <- tile.Neighbors.Length - 1 i >= 0 i--):
            Graph? candidate <- tile.Neighbors[i]
            if (candidate != null &&
                candidate?.BackStates != TileState.Visited &&

```

```

        candidate?.States = TileState.Visited
    ) :
    neighborsStuckData.Add(new Tuple<Graph?, int>(tile.Neighbors[i], i))

    neighborsCount += neighborsStuckData.Count

    AddToQueue(stucks, neighborsStuckData, id, neighborsCount)

    AddToQueue(graphsprio1, neighborsData, id, neighborsCount)

    return new Tuple<string, bool>(id, true)

```

4.1.2 Langkah DFS

Next(string) pada DFS.cs

```

function Next(previous: string) -> Tuple<string, bool>:
    int loc <- 1
    graphsprio1.TryPeek(out var el)
    if (el = null):
        int n <- graphsprio2.Count()
        for (int i <- n - 1 i >= 0 i--):
            graphsprio1.Push(graphsprio2.ElementAt(i))
        graphsprio2.Clear()
        graphsprio1.TryPeek(out el)

    if (el = null):
        loc <- 2
        stucks.TryPeek(out el)
    if (el = null):
        throw new Exception("No solution")

    if (IsBack ? el.Item2.BackStates = TileState.Visited : el.Item2.States =
TileState.Visited):
        switch (loc):
            case 1:
                graphsprio1.TryPop(out el)
                break
            case 2:
                stucks.TryPop(out el)
                break
        return Next(previous)

    string id <- el.Item1

```

```

switch (loc):
    case 1:
        graphsprior1.TryPop(out el)
        break
    case 2:
        stucks.TryPop(out el)
        break

Graph tile <- el!.Item2

if (IsBack):
    tile.BackStates <- TileState.Visited
else:
    tile.States <- TileState.Visited

if (!IsBack && tile.IsTreasure):
    int len1 <- graphsprior1.Count()
    int len2 <- graphsprior2.Count()

    for (int i <- 0 i < len2 i++):
        graphsprior2.TryPop(out var duplicate)
        stucks.Push(RefactorRoute(duplicate!, id))
    for (int i <- 0 i < len2 i++):
        stucks.TryPop(out var duplicate)
        graphsprior2.Push(duplicate!)

    for (int i <- len1 - 1 i >= 0 i--):
        graphsprior2.Push(RefactorRoute(graphsprior1.ElementAt(i), id))
    graphsprior1.Clear()

treasureCounts++

if (treasureCounts = map.TreasuresCount):
    isTreasureDone <- true
    if (isTSP):
        graphsprior1.Clear()
        graphsprior2.Clear()
        graphsprior1.Push(new Tuple<string, Graph>(id, tile))
        return new Tuple<string, bool>(id, true)

    else if (IsBack && tile = map.Start):
        isTSPDone <- true
        return new Tuple<string, bool>(id, true)

List<Tuple<Graph?, int>> neighborsData <- new()

```

```

for (int i <- tile.Neighbors.Length - 1 i >= 0 i--):
    Graph? candidate <- tile.Neighbors[i]
    if (candidate = null):
        continue
    if (IsBack && candidate?.BackStates = TileState.Visited):
        continue
    if (IsBack && candidate = map.Start):
        neighborsData.Add(new Tuple<Graph?, int>(candidate, i))
    if (candidate?.States != TileState.Visited):
        neighborsData.Add(new Tuple<Graph?, int>(candidate, i))

int neighborsCount <- neighborsData.Count

if (IsBack):
    List<Tuple<Graph?, int>> neighborsStuckData <- new()
    for (int i <- tile.Neighbors.Length - 1 i >= 0 i--):
        Graph? candidate <- tile.Neighbors[i]
        if (candidate != null &&
            candidate?.BackStates != TileState.Visited &&
            candidate?.States = TileState.Visited
        ):
            neighborsStuckData.Add(new Tuple<Graph?, int>(tile.Neighbors[i], i))

neighborsCount += neighborsStuckData.Count

AddToStack(stucks, neighborsStuckData, id, neighborsCount)

AddToStack(graphspriol, neighborsData, id, neighborsCount)

return new Tuple<string, bool>(id, true)

```

4.2 Struktur Data

Dalam tugas besar ini, terdapat folder utama yaitu src dan di algoritma terdapat pada folder Models. Berikut merupakan rincian serta deskripsi (atribut dan metode) dari setiap file dalam folder tersebut:

```
.
|____ README.md
|____ .gitignore
|____ bin
|____ test
|____ doc
```

```

|   src
|   |   Spongbob.sln
|   |   Spongbob
|   |   |   ViewLocator.cs
|   |   |   Models
|   |   |   |   Graph.cs
|   |   |   |   Algorithm.cs
|   |   |   |   DFS.cs
|   |   |   |   Parser.cs
|   |   |   |   Map.cs
|   |   |   |   Result.cs
|   |   |   |   BFS.cs
|   |   |   |   App.axaml
|   |   |   |   Styles.axaml
|   |   |   app.manifest
|   |   |   App.axaml.cs
|   |   |   Spongbob.csproj
|   |   Views
|   |   |   |   MainWindow.axaml
|   |   |   |   ToggleButtonView.axaml.cs
|   |   |   |   MainWindow.axaml.cs
|   |   |   |   TileView.axaml.cs
|   |   |   |   ToggleButtonView.axaml
|   |   |   |   SidebarView.axaml.cs
|   |   |   |   ResultView.axaml.cs
|   |   |   |   ResultView.axaml
|   |   |   |   SidebarView.axaml
|   |   |   |   TileView.axaml
|   |   |   ViewModels
|   |   |   |   ViewModelBase.cs
|   |   |   |   EnumConverter.cs
|   |   |   |   MainWindowViewModel.cs
|   |   |   |   ToggleButtonViewModel.cs
|   |   |   |   ResultViewModel.cs
|   |   |   |   SidebarViewModel.cs
|   |   |   |   TileViewModel.cs
|   |   |   Program.cs

```

4.2.1 Algorithm.cs

Tabel 4.2.1.1 Atribut dan kelas pada file Algorithm.cs

Class Algorithm		
Atribut		
Atribut	Tipe Data	Deskripsi
map	Map	objek yang merepresentasikan peta

started	bool	menandakan apakah algoritma sudah dimulai atau belum
isTSP	bool	menandakan apakah algoritma yang dijalankan adalah TSP atau bukan
treasureCounts	int	jumlah harta karun yang sudah ditemukan
lastTreasure	Graph?	graf terakhir tempat harta karun ditemukan
isTreasureDone	bool	menandakan apakah semua harta karun sudah ditemukan atau belum
isTSPDone	bool	menandakan apakah TSP sudah selesai dijalankan atau belum
nonTSPRoute	List	rute yang dilalui algoritma selain TSP
IsDone	bool	menandakan apakah algoritma sudah selesai dijalankan atau belum
IsBack	bool	menandakan apakah algoritma sedang kembali ke posisi awal
Method		
Method	Parameter/Return Type	Description
Constructor	Map, bool	Membuat objek Algorithm dengan parameter Map dan isTSP, isTSP adalah boolean yang menentukan apakah objek digunakan untuk TSP atau tidak
GenerateResult	Result, string	Mengisi hasil Route dan Tiles pada objek Result dengan parameter string final yang berisi urutan gerakan
SetNonTSPRoute	string	Mengisi List<Tuple<int, int>> nonTSPRoute dengan posisi pada setiap langkah yang diambil pada rute non-TSP
Initialize	void	Mempersiapkan eksekusi algoritma dengan menonaktifkan semua flag visited dan mengosongkan semua queue atau stack dan hanya mengisi dengan posisi awal.
Next	string	Mengembalikan Tuple yang berisi string rute berikutnya dan boolean isDone yang menunjukkan apakah rute sudah selesai
JustRun	Result	Menjalankan algoritma tanpa visualisasi dan mengembalikan objek Result
NextVisualize	string, Graph, Graph, bool	Menjalankan algoritma dengan visualisasi dan mengembalikan Tuple yang berisi string rute, Graph sebelum dan sesudah pemrosesan, dan boolean isDone
RunAndVisualize	Callback, Func<int>, CancellationTokenSource	Menjalankan algoritma dengan visualisasi dan menunjukkan setiap langkah dalam Callback menggunakan CancellationTokenSource untuk pembatalan
GetGraphStep	char, Graph, bool	Mengembalikan Graph tetangga yang sesuai dari Graph input berdasarkan karakter arah dan bool reversed
RefactorRoute	Tuple<string, Graph>, string	Mengembalikan Tuple yang berisi rute yang sudah diperbarui dan Graph sebelumnya untuk rute TSP

4.2.2 Map.cs

Tabel 4.2.2.1 Atribut dan kelas pada file Map.cs

Class Map		
Atribut		
Nama	Tipe	Deskripsi
tiles	Graph?[,]	Matriks dua dimensi yang merepresentasikan petak-petak pada peta
Width	int	Lebar dari peta
Height	int	Tinggi dari peta
StartPos	Tuple<int, int>	Tuple yang merepresentasikan koordinat posisi awal
TreasuresCount	int	Jumlah harta karun pada peta
Method		
Method	Deskripsi	
Map	Constructor class Map, inisialisasi matriks tiles dengan ukuran height x width, kemudian inisialisasi Width dan Height	
SetTile	Mengatur nilai pada petak yang diwakili oleh parameter i dan j dengan Graph baru	
ResetState	Mengubah status setiap Graph pada tiles menjadi NotFound	
CheckValid	Mengecek apakah suatu petak di dalam matriks tiles valid atau tidak	
Print	Menampilkan peta yang direpresentasikan oleh matriks tiles pada konsol	

4.2.3 Graph.cs

Tabel 4.2.3.1 Atribut dan kelas pada file Graph.cs

Class Graph		
Atribut		
Atribut	Tipe data	Deskripsi
Id	int	ID graph
isTreasure	bool	Menunjukkan apakah graph berisi harta karun atau tidak
Pos	Tuple<int, int>	Koordinat posisi graph
TileView	TileView	Status tampilan tile
states	TileState	Status tile
backStates	TileState	Status tile saat kembali

neighbors	Graph?[]	Array yang berisi tetangga graph
Method		
Method	Tipe kembalian	Deskripsi
Graph(int x, int y, bool isTreasure = false)	void	Constructor graph dengan koordinat dan nilai isTreasure opsional
Graph(Graph? top, Graph? right, Graph? bottom, Graph? left, bool isTreasure = false)	void	Constructor graph dengan tetangga dan nilai isTreasure opsional
GetNeighbor	Graph?	Mengembalikan tetangga graph di posisi tertentu
SetNeighbor	void	Menetapkan tetangga graph di posisi tertentu dengan nilai yang diberikan

4.2.4 Parser.cs

Tabel 4.2.4.1 Atribut dan kelas pada file Parser.cs

Class Parser		
Method		
Method	Deskripsi	
ParseFile	Menerima argumen string filename dan mengembalikan objek Map yang merupakan hasil parsing dari file tersebut.	

4.2.5 Result.cs

Tabel 4.2.5.1 Atribut dan kelas pada file Result.cs

Class Result		
Atribut	Tipe Data	Deskripsi
Route	List<char>	Merupakan daftar rute yang ditempuh untuk mencapai solusi. List ini bersifat read-only, artinya hanya bisa diakses untuk membaca dan tidak bisa diubah nilainya.
Tiles	int[,]	Merupakan matriks yang merepresentasikan kondisi papan permainan pada saat solusi ditemukan. Matriks ini berukuran height x width.

NodesCount	int	Merupakan jumlah simpul (node) yang dijelajahi pada saat pencarian solusi. Atribut ini bisa diakses dan diubah nilainya.
Steps	int	Merupakan jumlah langkah yang diambil untuk mencapai solusi. Nilainya dihitung dari panjang list Route.
Time	long	Merupakan waktu yang diperlukan untuk menyelesaikan pencarian solusi, diukur dalam satuan millisecond. Atribut ini bisa diakses dan diubah nilainya.
Found	bool	Merupakan indikator apakah solusi ditemukan atau tidak. Jika solusi ditemukan, nilai atribut ini adalah true, dan sebaliknya. Atribut ini bisa diakses dan diubah nilainya.

4.2.6 BFS.cs

Program BFS.cs merupakan *derived class* dari class algorithm dengan method Next dan NextVisualize yang ada pada kelas Algorithm diimplementasikan menggunakan algoritma *breadth-first Search* (BFS). Kelas BFS sendiri memiliki method “addToQueue” yang berfungsi untuk menambahkan elemen ke dalam antrian (queue) dalam bentuk Tuple<string, Graph> berdasarkan data tetangga yang diberikan. Kelas BFS juga mempunyai atribut 3 buah queue bernama graphprio1, graphprio2, dan stucks. Implementasi algoritma *Breadth First Search* tersebut tertera pada bagian 4.1.2.

4.2.7 DFS.cs

Program DFS.cs merupakan *derived class* dari class algorithm dengan method Next dan NextVisualize yang ada pada class Algorithm diimplementasikan menggunakan algoritma *depth-first search* (DFS). Kelas DFS sendiri memiliki method “addToStack” yang berfungsi untuk menambahkan elemen ke dalam stack dalam bentuk Tuple<string, Graph> berdasarkan data tetangga yang diberikan. Kelas DFS juga mempunyai atribut 3 buah stack bernama graphprio1, graphprio2, dan stucks. Implementasi algoritma *Depth First Search* tersebut tertera pada bagian 4.1.3.

4.3 Pengujian Program

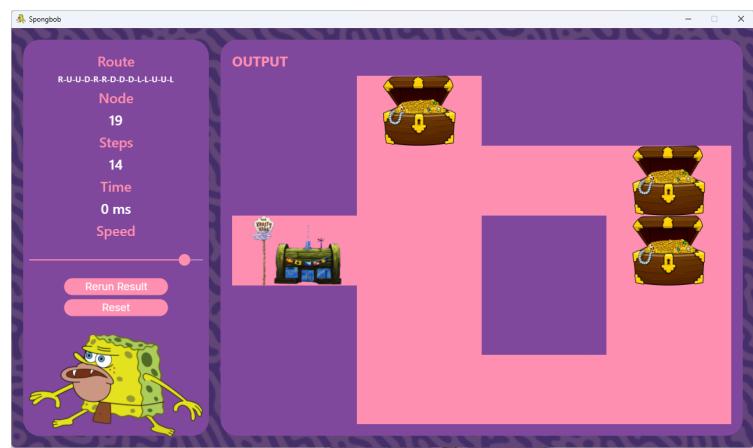
Berikut adalah beberapa hasil pengujian pada program penyelesaian maze menggunakan *test case* yang diberikan:

4.3.1 sampel-1.txt

Tabel 4.3.1.1 Hasil pengujian program pada file sampel-1.txt

BFS	
DFS	

BFS-TSP



DFS-TSP



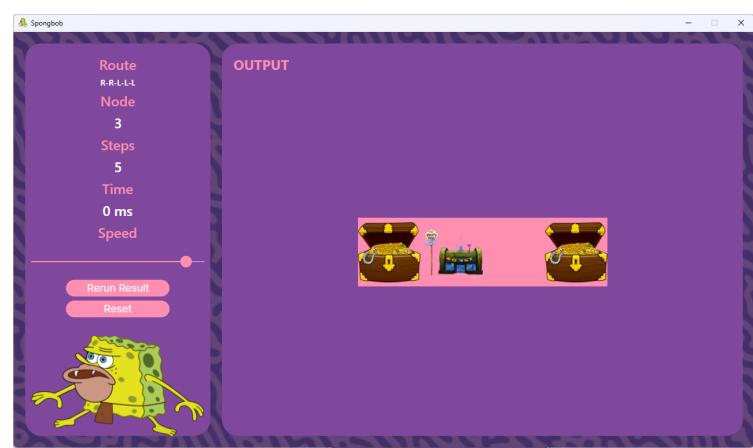
4.3.2 sampel-2.txt

Tabel 4.3.2.1 Hasil pengujian program pada file sampel-2.txt

BFS



DFS



BFS-TSP

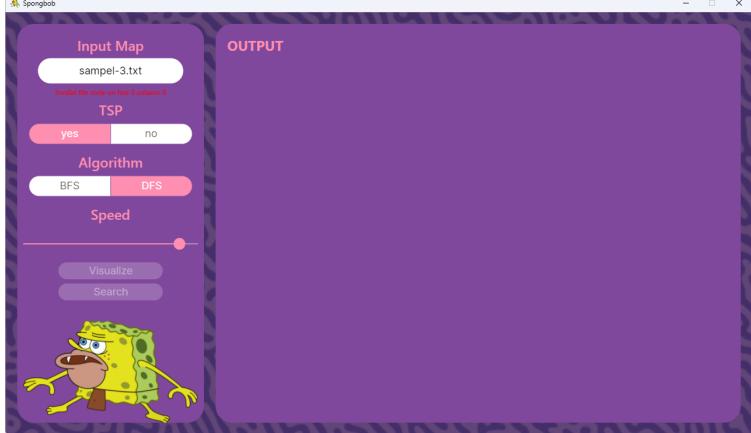
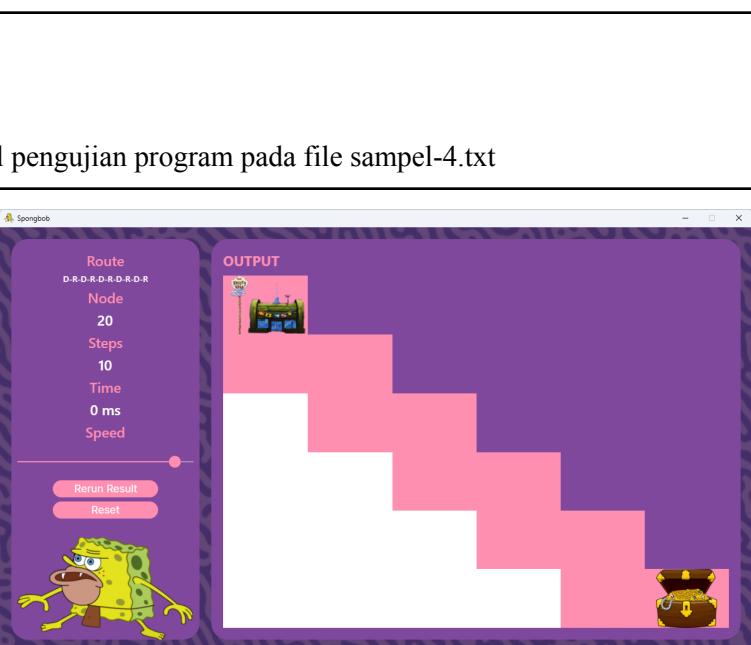
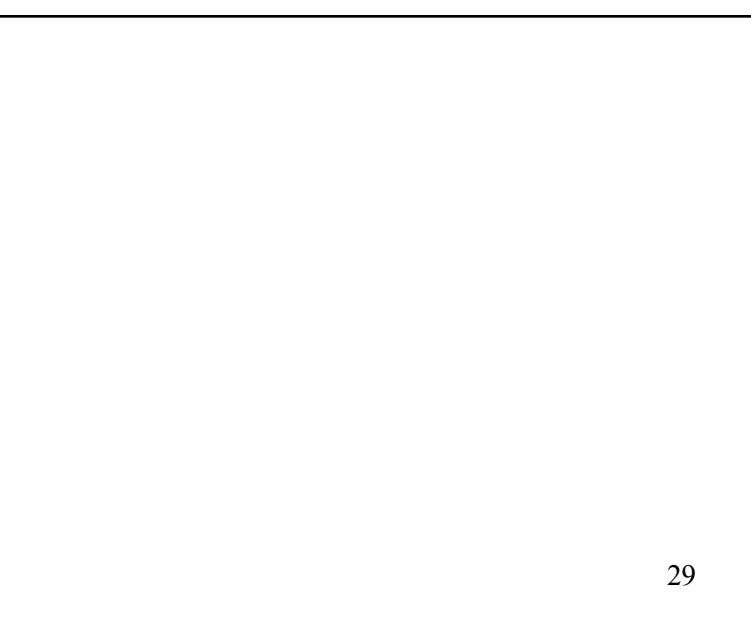


DFS-TSP



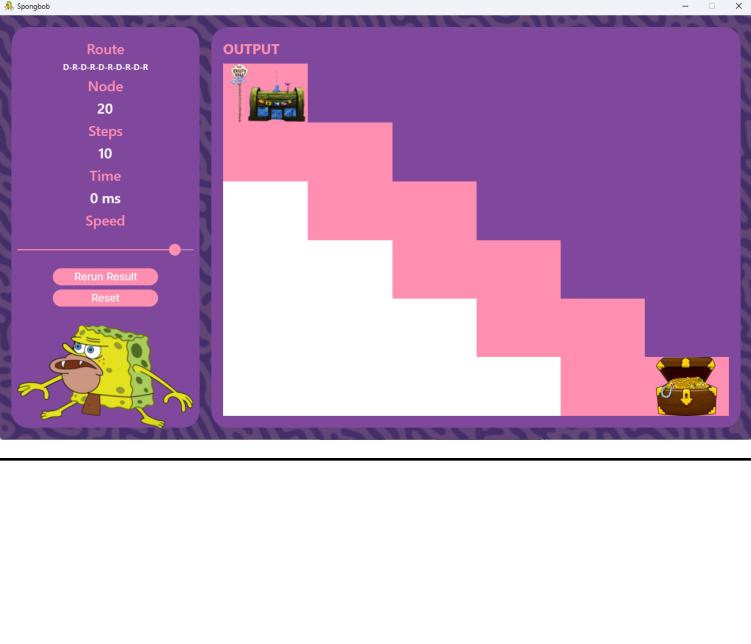
4.3.3 sampel-3.txt

Tabel 4.3.3.1 Hasil pengujian program pada file sampel-3.txt

BFS	
DFS	
BFS-TSP	
DFS-TSP	

4.3.4 sampel-4.txt

Tabel 4.3.4.1 Hasil pengujian program pada file sampel-4.txt

BFS	
-----	---

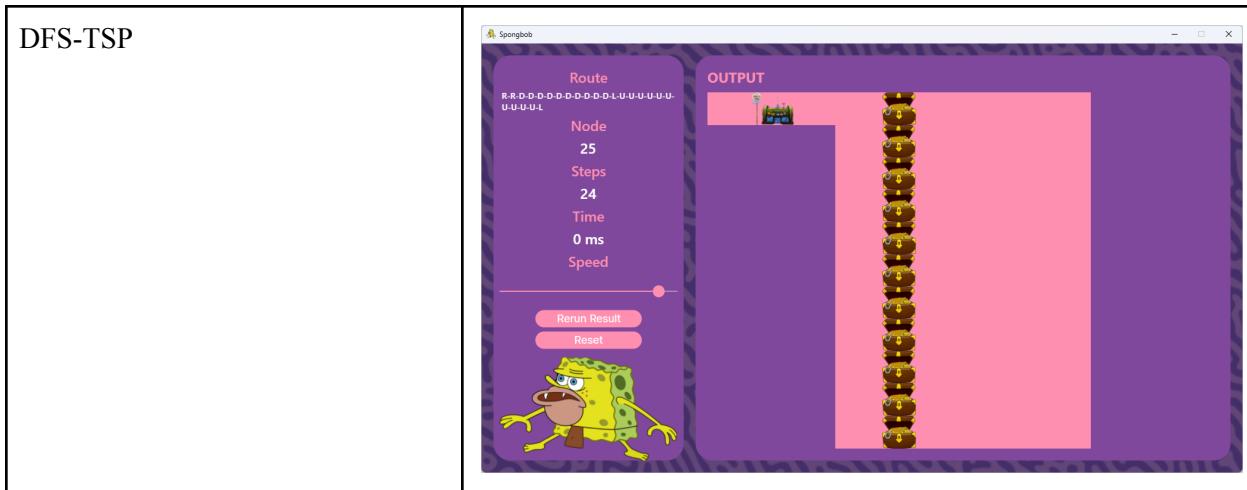
The figure displays three screenshots of a Traveling Salesman Problem (TSP) application interface, each showing a different search algorithm's results:

- DFS (Top Left):** Shows a route of length 10 steps. The route starts at the top-left corner, moves right, then down, then right again, then up, then right, then down, then right, then up, and finally right to complete the loop. The total time is 0 ms.
- BFS-TSP (Middle Left):** Shows a route of length 41 steps. This route is significantly longer than the DFS route, indicating a less efficient search. The total time is 0 ms.
- DFS-TSP (Bottom Left):** Shows a route of length 24 steps. This route is shorter than the BFS-TSP route but longer than the DFS route. The total time is 0 ms.

The interface includes a sidebar with a cartoon character (SpongeBob SquarePants), a "Route" summary table, and buttons for "Rerun Result" and "Reset". The "OUTPUT" section shows a map with colored regions (pink and purple) and icons representing the start point, destination, and treasure chest.

4.3.5 sampel-5.txt

Tabel 4.3.5.1 Hasil pengujian program pada file sampel-5.txt



4.4 Analisis Desain

Dari hasil pengujian yang telah dilakukan, algoritma DFS dan BFS telah menjalankan tujuannya dengan baik. Namun dari segi desain penjelajahan terdapat beberapa kendala inkonsistensi yang disebabkan oleh kurang spesifiknya penjelasan penerapan algoritma yang diberikan pada spesifikasi. Desain yang diimplementasikan pada program ini tidak memperbolehkan pencari petak untuk melakukan perpindahan yang jauh atau melakukan teleportasi ke petak yang tidak bersebelahan. Hal ini tentunya menambah kompleksitas serta memperlambat keberjalanannya dari algoritma. Selain itu desain untuk memprioritaskan jalur yang belum pernah dilalui sebelumnya juga masih terbatas dalam menyelesaikan peta tertentu seperti sampel-4.txt. Disamping itu, GUI sudah berjalan sesuai yang diharapkan tetapi masih dapat dikembangkan untuk menyelesaikan peta yang berukuran sangat besar. Secara keseluruhan, program sudah menjalani semua spesifikasi wajib dan tambahan dengan cukup baik namun masih dapat dikembangkan dalam beberapa aspek yang telah disebutkan sebelumnya.

BAB 5

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pada Tugas Besar II IF2211 Strategi Algoritma ini telah diimplementasikan algoritma BFS dan DFS beserta fungsi - fungsi dan kelas - kelas pendukung dalam tujuan untuk menyelesaikan masalah pencarian *treasure* di sebuah peta. Dari eksperimen yang telah dilakukan, didapatkan bahwa kedua algoritma tersebut dapat digunakan untuk menyelesaikan masalah tersebut.

Dari penggerjaan tugas besar ini, kami mendapatkan beberapa kesimpulan:

1. Pencarian BFS dengan backtracking bukanlah hal yang baik untuk diimplementasikan. BFS lebih baik dilakukan dengan teleportasi. Dalam tugas ini dapat terlihat bahwa visualisasi DFS akan dilakukan dengan jauh lebih cepat dibanding visualisasi BFS.
2. Kecepatan pencarian menggunakan BFS ataupun DFS bergantung terhadap tempat mulai serta peletakan harta karun pada peta dengan tidak ada yang selalu lebih cepat maupun lambat. Hal ini dapat dilihat dengan menggunakan tombol search program tanpa melakukan visualisasi untuk beberapa peta berbeda.

5.2 Saran

Berdasarkan pengalaman penulis mengerjakan tugas ini, berikut merupakan saran untuk pembaca yang ingin melakukan atau mengerjakan hal yang serupa:

1. Backtracking pada algoritma BFS dan DFS tidak perlu dimasukkan ke dalam algoritma dan dapat disederhanakan dengan teleportasi jika ingin memaksimalkan performa. Backtracking pada program ini hanya dilakukan untuk tujuan visualisasi dan tidak diperlukan jika hanya mencari hasilnya saja.
2. Perancangan algoritma dan struktur program perlu diperhatikan dalam pembuatan program. Struktur data yang baik akan mengubah keberjalanan penggerjaan program dan dapat mempermudah maupun mempersulit implementasi algoritma ke depannya.
3. Kerja sama antar tim merupakan hal yang penting dalam penggerjaan tugas ini, khususnya untuk menghubungkan algoritma terhadap antarmuka yang disajikan kepada pengguna. Penulis merekomendasikan penggunaan aplikasi *version control system* seperti github untuk mengelola pekerjaan secara asinkron.

5.3 Refleksi

Tugas Besar 2 IF2211 Strategi Algoritma merupakan salah satu tugas besar menarik yang penulis dapatkan pada semester ini. Proses penggerjaan tugas ini tentunya melalui berbagai rintangan. Dengan tugas ini, penulis mendapatkan berbagai pengetahuan mengenai penggunaan algoritma BFS dan DFS, yaitu untuk mencari rute dalam suatu peta pencarian. Penulis diberi kesempatan untuk mengimplementasikan hal tersebut dalam sebuah aplikasi. Keberhasilan akan mengimplementasikan hal tersebut memberi rasa pencapaian dengan keberhasilan membuat hal baru. Dalam tugas besar ini penulis juga mendapatkan kesempatan untuk melakukan eksplorasi lebih dalam terhadap bahasa C# beserta sisi *user interface framework* yang tersedia dalam bahasa tersebut yaitu Avalonia UI.

Tantangan utama yang penulis hadapi adalah optimisasi dan *debugging* untuk masalah - masalah khusus yang tidak langsung terlihat. Rintangan ini dihadapi dengan melihat program dengan lebih teliti serta mencoba program dengan kasus yang lebih sulit. Eksplorasi dalam penggunaan bahasa C# serta implementasi algoritma BFS dan DFS juga membantu penulis untuk menghadapi rintangan tersebut.

5.4 Tanggapan

Tanggapan penulis mengenai tugas besar ini adalah bahwa tugas ini cukup menarik untuk dilaksanakan karena menggunakan hal baru seperti bahasa C#. Akan tetapi, penulis merasa spesifikasi yang diberikan kurang jelas dalam beberapa kasus dan implementasi dari algoritma, contohnya untuk algoritma BFS pada contoh gambar spesifikasi perlu penjelasan lebih lanjut mengapa bisa didapatkan hasil seperti itu. Dalam beberapa kali pertanyaan pada QnA maupun pribadi pernah didapatkan hasil yang tidak konsisten seperti teleportasi dan *backtracking*.

DAFTAR PUSTAKA

Munir, R, and Maulidevi, N. U.. (2022). Breadth/Depth First Search (BFS/DFS) (1).

Retrieved from Homepage Rinaldi Munir: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>

Munir, R, and Maulidevi, N. U.. (2022). Breadth/Depth First Search (BFS/DFS) (2).

Retrieved from Homepage Rinaldi Munir: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>

LAMPIRAN

[Tautan repositori Github](#)

[Link video singkat](#)